Contributions à la conception sûre des systèmes embarqués sûrs

Alain Girault

▶ **To cite this version:**

Alain Girault. Contributions à la conception sûre des systèmes embarqués sûrs. Réseaux et télécommunications [cs.NI]. Institut National Polytechnique de Grenoble - INPG, 2006. tel-00177048

HAL Id: tel-00177048
https://theses.hal.science/tel-00177048

Submitted on 5 Oct 2007

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

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

HABILITATION À DIRIGER DES RECHERCHES

présentée par

Alain GIRAULT

pour obtenir le diplôme d'HABILITATION À DIRIGER LES RECHERCHES
de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(Spécialité : Informatique)

Contributions à la conception sûre des systèmes embarqués sûrs.

Date de soutenance : 5 septembre 2006

Composition du jury :	Président	Denis Trystram
	Rapporteurs	David Powell
		Alberto Sangiovanni-Vincentelli
		Jean-Bernard Stéfani
	Examinateurs	Albert Benveniste
		Étienne Closse
		André Schiper

Travaux de recherche effectués successivement au sein des équipes MEIJE (INRIA Sophia-Antipolis / Centre de Mathématiques Appliquées de l'École des Mines de Paris), PTOLEMY et PATH (département EECS de l'Université de Californie à Berkeley), et enfin BIP et POP ART (INRIA Rhône-Alpes).

Table des matières

Remerciements

Ce document est la synthèse de mes activités de recherche depuis la fin de ma thèse (janvier 1994) jusqu'à maintenant. Cela représente donc douze années de recherche. Je tiens à remercier tous les collègues qui m'ont accueilli, aidé et accompagné pendant ces douze années. En premier lieu Paul Caspi qui m'a beaucoup appris et qui a constamment éclairé mes réflexions par ses conseils avisés et ses profondes connaissances scientifiques ; Paul est, pour moi, l'idéal du chercheur et le chemin est encore très long avant d'atteindre un tel niveau de compréhension des problèmes, tant en informatique qu'en automatique. Gérard Berry m'a fait l'honneur de m'accueillir à Sophia-Antipolis dans l'équipe MEIJE pour travailler sur le langage ESTEREL. Après avoir fait ma thèse sur LUSTRE, ce fut très instructif de travailler sur ESTEREL et je remercie Gérard pour tout ce qu'il m'a appris, des circuits aux nombres 2-adiques. J'ai eu beaucoup de chance qu'Edward Lee me prenne en postdoc à Berkeley et j'ai beaucoup appris aussi avec lui, en particulier sur le modèle PTOLEMY. Je lui suis très reconnaissant de cela et de l'excellente année passée dans la Bay Area. Pravin Varaiya m'a offert la possibilité de prolonger mon séjour à Berkeley dans son laboratoire PATH sur les autoroutes automatisées. Les travaux que j'ai pu y démarrer sont à l'origine du chapitre 4 de mon habilitation. De retour en France, je dois un remerciement tout particulier à Bernard Espiau pour m'avoir laissé une entière liberté à mon arrivée à l'INRIA ; grâce à cette liberté, j'ai pu explorer sereinement deux nouvelles pistes de recherche, d'une part la commande continue des véhicules autonomes pour les autoroutes automatisées, et d'autre part la tolérance aux fautes, sujet que j'ai abordé avec l'insouciance de la jeunesse (j'ai même coordonné et animé une ARC), en n'y connaissant presque rien au départ (j'ai beaucoup appris depuis, fort heureusement), et sur lequel j'ai réussi à obtenir des résultats intéressants (du moins je crois) ; ceci constitue, à mon avis, un bel exemple de la nécessité de laisser aux jeunes chercheurs une grande liberté de choix sur leurs sujets de recherche. Enfin, mon dernier « chef » fut Éric Rutten, que je remercie pour m'avoir fait découvrir la synthèse de contrôleurs discrets et m'avoir poussé à écrire mon rapport d'habilitation.

Je remercie également tout particulièrement les membres de mon jury d'habilitation : mes trois rapporteurs David Powell, Alberto Sangiovanni-Vincentelli et Jean-Bernard Stéfani, mes trois examinateurs Albert Benveniste, André Schiper et Étienne Closse, et enfin mon président de jury Denis Trystram. Parmi eux, je dois des remerciements très appuyés à David Powell pour les nombreuses corrections qu'il m'a fait faire sur le chapitre 3. Certes, sur le moment cela ne m'a pas fait plaisir du tout de recevoir 15 pages de corrections (!), mais la qualité de mon document s'est grandement amélioré et je l'en remercie beaucoup. Suite à une grève non annoncée à l'aéroport Lyon Saint-Exupéry, ma soutenance a du être reportée ; aussi, Albert, David et André ont dû se lever aux aurores deux fois, et Alberto quant à lui s'est levé pour rien puisqu'il n'a pas pu venir à la seconde soutenance (la vraie) ☹. J'en suis sincèrement désolé.

Je continue mes remerciements avec mes collègues, et néanmoins amis, Thao Dang, Jean-Claude Fernandez, Fabienne Lagnier, Pascal Raymond, Nicolas Halbwachs, Florence Maraninchi, Xavier Nicollin, Sergio Yovine, Anne Rasse, Susane Graf, Laurent Mounier, Yassine Lacknech, Oded Maler, Stavros Tripakis et bien sûr Joseph Sifakis (les sbires de VERIMAG). Puis Xavier Fornari, Amar Bouali, Jean-Paul Marmorat, Illaria Castellani, Davide Sangiorgi, Gérard Boudol, Valérie Roy, Frédéric Boussinot et Robert de Simone (la salade niçoise), Christopher Highlands, John Reeckie, José Pino, Stephen

Edwards, Wan-Teh Chang, Bilung Lee, Akash Deshpande, Aleks Göllü, Marco Antoniotti et Daniel Weisman (les cowboys de Berkeley), Daniel Simon, Gregor Gößler, Pascal Fradet, Hamoudi Kalla, Gwenaël Delaval Tolga Ayav, Massimo Tivoli et Simplice Djoko-Djoko (les pop artistes), Pierre-Brice Wieber, Sophie Chareyron, Bernard Brogliato, Claude Lemaréchal, Vincent Acary et Jérôme Malik (les bipopiens), Florence Forbes, Gilles Celeux, Cécile Delhumeau, Paulo Goncalves, Henri Bertholon et Stéphane Girard (les stateux du couloir D), Jacques Mossière, Daniel Hagimont, Vivien Quéma, Alan Schmitt, Ali-Erdem Ozcan, Oussama Layaida, Hubert Garavel, Frédéric Lang et Wendelin Serwe (les éminents collègues du couloir B), Muriel Jourdan, Nabil Layaida, Vincent Quint, Irène Vaton, Claude Castellucia, Vincent Roca et Jérôme Euzenat (idem mais à l'étage supérieur), Horia Toma, Radu Mateescu, Mihaela Sighireanu, Cătălin Dima, Radu Horaud et Emil Dumitrescu (le gang roumain), Roger Pissard-Gibollet, Hervé Mathieu, Gérard Baille, Soraya Arias et Nicolas Turro (les roboticiens parfois marcheurs, parfois roulants), Brigitte Plateau, Denis Trystram, Bruno Gaujal, Jean-Louis Roch, Jean-Marc Vincent, Grégory Mounié et Lionel Eyraud (les Apaches), Marc Pouzet, Louis Mandel, Grégoire Hamon et Thérèse Hardin (les parigots du LIP6), Jacques Pulou, Étienne Closse, Daniel Weil et Kathleen Milsted (les mousquetaires de France Telecom R&D), Daniel Pilaud, Patrick Munier et Cyrille Prévé (les rois des polyèdres de chez POLYSPACE), Philippe Audfray, Isabelle Guigues et Franck Combet-Blanc (les mélangeurs d'IGRIP et d'ESTEREL de chez ATHYS, maintenant DASSAULT SYSTÈME).

Je n'oublie pas mes étudiants en thèse (Laure France, Christophe Legal, Hamoudi Kalla et Gwenaël Delaval), ils m'ont beaucoup appris, ni mes étudiants de DEA ou de stage ingénieur (trop nombreux pour les citer tous).

J'accorde une très grande importance aux fonctions de support à la recherche. J'ai conscience à l'INRIA d'être un privilégié puisque nous disposons d'une administration à la fois compétente et efficace (ce qui ne veut pas pour autant dire que nous en soyons toujours satisfaits). Certes, parfois je m'emporte parce que j'ai l'impression que cette administration devient trop « pléthorique », parfois j'ai le sentiment qu'ils oublient que le 'R' de INRIA veut dire « Recherche », et que, en fin de compte, l'*unique* raison d'être de l'INRIA est de faire de la recherche. Mais d'un autre côté, quand je demande des billets de train en urgence, ils arrivent sur mon bureau presque toujours à temps, quand je cherche un article de 1969, on m'en fournit une photocopie ou un pdf en un temps record, et surtout le réseau informatique marche tous les jours, les imprimantes et le courrier électronique itou, et mon compte informatique est sauvegardé toutes les nuits. Pour tout cela, je remercie chaleureusement Françoise de Conninck, Véronique Roux, Élodie Toihein, Marie Wiernsperger et Stéphanie Berger (assistantes de projet), Isabelle Allegret, Sabine Maury, Bruno Marmol, Didier Chassignol, Éric Ragusi, Frédéric Saint-Marcel, Jean-Marc Joseph, Jean-Pierre Augé, Ludovic Bellier et Guillaume Lelaurain (administrateurs système et réseau), Cécile Bellini (service des missions), Karim Dekiouk, Nathalie Sce, Cédric di Tofano et Clare Buzon (service financier et juridique), Isabelle Rey et Chantal Baudin (documentation), Danièle Herzog, Babette Beaujard, Marie Collin et Marc Barret (valorisation), et enfin Ben Bouksbara, Yves Bard, Eric Anglès et Jean Panzuti (services généraux). Que celles et ceux que j'ai oublié me pardonnent.

Mes parents Claude et Vivette m'ont montré la voie puisqu'ils sont tous deux chercheurs. Claude a relu minutieusement mon rapport d'habilitation et je l'en remercie. J'ai beaucoup entendu parler de l'INRIA quand j'étais jeune. Je mesure pleinement aujourd'hui ma chance d'y travailler.

Pour terminer, ma femme Isabelle m'apporte beaucoup de bonheur et supporte que je rentre tard pour finir de rédiger des articles (ou une habilitation à diriger les recherches ◌ˇ|ˇ◌) et je l'adore. Également, mes deux fistons Bastien et Mathis mettent de la joie et de l'animation dans la maison et je les adore eux-aussi.

J'aurais aimé inviter à mon jury Gilles Kahn, malheureusement il nous a quitté au début de l'année 2006. C'était un très grand chercheur et je lui dédie cette habilitation.

Chapitre 1

Introduction

Ce document est la synthèse de mes activités de recherche depuis la fin de ma thèse (janvier 1994) à maintenant. Il comporte trois parties qui correspondent à trois étapes (entrelacées temporellement) de ma carrière de chercheur. Ces trois parties ont en commun de porter sur le thème général de la conception sûre des systèmes embarqués sûrs. J'ai écrit au début de chacune de ces partie une introduction spécifique, aussi le présent chapitre n'introduit-il le sujet que de façon très générale.

Les systèmes informatiques embarqués sont aujourd'hui omniprésents. Dans la vie de tous les jours, on en trouve dans les équipements électroménagers aussi bien que multimédia, dans les téléphones portables aussi bien que dans voitures. Certains sont à sûreté critique, comme les avions, d'autres moins, comme les montres ou des machines à laver. Avec la miniaturisation des composants, les nouvelles technologies de batteries, et le développement des moyens de communication, il faut s'attendre à en avoir de plus en plus autour de nous, à tel point que certains prédisent l'avènement prochain de l'informatique ubiquitaire. Mais au fait, qu'est-ce qu'un système informatique embarqué ? La définition qui semble la plus communément admise stipule que ce sont des systèmes informatiques dont les ressources sont *limitées*. Par ressource, on entend ressources de taille, de poids, de consommation électrique et de puissance de calcul. Certes, tout ordinateur de bureau a une consommation électrique limitée par la puissance de l'installation électrique de la prise sur laquelle il est branché. Il en va de même pour toutes les limites mentionnées ci-dessus puisque le monde est fini. Mais, dans le cas des ordinateurs de bureau, cette limite n'est en pratique pas prise en compte par les fabriquants. En revanche, pour un PDA, un téléphone portable ou un satellite, elle est critique.

Je m'intéresse dans ce document à la conception sûre de systèmes embarqués sûrs. Mon objectif est donc double : d'une part fournir des méthodes de conception de systèmes embarqués qui soient sûres, c'est-à-dire telles que le fossé entre l'idée du programmeur et le programme soit le plus petit possible et le moins possible source de faute de conception, et aussi telles que le programme objet obtenu soit conforme à la spécification ; et d'autre part fournir des méthodes de conception qui améliorent la sûreté de fonctionnement des systèmes produits.

La première partie (chapitre 2) concerne la répartition automatique de programmes synchrones. Le caractère automatique de la répartition apporte un réel degré de sûreté dans la conception de systèmes répartis car c'est la partie la plus délicate de la spécification qui est automatisée. Grâce à cela, l'absence d'inter-blocage et l'équivalence fonctionnelle entre le programme source centralisé et le programme final réparti peuvent être formellement démontrées. Quiconque ayant déjà eu à programmer un système réparti sait bien que ces deux points sont particulièrement difficiles à obtenir. Ces travaux ont démarré en 1990 pendant ma thèse sous la direction de Paul Caspi et Jean-Louis Bergerand. Les plus récents développements portent sur la prise en compte des rythmes du système, sur l'ordre supérieur et sur la mobilité de code. Pour ces recherches, j'ai été dans un premier temps encadré (en thèse puis en postdoc avec Gérard Berry et Edward Lee, jusqu'en 1996), puis autonome (après 1996). Dans ce chapitre, je ne relate que mes activités postérieures à ma thèse (donc après 1994).

La deuxième partie (chapitre 3) traite le sujet de l'ordonnancement et la répartition de graphes de tâches flots-de-données sur des architectures à mémoire répartie, avec contraintes de tolérance aux fautes et de fiabilité. Ces travaux ont débuté fin 1998 avec l'Action de Recherche Coordonnée INRIA TOLÈRE, que j'ai animée. Ce chapitre décrit donc huit années de recherches, principalement sur des heuristiques de répartition et d'ordonnancement avec pour but la tolérance aux fautes et la fiabilité des systèmes, mais également sur l'utilisation de méthodes formelles telles que la synthèse de contrôleurs discrets ou la programmation orientée aspects. Sur ce sujet, j'ai co-encadré quatre postdocs (Mihaela Sighireanu, Cătălin Dima, Emil Dumitrescu et Tolga Ayav), un étudiant en thèse (Hamoudi Kalla), et de nombreux stagiaires ingénieurs ou DEA.

Enfin, la troisième partie (chapitre 4) concerne les autoroutes automatiques, avec deux volets : la commande longitudinale de véhicules autonomes et les stratégies d'insertion dans les autoroutes. Ces travaux ont démarré en 1997 lors de mon postdoc à l'Université de Berkeley dans le groupe PATH de Pravin Varaiya. Cette dernière partie décrit donc neuf années de recherches (intermittentes) sur des lois d'accélération et les autoroutes automatisées, domaine appartenant à l'automatique des systèmes continus et au départ étranger à ma culture. Sur ce thème j'ai été complètement autonome et j'ai encadré deux étudiants, Jean-Philippe Roussel (stage ingénieur) et Fethi Bouziani (DEA).

En revanche, ce document ne décrit pas mes activités sur la simulation graphique de robots bipèdes et l'interpolation de mouvement entre positions-clés (« keyframing » en anglais), activité conduite de 1996 à 1999 quand j'ai co-encadré la thèse de Laure France. J'ai en effet estimé que cette activité de recherche était trop éloignée des systèmes embarqués. Ce document ne décrit pas non plus mes activités sur les algorithmes locaux de recherche de chemins disjoints dans les réseaux, parce que ces recherches ne sont pas encore assez abouties.

Enfin, ce document ne décrit pas non plus mes activités d'enseignement à l'ENSIMAG, au Département Télécom de l'INPG et à l'UFR IMA de l'Université Joseph Fourier. De 1998 à 2006, j'ai encadré le projet « Génie Logiciel et Compilation » et j'ai effectué des Cours et Travaux Dirigés de Compilation, de Théorie des Automates, de Programmation Temps-Réel et Réactive, d'Algorithmique en JAVA, et de Programmation Synchrone.

Chapitre 2

Répartition automatique de programmes synchrones

2.1 Introduction

2.1.1 Les systèmes réactifs embarqués

Le logiciel embarqué prend une part croissante dans la gestion de nombreux systèmes en contrôlant de plus en plus leur fonctionnement de façon automatique. Présents depuis longtemps dans des applications coûteuses (spatiales, aéronautiques, militaires...) où ils prennent une importance considérable, ces systèmes apparaissent dans des domaines grand public, tels que les appareils électroniques (téléphones portables, PDA, appareils photos...), mais aussi l'assistance à la conduite automobile (direction assistée, freinage assisté, en attendant les systèmes dits « drive-by-wire »). Les caractéristiques et contraintes principales de ces systèmes sont :

Dualité contrôle/commande : Ces systèmes présentent d'une part un aspect commande, lois modélisées par des équations différentielles en temps échantillonné, et d'autre part un aspect contrôle de ces lois de commande, représenté par des systèmes à événements discrets.

Temps réel : Ces systèmes sont soumis à des contraintes temporelles strictes, qui concernent à la fois le rythme des entrées et la latence entre les entrées et les sorties.

Sûreté critique : Une défaillance du système peut conduire à une catastrophe humaine, écologique ou financière. J'y consacre entièrement le chapitre 3.

Ressources limitées : Ces logiciels sont embarqués et disposent donc d'une puissance de calcul ainsi que d'une mémoire limitées, que ce soit pour des raisons de poids, de volume, de consommation énergétique (véhicules autonomes), de durcissement aux radiations (nucléaire, spatial), ou de prix (applications grand public).

Architecture répartie et hétérogène : Ces applications sont le plus souvent réparties, d'une part pour fournir une puissance de calcul importante et d'autre part, pour rapprocher les capteurs et actionneurs de cette puissance de calcul.

2.1.2 La programmation synchrone

Le modèle synchrone a précisément été proposé afin de faciliter la programmation de tels systèmes. Que dire des langages synchrones ? Ce domaine a débuté dans les années 80, aussi on trouvera dans la littérature de nombreux articles sur le sujet [16, 13], ainsi qu'un livre [64]. Une définition que j'aime est que ce sont des **langages temporels de haut niveau** munis une **sémantique formelle** :

- Ici, temporel de haut niveau signifie qu'ils offrent des primitives temporelles de haut niveau, telles que la préemption, la suspension, la diffusion... Comme exemple de formalisme temporel de bas niveau, on peut citer le langage C couplé à un système d'exploitation temps-réel (VXWORKS, QNX, RT LINUX...), ou même ADA.
- Ensuite ils sont munis d'une sémantique formelle, par opposition à ADA ou UML par exemple. Cela présente plusieurs avantages :
  - ils peuvent servir de langages de spécification formelle : c'est le cas notamment de SCADE chez EADS ;
  - tout programme est transformable en un modèle formel (par exemple un automate d'états fini), sur lequel on peut faire des analyses, (par exemple de la vérification formelle), des transformations (par exemple de la répartition), et enfin de la génération de code.

Ces deux concepts fondent un certain nombre de langages académiques, appelés langages synchrones, que sont ESTEREL [18], SL [21], SIGNAL [79], LUSTRE [65], le domaine SR de PTOLEMY [46], LUCID Synchrone [38], ARGOS [83], SYNCCHARTS [3], les Automates de Modes [83] et AUTOFOCUS [71], ainsi que plusieurs langages et outils industriels que sont SIMULINK, STATEFLOW, STATECHARTS, LCM STUDIO, SCADE (version industrielle de LUSTRE), ESTEREL STUDIO (version industrielle d'ESTEREL) et SILDEX (version industrielle de SIGNAL). Tous ont en commun de proposer des primitives temporelles *idéales* : par exemple, la préemption peut être *instantanée*, c'est-à-dire que quand un événement donné se produit localement dans un module, ce dernier peut interrompre un autre module en parallèle avec lui-même, et cette interruption a lieu à l'instant précis où l'événement se produit. Ceci facilite grandement les raisonnements temporels, et c'est en ce sens qu'on dit que ces langages sont temporels de haut niveau. Par exemple, il est possible d'écrire `every 60 seconde emit minute`, avec la signification que `minute` est émis *exactement en même temps* que la 60$^{\text{e}}$ `seconde`. On voit ici à quel point de telles primitives temporelles peuvent faciliter la tâche du programmeur, dans la mesure où elles ont un comportement intuitif. L'avantage est que cela réduit considérablement les *bugs temporels*. Or ceux-ci sont en général très difficiles à trouver et à corriger ; il suffit pour s'en convaincre de se reporter au Mars Rover Pathfinder [72] : ce cas est particulièrement édifiant et mérite qu'on s'y attarde. Le bug impliquait plusieurs tâches du système, le réseau de communication et l'ordonnanceur. C'est ce dernier qui avait provoqué une inversion de priorité qui, privant d'accès au réseau de communication une tâche prioritaire, entraînait une ré-initialisation du système complet ! Le bug fut particulièrement difficile à trouver pour au moins trois raisons : premièrement il impliquait plusieurs tâches, le réseau et l'ordonnanceur, d'où une complexité significative ; deuxièmement, la tâche qui entraînait la ré-initialisation n'était pas directement en cause et certainement pas fautive dans son comportement fonctionnel ; troisièmement, et c'est peut-être le pire problème qui s'est posé aux ingénieurs, le système dans son ensemble était *indéterministe*, donc le bug n'était même pas reproductible. Il me semble que, sur ces points au moins, l'approche synchrone apporte des réponses satisfaisantes.

Dans la discussion ci-dessus, on voit apparaître la notion d'**instantanéité**, qui est centrale dans les langages synchrones. En effet, si les primitives temporelles sont idéales, c'est justement parce qu'elles sont construites à partir de structures de contrôle qui sont instantanées. Ainsi, toutes les actions qui sont faites en parallèles partagent la **même échelle de temps** qui est de plus **discrète**. Elles peuvent donc être datées sur cette échelle, ce qui supprime l'indéterminisme issu de l'entrelacement. On rencontre cet indéterminisme dans tous les formalismes dérivés des systèmes d'exploitation, en particulier dans les « Communicating Sequential Processes » de Hoare [70]. Par exemple, considérons les deux actions a et b en parallèle : dans CSP, la construction $a \| b$ donne ou bien $a.b$ ou bien $b.a$, le choix entre les deux étant indéterministe, alors que dans un langage synchrone, cela donne le groupe ab, où les actions a et b sont simultanées. C'est cela qu'on appelle l'**abstraction synchrone**.

De nombreux articles du domaine expriment formellement cette abstraction sous le nom d'**hypothèse de synchronisme**, qui stipule que le programme est infiniment rapide, et donc que chacune de ses réactions est instantanée et atomique. Ces réactions divisent donc le temps en une séquence

d'instants discrets, et les sorties émises par le programme en réaction à des entrées sont simultanées avec ces dites entrées [17].

Bien sûr, le code que doit générer le compilateur reste séquentiel lui, selon le bon vieux modèle de Von Neumann. Par conséquent, tout compilateur synchrone doit résoudre les dépendances de contrôle entre les diverses actions en parallèle, afin de les séquencialiser. Cette étape est appelée **analyse de causalité** ; elle nécessite de connaître le comportement individuel de toutes les actions en parallèle. Je montrerai dans la section 2.2.1 ce que cela aura comme conséquences.

Enfin, une fois le code final généré, il est nécessaire de valider l'hypothèse de synchronisme. Celle-ci suppose en effet que la machine utilisée pour exécuter le programme soit infiniment rapide, ce qui est bien sur impossible. En pratique, il suffit de vérifier que le **temps de réaction** du programme est suffisamment petit pour que les contraintes temporelles soient satisfaites. Une condition suffisante est que le temps de réaction soit inférieur à la plus petite fréquence des entrées, car cela assure qu'aucun événement d'entrée ne sera perdu. La calcul de ce temps de réaction est facilité par le fait que le code généré par le compilateur est séquentiel. C'est la validation de l'hypothèse de synchronisme qui garantit, a posteriori, la légitimité du modèle de programmation synchrone.

Le temps de réaction est aussi appelé « Worst Case Execution Time », (WCET). Le problème de l'estimation du WCET d'un programme impératif déterministe est très complexe en lui-même, en raison des fonctionnalités dont sont équipés les processeurs modernes, qui les rendent intrinsèquement indéterministes : cache multi-niveau, pipe-line... Aussi, la technique la plus utilisée aujourd'hui consiste à exécuter le programme de multiples fois sur un processeur donné et à prendre le maximum des temps d'exécution *mesurés*. L'alternative consiste à calculer le nombre de cycles machines nécessaires à son exécution, ce qui est complexe en raison des caractéristiques que je viens de mentionner.

2.1.3 La compilation des programmes synchrones

Comme je l'ai dit précédemment, un compilateur synchrone doit réaliser l'analyse de causalité du programme source et produire du code séquentiel. Plus que la méthode de compilation, c'est le code généré qui m'intéresse ici. Il en existe de plusieurs types, qui diffèrent selon la nature de la structure de contrôle. Dans tout ce qui suit, je mentionne explicitement les formats intermédiaires produits par les compilateurs des langages synchrones (OC, SC, DC, CP...). Ces formats ne sont pas figés et évoluent en permanence. En particulier la société ESTEREL TECHNOLOGIES qui commercialise le compilateur ESTEREL a récemment procédé à une refonte complète des formats internes. Toutefois, ces refontes portent principalement sur la forme et pas sur le fond : il restera toujours fondamentalement un format en « automate d'états fini », un format en « circuit séquentiel », un format en « points de contrôles »... J'ai pris le parti ici de les identifier par leur nom actuel plutôt que par leur définition.

Le format OC (« Object Code ») est un format intermédiaire commun aux compilateurs LUSTRE et ESTEREL.[1] Un programme OC est un **automate d'états fini** couplé à une mémoire bornée pour les calculs sur les variables. Dans chaque état de l'automate, on trouve du code séquentiel représenté par un graphe acyclique d'actions (DAG, acronyme de « Directed Acyclic Graph »). Le problème de ce format est l'explosion combinatoire due au parallélisme synchrone. Pour un programme de taille moyenne, le nombre d'états de l'automate peut atteindre plusieurs millions.

C'est pour cela que deux autres formats ont été proposés, d'une part DC (« Data-flow Code ») pour les langages LUSTRE et SIGNAL, et d'autre part SC (« Sequential Code ») pour le langage ESTEREL.

Avant de décrire DC, il est nécessaire de parler des horloges. LUSTRE et SIGNAL ont en effet en commun d'être des **langages synchrones flot-de-données**, dans lesquels toute variable manipulée par le programme est un **flot**, c'est-à-dire une séquence infinie de valeurs d'un même type. Il en va de même de LUCID Synchrone, mais le format de sortie du compilateur est CAML, qui est radicalement

[1] Compilateur LUSTRE : `http://www-verimag.imag.fr/SYNCHRONE`. Compilateur ESTEREL académique : `http://www-sop.inria.fr/esterel.org/Html/Downloads/Downloads.htm`.

différent des deux autres, et je n'en parlerai pas ici. Dans ce contexte, une horloge est une forme de type temporel [29, 42]. L'**horloge** d'un flot est la séquence des instants logiques où ce flot porte une valeur. En LUSTRE et SIGNAL, n'importe quel flot booléen peut être une horloge. Des opérateurs permettent de sous-échantillonner ou de sur-échantillonner des flots : le **sous-échantillonnage** crée une horloge plus lente (opérateur `when`) ; le **sur-échantillonnage** projette un flot sur une horloge plus rapide (opérateur `current` en LUSTRE et `default` en SIGNAL). En LUSTRE une horloge prédéfinie existe toujours, c'est l'**horloge de base** du programme, c'est-à-dire la suite de ses instants d'activation. D'ailleurs, toutes les horloges d'un programme LUSTRE peuvent être placées dans un **arbre d'horloges** dont la racine est justement l'horloge de base. En SIGNAL cela n'est pas toujours vrai et cette structure peut être une **forêt**, avec plusieurs racines qui sont des horloges non comparables. Pour cette raison, les programmes LUSTRE sont dits **endochrones** alors que les programmes SIGNAL sont dits **polychrones**. Quant aux programmes ESTEREL, l'existence du `tick`, sorte d'horloge de base extérieure au programme, permet de réagir à l'absence, c'est-à-dire d'écrire `present S else P`, où le code `P` est exécuté à chaque `tick` où `S` est absent ; aussi les programmes ESTEREL sont-ils dits **exochrones**.

Passons maintenant à DC. Un programme DC est un **réseau parallèle d'opérateurs**, qui se comporte comme une machine réactive transformant à chaque réaction un vecteur d'entrées en un vecteur de sorties. La séquence des réactions du programme définit la séquence de ses instants logiques. Les objets manipulés sont des flots, qui ont une valeur à chaque instant logique de leur horloge respective. Le programme calcule les flots de sortie en fonction des flots d'entrée (et également de flots locaux) au moyen d'**équations conditionnées**. Une telle condition d'activation est un flot booléen : à chaque instant où elle vaut `true`, l'équation calcule une nouvelle valeur, sinon elle recopie sa valeur précédente. À part ça, les programmes sont structurés en nœuds qui encapsulent des équations ; ils peuvent faire appel à des fonctions extérieures ainsi qu'à des procédures impératives ayant des effets de bord. Enfin une table indique les dépendances entre les équations. Ce format convient très bien à la compilation sur silicium, la vérification symbolique, et la génération de code réparti.

Le cas de SIGNAL est particulier. La compilation usuelle consiste à produire un **graphe flot-de-données hiérarchique** avec des **équations de dépendances conditionnelles**, appelé GFDS (« Graphe Flot-de-Données Synchronisé »).[2] Les nœuds en sont les signaux d'entrée et sorties et les variables, alors que les arrêtes sont étiquetées par des horloges. Ainsi, chaque nœud est situé dans une hiérarchie d'horloges. C'est à partir de ce GFDS que l'on peut produire du code DC. Toutefois, d'un certain point de vue, le langage SIGNAL est moins strictement synchrone que LUSTRE. On considère ainsi que SIGNAL est un **langage relationnel** alors que LUSTRE est un **langage fonctionnel**. Il est par exemple possible d'écrire des programmes dans lesquels les sorties sont « plus fréquentes » que les entrées, alors qu'un tel sur-échantillonnage est impossible en LUSTRE. Pour cette raison, SIGNAL permet de programmer des systèmes dits « ouverts ». En terme d'horloges, l'ensemble des horloges d'un programme LUSTRE est un **arbre**, dont la racine est l'horloge de base du programme, alors que pour un programme SIGNAL c'est en général une **forêt**, avec plusieurs racines qui ne sont pas liées entre elles. Aussi, tous les programmes SIGNAL ne peuvent pas être traduits vers DC (où par construction il n'y a qu'une seule horloge et des conditions d'activation, ce que l'on peut voir comme un arbre d'horloges plat, mais certainement pas une forêt). Très exactement, pour compiler un programme SIGNAL ayant une forêt d'horloges avec plusieurs racines vers DC, il faut lui ajouter un moniteur chargé de fournir l'information de présence et d'absence de chaque horloge racine par l'intermédiaire de booléens ; c'est-à-dire qu'il faut ajouter une horloge maître coiffant toutes les racines et transformant donc la forêt en arbre ; donc cela revient à renoncer au caractère faiblement synchronisé de SIGNAL. C'est pourquoi plusieurs autres formats, plus généraux que DC, ont été proposés, entre autres DC+ qui en est un sur-ensemble. Dans le cadre de ce document, je me limiterai volontairement aux programmes SIGNAL que l'on peut compiler vers DC, c'est-à-dire aux programmes que l'on pourrait traduire directement en LUSTRE, aux différences syntaxiques près.

Un programme SC est un **circuit séquentiel booléen** couplé à une table d'actions [14]. Le circuit

[2] Le compilateur associé au langage SIGNAL s'appelle POLYCHRONY : `http://www.irisa.fr/espresso/Polychrony`.

séquentiel code la **partie contrôle** du programme. La table d'actions code la **partie données** du programme : les actions manipulent en particulier les signaux d'entrée/sortie, les variables internes, les procédures et fonctions externes... Le circuit commande la table d'action, c'est-à-dire que certains fils du circuit déclenchent une action dès que leur valeur passe de `0` à `1`. Une fois le circuit généré, il est possible de générer du code cible en VHDL, C, JAVA... En fait, la représentation en circuit séquentiel est duale de la représentation en automate d'états fini : un circuit à n registres correspond à un automate à 2^n états. La génération de circuits séquentiels permet de produire du code de petite taille, généralement linéaire par rapport à la taille du programme ESTEREL source. Toutefois, le temps d'exécution du code circuit est en général prohibitif pour envisager d'embarquer le code. Mais il est à noter qu'il convient très bien pour la compilation sur silicium, ainsi que pour tout un éventail de méthodes d'analyse : simulation, vérification formelle, génération de tests...

Une troisième méthode de compilation des programmes ESTEREL a donc été proposée (avec des variantes) par une équipe de France Telecom R&D (Grenoble) [99], par des chercheurs de l'Université de Columbia (New-York) [47] et par des chercheurs de l'INRIA à Sophia-Antipolis [89]. Elle consiste à découper le code source en petites séquences d'instructions, séparées par des **points d'arrêt**, puis à les ordonnancer statiquement en restant compatible avec la sémantique synchrone d'ESTEREL. À chaque réaction du programme, seulement les séquences actives sont exécutées, ce qui optimise à la fois le temps de cycle et la taille du code. Les trois solutions varient selon le graphe de dépendance qui permet d'ordonnancer statiquement les séquences d'instructions et la manière de le traiter. Mais dans les trois cas, le format du code généré est sensiblement le même ; j'ai appelé ce format intermédiaire CP (« Control Points »). Cette méthode de compilation présente plusieurs avantages : le code généré est à la fois petit (presque autant que SC) et rapide (presque autant que OC), et de plus le temps de compilation est court (beaucoup plus que les autres méthodes de compilation d'ESTEREL).

Ces quatre formats intermédiaires (OC, DC, SC et CP) diffèrent de par la nature de la structure de contrôle des programmes. Ceci est résumé dans le tableau suivant :

format	structure du contrôle		
OC	séquentielle	explicite	statique
DC	parallèle	implicite	dynamique
SC	parallèle	implicite	dynamique
CP	parallèle	explicite	statique

TAB. 2.1 – Formats de sortie des compilateurs synchrones.

2.1.4 Exécution centralisée des programmes synchrones

Une fois compilé, le programme objet (OC, DC, SC ou CP) est immergé dans une **boucle d'exécution** : à chaque cycle les entrées sont lues depuis l'environnement, une réaction du programme est effectuée, au cours de laquelle les sorties sont calculées, et l'état suivant est déterminé :

```
loop each tick
   read inputs
   react and compute outputs
   compute next state
end loop
```

Bien sûr, la représentation de l'état interne et la façon dont il est calculé dépendent du format : en OC l'état de l'automate est un entier ; en DC l'état du réseau est la valeur des variables d'état (mémoires et conditions d'activation) ; en SC l'état du circuit est la valeur des registres booléens ; et en CP l'état du programme est la valeur des booléens qui indiquent quels points de contrôles doivent être exécutés.

Le `tick` qui déclenche chaque boucle est une horloge temps-réel du système. La succession des instants d'activation de la boucle d'exécution est l'**horloge de base** du programme (voir la section 2.1.3). C'est elle qui définit le **rythme** d'exécution du programme et le rythme de lecture / écriture de ses entrées / sorties.

Une telle boucle d'exécution offre l'énorme avantage de faciliter grandement la vérification des contraintes temporelles : il suffit en effet de borner la durée du corps de la boucle et de vérifier que cette durée est inférieure au temps maximal alloué par des contraintes temporelles. On se ramène ainsi au problème de l'évaluation du WCET du corps de la boucle, qui est un programme impératif. Des détails sur l'implémentation des boucles d'exécution peuvent être trouvés dans [4].

2.2 Mises en œuvre réparties

2.2.1 Le besoin de répartition automatique

Il y a cependant une caractéristique essentielle des systèmes embarqués à laquelle le modèle synchrone n'apporte pas de réponse satisfaisante, c'est que ce sont des **systèmes répartis**, c'est-à-dire qu'ils sont censés être exécutés sur des architectures multi-processeurs à mémoire répartie. Le format final doit donc être celui de plusieurs code objets communicant entre eux au moyen de diverses primitives de communication (rendez-vous, files d'attente, boîtes aux lettres...).

Le domaine de la répartition automatique de programmes est vaste et des recherches actives existent depuis longtemps. Par exemple, [48] présente un état de l'art de ce domaine. Il y a même, dans certains articles, une distinction entre **parallélisme** et **répartition** : le premier concerne plutôt les systèmes où le parallélisme est massif alors que le second concerne plutôt les systèmes où le parallélisme a un gros grain. Dans ce document, je parle donc de répartition et non pas de parallélisation, tout en continuant à parler du parallélisme du système.

Dans la communauté scientifique française, l'expression « distribution automatique » est souvent confondue avec « répartition automatique ». Pourtant, les deux verbes répartir et distribuer ne sont pas exactement synonymes : **répartir** c'est « partager une quantité ou un ensemble en plusieurs parts », alors que **distribuer** c'est « donner une partie d'une chose à plusieurs personnes ». Donc, quand il s'agit d'un programme informatique, le verbe correct est répartir. Quant à l'anglais, il existe un seul verbe, « to distribute », d'où l'expression « automatic distribution ».

Je m'intéresse donc à la **répartition des programmes synchrones**. Cela présente plusieurs particularités :

- Le programme source est parallèle et non pas séquentiel comme avec un langage de programmation classique (C, PASCAL...). Mais ce **parallélisme d'expression** sert au programmeur à concevoir son application en termes de modules parallèles coopérant au comportement global désiré ; il n'est pas, a priori, en rapport avec le **parallélisme d'exécution**, qui lui vient du au fait que l'architecture cible est répartie.
- L'**analyse de causalité** faite par le compilateur empêche, a priori, de faire de la compilation modulaire. L'analyse de causalité sert à déterminer le comportement séquentiel de plusieurs modules parallèles ayant entre eux des relations de synchronisation et de communication. Elle permet de plus de déterminer exactement quelles sont les variables présentes et absentes durant la réaction courante. D'ailleurs, un programme est déclaré non causal quand il n'est pas possible de déterminer cette information de présence/absence pour toutes ses variables. C'est cette information qui permet aux programmes synchrones de **réagir à l'absence**. À ma connaissance ce sont même les seuls langages à offrir cette capacité, laquelle est bien utile pour spécifier des comportements réactifs. Par exemple en ESTEREL on peut écrire aussi bien l'instruction de contrôle `present X then P`, dans laquelle `P` est exécuté si l'entrée `X` est présente, que `present X else P`, dans laquelle `P` est exécuté si l'entrée `X` est absente.

Ces deux caractéristiques expliquent qu'adapter une méthode classique de répartition automatique au cas des langages synchrones n'est certainement pas immédiat, voire même problématique. Ceci explique que le domaine de la répartition automatique des programmes synchrones ait été un domaine aussi actif pendant les quinze dernières années.

Avant d'étudier les différentes méthodes possibles de répartition automatique (à la section 2.2.3), je présente tout d'abord les moyens de communication utilisés.

2.2.2 Les primitives de communication

Une certaine forme de communication et de synchronisation entre les programmes répartis est requise. Ici, mon but est d'être efficace, simple, et de maximiser le parallélisme effectif. La **communication asynchrone** permet d'insérer les émissions le plus tôt possible dans la structure de contrôle du programme, et d'insérer les réceptions le plus tard possible, ce qui minimise l'impact de la latence de communication induite par le réseau [45].

Je choisis par conséquent de transmettre les valeurs d'un site à l'autre au moyen files d'attente FIFO (« First In First Out »). Chaque file étant identifiée par un unique identifiant `id` (qui comprend au minimum le site destination `dst`), j'utilise donc les deux primitives de communication suivantes :

- Sur le site source `src`, la primitive d'émission `send(id,var)` envoie la valeur courante de la variable `var` dans la file `id`.
- Sur le site destination `dst`, la primitive de réception `var:=receive(id)` extrait la première valeur de la file `id` et l'affecte à la variable `var`. Puisque l'architecture cible est à mémoire distribuée, `var` est la copie locale de la variable distante, tenue à jour par le site `src` grâce aux `receive`.

Ces primitives réalisent à la fois le transfert de données et la synchronisation entre le site émetteur et le site récepteur. En effet, la réception d'une valeur a toujours lieu *après* son émission. En particulier, quand la file est vide, le `receive` est bloquant. La seule contrainte que j'impose au réseau est que celui-ci doit préserver l'intégrité et l'ordre des messages. Ensuite, à condition que les `send` soient insérés sur un site dans le même ordre que les `receive` correspondants sur l'autre site, cela garantira que les valeurs ne seront pas mélangées.

2.2.3 Les différentes méthodes de répartition automatique

Les méthodes de répartition automatique peuvent être classifiées en trois classes ; dans tous les cas, le point de départ est un programme source centralisé écrit dans un langage synchrone :

1. Découper le programme source en plusieurs fragments, les compiler séparément et les faire communiquer **harmonieusement**, c'est-à-dire sans inter-blocage et de telle façon que le comportement du programme source centralisé soit équivalent au comportement conjoint des fragments compilés.

2. Obtenir par compilation globale un programme séquentiel pour chaque fragment, là encore communicant harmonieusement.

3. Compiler tout d'abord le programme source vers un programme objet centralisé et séquentiel, puis le répartir.

La **première solution** semble idéale car c'est la plus simple. Toutefois, Pascal Raymond a démontré que c'est impossible en toute généralité [91]. Le contre-exemple le plus simple est illustré par la figure 2.1 :

```
node MAIN (I1:int) returns (O2:int);
var O1,I2:int;
let
  O1 = I1;
  O2 = I2;
  I2 = O1;
tel;
```

(a)

```
I2 = O1;
```

(b)

```
O1 = I1;
O2 = I2;
```

(c)

FIG. 2.1 – (a) Le programme LUSTRE `MAIN` ; (b) un premier fragment ; (c) un second fragment.

La compilation du programme `MAIN` de la figure 2.1(a) donne le code séquentiel `O2:=I1`. Quand je compile séparément le fragment de la figure 2.1(b), j'obtiens le code séquentiel `I2:=O1`. Mais quand je compile séparément le fragment de la figure 2.1(c), je peux obtenir deux entrelacements différents :

1. Ou bien `O1:=I1; O2:=I2`. Dans ce cas, les communications qui sont ajoutées pour communiquer avec le premier fragment permettent d'obtenir le même comportement qu'en compilant le programme `MAIN`. En effet, les communications ont lieu par l'intermédiaires de FIFOs et le `receive` est bloquant quand la file est vide (voir la section 2.2.2).

```
O1 := receive(B);
I2 := O1;
send(B,I2);
```

(a)

```
O1 := I1;
send(A,O1);
I2 := receive(A);
O2 := I2;
```

(b)

FIG. 2.2 – (a) Le programme du premier fragment ; (b) Le programme du second fragment.

2. Ou bien `O2:=I2; O1:=I1`. Dans ce cas, les communications qui sont ajoutées entraînent un inter-blocage puisque chaque programme est en attente de l'autre :

```
O1 := receive(B);
I2 := O1;
send(B,I2);
```

(a)

```
I2 := receive(A);
O2 := I2;
O1 := I1;
send(A,O1);
```

(b)

FIG. 2.3 – (a) Le programme du premier fragment ; (b) Le programme du second fragment.

La **seconde solution** est séduisante mais très complexe du fait que le répartiteur de code doit également résoudre l'analyse de causalité et faire en sorte que le programme réparti obtenu soit toujours capable de réagir à l'absence. Or, si on découpe un programme synchrone en deux (ou plus) fragments, comment faire pour qu'un fragment donné réagisse à l'absence d'une entrée de l'autre fragment ? Cette solution est celle retenue pour les programmes SIGNAL [5, 6, 82, 26, 10, 90] (dont on trouvera en section 2.7.7 un rapide compte-rendu). Sans entrer dans les détails, elle consiste à compiler chaque fragment en tenant compte des contraintes d'ordonnancement induites par les autres fragments. Si je reprends l'exemple de la figure 2.1, il faut séquentialiser les équations du programme LUSTRE `MAIN`, ce qui donne `O1 = I1; I2 = O1; O2 = I2`, puis compiler chaque fragment en tenant compte de cet ordre total pour être sûr d'aboutir au programme réparti de la figure 2.2 et non à celui de la figure 2.3.

Je me place quant à moi dans le cadre de la **troisième solution**. Elle présente l'avantage de permettre la mise au point au préalable de l'application sur le code séquentiel centralisé, ce qui est toujours plus facile et plus rapide que de mettre au point un programme réparti. Il peut sembler paradoxal de partir d'une spécification synchrone intrinsèquement parallèle, de la compiler en un programme objet séquentiel, pour après la re-paralléliser. En fait ça ne l'est pas car le **parallélisme de spécification** (celui du programme source) est a priori différent du **parallélisme d'exécution** (celui du programme réparti obtenu). En effet, le parallélisme de spécification est le résultat du programmeur qui a choisi de concevoir son application sous la forme d'un ensemble de composants évoluant en parallèle pour atteindre le comportement global souhaité, alors que le parallélisme d'exécution est une exigence venant de l'architecture cible.

Remarquons que dans les formats SC et CP, la structure de contrôle est certes parallèle, mais beaucoup de calculs ont en réalité été séquencialisés.

2.2.4 Systèmes globalement asynchrones localement synchrones (GALS)

À l'issue de la répartition, j'obtiens un programme réparti où chaque composant est synchrone et où les composants communiquent entre eux de façon asynchrone. Un tel système peut donc être qualifié de **globalement asynchrone localement synchrone** (GALS [39]). Ce concept est utilisé à la fois en logiciel et en matériel. En logiciel, ce concept sert à composer des blocs sous la forme d'automates d'états finis et à les faire communiquer de façon asynchrone [7]. Cette approche est particulièrement bien adaptée à la conception de systèmes embarqués. En matériel, de plus en plus de circuits sont conçus sous la forme de blocs synchrones communicant de façon asynchrone plutôt que comme de gros circuits synchrones [86]. Cette méthode évite d'avoir à propager l'horloge globale partout dans le circuit, tâche toujours difficile et coûteuse en énergie, et contribue à diminuer la consommation électrique du circuit GALS obtenu [67].

2.2.5 Plan de lecture

Je m'intéresse aux mises en œuvre réparties de programmes synchrones, selon la troisième méthode (voir la section 2.2.1). Mon but est de produire **automatiquement du code réparti et embarqué**. C'est la conjonction de ces deux adjectifs qui fait toute la difficulté de la tâche.

Comme le suggère le tableau 2.1, quatre approches sont envisageables pour la répartition automatique, à partir des quatre formats intermédiaires OC, DC, SC et CP. Ces quatre approches ont en commun le fait de compiler au préalable le programme source en un programme objet centralisé unique, ce qui offre comme avantages de lui appliquer toutes les méthodes et outils classiques d'optimisation, de vérification formelle, de génération de tests... Dans tous les cas, si l'utilisateur désire obtenir n processus répartis, alors il doit fournir une partition de l'ensemble des signaux d'entrée/sortie du programme en n sous-ensembles. Le but est alors de construire n programmes communicant harmonieusement.

Je présente dans ce chapitre les trois approches correspondant aux formats OC, SC et CP. La répartition de programmes DC semble faisable mais n'a aujourd'hui été abordée par personne.

Comme je l'expliquerai à la section 2.3, la répartition de programmes OC souffre du principal défaut de ce format, à savoir l'explosion combinatoire de la taille du code. Néanmoins, pour toute une classe de programme de taille raisonnable, elle est parfaitement réalisable.

La répartition de programmes SC est quant à elle présentée à la section 2.4. Elle souffre du défaut suivant : pour que la répartition apporte un gain, il faut que le programme source SC soit orienté données ; seulement de tels programmes sont plutôt programmés en LUSTRE ou en SIGNAL qu'en ESTEREL qui est un langage adapté aux problèmes orientés contrôle. Ce défaut définit en fait d'un des verrous principaux au succès de toute méthode de répartition automatique : les optimisations transversales à la structure de contrôle des programmes répartis. En effet, en SC, il n'y a pas un mais plusieurs chemins de contrôle, et de plus ils sont dynamiques, au sens où des branchements peuvent dépendre de la sortie d'autres portes du circuit.

En revanche, en OC, de nombreuses optimisations sont possibles, par exemple la minimisation de la structure de contrôle et l'élimination des messages redondants (voir la section 2.3). Pour prendre une comparaison, c'est le passage du verrou inter-procédural qui a fait le succès de l'interprétation abstraite.

La répartition de programmes CP est présentée à la section 2.5. Ici, le problème du parallélisme de la structure de contrôle se pose également, mais de façon moins difficile que pour SC. En effet, la structure de contrôle a un unique chemin de contrôle bien défini, même si celui-ci n'est pas parfaitement séquentiel puisqu'il comporte des branches parallèles. L'autre avantage sur SC est que ce chemin de contrôle est connu statiquement.

La section 2.6 s'attaque à un problème orthogonal et néanmoins lié, celui de l'exécution des programmes synchrones répartis. La section 2.7 compare les approches qui ont été développées dans ce chapitre avec d'autres approches existant dans la littérature. Enfin, la section 2.8 conclut ce chapitre en proposant des pistes de recherches futures.

Tout au long de ce chapitre, on trouvera des références aux autres approches présentes dans la littérature. Quand elles concernent directement une de mes approches, je les mentionne dans la section concernée. Quand elles concernent plusieurs approches en même temps ou quand le lien est moins direct, je les mentionne à la fin de ce chapitre, dans la section 2.7.

2.2.6 Le problème des tâches de longue durée

Un autre problème qui se pose est la prise en compte des **tâches de longue durée** dans la programmation synchrone. De telles tâches sont définies par :

1. leur **durée d'exécution** qui est *supérieure* au temps de réaction maximal du programme, mais qui est en revanche *connue* et *bornée* ;
2. leur **rythme d'exécution maximal** qui est lui aussi *connu* et *borné*.

Bien sûr, le rythme et la durée d'exécution d'une tâche de longue durée doivent être consistants avec les performances du processeur sur lequel tourne l'application. Sinon, les contraintes temps-réel ne pourront jamais être satisfaites. De nombreux programmes réactifs comportent de telles tâches. C'est le cas du logiciel de contrôle/commande CO3N4 développé par SCHNEIDER ELECTRIC pour les centrales nucléaires.

Le principal problème soulevé par les tâches de longue durée est que leur temps d'exécution est trop grand pour qu'elles puissent apparaître telles quelles dans le programme objet, qu'il soit au format OC, SC ou CP. En effet, considérons un automate OC : son temps de réaction est défini comme étant le $\max$ des temps d'exécution de toutes ses transitions. Si une tâche de longue durée apparaissait dans une transition, alors le temps de réaction serait forcément supérieur à la durée de cette tâche, ce qui rendrait caduque l'hypothèse de synchronisme (voir la section 2.1.2). Pour un circuit SC, le temps de réaction est défini comme étant le temps de calcul des sorties et des registres à partir des entrées du circuit. Enfin, pour un programme CP, il est défini comme étant le temps d'exécution de la liste courante des séquences d'instructions. Donc dans ces deux cas, les tâches de longue durée posent le même problème qu'en OC. Je présenterai dans la section 2.3.7 une méthode permettant, grâce à la répartition sur plusieurs sites de calculs et à la désynchronisation du rythme, d'exécuter des programmes synchrones ayant des tâches de longue durée tout en respectant la sémantique synchrone.

2.3 Répartition automatique de programmes OC

Mes travaux sur la répartition automatique de programmes OC ont été principalement effectués pendant ma thèse, sous la direction de Paul Caspi (VERIMAG, Grenoble). La raison pour laquelle je les mentionne dans ce document d'habilitation est qu'ils constituent la base de nombreux travaux effectués par la suite, sur la répartition de code SC, sur la répartition de code CP, sur la prise en compte des tâches

de longue durée... Au cours de ces récents travaux, j'ai aussi travaillé en collaboration avec Jean-Claude Fernandez et Xavier Nicollin (VERIMAG, Grenoble), Marc Pouzet (LRI, Orsay) et Fabrice Salpétrier (stagiaire ESISAR, Valence).

2.3.1 Modèle des programmes OC

Un programme OC est un **automate d'états fini** avec dans chaque état un graphe acyclique orienté binaire d'actions (un DAG binaire). Chaque DAG possède une racine, plusieurs nœuds unaires et binaires, et une ou plusieurs feuilles :

- Les nœuds unaires sont des actions séquentielles, qui peuvent être ou bien :
  - une indication que les variables d'entrée du programme ont été lues et que leur valeurs sont disponibles dans la mémoire locale du programme : `go(...,`$\texttt{in}_i$`,...)`, où les $\texttt{in}_i$ sont les entrées,
  - une affectation à une variable locale ou de sortie : `var:=exp`, où `exp` peut contenir des appels à des fonctions externes,
  - une écriture de sortie : `write(var)`,
  - un appel de procédure externe : `proc(...,`$\texttt{var}_i$`,...)(...,`$\texttt{val}_j$`,...)`, où les $\texttt{var}_i$ et $\texttt{val}_j$ sont respectivement les paramètres variable et valeur.
- Les nœuds binaires sont des branchements déterministes : `if (var) then p else q endif`, où `p` et `q` sont des sous-DAGs.
- Les feuilles, et elles-seules, sont des changements d'état : `goto n`.

Un programme OC interagit avec son environnement explicitement via les actions `go`. Dans un programme avec une liste d'entrées $\texttt{in}_1,\ldots,\texttt{in}_n$, le premier nœud de chaque DAG est l'action `go(`$\texttt{in}_1$`,...,`$\texttt{in}_n$`)`.

Ce format d'automate est très général puisque tout programme écrit dans un langage classique impératif peut être compilé vers ce format. En fait, tout programme OC peut être traduit en un organigramme de blocs de base et vice-versa.

Enfin, comme je l'ai dit à la section 2.1.4, un programme OC est immergé dans une boucle d'exécution : à chaque cycle les entrées sont lues depuis l'environnement, une transition de l'automate est exécutée (c'est-à-dire le code du DAG courant), au cours de laquelle les sorties sont calculées ainsi que le numéro de l'état suivant.

2.3.1.1 Primitives de communication

Dans la mesure où la structure de contrôle des programmes OC est *séquentielle*, l'ordre dans lequel seront exécutées des communications, depuis un site donné vers un autre site donné, peut être déterminé statiquement. Je choisis donc d'identifier chaque file FIFO par un doublet $\langle \texttt{src}, \texttt{dst} \rangle$, où `src` est le site source et `dst` est le site destination. Les primitives sont celles décrites dans la section 2.2.2 :

- Sur le site `src`, la primitive d'émission `send(dst,var)` envoie la valeur courante de la variable `var` dans la file $\langle \texttt{src}, \texttt{dst} \rangle$; elle est non-bloquante.
- Sur le site `dst`, la primitive de réception `var:=receive(src)` extrait la première valeur de la file $\langle \texttt{src}, \texttt{dst} \rangle$ et l'affecte à la variable `var` ; elle est bloquante tant que la file est vide.

2.3.2 Localisation des actions

Comme je l'ai dit à la section 2.2.3, je n'essaye pas d'obtenir la *meilleure* répartition possible. Je détermine la localisation de toutes les actions et variables à partir de la localisation des entrées/sorties et des dépendances de données existant entre les actions. Par exemple, si la sortie `X` appartient au site `L`, alors je vais y localiser l'action `emit X(Y)`. Et donc la variable `Y` et l'action `Y:=2` vont toutes deux être également localisées sur `L`, à moins qu'elles ne soient déjà localisées ailleurs.

De nombreux travaux existent sur le sujet précis de trouver la meilleure répartition modulo un critère, par exemple l'équilibrage de la charge entre les sites de calcul. Ce problème est connu pour être NP-difficile [81], donc les gens ont souvent recours à des heuristiques. Il serait tout à fait possible d'utiliser une méthode existante et de l'adapter à mon problème de répartition des programmes synchrones, mais d'une part c'est hors-sujet par rapport à ce présent document, et d'autre part c'est un objectif contradictoire avec celui d'une répartition dirigée par l'utilisateur en fonction des besoins propres de son système embarqué (avec notamment une localisation imposée des capteurs et des actionneurs).

2.3.3 Aperçu de l'algorithme de répartition

L'idée originale est de Paul Caspi. La première mention en a été faite en 1988 dans [25], article concomitant avec [28] qui expose un algorithme de répartition de programmes FORTRAN basé sur les mêmes principes. L'algorithme est présenté en détails dans [49] et [35]. Il comporte les étapes successives suivantes :

1. Assigner un site de calcul unique à chaque action séquentielle, en fonction des **directives de répartition** fournies par l'utilisateur ; ces directives se présentent sous la forme d'une partition de l'ensemble des entrées et sorties du programme en n sous-ensembles, un pour chaque site de calcul du programme réparti final ; comme je l'ai dit à la section 2.3.2, je n'essaye pas ici de déterminer la meilleure localisation possible.
2. Répliquer le programme sur chaque site de calcul.
3. Sur chaque site de calcul, supprimer les actions séquentielles qui n'appartiennent pas au site courant.
4. Dans chaque état de l'automate OC, insérer des actions d'émission (`send`) de manière à résoudre les dépendances de données entre toute paire de sites de calcul ; ces dépendances de données sont dues aux suppressions de l'étape 3.
5. Dans chaque état de l'automate OC, insérer les actions de réception (`receive`) de manière à coïncider avec les actions d'émission précédemment insérées.
6. Si cela est requis par l'utilisateur, resynchroniser le programme réparti résultant afin d'éviter que certains sites soient purement producteurs de valeurs et que d'autres soient purement consommateurs de valeurs, ce qui pourrait provoquer une explosion de la taille des files d'attente ; la resynchronisation consiste, dans chaque état de l'automate OC, à insérer des communications vides de manière à ajouter des communications supplémentaires sans valeur.

À partir d'un programme OC centralisé et d'un fichier de directives de répartition, on obtient ainsi un programme réparti constitué de n fragments, un pour chaque site de calcul. Le programme de chaque site de calcul est lui-même un automate OC, qui ne calcule que les variables d'entrée/sortie qui lui ont été assignées par les directives de répartition. Les fragments communiquent harmonieusement (c'est-à-dire sans inter-blocage) entre eux au moyen des primitives `send` et `receive`, afin de continuer à calculer les sorties exactement comme le ferait le programme centralisé initial.

Afin de rendre le programme réparti ainsi obtenu le moins sensible possible à la latence du réseau de communication, les émissions sont insérées le plus tôt possible dans les états des automates OC (c'est-à-dire le plus proche possible de la racine de chaque état), alors que les réceptions sont insérées le plus tard possible (c'est-à-dire le plus proche possible des feuilles de chaque état). Cependant, en raison de la notion physique d'état (matérialisée par l'action `go` présente au début de chaque état), aucun des deux algorithmes d'insertion ne franchit la barrière des états. Autrement dit, les émissions et les réceptions sont insérées état par état.

2.3.4 Preuve de correction de l'algorithme de répartition

Avec Benoît Caillaud, Paul Caspi et Claude Jard, nous avons établi la preuve de correction de cet algorithme [27]. Il faut pour cela démontrer que le comportement du programme centralisé initial P est équivalent au comportement du programme réparti final $\tilde{P}$. Voici les étapes principales de cette preuve.

Tout d'abord, P est modélisé par un **système de transitions étiquetées** par ses actions (STE) ; l'alphabet des actions est noté Σ. Son **comportement** $\|P\|$ est le langage de ce STE, c'est-à-dire l'ensemble des traces d'actions finies et infinies qu'il engendre. Puis, les **directives de répartition** sont modélisées par une partition de Σ en n sous-ensembles, n étant le nombre de sites de calcul. Ensuite, les dépendances de données entre les actions définissent une **relation de commutation** notée $\propto$ et telle que : $a \propto b \Leftrightarrow def(a) \cap use(b) = \emptyset \wedge a \neq b$, où $def(a)$ et $use(a)$ sont respectivement les ensembles de variables définies et utilisées par l'action a. Cette relation $\propto$ induit une **relation de réécriture** $\rightharpoonup$ sur les traces d'actions : $u \rightharpoonup v$ si v est une des réécritures de u selon $\propto$. En raison de l'asymétrie de la définition de $\propto$, $\rightharpoonup$ est en réalité une **semi-commutation** [40]. L'ensemble de toutes les réécritures possibles de u par $\rightharpoonup$ est la fermeture transitive de u par $\rightharpoonup$, notée $[u]_{\rightharpoonup}$. Répartir un programme séquentiel P revient à autoriser des ordres d'exécution différents de l'ordre total exprimé par P, en fonction de la relation de commutation entre les actions. Ainsi, $[\|P\|]_{\rightharpoonup}$ est l'ensemble maximal des comportements qui peuvent être obtenus par répartition de P. Le problème est que, en général, $[\|P\|]_{\rightharpoonup}$ ne peut pas être reconnu par un automate fini déterministe, car ce n'est pas un langage régulier. En revanche, $[\|P\|]_{\rightharpoonup}$ est égal à l'ensemble des **extensions linéaires** d'un certain **ordre partiel**. C'est pourquoi nous avons introduit un nouveau modèle basé sur les ordres partiels :

1. Tout d'abord, P est transformé en un **automate d'ordres** $\hat{P}$, c'est-à-dire un STE étiqueté par des ordres partiels, en transformant chaque action a étiquetant P en un ordre partiel $\hat{a}$ qui représente les dépendances de données entre a et les autres actions. Le langage de cet automate d'ordre est l'ensemble des traces finies et infinies d'ordres partiels qu'il engendre. En définissant une **opération de concaténation** sur les ordres partiels, chaque trace est alors elle-même un ordre partiel. Ainsi, le langage de $\hat{P}$ est un ensemble $\|\hat{P}\|$ d'ordres partiels finis et infinis. Le résultat principal de [27] est que $[\|P\|]_{\rightharpoonup}$ est l'ensemble des extensions linéaires de la projection de $\|\hat{P}\|$ sur Σ.
2. Ensuite, $\hat{P}$ est transformé en un ensemble d'automates communicants $\tilde{P}$, cela en transformant les dépendances de données liant des actions appartenant à des sites de calculs distincts en **actions de communication** (`send` et `receive`). Le second résultat de [27] est que ces transformations préservent le comportement de $\hat{P}$. Ainsi, $\|\hat{P}\| = \|\tilde{P}\|$.
3. Il reste à démontrer que les **propriétés de sûreté** satisfaites par P le sont aussi par $\tilde{P}$. De telles propriétés expriment le fait que quelque chose ne se produira jamais, ou qu'un prédicat sera toujours satisfait : elles sont spécifiées par des formules de logique temporelle entre les signaux d'entrée et de sortie du programme. Or, dans le cas d'un programme synchrone, l'évolution temporelle d'un signal est représentée par le flot de ses valeurs. Aussi, pour garantir que les propriétés de sûreté restent satisfaites, il faut resynchroniser fortement $\tilde{P}$ (c'est l'étape 6 de l'algorithme de répartition de la section 2.3.3). La **synchronisation forte** préserve le cycle global de P : les signaux de sorties émis par P au cours d'un même cycle seront émis par $\tilde{P}$ au cours du même cycle global.

2.3.5 Élimination des messages redondants

Ne pas franchir la barrière des états lors des algorithmes d'insertions des émissions et des réceptions les rend plus facile à mettre au point et à en démontrer formellement la correction. Mais cela peut entraîner des **messages redondants**, au sens où la valeur transmise est déjà connue par le site destinataire, par exemple parce qu'elle a déjà été reçue dans un état précédent et que cette variable n'a pas été modifiée par le site émetteur depuis. Supprimer les messages redondants est intéressant car cela diminue le temps

de réaction global du programme réparti. Dans ce but, j'ai proposé un algorithme d'analyse statique qui fonctionne en deux passes :

1. Une première passe globale sur l'automate OC complet de chaque site de calcul `s` détermine, pour chaque état, l'ensemble $\texttt{Known}_\texttt{s}$ des variables distantes dont la copie locale est à jour au début de cet état. Pour cela, à partir des ensembles $use(a)$ et $def(a)$ de chaque action a, je calcule l'ensemble des variables modifiées par le DAG de chaque état ; puis je calcule par un point fixe l'ensemble des variables connues au début de chaque état `e` en tenant compte de toutes les transitions de l'automate OC qui aboutissent à `e`.
2. Une seconde passe locale sur chaque état du programme de chaque site de calcul élimine les émissions redondantes. Pour cela, je parcours le DAG de chaque état de sa racine jusqu'aux feuilles en mettant à jour l'ensemble des variables connues grâce aux ensembles $use(a)$ et $def(a)$ de chaque action a et en supprimant toute action `send(dst,v)` telle que $\texttt{v} \in \texttt{Known}_{\texttt{dst}}$.

J'applique cet algorithme entre les étapes 4 et 5 de l'algorithme de répartition, ce qui permet de n'insérer les réceptions qu'une seule fois, sur un programme n'ayant plus d'émissions redondantes. Il est publié dans les articles [51, 50].

On trouve dans la littérature de nombreux articles sur l'optimisation des communications dans les programmes parallèles [60, 57, 2, 44, 98, 74, 61, 1]. Toutes ces approches partent d'un programme dont le flot de contrôle est **bien structuré**, c'est-à-dire qu'il ne contient que des boucles et des instructions conditionnelles. En particulier, toute boucle est supposée avoir un unique point d'entrée. Pour cette raison, les algorithmes décrits dans ces articles se concentrent sur le déplacement des communications en dehors des boucles. Mon algorithme lui fonctionne avec des programmes dont la structure de contrôle est quelconque, donc comprenant des `goto`. Aussi est-il capable de traiter des boucles ayant plusieurs points d'entrée et de sortie. En revanche, mon algorithme ne traite pas les tableaux, ce que font très bien la plupart des autres algorithmes ; la raison est que les types tableaux sont traités comme des types *externes* dans un programme OC.

2.3.6 Désynchronisation du rythme

Une autre conséquence du non-franchissement de la barrière des états lors de l'insertion des émissions et des réceptions est que tous les fragments d'un même programme réparti ont la même structure de contrôle, c'est-à-dire que ces automates OC ont tous le même nombre d'états et dans chaque état les mêmes branchements `if` et les mêmes feuilles `goto`. En fait ils sont tous synchronisés et ils avancent tous au *même rythme*, celui des entrées du programme centralisé initial. Cela présente deux inconvénients. Le premier est que, puisque tous les fragments font les mêmes branchements, tous ceux dont le site de calcul n'est pas propriétaire de la variable testée par un branchement donné doivent recevoir cette variable au moyen d'un `send`/`receive`. Le second inconvénient est que, tous les sites de calculs évoluant au même rythme, il est difficile de programmer des applications où certains calculs doivent intrinsèquement être effectués à des rythmes différents, par exemple parce que certaines entrées/sorties ont des horloges différentes. Ce problème particulier est impossible à résoudre tant que le programme reste centralisé, car la contrainte des rythmes différents viole l'hypothèse de synchronisme. En effet, le programme objet centralisé obtenu par compilation doit forcément évoluer au rythme des entrées/sorties les plus rapides, c'est-à-dire que le rythme de base du programme (défini à la section 2.1.4) doit forcément être celui des entrées/sorties les plus rapides, ce qui empêche de spécifier des calculs à des rythmes plus lents.

Afin de prendre en compte des rythmes différents, plusieurs approches existent. La programmation séparée de chaque site selon la méthode CRISYS [36, 30, 24] et les tâches asynchrones d'ESTEREL [87] ou de SIMULINK présentent l'inconvénient de repousser le problème à l'extérieur du programme synchrone, ce qui pose des problème de validation formelle. J'y reviendrai en détails dans la section 2.7.3.

J'ai proposé quant à moi une méthode permettant de désynchroniser les rythmes des fragments d'un programme réparti [55, 56]. Le principe consiste à partitionner les entrées/sorties du programme en sous-ensembles évoluant au même rythme, à répartir le programme OC selon ces directives de répartition, puis à optimiser localement la structure de contrôle des fragments répartis de telle manière que chaque fragment évolue au rythme de ses propres entrées/sorties, afin de lui laisser le temps de terminer son calcul, fut-il de longue durée. L'optimisation locale de la structure de contrôle des fragments répartis utilise une **bisimulation originale** (dont la formalisation est en partie due à Jean-Claude Fernandez), qui permet de déterminer, dans un STE donné S, quels branchements binaires peuvent être supprimés (c'est-à-dire remplacés par une transition simple vers une des deux branches du branchement) sans modifier le comportement de S [32] (comportement défini comme l'ensemble des traces finies ou infinies que S peut engendrer). Cette bisimulation est donc utilisable dans un cadre plus général. Or, dans un programme OC, les changements de rythme du programme sont exprimés justement par certains branchements, et la branche `then` est alors celle qui est exécutée suite au changement de rythme. Par conséquent, remplacer un branchement par une transition simple vers la branche `then` permet d'exécuter un des fragments répartis à un rythme plus lent. L'extension à des programmes SC ou CP de cette méthode représente, à mon avis, un challenge intéressant.

Dans tous les cas (programmation séparée ou désynchronisation après répartition automatique), cela n'est possible que si les fragments répartis ne communiquent qu'à travers des **délais unitaires** (quand il y a des cycles de dépendances de données, sinon c'est inutile). Mais dans ce cas, la compilation séparée est également possible, ce que propose par exemple le compilateur SIGNAL.

Remarquons que l'opérateur de délai unitaire existe dans tous les formalismes de programmation flots-de-données : « `pre` » en LUSTRE, « `$1` » en SIGNAL (car le temps, c'est de l'argent)... C'est l'équivalent du registre dans les circuits synchrones. Dans les langages impératifs, l'équivalent est obtenu en rangeant la valeur courante de la variable de communication dans une mémoire et en envoyant la valeur du cycle précédent. En ESTEREL, la primitive `pre` a même été ajoutée pour améliorer la lisibilité des programmes (même si la sémantique du `pre` en ESTEREL n'est pas exactement la même que celle du `pre` en LUSTRE).

2.3.7 Répartition de programmes LUSTRE dirigée par les horloges

Une conséquence utile de la désynchronisation des rythmes de calcul est que cela autorise à programmer des **tâches de longue durée** (définies à la section 2.2.6). La méthode que j'ai mise au point consiste à [55, 56] :

1. écrire un programme LUSTRE dans lequel chaque tâche de longue durée est cadencée à une horloge lente correspondant à son rythme d'exécution ;
2. puis à extraire l'arbre des horloges complet du programme et à le décorer avec les entrées/sorties ;
3. puis à attribuer un site de calcul à chaque horloge (ou, si on veut un grain plus grossier, à des sous-arbres) ;
4. puis à en déduire la localisation des entrées/sorties ;
5. puis à compiler ce programme LUSTRE en OC ; une des caractéristiques fondamentales du programme OC ainsi obtenu est que les calculs lents sont forcément situés dans la branche `then` d'un test de la forme `if (CK)`, où `CK` est justement l'horloge de ce calcul lent ;
6. et enfin à lui appliquer l'algorithme de répartition automatique avec les directives de répartition issues des horloges, en veillant bien à optimiser localement la structure de contrôle des fragments répartis à l'aide de la bisimulation décrite dans la section 2.3.6 ; cette bisimulation permet en effet de supprimer, sur le site de l'horloge `CK`, les branchements `if (CK)` mentionnés ci-dessus ; ceci a donc bien pour résultat de produire un programme réparti tel que les fragments correspondants aux horloges lentes peuvent être exécutés à un rythme lent.

Cette méthode est particulièrement intéressante : qu'on y songe, elle permet de redécouvrir des horloges dans des programmes répartis pourtant obtenus à partir de programmes OC dans lesquels il n'y a aucune horloge ! Mieux, à partir de programmes ESTEREL dans lesquels cette notion n'existe même pas, cette méthode permet de créer des programmes répartis désynchronisés, c'est-à-dire tels que les fragments ont des horloges différentes ! À ma connaissance, c'est la seule méthode offrant un tel résultat. Il serait possible de l'adapter au compilateur SIGNAL, mais cela reste à faire [78]. En effet, comme je l'ai dit à la section 2.3.6, le compilateur SIGNAL autorise la compilation séparée des fragments du programme source, à condition que ceux-ci ne communiquent qu'à travers des délais unitaires, ce qui est justement une condition nécessaire à la désynchronisation du rythme.

2.3.8 Implémentation et discussion

J'ai implémenté tous ces algorithmes dans l'outil `ocrep` [33], disponible sur le web.[3] Il fait environ 17.000 lignes de code C++. Une partie d'`ocrep` a fait l'objet d'un contrat de transfert technologique avec la société France Telecom R&D (voir la section 2.5).

Les travaux présentés dans cette section ont été publiés dans les articles [32, 27, 35, 51, 50, 55, 52, 56].

Outre SIGNAL (dont je présente les travaux relatifs à la répartition automatique à la section 2.7.7), d'autres approches pour la prise en compte des tâches de longue durée existent, mais à mon avis pas aussi abouties : il s'agit de GIOTTO, Real-Time Workshop pour SIMULINK, et les tâches asynchrones en ESTEREL. Je présente ces trois approches ci-dessous.

GIOTTO : C'est un compilateur pour systèmes embarqués, fondé sur des hypothèses très proches de l'abstraction synchrone : communications instantanées et calculs déterministes par rapport aux valeurs et aux temps [68, 69].[4] L'unité fonctionnelle de base en GIOTTO est la **tâche**, qui est un morceau de code exécuté périodiquement, avec une période statique. Au contraire, une horloge LUSTRE au sens le plus général peut définir un rythme dynamique, même si la vérification de la consistance entre l'horloge (qui est donc non périodique), la durée de la tâche (qui peut varier dynamiquement elle aussi) et la performance du processeur en est rendue plus complexe. Un programme GIOTTO peut également être annoté avec des **contraintes de plate-forme**, qui sont similaires à mes directives de répartition : par exemple, une contrainte peut indiquer qu'une tâche donnée doit être exécutée sur un processeur donné. Le compilateur GIOTTO ordonnance les tâches sur l'architecture cible, et garantit que la sémantique logique du programme source est préservée (à la fois la sémantique fonctionnelle et la sémantique temporelle). Cependant, un programme GIOTTO peut être **sur-contraint**, quand il n'autorise aucune exécution consistante avec les contraintes de plate-forme. Dans un tel cas, le compilateur rejette le programme comme étant non-valide. Au contraire, ma méthode produit toujours un programme réparti exécutable, mais la vérification après-coup des contraintes temps-réel est laissée aux soins de l'utilisateur. Dans la mesure où les programmes OC sont des automates d'état finis et déterministes, le calcul du WCET est facilité. Il reste qu'avec les processeurs modernes de type super-scalaire, avec plusieurs unités de calcul, cache à plusieurs niveaux et pipeline, l'indéterminisme est tel que le calcul précis du WCET est en réalité très compliqué.

REAL TIME WORKSHOP (RTW): C'est un générateur de code réparti pour programmes SIMULINK [97]. Dans SIMULINK, l'utilisateur peut affecter à différents blocs des rythmes différents, appelés « triggers », et similaires aux conditions d'activation de SCADE. Cela permet de traiter les blocs de longue durée. De même que les horloges de LUSTRE et les conditions d'activations de SCADE, les triggers de SIMULINK sont dynamiques. La figure 2.4(a) est un exemple d'un tel programme : la tâche B est cinq fois plus

[3] `ocrep` : `http://pop-art.inrialpes.fr/~girault/Ocrep`
[4] GIOTTO : `http://embedded.eecs.berkeley.edu/giotto`

longue que la tâche A. RTW produit alors un ordonnancement où la tâche B est préemptée par la tâche A, qui est plus rapide et a par conséquent une plus grande priorité : cet ordonnancement est montré dans la figure 2.4(b). À cause des préemptions successives de la tâche B par la tâche A, la tâche B est découpée en cinq fragments : B1, B2, B3, B4 et B5.

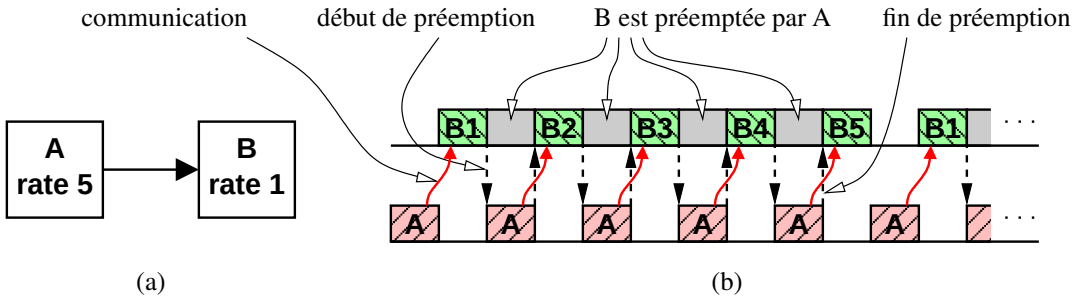


FIG. 2.4 – (a) Deux blocs SIMULINK ; (b) L'ordonnancement produit par RTW.

La communication de A vers B soulève un problème : comment la donnée produite par A durant son premier cycle, et non pas celles produites dans les cycles suivants, peut-elle être utilisée par B ? Comme on le constate dans l'ordonnancement de la figure 2.4(b), la tâche A envoie sa sortie *à chacun de ses cycles*, alors que la tâche B n'attend une donnée en entrée qu'*au début de son propre cycle*. La solution consiste à ajouter un **oracle nul** (« zero oracle » en anglais) entre A et B, qui hérite de la priorité de A et du rythme de B.[5] Le résultat est illustré dans la figure 2.5(a). Dans le nouvel ordonnancement de la figure 2.5(b), une seule donnée est envoyée et reçue.

Quand la communication a lieu d'une tâche lente vers une tâche rapide (situation symétrique de la figure 2.4(a)), la solution consiste à ajouter un **délai unitaire** (« unit delay » en anglais) entre les deux blocs, ce qui permet la compilation séparée. Enfin, il est à noter que cette solution est utilisée couramment par les ingénieurs qui développent en SIMULINK des systèmes ayant plusieurs rythmes.

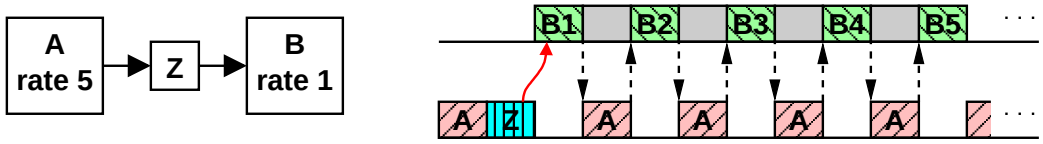


FIG. 2.5 – (a) Deux blocs SIMULINK avec un oracle nul ; (b) Le nouvel ordonnancement.

Les tâches asynchrones d'ESTEREL : Un programme ESTEREL peut démarrer une tâche asynchrone au moyen d'un signal de sortie dédié, appelé `exec`, et ensuite être averti de la terminaison au moyen d'un autre signal d'entrée dédié, appelé `return` [87]. Bien que moins puissant et flexible que les horloges de LUSTRE, ce mécanisme permet néanmoins de prendre en compte les tâches de longue durée dans un programme synchrone. Toutefois, de telles tâches sont traitée *externalement* par rapport au programme synchrone, alors que ma méthode permet de les traiter *à l'intérieur* même du programme LUSTRE.

[5] Un grand merci à Edward Lee, professeur à UC Berkeley, pour son explication détaillée du fonctionnement de RTW.

2.4 Répartition automatique de programmes SC

Mes travaux sur la répartition automatique de programmes SC ont été effectués en collaboration avec Clément Ménier (stagiaire ENS Lyon). Ils ont été publiés dans l'article [54].

2.4.1 Modèle des programmes SC

Un programme SC se compose d'une **partie contrôle**, un circuit booléen séquentiel synchrone, et d'une **partie donnée**, une table d'actions externes permettant de manipuler les variables du programme. Ces actions sont les mêmes que les actions séquentielles des programmes OC (voir la section 2.3.1).

Un programme a un ensemble de signaux d'entrée et de sortie, qui peuvent être **purs** ou **valués**, auquel cas le signal est associé à une variable locale du type correspondant. Les variables locales sont manipulées par les actions de la table. Les types de base sont prédéfinis (booléen, entier, réel), et les types complexes peuvent être définis et manipulés par des appels de procédures externes.

Le circuit séquentiel se compose de portes booléennes, de registres et de fils spéciaux qui déclenchent des actions de la table. Le programme a un comportement périodique avec une horloge qui commande tous les registres. Un des registres a pour rôle d'initialiser le contrôle : le registre `boot`. C'est le même fonctionnement que la boucle d'exécution des programmes OC (voir la section 2.1.4).

Dans sa représentation textuelle, un circuit est simplement une liste de fils numérotés. Chaque fil est connecté en entrée à une porte logique, représentée par une **expression booléenne d'entrée**, qui est ou bien une conjonction ou bien une disjonction de fils ou de négations de fils. De telles expressions ne peuvent pas être imbriquées, et deux expressions sont prédéfinies : `0` et `1`. La liste complète des fils est :

- Un fil `standard` définit un fil avec une expression booléenne d'entrée. Cela permet de construire des expressions booléennes complexes.
- Un fil `action` déclenche une action définie dans la table des actions. Cette action est exécutée dès que le fil porte la valeur `1`. Une action peut être ou bien une affectation de variable avec une expression quelconque en partie droite, ou bien un appel de procédure externe avec le cas échéant des paramètres variables et des expressions quelconques comme paramètres valeurs.
- Un fil `ift` déclenche une action de test d'expression, définie dans la table des actions. Cette action de test est exécutée dès que le fil porte la valeur `1`. Le fil `ift` prend alors comme nouvelle valeur le résultat de l'évaluation de l'expression.
- Un fil `input` a deux effets : d'une part il lit un signal d'entrée `E` et positionne la variable booléenne de présence correspondante à `1` si `E` est présent et à `0` sinon ; et d'autre part il propage la valeur `1` si `E` est présent, et `0` sinon. Un tel fil est donc en fait représenté par une partie `input` et une partie `ift`. Le `ift` teste le booléen de présence de l'entrée, et agit exactement comme un fil `ift`. Quand le signal est valué, il positionne également la valeur de la variable associée. C'est le seul fil sans expression d'entrée car il est exécuté à chaque cycle de l'horloge du programme.
- Un fil `output` correspond à un signal de sortie. Il agit exactement comme un fil `action` en déclenchant une action `emit` dès qu'il porte la valeur `1`. Si le signal de sortie est valué, alors cette action `emit` doit avoir en paramètre une expression du même type que celui du signal.
- Un fil `register` est un registre avec une seule broche d'entrée et une valeur initiale (`0` ou `1`). La valeur initiale du registre `boot` est `1` et sa broche d'entrée est connectée à `0`.

Je distingue donc deux classes de fils : ceux qui déclenchent une action (`action`, `ift`, `input` et `output`) et ceux qui n'en déclenchent pas (`standard` et `register`).

La sémantique des programmes SC est est héritée de la sémantique d'ESTEREL. Elle est basée sur **l'hypothèse de temps nul** : le circuit est vu comme un ensemble d'équations booléennes qui diffusent leurs résultats à toutes les autres instantanément. Puisque le circuit est acyclique (tous les cycles étant coupés par les registres), les équations peuvent être totalement ordonnées, de telle sorte que toute variable ne dépende que de variables précédemment définies. Ainsi, pour toute valuation du vecteur des

registres et des entrées, cet ensemble d'équations admet une solution unique, ce qui signifie que le circuit a un comportement unique [15].

De plus, je ne considère que des **programmes causaux**, c'est-à-dire tels que toute variable ne peut être modifiée que dans une branche parallèle de la structure de contrôle. Le but de cette propriété de causalité, qui n'a rien à voir avec la structure de contrôle elle-même, est uniquement d'éviter les programmes non-déterministes. Tous les programmes SC produits par le compilateur ESTEREL vérifient cette propriété.

Les programmes SC sont représentés graphiquement sous la forme d'un circuit synchrone où chaque action est directement attachée à son fil respectif. Un fil dont l'expression booléenne d'entrée est l'identité est représenté par un simple buffer (dessiné -▷-). Sinon il est représenté par la porte booléenne correspondant à son expression booléenne d'entrée.

La figure 2.6 est un exemple de programme SC. Il a deux entrées pures `I1` et `I2` (avec deux variables booléennes de présence associées, respectivement `PI1` et `PI2`), et deux sorties valuées `O1` et `O2` (avec deux variables entières associées, respectivement `N1` et `N2`). Le contrôle démarre du registre le plus à gauche (`boot`) et dans les deux fils `input` (donc ces trois fils sont en parallèle), se propage dans les deux portes `and` les plus à gauche (donc les deux fils `N2:=N2+1` et `N2:=N2*N1` sont en parallèle), puis reboucle dans le second registre, après avoir émis les sorties `O1` et `O2`.

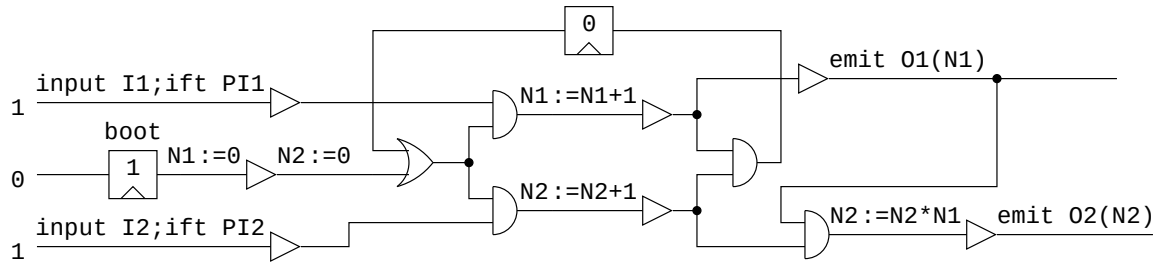


FIG. 2.6 – Un exemple de circuit synchrone : le programme `foo`.

La table d'actions du programme SC `foo` est donnée dans la table 2.2. Elle contient toutes les actions déclenchées par les fils `action`, `ift`, `input` et `output`.

`input I1; ift PI1`
`input I2; ift PI2`
`N1 := 0`

`N2 := N2 + 1`
`N2 := N2 * N1`
`emit O1(N1)`

`N2 := 0`
`emit O2(N2)`
`N1 := N1 + 1`

TAB. 2.2 – Table des actions du programme `foo`.

2.4.2 Algorithme de répartition

2.4.2.1 Principe

Supposons que l'on désire répartir le programme `foo` de la figure 2.6 sur deux sites de calcul `L` et `M` et selon les directives de la table 2.3.

site `L`	site `M`
`I1,O2`	`I2,O1`

TAB. 2.3 – Directives de répartition pour le programme `foo`.

Le principe de la répartition automatique de programmes SC est le suivant :

1. Il faut d'abord attribuer un ensemble de sites de calcul à chaque action du circuit : chaque action sera exécutée par *un seul* site de calcul, sauf les `ift` qui seront exécutées par *tous* les sites. Cela permet d'obtenir le circuit de chaque site, de la manière suivante :
   – La partie donnée est obtenue en enlevant à la table initiale toutes les actions non concernées par le site de calcul courant.
   – La partie contrôle est obtenue à partir de la partie contrôle initiale en remplaçant chaque fil `action` et `output` non concerné par un fil `standard`, et en remplaçant chaque fil `input` par un bloc de simulation de l'entrée (voir les détails dans la section 2.4.2.7). En revanche, les fils `ift` sont répliqués sur la partie contrôle de *tous* les sites.

   À ce stade il n'y a *qu'un seul* circuit. Chaque site de calcul n'a donc qu'un circuit virtuel. Ce n'est qu'à l'étape 4 que seront réellement générés un circuit par site de calcul.
2. À ce stade, le circuit virtuel de chaque site de calcul fait référence à des variables qui ne sont pas calculées localement, et à des entrées qui ne sont pas reçues localement. Puisque l'architecture est à mémoire répartie, le programme de chaque site doit tenir à jour une copie locale de chaque variable et entrée distante (celles appartenant à d'autres sites). Pour cela, des communications sont ajoutées pour résoudre les dépendances aux variables distantes.
3. Chaque porte non purement dépendante des entrées du programme est détectée et ses broches d'entrée sont marquées afin de savoir de quelles entrées elles dépendent. Cette information sera utilisée à l'étape 5 pour résoudre les dépendances aux entrées distantes.
4. Un circuit concret est produit pour chaque site de calcul, en copiant chaque circuit virtuel dans un fichier séparé.
5. Enfin, des blocs de simulation des entrées sont créés et connectés aux fils requis (ceux détectés lors de l'étape 3 ci-dessus) pour résoudre les dépendances aux entrées distantes.

2.4.2.2 Primitives de communication

Dans la mesure où la structure de contrôle des programmes SC est *parallèle*, l'ordre dans lequel seront exécutées des communications situées dans des branches concurrentes ne peut pas être connu statiquement. De plus, comme les fragments du programme répartis sont appelés à être exécutés en vrai parallélisme, un site émetteur peut tout à fait envoyer deux valeurs successives d'une même variable avant que le site récepteur ne fasse les réceptions correspondantes. Cependant, grâce à la propriété de causalité (section 2.4.1), toute variable ne peut être modifiée que dans une seule branche parallèle du circuit de contrôle. Par conséquent, l'ordre des communications, *pour une variable donnée*, peut être déterminé statiquement. Chaque file FIFO est donc identifiée par un triplet $\langle$`src`, `var`, `dst`$\rangle$, où `src` est le site source, `var` est la variable dont les valeurs sont transmises, et `dst` est le site destination. Les primitives sont semblables à celles décrites dans la section 2.2.2 :

– Sur le site `src`, la primitive d'émission `send(dst,var)` envoie la valeur courante de la variable `var` dans la file $\langle$`src`, `var`, `dst`$\rangle$; elle est non-bloquante.
– Sur le site `dst`, la primitive de réception `var:=receive(src,var)` extrait la première valeur de la file $\langle$`src`, `var`, `dst`$\rangle$ et l'affecte à la variable `var` ; elle est bloquante tant que la file est vide.

2.4.2.3 Localisation des actions

Ici aussi je n'essaye pas d'obtenir la *meilleure* répartition possible. Je détermine la localisation de toutes les actions et variables à partir de la localisation des entrées/sorties et des dépendances de données existant entre les actions. Une telle localisation est unique pour un ordre de traitement des entrées et sorties du circuit donné.

Par exemple, si le signal de sortie `O2` doit être calculé sur le site `L`, alors il en va de même de l'action action `emit O2(N2)`. Par suite, la variable `N2` et l'action `N2:=N2*N1` doivent toutes deux être aussi localisées sur le site `L`. Vient alors le tour de la variable `N1`, mais celle-ci est déjà localisée sur le site `M` en raison de la localisation de la sortie `O1`.

La localisation des actions du programme `foo` selon les directives de répartition de la table 2.3 est indiquée dans la table 2.4.

site	action
L	`input I1; ift PI1`
M	`input I2; ift PI2`
M	`N1 := 0`

site	action
L	`N2 := N2 + 1`
L	`N2 := N2 * N1`
M	`emit O1(N1)`

site	action
L	`N2 := 0`
L	`emit O2(N2)`
M	`N1 := N1 + 1`

TAB. 2.4 – Localisation des actions du programme `foo`.

Une fois que chaque action a un site de calcul unique, deux problèmes se posent :

1. le **problème des variables distantes** : certaines variables ne sont pas calculées localement ;
2. le **problème des entrées distantes** : certaines entrées ne sont pas reçues localement.

Le premier problème se résout en ajoutant des fils `action` déclenchant des `send` et `receive` au programme de chaque site de calcul, selon les mêmes principes que l'insertion des émissions et des réceptions dans les programmes OC répartis. Je traite ce problème dans la section 2.4.2.4.

Le second problème ne peut pas être résolu avec la même technique, car un signal d'entrée véhicule *deux* informations (voir la section 2.4.1) : la présence de l'entrée et, dans le cas d'un signal valué, sa valeur. Sa valeur peut certes être traitée comme une variable ordinaire par l'algorithme d'insertion des émissions et des réceptions. Mais sa présence est, elle, directement *codée dans le circuit de contrôle*, puisque le fil `input` propage `1` si l'entrée est présente et `0` sinon. Par conséquent, un fil `input` correspondant à une entrée distante *ne peut pas propager la valeur correcte*. Cela empêche donc les tests portant sur les variables de présence des entrées distantes de retourner le bon résultat. Ce problème est résolu en modifiant les circuits et en ajoutant des **blocs de simulation des entrées**. Je traite ce problème dans la section 2.4.2.7.

2.4.2.4 Résolution du problème des variables distantes

Parcours de la structure de contrôle. Comme je l'ai dit dans la section 2.1.3, la structure de contrôle des programmes SC est *parallèle, implicite et dynamique* :

- **Parallèle** parce qu'à un instant donné le contrôle peut être dans plusieurs branches parallèles. Par conséquent les valeurs peuvent être envoyées d'un site à un autre en concurrence, donc il faut éviter les conflits. Comme les programmes considérés sont causaux, de tels conflits ne peuvent porter que sur des variables *distinctes*.
- **Implicite** parce que l'état interne du programme est codé dans les valeurs mémorisées dans les registres du circuit. Par conséquent il faut démarrer le parcours de la structure de contrôle à *chaque* entrée et registre du circuit.

– **Dynamique** parce que les valeurs mémorisées dans les registres du circuit ne sont pas connues à la compilation. Par conséquent, pour chaque fil de sortie d'une porte `or`, il n'est pas possible de connaître *statiquement* de quelle broche d'entrée de cette porte le contrôle va arriver. Donc il faut travailler *séparément* sur chaque chemin de buffers du circuit. Un **chemin de buffers** est une séquence de fils connectés entre eux par des simples buffers, donc sans la moindre porte booléenne.

Afin de parcourir la structure de contrôle d'un programme SC, je démarre donc à chaque entrée et chaque registre du circuit. Pour chacun de ces points de départ, je parcours la structure de contrôle en avant, tout en marquant les fils visités. Quand j'atteins une porte booléenne, je marque son fil de sortie en tant que **tête** ; puis quand j'atteins la porte booléenne suivante, je marque le dernier fil visité en tant que **queue**, et j'applique l'algorithme d'insertion des `send` et `receive` au chemin de buffers qui part de la tête précédente et qui termine à cette queue. Puis je poursuis le parcours à partir de chaque fil connecté au fil de sortie de cette porte de queue, exceptés ceux qui sont déjà visités.

Insertion des émissions. J'insère donc des émissions dans chaque chemin de buffers du circuit de contrôle. Une émission est un fil `action` déclenchant une action `send`. L'algorithme pour insérer les émissions est dérivé de celui pour les programmes OC [35]. Il consiste en un parcours en arrière du chemin de buffers pour déterminer les variables distantes nécessaires à chaque action du chemin. Le parcours est fait en arrière afin d'insérer les émissions le plus tôt possible dans la structure de contrôle du programme (c'est-à-dire le plus proche possible de la tête de chaque chemin de buffers). De même que pour les programmes OC, le but ici est de rendre le programme réparti ainsi obtenu le moins sensible possible à la latence du réseau de communication. De façon similaire aux instructions dans un langage de programmation impératif, je définis donc deux ensembles `use` et `def` pour chaque fil déclenchant une action (`action`, `ift`, `input` ou `output`) :

– Une action déclenchée par un fil `action` peut être ou bien une affectation ou bien un appel de procédure externe. Dans les deux cas, les ensembles `use` et `def` contiennent respectivement les variables utilisées et modifiées par cette action. Par exemple, les deux ensembles associés à l'action `x:=y*z` sont `use={y,z}` et `def={x}`.
– Pour un fil `ift`, `use` contient les variables utilisées par l'expression testée ; `def` est quant à lui vide.
– Pour un fil `input`, `use` est vide ; `def` contient quant à lui la variable de présence du signal d'entrée, ainsi que la variable associée si le signal est valué.
– Pour un fil `output`, `def` est vide ; `use` est quant à lui vide si le signal de sortie est pur, et contient les variables utilisées par l'expression associée sinon.

Afin d'insérer les émissions, je définis, pour chaque site de calcul `s`, l'ensemble $\texttt{Need}_\texttt{s}$ de toutes les variables distantes dont le site `s` va certainement avoir besoin, sauf si leur valeur a déjà été envoyée par leur site propriétaire respectif. Le calcul des ensembles $\texttt{Need}_\texttt{s}$ permet l'insertion des émissions de telle sorte que tout site qui a besoin d'une variable pour un fil `action` la recevra *avant* ce fil. Pour chaque chemin de buffers et chaque site `s`, l'algorithme consiste à placer un ensemble $\texttt{Need}_\texttt{s}$ vide à sa queue, et à le propager en arrière de la manière suivante :

– Quand j'atteins un fil `f` de type `action`, appartenant au site `s`, pour chaque $\texttt{x} \in \texttt{use}$, si `x` est une variable *distante* de `s`, alors j'ajoute `x` à $\texttt{Need}_\texttt{s}$. De plus, pour chaque $\texttt{y} \in \texttt{def}$ et pour chaque site `s` tel que $\texttt{y} \in \texttt{Need}_\texttt{s}$, j'insère un fil déclenchant l'action `send(s,y)` juste après le fil `f`. Enfin j'enlève `y` de chaque ensemble $\texttt{Need}_\texttt{s}$ concerné.
– Quand j'atteins la tête du chemin de buffers, pour chaque site `s`, j'insère juste après la tête un fil déclenchant l'action `send(s,x)` pour chaque variable `x` de l'ensemble $\texttt{Need}_\texttt{s}$. Le parcours est alors terminé.

Insertion des réceptions. Pour insérer les réceptions (en réalité des fils `action` déclenchant une action `receive`), je simule à la compilation le contenu des files FIFO. Puisque chaque file correspond à une variable, je n'ai besoin que de comptabiliser *le nombre de valeurs* présentes dans la file à un instant donné (c'est-à-dire à un point donné du circuit de contrôle). Donc je définis pour chaque file $\langle \mathtt{t}, \mathtt{x}, \mathtt{s} \rangle$ un entier $\mathtt{Queue}^{\mathtt{x}}_{\mathtt{t}\triangleright\mathtt{s}}$ stockant le nombre de valeurs de `x` qui ont été envoyées par le site `t` et qui n'ont pas encore été reçues par le site `s`.

L'algorithme consiste à initialiser chaque entier $\mathtt{Queue}^{\mathtt{x}}_{\mathtt{t}\triangleright\mathtt{s}}$ à zéro, et à les propager en avant de la tête jusqu'à la queue de chaque chemin de buffers, de la manière suivante :

- Quand j'atteins un fil `action` déclenchant un `send(s,x)` sur le site `t`, j'incrémente l'entier $\mathtt{Queue}^{\mathtt{x}}_{\mathtt{t}\triangleright\mathtt{s}}$.
- Quand j'atteins un fil déclenchant une action localisée sur le site `s`, alors pour chaque $\mathtt{x} \in \mathtt{use}$, si `x` est une variable *distante* pour `s`, je teste l'entier $\mathtt{Queue}^{\mathtt{x}}_{\mathtt{t}\triangleright\mathtt{s}}$. S'il est strictement positif, alors je le décrémente et j'insère le fil `action` déclenchant la réception `x:=receive(t,x)` sur le site `s`. S'il est nul, alors je ne fais rien car cela signifie que la valeur de `x` est déjà connue par le site `s`.
- Quand j'atteins la queue du chemin de buffers, le parcours est terminé puisque chaque entier $\mathtt{Queue}^{\mathtt{x}}_{\mathtt{t}\triangleright\mathtt{s}}$ est par construction nul. La raison en est que chaque variable envoyée par un `send` est attendue puisque le site destinataire en a besoin. La preuve formelle peut en être établie grâce aux mêmes techniques que pour les programmes OC (section 2.3.4).

Le résultat de ces deux algorithmes sur le programme `foo` est illustré dans la figure 2.10.

2.4.2.5 Identification des portes non purement dépendantes des entrées

Le but de cette étape est de distinguer les fils **purement dépendant** des entrées, qui ne dépendent que de booléens de présence, et les fils **non purement dépendant** des entrées, qui dépendent également du flot de contrôle issu des fils `input`.

Cette opération consiste à décorer chaque fil avec un ensemble `SInput` d'entrées dont la présence est requise pour calculer la valeur de sortie de ce fil. Initialement, ces ensembles sont vides. À partir de chaque fil `input`, je parcours partiellement en avant le circuit, tout en marquant les fils visités et en propageant les ensembles `SInput` de la manière suivante :

- Au départ, le fil `input` de l'entrée `s` propage $\{\mathtt{s}\}$ à toutes ses broches de sorties.
- Un fil avec une seule broche d'entrée propage l'ensemble $\mathtt{SInput}^{\mathtt{in}}$ qui lui arrive par cette broche à toutes ses broches de sortie, et remet à vide son propre ensemble puisque c'est un fil purement dépendant des entrées : $\mathtt{SInput} := \emptyset$.
- Si un fil possède plusieurs broches d'entrée, alors :
  - Pour chaque ensemble $\mathtt{SInput}^{\mathtt{in}}$ qui lui arrive par une broche d'entrée, je calcule $\mathtt{SInput} := \mathtt{SInput} \cup \mathtt{SInput}^{\mathtt{in}}$;
  - tant que ses broches d'entrées ne sont pas toutes visitées, il ne propage rien ; si elles le sont, alors c'est un fil purement dépendant des entrées : il propage son ensemble courant `SInput` à toutes ses broches de sorties, et il remet à vide son propre ensemble : $\mathtt{SInput} := \emptyset$.

À l'issue de ce parcours partiel, l'ensemble `SInput` d'un fil donné est non-vide si et seulement si ce fil est non purement dépendant des entrées, et dans ce cas l'ensemble `SInput` contient les entrées du circuit qui sont nécessaires au calcul de sa valeur. Au contraire, un fil purement dépendant des entrées est identifié par un ensemble `SInput` vide et est marqué comme étant visité. Le résultat de cette identification sur le programme `foo` de la figure 2.6 est illustré sur la figure 2.7.

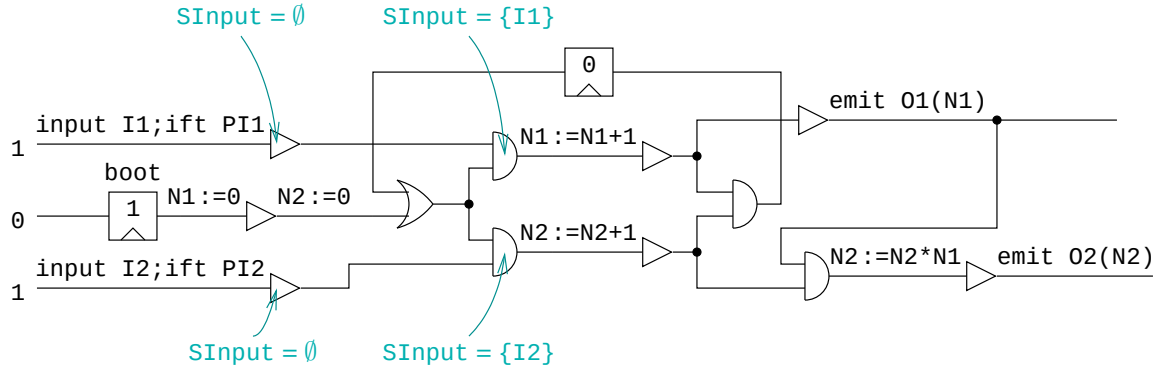


FIG. 2.7 – Le programme `foo` dans lequel les portes purement dépendantes et non purement dépendantes des entrées ont été détectées.

2.4.2.6 Projection vers n programmes SC

À ce stade, il n'y a toujours qu'un seul programme SC, dont le circuit de contrôle contient tous les fils `action` localisés sur les différents sites de calcul. Je projette donc ce programme sur chaque site de calcul de manière à obtenir n programmes SC, un par site : pour chaque site, tout fil `action` ou `output` n'appartenant pas à ce site est transformé en un fil `standard`; tous les autres fils sont conservés tels quels.

2.4.2.7 Résolution du problème des entrées distantes

Principe. Pour résoudre ce problème, une première solution serait que chaque site de calcul envoie, à chaque top d'horloge, le booléen de présence de *chaque* signal d'entrée qui lui appartient à chacun des autres sites de calcul. Ainsi, tous les fils `input` propageraient toujours la valeur correcte.

Cependant, dans le cas général, un programme SC a un très grand nombre d'entrées, donc cette solution est trop coûteuse. Aussi mon but est-il de n'envoyer le booléen de présence qu'aux sites qui en ont *besoin*. Pour cela, la solution proposée modifie la structure de contrôle des programmes répartis. Elle comporte trois étapes :

1. détection des fils non purement dépendants des entrées, ainsi que des entrées dont ils ont besoin ;
2. création des blocs de simulation des entrées ;
3. connexion des fils non purement dépendant précédemment détectés aux blocs de simulation requis.

La première étape a été décrite dans la section 2.4.2.5. Les deux autres sont décrites dans les deux paragraphes qui suivent.

Création des blocs de simulation des entrées. La deuxième étape consiste à transformer chaque fil `input` de la manière indiquée par la figure 2.8. Les portes 1 et 2 verront leurs broches d'entrée connectées durant la troisième étape : elles seront connectées aux portes qui ont réellement besoin de la présence des entrées correspondantes (voir ci-dessous).

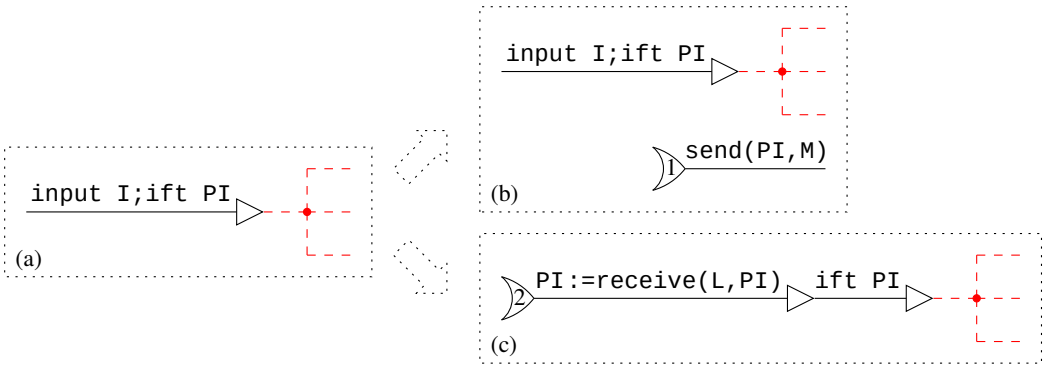


FIG. 2.8 – (a) Transformation d'un fil `input I` en : (b) un fil `input` couplé à un fil `send` sur un site `L` possédant l'entrée `I` ; (c) et un bloc de simulation de l'entrée `I` sur le site `M` ne possédant pas l'entrée `I`.

Connexion des blocs de simulation des entrées. La troisième étape consiste à connecter chaque fil non purement dépendant des entrées aux blocs de simulation requis. Si l'expression d'entrée d'un tel fil est un `and`, alors un nouveau `and` est créé de la façon suivante : ses broches d'entrée sont connectées aux broches d'entrées du premier `and`, exceptées celles qui sont connectées à des fils purement dépendant des entrées (identifiés par leur ensemble `SInput` vide) ; quant à sa broche de sortie, elle est connectée à l'entrée de chaque bloc de simulation des entrées qui sont dans son ensemble `SInput`. Dans le cas d'une expression d'entrée `or`, c'est une porte `and` qui est créée : ses broches d'entrée et sa broche de sortie sont connectées de la même façon que pour un `and`, sauf que ses broches d'entrées sont inversées. La figure 2.9 illustre le résultat de ces connexions dans le cas d'un `and` (figure 2.9(a)) et d'un `or` (figure 2.9(b)).

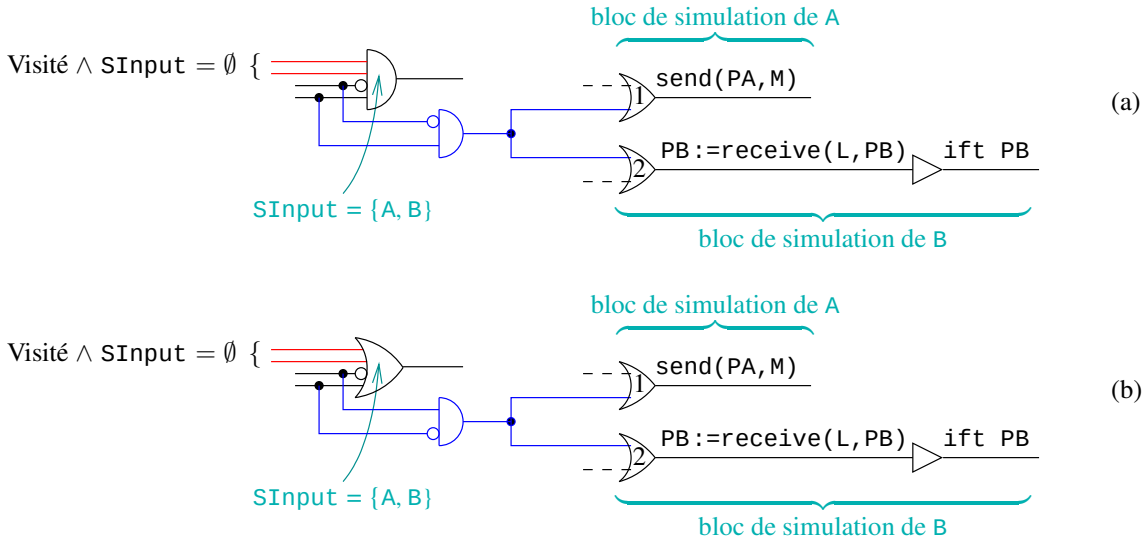


FIG. 2.9 – (a) Connexion d'un fil `and` non purement dépendant des entrées aux blocs de simulation requis ; (b) même chose pour un fil `or` ; dans les deux cas l'entrée `A` appartient au site courant alors que l'entrée `B` ne lui appartient pas.

Ces connexions sont établies de telle sorte que la simulation d'une entrée, ainsi que l'émission et la réception du booléen de présence qui en découle, n'est effectuée que quand sa valeur est réellement requise, évitant ainsi des communications inutiles entre les différents sites de calcul. La figure 2.10 montre le programme réparti final obtenu pour le programme SC `foo` et les directives de répartition

de la table 2.3. Le fil `N2:=N2+1` est non-purement dépendant des entrées puisque le contrôle arrive par deux broches, l'une venant du registre `boot` et l'autre de l'entrée `I2`. Donc le circuit du site `M`, propriétaire de l'entrée `I2` envoie le booléen de présence `PI2` précisément au moment où le contrôle arrive du registre `boot`, et le circuit du site `L` simule l'entrée `I2` en recevant ce booléen `PI2` et en codant sa valeur dans le circuit au moyen du fil `ift PI2`.

2.4.2.8 Résultat final

La table 2.5 montre les actions du programme réparti `FOO` sur les sites `L` et `M`, tandis que la figure 2.10 montre le programme réparti `FOO`.

site `L`
`input I1; ift PI1`
`ift PI2`
`N2 := 0`
`N2 := N2 + 1`
`N2 := N2 * N1`
`emit O2(N2)`
`send(M,PI1)`
`PI2 := receive(M,PI2)`
`N1 := receive(M,N1)`

site `M`
`input I2; ift PI2`
`ift PI1`
`N1 := 0`
`N1 := N1 + 1`
`emit O1(N1)`
`PI1 := receive(L,PI1)`
`send(L,PI2)`
`send(L,N1)`

TAB. 2.5 – Tables des actions du programme `foo` sur les sites `L` et `M`.

C'est bien un programme GALS puisque chaque fragment local est un circuit synchrone, et que ces circuits communiquent de façon asynchrone.

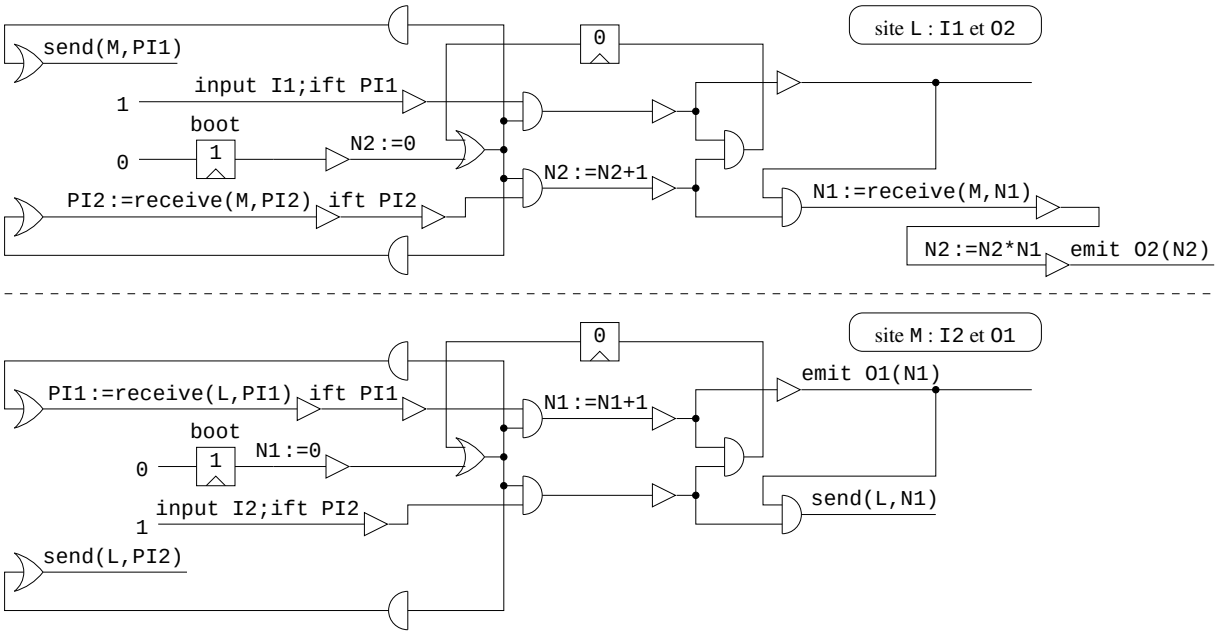


FIG. 2.10 – Le programme SC `foo` réparti sur les deux sites `L` et `M`.

Notons que certaines des portes qui ont été ajoutées au moment de connecter les blocs de simulation des entrées peuvent être remplacées par de simples buffers puisque ce sont des portes `and` et `or` avec une unique broche d'entrée.

Enfin, dans la mesure ou cette méthode ne répartit que la partie données des programmes SC (la structure de contrôle étant répliquée sur tous les sites de calcul), on ne doit s'attendre à des gains de performance par rapport au programme centralisé que si celui-ci a une grosse partie données, c'est-à-dire s'il n'est pas trop orienté contrôle.

2.4.3 Implémentation et discussion

Cet algorithme a été implémenté par Clément Ménier dans l'outil `screp` [53], disponible sur le web.[6] Il fait environ 15.000 lignes de code C++.

Les travaux les plus proches dans la littérature se trouvent dans le domaine de la production de systèmes GALS à partir de circuits. Ce sont ceux de Berry et Sentovich [19] : ils mettent en œuvre des circuits synchrones constructifs sous la forme de réseaux de CFMSs (« Codesign Finite State Machine ») communicant dans POLIS [7], qui sont par définition des systèmes GALS. Il y a plusieurs différences avec la méthode que je viens de présenter :

- Berry et Sentovich prennent comme point de départ des circuits synchrones **cycliques**, avec la restriction que ces circuits doivent être **constructifs** [93]. Un circuit constructif est un circuit cyclique avec un « bon comportement », c'est-à-dire tel qu'il existe un circuit **acyclique** calculant les mêmes sorties à partir des mêmes entrées. Cependant, les circuits qu'ils considèrent ne manipulent que des booléens. Par comparaison, je me limite à des circuits synchrones **acycliques** (résultant de la compilation de programmes ESTEREL ou autres), mais manipulant des signaux **valués** tels que des entiers, des réels... (voir la section 2.4.1).
- Leurs CFSMs communiquent entre elles au moyen de buffers **à 1 place et non bloquants**, tandis que j'utilise des files d'attente FIFO **bloquantes et à n places**.
- Leur méthode pour obtenir des systèmes GALS consiste à partitionner l'ensemble des portes logiques en **groupes de portes** (« clusters » en anglais), puis à implémenter chaque groupe sous la forme d'une CFSM, et enfin à connecter ces groupes dans un réseau POLIS. Cela offre donc la possibilité de choisir entre plusieurs granularités, depuis une porte par groupe jusqu'à un seul groupe pour le circuit tout entier. En revanche, ils ne fournissent aucune méthode permettant de réaliser cette partition en groupes de portes.
- Les CFSMs communiquent entre elles de façon à mettre en œvre la **sémantique constructive** d'ESTEREL (c'est obligatoire puisque les circuits considérés peuvent être acycliques). Cela signifie que chaque CFSM envoie aux autres CFMSs les **faits** relatifs à la stabilisation de ses propres portes, de telle sorte que les autres CFSMs puissent réagir à ces faits (en stabilisant à leur tour certaines de leurs portes). Le principe est que le réseau de CFSMs dans son ensemble se comporte exactement comme le circuit source centralisé. Au contraire, dans mon programme réparti final, le circuit de chaque site de calcul envoie aux autres des **valeurs** et la cohérence globale est assurée parce que chaque circuit implémente la structure de contrôle entière (voir la section 2.4.2). En définitive, dans mon approche les programmes de chaque site sont plus gros mais il y a beaucoup moins de communication entre les sites. Dans un système réparti où les communication sont souvent plus coûteuses que les calcul, cela peut représenter un avantage certain.

D'autres travaux ont également été effectués sur la répartition de programmes ESTEREL, en connexion avec le logiciel SYNDEX. Ils seront présentés dans la section 2.7.6.

[6] `screp` : `http://pop-art.inrialpes.fr/~girault/Screp`

2.5 Répartition automatique de programmes CP

En 2002, France Telecom R&D m'a sollicité pour réaliser la répartition automatique de programmes CP, afin de proposer une offre de génération de code réparti dans leur compilateur SAXO-RT. Ce compilateur ESTEREL produit du code C particulièrement adapté aux systèmes embarqués [99] ; il était commercialisé à l'époque par la société ESTEREL TECHNOLOGIES. Ces travaux ont été effectués en collaboration avec Étienne Closse (à l'époque chez France Telecom R&D, aujourd'hui chez ATHYS) et Fabien Giraud (stagiaire ISIMA, Clermont-Ferrand).

2.5.1 Modèle des programmes CP

La figure 2.11 illustre la structure de contrôle d'un programme CP composé de p **points de contrôle**. Chaque point de contrôle est lui-même composé d'un booléen $\texttt{exe}_i$, d'un bloc de code séquentiel « tâche i » et d'un second booléen $\texttt{pause}_i$. À chaque cycle du programme, les booléens $\texttt{exe}_i$ sont testés en séquence et pour tous ceux qui valent `true`, la tâche i est exécutée. Chaque tâche i peut modifier la valeur des booléens $\texttt{exe}_{i+1}$ à $\texttt{exe}_p$ (donc uniquement des points de contrôle qui lui sont postérieurs dans le même cycle) ainsi que la valeur de tous les booléens $\texttt{pause}_j$ (de tous les points de contrôle du cycle suivant). Ainsi la causalité est respectée. À la fin du cycle, le vecteur $(\texttt{pause})_{1,p}$ est recopié dans le vecteur $(\texttt{exe})_{1,p}$.

Sur la figure 2.11, les flèches bleues correspondent aux actions de contrôle (passage du point i au point $i+1$ et recopie du vecteur $(\texttt{pause})_{1,p}$ dans le vecteur $(\texttt{exe})_{1,p}$), les flèches vertes à la mise à `true` d'un booléen $\texttt{exe}_i$ ou $\texttt{pause}_i$, et enfin les flèches rouges à la mise à `false` d'un de ces booléens.

Enfin, le code des tâches ne peut contenir comme instruction de contrôle que des `if...then...else...`, donc pas de boucle. C'est du code purement séquentiel comme celui qu'on trouve dans les états d'un automate OC.

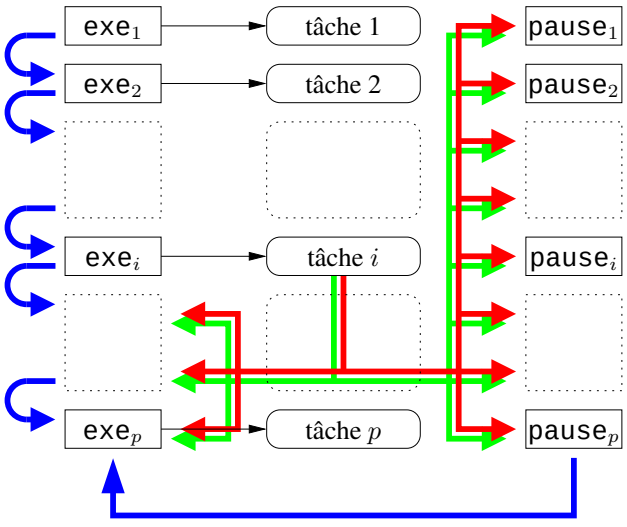


FIG. 2.11 – Structure de contrôle d'un programme CP.

2.5.2 Algorithme de répartition

Le travail demandé par France Telecom R&D était de mettre au point un algorithme de répartition pour ce type de programmes. De même que dans la section 2.3.3, l'utilisateur fournit un fichier de directives de répartition, consistant en une partition de l'ensemble des entrées/sorties du programmes en

n sous-ensembles, un pour chaque site de calcul qu'il souhaite. La solution retenue comporte les étapes suivantes :

1. Répliquer la structure de contrôle de la figure 2.11 sur chacun des n sites de calcul spécifiés dans les directives de répartition fournies par l'utilisateur.
2. Pour chaque point de contrôle i, appliquer l'algorithme d'`ocrep` (section 2.3.3) au code séquentiel de la tâche i. Cette étape requerrait ou bien de modifier la structure de données interne au compilateur SAXO-RT afin de l'adapter aux fonctions utilisées par `ocrep`, ou bien de modifier les fonctions utilisées par `ocrep` afin de les adapter à la structure de données de SAXO-RT. Dans un premier temps, des traducteurs ont été écrits entre les deux structures de données, celle de SAXO-RT et celle d'`ocrep` [48]. Mais il se trouve que la structure de données utilisée par les fonctions d'`ocrep` permettait des manipulations plus aisées, et les ingénieurs de France Telecom R&D ont donc opté, dans un second temps, pour la première solution.

2.5.3 Implémentation et discussion

Cet algorithme est implémenté dans le compilateur SAXO-RT de France Telecom R&D. Il a été testé sur l'intégralité de la base de test ESTEREL de France Telecom R&D, soit 125 programmes ESTEREL. Ce travail a fait l'objet d'un contrat de transfert technologique entre l'INRIA Rhône-Alpes et la société France Telecom R&D, concernant les algorithmes d'insertion des `send` et des `receive` implémentés dans `ocrep`.

L'intérêt pratique de ce travail est de permettre la répartition automatique de gros programmes ESTEREL. En effet, pour les gros programmes ESTEREL, la compilation en OC produit un automate en général trop gros pour pouvoir être manipulé ; quant à sa compilation en SC, elle produit un circuit trop lent pour être utilisable. Au contraire, sa compilation en CP produit du code à la fois petit et rapide.

2.6 Exécution de programmes synchrones répartis

Une fois que le programme synchrone est réparti, le problème de son exécution sur une architecture répartie se pose. J'ai déjà proposé des solutions à ce problème dans ma thèse [49, 34], aussi je mentionne ces résultats très rapidement dans la section 2.6.1 ci-dessous. Puis, je présente dans la section 2.6.2 une solution alternative permettant l'exécution de programmes synchrones répartis dans l'environnement de conception de logiciels hétérogènes et complexes PTOLEMY [23].

2.6.1 Interfaces synchrone/asynchrone réparties

La solution directe consiste en un **module d'interface** entre l'environnement asynchrone et le programme synchrone pour chaque site de calcul [49, 34], chaque module d'interface étant chargé de produire pour son programme une séquence de vecteurs de valeurs au moyen d'une boucle d'exécution comme celle décrite en section 2.1.4. De plus, des communications asynchrones entre chaque fragment du programme réparti et les modules d'interface des autres fragments permettent la réaction à l'absence du programme réparti.

La figure 2.12 illustre cette solution : soit un programme P réparti en deux fragments P1 et P2, chacun ayant son propre module d'interface G1 et G2 ; l'ensemble des entrées du programmes, {A, B, C, D}, est partitionné par l'utilisateur en deux sous-ensembles : {A,B} sur le site 1 et {C,D} sur le site 2 ; la communication asynchrone entre G1 et G2 permet à G2 de produire un vecteur $(\varepsilon, \varepsilon)$ sur le site 2 correspondant au vecteur (a_1, b_1) sur le site 1. Ainsi le programme P2 sait que les entrées C et D sont absentes lors de cette réaction. À chaque réaction du programme réparti, la concaténation des vecteurs produits par G1 et G2 est identique au vecteur produit par le module d'interface du programme centralisé.

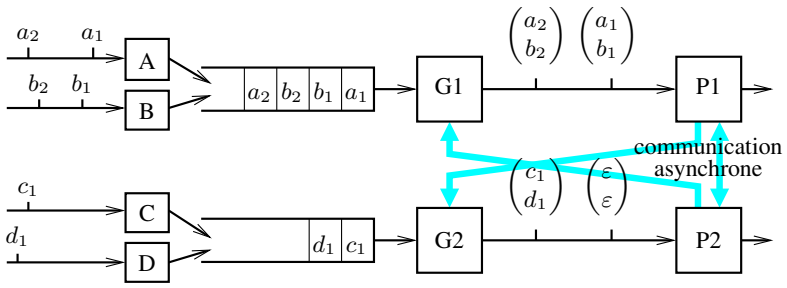


FIG. 2.12 – Interface répartie avec communication asynchrone

La figure 2.13 illustre le comportement du module d'interface centralisé avec les mêmes séquences d'entrées qu'à la figure 2.12.

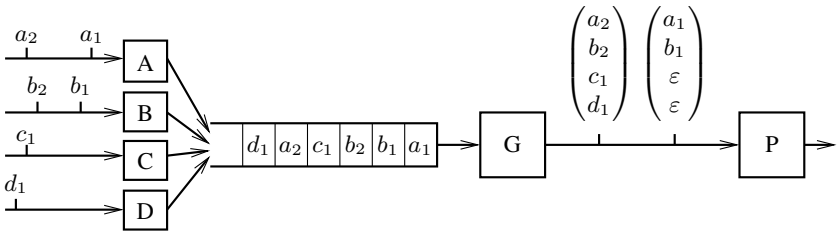


FIG. 2.13 – Interface centralisée.

Toutefois, cette solution ne peut pas garantir formellement l'équivalence entre le comportement d'un module d'interface centralisé (c'est-à-dire la séquence des vecteurs qu'il produit) et celui d'un ensemble de modules d'interfaces répartis (c'est-à-dire la concaténation des vecteurs produits par chacun d'eux), pour une même séquence d'événements d'entrée. La raison en est l'*indéterminisme* existant entre les périodes d'exécution des modules d'interfaces, à moins de mettre en place un mécanisme très coûteux de synchronisation d'horloge.

2.6.2 Exécution des programmes répartis dans PTOLEMY

PTOLEMY est un environnement de conception de logiciels hétérogènes et complexes [23].[7] Le modèle sous-jacent est celui des réseaux de processus flot-de-données, appelés **réseaux de Kahn** [73]. Les données qui circulent dans le réseau sont appelés **tokens** et les processus sont appelés **étoiles**. En outre, PTOLEMY permet de faire interagir plusieurs **modèles d'exécution**, grâce aux principes de polymorphisme et d'encapsulation. Chaque modèle d'exécution est mis en œuvre dans un **domaine** et un ensemble d'étoiles interconnectées dans un domaine est appelé une **galaxie**. Chaque domaine est défini par la façon dont les processus flot-de-données consomment les tokens qui arrivent en entrée et produisent les tokens en sortie. Ces règles de consommation et de production sont appelées les **règles d'activation**.

Au début, les modèles d'exécution supportés par PTOLEMY étaient tous des modèles flots-de-données : c'étaient, du plus synchrone au plus asynchrone, Synchronous Data-Flow (SDF), Boolean Data-Flow (BDF), Dynamic Data-Flow (DDF), et Process Network (PN). Par la suite, de nombreux autres modèles d'exécution ont été ajoutés, pas forcément flots-de-données : par exemple Finite State Machine (FSM), Synchronous Reactive (SR)...

[7] PTOLEMY : `http://ptolemy.eecs.berkeley.edu`. Il permet d'utiliser et de mélanger plusieurs modèles d'exécution de processus, qui prenent en compte les aspects de communication et d'ordonnancement des calculs. La version actuelle est PTOLEMY II ; la version sur laquelle j'ai travaillé en 1996 était PTOLEMY CLASSIC.

Lors de mon séjour post-doctoral à UC Berkeley, dans l'équipe d'Edward Lee, j'ai mis au point une méthode permettant d'exécuter des programmes OC répartis dans PTOLEMY. L'intérêt était d'offrir à l'utilisateur les vastes bibliothèques d'étoiles pour générer les entrées et exploiter les sorties d'un programme, avec les grandes facilités offertes par PTOLEMY pour la simulation et le prototypage. Une telle intégration soulevait deux problèmes :

- **Les interactions entre programmes répartis.** Il a fallu écrire la sémantique des programmes OC répartis, en prenant en compte les diverses possibilités de resynchronisation existantes. En effet `ocrep` force la resynchronisation des programmes répartis, afin de préserver la sémantique temporelle du programme centralisé initial, tout en laissant le choix entre plusieurs possibilités à l'utilisateur (resynchronisation forte, faible ou si besoin).
- **L'immersion des programmes répartis dans PTOLEMY.** Il a fallu considérer chaque programme réparti comme un processus flot-de-données, et définir ses règles d'activation.

Ces deux problèmes ont abouti au choix du modèle d'exécution mis en œuvre dans le domaine PN, celui dont la sémantique est la plus proche des réseaux de Kahn. Ce modèle d'exécution est en effet le seul à n'imposer aucune contrainte de synchronisation cachée aux processus du réseau flot-de-données, tout en assurant la sémantique FIFO des canaux de communication.

J'ai réalisé un compilateur qui génère une étoile PTOLEMY à partir de chaque fragment d'un programme OC réparti. Chaque étoile a un port d'entrée (resp. de sortie) pour tout signal d'entrée (resp. de sortie) du programme OC, plus un port d'entrée (resp. de sortie) pour toute file FIFO provenant de (resp. dirigée vers) tout autre programme OC réparti. Ce compilateur s'appelle `ocpn`.[8]

La figure 2.14 illustre un exemple de programme réparti sur deux sites dans une galaxie PTOLEMY du domaine PN. Les deux étoiles `pingpong_2_0` et `pingpong_2_1` sont les deux fragments OC répartis ; `TkEntry2` est une étoile qui ouvre une fenêtre `Tcl/Tk` permettant de choisir la valeur d'une entrée et son horloge ; `When` est une étoile qui implémente la sémantique de l'opérateur LUSTRE `when` (opérateur de sous-échantillonnage sur une horloge logique plus lente) ; enfin `TkShowValues` est une étoile qui permet de visualiser une valeur dans une fenêtre `Tcl/Tk`.

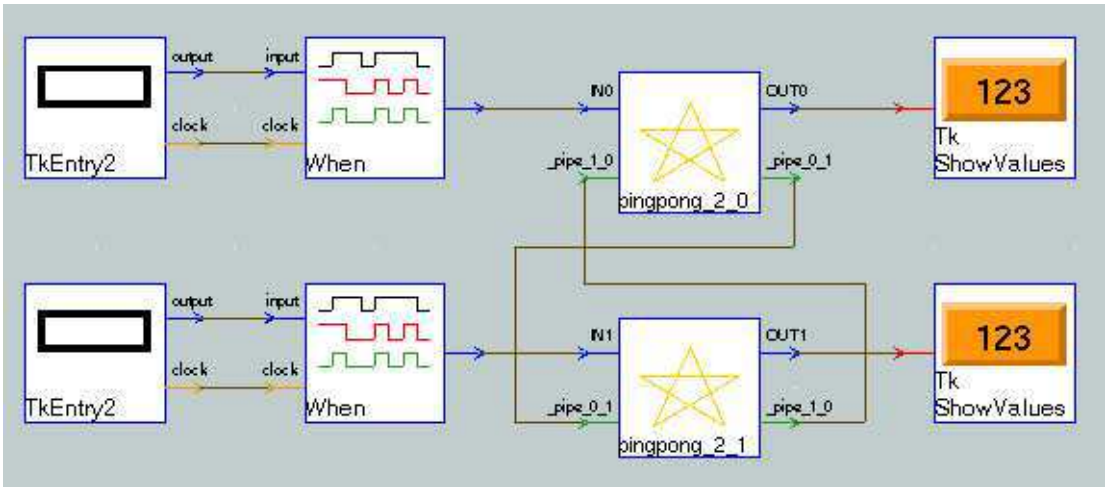


FIG. 2.14 – Le programme `pingpong` réparti sur deux sites et immergé dans une galaxie PTOLEMY.

2.6.3 Relation sémantique entre programmes centralisés et répartis

Dans cette section, j'utilise le « tagged signal model » de Lee et Sangiovanni-Vincentelli [80] afin de modéliser la relation sémantique entre un programme centralisé et un programme réparti communicant par FIFO. Je m'inspire pour cela de deux articles récents de Benveniste et al. [9, 8]. Formellement, une

[8] `ocpn` : `http://pop-art.inrialpes.fr/~girault/Ocpn`

structure de tags est un triplet $\langle \mathcal{T}, \leq, \sqsubseteq \rangle$, où $\mathcal{T}$ est un ensemble de tags et $\leq$ et $\sqsubseteq$ sont deux relations d'ordre partiel telles que $\forall \tau, \tau' \in \mathcal{T} : \tau \sqsubseteq \tau' \Rightarrow \tau \leq \tau'$ et $\sqsubseteq$ induit une borne supérieure partielle notée $\sqcup$. Ici, $\leq$ est l'**ordre sur les étiquettes temporelles** alors que $\sqsubseteq$ est l'**ordre d'unification des tags**. On note alors $\tau_1 \bowtie \tau_2$ si τ_1 et τ_2 possèdent une plus petite borne supérieure pour $\sqsubseteq$, laquelle est notée $\tau_1 \sqcup \tau_2$; autrement dit, si $\tau_1 \bowtie \tau_2$ est une paire unifiable de tags, alors $\tau_1 \sqsubseteq \tau_1 \sqcup \tau_2 \sqsupseteq \tau_2$ et de plus $\forall \tau' \in \mathcal{T} : \tau_1 \sqsubseteq \tau' \wedge \tau_2 \sqsubseteq \tau' \Rightarrow \tau_1 \sqcup \tau_2 \sqsubseteq \tau'$. Par exemple, la structure de tags permettant de modéliser le **synchronisme** est $\mathcal{T}_{\text{synch}} = \langle \mathbb{N}, \leq, = \rangle$ où $\leq$ est l'ordre total usuel des entiers et $=$ est l'égalité. Autre exemple, la structure de tags permettant de modéliser l'**asynchronisme** est $\mathcal{T}_{\text{triv}} = \langle \{.\}, \leq, \leq \rangle$ où $\leq$ est l'unique relation d'ordre sur le singleton $\{.\}$.

Ensuite, un **système taggé** est un triplet $\langle V, \mathcal{T}, \Sigma \rangle$ où V est un ensemble fini de variables, $\mathcal{T}$ est une structure de tags et Σ est un ensemble de **comportements** sur V, c'est-à-dire un ensemble de fonctions $\sigma \in 2^{V \times \mathbb{N} \times (\mathcal{T} \times D)}$, D étant le domaine des valeurs des variables de V. Un comportement σ est donc un ensemble d'**événements** (v, n, τ, x), et la n-ième occurrence de v dans le comportement σ est un couple $\langle$tag, valeur$\rangle$. La fonction $\sigma(v) : \mathbb{N} \mapsto (\mathcal{T} \times D)$ est un **signal**. La première projection de $\sigma(v)$ est une fonction croissante de $(\mathbb{N}, \leq)$ dans $(\mathcal{T}, \leq)$: on l'appelle l'**horloge** de v dans σ.

Quand deux systèmes taggés $P_1 = \langle V_1, \mathcal{T}, \Sigma_1 \rangle$ et $P_2 = \langle V_2, \mathcal{T}, \Sigma_2 \rangle$ ont la même structure de tags $\mathcal{T}$, leur **composition parallèle homogène** est $P_1 \| P_2 = \langle V_1 \cup V_2, \mathcal{T}, \Sigma_1 \wedge \Sigma_2 \rangle$, où $\wedge$ est la **conjonction** des comportements, opération qui consiste à unifier les tags (par l'opération $\sqcup$) des comportements de P_1 et P_2 partageant la même variable : $\Sigma_1 \wedge \Sigma_2 = \{\sigma_1 \sqcup \sigma_2 \,|\, \sigma_1 \in \Sigma_1, \sigma_2 \in \Sigma_2, \sigma_1 \bowtie \sigma_2\}$. L'opération $\|$ est associative et commutative. Quand la structure de tags de P_1 et P_2 est $\mathcal{T}_{\text{triv}}$, on obtient leur produit asynchrone, alors qu'avec $\mathcal{T}_{\text{synch}}$ on obtient leur produit synchrone.

Ensuite, les morphismes de structures de tags permettent de modéliser les désynchronisations : un **morphisme** ρ de $\langle \mathcal{T}, \leq, \sqsubseteq \rangle$ vers $\langle \mathcal{T}', \leq', \sqsubseteq' \rangle$ est une fonction totale surjective, croissante pour les deux ordres $\leq$ et $\sqsubseteq$. Par exemple, le morphisme $\delta : \mathcal{T}_{\text{synch}} \mapsto \mathcal{T}_{\text{triv}}$ projette tous les tags vers la même valeur, il modélise donc la désynchronisation ! De plus, les morphismes introduisent une relation de préordre $\preceq$ entre les structures de tags : $\mathcal{T} \preceq \mathcal{T}'$ s'il existe un morphisme $\rho : \mathcal{T} \mapsto \mathcal{T}'$. Par exemple, pour toute structure de tags $\mathcal{T}$, on a $\mathcal{T}_{\text{triv}} \preceq \mathcal{T}$. Puis, pour deux morphismes ρ_1 et ρ_2 tels que $\mathcal{T}_1 \xrightarrow{\rho_1} \mathcal{T} \xleftarrow{\rho_2} \mathcal{T}_2$, Benveniste et al. définissent leur **produit fibré** : $\mathcal{T}_1 \,{}_{\rho_1}\!\times_{\rho_2} \mathcal{T}_2 = \langle \{(\tau_1, \tau_2) \in \mathcal{T}_1 \times \mathcal{T}_2 \,|\, \rho_1(\tau_1) = \rho_2(\tau_2)\}, \leq_1 \times \leq_2, \sqsubseteq_1 \times \sqsubseteq_2 \rangle$.

Quand deux systèmes taggés $P_1 = \langle V_1, \mathcal{T}_1, \Sigma_1 \rangle$ et $P_2 = \langle V_2, \mathcal{T}_2, \Sigma_2 \rangle$ ont des structures de tags différentes $\mathcal{T}_1$ et $\mathcal{T}_2$ telles qu'il existe deux morphismes ρ_1 et ρ_2 tels que $\mathcal{T}_1 \xrightarrow{\rho_1} \mathcal{T} \xleftarrow{\rho_2} \mathcal{T}_2$, leur **composition parallèle hétérogène** est $P_1 \,{}_{\rho_1}\|_{\rho_2}\, P2 = \langle V_1 \cup V_2, \mathcal{T}_1 \,{}_{\rho_1}\!\times_{\rho_2} \mathcal{T}_2, \Sigma_1 \,{}_{\rho_1}\!\wedge_{\rho_2} \Sigma_2 \rangle$, où ${}_{\rho_1}\!\wedge_{\rho_2}$ est la **conjonction hétérogène** des comportements, opération qui consiste à unifier les comportements de P_1 et P_2 partageant la même variable : $\Sigma_1 \,{}_{\rho_1}\!\wedge_{\rho_2} \Sigma_2 = \{\sigma_1 \,{}_{\rho_1}\!\sqcup_{\rho_2} \sigma_2 \,|\, \sigma_1 \in \Sigma_1, \sigma_2 \in \Sigma_2, \sigma_1 \,{}_{\rho_1}\!\bowtie_{\rho_2} \sigma_2\}$.

Maintenant que je dispose de toute cette formalisation, je peux étudier le déploiement, sur une architecture concrète, d'un programme obtenu par un algorithme de répartition automatique. Soit P le programme centralisé initial et P_1 et P_2 les deux fragments obtenus après répartition sur deux sites de calcul. J'interprète maintenant P comme le système taggé $\langle V, \mathcal{T}_{\text{synch}}, \Sigma \rangle$, avec $\mathcal{T}_{\text{synch}}$ comme structure de tags puisque P est synchrone. De même, P_1 et P_2 sont deux systèmes synchrones, respectivement $\langle V_1, \mathcal{T}_{\text{synch}}, \Sigma_1 \rangle$ et $\langle V_2, \mathcal{T}_{\text{synch}}, \Sigma_2 \rangle$. P_1 et P_2 communiquent via deux FIFOs. Soit $W_i \subseteq V_i$ le sous-ensemble des variables qui sont envoyées par P_i, avec $i = 1, 2$. Puisque P_1 et P_2 ne s'envoient que des valeurs, je définis un système $FIFO_{1,2} = \langle \{w_{1\triangleright 2}, w_{2\triangleright 1}\}, \mathcal{T}_{\text{synch}}, \Sigma \rangle$, où $w_{i\triangleright j}$ est une variable à valeurs dans $2^{D(W_i)}$: ses valeurs sont des suites de valeurs des variables pour lesquelles P_i fait des `send`. La composition entre les P_i et $FIFO_{1,2}$ est asynchrone, donc je définis le système GALS $\hat{P}$ comme étant $\hat{P} = P_1 \,{}_{\delta}\|_{\delta}\, FIFO_{1,2} \,{}_{\delta}\|_{\delta}\, P_2$.

Dans [8], Benveniste et al. étudient le déploiement correct de systèmes sur divers types d'architectures. Par exemple, soit un système S constitué de deux sous systèmes synchrones S_i, $i = 1, 2$ qui sont composés de façon synchrone. Son déploiement sur une architecture M est correct si $S_1 \,\|\, Id \,\|\, S_2 = S_1 \,{}_{\rho}\|_{\rho}\, M \,{}_{\rho}\|_{\rho}\, S_2$, où Id est le processus identité synchrone et où ρ est le morphisme de $\mathcal{T}_{\text{synch}}$ vers $\mathcal{T}_M$,

la structure de tags de l'architecture M. Pour le démontrer, il faut modéliser les comportements de M et la façon dont S_1 et S_2 communiquent avec M.

Dans mon cas, le déploiement est correct si $P = \hat{P}$. Comme P_1 et P_2 ont été obtenus par un algorithme de répartition automatique, le preuve que le déploiement est correct est fournie par la section 2.3.4.

2.7 Autres approches connexes

2.7.1 Répartition de programmes LUSTRE et SIMULINK sur TTA

Au sein du laboratoire VERIMAG, Paul Caspi et ses collègues ont mené des travaux sur la répartition de programmes LUSTRE [31, 43] et leur déploiement sur des architectures TTA [67] (« Time-Triggered Architecture »). L'architecture TTA est construite autour de plusieurs bus de communication et est basée sur le protocole de communication TTP (« Time-Triggered Protocol »), qui est une implémentation du protocole TDMA (« Time Division Multiple Access »). Le principe est que chaque machine désirant communiquer se voit allouer un créneau, la succession des créneaux étant la même à chaque cycle. C'est un fonctionnement qui respecte parfaitement l'hypothèse de synchronisme.

Dans ce travail, les auteurs partent du constat que, dans l'industrie des logiciels de contrôle/commande, SIMULINK constitue un standard de facto, et ils proposent de traduire les programmes SIMULINK vers LUSTRE, avant de les déployer sur des architectures TTA. Ils identifient tout d'abord quatre différences principales entre LUSTRE et SIMULINK :

1. la sémantique de LUSTRE est en temps discret alors que celle de SIMULINK est en temps continu ;
2. la sémantique de LUSTRE est unique et précise alors que celle de SIMULINK dépend du mode de simulation choisi (par exemple, un modèle SIMULINK peut être accepté avec le mode de simulation à pas variable, mais refusé avec le mode à pas fixe ou automatique) ;
3. les types en LUSTRE sont fort et explicites, alors qu'ils sont implicites en SIMULINK ;
4. LUSTRE permet une conception modulaire alors que SIMULINK non (en particulier, un sous-système B d'un système A peut être échantillonné à n'importe qu'elle période, y compris une période plus rapide que celle de A).

De ces différences, les auteurs tirent six conséquences quant à la traduction des programmes SIMULINK vers LUSTRE :

1. seule la partie en temps discret et non ambiguë de SIMULINK peut être traduite ;
2. la méthode de simulation de SIMULINK doit être `solver=fixedstep+discrete` et `mode=auto` ;
3. le programme LUSTRE doit être exécuté à la même période que celle utilisée pour simuler le modèle SIMULINK ;
4. le modèle SIMULINK doit avoir l'indicateur de signaux booléens positionné à `true` ;
5. les modèles SIMULINK avec boucles algébriques doivent être rejetés (ce sont des boucles qui ne sont pas cassées par un délai unitaire — j'en ai déjà parlé dans la section 2.3.7) ;
6. la hiérarchie SIMULINK doit être préservée.

Puis les auteurs proposent quatre extensions de LUSTRE afin de permettre la génération de code réparti. Ces extensions consistent en des annotations du programme source. Ce sont :

1. `location=P` afin de spécifier sur quel site de calcul un bloc donné doit être exécuté ;
2. `(hyp) basic_period=p` afin de spécifier la période de base d'un bloc donné ; une horloge périodique de période `p` et de déphasage `k` se définit au moyen de l'instruction `periodic_cl(k,p)` ;

3. `(hyp) exec_time(A) in [l,u]` afin de spécifier que le WCET de `A` est entre `l` et `u` unités de temps ;
4. `(req) date(y) - date(x) < 5` afin de spécifier que la variable `y` est disponible au plus `5` unités de temps après l'instant auquel `x` est disponible.

Enfin, Paul Caspi est ses collègues s'attaquent au déploiement proprement dit de programmes LUSTRE sur des architectures TTA. Le point de départ est un programme LUSTRE annoté avec les extensions présentées ci-dessus. À partir de là, le compilateur LUSTRE produit le graphe G de l'ordre partiel des dépendances de données. Grâce aux annotations `location`, ce graphe G peut être partitionné en plusieurs fragments, qui forment chacun une tâche. Ces tâches doivent ensuite être ordonnancées sur l'architecture TTA. Mais les annotations `basic_period`, `exec_time` et `date` induisent des contraintes sur ce problème d'ordonnancement. Les auteurs se trouvent ainsi face à un un problème d'ordonnancement multi-période multi-processeur, qui est le point clé de leur proposition. Si aucune solution n'est trouvée, le graphe G est raffiné, et ainsi de suite jusqu'à ce qu'une solution soit trouvée.

Cependant, si on veut prendre en compte des tâches de longue durée (définies à la section 2.2.6), il est nécessaire de partitionner le programme source en blocs a priori plus petits. Une telle partition doit être spécifiée par l'utilisateur lui-même au niveau du programme source, alors que la méthode que je présente à la section 2.3 réalise cette partition automatiquement.

Récemment, Zennaro et Sengupta ont proposé une méthode pour la répartition de programmes SIMULINK [100]. Ils se restreignent aux programmes SIMULINK dont la sémantique est obtenue avec la méthode de simulation `solver=fixedstep+discrete`, et produisent un programme réparti dont les fragments communiquent par des FIFOs bornées. Une limitation de leur approche est l'impossibilité d'avoir des boucles de dépendance instantanées dans le programme SIMULINK initial, même si elles sont cassées au niveau des sous-programmes par des délais. C'est l'inconvénient classique des méthodes de répartition du code source (voir la section 2.2.3).

2.7.2 Répartition de programmes AUTOFOCUS

En parallèle, Romberg et Bauer de TU München ont également étudié le déploiement de programmes flot-de-données, écrits dans le formalisme AUTOFOCUS [71], sur des architectures **faiblement time triggered** telles que LTTA [12]. Ils se sont placés dans le domaine des **systèmes temps-réel fermes**, où la perte d'un nombre borné de communications entre les différentes parties du système n'est pas fatale [92]. Dans ce contexte, ils ont défini une **cascade de synchronisation** comme un arbre, où chaque nœud correspond à un processeur de l'architecture répartie cible, et où chaque arête est un **lien de synchronisation** transmettant périodiquement des messages utilisés par le nœud fils (dans l'arbre) pour se synchroniser avec le nœud père (dans l'arbre). La racine de l'arbre est le **nœud maître**, celui qui enverra l'horloge de base à tout le programme. Une cascade fournit une couche de synchronisation et de communication, et peut être plaquée sur la topologie physique de l'architecture répartie cible. Projeter un programme flot-de-données réparti sur une cascade permet une forme d'exécution faiblement synchronisée : le fragment de programme projeté sur le nœud maître est auto-déclenché (en réalité, il est déclenché par une horloge système), alors que tous les autres fragments sont déclenchés de façon externe (c'est-à-dire par leur parent respectif dans la cascade). Si un fragment veut communiquer avec un autre fragment qui se trouve projeté ni sur son père ni sur son fils dans la cascade, un **lien non synchronisant** de communication est ajouté dans ce but à la cascade, ce qui en fait donc un graphe orienté et non plus un arbre.

Les auteurs définissent alors plusieurs propriétés clés concernant les liens de communication (le nombre de pertes de messages) et sur les horloges des nœuds (la dérive de leur horloge), qui garantissent l'exécution synchronisée du programme réparti plaqué sur une cascade. En particulier, dans des conditions de fonctionnement normal ou en cas de défaillances transitoires, l'exécution du programme réparti est fonctionnellement correcte et les nœuds de la cascade sont tous correctement synchronisés,

c'est-à-dire qu'ils reçoivent les bons messages de synchronisation de leur père respectif. Ce résultat permet proposer une méthode pour l'ingénierie des systèmes répartis : le concepteur de l'algorithme de contrôle/commande doit déterminer le nombre maximal de messages qui peuvent être perdus tout en préservant la robustesse du système, ce qui permet de fixer les conditions sur les liens de synchronisation de la cascade utilisée pour déployer le programme.

Romberg et Bauer décrivent enfin succinctement le déploiement d'un programme flot-de-données sur de telles cascades. Le découpage du programme source en modules doit faire en sorte que ceux-ci communiquent entre eux au moyen de délais unitaires (exactement comme l'opérateur `pre` de LUSTRE). Contrairement aux méthodes que j'ai présentées, le découpage du programme doit en outre être effectué manuellement par l'utilisateur, alors que mes directives de répartition consistent juste en un découpage de l'ensemble des entrées et sorties du programme.

2.7.3 La répartition robuste – Le projet CRISYS

Le projet européen CRISYS [9] s'est intéressé aux pratiques existantes dans l'industrie pour la programmation sûre d'automatismes répartis critiques [30]. C'est dans ce cadre qu'une approche robuste pour la conception de programmes synchrones répartis sur des architectures quasi-synchrones a été proposée [36, 24].

Ici, **robuste** signifie que le système réparti doit être le moins sensible possible aux défaillances matérielles ; c'est-à-dire que la défaillance d'un composant ne doit pas se propager aux autres composants du système ; la seule conséquence de la défaillance d'un composant est donc que les sorties calculées par ce composant n'évoluent plus et restent figées. Pour cela, il est nécessaire que les fragments du programme réparti ne communiquent pas entre eux au moyen de primitives synchronisantes. Aussi le moyen de communication choisi est-il la **mémoire partagée**, ce qui peut être mis en œuvre aussi bien dans des réseaux de communication point-à-point qu'avec des bus périodiques. Concrètement, les deux primitives suivantes sont utilisées pour lire et écrire dans la mémoire partagée :

- `u' = u when cr` pour lire la variable `u` à l'horloge de lecture `cr` et l'affecter à la variable locale `u'` ;
- `write(v,cw,u) = v->pre (if cw then current u else write(v,cw,u))` pour écrire la variable `u` à l'horloge d'écriture `cw`.

Quant aux architectures réparties **quasi-synchrones**, elles n'ont pas d'horloge unique mais plusieurs horloges, une par site de calcul, qui ne sont pas synchronisées ; la seule hypothèse que doivent vérifier ces horloges périodiques est qu'elles doivent avoir des périodes **presque identiques**, c'est-à-dire que chaque horloge doit avoir au plus deux valeurs `true` entre deux valeurs `true` successives de toute autre horloge.

Par conséquent, si deux processus périodiques sont exécutés sur deux processeurs dont les horloges sont presque identiques, et communiquent par mémoire partagée, des valeurs peuvent être perdues ou écrasées ! Néanmoins, cela ne pose pas de problème pour tout une classe de systèmes périodiques distribués où les signaux sont continus et surtout **lisses** ; c'est même une pratique courante dans l'industrie. Par définition, un signal échantillonné `u` est lisse si et seulement `|u - pre u| < epsilon`, avec `epsilon` fixé.

En définitive, ce n'est pas réellement une méthode de répartition de programmes synchrones qui est proposée dans [36], mais plutôt une méthode de vérification qu'une distribution déjà existante d'un programme synchrone en plusieurs fragments a le même comportement que le programme centralisé. Trois conditions sont pour cela nécessaires :

1. les fragments doivent avoir la même horloge logique ou des sous-multiples d'une même horloge logique ;

[9] CRISYS : « Critical instrumentation of control systems », Esprit Project 25.514, 1997 à 2000.

2. il doivent être implantés sur des machines dont les horloges sont presque identiques ;
3. et les variables de communication doivent être numériques et lisses ; pour chaque variable de communication `u`, de variation maximale `epsilon`, la propriété d'échantillonnage est alors `|write(v,cr,u') - write(v,cw,u)| < 2*epsilon`, où `cr` est l'horloge de lecture, `cw` est l'horloge d'écriture, et `u' = write(v,cw,u) when cr`.

L'avantage, démontré par les auteurs, est que le programme réparti est robuste et qu'il n'y a pas besoin d'un mécanisme de communication bloquant comme les FIFOs. De plus, cette approche est très utilisée dans l'industrie, par exemple AIRBUS (A320 et successeurs), MERLIN-GERIN (centrales nucléaires), MAGGALY (ligne D du métro de Lyon), METEOR (ligne 14 du métro de Paris)... C'est précisément de ces réalisations que les auteurs se sont inspirés pour formaliser leur approche.

Par ailleurs, dans [24], Budde et Poigné proposent également une méthode de conception de programmes répartis sur architectures quasi-synchrones. Le moyen de communication utilisé est le **tableau noir**, sorte de mémoire partagée avec un écrivain et plusieurs lecteurs ; ni la lecture ni l'écriture ne sont bloquantes, donc c'est bien un moyen de communication asynchrone et robuste. Afin de vérifier formellement des propriétés logiques sur de tels programmes, les auteurs proposent d'abstraire le comportement d'un tableau noir par un nœud LUSTRE, d'abstraire également le programme complet par un nœud LUSTRE, et enfin d'utiliser des outils de model-checking sur le programme LUSTRE résultant [66], c'est-à-dire incluant un nœud pour chaque fragment du programme réparti, chaque tableau noir et le programme principal.

2.7.4 Compilation séparée et exécution distribuée de programmes ESTEREL

Au sein de l'équipe ASTRE de l'ENST, Olivier Hainque a travaillé à la compilation séparée et l'exécution répartie de programmes ESTEREL [63]. La motivation principale était de diminuer le temps de compilation des gros programmes ESTEREL. Son point de départ est un programme ESTEREL, à partir duquel il génère du code pour chacun des modules instanciés au niveau du programme principal, puis du code réparti vers une architecture cible donnée (du code centralisé étant également possible). Il considère que le module est l'unité de base de répartition, c'est-à-dire que les directives de répartition de l'utilisateur doivent indiquer sur quel site de calcul doit s'exécuter chaque instance de module.

Trois hypothèses restrictives sont faites sur la structure et l'interface des programmes ESTEREL admissibles :

1. le programme principal doit être constitué de la mise en parallèle d'instances de modules et son interface ne doit contenir que des signaux d'entrée/sortie ;
2. aucun de ces modules ne doit contenir d'émission sur entrée, d'évaluation de sorties, ou de substitution entrée/sortie ou sortie/entrée lors d'instanciations ;
3. et enfin chacun de ces modules doit être compilable indépendamment de son contexte d'appel ; pour que cela soit possible, il faut qu'il n'y ait aucun cycle instantané de communication entre les modules instanciés par le programme principal.

Des alternatives sont proposées afin de récrire un programme ESTEREL de façon à ce qu'il se conforme aux deux premières hypothèses. En revanche, rien n'est proposé concernant la troisième hypothèse.

Cette troisième hypothèse et le fait que le module constitue l'unité de base de répartition constituent les principales différences avec les approches que j'ai proposées dans ce chapitre : mes trois algorithmes de répartition automatiques acceptent en effet des programmes ESTEREL sans restriction et le grain de répartition est le signal d'entrée/sortie et non pas le module.

Enfin, à ce jour, la compilation séparée de programmes synchrones constitue toujours un challenge. Je vois à cela une raison principale : la communication entre les modules est instantanée (diffusion synchrone), donc le comportement d'un module peut influer à un niveau très profond sur le comportement

d'un autre module en parallèle ; il en résulte entre autres que l'analyse de causalité nécessite d'exhiber le comportement global du programme et ne peut pas généralement être réalisée sur un module indépendamment de son contexte d'appel. Donc, l'approche proposée par Olivier Hainque pour ESTEREL marche grâce aux hypothèses faites sur les programmes admissibles : cela permet de tester des conditions sur les dépendances entre les entrées et les sorties de chaque module, de vérifier leur compatibilité par rapport à l'instanciation de ces modules au niveau du programme principal, et ainsi de repousser l'édition de lien le plus tard possible. Pascal Raymond a démontré quant à lui dans le cas de LUSTRE que c'est impossible en toute généralité, et il a exhibé les conditions sur les entrées et sorties d'un module pour que cela soit possible [91]. Enfin dans le cas de SIGNAL, les chercheurs de Rennes ont établi les conditions sous lesquelles la compilation séparée est possible (voir la section 2.7.7). Enfin, je remarque que les nouvelles méthodes de compilation d'ESTEREL en points de contrôle donnent moins de motivation à la compilation séparée puisque le temps de compilation est très court [99, 47, 89].

2.7.5 Machines réparties réactives

Dans [96], Boussinot et ses collègues définissent un **système synchronisé** comme étant composé de plusieurs **machines réactives** (RM) et de plusieurs **synchroniseurs** (SYNC). Les auteurs présentent également une implémentation de tels systèmes synchronisés à l'aide des SUGARCUBES [22]. Une RM peut être ou bien déconnectée, ou bien connectée à un unique SYNC. Toutes les RMs connectées à un même SYNC partagent la même horloge globale et ont une vue cohérente de l'instant logique. Les RMs communiquent par **diffusion synchrone** : une RM peut envoyer un message à toutes les autres RMs connectées au même SYNC ; ce message sera reçu pendant le même instant logique. Si l'envoi est fait vers une RM connectée à un autre SYNC, alors le message lui sera retransmis de manière asynchrone. Par ailleurs, une RM non connectée peut se connecter dynamiquement à un SYNC, et une RM connectée peut se déconnecter dynamiquement. Un système synchronisé est donc un réseau de RMs et de SYNCs évoluant dynamiquement.

Un aspect important est le calcul de la réaction de l'ensemble des RMs connectées à un même SYNC. Cela comporte plusieurs phases : au début d'une réaction, la première phase consiste à ce que toutes ces RMs réagissent indépendamment les unes des autres, jusqu'à ce qu'elles aient toutes terminé ou soient suspendues ; à ce moment, leur SYNC diffuse tous les événements reçus pendant la phase précédente ; une deuxième phase suit alors, consistant à ce que les RMs réagissent à ces nouveaux événements, et encore jusqu'à ce qu'elles aient toutes terminé ou soient suspendues ; puis leur SYNC diffuse les nouveaux événements reçus pendant la deuxième phase ; et ainsi de suite jusqu'à ce que les SYNCs n'aient plus aucun événement à diffuser ; à la fin de la réaction, tous les événements qui n'ont pas été présents sont déclarés absents, afin que les RMs puissent réagir à de telles absences au cours de la *réaction suivante*. Afin de garantir la cohérence de la réaction, une RM ne peut se connecter à un SYNC qu'au début d'une réaction, et ne peut se déconnecter qu'à la fin d'une réaction.

On voit ainsi que l'approche des machines réactives réparties est une approche *interprétée*, où la réaction à l'absence est retardée à l'instant suivant. Au contraire, les langages synchrones classiques ont une approche *compilée*, et la réaction à l'absence se fait dans le même instant.

2.7.6 La méthodologie AAA et l'outil de CAO SYNDEX

SYNDEX est un outil permettant la répartition automatique de programmes synchrones spécifiés dans un formalisme flot-de-données. Il est développé par Yves Sorel et son équipe dans le projet AOSTE de l'INRIA [76, 110, 77, 46, 47].[10] Il prend en entrée un **algorithme flot-de-données** représenté par un graphe $\mathcal{A}lg$, une **architecture multi-processeurs hétérogène à mémoire répartie** représentée par un graphe $\mathcal{A}rc$, les **caractéristiques d'exécution** $\mathcal{E}xe$ des composants logiciels de $\mathcal{A}lg$ sur les composants

[10] SYNDEX = « SYNchronized Distributed EXecutive ». `http://www-rocq.inria.fr/syndex`.

matériels de $\mathcal{A}rc$ (c'est-à-dire les WCET), et une **contrainte de temps maximal d'exécution** Rose et al.. Il est également possible de spécifier des contraintes de répartition si l'on désire que certains blocs de $\mathcal{A}lg$ soient exécutables seulement sur certains processeurs spécifiques de $\mathcal{A}rc$, un peu comme les directives de répartition d'`ocrep`. SYNDEX produit en sortie un **ordonnancement statique réparti** de $\mathcal{A}lg$ sur $\mathcal{A}rc$. Cet ordonnancement étant statique, il est possible de calculer la longueur de son chemin critique en fonction de $\mathcal{E}xe$, et donc de la comparer à la contrainte de temps d'exécution maximal imposée par l'utilisateur Rose et al.. Les concepteurs de SYNDEX appellent cette méthodologie AAA (« Adéquation Algorithme-Architecture »). En outre, SYNDEX génère, à partir de l'ordonnancement, un **exécutif réparti synchronisé**, qui permet de déployer et d'exécuter l'algorithme sur l'architecture sans avoir recours à un système d'exploitation.

J'identifie plusieurs différences entre SYNDEX et les algorithmes automatique de répartition sur lesquels j'ai travaillé :

- SYNDEX entre dans la classe des logiciels de répartition qui tentent de trouver la *meilleure répartition* modulo un critère, ici la longueur du chemin critique de l'ordonnancement. Au contraire, les méthodes de répartition sur lesquelles j'ai travaillé ne se préoccupent pas de cet aspect.
- L'algorithme $\mathcal{A}lg$ à répartir est décrit à un niveau macroscopique puisque c'est un graphe de blocs logiciels, alors que mes algorithmes de répartition travaillent au niveau des instructions du programme à répartir. C'est d'ailleurs ce niveau macroscopique qui permet à SYNDEX d'optimiser le chemin critique de gros programmes dans un temps acceptable.
- Il ne semble pas possible en SYNDEX de désynchroniser le rythme (comme dans la section 2.3.6) ou de faire de la répartition dirigée par les horloges (comme dans la section 2.3.7).

Par ailleurs, plusieurs chercheurs ont eu l'idée d'utiliser SYNDEX comme post-processeur pour générer du code réparti à partir de langages de haut niveau. Ainsi, Mihaela Sighireanu a proposé une traduction du format SC (voir la section 2.4.1) vers DC (voir la section 2.1.3) et enfin vers SYNDEX [94]. Le but était de relier l'outil SYNDEX et le langage ESTEREL. Cette traduction se fait donc à partir de la représentation « circuit » du programme source ESTEREL.

Plus récemment, Fabrice Peix a proposé une traduction du format GRC (« GRaph Code ») vers SYNDEX [88]. Le format GRC a été à l'origine développé par Jacky Potop pour la simulation efficace des programmes ESTEREL [89]. Un programme GRC comporte deux structures : d'une part un **arbre de sélection** pour représenter l'état du programme de manière explicite et hiérarchique, et d'autre part un **graphe flot-de-données et de contrôle** représentant les actions effectuées par le programme. Cette traduction se fait donc à partir d'une représentation plus proche du programme source ESTEREL, dans laquelle les relations de parallélisme et d'exclusion sont mieux représentées que dans la représentation « circuit ». Concrètement, elle présente les avantages suivants par rapport à la traduction depuis SC / DC :

- Dans la représentation SC, le contrôle et les données sont mélangés, et l'ensemble du programme est évalué par la propagation de valeurs `1` et `0` : les `1` activent des actions alors que les `0` ne les activent pas. Cela rend impossible l'exploitation d'**informations d'exclusion** entre les différentes parties du programme. Dans la traduction depuis GRC, ce défaut est résolu en regroupant dans des **blocs d'activité** des instructions ayant la même activation.
- Le parallélisme ESTEREL est mieux préservé par le format GRC, ce qui permet à SYNDEX de mieux exploiter le parallélisme potentiel du graphe flot-de-données qui lui est fourni en entrée. Il en résulte une meilleure répartition sur l'architecture répartie cible.
- La non-émission des signaux est codée explicitement par l'ajout d'instructions au programme GRC, alors que dans la traduction depuis SC / DC, cette information est directement codée dans le circuit de contrôle, ce qui induit beaucoup de dépendances inutiles.

2.7.7 Désynchronisation de programmes SIGNAL

Plusieurs chercheurs de l'INRIA-Rennes, sous la direction d'Albert Benveniste et Paul Le Guernic, ont effectué un grand nombre de travaux sur la répartition automatique de programmes SIGNAL, aussi bien du côté théorique que pratique [5, 6, 82, 26, 10, 90]. En particulier, le projet européen SACRES [13] a été consacré à la répartition de programmes DC+ (un des formats de sortie du compilateur SIGNAL, présenté dans la section 2.1.3).

Du point de vue pratique, la désynchronisation d'un programme SIGNAL intervient après la phase de compilation du programme source en un GFDS (voir la section 2.1.3). Puis le GFDS est projeté en n sous-graphes, un par site de calcul, et enfin du code séquentiel est généré à partir de chaque sous-graphe. Toute la difficulté provient du fait que, en général, il n'est pas possible de produire le code séquentiel pour un sous-graphe sans tenir compte des contraintes d'ordonnancement dues aux autres sous-graphes (voir la section 2.2.3 pour un exemple).

Du point de vue théorique, étant donné un déploiement désiré du programme SIGNAL sur une architecture cible, la désynchronisation consiste à vérifier que le déploiement considéré peut être effectué en préservant le sens de la spécification synchrone initiale. Cela est réalisé en s'assurant que cette spécification satisfait aux critères de l'**endo-isochronie** [10]. Plus précisément, considérons un programme SIGNAL que l'utilisateur désire déployer sur une architecture à n processeurs. Chacun des n fragments est d'abord compilé en un « Système de Transition Synchrone » (STS) S_i, avec $1 \leq i \leq n$. Il faut d'une part que chaque S_i soit **endochrone**, et d'autre part que chaque paire $\langle S_i, S_j \rangle$ soit **isochrone** :

- Un STS est **endochrone** si, à tout instant, la présence et l'absence de chacune de ses variables peut être inférée incrémentalement à partir des valeurs déjà connues des variables présentes et des variables d'état. Autrement dit, le contrôle ne doit dépendre que de l'état précédant et des valeurs des variables d'environnement, mais en aucun cas de la présence ou de l'absence de ces variables. Une conséquence est que, pour un STS endochrone, il est possible de reconstruire les traces d'un STS à partir des traces désynchronisées : la perte de la barrière synchrone qui définit les réactions successives du programme ne va pas entraîner un changement de la sémantique du STS. En terme d'horloges, un STS endochrone a un arbre d'horloges alors qu'un STS exochrone a une forêt d'horloges (voir aussi à ce sujet la section 2.1.3).
- Deux STSs S_1 et S_2 sont **isochrones** si le résultat de leur composition synchrone est équivalent au résultat de leur composition asynchrone ; plus précisément, si S_1 et S_2 **concordent** sur toutes les variables qu'ils ont en commun et qui sont présentes des deux côtés (concordent c'est-à-dire que ces variables ont la même valeur des deux côtés) et si il existe au moins une telle variable, alors S_1 et S_2 doivent concorder sur toutes leurs variables communes (c'est-à-dire qu'elles doivent être absentes des deux côtés ou présente avec la même valeur).

Cependant, les deux notions d'endo-isochronie, telles qu'elles sont définies dans [10], ne conviennent pas parfaitement, car d'une part l'endochronie n'est pas compositionnelle, et d'autre part l'isochronie ne se généralise pas à plus de deux STS, donc elle n'est pas associative. Aussi, ces notions ont été par la suite relaxées pour donner l'**endochronie faible** et l'**isochronie faible** [90].

2.8 Discussion et développements futurs

2.8.1 Bilan sur mes contributions

Historiquement, j'ai commencé par travailler sur la répartition de programmes OC, et j'ai proposé dans ce cadre un grand nombre d'optimisations : élimination des messages redondants, désynchronisation du rythme, répartition dirigée par les horloges. Puis j'ai proposé des variantes de cette méthode afin de répartir les programmes SC et les programmes CP. Enfin, j'ai contribué à l'obtention de résultats théoriques et pratiques importants, concernant la correction de l'algorithme de répartition, l'équivalence

sémantique entre un programme centralisé et le programme réparti correspondant, et l'exécution correcte de programmes synchrones répartis dans un environnement asynchrone.

2.8.2 Choix pertinent de la bonne méthode de répartition

En définitive, il n'existe pas une mais de nombreuses méthodes automatiques de répartition pour les programmes synchrones. Je me suis attaché à présenter dans ce chapitre celles qui sont le fruit de mes propres recherches. Comme on le voit à la lecture des autres travaux du domaine, le choix pertinent d'une méthode de répartition dépend du langage source (LUSTRE, ESTEREL, SIGNAL...), du type du programme source (orienté donnée ou orienté contrôle), du type de l'architecture cible considéré (synchrone, quasi synchrone ou asynchrone), et des besoins de répartition (dirigée par les horloges, dirigée par les capteurs et actionneurs, requise pour la sûreté de fonctionnement...). Il se peut tout à fait que pour un problème donné, il n'existe pas de solution pleinement satisfaisante et qu'il faille alors combiner plusieurs méthodes. Enfin, quand le but recherché est la sûreté de fonctionnement du système, je présente dans le chapitre suivant plusieurs méthodes d'ordonnancement & répartition basées sur la méthode AAA et l'outil SYNDEX.

2.8.3 Répartition de programmes fbt-de-données d'ordre supérieur

Lors du colloque sur la programmation synchrone à Saint-Nazaire en décembre 2000, Marc Pouzet (au laboratoire LRI, Orsay) et moi-même avons eu l'idée de combiner la **répartition automatique de programmes synchrones** avec la **programmation fonctionnelle d'ordre supérieur**. Marc Pouzet travaillait déjà à ce moment sur le langage LUCID Synchrone [37] [11], une extension de LUSTRE à l'ordre supérieur, mais où l'ordre supérieur était **statique**. En effet une fonction pouvait être paramétrée par une autre fonction, mais cette dernière devait être connue statiquement et ne pouvait donc pas évoluer en fonction des valeurs calculées pendant l'exécution du programme.

Quatre ans plus tard, cette idée a donné lieu à DECADE, une extension à l'ordre supérieur **dynamique** de LUSTRE [41]. Ce langage est appelé DECADE en référence à LUSTRE. Dans DECADE, une fonction paramètre d'une autre fonction peut évoluer dynamiquement pendant l'exécution du programme.

Dans un langage flot-de-données classique (j'entends par là de premier ordre, comme LUSTRE, SIGNAL...), une fonction est une machine d'états finie transformant des flots de valeurs d'entrée en flots de valeurs de sorties. Il y a donc une séparation nette entre les données (qui sont des flots) et les fonctions (qui sont des machines d'états finies). Au contraire, dans DECADE, *tous* les éléments manipulés (variables et fonctions) sont des flots, les fonctions étant en réalité des flots de fonctions (de fonctions de flots bien sûr). Ce langage est donc le premier langage vraiment flot-de-donnée, puisque tous les éléments manipulés sont des flots. Enfin, DECADE est une extension conservative de LUSTRE, dans le sens où sa restriction au premier ordre a exactement la même sémantique que LUSTRE.

Notre idée est que la répartition de programme DECADE permettra d'exprimer dans un cadre synchrone la **mobilité de code**, puisque les fragments répartis pourront s'échanger des flots de fonctions évoluant dynamiquement. Cette approche devrait trouver des applications dans la **radio logicielle** (« software defined radio » en anglais [85, 20]), où justement des systèmes embarqués tels que des téléphones cellulaires doivent modifier dynamiquement des modules de leur chaîne de traitement du signal, voire même télécharger à la volée de nouveaux modules. Ce projet de recherche soulève des problèmes absolument non triviaux, liés à la structure de contrôle des programmes DECADE et à l'ordre supérieur. Par exemple, quel sens donner à un flot de fonctions séquentielles n'ayant pas toutes les mêmes variables d'état ? Où encore, comment rendre mobile un nœud ayant des variables libres ? Marc Pouzet et moi-même co-encadrons actuellement la thèse de Gwenaël Delaval sur ce passionnant sujet de recherche.

[11] LUCID Synchrone : `http://www.lri.fr/~pouzet/lucid-synchrone`.

Bibliographie

[1] G. Agrawal. Interprocedural partial redundancy elimination with application to distributed memory compilation. *IEEE Trans. on Parallel and Distributed Systems*, 9(7) :609–625, 1998.

[2] S.P. Amarasinghe and M.S. Lam. Communication optimization and code generation for distributed memory machines. In *Conference on Programming Language Design and Implementation, PLDI'93*, Albuquerque (NM), USA, June 1993. ACM SIGPLAN.

[3] C. André. Representation and analysis of reactive behaviors : A synchronous approach. In *CESA'96*, Lille, France, July 1996. IEEE-SMC.

[4] C. André, F. Boulanger, and A. Girault. Software implementation of synchronous programs. In *International Conference on Application of Concurrency to System Design, ACSD'01*, pages 133–142, Newcastle, UK, June 2001. IEEE.

[5] P. Aubry and P. Le Guernic. On the desynchronization of synchronous applications. In *11th International Conference on Systems Engineering, ICSE'96*, Las Vegas (NV), USA, June 1996.

[6] P. Aubry, P. Le Guernic, and S. Machard. Synchronous distributions of Signal programs. In *Hawaii International Conference on System Sciences, HICSS'96*, pages 656–665, Honolulu (HA), USA, January 1996. IEEE.

[7] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems : The Polis Approach*. Kluwer Academic Publishers, June 1997.

[8] A. Benveniste, B. Caillaud, L. Carloni, P. Caspi, and A. Sangiovanni-Vincentelli. Composing heterogeneous reactive systems. *Submitted to ACM TECS*, 2005.

[9] A. Benveniste, B. Caillaud, L. Carloni, and A. Sangiovanni-Vincentelli. Tag machines. In W. Wolf, editor, *International Conference on Embedded Software, EMSOFT'05*, pages 255–263, Jersey City, USA, September 2005. ACM.

[10] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages : Specification and distributed code generation. *Information and Computation*, 163 :125–171, 2000.

[11] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1) :64–83, January 2003. Special issue on embedded systems.

[12] A. Benveniste, P. Caspi, P. Le Guernic, H. Marchand, J.-P. Talpin, and S. Tripakis. A protocol for loosely time-triggered architectures. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *International Workshop on Embedded Software, EMSOFT'02*, volume 2491 of *LNCS*, pages 266–281, Grenoble, France, October 2002. Springer-Verlag.

[13] A. Benveniste et al. Safety critical embedded systems design : the SACRES approach. Tutorial at the Symposium on Formal Techniques in Real-Time and Fault Tolerant systems, FTRTFT'98, September 1998.

[14] G. Berry. Esterel on hardware. *Philosophical Transaction Royal Society of London*, 339 :87–104, 1992.

[15] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction : Essays in Honour of Robin Milner*, pages 425–454. MIT Press, 2000.

[16] G. Berry and A. Benveniste. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9) :1270–1282, September 1991.

[17] G. Berry and L. Cosserat. The Esterel synchronous language and its mathematical semantics. In S.D. Brooks, A.W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, pages 389–448. Springer-Verlag, 1984.

[18] G. Berry and G. Gonthier. The Esterel synchronous programming language : Design, semantics, implementation. *Science of Computer Programming*, 19(2) :87–152, 1992.

[19] G. Berry and E. Sentovich. An implementation of constructive synchronous constructive programs in Polis. *Formal Methods in Systems Design*, 17(2) :165–191, October 2000.

[20] S. Blust. SDR forum roles and global work focus on radio software download. *IEICE Trans. on Communications*, E85-B(12) :2581–2587, December 2002.

[21] F. Boussinot and R. de Simone. The SL synchronous language. *IEEE Trans. on Software Engineering*, 22(4) :256–266, April 1996.

[22] F. Boussinot and J.-F. Susini. The SugarCubes tool box : A reactive JAVA framework. *Software Practice and Experience*, 28(14) :1531–1550, December 1998.

[23] J.T. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt. Ptolemy : a framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, 4 :155–182, April 1994. Special Issue on Simulation Software Development.

[24] R. Budde and A. Poigné. On the synthesis of distributed synchronous processes. In *Modelling and Verification of Parallel Processes, MOVEP'00*, Nantes, France, June 2000. IRCYN. Crisys Esprit Project 25.514.

[25] B. Buggiani, P. Caspi, and D. Pilaud. Programming distributed automatic control systems : A language and compiler solution. Research report Spectre L4, LGI/IMAG, Grenoble, France, July 1988.

[26] B. Caillaud. *Contribution à la Modélisation du SPMD : Distribution Asynchrone d'Automates*. Thèse de doctorat, IFSIC / Université de Rennes I, Rennes, France, June 1994.

[27] B. Caillaud, P. Caspi, A. Girault, and C. Jard. Distributing automata for asynchronous networks of processors. *Journal Européen des Systèmes Automatisés (RAIRO-APII-JESA)*, 31(3) :503–524, 1997. Research report Inria 2341.

[28] D. Callahan and K. Kennedy. Compiling programs for distributed memory multiprocessors. *Journal of Supercomputing*, 2 :151–169, 1988.

[29] P. Caspi. Clocks in data-flow languages. *Theoretical Computer Science*, 94 :125–140, 1992.

[30] P. Caspi. The quasi-synchronous approach to distributed control system design. Rapport de recherche, Crisys Esprit Project 25.514, October 2000. `http://www-verimag.imag.fr/~caspi/PAPIERS/cooking.html`.

[31] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to Scade/Lustre to TTA : A layered approach for distributed embedded applications. In *International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES'03*, pages 153–162, San Diego (CA), USA, June 2003. ACM.

[32] P. Caspi, J.-C. Fernandez, and A. Girault. An algorithm for reducing binary branchings. In P.S. Thiagarajan, editor, *15th Conference on the Foundations of Software Technology and Theoretical Computer Science, FST&TCS'95*, volume 1026 of *LNCS*, pages 279–293, Bangalore, India, December 1995. Springer-Verlag.

[33] P. Caspi and A. Girault. *OCREP : An Automatic Parallelization Tool for Synchronous Language*. Inria. User Manual. `http://pop-art.inrialpes.fr/~girault/Ocrep`.

[34] P. Caspi and A. Girault. Execution of distributed reactive systems. In S. Haridi, K. Ali, and P. Magnusson, editors, *1st International Conference on Parallel Processing, EURO-PAR'95*, volume 966 of *LNCS*, pages 15–26, Stockholm, Sweden, August 1995. Springer-Verlag.

[35] P. Caspi, A. Girault, and D. Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Trans. on Software Engineering*, 25(3) :416–427, May 1999.

[36] P. Caspi, C. Mazuet, R. Salem, and D. Weber. Formal design of distributed control systems with Lustre. In *International Conference on Computer Safety, Reliabilitiy, and Security, SAFECOMP'99*, number 1698 in LNCS, pages 396–409, Toulouse, France, September 1999. Springer-Verlag. Crisys Esprit Project 25.514.

[37] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'96*, Philadelphia (PA), USA, May 1996. ACM Press.

[38] P. Caspi and M. Pouzet. Lucid synchrone : une extension fonctionnelle de Lustre. In *Journées Francophones des Langages Applicatifs (JFLA)*, Morzine, France, February 1999. Inria.

[39] D.M. Chapiro. *Globally Asynchronous Locally Synchronous Systems*. Thèse de doctorat, Stanford University, October 1984.

[40] M. Clerbout and M. Latteux. Semi-commutations. *Information and Computation*, 73 :59–74, 1987.

[41] J.-L. Colaço, A. Girault, G. Hamon, and M. Pouzet. Towards a higher-order synchronous dataflow language. In G. Buttazzo, editor, *International Conference on Embedded Software, EMSOFT'04*, pages 230–239, Pisa, Italy, September 2004. ACM.

[42] J.-L. Colaço and M. Pouzet. Clocks as first class abstract types. In R. Alur and I. Lee, editors, *International Conference on Embedded Software, EMSOFT'03*, volume 2855 of *LNCS*, pages 134–155, Philadelphia (PA), USA, October 2003. Springer-Verlag.

[43] A. Curic. *Implementing Lustre Programs on Distributed Platforms with Real-Time Constraints*. Thèse de doctorat, INPG, Grenoble, France, November 2005.

[44] D.M. Dhamdhere and H. Patil. An elimination algorithm for bidirectional data-flow problems unig edge placement. *ACM Trans. on Programming Languages and Systems*, 15(2) :312–336, April 1993.

[45] A. Dinning. A survey of synchronization methods for parallel computers. *IEEE Computer*, pages 66–76, July 1989.

[46] S.A. Edwards. *The Specification and Execution of Heterogeneous Synchronous Reactive System*. Thèse de doctorat, UC Berkeley, Berkeley (CA), USA, 1997.

[47] S.A. Edwards, V. Kapadia, and M. Halas. Compiling Esterel into static discrete-event code. In F. Maraninchi, A. Girault, and M. Pouzet, editors, *International Workshop on Synchronous Languages, Applications and Programs, SLAP'04*, volume 153(4) of *ENTCS*, Barcelona, Spain, March 2004.

[48] F. Giraud. Répartition automatique de programmes Esterel. Master's thesis, ISIMA, Clermont-Ferrand, France, September 2002.

[49] A. Girault. *Sur la Répartition de Programmes Synchrones*. Thèse de doctorat, INPG, Grenoble, France, January 1994.

[50] A. Girault. Elimination of redundant messages with a two-pass static analysis algorithm. In *9th Euromicro Workshop on Parallel and Distributed Processing, PDP'01*, pages 178–185, Mantova, Italy, February 2001.

[51] A. Girault. Elimination of redundant messages with a two-pass static analysis algorithm. *Parallel Computing*, 28(3) :433–453, March 2002.

[52] A. Girault. A survey of automatic distribution method for synchronous programs. In F. Maraninchi, M. Pouzet, and V. Roy, editors, *International Workshop on Synchronous Languages, Applications and Programs, SLAP'05*, ENTCS, Edinburgh, UK, April 2005. Elsevier Science.

[53] A. Girault and C. Ménier. *SCREP : A Tool to Produce Automatically GALS Systems from Synchronous Circuits*. Inria. User Manual. `http://pop-art.inrialpes.fr/~girault/Screp`.

[54] A. Girault and C. Ménier. Automatic production of globally asynchronous locally synchronous systems. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *International Workshop on Embedded Software, EMSOFT'02*, volume 2491 of *LNCS*, pages 266–281, Grenoble, France, October 2002. Springer-Verlag.

[55] A. Girault and X. Nicollin. Clock-driven automatic distribution of Lustre programs. In R. Alur and I. Lee, editors, *International Conference on Embedded Software, EMSOFT'03*, volume 2855 of *LNCS*, pages 206–222, Philadelphia (PA), USA, October 2003. Springer-Verlag.

[56] A. Girault, X. Nicollin, and M. Pouzet. Automatic rate desynchronization of embedded reactive programs. *ACM Trans. on Embedded Computing Systems*, 5(3) :687–717, August 2006.

[57] C. Gong, R. Gupta, and R. Melhem. Compiling techniques for optimizing communication on distributed-memory systems. In *International Conference on Parallel Processing, IPPS'93*, St. Charles, USA, August 1993.

[58] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *7th International Workshop on Hardware/Software Co-Design, CODES'99*, Rome, Italy, May 1999. ACM.

[59] T. Grandpierre and Y. Sorel. From algorithm and architecture specifications to automatic generation of distributed real-time executives : A seamless flow of graphs transformations. In *International Conference on Formal Methods and Models for Codesign, MEMOCODE'03*, Mont Saint-Michel, France, June 2003. IEEE.

[60] E.D. Granston and A.V. Veidenbaum. Detecting redundant accesses to array data. In *Supercomputing Conference*, Albuquerque (NM), USA, November 1991.

[61] M. Gupta, E. Schonberg, and H. Srinivasan. A unified framework for optimizing communication in data-parallel programs. *IEEE Trans. on Parallel and Distributed Systems*, 7(7) :689–704, July 1996.

[62] R. Gupta, S. Pande, K. Psarris, and V. Sarkar. Compilation techniques for parallel systems. *Parallel Computing*, 25(13) :1741–1783, 1999.

[63] O. Hainque. *Étude d'un environnement d'exécution temps-réel, distribué et tolérant aux pannes pour le modèle synchrone*. PhD thesis, ENST Paris, June 2000.

[64] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[65] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language Lustre. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991.

[66] N. Halbwachs, F. Lagnier, and C. Ratel. An experience in proving regular networks of processes by modular model checking. *Acta Informatica*, 29(6/7) :523–543, 1992.

[67] A. Hemani, T. Meincke, S. Kumar, A. Postula, T. Olsson, P. Nilsson, J. Oberg, P. Ellervee, and D. Lundqvist. Lowering power consumption in clock by using globally asynchronous locally synchronous design style. In *Design Automation Conference, DAC'99*, pages 873–878, New Orleans (LA), USA, June 1999. ACM/IEEE.

[68] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto : A time-triggered language for embedded programming. In *International Workshop on Embedded Software, EMSOFT'01*, volume 2211 of *LNCS*, Tahoe City (CA), USA, October 2001. Springer-Verlag.

[69] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto : A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91 :84–99, 2003.

[70] C.A.R. Hoare. Communicating sequential processes. *Communication of the ACM*, 21 :666–677, 1978.

[71] F. Huber, B. Schätz, and G. Einert. Consistent graphical specification of distributed systems. In *Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313 of *LNCS*, pages 122–141. Springer Verlag, 1997.

[72] M. Jones. What really happened on Mars ? Available on the web, December 1997. `http://research.microsoft.com/~mbj/Mars_Pathfinder`.

[73] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP Congress on Information Processing'74*, pages 471–475. North Holland, August 1974.

[74] K. Kennedy and N. Nedeljković. Combining dependence and data-flow analyses to optimize communication. In *9th International Parallel Processing Symposium, IPPS'95*, Santa Barbara (CA), USA, April 1995.

[75] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1) :112–126, October 2003.

[76] 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*, pages 1684–1689. Hermès, July 1991.

[77] C. Lavarenne and Y. Sorel. Modèle unifié pour la conception conjointe logiciel-matériel. *Traitement du Signal*, 14(6) :569–578, 1997.

[78] P. Le Guernic. Personnal communication, May 2003.

[79] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Lemaire. Programming real-time applications with Signal. *Proceedings of the IEEE*, 79(9) :1321–1336, September 1991.

[80] E.A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 17(12) :1217–1229, December 1998.

[81] J.K. Lenstra and A.H.G. Rinnoy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26(1) :22–35, 1978.

[82] O. Maffeïs. *Ordonnancements de graphes de flots synchrones ; Application à la mise en œuvre de Signal*. Thèse de doctorat, Université de Rennes I, Rennes, France, January 1993.

[83] F. Maraninchi and Y. Rémond. Argos : An automaton-based synchronous language. *Computer Languages*, 27(1–3) :61–92, April 2001.

[84] F. Maraninchi and Y. Rémond. Mode-automata : a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, 46(3) :219–254, 2003.

[85] J. Mitola. The software radio architecture. *IEEE Communications Magazine*, 33(5) :26–38, May 1995.

[86] J. Muttersbach, T. Villiger, and W. Fichtner. Practical design of globally asynchronous locally synchronous systems. In *Int. Symp. on Advanced Research in Asynchronous Circuits and Systems, ASYNC'00*, pages 52–61, Eilat, Israel, April 2000. IEEE.

[87] J.-P. Paris. *Exécution de tâches asynchrones depuis Esterel*. Thèse de doctorat, Université de Nice, Nice, France, 1992.

[88] F. Peix. *Distribution de programmes synchrones : le cas d'Esterel*. Thèse de doctorat, Université de Nice-Sophia Antipolis, Nice, France, July 2004.

[89] D. Potop-Butucaru. *Optimizations for Faster Simulation of Esterel Programs*. Thèse de doctorat, Ecole des Mines, Paris, France, November 2002.

[90] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. In *International Conference on Application of Concurrency to System Design, ACSD'04*, pages 67–78, Toronto, Canada, June 2004. IEEE. Irisa Research Report 1605.

[91] P. Raymond. Compilation séparée de programmes Lustre. Rapport de DEA, Université Joseph Fourier, Grenoble, France, June 1988. Research report Spectre L5.

[92] J. Romberg and A. Bauer. Loose synchronization of event-triggered networks for distribution of synchronous programs. In G. Buttazzo, editor, *International Conference on Embedded Software, EMSOFT'04*, Pisa, Italy, September 2004. ACM.

[93] T. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *European Design and Test Conference*, pages 328–333, Paris, France, March 1996.

[94] M. Sighireanu. The specification of the DC2SDX translator. Technical report. Unpublished. `http://www.liafa.jussieu.fr/~sighirea/DC2SDX/RR-DC2SDX.html`, March 1999.

[95] Y. Sorel. Massively parallel computing systems with real time constraints, the "algorithm architecture adequation" methodology. In *Massively Parallel Computing Systems Conference*, pages 44–53, Ischia, Italy, May 1994.

[96] J.-F. Susini, F. Boussinot, and L. Hazard. Distributed reactive machines. In *5th International Conference on Real-Time Computing Systems and Applications, RTCSA'98*, pages 267–274, Hiroshima, Japan, 1998. IEEE. Research Report INRIA 3376.

[97] The MathWorks, Inc. *Real-Time Workshop User's Guide, Version 3*, January 1999.

[98] R. von Hanxleden and K. Kennedy. GIVE-N-TAKE – A balanced code placement framework. In *Conference on Program Language Design and Implementation, PLDI'94*, pages 107–120. ACM SIGPLAN, March 1994.

[99] D. Weil, V. Bertin, E. Closse, M. Poize, P. Venier, and J. Pulou. Efficient compilation of Esterel for real-time embedded systems. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES'00*, pages 2–8, San Jose (CA), USA, November 2000. ACM.

[100] M. Zennaro and R. Sengupta. Distributing synchronous programs using bounded queues. In W. Wolf, editor, *International Conference on Embedded Software, EMSOFT'05*, pages 325–334, Jersey City (NJ), USA, September 2005. ACM.

Chapitre 3

Méthodes de conception pour systèmes répartis, embarqués et fiables

3.1 Avant propos : Le loup, la chèvre et le chevreau

En avant propos de ce chapitre, je désire citer la fable de Jean de la Fontaine « Le loup, la chèvre et le chevreau », qui illustre le besoin de tolérance aux fautes. La voici :

La Bique allant remplir sa traînante mamelle
Et paître l'herbe nouvelle,
Ferma sa porte au loquet,
Non sans dire à son Biquet :
Gardez-vous sur votre vie
D'ouvrir que l'on ne vous die,
Pour enseigne et mot du guet :
Foin du Loup et de sa race !
Comme elle disait ces mots,
Le Loup de fortune passe ;
Il les recueille à propos,
Et les garde en sa mémoire.
La Bique, comme on peut croire,
N'avait pas vu le glouton.
Dès qu'il la voit partie, il contrefait son ton,
Et d'une voix papelarde
Il demande qu'on ouvre, en disant Foin du Loup,
Et croyant entrer tout d'un coup.
Le Biquet soupçonneux par la fente regarde.
Montrez-moi patte blanche, ou je n'ouvrirai point,
S'écria-t-il d'abord. (Patte blanche est un point
Chez les Loups, comme on sait, rarement en usage.)
Celui-ci, fort surpris d'entendre ce langage,
Comme il était venu s'en retourna chez soi.
Où serait le Biquet s'il eût ajouté foi
Au mot du guet, que de fortune
Notre Loup avait entendu ?
Deux sûretés valent mieux qu'une,
Et le trop en cela ne fut jamais perdu.

Comme souvent, c'est la morale de la fable qui est importante, ici les deux derniers vers. Je trouve passionnant et amusant qu'au XVII^{e} siècle, Jean de la Fontaine ait écrit une fable qui soit à ce point d'actualité (et j'aime aussi beaucoup la « voix papelarde »).

3.2 Introduction

3.2.1 La sûreté de fonctionnement informatique

La **sûreté de fonctionnement** (« dependability ») d'un système informatique est son aptitude à délivrer un service de confiance justifiée [73]. Voici quelques articles et livres célèbres qui traitent de ce sujet : [81] parle déjà de défaillance et de fiabilité, [7] semble être la première référence à la notion de tolérance aux fautes (une des composantes de la sûreté de fonctionnement), [75] présente exhaustivement les concepts et la terminologie en sûreté de fonctionnement, [98] donne un panorama et une taxonomie des systèmes temps-réel sûrs de fonctionnement, et enfin [25, 53, 35] traitent en détails le cas spécifique de la tolérance aux fautes dans les systèmes répartis. Plus récemment, [8] et [73] présentent une mise à jour des concepts et de la terminologie en sûreté de fonctionnement informatique, tout en mettant l'accent sur les entraves et les attributs. L'introduction de ce chapitre est un condensé de ces différents ouvrages.

Selon le point de vue adopté, la sûreté de fonctionnement se décline en :

la disponibilité (« availability ») qui est le fait d'être prêt à l'utilisation ;

la fiabilité (« reliability ») qui mesure la continuité de service ;

la sécurité-innocuité (« safety ») qui est l'absence de conséquences catastrophiques pour l'environnement ;

la confidentialité (« confidentiality ») qui est l'absence de divulgations non autorisées d'informations ;

l'intégrité (« integrity ») qui est l'absence d'altérations inappropriées de l'information ;

la maintenabilité (« maintainability ») qui est l'aptitude aux réparations et aux évolutions.

La sûreté de fonctionnement est-elle vraiment nécessaire pour les systèmes informatiques ? La réponse est clairement « Oui ». Pour s'en convaincre, il suffit de consulter les deux pages web suivantes, qui recensent des cas célèbres de fautes informatiques ayant entraîné des catastrophes, depuis l'explosion de la fusée Ariane 5 jusqu'aux erreurs de division en virgule flottante du processeur Pentium : `http://www5.in.tum.de/~huckle/bugse.html` et `http://www.qucis.queensu.ca/Software-Engineering/archive/horror`. On peut aussi considérer les cas, encore plus nombreux, de systèmes informatiques qui fonctionnent correctement, en dépit des inévitables défaillances matérielles et des « bugs » : des bases de données bancaires aux systèmes de contrôle/commande des avions, en passant par le réseau de télécommunication planétaire (à la fois filaire, hertzien et satelitaire).

3.2.2 Entraves à la sûreté de fonctionnement

On distingue trois **entraves** à la sûreté de fonctionnement : la faute, l'erreur et la défaillance. Une **faute** est un défaut dans un composant du système informatique (composant matériel ou logiciel ; par exemple, un « bug » est une faute de conception). Une **erreur** est une manifestation ou activation d'une faute. Une **défaillance** est un événement qui survient lorsque le service délivré dévie du service correct, suite à la propagation d'une erreur dans le système. Un **service correct** est délivré par un système lorsqu'il accomplit sa fonction. Enfin, la conséquence de la défaillance d'un sous-système constitue une faute pour le système englobant. Il en résulte la chaîne de causalité suivante entre les entraves à la sûreté de fonctionnement :

$$\cdots \longrightarrow \text{faute} \xrightarrow{\textit{activation}} \text{erreur} \xrightarrow{\textit{propagation}} \text{défaillance} \xrightarrow{\textit{causalité}} \text{faute} \longrightarrow \cdots$$

Le terme **panne** existe également et est très populaire, notamment dans l'expression **tolérance aux pannes**. Cependant, les spécialistes de la sûreté de fonctionnement le trouvent mal défini et préfèrent utiliser plutôt celui de défaillance (quand ou veut dire que le système ne délivre plus le service attendu) ou celui de faute (quand on cherche la raison pour laquelle le système est défaillant).

Dans un système informatique, trois principales classes de fautes sont à envisager : les fautes matérielles, les fautes logicielles et les fautes d'interaction. Les fautes **matérielles** affectent les composants matériels du système : processeurs, mémoires, capteurs, actionneurs, media de communication... Les fautes **logicielles** affectent les composants logiciels du système et sont communément appelées « bugs ». Enfin, les fautes **d'interaction** concernent les interactions entre le système et son environnement.

Les fautes sont classifiées selon les critères suivants :

leur nature : accidentelle ou intentionnelle ;

leur origine : physique, humaine, interne, externe, de conception, opérationnelle ;

leur persistance : permanente ou temporaire.

Par exemple, toutes les fautes n'entraînent pas une défaillance immédiatement. Certaines sont latentes, c'est-à-dire qu'elles sont activées mais n'ont pas d'effet visible sur le service rendu par le système (elles restent donc à l'état d'erreur). Le but de la tolérance aux fautes est justement d'essayer de détecter et de corriger les erreurs latentes avant qu'elles ne deviennent effectives.

Quant aux défaillances, elles sont classifiées selon les critères suivants :

leur domaine : en valeur ou temporelles ;

leur détectabilité par l'utilisateur ;

leur cohérence vis-à-vis de l'ensemble des utilisateurs ;

leurs conséquences pour l'environnement du système.

En particulier, les systèmes dont toutes les défaillances sont temporelles *et* permanentes sont appelés **systèmes à silence sur défaillance** (« fail-silent » en anglais). Ces systèmes sont relativement faciles à étudier en raison de la nature même de leurs défaillances (qu'on appelle aussi « crash »). À l'inverse, les systèmes ayant des défaillances en valeur et incohérentes sont beaucoup plus difficile à étudier. Suite aux travaux de Lamport et al., les défaillances en valeurs incohérentes sont communément appelées **défaillances byzantines** [72]. Dans les applications critiques, l'hypothèse de silence sur défaillance a longtemps été considérée, et à juste titre, comme étant trop restrictive pour des composants non-dotés de mécanismes d'autotest [90], ce qui est le cas pour la très grande majorité des composants sur étagère. Ainsi, Brasileiro et al. ont proposé une mise en œuvre logicielle de nœuds à silence sur défaillance avec deux processeurs [16] ; selon le type de protocole de communication utilisé, la perte de performances par rapport à un nœud monoprocesseur varie de 80% à 60%. Toutefois, une étude récente de Baleani et al. propose des architectures de système-sur-puce (« system-on-chip » en anglais) à silence sur défaillance [11] ; ces circuits ont un surcoût acceptable par rapport à des circuits usuels, ce qui en fait de bons candidats pour être vendus sur étagère.

3.2.3 Moyens de la sûreté de fonctionnement

Le développement d'un système sûr de fonctionnement repose sur l'utilisation combinée de plusieurs **moyens** :

la prévention des fautes qui sert à empêcher l'occurrence ou l'introduction de fautes ;

la tolérance aux fautes qui sert à fournir un service même en présence de fautes ;

l'élimination des fautes qui sert à réduire le nombre et la sévérité des fautes ;

la prévision des fautes qui sert à estimer la présence, le taux futur et les conséquences possibles des fautes.

Parmi ces divers moyens, je me suis quant à moi concentré sur la tolérance aux fautes. Il existe pour cela deux types de moyens :

Le traitement d'erreur consiste à éliminer les erreurs, si possible avant qu'une défaillance ne survienne ; il peut être réalisé ou bien par le **recouvrement** d'erreur (reprise à partir d'un état stable et correct du système), ou bien par la **compensation** d'erreur (masquage des erreurs grâce à la redondance interne du système).

Le traitement de faute consiste à éviter qu'une faute ne soit activée à nouveau, et comporte deux étapes : le **diagnostic** (détermination de la cause, nature et localisation de la faute) suivi de la **passivation** (empêcher la faute d'être à nouveau activée).

Afin de traiter les erreurs, ces différents moyens utilisent la **redondance**, dont il existe deux formes générales : d'une part la **redondance matérielle** consiste à utiliser la redondance entre les composants matériels de l'architecture, voire à ajouter un ou plusieurs composants supplémentaires, et d'autre part la **redondance logicielle** consiste à exécuter plusieurs fois un ou plusieurs modules logiciels. Ces copies d'un module logiciel peuvent être identiques ou distinctes, et dans ce dernier cas obtenues par **programmation N-version** à partir d'une même spécification [74]. Dans le premier cas, la redondance ne permet de traiter que les erreurs dues aux fautes matérielles, alors que dans le second cas, la redondance permet de traiter aussi les erreurs dues aux fautes logicielles.

Il faut donc distinguer les techniques matérielles, qui accroissent la redondance matérielle du système pour tolérer des fautes matérielles, et les techniques logicielles, qui accroissent la redondance logicielle du système pour tolérer des fautes matérielles ou logicielles, en utilisant si besoin la redondance matérielle intrinsèque offerte par l'architecture du système. Les techniques logicielles utilisent diverses formes de **réplication** pour augmenter la redondance : ce sont d'une part la **réplication active** qui consiste à exécuter plusieurs copies d'un même module logiciel en même temps sur plusieurs processeurs, et d'autre part en la **réplication passive** qui consiste à ré-exécuter un module logiciel sur un autre processeur après qu'une défaillance soit détectée. La réplication active s'appelle aussi la « state machine approach » et la réplication passive s'appelle aussi la « primary-backup approach » [103, 102].

3.2.4 Notion probabiliste et quantitative : la fiabilité

J'ai introduit dans la section 3.2.1 la notion de **fiabilité** comme mesurant la continuité de service. Elle permet de mesurer le niveau de sûreté de fonctionnement d'un système : elle peut donc être définie comme étant la probabilité qu'il fonctionne correctement pendant un intervalle de temps donné. Comme toute probabilité, son intervalle de valeur est $[0, 1]$. Par exemple, dans l'avionique civile, on exige du système gérant les commandes de vol que sa fiabilité soit supérieure à 0.999999999 pendant 10 heures (règle des « neuf neufs ») [89]. De façon duale, la **probabilité de défaillance** d'un système est égale à 1 moins sa fiabilité.

Il existe plusieurs façons de mesurer la fiabilité d'un système, et je reviendrai dessus dans la section 3.4.5. Quelle que soit la façon utilisée pour mesurer la fiabilité d'un système, il est *toujours* utile d'améliorer celle-ci. La tolérance aux fautes est justement un des moyens permettant d'améliorer la fiabilité d'un système.

Notons enfin que la fiabilité n'est pas un *moyen* de la tolérance aux fautes mais une *mesure*. Néanmoins, je montrerai dans la section 3.4.5 qu'une telle mesure peut parfaitement être utilisée comme guide afin d'améliorer la sûreté de fonctionnement d'un système, donc, d'un certain point de vue, comme un moyen.

3.2.5 Hypothèse de faute et couverture

Enfin, quand on conçoit des systèmes tolérants aux fautes, il est impératif de spécifier clairement, d'une part les **hypothèses de fautes**, c'est-à-dire les hypothèses sur la nature des fautes que l'on veut que

le système tolère et sur la manifestation de ces fautes sur le comportement des composants défaillants, et d'autre part la **couverture de ces hypothèses**, qui est la probabilité que ces hypothèses soient satisfaites par le système physique réel quand une faute se produit [89]. Les hypothèses de fautes doivent porter sur toutes les caractéristiques des fautes, qu'elles soient logicielles, matérielles ou d'interaction : on doit donc spécifier l'hypothèse sur la nature des fautes (accidentelle ou intentionnelle), l'hypothèse sur l'origine des fautes (physique, humaine, interne, externe, de conception, opérationnelle) et l'hypothèse sur la persistance des fautes (permanente ou temporaire). De plus, dans le cas où les fautes peuvent affecter plusieurs composants du système (composants logiciels ou matériels), il faut spécifier l'hypothèse d'indépendance des fautes.

Concernant les hypothèses sur le comportement des composants défaillants (c'est-à-dire, leurs modes de défaillance), celles-ci peuvent se formaliser en termes du service défini comme une séquence d'**éléments de service** $(s_i)_{i\in\mathbb{N}}$, chacun défini comme un couple (vs_i, ts_i), où vs_i et ts_i sont respectivement la valeur et l'instant de s_i. De plus, SV_i et ST_i sont respectivement l'ensemble des valeurs et des instants possibles de s_i. Cela permet de définir des classes de service incorrect : erreurs en valeur arbitraires, erreurs de code, erreurs temporelles arbitraires, erreurs d'omission... Puis, une **hypothèse de mode de défaillance** est définie comme étant une **assertion logique** : par exemple, l'assertion $\forall i, (ts_i \in ST_i) \vee (ts_i = +\infty)$ est celle des erreurs d'omission. Les implications logiques entre ces assertions induisent une relation d'ordre partiel entre les modes de défaillance. Cet ordre partiel $\rightarrow$ implique que si les mécanismes de tolérance aux fautes d'un système permettent de traiter les erreurs définies par l'assertion Y, alors ces mêmes mécanismes peuvent aussi traiter les erreurs définies par l'assertion X si $X \rightarrow Y$.

Enfin, la **couverture d'hypothèse** d'un mode de défaillance est définie comme la probabilité p_X que l'assertion X définissant le comportement supposé d'un composant soit vraie en pratique, conditionnée pas le fait que le composant soit défaillant : $p_X = Pr\{X = true|\text{composant défaillant}\}$. En particulier, la couverture de l'hypothèse d'erreurs arbitraires en valeur et en temps est égale à 1, alors que la couverture de l'hypothèse qu'un composant défaillant de cause aucune erreur est égale à 0. De façon similaire, la **couverture des mécanismes de tolérance aux fautes** destinés à traiter les erreurs de l'hypothèse X est $Pr\{\text{traitement d'erreur correct}|X = true\}$. Par conséquent, la couverture globale est le produit $Pr\{\text{traitement d'erreur correct}|X = true\} \times Pr\{X = true|\text{composant défaillant}\}$.

3.3 Préliminaires

3.3.1 Contexte

Mes travaux sur la tolérance aux fautes ont commencé en 1999 dans le cadre de l'Action de Recherche Coordonnée TOLÈRE [1], que j'ai animée. J'ai encadré et co-encadré quatre chercheurs post-doctorants, Mihaela Sighireanu (aujourd'hui maître de conférences à l'université de Paris VII), Cătălin Dima (aujourd'hui maître de conférences à l'université de Paris XII Créteil), Emil Dumitrescu (aujourd'hui maître de conférences à l'INSA de Lyon) et Tolga Ayav (aujourd'hui professeur assistant à l'Université d'Izmir). De plus, dans le cadre du projet EAST-EEA [2] portant sur les fonctionnalités électroniques embarquées dans l'automobile, j'ai co-encadré avec Yves Sorel un étudiant en thèse, Hamoudi Kalla (aujourd'hui post-doctorant à l'IRISA Rennes). J'ai en outre encadré et co-encadré de nombreux stagiaires de DEA/M2R et ingénieurs : Claudio Pinello, Ismail Assayad, Mohamed Abdennebi, Nicolas Leignel, Safouan Taha, Huafeng Yu, Thomas Lévêque, Érik Saule, Nour Brinis et Sumit Kumar.

Notre objectif dans TOLÈRE était de proposer de nouvelles méthodes pour la conception des **systèmes informatiques répartis, embarqués et tolérants aux fautes**. Répartis implique que leurs architectures matérielles sont des architectures multiprocesseur à mémoires réparties. Embarqués implique

[1] TOLÈRE : `http://pop-art.inrialpes.fr/~girault/Projets/Incitative`.
[2] EEA = « Embedded Electronic Architecture » : `http://www.east-eea.net`.

que les ressources matérielles dont ils disposent sont limitées. Ces ressources sont les processeurs, les mémoires, les media de communication, les capteurs, les actionneurs... Enfin, deux types de fautes sont à considérer : les fautes matérielles et les fautes logicielles. Partant du constat qu'il existe un vaste éventail de méthodes visant à supprimer les fautes logicielles dans un système — vérification de conformité entre la spécification et l'implémentation, vérification de propriétés, preuve formelle d'invariants, génération automatique de tests, interprétation abstraite — nous avons décidé dès le début de nous concentrer exclusivement sur les fautes matérielles. Certes, ces méthodes ne garantissent en aucun cas que *toutes* les fautes logicielles seront évitées. Mais il est communément admis qu'elles permettent d'en supprimer une très grande partie. Ce choix nous a donc paru raisonnable.

Je veux insister dès à présent sur l'antagonisme de moyen existant entre, d'une part les caractéristiques des systèmes répartis et embarqués, et d'autre part le besoin de tolérance aux fautes : en effet, comme je l'ai dit dans la section 3.2.3, les moyens utilisés pour la tolérance aux fautes utilisent la redondance. Or, accroître la redondance matérielle dans un système va à l'encontre de la contrainte sur la limitation des ressources disponibles. C'est pourquoi il m'a semblé naturel d'opter pour la redondance logicielle plutôt que pour la redondance matérielle. L'inconvénient de ce choix est un surcoût en temps d'exécution puisqu'il faut exécuter des modules logiciels supplémentaires sans augmenter les ressources matérielles. S'il est trop important, ce surcoût peut être pénalisant, puisque les systèmes visés sont en plus soumis à des contraintes temps-réel. Il conviendra donc, d'une part, d'évaluer le surcoût du à la tolérance aux fautes, et d'autre part, de toujours viser à le minimiser en utilisant au mieux les composants matériels de l'architecture cible.

3.3.2 Résumé de mes contributions

Je présente dans la suite de cette section les modèles et outils utilisés dans la suite de ce chapitre. Puis je présente mes contributions au domaine de la tolérance aux fautes dans les sections 3.4, 3.5 et 3.6 :

- La section 3.4 traite de trois méthodes permettant de générer automatiquement des ordonnancements multiprocesseur de graphes d'algorithmes flot-de-données sur des architectures réparties et hétérogènes. Les ordonnancements ainsi générés doivent être à la fois tolérants à un nombre donné de fautes de processeurs et à un nombre donné de fautes de media de communication, ces deux nombres étant fixés par l'utilisateur. La sous-section 3.4.2 traite du cas particulier des architectures avec liens de communication point-à-point. La sous-section 3.4.3 traite du cas particulier des architectures avec bus de communication. Enfin, la sous-section 3.4.5 traite un besoin légèrement différent puisqu'il s'agit de générer automatiquement des ordonnancements multiprocesseur dont la fiabilité soit supérieure à une borne fixée par l'utilisateur (et non pas tolérants à un nombre de fautes donné).
- Puis, la section 3.5 traite d'une méthode originale, proposée par Cătălin Dima dans le cadre de son postdoc, et dont l'objectif est également de générer automatiquement des ordonnancements multiprocesseur de graphes d'algorithme flot-de-données sur des architectures réparties et hétérogènes. Mais ici, les fautes à tolérées sont spécifiées par l'utilisateur sous la forme de patrons de fautes, qui permettent de prendre en compte, dans un même modèle, les fautes de tous les composants matériels de l'architecture (processeurs et media de communication, qu'ils soient de type point-à-point ou multipoint), et également de spécifier exactement quels groupes de composants matériels peuvent défaillir simultanément.
- Enfin, la section 3.6 traite d'une méthode originale utilisant la synthèse de contrôleurs discrets afin d'obtenir automatiquement des systèmes tolérants aux fautes. Ici, les fautes à tolérer sont spécifiées par l'utilisateur sous la forme d'une propriété qui est utilisée par un outil de synthèse de contrôleurs. Qui plus est, l'occurrence d'une faute est un événement incontrôlable. Trois applications de la synthèse de contrôleurs sont étudiées : pour tolérer des fautes de type « crash » de processeurs, pour tolérer des fautes en valeurs de capteurs, et pour résoudre le problème des généraux byzantins.

3.3.3 Modèles utilisés

J'introduis ici tous les modèles utilisés par la suite. Les systèmes sur lesquels je travaille sont des systèmes informatiques répartis et embarqués. Ils sont modélisés par, d'une part une spécification d'algorithme représentée par un **graphe flot-de-données** $\mathcal{A}lg$, et d'autre part une spécification d'architecture répartie cible, représentée par un **graphe biparti non-orienté** $\mathcal{A}rc$.

Le graphe $\mathcal{A}lg$ est un **graphe orienté sans cycle**. Ses sommets sont des blocs logiciels appelés **opérations**. La plupart sont sans effet de bord, à l'exception des opérations d'entrée/sortie : une opération d'entrée est un appel à un ou plusieurs pilotes de capteurs physiques, tandis qu'une opération de sortie est un appel à un ou plusieurs pilotes d'actionneurs physiques. Chacun des arcs de $\mathcal{A}lg$ est une **dépendance de données** entre deux opérations (raison pour laquelle le graphe $\mathcal{A}lg$ est appelé graphe flot-de-données). Si X$\triangleright$Y est une dépendance de données, alors X est une opération **prédécesseur** de l'opération Y, tandis que Y est une opération **successeur** de l'opération X. L'opération X est aussi appelée la **source** de la dépendance de données X$\triangleright$Y, et Y est sa **destination**. L'ensemble des prédécesseurs de X est noté $pred(\mathrm{X})$. L'ensemble des successeurs de X est noté $succ(\mathrm{X})$. Le graphe $\mathcal{A}lg$ induit donc une **relation d'ordre partiel** entre les opérations ; cet ordre partiel exprime le parallélisme potentiel de la spécification d'algorithme.

Le graphe $\mathcal{A}lg$ est acyclique — raison pour laquelle je dis que c'est un DAG (« Directed Acyclic Graph » en anglais) —- et il est de plus répété infiniment afin de rendre compte du caractère réactif du système modélisé. Ce modèle est très semblable à la boucle périodique d'exécution que j'ai déjà présentée dans la section 2.1.4. Autrement dit, le graphe $\mathcal{A}lg$ modélise l'intérieur de la boucle d'exécution (phases `read inputs`, `react and compute outputs` et `compute next state`). Chaque exécution du graphe $\mathcal{A}lg$ s'appelle une **itération**.

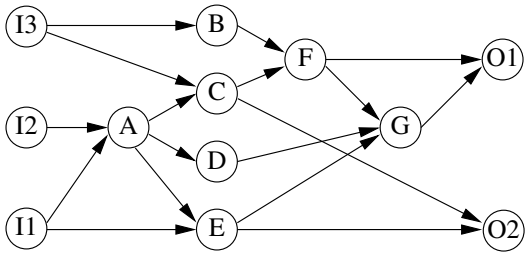


FIG. 3.1 – Un exemple de graphe d'algorithme $\mathcal{A}lg$.

La figure 3.1 représente un exemple de graphe d'algorithme. I1, I2 et I3 sont des opérations d'entrée car elles n'ont aucun prédécesseur. O1 et O2 sont des opérations de sortie car elles n'ont aucun successeur. A, B, C, D, E, F et G sont des opérations de calcul.

Le graphe d'architecture $\mathcal{A}rc$ est **graphe biparti non-orienté** $(\mathcal{P}, \mathcal{M}, \mathcal{A})$ dont l'ensemble des sommets est $\mathcal{P} \cup \mathcal{M}$ et l'ensemble des arêtes est $\mathcal{A}$; $\mathcal{P}$ est l'ensemble des processeurs et $\mathcal{M}$ est l'ensemble des media de communication. Le **degré** d'un sommet est le nombre d'arêtes auquel il est relié. Un **processeur** est constitué d'une unité de calcul, lui permettant de séquencer les opérations, et d'une ou plusieurs unités de communication, lui permettant de séquencer les communications. Chacune des arêtes de $\mathcal{A}rc$ relient forcément un processeur et un medium de communication. Dans le cas d'un **lien de communication point-à-point**, le sommet représentant ce medium est connecté à exactement deux processeurs (il est donc de degré 2), alors que dans le cas d'un **bus de communication**, le sommet représentant ce medium est connecté à plusieurs processeurs (il est donc de degré $n \geq 2$). Un **chemin** entre les processeurs P1 et Pn est une succession P1-A1-M1-A2-P2-A3-M2-A4-$\cdots$-Pn, telle que Pi$\in\mathcal{P}$, Mi$\in\mathcal{M}$, Ai$\in\mathcal{A}$, A1=(P1,M1), A2=(M1,P2), A3=(P2,M2), A4=(M2,P3)... (formellement, c'est un sous-graphe connexe de degré compris entre 1 et 2). Le graphe $\mathcal{A}rc$ est **connexe** s'il existe un chemin entre toute paire de

sommets ; cette notion est définie normalement pour les graphes non-bipartis et elle s'applique au cas particulier des graphes bipartis ; ce qui m'intéresse ici est que, dans un graphe $\mathcal{A}rc$ connexe, il existe un chemin entre toute paire de processeurs. Par analogie avec les graphes non-bipartis, le graphe $\mathcal{A}rc$ est **complet** s'il existe un lien de communication entre toute paire de processeurs (et non pas une arête entre toute paire de sommets). En parlant de l'architecture, je dirai qu'elle est **complètement connectée** si son graphe $\mathcal{A}rc$ est complet. Enfin, l'architecture est dite **homogène** si tous ses processeurs et tous ses media de communication ont les mêmes caractéristiques (vitesse, mémoire, fiabilité, bande passante...), et **hétérogène** sinon.

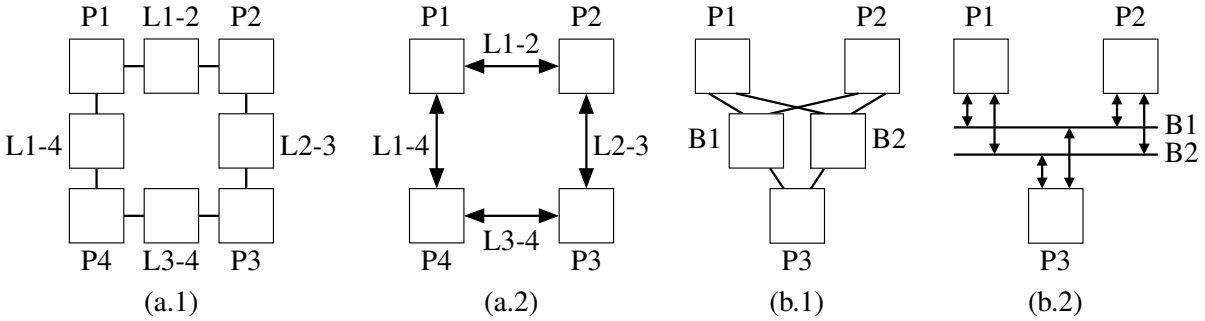


FIG. 3.2 – Deux exemples de graphe d'architecture $\mathcal{A}rc$: (a.1) avec liens de communication point-à-point en représentation bipartie et (a.2) en représentation classique ; (b.1) avec bus de communication en représentation bipartie et (b.2) en représentation classique.

La figure 3.2(a) représente un exemple de graphe d'architecture avec quatre processeurs, P1, P2, P3 et P4, et quatre liens de communication point-à-point, L1-2, L2-3, L3-4 et L1-4. La figure 3.2(b) représente un second exemple de graphe d'architecture avec trois processeurs, P1, P2 et P3, et deux bus de communication, B1 et B2. La représentation bipartie est donnée à côté de la représentation classique, qui sera elle utilisée dans le reste de ce chapitre.

3.3.4 Problème d'ordonnancement & répartition

Les deux graphes $\mathcal{A}lg$ et $\mathcal{A}rc$ constituent la **spécification** du système. Son **implantation** consiste à trouver un **ordonnancement multiprocesseur** de $\mathcal{A}lg$ sur $\mathcal{A}rc$, constitué d'une part d'une fonction d'affectation spatiale et d'autre part d'une fonction d'affectation temporelle. La **fonction d'affectation spatiale** donne, pour chaque opération de $\mathcal{A}lg$ (resp. chaque dépendance de données), le processeur de $\mathcal{A}rc$ (resp. le medium de communication dans le cas d'une communication inter-processeur, ou le processeur dans le cas d'une communication intra-processeur) qui est chargé de l'exécuter. Quant à la **fonction d'affectation temporelle**, elle donne la date de début d'exécution de chaque opération (resp. chaque dépendance de données) sur son processeur (resp. sur son medium de communication). En général, pour deux graphes $\mathcal{A}lg$ et $\mathcal{A}rc$ donnés, il existe plusieurs ordonnancements multiprocesseur possibles.

Il existe deux grandes classes d'ordonnancement : les ordonnancements statiques et les ordonnancements dynamiques. Un ordonnancement d'un graphe d'algorithme $\mathcal{A}lg$ sur un graphe d'architecture $\mathcal{A}rc$ est **statique** si, pour chaque processeur de $\mathcal{A}rc$ (resp. chaque medium de communication), toutes les opérations (resp. toutes les dépendances de données) qui lui sont affectées sont statiquement ordonnées. Autrement dit, la fonction d'affectation temporelle est statique. Dans le cas contraire, l'ordonnancement est **dynamique**. On trouve également dans la littérature la classification, à mon avis équivalente, entre ordonnancement hors-ligne et en-ligne : dans un ordonnancement **hors-ligne**, toutes les décisions sont prises avant l'exécution de l'ordonnancement, alors que dans un ordonnancement **en-ligne**, une partie des décisions est prise pendant l'exécution de l'ordonnancement. Généralement, un ordonnance-

ment hors-ligne n'est possible que si tous les paramètres sont connus avant l'exécution (en particulier le nombre et le type des tâches, leurs temps d'exécution...).

Les systèmes considérés sont soumis à des contraintes temps-réel, matérialisées par le fait que l'ordonnancement de $\mathcal{A}lg$ sur $\mathcal{A}rc$ doit avoir une longueur inférieure à une borne $\mathcal{L}_{obj}$ imposée par l'utilisateur. La **longueur** d'un ordonnancement est définie comme étant le $\max$ des dates de terminaison de toutes les opérations ordonnancées sur tous les processeurs de $\mathcal{A}rc$. Dans la litérature, on trouve les termes équivalents de makespan et chemin critique ; cette longueur est notée $\mathsf{C}_{\mathsf{max}}$:

$$\mathsf{C}_{\mathsf{max}} = \max_i \{C_i\} \tag{3.1}$$

où C_i est la date de terminaison de l'opération numéro i.

Afin de calculer la longueur des ordonnancements, l'utilisateur fournit les **temps d'exécution maximaux** des sommets de $\mathcal{A}lg$ sur les processeurs de $\mathcal{A}rc$ (noté WCET, acronyme de « Worst Case Execution Time ») et les **temps de transmission maximaux** des arêtes de $\mathcal{A}lg$ sur les media de communication de $\mathcal{A}rc$ (noté WCTT, acronyme de « Worst Case Transmission Time ») : ces temps sont exprimés dans un tableau $\mathcal{E}xe$. La case $\langle \mathrm{X}, \mathrm{P} \rangle$ de ce tableau est égale au WCET de l'opération X sur le processeur P, tandis que la case $\langle \mathrm{X} \triangleright \mathrm{Y}, \mathrm{M} \rangle$ est égale au WCTT de la dépendance de données X$\triangleright$Y sur le medium M (que ce soit un lien point-à-point ou un bus). Dans le cas d'une architecture homogène, pour tous X, P et P', on a $\mathcal{E}xe\langle \mathrm{X}, \mathrm{P} \rangle = \mathcal{E}xe\langle \mathrm{X}, \mathrm{P'} \rangle$, et de même, pour tout X$\triangleright$Y, M et M', on a $\mathcal{E}xe\langle \mathrm{X} \triangleright \mathrm{Y}, \mathrm{M} \rangle = \mathcal{E}xe\langle \mathrm{X} \triangleright \mathrm{Y}, \mathrm{M'} \rangle$. Dans le cas d'une architecture hétérogène, cela n'est pas vrai.

La contrainte temps-réel $\mathcal{L}_{obj}$ est beaucoup plus facile à vérifier si l'ordonnancement est statique que s'il est dynamique. D'ailleurs, un ordonnancement statique est également appelé **prédictif**. Aussi, dès le début de l'ARC TOLÈRE, il a été décidé de ne considérer que les ordonnancements statiques. D'une façon générale pour les systèmes temps-réels, il y a débat entre les partisans des ordonnancements statiques et ceux des ordonnancements dynamiques. Les inconvénients des ordonnancements dynamiques proviennent de la nécessité d'exécuter l'ordonnancement au-dessus un **système d'exploitation temps-réel** (RTOS) permettant la préemption. Plusieurs problèmes graves peuvent en résulter : l'inversion de priorité [106, 122], le fait qu'une tâche qui prend moins de temps pour s'exécuter que prévu peut faire rater son échéance à une autre tâche (connu sous le nom d'anomalies de Richards [96, 82]), et plus généralement tout ce qui découle de la non-prédictabilité de l'ordonnancement. En outre, le fait d'avoir un RTOS résident induit un surcoût qui est difficile à mesurer. Les avantages sont de permettre au système d'adapter son comportement aux évolutions imprévues de l'environnement, comme par exemple une modification de la charge des processeurs, une tâche supplémentaire imprévue ou une défaillance. Un ordonnancement statique est préférable pour des systèmes périodiques échantillonnés ainsi que pour tous les systèmes où l'utilisation d'un RTOS est proscrite en raison de leur très grande criticité (par exemple les commandes de vol d'un avion). En revanche, un ordonnancement dynamique est préférable pour des systèmes non-prédictibles (par exemple le suivi de cibles pour un avion de chasse), encore que certains défendent l'utilisation d'ordonnancements statiques y compris pour ce type de systèmes [120].

En pratique, afin de garantir qu'un ordonnancement statique respecte la contrainte $\mathcal{L}_{obj}$, on le construit en visant à minimiser sa longueur, et une fois entièrement construit on vérifie si la contrainte $\mathcal{L}_{obj}$ est satisfaite.

De plus, l'utilisateur a la possibilité de spécifier des **contraintes de répartition** $\mathcal{D}is$, qui reflètent les caractéristiques des composants matériels de l'architecture et des blocs logiciels de l'algorithme. Par exemple, une certaine opération X de l'algorithme $\mathcal{A}lg$ peut nécessiter, pour son exécution, un modèle spécifique de processeur, et donc ne pas pouvoir être exécutée sur certains processeurs de $\mathcal{A}rc$. Ceci est matérialisé par des cases $\mathcal{E}xe\langle \mathrm{X}, \mathrm{P} \rangle$ contenant la valeur ∞.

En résumé, le problème de l'ordonnancement & répartition peut donc être exprimé de la manière suivante :

Problème 3.1 (AAA) *Soit un graphe d'algorithme $\mathcal{A}lg$, un graphe d'architecture $\mathcal{A}rc$, un tableau $\mathcal{E}xe$*

des WCET des opérations de $\mathcal{A}lg$ sur les processeurs de $\mathcal{A}rc$ et des WCTT des dépendances de données de $\mathcal{A}lg$ sur les liens de communication de $\mathcal{A}rc$, des contraintes de distribution $\mathcal{D}is$.

Trouver l'ordonnancement multiprocesseur et statique de $\mathcal{A}lg$ sur $\mathcal{A}rc$, qui respecte $\mathcal{D}is$, et dont la longueur calculée grâce à $\mathcal{E}xe$ soit minimale.

Comme je l'ai dit plus haut, il faut bien souvent trouver en fait un ordonnancement multiprocesseur et statique de $\mathcal{A}lg$ sur $\mathcal{A}rc$, qui respecte $\mathcal{D}is$, et dont la longueur calculée grâce à $\mathcal{E}xe$ soit *inférieure à* $\mathcal{L}_{obj}$ (et non pas *minimale*) ; mais cela ne rend pas le problème plus facile. En pratique, la méthode usuelle de résolution de ce problème consiste, dans un premier temps à résoudre le problème 3.1, donc sans tenir compte de la contrainte $\mathcal{L}_{obj}$, puis dans un second temps à vérifier que la longueur de l'ordonnancement ainsi obtenu est inférieure à $\mathcal{L}_{obj}$. C'est exactement ce que fait la méthode AAA que je présente à la section 3.3.6.

Les recherches que j'ai entreprises en 1999 ont consisté à trouver des méthodes permettant de résoudre une variante de ce problème, en ajoutant une exigence de tolérance aux fautes et/ou de fiabilité. Je présente mes résultats dans les sections 3.4, 3.5 et 3.6. Les méthodes que je présente sont toutes basées sur des algorithmes heuristiques puisque le problème est NP-difficile. Pour la plupart, ces méthodes sont basées sur la méthode AAA et l'outil de conception SYNDEX. Je les présente tous les deux dans la section suivante.

3.3.5 Complexité

Dans la classification des problèmes d'ordonnancement introduite par Graham et al. [45], le problème 3.1 est noté `Rm|prec|Cmax` : en effet, l'architecture est composée de `m` processeurs quelconques, les dépendances de données du graphe $\mathcal{A}lg$ induisent une relation de précédence entre les tâches, et le critère à optimiser est le `Cmax` de l'ordonnancement. Il se trouve que le problème `P2||Cmax`, où `P2` signifie que l'architecture est composée de deux processeurs identiques, est lui-même NP-difficile [19], mais au sens faible (« weakly NP-hard »). Comme le problème `P2||Cmax` est une instance du problème `Rm|prec|Cmax`, le problème `Rm|prec|Cmax` est donc lui-aussi NP-difficile. Il est même **NP-difficile au sens fort** (« strongly NP-hard ») car il est possible de réduire polynomialement `3-PART` en `R2|prec|Cmax`, et car `3-PART` est lui-même NP-difficile au sens fort. Ceci explique que les solutions qui ont été proposées pour `Rm|prec|Cmax` sont basées sur des heuristiques.

Voici à présent une réduction polynomiale de `3-PART` en `R2|prec|Cmax`, lesquels problèmes sont formellement définis de la façon suivante :

Problème 3.2 (`3-PART`) *Soit un entier $B > 0$ et soit un ensemble A de $3m$ entiers a_i, tels que $\Sigma_{i=1}^{3m} a_i = mB$ et $\forall 1 \leq i \leq 3m, \frac{1}{4}B < a_i < \frac{1}{2}B$.*

Trouver une partition de A en m sous-ensembles A_j tels que $\forall 1 \leq j \leq m, \Sigma_{a_i \in A_j} a_i = B$.

Problème 3.3 (`R2|prec|Cmax`) *Soient deux processeurs quelconques P_1 et P_2, et soient p tâches τ_i reliées entre elles par des contraintes de précédence. Pour chaque tâche τ_i, son temps d'exécution $\mathcal{E}xe\langle \tau_i, P_1 \rangle \geq 0$ et $\mathcal{E}xe\langle \tau_i, P_2 \rangle \geq 0$, respectivement sur P_1 et P_2, est connu.*

Trouver un ordonnancement des p tâches τ_i sur les deux processeurs P_1 et P_2, et dont le `Cmax` soit minimal.

Remarquons que, dans le cas particulier d'un ordonnancement sur deux processeurs, la formule (3.1) du `Cmax` est équivalente à $\texttt{Cmax} = \max\left(\max_{\tau_i \in P_1} C\langle \tau_i, P_1 \rangle, \max_{\tau_i \in P_2} C\langle \tau_i, P_2 \rangle\right)$, où $C\langle \tau_i, P_j \rangle$ est la date de terminaison de τ_i sur P_j [3].

Il s'agit donc de transformer une instance de `3-PART` en une instance de `R2|prec|Cmax` pour la résoudre. À partir des $3m$ entiers a_i, je définis $3m$ tâches τ_i, sans contraintes de précédence, et telles

[3] La notation « $\tau_i \in P_j$ » signifie « la tâche τ_i est ordonnancée sur le processeur P_j ».

que $\mathcal{E}xe\langle\tau_i, P_1\rangle = a_i$ et $\mathcal{E}xe\langle\tau_i, P_2\rangle = K$, où K est un nombre arbitrairement grand. Ceci assure que toutes les tâches τ_i seront ordonnancées sur P_1. Puis, je définis n tâches τ'_j telles que $\mathcal{E}xe\langle\tau'_j, P_1\rangle = K$ et $\mathcal{E}xe\langle\tau'_j, P_2\rangle = B$. Ceci assure que toutes les tâches τ'_j seront ordonnancées sur P_2. Je définis de plus $n-1$ tâches τ''_k telles que $\mathcal{E}xe\langle\tau''_k, P_1\rangle = k$ et $\mathcal{E}xe\langle\tau''_k, P_2\rangle = K$, où k est un nombre positif arbitrairement petit, en particulier strictement plus petit que $\min_i a_i$. Ceci assure que toutes les tâches τ''_k seront ordonnancées sur P_1. Enfin, je définis les contraintes de précédence suivantes : $\tau'_1 \rightarrow \tau''_1 \rightarrow \tau'_2 \rightarrow \tau''_2 \rightarrow \cdots \rightarrow \tau''_{n-1} \rightarrow \tau'_n$. La figure 3.3 illustre cette situation, où les tâches τ'_j et τ''_k sont déjà ordonnancées. Les $3m$ tâches τ_i restant à ordonnancer ne peuvent être placées que sur P_1, dans des intervalles de longueur B, puisque les intervalles de longueur k sur P_2 sont trop petits.

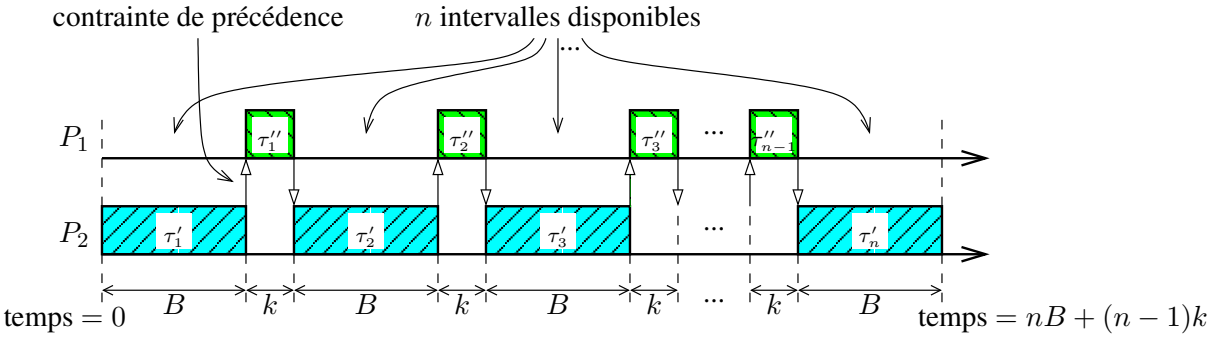


FIG. 3.3 – Réduction de `3-PART` en `R2|prec|Cmax`.

Sur P_2, j'ai $\max_{\tau \in P_2} C\langle\tau, P_2\rangle = C\langle\tau'_n, P_2\rangle = nB + (n-1)k$. Sur P_1, j'ai $\Sigma_{\tau \in P_1} \mathcal{E}xe\langle\tau, P_1\rangle = nB + (n-1)k$, ce qui est donc le minimum qui soit atteignable. Donc, s'il existe un ordonnancement qui soit solution de l'instance de `R2|prec|Cmax` ainsi construite, alors `Cmax` $= nB + (n-1)k$. Dans cet ordonnancement, les $3m$ tâches τ_i sont donc ordonnancées dans les m intervalles de longueur B. Par conséquent, j'obtiens bien une partition de l'ensemble des $3m$ tâches τ_i en m sous-ensembles A_j tels que $\Sigma_{\tau_i \in A_j} \mathcal{E}xe\langle tau_i, P_1\rangle = B$.

En conclusion, j'ai bien obtenu, par une réduction polynomiale, une solution de `3-PART` à partir d'une solution de `R2|prec|Cmax`.

3.3.6 L'adéquation algorithme architecture (AAA)

Yves Sorel travaille depuis plus de 15 ans sur le problème 3.1 de l'ordonnancement & répartition statique d'un algorithme flot-de-données sur une architecture répartie hétérogène [76, 110, 77, 46, 47]. Avec son équipe, il a défini une méthode d'ordonnancement & répartition, intitulée « Adéquation Algorithme Architecture » (AAA). La figure 3.4 montre le schéma général de la méthode AAA. La partie qui m'intéresse dans ce chapitre est la phase d'ordonnancement & répartition (en gras et rouge dans la figure 3.4).

Le problème 3.1 étant NP-difficile, la méthode AAA utilise un algorithme heuristique. Spécifiquement, c'est un algorithme d'**ordonnancement glouton de type liste** [121]. Son principe est de tenir à jour deux ensembles d'opérations : l'ensemble O_{ord} des opérations déjà ordonnancées, initialement vide, et l'ensemble O_{cand} des opérations candidates à être ordonnancées, contenant initialement les opérations sans prédécesseur, c'est-à-dire les opérations d'entrée de $\mathcal{A}lg$. À chaque étape, l'algorithme choisit parmi O_{cand} une opération X et la place sur un des processeurs P de $\mathcal{A}rc$, en utilisant une fonction de coût. Ce faisant, l'algorithme calcule quelles sont les dépendances de données qu'il doit placer sur les media de communication, de telle sorte que l'opération X reçoive toutes ses données d'entrée avant de débuter son exécution. Puis, l'opération qui vient d'être placée est retirée de l'ensemble O_{cand} et est ajoutée à l'ensemble O_{ord}. Enfin, sont ajoutées à l'ensemble O_{cand} les opérations de $\mathcal{A}lg$ qui ne sont

pas dans O_{ord} et dont toutes les opérations prédécesseur sont dans O_{ord}. L'algorithme termine lorsque toutes les opérations du graphe $\mathcal{A}lg$ sont placées sur les processeurs de $\mathcal{A}rc$; à ce moment, O_{cand} est vide tandis que O_{ord} contient toutes les opérations de $\mathcal{A}lg$.

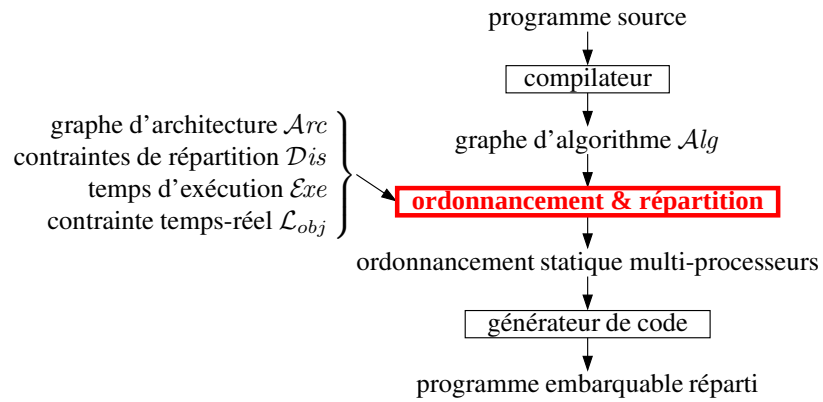


FIG. 3.4 – Schéma général de la méthode AAA.

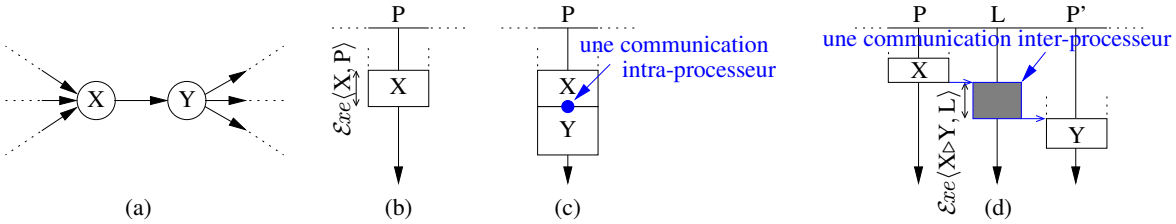


FIG. 3.5 – Principe de la méthode AAA pour le graphe d'algorithme partiel (a) : dans l'ordonnancement partiel avant le placement de Y, X est placée sur le processeur P (b) ; lors du placement de Y, il y a ajout d'une communication intra-processeur (c) ou inter-processeur (d).

La figure 3.5 illustre ce principe. Supposons que le graphe $\mathcal{A}lg$ contienne deux opérations, X et Y, reliées par une dépendance de données X▷Y : c'est la figure 3.5(a). La figure 3.5(b) montre l'ordonnancement partiel de ce graphe $\mathcal{A}lg$, où l'opération X a été placée sur le processeur P ; un processeur y est représenté par une droite verticale, et chaque opération qui lui est affectée est représentée par un rectangle blanc dont la largeur est arbitraire et dont la hauteur est proportionnelle à son WCET (ici $\mathcal{E}xe\langle \mathrm{X}, \mathrm{P}\rangle$). Si l'heuristique choisit de placer Y sur le même processeur P, alors la dépendance de données X▷Y est implantée sous la forme d'une **communication intra-processeur** : c'est la figure 3.5(c). Au contraire, si l'heuristique choisit de placer Y sur un autre processeur P', alors la dépendance de données X▷Y est implantée sous la forme d'une **communication inter-processeur** : c'est la figure 3.5(d). De même que pour les processeurs, le medium de communication L (ici un lien de communication point-à-point) est représenté dans cet ordonnancement par une ligne verticale, et chaque dépendance de données qui lui est affectée est représentée par un rectangle gris dont la largeur est arbitraire et dont la hauteur est proportionnelle à son WCTT (ici $\mathcal{E}xe\langle \mathrm{X} \triangleright \mathrm{Y}, \mathrm{L}\rangle$).

Afin de choisir une opération à ordonnancer parmi toutes celles de l'ensemble des opérations candidates, l'algorithme AAA utilise une fonction de coût, appelée « pression d'ordonnancement » et notée $\mathcal{FC}_{AAA}$ [116]. La pression d'ordonnancement de l'opération o_i sur le processeur p_j est donnée par l'équation (3.2) :

$$\mathcal{FC}_{AAA}(o_i, p_j) = P(o_i, p_j) - F(o_i, p_j) \tag{3.2}$$

où $P(o_i, p_j)$ est la pénalité d'ordonnancement, c'est-à-dire l'allongement du $\mathtt{C}_{\mathtt{max}}$ quand on décide de

placer o_i sur p_j, et $F(o_i, p_j)$ est la flexibilité d'ordonnancement, c'est-à-dire la différence entre la date de début d'exécution au plus tard de o_i sur p_j, calculée depuis le début de l'ordonnancement, et la date de début d'exécution au plus tôt de o_i sur p_j, calculée elle aussi depuis le début de l'ordonnancement.

Donc, pour chaque opération candidate X, l'algorithme calcule sa pression d'ordonnancement sur chaque processeur P de $\mathcal{A}rc$ et conserve le couple $\langle \mathrm{X}, \mathrm{P} \rangle$ pour lequel cette pression d'ordonnancement est *la plus petite* : chaque couple indique quel est le meilleur processeur pour exécuter X. Puis, parmi tous les couples conservés, l'algorithme choisit celui dont la pression d'ordonnancement est *la plus grande* : ce couple indique quelle est l'opération la plus pénalisante, c'est-à-dire celle qui accroît le plus la longueur de l'ordonnancement. Le fait de choisir l'opération la plus pénalisante permet, au final, de minimiser la longueur de l'ordonnancement. C'est dans la pression d'ordonnancement, utilisée comme fonction de coût, que réside l'aspect heuristique de l'algorithme AAA.

L'algorithme 3.1 décrit la séquence d'instructions de l'algorithme AAA. Il prend en entrée les graphes $\mathcal{A}lg$ et $\mathcal{A}rc$, ainsi que les temps d'exécution et de communication $\mathcal{E}xe$, les contraintes de répartition $\mathcal{D}is$ et la contrainte temps-réel $\mathcal{L}_{obj}$. En sortie, il fournit un ordonnancement multiprocesseur et statique $\mathcal{O}rd$ de $\mathcal{A}lg$ sur $\mathcal{A}rc$, représenté par ses deux fonctions d'affectation spatiale $\mathcal{AS}$ et d'affectation temporelle $\mathcal{AT}$: la première est calculée aux lignes 22 et 27, et la seconde aux lignes 23 et 28. Pour chaque processeur (resp. liens de communication), l'algorithme tient à jour la date de terminaison d_{term} de la dernière opération (resp. communication) implantée dessus, à la ligne 24 (resp. 29). Cela permet de calculer la date de terminaison maximale pour tous les processeurs, à la ligne 25. Si jamais cette date excède la contrainte temps-réel $\mathcal{L}_{obj}$, alors c'est un échec et l'algorithme termine, à la ligne 33. Enfin, cet algorithme suppose que l'architecture cible est complètement connectée. En effet, lors de l'implantation des communications nécessaires à l'opération choisie o^{pire} (lignes 19 à 23), l'algorithme implante chaque dépendance de données $o_i \triangleright o^{pire}$ sur un lien de communication, alors qu'en toute généralité il faudrait l'implanter sur un chemin avec du routage.

Algorithme 3.1 AAA

▷ *Initialisation des deux ensembles d'opérations*
1 $O_{ord} := \emptyset$;
2 $O_{cand} :=$ {opérations de $\mathcal{A}lg$ sans prédécesseur} ;
3 **tant que** $O_{cand} \neq \emptyset$ **faire**
▷ *Pour chaque opération candidate o_i, calcul du meilleur processeur $p_i^{meilleur}$*
4 $f^{pire} := 0$;
5 **pourtout** $o_i \in O_{cand}$ **faire**
6 $f_i^{meilleur} := +\infty$;
7 **pourtout** $p_j \in \mathcal{P}$ **faire**
8 **si** $\mathcal{FC}_{AAA}(o_i, p_j) < f_i^{meilleur}$ **alors**
9 $f_i^{meilleur} := \mathcal{FC}_{AAA}(o_i, p_j)$;
10 $p_i^{meilleur} := p_j$;
11 **fin si**
12 **fin pourtout**
▷ *Calcul du couple le plus pénalisant $\langle o^{pire}, p^{pire} \rangle$*
13 **si** $f_i^{meilleur} > f^{pire}$ **alors**
14 $f^{pire} := f_i^{meilleur}$;
15 $\langle o^{pire}, p^{pire} \rangle := \langle o_i, p_i^{meilleur} \rangle$;
16 **fin si**
17 **fin pourtout**
▷ *Implantation des communications nécessaires à o^{pire}*
18 $d_{max} := 0$;
19 **pourtout** $o_i \in pred(o^{pire})$ **faire**

$p_i := \mathcal{AS}(o_i)$;
$l_j := l_{p_i,p^{pire}}$;
$\mathcal{AS}(o_i \triangleright o^{pire}) := l_j$;
$\mathcal{AT}(l_j, o_i \triangleright o^{pire}) := \max\{d_{term}(l_j), \mathcal{AT}(p_i, o_i) + \mathcal{E}xe\langle p_i, o_i\rangle\}$;
$d_{term}(l_j) := \mathcal{AT}(l_j, o_i \triangleright o^{pire}) + \mathcal{E}xe\langle l_j, o_i \triangleright o^{pire}\rangle$;
$d_{max} := \max\{d_{max}, d_{term}(l_j)\}$;
fin pourtout
▷ *Implantation de o^{pire} sur p^{pire}*
$\mathcal{AS}(o^{pire}) := p^{pire}$;
$\mathcal{AT}(p^{pire}, o^{pire}) := \max\{d_{term}(p^{pire}), d_{max}\}$;
$d_{term}(p^{pire}) := \mathcal{AT}(p^{pire}, o^{pire}) + \mathcal{E}xe\langle p^{pire}, o^{pire}\rangle$;
▷ *Mise à jour des deux ensembles d'opérations*
$O_{ord} := O_{ord} \cup \{o^{pire}\}$;
$O_{cand} := O_{cand} - \{o^{pire}\} \cup \{$opérations de $\mathcal{A}lg$ qui avaient o^{pire} comme prédecesseur et dont tous les prédécesseurs sont dans $O_{ord}\}$;
▷ *Vérification de la contrainte temps-réel*
si $d_{max} > \mathcal{L}_{obj}$ **alors**
arrêter ;
fin si
fin tant que
fin

3.3.7 L'outil de CAO de systèmes embarqués répartis SYNDEX

SYNDEX est l'outil de conception de systèmes embarqués répartis qui met en œuvre la méthode AAA présentée ci-dessus.[4] La version actuelle est programmée en CAML et offre une interface graphique pour spécifier aussi bien le graphe d'algorithme $\mathcal{A}lg$ que le graphe d'architecture $\mathcal{A}rc$. En outre, SYNDEX est interfacé avec SIGNAL/POLYCHRONY [5], SCILAB/SCICOS [6] et SCADE/ESTEREL [7] pour construire le graphe d'algorithme $\mathcal{A}lg$ à partir d'un langage de haut niveau.

Afin d'effectuer des simulations sur des grands nombres de graphes, SYNDEX possède deux générateurs aléatoires de graphes, un pour les graphes d'architectures et un pour les graphes d'algorithmes. Le générateur de graphes d'architectures utilise le modèle de Waxman [117]. Ce modèle génère une grille de dimensions spécifiées par l'utilisateur, et place dans cette grille le nombre requis de sommets. La probabilité qu'une arête (c'est-à-dire dans notre cas un medium de communication) existe entre deux noeuds a et b est donnée par la formule (3.3) :

$$P(a,b) = \beta \exp^{-d(a,b)/\alpha L} \tag{3.3}$$

où β et α sont les paramètres fournis en entrée, L est la distance maximale entre deux sommets quelconques de la grille (c'est-à-dire $L = \sqrt{\text{largeur\_grille} \times \text{hauteur\_grille}}$), et $d(a,b)$ est la distance entre les sommets a et b. Plus la valeur de β est grande, plus la probabilité qu'une arête existe entre deux sommets est forte.

Pour chaque paire de sommets (a,b), un nombre aléatoire est généré dans l'intervalle $[0,1[$; s'il est inférieur à la probabilité de Waxman $P(a,b)$ donnée par l'équation (3.3), alors une arête est créée entre les deux sommets a et b, sinon ils ne sont pas reliés. Une boucle simple de cet algorithme ne garantit pas

[4] SYNDEX = « Synchronized Distributed Executive » : `http://www-rocq.inria.fr/syndex`.
[5] SIGNAL/POLYCHRONY : `http://www.irisa.fr/espresso/Polychrony`.
[6] SCILAB/SCICOS : `http://www.scilab.org`.
[7] SCADE/ESTEREL : `http://www.esterel-technologies.com`.

que le graphe obtenu sera connecté. Aussi le générateur aléatoire ajoute-t-il des arêtes supplémentaires jusqu'à ce que le graphe soit connecté.

Quant au générateur de graphes d'algorithmes, il utilise le modèle de Rose et al. [97]. Ce modèle représente un graphe comme une séquence de « niveaux », chacun composés de plusieurs nœuds connectés à au moins un nœud de niveau inférieur. Trois paramètres permettent de contrôler le générateur : le nombre n de nœuds à générer, le nombre maximal h de niveaux, et la nombre maximal ℓ de nœuds indépendants dans un même niveau. Ensuite, n nœuds sont générés aléatoirement dans une matrice $h \times \ell$, puis chaque nœud du niveau i est connecté à un nombre aléatoire de nœuds du niveau précédent $i-1$ ainsi qu'à un nombre aléatoire de nœuds de niveau $j < i-1$, de telle façon que seuls les nœuds de niveau 1 n'ait aucun prédécesseur (ce sont donc les opérations d'entrée de $\mathcal{A}lg$). Ce modèle fonctionne de façon satisfaisante pour des graphes dont la taille est celle des programmes embarqués typiques (des graphes d'algorithme jusqu'à quelques milliers de sommets).

3.3.8 Modèle d'exécution de SYNDEX

Une fois que l'adéquation de l'algorithme $\mathcal{A}lg$ sur l'architecture répartie hétérogène $\mathcal{A}rc$ est faite, on obtient un ordonnancement multiprocesseur de $\mathcal{A}lg$ sur $\mathcal{A}rc$. À partir de cet ordonnancement, SYNDEX est capable de générer automatiquement l'exécutif réparti correspondant. L'ordonnancement étant statique, cet exécutif ne nécessite pas de RTOS résident pour s'exécuter.

Les communications entre les fragments répartis de cet exécutif se font au moyen des deux primitives suivantes :

- `send` pour envoyer une valeur (opération non bloquante),
- `receive` pour recevoir une valeur (opération bloquante tant que la valeur n'a pas été reçue).

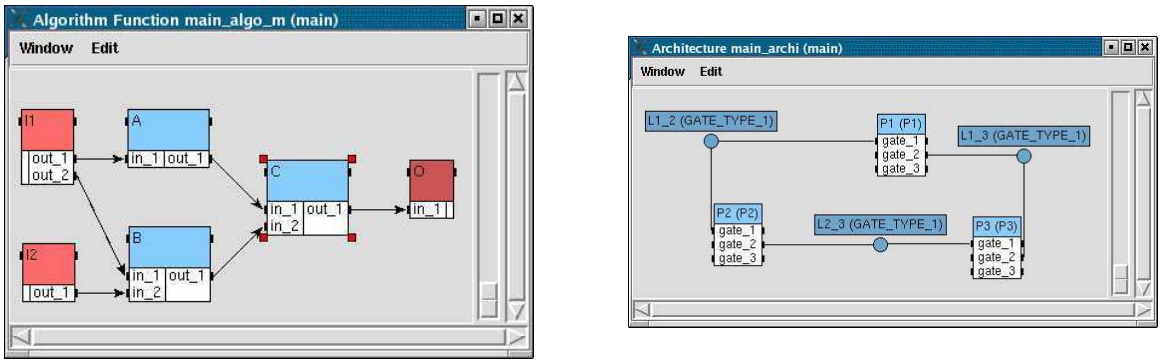
La sémantique de la communication est celle des files d'attente FIFO (« First In First Out »). Les primitives `send` et `receive` sont mises en œuvre au moyen de **sémaphores**. Concrètement, pour chaque dépendance de données, un sémaphore est initialisé en début d'itération avec la valeur 1 [47]. Au cours de l'itération, le `receive` fait l'opération `P` (réservation du sémaphore) et le `send` fait l'opération `V` (libération du sémaphore). Ainsi, le `receive` se bloque jusqu'à ce que le `send` le libère. Cette synchronisation `send`–`receive` a lieu à l'intérieur de l'itération et garantit la correction de l'exécution répartie. De plus, un second sémaphore est utilisé afin que le `send` *de l'itération suivante* soit bloqué jusqu'à ce que la donnée ait été consommée par le `receive` *de l'itération courante*. Cette synchronisation `receive`–`send` a lieu « à cheval » sur deux itérations consécutives et garantit qu'aucun fragment réparti ne prend plus d'une itération d'avance sur les autres fragments.

3.3.9 Exemple d'ordonnancement produit par SYNDEX

Les figures 3.6(a) et (b) montrent un graphe $\mathcal{A}lg$ et un graphe $\mathcal{A}rc$ dans le logiciel SYNDEX. Le graphe $\mathcal{A}lg$ comporte six opérations : deux opérations d'entrée I1 et I2, trois opérations de calcul A, B et C, et une opération de sortie O. Le graphe $\mathcal{A}rc$ comporte trois processeurs P1, P2 et P3, et trois liens de communication point-à-point L12, L13 et L23.

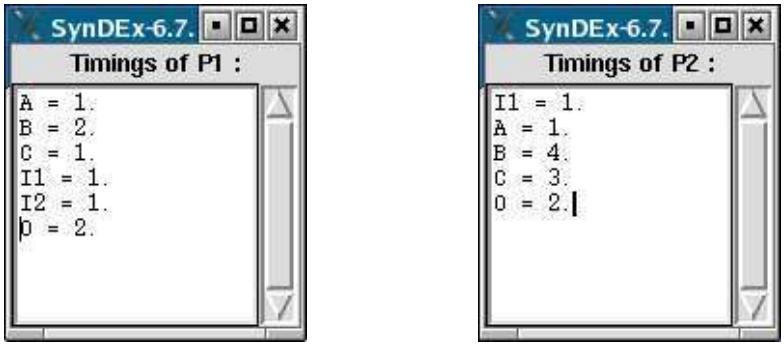
Les figures 3.7(a) et 3.7(b) montrent respectivement les caractéristiques d'exécution des opérations de $\mathcal{A}lg$ sur les processeurs P1 et P2 (l'utilisateur doit également fournir les caractéristiques pour les autres processeurs et pour les media de communication).

La figure 3.8 montre l'ordonnancement produit par AAA pour le graphe $\mathcal{A}lg$ de la figure 3.6(a) sur le graphe $\mathcal{A}rc$ de la figure 3.6(b), avec les caractéristiques d'exécution de la figure 3.7. SYNDEX colorie en orange l'opération pointée par la souris (ici A), en rouge les opérations successeur (ici C mais aussi A▷C), et en vert les opérations prédécesseur (ici I1). La longueur de cet ordonnancement est égale à 7 unités de temps.



(a) (b)

FIG. 3.6 – (a) Exemple de graphe $\mathcal{A}lg$; (b) Exemple de graphe $\mathcal{A}rc$.



(a) (b)

FIG. 3.7 – Caractéristiques d'exécution sur P1 (a) et sur P2 (b).

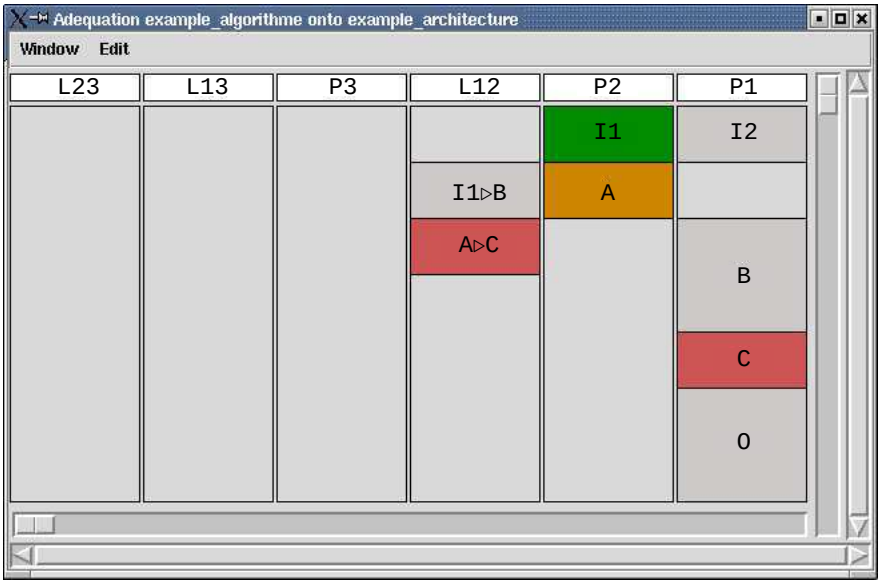


FIG. 3.8 – Ordonnancement produit par AAA.

3.4 Méthodes d'ordonnancement & répartition basées sur AAA

L'idée qui a été suivie dans le cadre de l'ARC TOLÈRE a été de modifier la méthode AAA afin de générer des ordonnancements multiprocesseur qui soient en plus tolérants aux fautes. Cela a abouti à trois méthodes, présentées successivement dans cette section. La section 3.4.2 présente une méthode pour systèmes tolérants aux fautes des composants matériels avec liens point-à-point ; par analogie, elle s'intitule AAA-TP. La section 3.4.3 présente une méthode pour systèmes tolérants aux fautes des composants matériels avec bus de communication, appelée quant à elle AAA-TB. Enfin, la section 3.4.5 présente une méthode bicritère pour garantir à la fois la fiabilité et le temps d'exécution d'un système, appelée AAA-F.

3.4.1 Principes généraux

Le premier principe commun à ces trois méthodes est la **réplication active des opérations** (voir la section 3.2.3), ce qui permet de tolérer les fautes des processeurs de $\mathcal{A}rc$. En revanche, concernant les fautes des media de communication, leur prise en compte dépendra de leur type : pour des liens point-à-point ce sera la **réplication active des communications**, car celles-ci peuvent être parallélisées sur tous les chemins disjoints existant entre deux processeurs donnés ; sauf quand la charge du réseau est trop importante, cela permet de diminuer les délais de transmission (en effet, l'opération destinataire d'un message démarre dès qu'elle reçoit la première copie de ce message), mais aussi les variations des délais de transmission [60]. Enfin, concernant les bus, ce sera la **réplication passive des communications** afin d'éviter de sérialiser trop de communications sur un même bus, et la **fragmentation des données** afin d'améliorer la détection et le recouvrement d'erreur. Dans les deux cas (bus et liens point-à-point), le but est de réduire le surcoût dû à la tolérance aux fautes.

Le second principe commun à ces trois méthodes est d'effectuer une phase de **transformation de graphe** sur $\mathcal{A}lg$ en plus de la phase d'ordonnancement & répartition. L'idée est d'introduire dans $\mathcal{A}lg$ les redondances nécessaires pour tolérer les fautes requises, et d'obtenir ainsi un nouveau graphe $\mathcal{A}lg^*$, contenant plusieurs répliques pour chaque opération et chaque dépendance de données.

- Dans les méthodes AAA-TP et AAA-TB, la phase de transformation a lieu avant la phase d'ordonnancement & répartition. Cela est possible parce que le taux de réplication de chaque opération et de chaque dépendance de données est constant. Aussi, en plus de ce graphe $\mathcal{A}lg^*$, la phase de transformation produit un ensemble de **contraintes de placement**, qui sont ou bien des **contraintes d'exclusions** ou bien des **contraintes intra-processeur**. Elles sont utilisées ensuite par l'heuristique d'ordonnancement & répartition de la façon suivante. Une contrainte d'exclusion entre des opérations (resp. des dépendances de données) indique qu'elles doivent être placées sur des processeurs distints (resp. sur des chemins disjoints). Une telle contrainte d'exclusion est produite pour chaque ensemble de répliques d'une même opération (resp. d'une même dépendance de données), garantissant ainsi un certain niveau de tolérance aux fautes des processeurs (resp. des media de communication). En revanche, une contrainte intra-processeur entre des opérations indique qu'elles doivent être placées sur le même processeur.

  L'heuristique d'ordonnancement & répartition est donc commune aux deux méthodes AAA-TP et AAA-TB. Elle est quasiment identique à l'heuristique d'AAA (algorithme 3.1). Concrètement, voici comment les différentes contraintes de placement sont prises en compte :
  - À la ligne 7, il faut enlever de $\mathcal{P}$ le ou les processeurs sur lesquels sont déjà placées des opérations liées par une contrainte d'exclusion avec l'opération candidate o_i. À l'inverse, il faut réduire $\mathcal{P}$ au singleton $\{p\}$ si l'opération candidate o_i est liée par une contrainte intra-processeur avec une autre opération déjà implantée sur le processeur p.
  - À la ligne 21, il faut choisir un autre lien que $l_{p_i,p^{pire}}$ si celui-ci implante déjà une dépendance de donnée liée par une contrainte d'exclusion avec la dépendance de donnée courante.

– En revanche, dans la méthode AAA-F, la phase de transformation a lieu conjointement avec la phase d'ordonnancement & répartition. En effet, le taux de réplication de chaque opération et de chaque communication n'est pas constant. La phase de transformation doit donc se dérouler en même temps que l'heuristique d'ordonnancement & répartition. Elle diffère de l'heuristique de AAA parce qu'elle doit ordonnancer plusieurs répliques d'une même opération au lieu d'une seule pour l'heuristique d'AAA.

Je termine cette section par un point de vocabulaire. En français, le verbe « répliquer » veut dire « répondre à quelqu'un » ; il n'est donc pas synonyme de « dupliquer », qui lui veut dire « faire un ou plusieurs autres exemplaires de quelque chose ». En anglais, les deux verbes « to duplicate » et « to replicate » sont synonymes. Quant à « duplication » et « réplication », le premier veut dire « doubler » alors que le second n'a pas cette restriction, et c'est donc « réplication » que j'emploierai. Enfin, « duplique » n'existe pas en français, donc le seul terme possible est sans ambiguïté « réplique » (« replica » en anglais), ou ses synonymes « copie », « double », « exemplaire ».

3.4.2 Méthode pour systèmes tolérants aux fautes avec liens point-à-point

Par analogie avec l'acronyme AAA, cette méthode s'appelle AAA-TP : « Adéquation Algorithme Architecture avec Tolérance aux fautes des liens Point-à-point ». Les travaux présentés dans cette section ont été publiés dans les articles [37, 38, 40, 39] et dans la thèse d'Hamoudi Kalla [57].

3.4.2.1 Énoncé du problème

Problème 3.4 (AAA-TP) *Soit un graphe d'algorithme $\mathcal{A}lg$, un graphe d'architecture $\mathcal{A}rc$ avec liens de communication point-à-point, un tableau $\mathcal{E}xe$ des WCET des opérations de $\mathcal{A}lg$ sur les processeurs de $\mathcal{A}rc$ et des WCTT des dépendances de données de $\mathcal{A}lg$ sur les liens de communication de $\mathcal{A}rc$, des contraintes de distribution $\mathcal{D}is$, et deux nombres $\mathcal{N}pf$ et $\mathcal{N}lf$.*

Trouver un ordonnancement multiprocesseur et statique de $\mathcal{A}lg$ sur $\mathcal{A}rc$, qui respecte $\mathcal{D}is$, tolérant à $\mathcal{N}pf$ fautes de processeurs et à $\mathcal{N}lf$ fautes de liens de communication, et dont la longueur calculée grâce à $\mathcal{E}xe$ soit minimale.

De même que le problème 3.1, ce problème est NP-difficile, et donc la méthode AAA-TP que je propose pour le résoudre se base sur un algorithme heuristique d'ordonnancement & répartition. Une contrainte temps-réel $\mathcal{L}_{obj}$ doit de plus être respectée, donc il faut vérifier à la fin que la longueur de l'ordonnancement multiprocesseur obtenu est inférieure à $\mathcal{L}_{obj}$.

Dans toute la section 3.4.2, je noterai respectivement $\mathcal{P}$ et $\mathcal{L}$ l'ensemble des processeurs et des liens de communication de $\mathcal{A}rc$.

3.4.2.2 Hypothèse de faute

Les composants matériels de $\mathcal{A}rc$ (processeurs et liens de communication) sont supposées être à silence sur défaillance. C'est-à-dire qu'un composant matériel est ou bien sain et délivre toujours des sorties correctes et à temps, ou bien défaillant et ne produit plus aucune sortie. De plus, la durée des défaillances est quelconque. Je suppose de plus que $|\mathcal{P}| \geq \mathcal{N}pf + \mathcal{N}lf + 1$ et qu'entre toute paire de processeurs, il existe au moins $\mathcal{N}pf$+$\mathcal{N}lf$+1 chemins disjoints. Cela revient à supposer que, même en présence de $\mathcal{N}pf$ fautes de processeurs et de $\mathcal{N}lf$ fautes de liens de communication, le graphe $\mathcal{A}rc$ n'est jamais déconnecté.

3.4.2.3 Schéma général de la méthode AAA-TP

La figure 3.9 montre le schéma général de la méthode AAA-TP. Les recherches que j'ai dirigées et menées depuis l'ARC TOLÈRE ont donc consisté à modifier l'heuristique d'ordonnancement & réparti-

tion de la méthode AAA (voir la figure 3.4) de façon à ce que l'ordonnancement soit tolérant aux fautes requises des processeurs et des liens de communication, tout en continuant à en minimiser la longueur. Pour cela, l'ordonnancement produit par la méthode AAA-TP présentera de la redondance ; c'est-à-dire que sa fonction d'affectation spatiale associera à chaque opération de $\mathcal{A}lg$ un *ensemble* de processeurs de $\mathcal{A}rc$. Il en ira de même pour les dépendances de données sur les liens de communication.

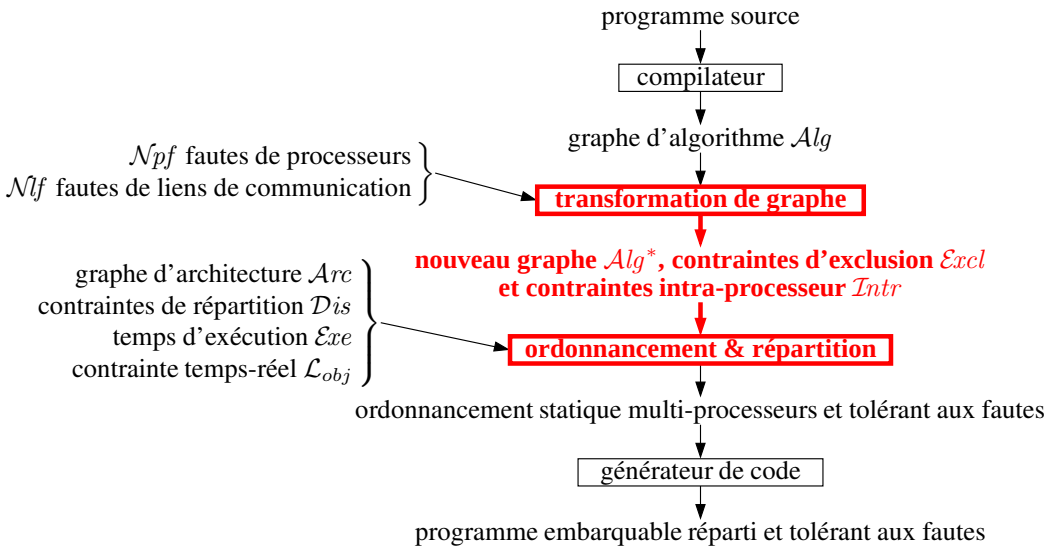


FIG. 3.9 – Schéma général de la méthode AAA-TP.

Dans les sections suivantes, je présente successivement la phase de transformation de graphe, suivie des résultats d'exécution de la méthode AAA-TP et d'un état de l'art. Comme je l'ai dit dans la section 3.4.1, l'heuristique gloutonne d'ordonnancement & répartition est similaire à celle d'AAA. La seule modification par rapport à l'algorithme 3.1 concerne la prise en compte des contraintes d'exclusion $\mathcal{E}xcl$ et des contraintes intra-processeur $\mathcal{I}ntr$ lors de l'implantation d'une nouvelle opération et des dépendances de données qui lui sont nécessaires.

3.4.2.4 Transformation de graphe dans la méthode AAA-TP

En accord avec la section 3.4.1, le moyen choisi pour tolérer les $\mathcal{N}pf$ fautes requises de processeurs est la **réplication active des opérations** ; c'est-à-dire que, pour chaque opération X de $\mathcal{A}lg$, il y aura $\mathcal{N}pf$+1 répliques identiques de X exécutées sur autant de processeurs distincts. De même, le moyen choisi pour tolérer les $\mathcal{N}lf$ fautes requises de liens de communication est la **réplication active des communications** ; c'est-à-dire que, pour chaque dépendance de données X▷Y de $\mathcal{A}lg$, il y aura $\mathcal{N}pf$+$\mathcal{N}lf$+1 répliques identiques transmises sur autant de chemins disjoints ; ces $\mathcal{N}pf$+$\mathcal{N}lf$+1 répliques seront produites par les $\mathcal{N}pf$+1 exemplaires de X, et transmises aux $\mathcal{N}pf$+1 exemplaires de Y.

Ce nombre de $\mathcal{N}pf$+$\mathcal{N}lf$+1 répliques actives des dépendances de données à transmettre sur autant de chemins disjoints est nécessaire. En effet, dans le pire des cas, $\mathcal{N}pf$ processeurs parmi ceux servant à router une de ces répliques seront défaillants, empêchant ainsi $\mathcal{N}pf$ répliques d'être transmises, et en même temps, $\mathcal{N}lf$ liens parmi ceux servant à router les autres répliques seront aussi défaillants, empêchant ainsi $\mathcal{N}lf$ autres répliques d'être transmises. Donc, dans le pire des cas, $\mathcal{N}pf$+$\mathcal{N}lf$ répliques d'une dépendance de données peuvent échouer.

Donc, chaque opération est dupliquée $\mathcal{N}pf$+1 fois : je note X^i la i-ième réplique de X et $\mathcal{R}ep$(X) l'ensemble de toutes les répliques de X ; sur un processeur P donné, les temps d'exécution $\mathcal{E}xe\langle X^i, P\rangle$ sont identiques pour tous les $i \in [1, \mathcal{N}pf + 1]$. De même, chaque dépendance de données est dupliquée

$\mathcal{N}pf$+$\mathcal{N}lf$+1 fois : je note X▷Y^j la j-ième réplique de X▷Y et $\mathcal{R}ep$(X▷Y) l'ensemble de toutes les répliques de X▷Y ; sur un lien de communication L donné, les temps de transmission $\mathcal{E}xe\langle$X▷Y, L$\rangle$ sont identiques pour tous les $j \in [1, \mathcal{N}pf + \mathcal{N}lf + 1]$.

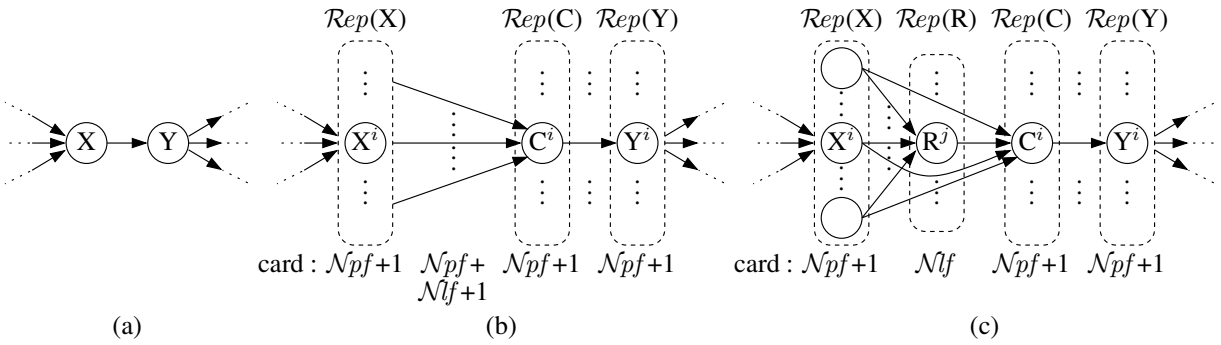


FIG. 3.10 – Transformation du graphe $\mathcal{A}lg$ (a) en graphe (b) : réplication des opérations en $\mathcal{N}pf$+1 exemplaires, réplication des dépendances de données en $\mathcal{N}pf$+$\mathcal{N}lf$+1 exemplaires, et ajout des $\mathcal{N}pf$+1 opérations de choix ; puis, transformation du graphe (b) en graphe $\mathcal{A}lg^*$ (c) : ajout des $\mathcal{N}lf$ opérations de routage et connexion des dépendances de données.

Pour faire en sorte que chaque réplique Y^i de chaque opération Y destination d'une dépendance de données X▷Y débute son exécution dès qu'elle reçoit la *première* des $\mathcal{N}pf$+$\mathcal{N}lf$+1 répliques de cette communication, j'ajoute $\mathcal{N}pf$+1 **opérations de choix**, notées C^i. Chacune possède $\mathcal{N}pf$+1 entrées connectées aux $\mathcal{N}pf$+1 premières dépendances de données X▷Y provenant de $\mathcal{R}ep$(X), et une unique sortie connectée à une des $\mathcal{N}pf$+1 réplique Y^i. Toutes ces opérations de choix sont *identiques* : le code de C^i consiste à attendre la réception de sa *première* entrée en provenance d'une des répliques X^i ; dès que cette donnée est reçue, elle la transmet à Y^i et termine son exécution ; les données reçues ultérieurement sont donc ignorées.

Ce fonctionnement nécessite d'adapter le modèle d'exécution de SYNDEX (section 3.3.8). Le sémaphore de la dépendance de données X▷Y étant initialisé avec la valeur 1, le premier `V` libère le `receive` et donc l'opération Y, destinataire de la dépendance de données, alors que les autres `V` ne font qu'incrémenter la variable du sémaphore. À la fin de l'itération, la variable du sémaphore est remise à 1, ce qui empêche les autres `V` de perturber l'itération suivante.

De plus, chaque opération de choix C^i est implantée sur le *même* processeur que l'opération Y^i à laquelle sa sortie est connectée ; ceci sera garanti grâce à une contrainte de placement intra-processeur. Enfin, les $\mathcal{N}pf$+$\mathcal{N}lf$+1 répliques de la dépendance de données X▷Y sont toutes connectées à chaque réplique C^i de C. Ce principe est illustré dans la figure 3.10(b).

Il reste à présent à connecter les $\mathcal{N}pf$+1 répliques de X aux $\mathcal{N}pf$+$\mathcal{N}lf$+1 répliques de la dépendance de données X▷Y. Tout d'abord, chaque réplique de X est connectée à une réplique de X▷Y. Il reste donc encore $\mathcal{N}lf$ répliques de X▷Y à connecter. Pour cela, j'ajoute $\mathcal{N}lf$ **opérations de routage**, notées R^j. Chacune possède $\mathcal{N}pf$+1 entrées connectées aux $\mathcal{N}pf$+1 répliques X^i, et une unique sortie connectée à une des $\mathcal{N}lf$ dépendances de données X▷Y, non-encore connectée, de chaque réplique C^k. Toutes ces opérations de routage sont *identiques* : le code de R^j est le même que celui d'une opération de choix, mis à part le fait qu'elle effectue en plus le routage de sa sortie vers le processeur implantant C^i. Ces $\mathcal{N}lf$ opérations de routage servent à créer $\mathcal{N}lf$ sources supplémentaires pour transmettre la donnée X▷Y aux répliques de Y (via les répliques de C), donnant ainsi un total de $\mathcal{N}pf$+$\mathcal{N}lf$+1 opérations sources pour les $\mathcal{N}pf$+$\mathcal{N}lf$+1 répliques de X▷Y. Pour que cela fonctionne, elles doivent être placées sur des processeurs *distincts* de ceux implantant les répliques de X ; ceci sera garanti grâce à une contrainte d'exclusion. Ce

principe est illustré dans la figure 3.10(c).

En plus du graphe $\mathcal{A}lg^*$, les contraintes de placement suivantes sont produites :

$$\begin{aligned}
\mathcal{E}xcl &= \{X^1,\ldots,X^{\mathcal{N}pf+1},R^1,\ldots,R^{\mathcal{N}lf}\},\{C^1,\ldots,C^{\mathcal{N}pf+1}\},\{Y^1,\ldots,Y^{\mathcal{N}pf+1}\},\\
&\quad \{X^1 \triangleright C^1,\ldots,X^{\mathcal{N}pf+1} \triangleright C^1,R^1 \triangleright C^1,\ldots,R^{\mathcal{N}lf} \triangleright C^1\},\ldots,\\
&\quad \{X^1 \triangleright C^{\mathcal{N}pf+1},\ldots,X^{\mathcal{N}pf+1} \triangleright C^{\mathcal{N}pf+1},R^1 \triangleright C^{\mathcal{N}pf+1},\ldots,R^{\mathcal{N}lf} \triangleright C^{\mathcal{N}pf+1}\},\\
&\quad \{X^1 \triangleright R^1,\ldots,X^{\mathcal{N}pf+1} \triangleright R^1\},\ldots,\{X^1 \triangleright R^{\mathcal{N}lf},\ldots,X^{\mathcal{N}pf+1} \triangleright R^{\mathcal{N}lf}\}.\\
\mathcal{I}ntr &= \{C^1,Y^1\},\ldots,\{C^{\mathcal{N}pf+1},Y^{\mathcal{N}pf+1}\}.
\end{aligned}$$

Remarquons que la première contrainte d'exclusion requiert que $|\mathcal{P}|$ soit supérieur à $\mathcal{N}pf$+$\mathcal{N}lf$+1, hypothèse que j'ai bien faite dans la section 3.4.2.2.

3.4.2.5 Détection des défaillances

La détection des défaillances se fait au moyen de **chiens de garde** (« timeout » en anglais) sur les communications. Puisque l'ordonnancement est statique, la date de terminaison de chaque communication implantée sur chaque lien de communication est connue. Au moment de faire une réception, tout processeur arme donc un chien de garde, et si le délai expire avant que la réception n'ait effectivement eu lieu, alors c'est que le lien de communication ou le processeur émetteur est défaillant.

Alternativement, le modèle d'exécution décrit à la section 3.3.8 permet de détecter les défaillances sans avoir recours à des chiens de garde. En effet, tout processeur qui aurait dû effectuer une opération P sur un sémaphore et qui ne l'a pas fait est forcément défaillant.

Comme l'ordonnancement est, par construction, tolérant à un nombre pré-requis de défaillances de processeurs et de liens de communication, il n'y a rien à reconfigurer dynamiquement, ni pour l'itération où se produit la défaillance, ni pour les itérations suivantes.

3.4.2.6 Discussion sur la méthode AAA-TP

J'ai montré qu'il fallait transmettre $\mathcal{N}pf$+$\mathcal{N}lf$+1 répliques de la dépendance de données X▷Y sur autant de chemins disjoints. Comme contre-exemple, considérons deux opérations X et Y reliées par la dépendance de données X▷Y (figure 3.11(a)), et supposons que $\mathcal{N}pf$=1 et $\mathcal{N}lf$=1. X est donc dupliquée en deux exemplaires X^1 et X^2, placées respectivement sur P1 et P3 ; de même, Y est dupliquée en deux exemplaires Y^1 et Y^2, placées respectivement sur P1 et P2. Supposons que chaque réplique d'opération n'envoie sa donnée de sortie qu'aux deux répliques de son opération successeur, et non pas en trois exemplaires comme cela devrait être le cas puisque $\mathcal{N}pf$+$\mathcal{N}lf$+1=3. Supposons alors que P1 et L2-3 défaillent. P1 étant défaillant, seule la réplique Y^2 peut être exécutée ; mais Y^2 ne peut ni recevoir la donnée de X▷Y envoyée par X^1 puisque P1 est défaillant, ni celle envoyée par X^2 puisque L2-3 est défaillant (figure 3.11(b)).

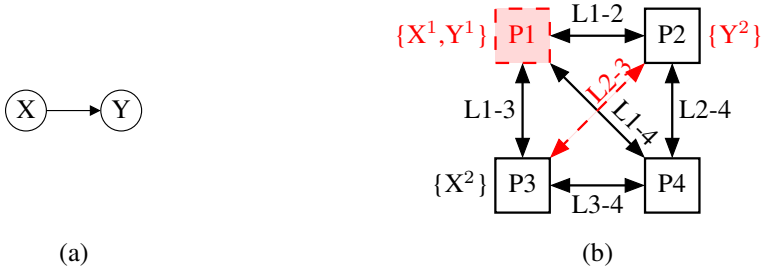


FIG. 3.11 – (a) Graphe d'algorithme $\mathcal{A}lg$; (b) Exemple de réplication sur un graphe $\mathcal{A}rc$ complet à trois processeurs.

Par ailleurs, par hypothèse, le graphe $\mathcal{A}rc$ ne doit jamais être déconnecté même en présence de $\mathcal{N}pf$ fautes de processeurs et de $\mathcal{N}lf$ fautes de liens de communication. Donc, pour $\mathcal{N}pf$=$\mathcal{N}lf$=1, il faut au moins quatre processeurs. La figure 3.12 illustre ce qui se passe avec deux graphes $\mathcal{A}rc$ différents, le premier avec trois processeurs et le second avec quatre processeurs.

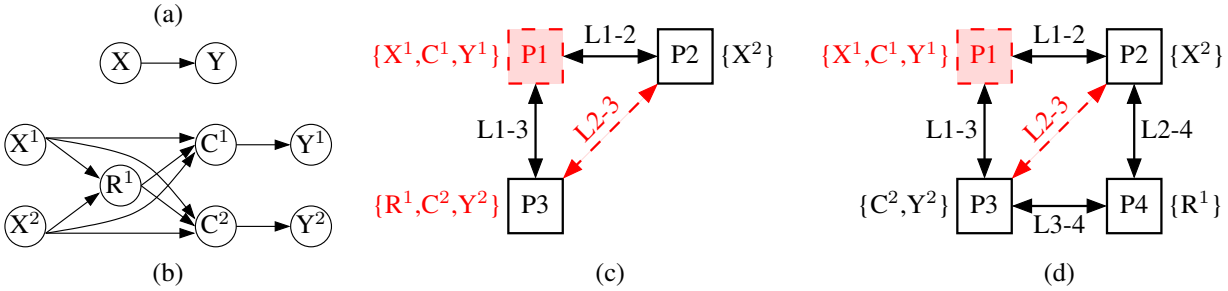


FIG. 3.12 – (a) Graphe d'algorithme $\mathcal{A}lg$; (b) Graphe transformé $\mathcal{A}lg^*$; (c) Placement sur un graphe $\mathcal{A}rc$ à trois processeurs, déconnecté après la défaillance de P1 et de L2-3 : il n'y a alors plus aucun chemin entre P2 et P3, donc le graphe $\mathcal{A}lg^*$ ne peut pas être exécuté entièrement ; (d) Placement sur un graphe $\mathcal{A}rc$ à quatre processeurs, toujours connecté même après la défaillance de P1 et de L2-3, donc le graphe $\mathcal{A}lg^*$ peut être exécuté entièrement.

Enfin, il est toujours possible de dupliquer plus certaines opérations afin d'améliorer la **localité des calculs**, et ainsi d'éviter des communications. Cette idée est utilisée couramment dans les diverses approches de répartition de programmes [48] et plusieurs algorithmes heuristiques existent. C'est particulièrement utile pour les architectures réparties où les communications sont très coûteuses. Ainsi, Hamoudi Kalla a implanté l'algorithme d'Ahmad et Kwok [3] dans SYNDEX, afin d'améliorer les performances de l'algorithme AAA-TP.

3.4.2.7 État de l'art sur la génération d'ordonnancements tolérants aux fautes pour architectures réparties avec liens point-à-point

Plusieurs travaux ont abordé le problème de la génération d'ordonnancements multiprocesseur, statiques et tolérants aux fautes en utilisant la réplication active des opérations [95, 34, 15, 93, 50], et ce sont ceux-là que je vais présenter dans cette section. J'exclus en revanche de cet état de l'art les approches mono-processeur ou basées sur des ordonnancements dynamiques avec RTOS préemptifs, ainsi que les approches intégrées comme le projet Delta-4 [92, 90], qui permettent certes le déploiement d'applications avec un haut niveau de sûreté de fonctionnement (grâce à la réplication active et passive et à l'ajout de composants spécifiques à silence sur défaillance), mais qui n'offrent pas de moyen de déploiement automatique sur l'architecture répartie (c'est-à-dire pas d'ordonnancement & répartition automatique).

Ramamritham considère des tâches périodiques avec contraintes de précédence, de communication et de réplication [95]. Chaque tâche est représentée par un graphe flot-de-données dont les sommets sont des modules logiciels et les arcs les dépendances de données entre ces modules. La tolérance aux fautes est exprimée explicitement par le concepteur puisque certains modules peuvent être dupliqués un nombre fixé de fois ; dans ce cas, un voteur est chargé de combiner les résultats des répliques. L'architecture cible est à mémoire répartie et homogène, alors qu'AAA-TP fonctionne pour des architectures hétérogènes. L'algorithme d'ordonnancement & répartition de Ramamritham procède en deux étapes. Premièrement, certains sommets du graphe sont agrégés en groupes, de telle sorte que tous les sommets d'un même groupe soient placés sur un même processeur (« clustering » en anglais). Concrètement, deux sommets X

et Y sont agrégés si le rapport

$$\frac{\text{WCET(X) + WCET(Y)}}{\text{WCTT(X}\triangleright\text{Y)}}$$

est inférieur à un paramètre appelé « Facteur de Communication » (FC). Ainsi, un nouveau graphe flot-de-données est produit pour un FC donné, ses sommets étant les groupes déterminés par le paramètre FC. Ensuite, ce graphe est ordonnancé et réparti sur les processeurs de l'architecture cible par un algorithme heuristique. Si l'ordonnancement obtenu ne satisfait par les échéances de toutes les tâches, alors le paramètre FC est modifié et les deux étapes sont recommencées. Cette technique d'agrégation est reconnue pour son efficacité [48]. En comparaison, AAA-TP procède en une seule étape, mais fait appel à l'heuristique d'Ahmad et Kwok afin d'améliorer la localité des calculs [3]. Les deux méthodes utilisent la réplication active pour tolérer les fautes des processeurs, mais comme celle de Ramamritham utilise en plus des voteurs, elle permet de traiter en plus les fautes logicielles. En revanche, AAA-TP tolère les fautes des liens de communication, ce que la méthode de Ramamritham ne fait pas.

L'algorithme proposé par Fohler utilise lui une méthode hybride hors-ligne / en-ligne afin de rendre tolérants aux fautes des systèmes répartis [34]. Les tâches considérées sont ou bien périodiques avec des contraintes de précédence, ou bien apériodiques ; toutes sont préemptibles. Une faute affecte une seule tâche à la fois, celle qui est en cours d'exécution sur le processeur au moment de la faute (ceci correspond à une hypothèse de fautes temporaires d'une durée inférieure à la durée maximale des tâches, de manière à n'affecter qu'une seule tâche). De plus, des algorithmes sont supposés exister pour la détection des fautes, les tests d'acceptation et le vote (ces algorithmes sont supposés être parfaitement fiables et non-sujets à défaillances). La méthode proposée par Fohler comporte une phase hors-ligne et une phase en-ligne. Tout d'abord un ordonnancement statique des tâches périodiques et des tâches de tolérance aux fautes (répliques, voteurs...) est construit ; il est constitué d'une succession de créneaux temporels [33]. Puis, à l'exécution, un ordonnanceur dynamique est invoqué à la fin de chaque créneau afin de tester la présence de nouvelles tâches apériodiques, d'exécuter un algorithme qui garantit le respect des échéances, et de sélectionner une tâche pour être exécutée lors du créneau suivant. En comparaison, d'une part AAA-TP produit un ordonnancement purement statique, ce qui simplifie le test d'ordonnancabilité mais ne permet pas de traiter des tâches supplémentaires apériodiques. D'autre part, l'hypothèse de faute est à mon avis plus forte que celle d'AAA-TP puisqu'il est possible de mettre en œuvre le silence sur défaillance si on dispose d'un algorithme de détection des fautes parfaitement fiable et non-sujet à défaillances. Enfin, l'hypothèse que les fautes sont temporaires et ne peuvent affecter qu'une seule tâches est plus forte que celle des fautes de durée quelconque.

L'algorithme proposé par Bertossi et Mancini utilise la réplication des tâches pour obtenir de la tolérance aux fautes [15]. Les tâches sont périodiques et pour chacune, la période et le temps d'exécution maximal sont connus ; les tâches sont de plus indépendantes. Les processeurs sont identiques et à silence sur défaillance. Chaque tâche τ placée sur un processeur possède une autre réplique τ', active ou bien passive, ordonnancée sur un autre processeur. Une réplique active est toujours exécutée, alors qu'une réplique passive n'est exécutée qu'en cas de faute du processeur exécutant τ. L'ordonnancement des tâches est de type « Rate Monotonic » [80]. En comparaison, AAA-TP fonctionne pour des architectures hétérogènes et produit des ordonnancements purement statiques, ce qui ne nécessite pas de support système pour ordonnancer les tâches.

L'algorithme proposé par Qin et al. dans [93] s'appelle eFCRD (« Efficient Fault-Tolerant Reliability Cost-Driven Algorithm »). Il produit un ordonnancement statique d'un DAG de tâches sur une architecture répartie et hétérogène, tolérant à une unique faute d'un processeur. Chaque tâche est ordonnancée deux fois, en une réplique primaire et une réplique secondaire. Deux répliques secondaires ordonnancées sur un même processeur sont autorisées à se *chevaucher* si leurs répliques primaires sont ordonnancées sur des processeurs distincts. Cela veut dire que, pour deux tâches τ_1 et τ_2, les intervalles $[d_1, f_1]$ et $[d_2, f_2]$ ne sont pas forcément disjoints, d_i et f_i étant respectivement la date de début et de fin de la tâche τ_i. Ceci est possible car, par hypothèse, un seul processeur peut être défaillant. Concernant

l'architecture répartie, eFCRD tient compte à la fois de son hétérogénéité en temps de calcul et en temps de communication. Il tient aussi compte de la fiabilité des processeurs afin d'améliorer la fiabilité des ordonnancements qu'il produit, et je reviendrai dans la section 3.4.6 sur cet aspect. Concernant la tolérance aux fautes, d'une part une seule faute de processeur est tolérée, et d'autre part les fautes des liens de communication du réseau ne sont pas prises en compte. Le champ d'application de eFCRD est donc beaucoup plus restreint que AAA-TP.

Enfin, l'algorithme proposé par Hashimoto et al. dans [50] s'appelle HBP (« Height-Based Partitionning »). Il produit un ordonnancement statique d'un DAG de tâches sur une architecture répartie et homogène, tolérant à une unique faute d'un processeur. Le moyen utilisé est la réplication active des opérations, donc chaque tâche est ordonnancée deux fois. Le champ d'application de HBP est donc lui aussi beaucoup plus restreint que AAA-TP. Toutefois, à ma connaissance, c'est le plus proche de AAA-TP en terme d'algorithme d'ordonnancement & répartition, et c'est pour cette raison qu'Hamoudi Kalla l'a implémenté dans SYNDEX, afin d'effectuer des comparaisons entre HBP et AAA-TP (voir la section 3.4.2.8).

3.4.2.8 Résultats d'exécution de l'heuristique d'AAA-TP et discussion

La figure 3.13 présente tout d'abord l'ordonnancement produit par AAA-TP pour $\mathcal{N}pf = 1$ et $\mathcal{N}lf = 0$, avec les graphes de la figure 3.6 et les caractéristiques d'exécution de la figure 3.7. Dans la figure 3.13, le bloc « X » désigne la première réplique de l'opération X (celle dont la date de terminaison est la plus petite), alors que le bloc « X_k# » désigne la k-ième réplique de cette même opération X. La longueur de cet ordonnancement est égale à 12 unités de temps.

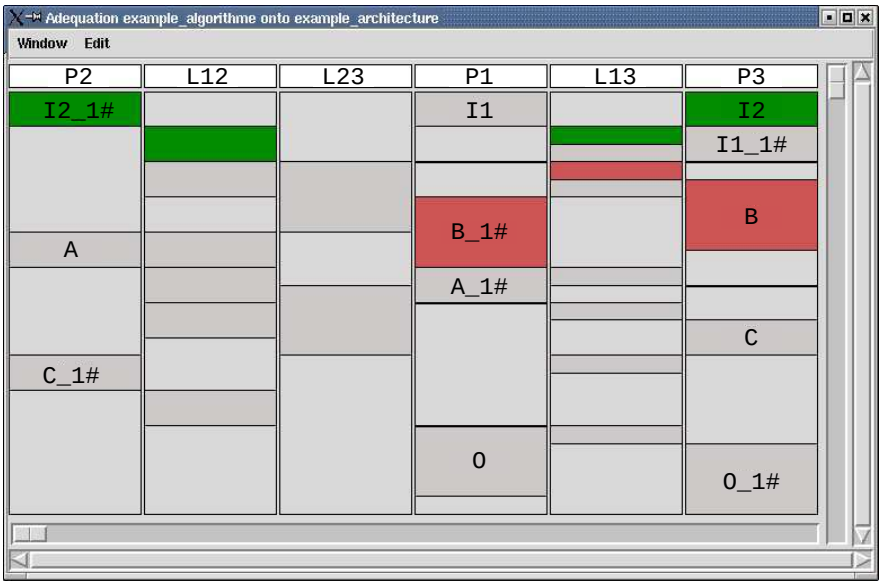


FIG. 3.13 – Ordonnancement produit par AAA-TP ($\mathcal{N}pf$=1,$\mathcal{N}lf$=0).

Afin d'évaluer les performances de l'heuristique d'AAA-TP, j'utilise le générateur aléatoire de graphes de SYNDEX (voir la section 3.3.7). Un des paramètres dont il est intéressant d'étudier l'influence est le rapport entre le WCTT moyen des communications et le WCET moyen des opérations : ce rapport est usuellement appelé « Communication to Computation Ratio » (CCR) ; un CCR supérieur à 1 indique que les communications sont plus coûteuses que les calculs.

La figure 3.14 montre le surcoût en longueur de l'ordonnancement dû à la réplication active des

opérations, en fonction de CCR. Le surcoût est calculé par la formule (3.4) :

$$\text{surcoût} = \frac{\text{longueur(AAA-TP}(\mathcal{N}pf, \mathcal{N}lf)) - \text{longueur(AAA)}}{\text{longueur(AAA)}} \times 100 \tag{3.4}$$

Chaque point est la moyenne de la longueur de l'ordonnancement calculé pour 50 graphes $\mathcal{A}lg$ de $N = 50$ tâches sur le même graphe $\mathcal{A}rc$ complet à $P = 6$ processeurs. Pour CCR$= 0.1$, le surcoût moyen est de 233%, ce qui est très élevé. En revanche, pour CCR$= 5$, le surcoût moyen tombe à seulement 41%. La raison en est que la réplication des opérations améliore la localité des calculs, ce qui tend à diminuer le surcoût.

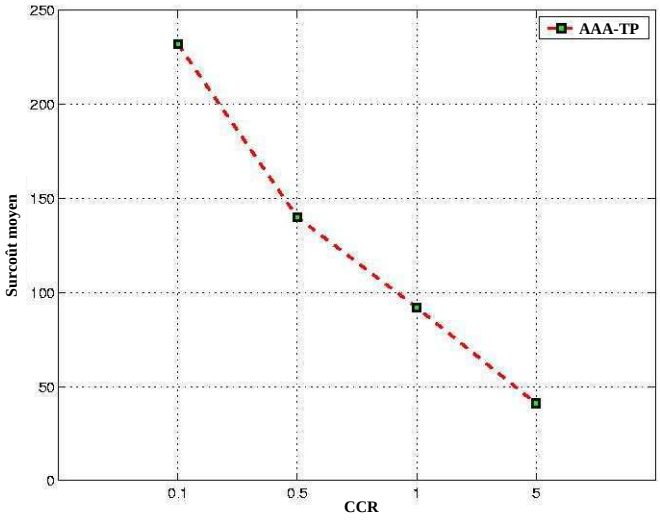


FIG. 3.14 – Surcoût dû à la réplication active des opérations pour $\mathcal{N}pf = 1, \mathcal{N}lf = 0, P = 6$ et $N = 50$.

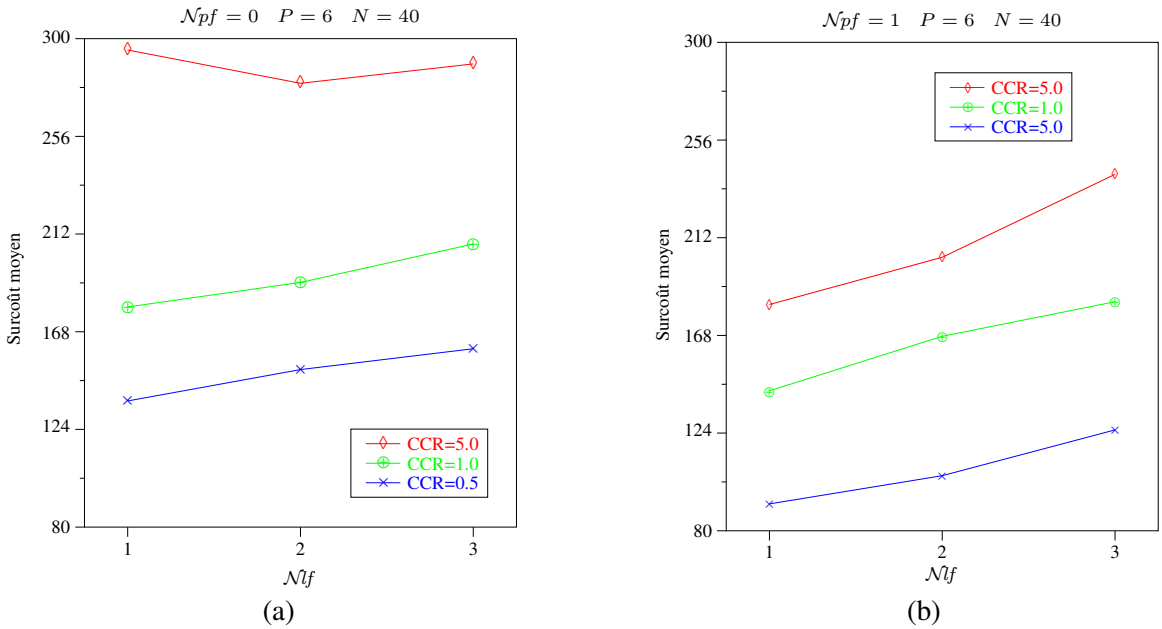


FIG. 3.15 – (a) Surcoût dû à la réplication active des dépendances de données pour $\mathcal{N}pf = 0, \mathcal{N}lf \in \{1, 2, 3\}$, $P = 6$ et $N = 40$; (b) Idem avec $\mathcal{N}pf = 1$.

Les figures 3.15(a) et (b) montrent le surcoût en longueur de l'ordonnancement dû à la réplication active des dépendances de données, en fonction de $\mathcal{N}lf$ et de CCR, la première sans réplication des opérations ($\mathcal{N}pf = 0$), et la seconde avec réplication active des opérations ($\mathcal{N}pf = 1$). Le surcoût est calculé par la formule (3.4). Chaque point est la moyenne de la longueur de l'ordonnancement calculé pour 40 graphes $\mathcal{A}lg$ de 40 tâches sur le même graphe $\mathcal{A}rc$ complet à 6 processeurs. Quand $\mathcal{N}pf = 0$, le surcoût est significatif : de 130% à 150% pour CCR$= 0.5$, de 180% à 205% pour CCR$= 1$, et de 280% à 300% pour CCR$= 5$! En revanche, l'augmentation du surcoût en fonction de $\mathcal{N}lf$ est modérée. Quand $\mathcal{N}pf = 1$, le surcoût est beaucoup plus modéré : de 90% à 125% pour CCR$= 0.5$, de 140% à 180% pour CCR$= 1$, et de 180% à 240% pour CCR$= 5$. Cette différence est due à la réplication des opérations qui améliore la localité des calculs, et donc rend la répartition moins sensible au coût des communications.

Les figures 3.16(a) et (b) comparent l'heuristique d'AAA-TP et celle HBP de Hashimoto et al. [50] en fonction de la taille du graphe $\mathcal{A}lg$, respectivement en l'absence de défaillance et en présence de la défaillance d'un processeur parmi les 4 du graphe $\mathcal{A}rc$. Le surcoût pour AAA-TP est toujours calculé par la formule (3.4), et le surcoût de HBP est lui calculé par la formule (3.5) :

$$\text{surcoût} = \frac{\text{longueur(HBP)} - \text{longueur(AAA)}}{\text{longueur(AAA)}} \times 100 \tag{3.5}$$

Chaque point est la moyenne de 50 graphes $\mathcal{A}lg$ générés aléatoirement avec CCR$= 5$, $\mathcal{N}pf = 1$ et $\mathcal{N}lf = 0$. Sauf pour $N = 10$ opérations, l'heuristique d'AAA-TP est systématiquement meilleure que celle de HBP. Dans cette comparaison, il ne m'est pas possible d'évaluer d'autres configurations des paramètres (par exemple $\mathcal{N}lf \neq 0$ ou $\mathcal{N}pf$>1) car HBP ne tolère qu'une seule faute de processeur et aucune faute de lien de communication.

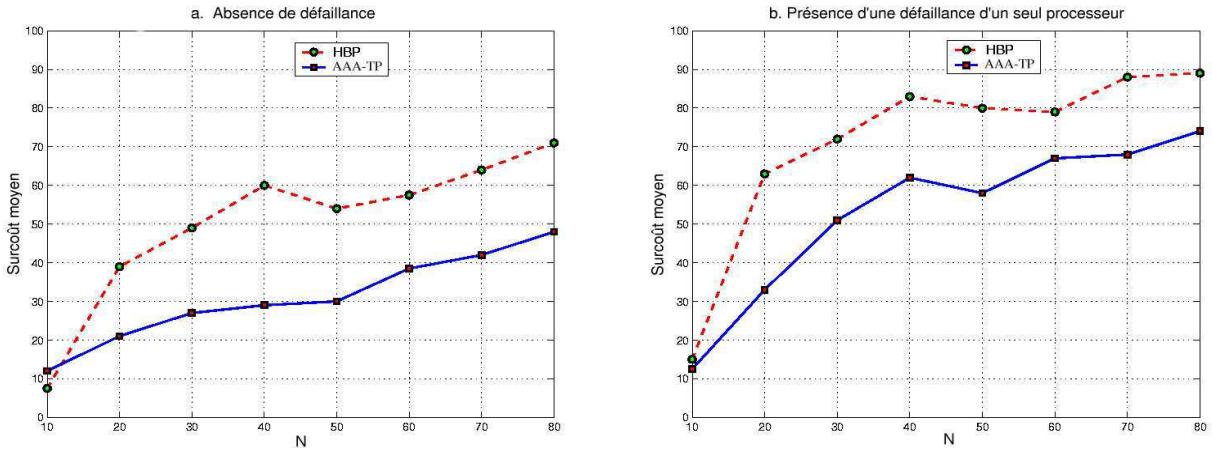


FIG. 3.16 – Comparaison des surcoûts de AAA-TP et HBP en fonction de N pour $\mathcal{N}bf = 1$, $\mathcal{N}lf = 0$, $P = 4$ et CCR$= 5$.

Les figures 3.17(a) et (b) comparent les heuristiques d'AAA-TP et de HBP en fonction de CCR, respectivement en l'absence de défaillance et en présence de la défaillance d'un processeur parmi les 4 du graphe $\mathcal{A}rc$. Chaque point est la moyenne de 50 graphes $\mathcal{A}lg$ générés aléatoirement avec $N = 50$ opérations, $\mathcal{N}bf = 1$ et $\mathcal{N}lf = 0$. Pour CCR< 1, HBP est légèrement meilleure qu'AAA-TP ; pour CCR≥ 1, AAA-TP est nettement meilleure que HBP. Enfin, ces deux figures confirment les résultats illustrés par la figure 3.14, à savoir que les performances des deux heuristiques augmentent avec CCR, en raison de l'amélioration de la localité des calculs.

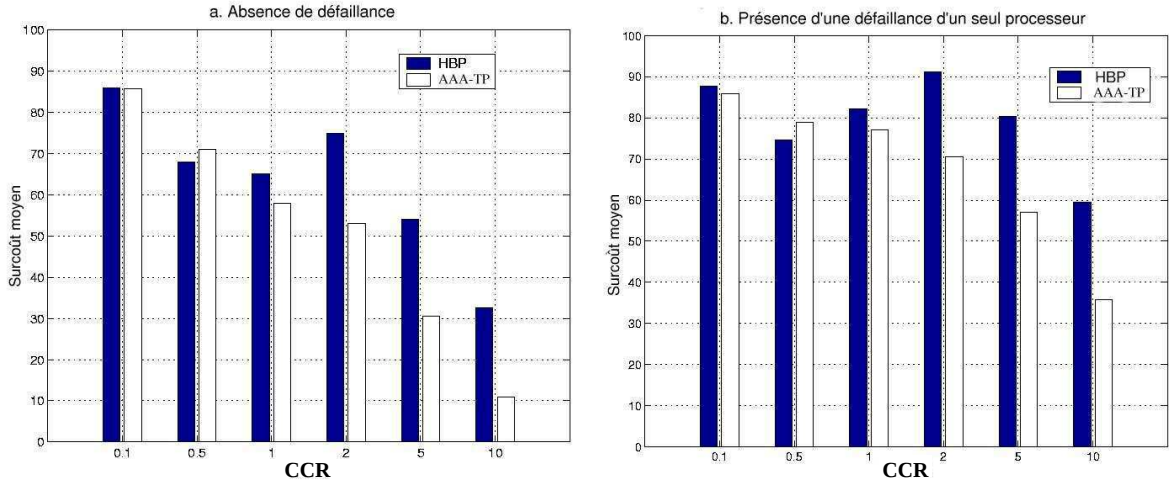


FIG. 3.17 – Comparaison de AAA-TP et HBP en fonction de CCR pour $\mathcal{N}bf = 1$, $\mathcal{N}lf = 0$, $P = 4$ et $N = 50$.

3.4.3 Méthode pour systèmes tolérants aux fautes avec bus

Dans le cadre de la thèse d'Hamoudi Kalla, nous avons proposé de traiter également les fautes des bus de communication. En effet, de nombreuses architectures de systèmes critiques sont construites autour de bus. C'est le cas par exemple dans l'automobile, où les bus de type CAN, TTP et FLEXRAY sont très répandus [99]. L'avantage par rapport aux architectures avec liens point-à-point (et donc par rapport à la méthode AAA-TP) est qu'une donnée envoyée par un processeur peut être reçue par tous les processeurs connectés au bus, ce qui facilite la détection des fautes. Je ne traite dans cette section que le cas de bus qui sont connectés à tous les processeurs de $\mathcal{A}rc$. Le cas contraire complexifie significativement le protocole de communication, à cause du routage des communications par les processeurs et des difficultés à détecter les fautes partielles des bus. L'inconvénient d'un bus est que deux communications envoyées par deux processeurs distincts reliés au même bus doivent être sérialisées au lieu d'être envoyées en parallèle sur deux liens point-à-point distincts, ce qui pénalise les approches basées sur la réplication active des communication. Par analogie avec l'acronyme AAA, cette méthode s'appelle AAA-TB : « Adéquation Algorithme Architecture avec Tolérance aux fautes des Bus ». Les travaux présentés dans cette section ont été publiés dans les articles [42, 41] et dans la thèse d'Hamoudi Kalla [57].

3.4.3.1 Énoncé du problème

Problème 3.5 (AAA-TB) *Soit un graphe d'algorithme $\mathcal{A}lg$, un graphe d'architecture $\mathcal{A}rc$ avec bus de communication, un tableau $\mathcal{E}xe$ des WCET des opérations de $\mathcal{A}lg$ sur les processeurs de $\mathcal{A}rc$ et des WCTT des dépendances de données de $\mathcal{A}lg$ sur les bus de communication de $\mathcal{A}rc$, des contraintes de distribution $\mathcal{D}is$, et deux nombres $\mathcal{N}pf$ et $\mathcal{N}bf$.*

Trouver un ordonnancement multiprocesseur et statique de $\mathcal{A}lg$ sur $\mathcal{A}rc$, qui respecte $\mathcal{D}is$, tolérant à $\mathcal{N}pf$ fautes de processeurs et à $\mathcal{N}bf$ fautes de bus de communication, et dont la longueur calculée grâce à $\mathcal{E}xe$ soit minimale.

De même que le problème 3.1, ce problème est NP-difficile, et donc la méthode AAA-TB que je propose pour le résoudre se base sur un algorithme heuristique d'ordonnancement & répartition. Une contrainte temps-réel $\mathcal{L}_{obj}$ doit de plus être respectée, donc il faut vérifier à la fin que la longueur de l'ordonnancement multiprocesseur obtenu est inférieure à $\mathcal{L}_{obj}$.

Dans toute la section 3.4.3, je noterai respectivement $\mathcal{P}$ et $\mathcal{B}$ l'ensemble des processeurs et des bus de $\mathcal{A}rc$.

3.4.3.2 Hypothèse de faute

Les composants matériels de $\mathcal{A}rc$ (processeurs et bus de communication) sont supposées à silence sur défaillance. Les fautes des bus son supposées être totales. Une **faute totale** rend inopérante toutes les connexions processeur-bus, ne permettant donc plus aucune communication par l'intermédiaire de ce bus. De plus, la durée des défaillances est quelconque. Je suppose de plus que $|\mathcal{P}| \geq \mathcal{N}pf + 1$ et que $|\mathcal{B}| \geq \mathcal{N}bf + 1$. Cela revient à supposer que, même en présence de $\mathcal{N}pf$ fautes de processeurs et de $\mathcal{N}bf$ fautes de bus, le graphe $\mathcal{A}rc$ n'est jamais déconnecté.

3.4.3.3 Schéma général de la méthode AAA-TB

La figure 3.18 montre le schéma général de la méthode AAA-TB. Les recherches que j'ai dirigées et menées depuis l'ARC TOLÈRE ont donc consisté à modifier l'heuristique d'ordonnancement & répartition de la méthode AAA (voir la figure 3.4) de façon à ce que l'ordonnancement soit tolérant aux fautes requises des processeurs et des bus de communication, tout en continuant à en minimiser la longueur. Pour cela, l'ordonnancement produit par la méthode AAA-TB présentera de la redondance ; c'est-à-dire que sa fonction d'affectation spatiale associera à chaque opération de $\mathcal{A}lg$ un *ensemble* de processeurs de $\mathcal{A}rc$. Il en ira de même pour les dépendances de données sur les bus de communication.

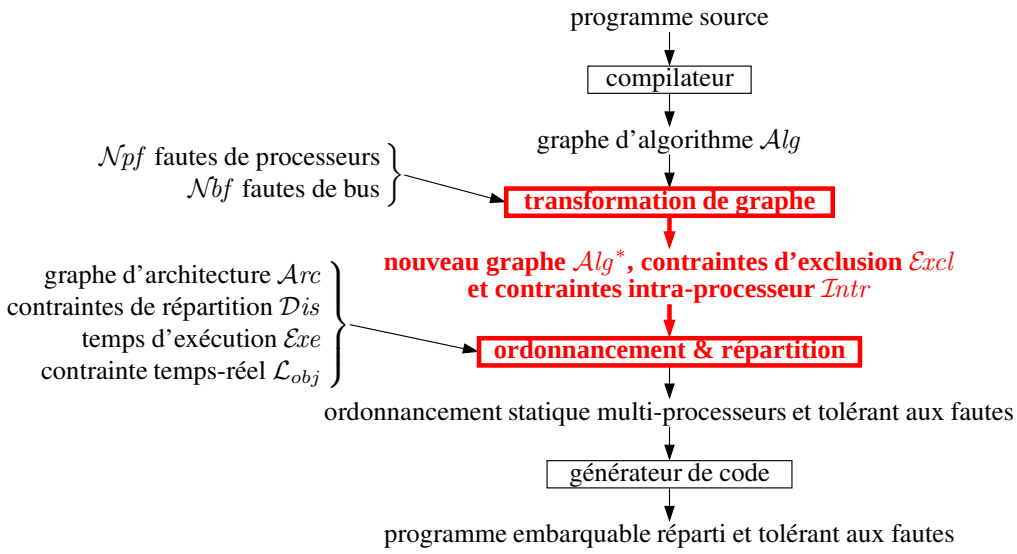


FIG. 3.18 – Schéma général de la méthode AAA-TB.

Dans les sections suivantes, je présente successivement la phase de transformation de graphe, puis le mécanisme de communication, les résultats d'exécution de la méthode AAA-TB, et enfin un état de l'art. Comme je l'ai dit dans la section 3.4.1, l'heuristique gloutonne d'ordonnancement & répartition est similaire à celle d'AAA. La seule modification par rapport à l'algorithme 3.1 concerne la prise en compte des contraintes d'exclusion $\mathcal{E}xcl$ et des contraintes intra-processeur $\mathcal{I}ntr$ lors de l'implantation d'une nouvelle opération et des dépendances de données qui lui sont nécessaires.

3.4.3.4 Transformation de graphe dans la méthode AAA-TB

En accord avec la section 3.4.1, le moyen choisi pour tolérer les $\mathcal{N}pf$ fautes requises de processeurs est la **réplication active des opérations** ; c'est-à-dire que pour chaque opération X de $\mathcal{A}lg$, il y aura $\mathcal{N}pf$+1 répliques identiques de X exécutées sur autant de processeurs distincts. Quant au moyen choisi pour tolérer les $\mathcal{N}bf$ fautes de bus requises, c'est la **réplication passive des communications** avec **fragmentation des données** : d'une part la réplication passive des communications évite de sérialiser trop de communications sur un même bus, et d'autre part la fragmentation des données améliore la détection et le recouvrement d'erreur. Par conséquent, chaque dépendance de données sera fragmentée en $\mathcal{N}bf$+1 paquets qui devront être transmis sur autant de bus en parallèle. Surtout, seule *une* des $\mathcal{N}pf$+1 répliques de l'opération source sera choisie pour transmettre ces $\mathcal{N}bf$+1 paquets à toutes les répliques de l'opération destination : cette réplique, appelée **réplique primaire**, sera celle dont la date de terminaison sera la plus petite. L'ordonnancement étant statique, ces dates de terminaison sont faciles à calculer à partir des temps d'exécution sur les processeurs (les WCET) et des temps de transmission sur les bus (les WCTT). L'idée est donc de tirer parti de la possibilité de diffusion d'une communication à tous les processeurs connectés à un même bus. Les autres répliques de l'opération source sont appelées **répliques secondaires** et elles n'ont pas à envoyer de communication, sauf en cas de défaillance du processeur implantant la réplique primaire. J'explique les détails dans la section 3.4.3.6.

Donc, chaque opération est dupliquée $\mathcal{N}pf$+1 fois : je note X^i la i-ième réplique de X, X^1 la réplique primaire, et $\mathcal{R}ep$(X) l'ensemble de toutes les répliques de X. De plus, chaque dépendance de données X▷Y est transformée en $\mathcal{N}bf$+1 nouvelles dépendances de données : je note $(X\triangleright Y)|^j$ la j-ième dépendance de données issue de X▷Y et $\mathcal{R}ep$(X▷Y) l'ensemble de toutes les dépendances de données issues de X▷Y ; T étant la donnée transmise par X▷Y, la donnée transmise par $(X\triangleright Y)|^j$ est le j-ième paquet obtenu en fragmentant T, que je note $T|^j$; enfin, je note $T{=}T|^1 \bullet T|^2 \bullet \ldots \bullet T|^{\mathcal{N}bf+1}$ où '•' est l'opérateur de défragmentation. Sur chaque bus B, le temps de transmission $\mathcal{E}xe\langle(X\triangleright Y)|^j, B\rangle$ doit être recalculé en tenant compte de la nouvelle taille du paquet et des caractéristiques du bus (bande passante, temps d'accès...) ; enfin, une opération de fragmentation F est ajoutée juste après la réplique primaire X^1, et une opération de défragmentation D^i est ajoutée juste avant chaque réplique Y^i. Les $\mathcal{N}bf$+1 dépendances de données $\mathcal{R}ep$(X▷Y) sont connectées aux $\mathcal{N}pf$+1 répliques $\mathcal{R}ep$(D) sans qu'il soit besoin de distinguer ces dernières ; en effet, les communications s'effectuant via des bus, *tous* les processeurs reçoivent la donnée transmise. Ceci est illustré dans la figure 3.19.

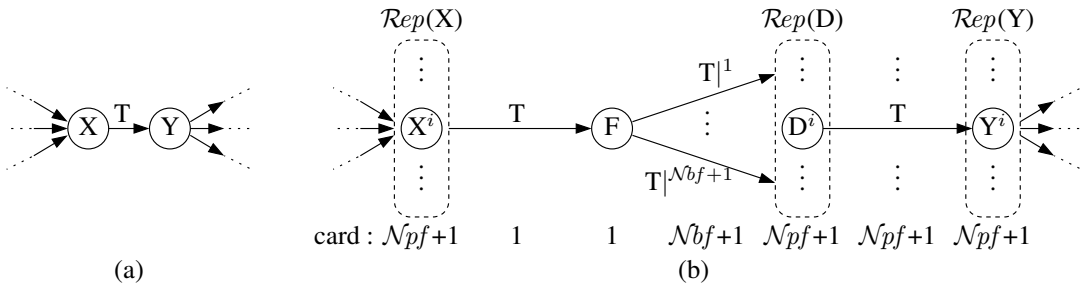


FIG. 3.19 – Transformation du graphe $\mathcal{A}lg$ (a) en graphe $\mathcal{A}lg^*$ (b) : réplication des opérations en $\mathcal{N}pf$+1 exemplaires, réplication des dépendances de données en $\mathcal{N}bf$+1 exemplaires, ajout de l'opération de fragmentation et des $\mathcal{N}pf$+1 opérations de défragmentation, et connexion des dépendances de données.

En plus du graphe $\mathcal{A}lg^*$, les contraintes de placement suivantes sont produites :

$$\begin{aligned}\mathcal{E}xcl &= \{X^1,\ldots,X^{\mathcal{N}pf+1}\},\{D^1,\ldots,D^{\mathcal{N}pf+1}\},\{Y^1,\ldots,Y^{\mathcal{N}pf+1}\},\\ &\quad \{(F\triangleright\mathcal{R}ep(D))|^1,\ldots,(F\triangleright\mathcal{R}ep(D))|^{\mathcal{N}bf+1}\}.\\ \mathcal{I}ntr &= \{X^1,F\},\{D^1,Y^1\},\ldots,\{D^{\mathcal{N}pf+1},Y^{\mathcal{N}pf+1}\}.\end{aligned}$$

Enfin, notons qu'à cette étape, la réplique primaire X^1 de X n'est pas encore connue. Elle ne le sera qu'au moment de l'étape d'ordonnancement & répartition, puisque c'est seulement à ce moment qu'il sera possible de calculer la date de terminaison de chaque réplique de X. Par conséquent, la dépendance de données $X^1 \triangleright F$ est laissée non-connectée, la connexion effective restant à la charge de l'algorithme d'ordonnancement & répartition.

3.4.3.5 Exemple

Je prends comme exemple le graphe d'algorithme $\mathcal{A}lg$ de la figure 3.20(a) et le graphe d'architecture $\mathcal{A}rc$ de la figure 3.20(b). L'utilisateur désire tolérer deux fautes de processeurs ($\mathcal{N}pf$=2) et une faute de bus ($\mathcal{N}bf$=1). Le graphe transformé $\mathcal{A}lg^*$ correspondant est dans la figure 3.20(c) : il contient, de gauche à droite, trois répliques de X (car $\mathcal{N}pf$+1=3), une opération de fragmentation F, deux communications (car $\mathcal{N}bf$+1=2) avec chacune un des fragments de la donnée T, trois opérations de défragmentation, et trois répliques de Y. De plus, les contraintes de placement suivantes sont produites :

$$\begin{aligned} \mathcal{E}xcl &= \{X^1, X^2, X^3\}, \{D^1, D^2, D^3\}, \{Y^1, Y^2, Y^3\}, \{(F \triangleright \mathcal{R}ep(D))|^1, (F \triangleright \mathcal{R}ep(D))|^2\}. \\ \mathcal{I}ntr &= \{X^1, F\}, \{D^1, Y^1\}, \{D^2, Y^2\}, \{D^3, Y^3\}. \end{aligned}$$

L'ordonnancement multiprocesseur obtenu à partir de ce graphe $\mathcal{A}lg^*$ et de ces contraintes de placement est dans la figure 3.20(d). La réplique primaire de X est X^1 car c'est elle dont la date de terminaison est la plus petite. L'opération de fragmentation F est donc ordonnancée sur P_1, juste entre X^1 et Y^1. Sur P_1 et P_3, la dépendance de données est transformée en une communication intra-processeur, donc les opérations de défragmentation D^1 et D^2 sont inutiles. Seule Y^3 sur P_2 a donc besoin de défragmenter les deux paquets transmis par les bus B_1 et B_2, et l'opération de défragmentation D^3 est donc ordonnancée sur P_2, juste avant Y^3.

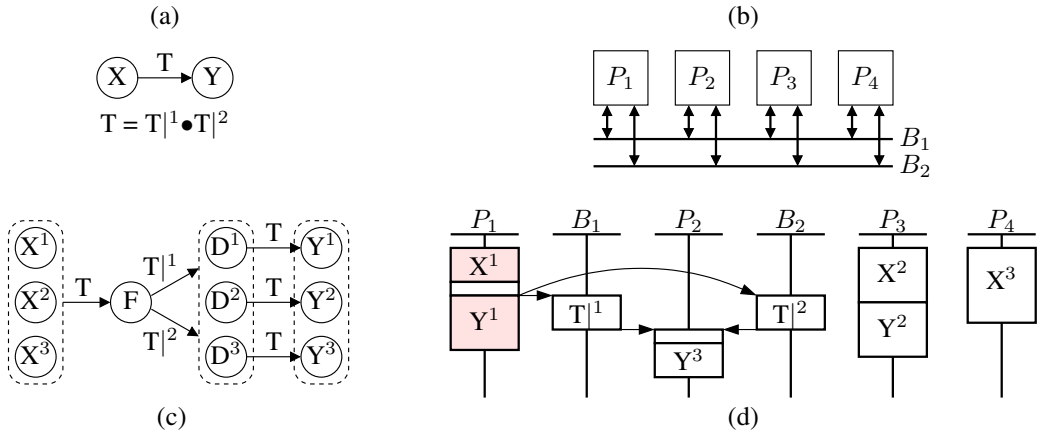


FIG. 3.20 – (a) Exemple de graphe $\mathcal{A}lg$; (b) Exemple de graphe $\mathcal{A}rc$; (c) Graphe transformé $\mathcal{A}lg^*$; (d) Ordonnancement multiprocesseur.

3.4.3.6 Mécanisme de communication

L'utilisation de la réplication hybride (au sens active / passive) pour tolérer à la fois les fautes des processeurs et des bus impose un processus complexe de détection et d'identification des défaillances. Le mécanisme de communication que je présente dans cette section permet de détecter rapidement les défaillances, ainsi que de distinguer rapidement entre les défaillances des processeurs et des bus. Comme

pour la méthode AAA-TP, j'utilise des chiens de garde sur les communications afin de détecter les défaillances (voir la section 3.4.2.5).

Les figures 3.21(a) et (b) montrent respectivement un exemple de graphe $\mathcal{A}lg$ et l'ordonnancement correspondant, dans lequel les processeurs ne sont toutefois pas explicités afin de ne pas surcharger la figure.

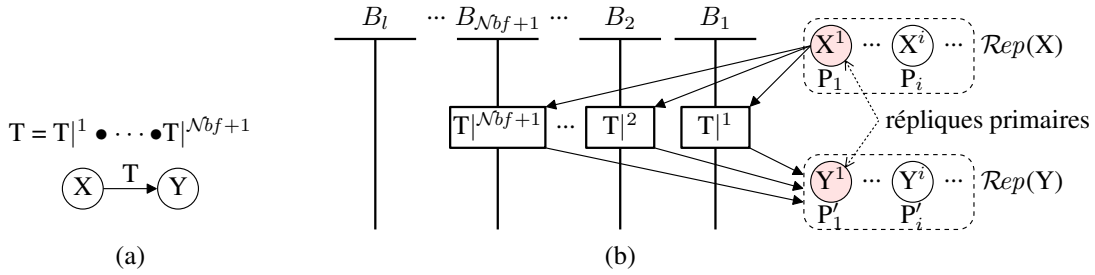


FIG. 3.21 – (a) Exemple de graphe $\mathcal{A}lg$; (b) Ordonnancement multiprocesseur correspondant.

À l'exécution de l'ordonnancement de la figure 3.21(b), trois cas peuvent se produire :

1. **Tous les paquets $\mathbf{T|^i}$ envoyés par $\mathbf{X^1}$ sont reçus**. Dans ce cas, chaque réplique de Y défragmente ces paquets et commence son exécution. Les autres répliques de X reçoivent aussi ces paquets, qu'elles ignorent.
2. **Aucun des paquets $\mathbf{T|^i}$ envoyés par $\mathbf{X^1}$ n'est reçu**. Cela fait donc $\mathcal{N}bf$+1 paquets, et comme, par hypothèse, au plus $\mathcal{N}bf$ bus peuvent défaillir, cela signifie que le processeur P_1 censé exécuter X^1 est défaillant. Pour compenser cela, une autre réplique de X est choisie pour renvoyer tous les paquets $T|^i$ via les mêmes bus. Toutes les répliques de X sont classées par ordre croissant de leur date de terminaison, de telle sorte que celle qui est choisie pour renvoyer les paquets est celle ayant la plus petite date de terminaison, juste après la réplique primaire X^1.
3. **Certains paquets $\mathbf{\{T|^i, \ldots, T|^k\}}$ envoyés par $\mathbf{X^1}$ ne sont pas reçus**. Soit P_r le processeur qui fait ce diagnostic, soit $\mathcal{T}^-$ l'ensemble de ces paquets non reçus, et soit $\mathcal{B}^- = \{B^i, \ldots, B^k\}$ l'ensemble des bus censés les avoir acheminés. Puisque d'autres paquets ont été reçus, cela signifie que P_1 n'est pas défaillant, donc que ce sont les bus de $\mathcal{B}^-$ qui sont défaillants. La même réplique X^1 renvoie donc les paquets $\mathcal{T}^-$, via d'autres bus choisis dans l'ensemble $\mathcal{B}\backslash\mathcal{B}^-$. Tous les processeurs marquent les bus de $\mathcal{B}-$ comme étant défaillants.

   Au cours des itérations suivantes, les données ne seront plus fragmentées qu'en $\mathcal{N}bf - |\mathcal{B}^-| + 1$ paquets, puisque le nombre de fautes de bus à tolérer aura diminué de $|\mathcal{B}^-|$.

   Si l'hypothèse de fautes avait imposé la distinction entre la défaillance totale et la défaillance partielle des bus (où seulement *certaines* des connexions bus–processeurs sont défaillantes), alors le mécanisme de communication aurait été beaucoup plus compliqué. En particulier, le cas 3 aurait nécessité de faire renvoyer les paquets $\mathcal{T}^-$ par une *autre* réplique X^i ($i \neq 1$), donc exécutée sur un autre processeur P^i, via les *mêmes* bus $\mathcal{B}^-$, ceci afin de distinguer entre, d'une part, les défaillances des connexions entre P_1 et $\mathcal{B}^-$, et d'autre part, les défaillances des connexions entre $\mathcal{B}^-$ et les processeurs exécutant les répliques de Y.

En résumé, ce mécanisme de communication présente les avantages suivants : détection rapide des défaillances ; distinction rapide entre les défaillances des processeurs et celles des bus ; et recouvrement rapide après une défaillance.

Enfin, si les caractéristiques physiques des bus utilisés nécessitent une *taille minimale* pour les paquets, alors cette donnée doit être prise en compte par le mécanisme de communication. Notamment, la procédure de fragmentation doit ajuster le nombre de paquets en fonction de leur taille minimale. Cela

concerne également le cas 3 ci-dessus, puisque le nombre de paquets obtenus après fragmentation est diminué de un à chaque fois que la défaillance totale d'un bus est déclarée.

3.4.3.7 État de l'art sur la génération d'ordonnancements tolérants aux fautes pour architectures réparties avec bus

De nombreux travaux ont été menés sur la tolérance aux fautes des processeurs et des bus en utilisant des techniques matérielles (voir par exemple [68]) ou des techniques logicielles (notamment SIFT [118], MAFT [64] et GUARDS [91]). Mais, à ma connaissance, très peu de travaux existent sur l'utilisation de techniques logicielles pour la génération d'ordonnancements multiprocesseur et tolérants aux fautes destinés à être exécutés sur des architectures génériques.

Dans [59], Kandasamy et al. partent d'un ensemble de graphes de tâches et d'un ensemble de processeurs identiques. Ils ordonnancent tout d'abord les tâches sur les processeurs sans redondance, puis ils déterminent le nombre minimal de bus nécessaires à l'acheminement dans les délais de k répliques de chaque message. C'est donc de la réplication active des communications. Les bus sont supposés être identiques, et le protocole de communication considéré est TDMA (« Time Division Multiple Access » [115]), c'est-à-dire le protocole utilisé dans les réseaux TTP et FLEXRAY. Pour déterminer ce nombre minimal de bus, les auteurs proposent un algorithme d'agrégation en groupes : initialement, chaque réplique de chaque message est dans un groupe distinct ; puis, les messages sont regroupés de telle sorte que deux répliques d'un même message ne sont jamais dans le même groupe, tous les messages sont transmis avant leur date limite, et le nombre de fenêtres de communications (les « slots » du protocole TDMA) nécessaires est minimisé pour chaque bus ; enfin, chaque groupe de message est affecté à un bus distinct.

Le protocole de communication utilisé dans l'architecture TTA (« Time Triggered Architecture » [67]) est basé sur les mêmes principes que le protocole TDMA. Ici, c'est le protocole de communication lui-même (dans sa version TTP/C) qui fournit un service de transmission de message tolérant aux fautes et avec délai connu, ainsi qu'un service de synchronisation d'horloge, et un service de détection des processeurs fautifs. Pour tolérer les fautes des bus il est possible de rajouter des bus supplémentaires, tandis que pour tolérer les fautes des processeurs il est possible de construire des unités tolérantes aux fautes contenant trois processeurs et un voteur. Toutefois, les auteurs n'abordent pas le problème précis de la génération d'ordonnancements tolérants aux fautes.

Dans le contexte du routage multichemin dans les réseaux sans fil (avec liens de communication logiques de type point-à-point), Dulman et al. utilisent les codes correcteurs d'erreurs pour fragmenter chaque message en k paquets, tout en ajoutant de la redondance aux données transmises [30]. L'objectif est que le processeur destination n'ait besoin que de E_k paquets parmi les k paquets transmis pour reconstruire le message d'origine, ce qui permet de tolérer la perte de $k - E_k$ paquets. Pour chaque message, le nombre k est calculé en tenant à jour, au cour de l'exécution, le coefficient de réputation de chaque processeur (un coefficient de réputation faible indique que le processeur a échoué à transmettre un message dans le passé, et ne sera pas utilisé pour des routages futurs), et en calculant le nombre de chemins disjoints entre la source et la destination : ce nombre est justement pris pour k. Puis, le nombre E_k est choisi en fonction de l'échange entre la probabilité de transmission correcte et le coût de la redondance ajoutée.

3.4.3.8 Résultats d'exécution de l'heuristique d'AAA-TB et discussion

La figure 3.22 présente tout d'abord un exemple d'ordonnancement produit par AAA-TB pour $\mathcal{N}pf = 1$ et $\mathcal{N}bf = 0$, avec les graphes de la figure 3.6 (où les trois liens de communication point-à-point ont été remplacés par trois bus B1, B2 et B3) et les caractéristiques d'exécution de la figure 3.7. Dans la figure 3.22, le bloc « X » désigne la première réplique de l'opération X (celle dont la date de

terminaison est la plus petite), alors que le bloc « X_k# » désigne la k-ième réplique de cette même opération X. La longueur de cet ordonnancement est égale à 7.5 unités de temps.

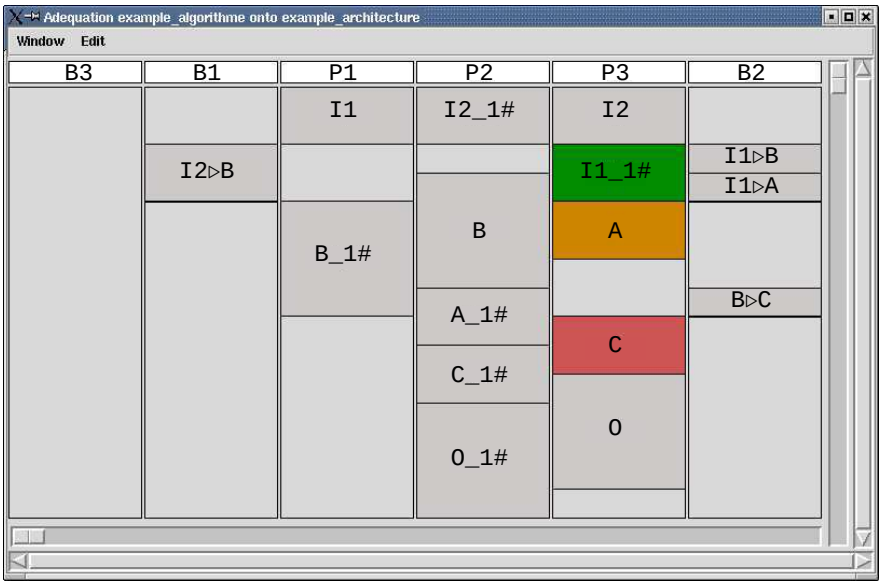
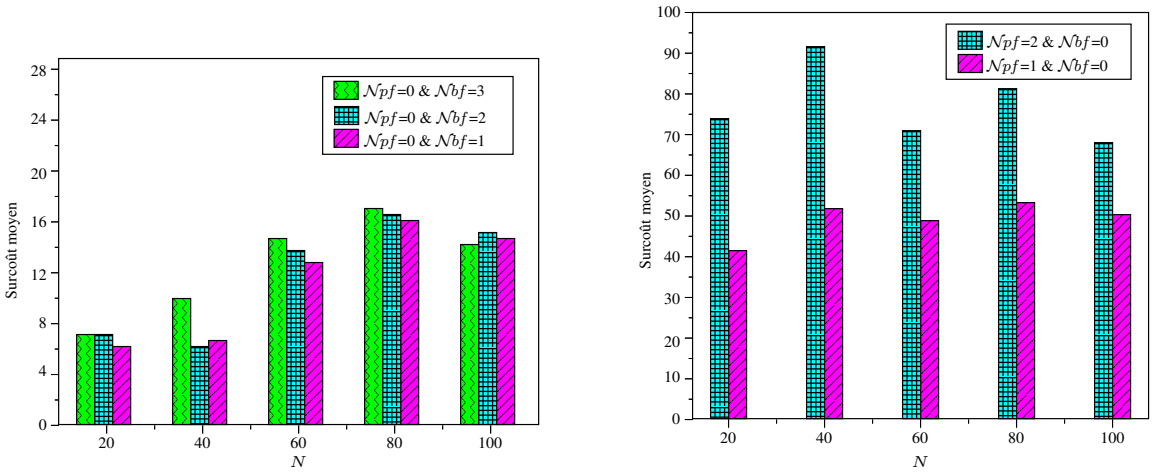


FIG. 3.22 – Ordonnancement produit par AAA-TB ($\mathcal{N}pf$=1,$\mathcal{N}bf$=0).

Il n'existe dans la littérature aucune autre méthode d'ordonnancement & répartition qui soit comparable à AAA-TB. J'évalue donc les performances d'AAA-TB en mesurant le surcoût dû à la tolérance aux fautes en fonction de plusieurs paramètres : N, CCR , $\mathcal{N}pf$ et $\mathcal{N}bf$. Le surcoût est calculé par la formule (3.6) :

$$\text{surcoût} = \frac{\text{longueur(AAA-TB}(\mathcal{N}pf,\mathcal{N}bf)) - \text{longueur(AAA)}}{\text{longueur(AAA)}} \times 100 \qquad (3.6)$$



(a) (b)

FIG. 3.23 – Surcoût moyen d'AAA-TB par rapport à AAA en fonction de N : (a) pour $\mathcal{N}pf = 0$ et $\mathcal{N}bf \in \{1, 2, 3\}$; (b) pour $\mathcal{N}pf \in \{1, 2\}$ et $\mathcal{N}bf = 0$.

Les figures 3.23(a) et (b) illustrent le surcoût moyen d'AAA-TB par rapport à AAA, respectivement

en faisant varier $\mathcal{N}bf$ et $\mathcal{N}pf$. Chaque point est la moyenne de la longueur de l'ordonnancement calculé pour 100 graphes $\mathcal{A}lg$ sur le même graphe $\mathcal{A}rc$ à 5 processeurs et à 4 bus, avec CCR$= 1$. Dans la figure 3.23(a), le surcoût est très faible (variant entre 6% et 18%), et augmente légèrement en fonction de N. La raison en est que les opérations ne sont pas dupliquées (car $\mathcal{N}pf = 0$) et que le moyen utilisé pour tolérer les fautes des bus est la réplication passive des communications. Dans la figure 3.23(b), le surcoût est 45% pour $\mathcal{N}pf = 1$ et 75% pour $\mathcal{N}pf = 2$. La raison en est que le moyen choisi pour tolérer les fautes des processeurs est la réplication active des opérations. Toutefois, ces chiffres sont bien plus faibles que ce qu'on pourrait attendre, à savoir 100% pour $\mathcal{N}pf = 1$ (car chaque opération est ordonnancée deux fois) et 200% pour $\mathcal{N}pf = 2$ (car chaque opération est ordonnancée trois fois), ce qui montre qu'AAA-TB est très efficace. Enfin, on ne constate pas de variation significative du surcoût moyen quand N augmente.

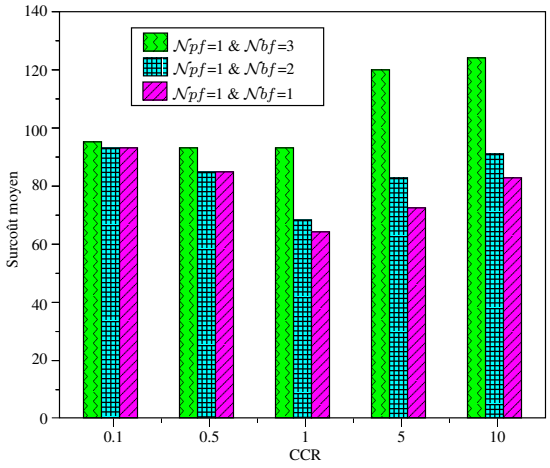


FIG. 3.24 – Surcoût moyen d'AAA-TB par rapport à AAA en fonction de CCR.

La figure 3.24 illustre le surcoût moyen d'AAA-TB par rapport à AAA, pour $\mathcal{N}pf = 1$ et $\mathcal{N}bf \in \{1, 2, 3\}$, et en faisant varier CCR. Chaque point est la moyenne de la longueur de l'ordonnancement calculé pour 100 graphes $\mathcal{A}lg$ de 40 tâches sur le même graphe $\mathcal{A}rc$ à 5 processeurs et à 4 bus, en fonction de CCR. Grâce à la fragmentation des données, pour CCR< 1 les performances d'AAA-TB sont presque identiques pour $\mathcal{N}bf = 1$ à 3. En revanche, pour CCR≥ 1, les performances se dégradent quand $\mathcal{N}bf$ augmente. Quant à l'influence de CCR, elle est limitée quand $\mathcal{N}bf \leq 2$, là encore en raison de la fragmentation des données. Cela n'est plus vrai quand $\mathcal{N}bf = 3$ car le nombre de bus de $\mathcal{A}rc$ devient limitatif.

3.4.4 Discussion sur AAA-TP et AAA-TB

La comparaison des figures 3.15(b) et 3.24 donne le tableau suivant :

	Surcoût moyen	
CCR	AAA-TP	AAA-TB
0.5	92%	85%
1.0	143%	64%
5.0	182%	72%

FIG. 3.25 – Comparaison des surcoûts moyens d'AAA-TP et AAA-TB par rapport à AAA et pour des graphes $\mathcal{A}lg$ de 40 tâches.

où $\mathcal{A}rc$ est composée de 6 processeurs complètement connectés pour AAA-TP, et de 5 processeurs et 4 bus pour AAA-TB. Le parallélisme intrinsèque de $\mathcal{A}rc$ était donc légèrement supérieur pour AAA-TP que pour AAA-TB. Malgré cela, le surcoût moyen est nettement supérieur avec AAA-TP. Cela s'explique par le fait qu'AAA-TB utilise la réplication *passive* des communications, alors qu'AAA-TP utilise la réplication *active* des communications.

Les méthodes AAA-TP et AAA-TB sont applicables en pratique dans les domaines de prédilection de SYNDEX, c'est-à-dire les systèmes embarqués pour lesquels l'utilisateur impose une contrainte temporelle stricte et ne veut pas avoir recours à un système d'exploitation pour ordonnancer ses tâches de calcul. Un exemple typique est la flotte de véhicules automatiques actuellement en cours de déploiement dans des sites fermés comme l'aéroport de Londres-Heathrow ou semi-fermés comme le Parc des Expositions de Rome, et inspirés du CYCAB [5, 30]. L'architecture informatique du CYCAB comporte cinq nœuds de calcul reliés par deux bus CAN. Le logiciel de pilotage automatique est entièrement programmé en SYNDEX, et l'utilisation de la méthode AAA-TB (ici, ce choix est imposé par le type de media de communication) permettrait de produire automatiquement un ordonnancement tolérant à une faute de processeur et à une faute de bus, puis un exécutif réparti synchronisé tolérant à ces mêmes fautes.

L'étude des résultats obtenus grâce aux deux méthodes AAA-TP et AAA-TB montre que le surcoût dû à la tolérance aux fautes est très souvent prohibitif. À mon avis, la raison en incombe, d'une part au choix de la réplication active pour tolérer les fautes, et d'autre part au fait que le taux de réplication est très élevé (il est égal à 3 pour tolérer 2 fautes de processeurs). Même si la réplication active présente l'inconvénient d'un surcoût plus grand qu'avec la réplication passive, elle possède en revanche deux avantages importants : d'une part les ordonnancements obtenus sont *statiques*, ce qui facilite grandement le calcul du $\mathsf{C}_{\mathsf{max}}$ et donc la vérification de la contrainte temps-réel (voir la section 3.3.3), et d'autre part le déploiement du système ne requiert pas de mécanisme d'exécution compliqué. Au contraire, dans un ordonnancement dynamique, les tâches doivent être *interruptibles*, ce qui qui nécessite un système d'exploitation temps-réel. Or dans de nombreuses applications à sûreté critiques, par exemple pour les commandes de vols des avions de ligne, cela est totalement exclu. Il est par conséquent difficile de se passer de la réplication active dès qu'on veut *garantir* une borne sur le temps d'exécution maximal.

C'est pour cette raison que j'ai étudié d'autres solutions, toujours basées sur la réplication active, mais pour lesquelles le taux de réplication est moindre, c'est-à-dire mieux adapté aux besoins de l'utilisateur. Je présente donc dans la section 3.4.5 une méthode bicritère originale qui garantit que l'ordonnancement obtenu satisfait à la fois une contrainte temps-réel et une contrainte de fiabilité, toutes deux exprimées par l'utilisateur. Cette méthode est basée sur AAA. Puis, je présente dans la section 3.5 une méthode originale de fusion d'ordonnancements simples qui garantit que l'ordonnancement obtenu tolère un ensemble de patrons de fautes désirés par l'utilisateur. Cette méthode peut être utilisée avec n'importe quelle heuristique pour générer les ordonnancements simples, en particulier celle de AAA.

3.4.5 Méthode bicritère pour garantir la fiabilité d'un système

En 2002, j'ai eu l'idée d'une méthode de génération d'ordonnancements multiprocesseur et fiables, basée sur la méthode AAA et utilisant la réplication active des opérations. Contrairement aux deux méthodes précédentes (sections 3.4.2 et 3.4.3), au lieu d'exiger qu'un certain nombre de fautes de processeurs et de liens de communication soient tolérées par l'ordonnancement multiprocesseur, l'utilisateur exige qu'une fiabilité minimale soit satisfaite par l'ordonnancement. Cela permet de dupliquer *moins* les opérations du graphe $\mathcal{A}lg$ puisqu'on aura le choix entre exécuter une opération sur *plusieurs* processeurs peu fiables ou sur *un seul* processeur très fiable. Par comparaison, dans les méthodes AAA-TP et AAA-TB chaque opération est dupliquée le *même nombre de fois* quelles que soient les fiabilités des processeurs. De plus, l'utilisateur continue à exiger que l'ordonnancement ne dépasse pas une longueur maximale. Par analogie avec l'acronyme AAA, cette méthode s'appelle AAA-F : « Adéquation

Algorithme Architecture avec Fiabilité ». Les travaux présentés dans cette section ont été publiés dans l'article [5], dans les rapports de DEA d'Ismail Assayad [4] et de Nicolas Leignel [78], ainsi que dans la thèse d'Hamoudi Kalla [57].

3.4.5.1 Énoncé du problème

Problème 3.6 (AAA - F) *Soit un graphe d'algorithme $\mathcal{A}lg$, un graphe d'architecture $\mathcal{A}rc$ avec liens de communication point-à-point, un tableau $\mathcal{E}xe$ des WCET des opérations de $\mathcal{A}lg$ sur les processeurs de $\mathcal{A}rc$ et des WCTT des dépendances de données de $\mathcal{A}lg$ sur les liens de communication de $\mathcal{A}rc$, des contraintes de distribution $\mathcal{D}is$, un tableau $\mathcal{R}el$ des fiabilités des processeurs et des liens de communication de $\mathcal{A}rc$.*

Trouver un ordonnancement multiprocesseur et statique de $\mathcal{A}lg$ sur $\mathcal{A}rc$, qui respecte $\mathcal{D}is$, dont la longueur calculée grâce à $\mathcal{E}xe$ soit minimale, et donc la fiabilité calculée grâce à $\mathcal{R}el$ soit maximale.

De même que le problème 3.1, ce problème est NP-difficile, et donc la méthode AAA-F que je propose pour le résoudre se base sur un algorithme heuristique d'ordonnancement & répartition. Deux contraintes, une temps-réel $\mathcal{L}_{obj}$ et une de fiabilité $\mathcal{F}_{obj}$, doivent de plus être respectée, donc il faut vérifier à la fin que la longueur de l'ordonnancement multiprocesseur obtenu est inférieure à $\mathcal{L}_{obj}$ et que sa fiabilité est supérieure à $\mathcal{F}_{obj}$.

3.4.5.2 Hypothèse de faute

Les composants matériels de $\mathcal{A}rc$ (processeurs et liens de communication) sont supposés être à silence sur défaillance. Les défaillances sont temporaires. La durée maximale d'une défaillance est telle qu'une défaillance affecte une seule opération. Cela veut dire que la défaillance d'un processeur n'affecte que l'opération qui est en cours d'exécution sur le processeur au moment de la défaillance, et pas les opérations suivantes (même chose pour les dépendance de données sur les liens de communication). De plus, l'occurrrence des défaillances de chaque processeur p de $\mathcal{A}rc$ suit une loi de Poisson de paramètre constant λ_p, appelé « taux de défaillance de p par unité de temps ». De façon similaire, la probabilité de défaillance de chaque lien de communication l de $\mathcal{A}rc$ suit une loi de Poisson de paramètre constant λ_l, appelé « taux de défaillance de l par unité de temps ». Enfin, les défaillances des composants matériels sont des événements statistiquement indépendants. Je reviendrai sur ces notions dans la section 3.4.5.6.

3.4.5.3 Schéma général de la méthode AAA-F

La figure 3.26 montre le schéma général de la méthode AAA-F.

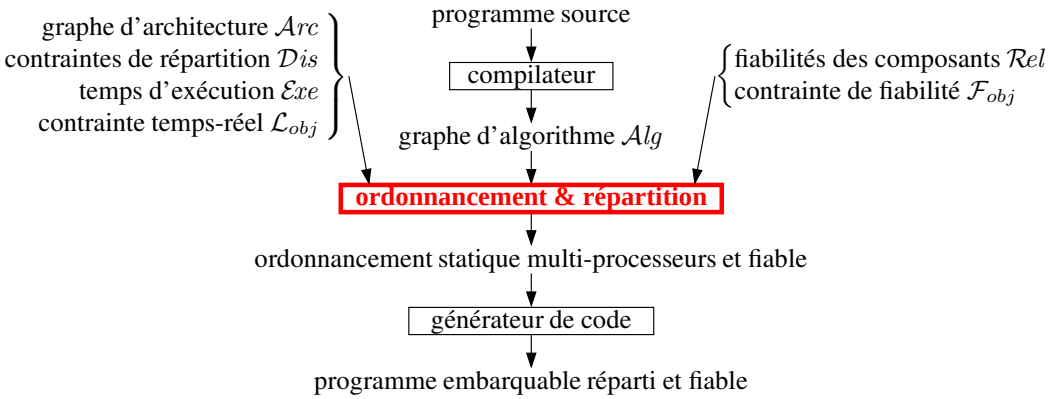


FIG. 3.26 – Schéma général de la méthode AAA-F.

3.4.5.4 Principe de la méthode AAA-F

La méthode AAA-F s'appuie sur une heuristique gloutonne de type ordonnancement de liste. Le principe de cette méthode d'ordonnancement & répartition est d'utiliser la mesure de la fiabilité d'un ordonnancement afin de guider, pour chaque opération X de $\mathcal{A}lg$, le choix du meilleur sous-ensemble de processeurs pour exécuter X, et donc le taux de réplication de X. Si le sous-ensemble choisi est un singleton, alors X n'est pas dupliquée. En revanche, si le sous-ensemble contient plusieurs processeurs, alors X est dupliquée sur chacun d'entre eux. J'ai donc deux objectifs à optimiser dans cette méthode : d'une part la fiabilité de l'ordonnancement doit être maximale et supérieure à $\mathcal{F}_{obj}$, et d'autre part sa longueur doit être minimale et inférieure à $\mathcal{L}_{obj}$. Or ces deux critères sont *antagonistes* ! En effet, pour augmenter la fiabilité, il faut plus de réplication, mais cela a tendance aussi à augmenter la longueur de l'ordonnancement. Il s'agit donc d'un **problème d'optimisation bicritère**, avec deux critères antagonistes.

3.4.5.5 Introduction à l'ordonnancement et l'optimisation bicritère

Optimisation bicritère. Une notion fondamentale pour les problèmes d'optimisation bicritère est celle d'optimum de Pareto [113]. Cette notion s'applique de façon générale aux problèmes d'optimisation multicritère ; je la présente ici dans le cas particulier bicritère. Une solution du problème d'optimisation est un **optimum de Pareto** si et seulement si elle est optimale par rapport aux deux critères, c'est-à-dire qu'il n'existe aucune autre solution qui soit meilleure pour les deux critères à la fois ; mais il peut tout à fait exister d'autres solutions qui soient meilleures pour un des deux critères. Formellement, la structure d'ordre utilisée dans $\mathbb{R}^2$ est définie par $x \leq y \Leftrightarrow \forall 1 \leq i \leq 2, x_i \leq y_i$ et $x = y \Leftrightarrow \forall 1 \leq i \leq 2, x_i = y_i$, où x_i est la i-ième composante du vecteur x. Comme avec toute structure d'ordre, on écrit $x \neq y \Leftrightarrow \neg(x = y)$, et $x < y \Leftrightarrow x \leq y \wedge x \neq y$. Ainsi donc, x est un optimum de Pareto ssi $\nexists y$ tel que $Z(y) \leq Z(x)$, où Z est la fonction de l'espace des solutions vers l'espace des critères (ici $\mathbb{R}^2$). Si $\nexists y$ tel que $\forall 1 \leq i \leq 2, Z_i(y) \leq Z_i(x)$ avec au moins une inégalité stricte, alors x est un **optimum de Pareto strict**. Si $\nexists y$ tel que $\forall 1 \leq i \leq 2, Z_i(y) < Z_i(x)$, alors x est un **optimum de Pareto faible** (notons au passage que strict implique faible).

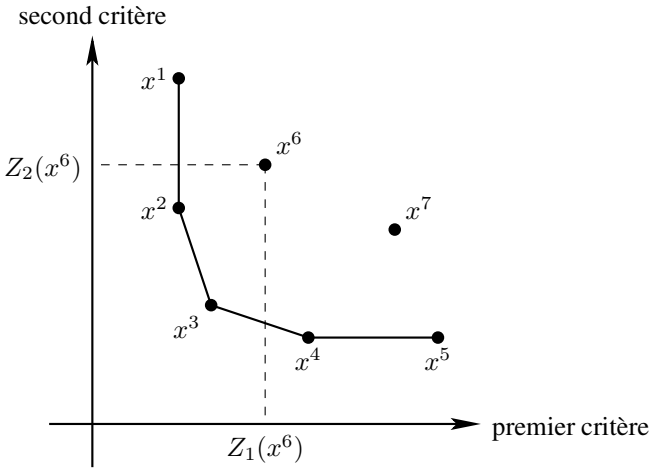


FIG. 3.27 – Optima et courbe de Pareto pour un problème de minimisation bicritère.

Graphiquement, on représente chaque solution du problème d'optimisation par un point dans le plan $\mathbb{R}^2$, dont l'abcisse est la mesure du premier critère (c'est-à-dire $Z_1(x)$) et dont l'ordonnée est la mesure du second critère (c'est-à-dire $Z_2(x)$). Ceci est valable dans le cas de deux critères ; dans le cas de n

critères, on est dans un espace à n dimensions. Sans perte de généralité, supposons que le problème consiste à minimiser les deux critères. La figure 3.27 illustre la notion d'optimum de Pareto : chaque point numéroté de x^1 à x^7 est une solution du problème d'optimisation bicritère ; l'enveloppe convexe des points est appelée **courbe de Pareto** (même si en réalité c'est une ligne brisée plutôt qu'une courbe) ; tous les points qui sont sur la courbe de Pareto (ici x^1, x^2, x^3, x^4 et x^5) sont des optima de Pareto ; les points x^1 et x^5 sont des optima faibles, alors que les point x^2, x^3 et x^4 sont des optima stricts.

En général, il n'existe pas de solution qui optimise *les deux critères à la fois*. C'est le cas dans la figure 3.27 : les points x^1 et x^2 optimisent le premier critère mais pas le second ; pour les points x^4 et x^5 c'est le contraire ; quant au point x^3 il n'en optimise aucun. Le choix d'une solution est donc un compromis entre les deux critères. Les optima de Pareto jouent un rôle important dans la résolution des problèmes d'optimisation bicritères puisqu'ils permettent à l'utilisateur de choisir son compromis parmi les meilleures solutions. Malheureusement, la recherche des optima de Pareto est un problème NP-difficile [113].

Plusieurs approches existent dans la littérature pour résoudre les problèmes d'optimisation bicritères :

1. **Agrégation des deux critères en un seul**, pour se ramener à un problème d'optimisation monocritère. Par exemple, on peut réaliser une combinaison linéaire des deux critères.
2. **Transformation d'un critère en contrainte**, puis résolution du problème en optimisant l'autre critère *sous contrainte* du premier.
3. **Hiérarchisation des critères**, ce qui permet d'ordonner totalement les optima de Pareto, puis résolution du problème dans l'ordre des critères : on optimise le premier critère ; puis, parmi les solutions optimales pour le premier critère, on optimise le second critère (et ainsi de suite dans le cas multicritère).
4. **Interaction avec un utilisateur**, afin de guider la recherche d'un optimum en privilégiant un des critères.

3.4.5.6 Modèle de fiabilité et calcul de la fiabilité

J'attaque maintenant le problème du calcul de la fiabilité d'un ordonnancement, éventuellement partiel, d'un graphe $\mathcal{A}lg$ sur une architecture répartie $\mathcal{A}rc$.

Modèle de fiabilité. Classiquement, j'associe à chaque composant matériel de $\mathcal{A}rc$ (processeur ou medium de communication) un **taux de défaillance par unité de temps** constant, noté λ [81]. Je suppose que la **l'occurrence d'une défaillance** d'un composant suit une loi de Poisson à paramètre constant : la probabilité qu'un composant de taux de défaillance λ soit opérationnel pendant une durée d, c'est-à-dire sa fiabilité, est donc égale à $\exp^{-\lambda.d}$. De façon duale, la probabilité que ce même composant soit défaillant est égale à $1 - \exp^{-\lambda.d}$.

Les processeurs modernes à silence sur défaillance peuvent avoir un taux de défaillance de l'ordre de $10^{-6}/hr$. Par exemple, la probabilité qu'un processeur, dont le taux de défaillance est égal à $10^{-6}/hr$, soit opérationnel pendant 2 heures est $\exp^{-10^{-6}*2} \simeq 0.999998$. Quant à sa probabilité de défaillance pendant cette même durée, c'est $1 - \exp^{-10^{-6}*2} \simeq 0.000002$.

De plus, il faut distinguer les fautes **« à chaud »** des fautes **« à froid »**. Dans le premier cas, un composant ne peut subir une défaillance que quand il est actif, alors que dans le second cas un composant peut subir une défaillance n'importe quand. L'hypothèse de faute étant qu'une défaillance n'affecte qu'une seule opération, c'est le modèle « à chaud » qui est pertinent. Il faut donc calculer la fiabilité F de l'opération o placée sur le processeur p par la formule :

$$F(o,p) = \exp^{-\lambda_p.\mathcal{E}xe\langle o,p\rangle} \tag{3.7}$$

Quant à la probabilité de défaillance de l'opération o placée sur le processeur p, c'est le complémentaire de $F(o,p)$:

$$\bar{F}(o,p) = 1 - F(o,P) = 1 - \exp^{-\lambda_p.\mathcal{E}xe\langle o,p\rangle} \tag{3.8}$$

Dans le cas d'une dépendance de données $o \triangleright o'$ placée sur le lien de communication l, sa fiabilité est :

$$F(o \triangleright o', l) = \exp^{-\lambda_l.\mathcal{E}xe\langle o\triangleright o',l\rangle} \tag{3.9}$$

Ici aussi, la probabilité de défaillance de la dépendance de données $o \triangleright o'$ placée sur le lien de communication l est le complémentaire de $F(o \triangleright o', l)$:

$$\bar{F}(o \triangleright o', l) = 1 - F(o \triangleright o', l) = 1 - \exp^{-\lambda_l.\mathcal{E}xe\langle o\triangleright o',l\rangle} \tag{3.10}$$

Fiabilité d'un ordonnancement multiprocesseur. Un des problèmes difficiles qui se pose dans les algorithmes d'ordonnancement & répartition est le calcul *efficace* de la fiabilité d'un ordonnancement. L'efficacité est nécessaire puisque ce calcul doit être effectué chaque fois qu'une opération candidate est évaluée pour être placée sur un processeur (par la fonction de coût). La **fiabilité** d'un ordonnancement (multiprocesseur ou non) est la probabilité qu'il fonctionne correctement, c'est-à-dire la probabilité que les sorties de l'algorithme $\mathcal{A}lg$ qui est ordonnancé calcule correctement ses sorties. C'est donc la probabilité que chaque opération de sortie de $\mathcal{A}lg$ soit opérationnelle. Une opération Y est **opérationnelle** si et seulement si elle est exécutée sur au moins un processeur P qui est lui-même opérationnel pendant toute la durée d'exécution de Y, et si P reçoit toutes les données d'entrée de Y avant de démarrer son exécution, c'est-à-dire si, pour chaque dépendance de donnée X▷Y, il existe un chemin opérationnel entre au moins un des processeurs qui exécute X et P. Un processeur P est **opérationnel** pendant un intervalle de temps si et seulement si il ne subit pas de défaillance pendant cet intervalle. Enfin, un chemin est **opérationnel** pendant un intervalle de temps si et seulement si il est constitué de processeurs et de liens de communication opérationnels pendant cet intervalle.

Le calcul de la fiabilité d'un ordonnancement nécessite donc de calculer, pour chaque dépendance de donnée X▷Y, la probabilité qu'il existe un chemin opérationnel depuis le processeur source (celui qui exécute l'opération X) jusqu'au processeur destination (celui qui exécute l'opération Y). Or ce problème, connu sous le nom de **problème des paires terminales** (« terminal-pair problem » [12]), est NP-difficile dans le cas d'un réseau quelconque. C'est pour cette raison que de nombreux travaux se sont limités au cas des réseaux exempts de cycle [107, 108, 62, 63, 52, 51], parce que le calcul de la probabilité qu'un chemin soit opérationnel devient linéaire en temps.

Dogan et Özgüner ont quant à eux résolu cette difficulté en approximant le calcul de $\exp^x$ par $1+x$, ce qui simplifie grandement les calculs [28, 29]. Comme x est le taux de défaillance de la ressource et est très petit, cette approximation est pertinente. Surtout, c'est une *sous*-approximation, donc s'il est établi que la probabilité *approximée* qu'un chemin soit opérationnel est supérieure à un seuil donné, alors la probabilité *exacte* est elle aussi supérieure à ce seuil.

Bloc-Diagrammes de Fiabilité. Afin de calculer la fiabilité d'un ordonnancement multiprocesseur statique d'un graphe $\mathcal{A}lg$ sur une architecture $\mathcal{A}rc$ dans le modèle décrit ci-dessus, j'utilise la méthode des **blocs-diagrammes de fiabilité** (BDF) [81, 61, 87, 1]. Formellement, un BDF est un **graphe orienté** (N, E), dont chaque sommet de N est un **bloc** représentant un élément du système, et chaque arc de E est un **lien de causalité** entre deux blocs. Dans tout BDF, deux blocs particuliers sont identifiés : ce sont sa **source** S et sa **destination** D. Un BDF représente un système et est utilisé pour en calculer la fiabilité : un BDF est **opérationnel** si et seulement si il existe au moins un chemin opérationnel de S à D. Un chemin est opérationnel si et seulement si tous les blocs de ce chemin sont opérationnels. La probabilité qu'un bloc soit opérationnel est sa fiabilité. Par construction, la probabilité qu'un BDF soit opérationnel est donc égale à la fiabilité du système qu'il représente.

Dans le cas qui m'intéresse, le système est l'ordonnancement multiprocesseur, éventuellement partiel, de $\mathcal{A}lg$ sur $\mathcal{A}rc$ (voir les sections 3.4.5.1 et 3.4.5.4). Chaque bloc représente une opération o placée sur un processeur p, ou une dépendance de donnée $o \triangleright o'$ placée sur un lien de communication l. La fiabilité d'un bloc est donc calculée par la formule (3.7) pour une opération, et par la formule (3.9) pour une dépendance de données.

Pour le calcul de la fiabilité, je suppose que les défaillances des diverses ressources physiques sont des **événements statistiquement indépendants**. Sans cette hypothèse, le fait que des blocs soient dans plusieurs chemins de S à D rend le calcul de la fiabilité très compliqué. Pour des fautes matérielles cette hypothèse est raisonnable, mais il est à noter que ça n'est pas le cas pour des fautes logicielles [66].

Exemple. Considérons le graphe $\mathcal{A}lg$ de la figure 3.1 et l'architecture $\mathcal{A}rc$ de la figure 3.2(a). La figure 3.28 représente deux ordonnancements partiels de ce graphe $\mathcal{A}lg$ sur cette architecture $\mathcal{A}rc$, le premier sans réplication et le second avec réplication.

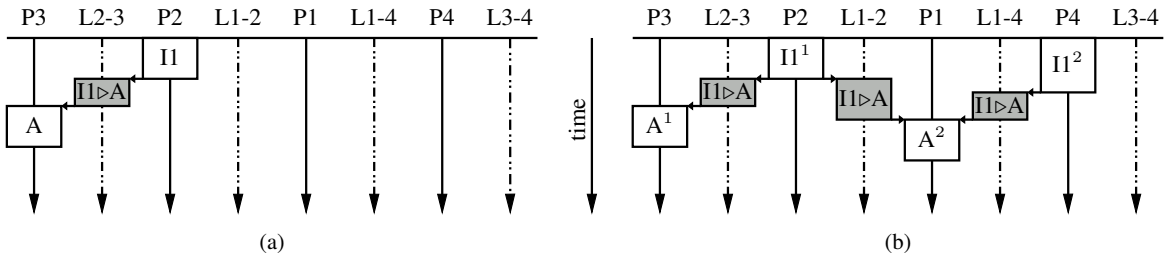


FIG. 3.28 – Ordonnancement partiel de $\mathcal{A}lg$ sur $\mathcal{A}rc$: (a) sans réplication et (b) avec.

La figure 3.29(a) montre le BDF construit à partir de l'ordonnancement partiel de la figure 3.28(a). Il comporte plusieurs blocs : (I1,P2), (I1▷A,L2-3), (A,P3)... reliés par des liens de causalité qui indiquent, par exemple, que le bloc (A,P3) est opérationnel si le bloc (I1▷A,L2-3) l'est aussi... Ce BDF est série. Les étiquettes « if », « and », « not fail » et « then sucess » ne sont utilisées que pour une meilleure compréhension.

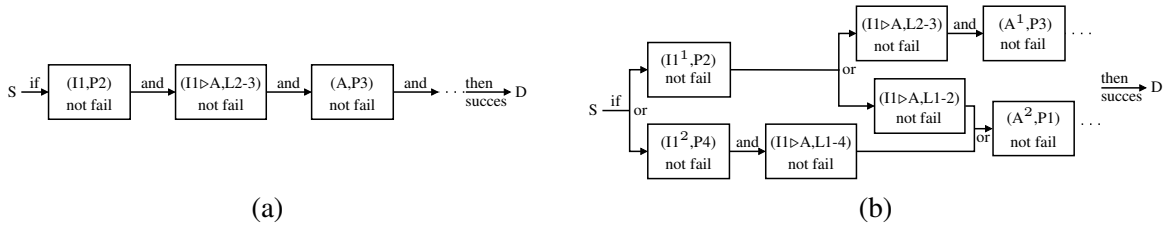


FIG. 3.29 – (a) BDF série de l'ordonnancement partiel de la figure 3.28(a) ; (b) BDF quelconque de l'ordonnancement partiel de la figure 3.28(b).

Plus généralement, pour tout ordonnancement sans réplication, toutes les opérations de l'ordonnancement doivent se terminer avec succès pour que l'ordonnancement soit opérationnel, donc tous les blocs du BDF doivent être opérationnels. Par conséquent, le BDF est toujours série. En fait, le BDF n'est autre qu'un ordre total compatible avec l'ordre partiel défini par l'ordonnancement.

En revanche, on constate sur la figure 3.29(b) que le BDF construit à partir de l'ordonnancement partiel de la figure 3.28(b) est quelconque (ni série, ni parallèle, ni série-parallèle, ni parallèle-série). Deux arcs partent du sommet S, ce qui est représenté dans la figure 3.29(b) en mettant en commun

leur point de départ. De même, deux arcs aboutissent au sommet (A^2,P1), ce qui est représenté dans la figure 3.29(b) en mettant en commun leur point d'arrivée.

Calcul exact de la fiabilité. Quand le BDF construit à partir de l'ordonnancement ord est **série**, sa fiabilité est :

$$F(ord) = \prod_{i=1..n} F(o_i, c_i) \tag{3.11}$$

où n est le nombre de blocs du BDF et $F(o_i, c_i)$ est la fiabilité du bloc i (opération o_i placée sur le composant c_i). En effet, ord est opérationnel si l'unique chemin de son BDF est opérationnel, et pour cela il faut que tous les blocs du chemin le soient, (d'où le produit des probabilités).

Par exemple, considérons les graphes $\mathcal{A}lg$ et $\mathcal{A}rc$ de la figure 3.6 et les caractéristiques d'exécution de la figure 3.7. L'ordonnancement correspondant produit par AAA est celui de la figure 3.8, page 72. Maintenant, supposons que les taux de défaillance par unité de temps soient donnés par la figure 3.30 (l'unité de temps étant l'heure) :

Composant matériel	Taux de défaillance
P1	$\lambda_1 = 10^{-5}$
P2	$\lambda_2 = 10^{-6}$
P3	$\lambda_3 = 10^{-7}$
L12	$\mu_{12} = 10^{-5}$
L13	$\mu_{13} = 10^{-6}$
L23	$\mu_{23} = 10^{-7}$

FIG. 3.30 – Taux de défaillance des processeurs et des liens de communication.

Alors la fiabilité de l'ordonnancement de la figure 3.8 est égale à 0.999918 (modèle « à chaud » et BDF série).

Quand le BDF construit à partir de l'ordonnancement ord est **parallèle**, sa fiabilité est :

$$F(ord) = 1 - \left(\prod_{i=1..n} \Big(1 - F(o_i, c_i)\Big) \right) \tag{3.12}$$

En effet, ord est non opérationnel (d'où le $1-$) si aucun des chemins de son BDF n'est opérationnel (d'où le produit des probabilités complémentaires).

Quand le BDF est **série-parallèle** ou **parallèle-série**, il suffit de combiner les deux formules (3.11) et (3.12) pour calculer la fiabilité de ord.

Mais dans le cas général, le BDF est **quelconque**, et le calcul exact de la fiabilité est exponentiel dans la taille de ord. Pour ce calcul exact, on peut par exemple construire un arbre binaire de la façon suivante :

- Chaque sommet est un bloc du BDF avec en fils gauche le sous-arbre correspondant au cas où le bloc est opérationnel, et en fils droit le sous-arbre correspondant au cas où le bloc n'est pas opérationnel.
- Chaque feuille f correspond à la combinaison des tous les blocs du BDF situés entre f et la racine de l'arbre ; sa valeur $val(f)$ est 1 si ord est opérationnel dans cette combinaison, et 0 sinon.

La profondeur n de cet arbre est égale au nombre de blocs du BDF, c'est-à-dire au nombre de composants de ord. Son nombre de feuilles est égal à 2^n. Grâce à cet arbre, la fiabilité de ord est alors :

$$F(ord) = \sum_{f|val(f)=1} \left(\prod_{i=1..n} F(o_i^f, c_i^f) \right) \tag{3.13}$$

où (o_i^f, c_i^f) est le i-ième sommet entre la racine de l'arbre et la feuille f.

Deux cas permettent d'arrêter la construction d'une branche de l'arbre avant la profondeur n : si les blocs opérationnels correspondants aux sommets situés entre la racine de l'arbre et le sommet courant font que l'ordonnancement est obligatoirement opérationnel, alors le sommet est remplacé par une feuille de valeur 1 ; de façon similaire, si les blocs non opérationnels correspondants aux sommets situés entre la racine de l'arbre et le sommet courant font que l'ordonnancement est obligatoirement non opérationnel, alors le sommet est remplacé par une feuille de valeur 0. Nicolas Leignel a proposé des stratégies permettant d'obtenir le plus vite possible de tels sommets, en choisissant judicieusement l'ordre dans lequel les blocs du BDF sont évalués [78].

Calcul approché de la fiabilité. Mais le problème est que ce calcul doit être effectué par l'algorithme d'ordonnancement & répartition *à chaque* décision d'ordonnancement, et ce plusieurs fois afin d'évaluer tous les choix possibles d'ordonnancement. Donc, en pratique, le calcul exact est impossible parce que trop coûteux (j'ai dit dans le paragraphe précédent qu'il est exponentiel dans la taille de ord).

Une méthode très répandue pour calculer une valeur approchée de $F(ord)$ est la **méthode des coupes minimales** [55, 21]. C'est, par exemple, la méthode vantée par des logiciels commerciaux tels que Reliability Workbench de la société ISOGRAPH [8] ou encore Item Toolkit de la société ITEM [9]. Une **coupe** dans un BDF est un ensemble $\mathcal{C}$ de blocs tels qu'il n'existe aucun chemin de S à D si on supprime du BDF tous les blocs de $\mathcal{C}$. Une coupe $\mathcal{C}$ est **minimale** si, quel que soit le bloc qu'on lui retire, l'ensemble ainsi obtenu n'est plus une coupe. La fiabilité de ord est approximée par la fiabilité du BDF constitué de toutes les coupes minimales mises en série. Or, la fiabilité de la coupe minimale $\mathcal{C}_i$ est celle de tous ses blocs mis en parallèle :

$$F(\mathcal{C}_i) = 1 - \prod_{(o,c)\in\mathcal{C}_i} \Big(1 - F(o,c)\Big) \tag{3.14}$$

Donc, pour approximer la fiabilité d'un BDF, il suffit donc d'en calculer toutes les coupes minimales $\mathcal{C}_i$, numérotées de 1 à n, puis de calculer la fiabilité du BDF série-parallèle constitué de toutes les coupes. Ce calcul est linéaire dans le nombre de coupes minimales :

$$F(ord) \geq F_{lower}(ord) = \prod_{i=1}^{n} \left(1 - \prod_{(o,c)\in\mathcal{C}_i} \Big(1 - F(o,c)\Big)\right) \tag{3.15}$$

Cette approximation est une **borne inférieure**. Par conséquent, si un ordonnancement ord est tel que $F_{lower}(ord) > \mathcal{F}_{obj}$, où $\mathcal{F}_{obj}$ est l'objectif de fiabilité du problème 3.6 que l'on cherche à résoudre, alors cela prouve que $F(ord)$ est supérieure à $\mathcal{F}_{obj}$, ce qui est très bien. En fait, le principal problème de cette méthode est que, selon la structure du BDF, le nombre total de ses coupes minimales est compris entre $n-1$ et 2^{n-2}, où n est le nombre de sommets du BDF. Donc en définitive, le calcul de la fiabilité approchée par la méthode des coupes minimales est lui-aussi exponentiel dans la taille de ord. C'est pour cette raison que Hamoudi Kalla a proposé dans sa thèse une approche originale consistant à produire un ordonnancement ord de telle manière que son BDF soit série-parallèle par construction [57] ; cela permet d'en calculer la fiabilité dans un temps polynomial dans la taille de ord. L'inconvénient est que l'ordonnancement ainsi produit est a priori plus long.

Influence de la longueur sur la fiabilité. La formule (3.7) de calcul de la fiabilité d'une opération placée sur un processeur indique aussi que la longueur de l'ordonnancement a aussi une influence déterminante sur sa fiabilité. En effet, en l'absence de réplication, on a tout autant intérêt à placer une

[8] ISOGRAPH : `http://www.isograph-software.com/rwboverrbd.htm`.
[9] ITEM : `http://www.itemuk.mcmail.com/rbd.html`

opération X sur un processeur P fiable (tel que λ_P soit maximal) que sur un processeur rapide (tel que $\mathcal{E}xe\langle \mathrm{X}, \mathrm{P}\rangle$ soit minimal). Quand il y a des réplications, l'influence de la longueur est contre-balancée par le fait qu'il existe plusieurs chemins de S à D dans le BDF. C'est pourquoi il peut en fait être plus avantageux pour la fiabilité de dupliquer X sur deux processeurs plutôt que de la placer sur un seul processeur, même plus fiable et plus rapide.

3.4.5.7 Algorithme d'ordonnancement & répartition AAA-F

L'algorithme AAA-F est une variante de l'algorithme AAA. Par rapport à l'algorithme 3.1, deux choses changent : d'une part la fonction de coût permettant de choisir sur quel sous-ensemble de processeurs de $\mathcal{A}rc$ est ordonnancée chaque opération de $\mathcal{A}lg$, et d'autre part le fait qu'une opération est maintenant susceptible d'être implantée sur *plusieurs* processeurs et non pas un seul (tout comme dans les méthodes AAA-TP et AAA-TB).

La méthode que j'ai choisie pour résoudre ce problème bicritère est l'agrégation des deux critères en un seul (voir la section 3.4.5.5). Mais comme les deux critères n'ont pas du tout le même ordre de grandeur, j'ai décidé de d'abord **normaliser** chaque critère par rapport à l'objectif correspondant ($\mathcal{L}_{obj}$ pour la longueur et $\mathcal{F}_{obj}$ pour la fiabilité).

Je note $Ord^{(n-1)}$ l'ordonnancement partiel à l'étape $n-1$, sa longueur $L(Ord^{(n-1)})$, et sa fiabilité $F(Ord^{(n-1)})$. À l'étape n, la décision d'ordonnancer l'opération candidate o_i sur l'ensemble de processeurs $\mathcal{P}_j$ donne l'ordonnancement $Ord^{(n)}(o_i, \mathcal{P}_j)$. Le gain en longueur et la perte en fiabilité qui en résultent sont donc :

$$\begin{aligned}
\text{gain en longueur} &= L(Ord^{(n)}(o_i, \mathcal{P}_j)) - L(Ord^{(n-1)}) \\
\text{marge en longueur} &= \mathcal{L}_{obj} - L(Ord^{(n-1)}) \\
\text{perte en fiabilité} &= F(Ord^{(n-1)}) - F(Ord^{(n)}(o_i, \mathcal{P}_j)) \\
\text{marge en fiabilité} &= F(Ord^{(n-1)}) - \mathcal{F}_{obj}
\end{aligned}$$

Donc, en normalisant par rapport aux deux objectifs, j'obtiens :

$$\text{gain normalisé en longueur} = \mathcal{G}nl^{(n)}(o_i, \mathcal{P}_j) = \frac{L(Ord^{(n)}(o_i, \mathcal{P}_j)) - L(Ord^{(n-1)})}{\mathcal{L}_{obj} - L(Ord^{(n-1)})} \tag{3.16}$$

$$\text{perte normalisée en fiabilité} = \mathcal{P}nf^{(n)}(o_i, \mathcal{P}_j) = \frac{F(Ord^{(n-1)}) - F(Ord^{(n)}(o_i, \mathcal{P}_j))}{F(Ord^{(n-1)}) - \mathcal{F}_{obj}} \tag{3.17}$$

La fonction de coût $\mathcal{FC}_{AAA-F}$ de l'heuristique AAA-F combine ces deux valeurs de la façon illustrée par la figure 3.31(a). Soit une opération candidate o_i : pour chaque sous-ensemble $\mathcal{P}_j$ de $\mathcal{P}$ (l'ensemble des processeurs de $\mathcal{A}rc$), elle projette le point $(\mathcal{G}nl^{(n)}(o_i, \mathcal{P}_j), \mathcal{P}nf^{(n)}(o_i, \mathcal{P}_j))$ sur la droite d'équation $\mathcal{P}nf = \tan(\theta)\mathcal{G}nl$, où θ est un paramètre de l'heuristique, et elle sélectionne le sous-ensemble $\mathcal{P}_{meilleur}$ tel que la projection du point correspondant sur cette droite soit le plus proche de l'origine. Autrement dit, la fonction de coût $\mathcal{FC}_{AAA-F}$ est donnée par la formule (3.18) :

$$\mathcal{FC}_{AAA-F}(o_i, \mathcal{P}_j) = \cos(\theta).\mathcal{G}nl^{(n)}(o_i, \mathcal{P}_j) + \sin(\theta).\mathcal{P}nf^{(n)}(o_i, \mathcal{P}_j) \tag{3.18}$$

Et $\mathcal{P}_{meilleur}$ est le sous-ensemble de processeurs qui réalise le $\min_{\mathcal{P}_j \in 2^{\mathcal{P}}} \mathcal{FC}_{AAA-F}(o_i, \mathcal{P}_j)$. Pour l'exemple de la figure 3.31(a), $\mathcal{P}_{meilleur} = \{p_1, p_3\}$, ce qui signifie que o_i doit être dupliquée sur les deux processeurs p_1 et p_3. Quand $\mathcal{P}_{meilleur}$ est un singleton, cela signifie que l'opération o_i ne doit pas être dupliquée. Dans l'algorithme 3.1, cette nouvelle fonction de coût $\mathcal{FC}_{AAA-F}$ remplace la fonction $\mathcal{FC}_{AAA}$ appelée à la ligne 8.

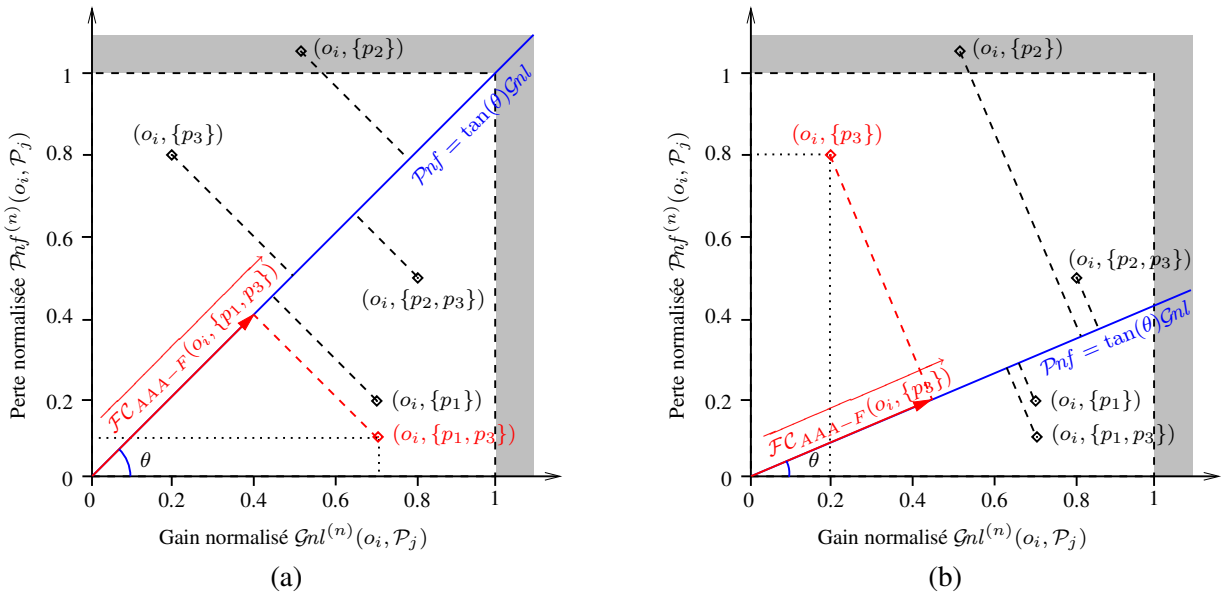


FIG. 3.31 – Calcul de la fonction de coût $\mathcal{FC}_{AAA-F}$ avec deux valeurs différentes de θ : (a) $\theta = \pi/4$; (b) $\theta = \pi/8$.

La seconde modification par rapport à l'algorithme 3.1 concerne l'implantation de l'opération choisie o^{pire} sur plusieurs processeurs au lieu d'un seul. Sans entrer dans les détails, il faut donc modifier les lignes 27 à 28 pour implanter o^{pire} sur l'ensemble de processeurs $\mathcal{P}^{pire}$, ainsi que les lignes 19 à 23 afin que les dépendances de données provenant des opérations de $pred(o^{pire})$ soient maintenant envoyées à tous les processeurs de l'ensemble $\mathcal{P}^{pire}$ et non plus à un seul processeur.

Seuls les points situés dans la portion du plan $[0, 1] \times [0, 1]$ doivent être considérés. Les autres entraînent un échec sur un des deux objectifs (voire les deux) : c'est le cas du point $(o_i, \{p_2\})$ de la figure 3.31(a). Si aucun des points n'est situé dans la portion du plan $[0, 1] \times [0, 1]$, alors l'algorithme termine. Il faut alors recommencer au début en diminuant le paramètre θ en cas d'échec à satisfaire l'objectif $\mathcal{L}_{obj}$, et en l'augmentant en cas d'échec à satisfaire l'objectif $\mathcal{F}_{obj}$. Cette modification a pour effet de donner plus de poids dans la fonction $\mathcal{FC}_{AAA-F}$ à l'objectif sur lequel l'heuristique vient d'échouer. Par exemple, la figure 3.31(b) illustre le cas où θ a été diminué car l'heuristique venait d'échouer à cause de l'objectif $\mathcal{L}_{obj}$. Cette fois-ci, c'est le point $(o_i, \{p_3\})$ qui est choisi.

Deux cas extrêmes existent : quand θ vaut 0 (resp. $\pi/2$), $\mathcal{FC}_{AAA-F}$ devient une fonction de coût mono-critère, visant à minimiser uniquement la longueur de l'ordonnancement (resp. la fiabilité).

3.4.6 État de l'art sur l'ordonnancement et l'optimisation bicritère

En plus des problèmes généraux d'optimisation, de nombreux travaux portent sur le problème spécifique de l'ordonnancement & répartition bicritère longueur/fiabilité sur architectures réparties hétérogènes par un algorithme statique. À mon avis, les plus aboutis sont [28, 29] et [93] car, contrairement à la plupart des autres résultats, ils ne supposent pas que le réseau de communication soit exempt de cycle. On trouve également les travaux de Srinivasan et Jha [111], mais ces derniers visent uniquement à augmenter la fiabilité et pas à diminuer la longueur de l'ordonnancement.

L'algorithme proposé par Dogan et Özgüner dans [28] est une extension de l'algorithme Dynamic Level Scheduling (DLS) de Sih et Lee [109]. Ce dernier est une heuristique d'ordonnancement de liste d'un DAG d'opérations $\mathcal{A}lg$ sur un ensemble de processeurs hétérogènes $\mathcal{A}rc$. À chaque étape, l'heuristique évalue tous les couples $\langle$opération, processeur$\rangle$ et choisit de placer sur le processeur p l'opération o telle que la fonction de coût $DL(o, p)$ soit maximale. Cette fonction de coût DL est calculée de la façon

suivante (« Dynamic Level », SL = « Static Level ») :

$$DL(o,p) = SL(o) - \max\{t^A_{o,p}, t^M_p\} + \Delta(o,p) \tag{3.19}$$

$$t^A_{o,p} = \text{ date à laquelle } p \text{ a reçu toutes les données d'entrée de } o \tag{3.20}$$

$$t^M_p = \text{ date à laquelle } p \text{ est libre pour exécuter une nouvelle opération} \tag{3.21}$$

$$t^E_{o,p} = \text{ durée d'exécution de } o \text{ sur } p \tag{3.22}$$

$$\begin{aligned} \bar{t}^E_o &= \text{ durée moyenne d'exécution de } o \text{ sur tous les processeurs de } \mathcal{A}rc \\ &= \frac{\sum_{p \in \mathcal{A}rc} t^E_{o,p}}{|\mathcal{A}rc|} \end{aligned} \tag{3.23}$$

$$\begin{aligned} SL(o) &= \text{ la plus grande somme des temps d'exécution moyens des opérations} \\ &\quad \text{le long de tous les chemins existant entre } o \text{ et la fin du DAG } \mathcal{A}lg \\ &= \max_{\substack{\text{chemins } \pi \text{ de } o \\ \text{à la fin de } \mathcal{A}lg}} \sum_{o_i \in \pi} \bar{t}^E_{o_i} \end{aligned} \tag{3.24}$$

$$\Delta(o,p) = \bar{t}^E_o - t^E_{o,p} \tag{3.25}$$

Dans l'algorithme DLS, le coût des communications est donc pris en compte grâce au terme $t^A_{o,p}$, équation (3.20). L'apport de Dogan et Özgüner est d'avoir ajouté un quatrième terme à la fonction de coût DL afin de tenir compte de la fiabilité de l'ordonnancement ; leur nouvelle fonction de coût s'appelle RDL (« Reliable Dynamic Level »). Ils considèrent que l'occurrence des défaillances des ressources matérielles (processeurs et liens de communication) suit une loi de Poisson de la forme $\exp^{-\lambda t}$, où λ est le taux de défaillance de la ressource et t est son temps d'utilisation. Ils supposent de plus, que l'architecture est hétérogène pour les fiabilités, que les défaillances des diverses ressources matérielles sont permanentes et que ce sont des événements statistiquement indépendants.

Dans [29], Dogan et Özgüner proposent une extension bicritère de l'algorithme DLS en construisant à chaque étape deux listes, contenant toutes les paires $\langle v_i, m_j \rangle$ de tâches v_i prêtes à être ordonnancées et de machines m_j : la première liste est triée par ordre décroissant de la fonction DL (equation (3.19), qui rend compte de l'accroissement de la longueur de l'ordonnancement dû à la décision de placer v_i sur m_j ; la seconde liste est triée par ordre croissant de la fonction $\Delta COST$, qui rend compte de la baisse de fiabilité dûe à cette même décision. Ainsi, chaque paire $\langle v_i, m_j \rangle$ a un rang dans chacune de des deux listes triées, respectivement notés $Rank^1_{i,j}$ et $Rank^2_{i,j}$. Ces deux rangs sont alors combinés en $Rank_{i,j} = \delta^1 Rank^1_{i,j} + \delta^2 Rank^2_{i,j}$, où les deux nombres δ^1 et δ^2 sont choisis empiriquement de manière à équilibrer les deux critères. Enfin, la paire $\langle v_i, m_j \rangle$ ayant le plus petit $Rank_{i,j}$ est choisie et v_i est alors placée sur m_j.

Dans ces deux travaux ([28] and [29]), la fiabilité est améliorée uniquement en choisissant d'ordonnancer certaines opérations de $\mathcal{A}lg$ sur un processeur plutôt qu'un autre, en étant pour cela guidé par le calcul de la fiabilité. On voit donc que la fiabilité, qui est a priori une mesure de la tolérance aux fautes, peut être utilisée comme un moyen. Les résultats des simulations effectuées par Dogan et Özgüner montrent que leur algorithme RDSL est systématiquement meilleur que l'algorithme DSL de Sih et Lee sur le critère fiabilité, mais systématiquement moins bon sur le critère longueur. La principale différence entre le travail de Dogan et Özgüner et AAA-F est qu'ils ne dupliquent aucune opération du graphe algorithme $\mathcal{A}lg$, alors que l'algorithme que je présente dans la suite duplique les opérations afin d'améliorer encore plus la fiabilité. L'algorithme d'AAA-F essaye de plus d'améliorer la localité des calculs grâce à la réplication [3], afin de diminuer le nombre de communications, dans le but d'optimiser également le critère longueur.

J'ai déjà présenté dans la section 3.4.2.7 l'algorithme eFCRD proposé par Qin et al. dans [93] : le but premier est la tolérance exactement d'une faute de processeur, donc le taux de réplication de chaque tâche est égal à 2. Concernant la fiabilité, eFCRD tient compte de l'hétérogénéité en fiabilité des processeurs. Le modèle de défaillance est le même que celui de Dogan et Özgüner. Qin et al. définissent le coût de fiabilité de la tâche v_i ordonnancée sur le processeur p_j comme étant le produit de λ_j par le temps d'exécution de v_i sur p_j. Puisque la fiabilité de l'ordonnancement augmente quand le coût de fiabilité diminue, l'algorithme eFCRD ordonnance chaque réplique primaire sur un processeur tel que l'accroissement du coût de fiabilité soit minimum (et tel que l'échéance de la tâche soit satisfaite bien sûr). Mais la fiabilité des liens de communication n'est pas prise en compte.

Dans [107, 108, 62, 63, 52, 51], le réseau de communication est supposé être exempt de cycle, ce qui me semble être une très grande restriction. En outre, dans [107, 108, 62, 63, 51], le système à ordonnancer & répartir consiste en un ensemble de modules logiciels qui communiquent entre eux, mais sans que ces communications n'influent sur l'ordre dans lequel les modules peuvent être exécutés par les processeurs : les communications entre deux modules sont supposées être bi-directionnelles et sont abstraites sous la forme d'un **temps de communication inter-module** (« inter-module communication cost » [23]). Ceci ne correspond pas aux modèles de programmation avec dépendances de données. De plus, tous ces algorithmes explorent l'espace d'état entièrement dans le but de trouver l'allocation optimale (pour le critère de fiabilité) des modules logiciels sur les processeurs. Même avec les techniques de réduction de l'espace d'état proposées dans [62, 63], cette exploration exhaustive reste très coûteuse, ce qui fait qu'ils ne sont capables de traiter que des problèmes de petite taille (pas plus de 8 modules). Enfin, [107, 62, 51] traitent du cas particulier des systèmes répartis redondants, dans lesquels chaque nœud de calcul consiste en n processeurs identiques, et chaque connexion entre deux nœuds de calcul consiste en m liens de communication identiques ; spécifiquement, ils considèrent des systèmes avec redondance de niveau 2 et 3, mais sans redondance logicielle.

3.4.6.1 Résultats d'exécution de l'heuristique d'AAA-F

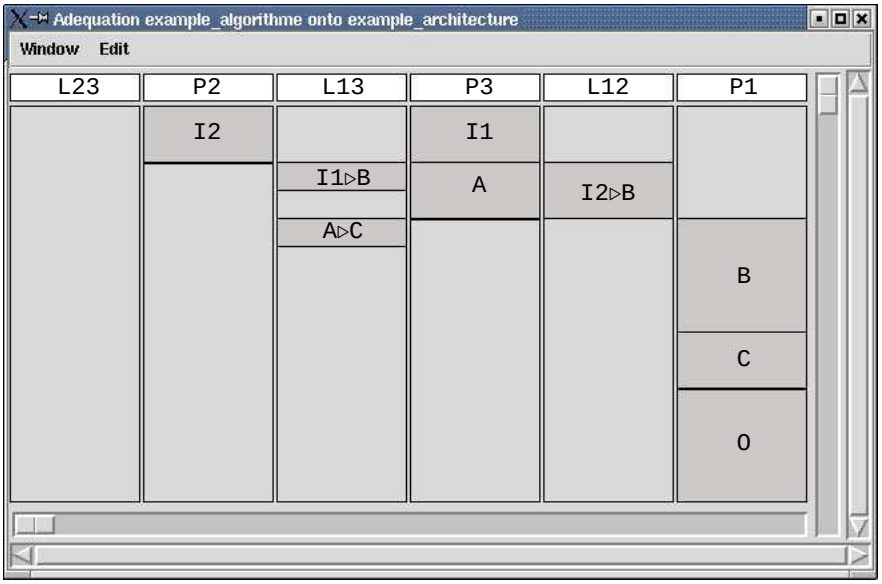
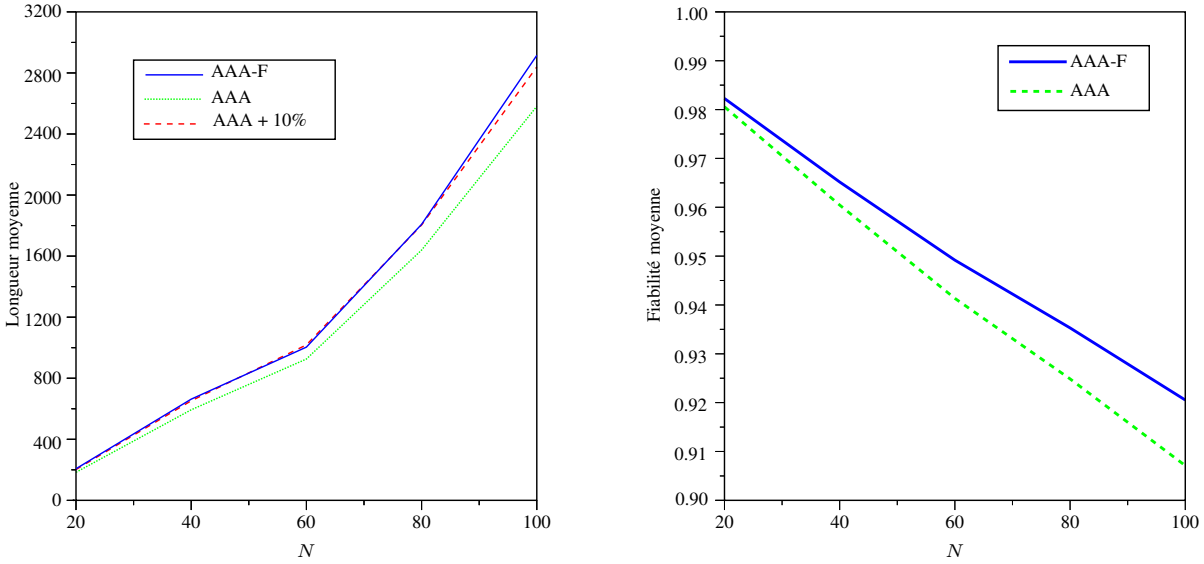


FIG. 3.32 – Ordonnancement produit par AAA-F ($\theta = \pi/4$).

La figure 3.32 présente tout d'abord un exemple d'ordonnancement produit par AAA-F, avec les graphes de la figure 3.6, les caractéristiques d'exécution de la figure 3.7 et les taux de défaillance de la figure 3.30. Les deux objectifs donnés à AAA-F sont $\mathcal{L}_{obj}$=7 unités de temps et $\mathcal{F}_{obj}$=0.999918 (ce

sont les résultats obtenus par AAA : voir la figure 3.8 page 72). La longueur de ordonnancement obtenu par AAA-F est égale à 7 unités de temps, et la fiabilité est égale à 0.999948. Par conséquent, sur cet exemple, AAA-F trouve un ordonnancement de la même longueur mais d'une fiabilité un tout petit peu meilleure. On remarque qu'AAA-F n'a répliqué aucune opération, et donc cette amélioration de la fiabilité a été obtenue grâce à un meilleur placement des opérations sur les processeurs (meilleur vis à vis de la fiabilité).

Les figures 3.33(a) et (b) comparent respectivement la longueur et la fiabilité moyenne des ordonnancement générés par les heuristiques d'AAA-F et d'AAA. Chaque point est la moyenne de 50 graphes $\mathcal{A}lg$ générés aléatoirement avec CCR$=1$ et N variant de 20 à 100 opérations, et ordonnancés sur un graphe $\mathcal{A}rc$ complet à 5 processeurs avec liens point-à-point. Les taux de défaillance des processeurs (resp. des liens de communication) ont été tirés aléatoirement entre 5×10^{-5} et 10^{-4} (resp. entre 15×10^{-5} et 3×10^{-4}). À chaque invocation de l'heuristique d'AAA-F, les deux objectifs $\mathcal{L}_{obj}$ et $\mathcal{F}_{obj}$ qui lui sont fournis sont respectivement la longueur mesurée sur l'ordonnancement produit avec ce graphe $\mathcal{A}lg$ par l'heuristique d'AAA $+10\%$ (afin d'éviter trop d'échecs) et la fiabilité mesurée sur ce même ordonnancement ; de plus, θ vaut initialement $\pi/4$. Les ordonnancements obtenus avec AAA-F sont systématiquement meilleurs en fiabilité mais moins bon de 10% en longueur environ.



(a) (b)

FIG. 3.33 – Comparaison entre AAA-F et AAA : (a) longueur moyenne des ordonnancements ; (b) fiabilité moyenne des ordonnancements.

3.4.7 Lien entre fiabilité et tolérance aux fautes

Dans [111], Srinivasan et Jha proposent une méthode d'ordonnancement et répartition en deux phases, qui vise à maximiser à la fois la sécurité-innocuité et la fiabilité. Ils définissent la fiabilité de la façon usuelle [81] (comme dans la section 3.4.5.6), et la sécurité-innocuité comme la propriété que, ou bien le résultat calculé par le système soit correct, ou bien le système soit capable de détecter la présence des fautes quand elles se produisent. Le modèle du système consiste en un graphe $\mathcal{A}lg$ de n tâches (un DAG dont chaque nœud est une tâche et chaque arc une dépendance de données), un graphe $\mathcal{A}rc$ de p processeurs, et une matrice $\mathcal{E}xe \in (\mathbb{R} \cup \{+\infty\})^{n \times p}$ de coûts d'exécution. Les processeurs sont ou bien opérationnels, ou bien défaillants, et dans ce cas ils sont remplacés. La méthode de Srinivasan et Jha comporte deux phases. La première phase alloue les tâches de $\mathcal{A}lg$ sur les processeurs de $\mathcal{A}rc$ avec la fiabilité comme objectif à maximiser. C'est une heuristique qui regroupe ensemble les tâches qui

communiquent beaucoup, de telle sorte que toutes les tâches d'un même groupe soient placées sur un même processeur. De plus, les tâches (resp. les dépendances de données) les plus coûteuses sont allouées aux processeurs (resp. aux liens de communication) les plus fiables. La seconde phase alloue des tâches supplémentaires, dites « tolérantes aux fautes », tout en visant à minimiser la perte de fiabilité. Une tâche tolérante aux fautes est ou bien une tâche de vérification de conformité des résultats d'une autre tâche (par exemple un calcul de sommes de contrôle dans une multiplication de matrices...), ou bien une réplique d'une autre tâche couplée avec un comparateur de leurs résultats. Dans les deux cas, elle permet de déterminer si une tâche est fautive. Selon les types de graphes $\mathcal{A}lg$ générés aléatoirement, la perte en fiabilité varie entre 44% et 56%. Cette diminution s'explique par le fait que, quand on ajoute une tâche supplémentaire tolérante aux fautes τ_f à une tâche τ, les deux tâches τ et τ_f sont en *série* du point de vue de la fiabilité, et non pas en *parallèle* ; aussi la défaillance de *l'une des deux* entraîne la défaillance de l'ensemble, ce qui explique la perte en fiabilité. En revanche, la sécurité-innocuité augmente.

Pour aller plus loin, Hamoudi Kalla a effectué une série de simulations afin de comparer AAA-F et AAA-TP, pour $\mathcal{N}pf \in \{1, 2, 3\}$ et $\mathcal{N}lf = 0$. Les paramètres étaient les suivants :

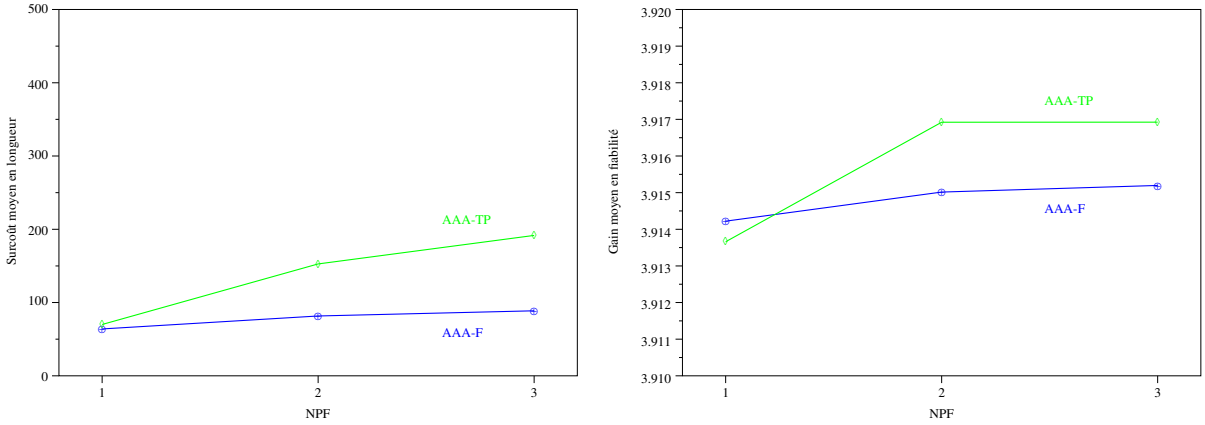
- la même série de 50 graphes $\mathcal{A}lg$ de 40 tâches avec CCR $= 1$, générés aléatoirement ;
- le même graphe $\mathcal{A}rc$ complet avec 6 processeurs : P_1, P_2, P_3, P_4, P_5 et P_6 ;
- les taux de défaillance suivants : $\lambda_{P_1} = \lambda_{P_2} = \lambda_{P_3} = 10^{-5}$ et $\lambda_{P_4} = \lambda_{P_5} = \lambda_{P_6} = 10^{-7}$, c'est-à-dire que l'architecture est fortement hétérogène du point de vue de la fiabilité.

Le but de ces simulations était d'estimer l'influence des deux valeurs $\mathcal{L}_{obj}$ et $\mathcal{F}_{obj}$ données comme objectifs à l'heuristique d'ordonnancement & répartition AAA-F. À chaque fois, ces objectifs étaient obtenus en mesurant la longueur (resp. la fiabilité) de l'ordonnancement produit par AAA-TP et en ajoutant (resp. en retranchant) une certaine marge, par exemple $+20\%$ en longueur (resp. -10^{-2} en fiabilité). Puis, les ordonnancements obtenus étaient comparés au moyen des deux formules suivantes :

$$\text{surcoût en longueur} = \frac{\text{longueur(AAA-F ou AAA-TP)} - \text{longueur(AAA)}}{\text{longueur(AAA)}} \times 100 \tag{3.26}$$

$$\text{gain en fiabilité} = \frac{\text{fiabilité(AAA-F ou AAA-TP)} - \text{fiabilité(AAA)}}{\text{fiabilité(AAA)}} \times 100 \tag{3.27}$$

Hamoudi Kalla a fait varier la marge en longueur de $+0\%$ à $+80\%$, et la marge en fiabilité de -0 à -10^{-4}. Les meilleurs résultats furent obtenus en prenant comme objectifs longueur(AAA-TP) $+80\%$ d'une part, et fiabilité(AAA-TP) -10^{-4} (c'est donc empirique) ; ils sont montrés dans la figure 3.34.



(a) (b)

FIG. 3.34 – Comparaison entre AAA-F et AAA-TP avec $\lambda_{P_1} = \lambda_{P_2} = \lambda_{P_3} = 10^{-5}$, $\lambda_{P_4} = \lambda_{P_5} = \lambda_{P_6} = 10^{-7}$, N $= 40$ tâches, CCR $= 1$, $\mathcal{N}lf = 0$ et $|\mathcal{P}| = 6$: (a) surcoût en longueur moyen des ordonnancements ; (b) gain en fiabilité moyen des ordonnancements.

D'une part, on constate sur la figure 3.34(a) que AAA-F donne des ordonnancements systématiquement meilleurs en longueur (le surcoût est moindre), et même très significativement quand $\mathcal{N}pf \in \{2, 3\}$. D'autre part, on constate sur la figure 3.34(b) que AAA-F donne des ordonnancements un tout petit peu meilleurs en fiabilité quand $\mathcal{N}pf = 1$ (le gain est supérieur) mais moins bons quand $\mathcal{N}pf \in \{2, 3\}$ (le gain est moindre), mais dans tous les cas très marginalement (puisque le gain reste inférieur à 4%). Une donnée supplémentaire qui permet d'appréhender les ordonnancements produits par AAA-F est leur taux de réplication moyen. Ceux qui correspondent à la figure 3.34 sont les suivants :

	Taux de réplication moyen	
$\mathcal{N}pf$	AAA-F	AAA-TP
1	1.16	2
2	1.25	3
3	1.29	4

Ainsi, l'heuristique d'AAA-F réussit à ne répliquer les opérations que très peu par rapport à AAA-TP, ce qui explique que la longueur des ordonnancements qu'elle produit soit nettement moindre.

3.5 Méthode par fusion d'ordonnancements simples

En 1999, Cătălin Dima a proposé une méthode très originale pour produire des ordonnancements tolérants aux fautes des processeurs et des liens de communication. Mais ici, au lieu d'exiger qu'un nombre fixe de fautes de processeurs et de media de communication soit toléré (comme dans les sections 3.4.2 et 3.4.3), l'utilisateur spécifie un ensemble de patrons de fautes à tolérer, chacun étant un sous-ensemble des composants matériels de l'architecture pouvant être défaillants simultanément.

Pour spécifier de tels patrons de fautes, une possibilité serait de définir, au niveau matériel, des domaines de confinement de fautes, et d'en déduire les sous-ensembles de composants qui peuvent être fautifs simultanément. Il est également possible de se baser sur la connaissance des mécanismes de propagation d'erreurs d'un composant matériel à l'autre dans le système.

Cette méthode s'appelle HFOS : « Heuristique de Fusion d'Ordonnancements Simples ». Les travaux présentés dans cette section ont été publiés dans les articles [26, 27]. De plus, cette méthode a été implémentée avec succès par Thomas Lévêque à l'INRIA [79] ainsi que par Claudio Pinello à UC Berkeley [88].

3.5.1 Modèles utilisés

Je considère toujours des systèmes informatiques répartis et embarqués. Ils sont modélisés par, d'une part une spécification d'algorithme, et d'autre part une spécification d'architecture répartie cible. Le modèle du graphe d'architecture $\mathcal{A}rc$ est le même que dans la section 3.3.3 : c'est un **graphe biparti non-orienté**, reliant les processeurs et les media de communication (ici des bus, pas nécessairement connectés à tous les processeurs). En revanche, le modèle du graphe d'algorithme $\mathcal{A}lg$ est un peu plus compliqué que dans la section 3.3.3 : c'est un **graphe biparti flot-de-données**, reliant les opérations de calcul et les dépendances de données. La raison en est qu'il va être nécessaire de représenter explicitement le routage des dépendances de données par les processeurs, en particulier afin de détecter d'éventuels cycles dans les routages.

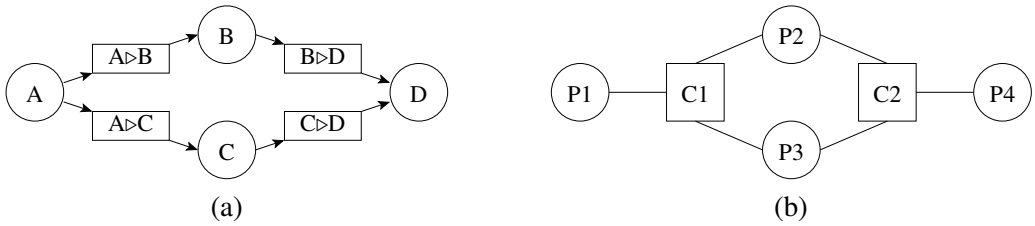


FIG. 3.35 – (a) Exemple de graphe $\mathcal{A}lg$ avec quatre opérations et quatre dépendances de données ; (b) Exemple de graphe $\mathcal{A}rc$ avec quatre processeurs et deux bus de communication.

La figure 3.35(a) montre un exemple de graphe $\mathcal{A}lg$ comportant quatre opérations — A, B, C et D — et quatre dépendances de données — A▷B, A▷C, B▷D et C▷D. La figure 3.35(b) montre un exemple de graphe $\mathcal{A}rc$ comportant quatre processeurs — P1, P2, P3 et P4 — et deux bus de communication — C1 et C2.

3.5.2 Énoncé du problème

Problème 3.7 (HFOS) *Soit un graphe d'algorithme $\mathcal{A}lg$, un graphe d'architecture $\mathcal{A}rc$ avec canaux de communication multipoint, un tableau $\mathcal{E}xe$ des WCET des opérations de $\mathcal{A}lg$ sur les processeurs de $\mathcal{A}rc$ et des WCTT des dépendances de données de $\mathcal{A}lg$ sur les canaux de communication de $\mathcal{A}rc$, des contraintes de distribution $\mathcal{D}is$, et un ensemble $\mathcal{P}df$ de n patrons de fautes.*

Trouver un ordonnancement multiprocesseur et statique de $\mathcal{A}lg$ sur $\mathcal{A}rc$, qui respecte $\mathcal{D}is$, qui soit tolérant aux fautes des patrons de $\mathcal{P}df$, et dont la longueur calculée grâce à $\mathcal{E}xe$ soit minimale.

De même que le problème 3.1, ce problème est NP-difficile, et donc la méthode HFOS que je propose pour le résoudre se base sur un algorithme heuristique d'ordonnancement & répartition. Une contrainte temps-réel $\mathcal{L}_{obj}$ doit de plus être respectée, donc il faut vérifier à la fin que la longueur de l'ordonnancement multiprocesseur obtenu est inférieure à $\mathcal{L}_{obj}$.

3.5.3 Hypothèse de faute

Les composants matériels de $\mathcal{A}rc$ (processeurs et bus de communication) sont supposées être à silence sur défaillance. De plus, les défaillances sont permanentes. Les fautes que l'utilisateur veut voir tolérer sont spécifiées sous la forme d'un ensemble $\mathcal{P}df$ de n patrons de fautes $\{\Pi_i, 1 \leq i \leq n\}$. Un **patron de fautes** est un sous-ensemble de l'ensemble des composants matériels de l'architecture pouvant être défaillants simultanément. La défaillance d'un bus de communication empêche toute communication via ce bus (défaillance totale). Je suppose de plus que, pour tout $1 \leq i \leq n$, le graphe de l'architecture réduite $\mathcal{A}rc - \Pi_i$ est connexe.

3.5.4 Schéma général de la méthode HFOS

La figure 3.36 montre le schéma général de la méthode HFOS.

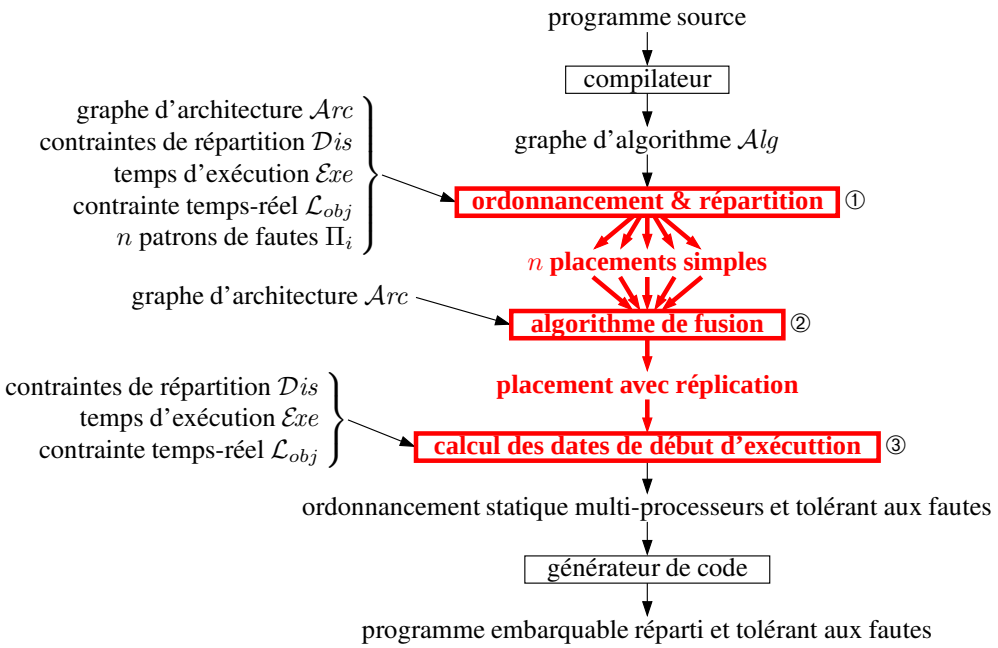


FIG. 3.36 – Schéma général de la méthode HFOS.

La méthode HFOS comporte trois étapes. La première consiste à produire, pour chaque patron de faute Π_i, l'architecture réduite $\mathcal{A}rc_i$ constituée de l'architecture cible $\mathcal{A}rc$ à laquelle les composants du patron de faute Π_i ont été retirés. Pour chacune des n architectures réduites $\mathcal{A}rc_i$, il faut alors générer un ordonnancement statique de $\mathcal{A}lg$ sur cette architecture réduite, ce qui fait donc n ordonnancements statiques (étape ①). J'appelle chacun de ces ordonnancements un **placement simple**, parce que ce sont des ordonnancements sans les dates de début d'exécution (d'où le terme « placement » au lieu d'« ordonnancement ») et sans réplication (d'où le terme « simple »). Chacun de ces placements simples peut être généré par n'importe quel algorithme d'ordonnancement & répartition, par exemple l'algorithme 3.1 de AAA. La deuxième étape consiste à **fusionner** ces n placements simples de façon à obtenir un unique placement de l'algorithme $\mathcal{A}lg$ sur l'architecture $\mathcal{A}rc$, qui soit tolérant aux fautes des n patrons Π_i. C'est un placement avec réplication (étape ②). La dernière étape consiste à calculer les dates de début d'exécution de chaque opération et dépendance de données du placement avec réplication calculé précédemment, afin d'obtenir un véritable ordonnancement multiprocesseur et tolérant aux fautes (étape ③).

3.5.5 Génération des placements simples

Comme exemple, je considère l'algorithme $\mathcal{A}lg$ et l'architecture $\mathcal{A}rc$ de la figure 3.35, ainsi que trois patrons de fautes : $\Pi_1 = \{$P1,C1$\}$, $\Pi_2 = \{$P2$\}$, et $\Pi_3 = \{$P4,C2$\}$. Pour chacun de ces trois patrons Π_i, je peux obtenir, grâce à l'heuristique de répartition & ordonnancement AAA (décrite à la section 3.1), un placement simple de $\mathcal{A}lg$ sur l'architecture réduite $\mathcal{A}rc \setminus \Pi_i$. Chacun de ces placements simples est un DAG. Les trois figures 3.37(a–c) montrent respectivement ces trois placements simples. Dans chaque figure, l'opération X placée sur le processeur P est représentée graphiquement par une ellipse avec X/P en son centre, la dépendance de données X▷Y transmise sur le bus C est représentée par un rectangle avec X▷Y/C, et enfin la même dépendance, quand elle est interne ou routée par le processeur P, est représentée par un rectangle à bords arrondis avec X▷Y/P. Par exemple, sur le placement de la figure 3.37(b), on voit que la dépendance de données A▷B est d'abord transmise sur le bus C2, puis routée par le processeur P3, avant d'être transmise sur le bus C1 et enfin reçue par le processeur P1 qui attend cette donnée justement pour exécuter l'opération B.

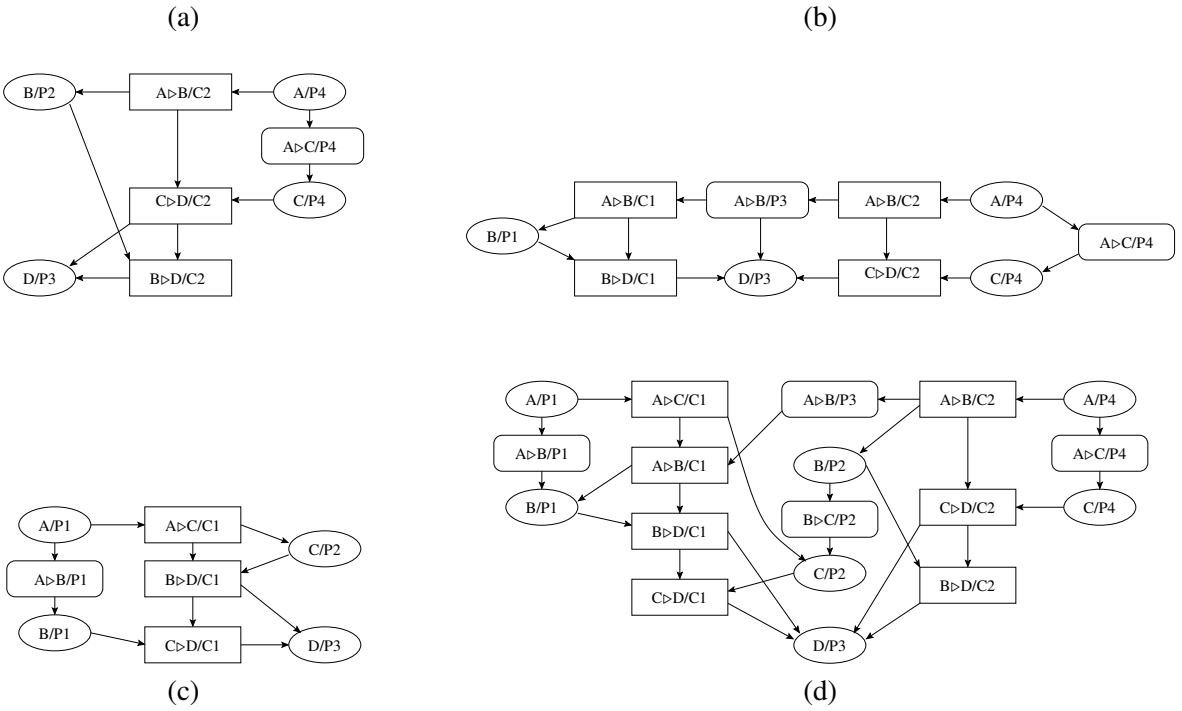


FIG. 3.37 – (a), (b) et (c) Trois placements simples tolérant respectivement les patrons Π_1, Π_2 et Π_3 ; (d) Placement avec réplication résultant de la fusion de ces trois placements simples.

3.5.6 Fusion des placements simples

En fusionnant ces trois placements simples, j'obtiens un nouvel placement qui est, par construction, tolérants aux trois patrons de fautes Π_1, Π_2 et Π_3. C'est bien un placement avec réplication : par exemple, l'opération A est dupliquée sur deux processeurs : P1 et P4. L'algorithme de fusion consiste, pour chaque processeur Pi :

1. à calculer l'union sans répétition des ensembles d'opérations et de dépendances de données qui sont respectivement ordonnancées et routées sur Pi dans tous les placements simples,

2. puis à calculer un nouveau placement en ordonnançant chaque ensemble obtenu précédemment, c'est-à-dire un ordre total de ces opérations et dépendances de données, qui soit compatible avec l'ordre partiel défini par le graphe $\mathcal{A}lg$.

L'algorithme est le même pour les dépendances de données sur les bus de communication. De plus, chaque opération doit recevoir toutes ses dépendances de données entrantes, d'où des relations d'ordre supplémentaires. Par exemple, sur le processeur P1, les ensembles d'opérations et de dépendances de données provenant des placements simples des figures 3.37(a), (b) et (c) sont respectivement $\emptyset$, $\{$B$\}$ et $\{$A, A▷B, B$\}$. Leur union sans répétition est donc $\{$A,B, A▷B$\}$, et l'ordre total compatible avec le graphe $\mathcal{A}lg$ est A→A▷B →B, où « → » dénote l'ordre total. Le problème est que, en général, la fusion de deux DAGs n'est pas un DAG ! C'est ce que montre la figure 3.38.

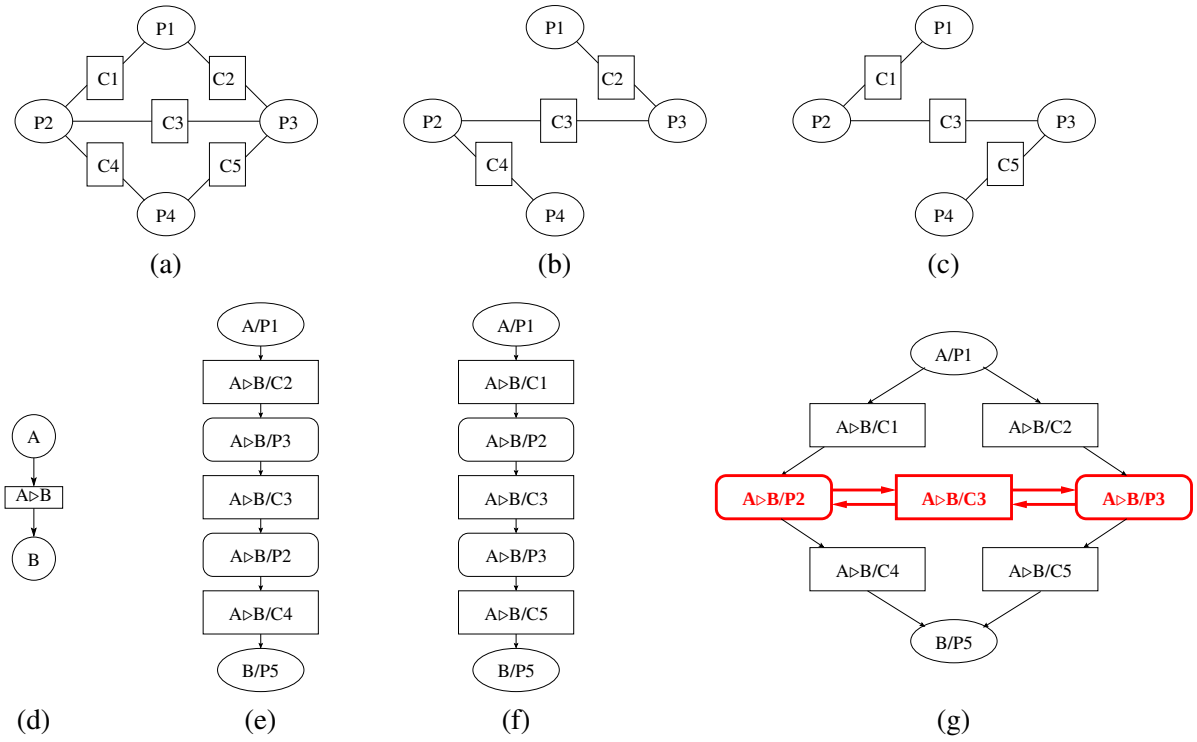


FIG. 3.38 – (a) Un graphe $\mathcal{A}rc$; (b-c) Deux architectures réduites correspondant aux patrons de fautes $\Pi_i = \{\text{C1,C5}\}$ et $\Pi_2 = \{\text{C2,C4}\}$; (d) Un graphe $\mathcal{A}lg$; (e-f) Les deux placement simples de (d) respectivement sur (b) et (c) ; (g) L'union de (e) and (f), qui n'est manifestement pas un DAG puisqu'il y a des cycles !

3.5.7 Ordre pseudo-topologique

Toutefois, on peut remarquer que tout cycle dans un DAG, résultant de la fusion de DAGs représentant chacun un placement simple d'un même $\mathcal{A}lg$ sur un même $\mathcal{A}rc$, est tel que tous ses sommets sont étiquetés avec la *même* dépendance de données. C'est le cas du cycle du DAG de la figure 3.38, dont tous les sommets sont étiquetés par $A \triangleright B$. Cette propriété est essentielle pour relâcher la notion d'ordre total et introduire une notion plus faible qui est l'**ordre pseudo-topologique**.

Formellement, un **graphe d'algorithme** est un graphe orienté acyclique (DAG) biparti $G_A = (O_A, D_A, E_A)$, où O_A est l'ensemble des opérations, D_A est l'ensemble des dépendances de données entre les opérations de O_A, et E_A est l'ensemble des arcs entre éléments de O_A et de D_A. Un **graphe d'architecture** est un graphe non-orienté biparti $G_S = (P_S, C_S, E_S)$, où P_S est l'ensemble des processeurs, C_S est l'ensemble des bus de communication, et E_S est l'ensemble des arcs entre éléments de P_S et de C_S. Puis, un **placement avec réplication** de G_A sur G_S est un DAG étiqueté $G = (V, E, \lambda)$, où V est un ensemble de sommets, E est un ensemble d'arcs, et λ est une fonction d'étiquetage de V vers $O_A \cup D_A$, telle que :

S1 Pour tout $v \in V$ tel que $\lambda(v) = a \in O_A$, et pour tout b tel que $(b, a) \in E_A$, il existe $v' \in V$ avec $\lambda(v') = b$ et $(v', v) \in E$. Autrement dit, toute opération placée doit recevoir toutes ses dépendances de données entrantes.

S2 Pour tout $v \in V$ tel que $\lambda(v) = a \in D_A$, et pour tout b tel que $(b, a) \in E_A$, il existe une séquence $v_1, \ldots, v_k \in V$ avec $v_k = v$, $\lambda(v_1) = b$, $\lambda(v_2) = \ldots = \lambda(v_k) = a$ et $(v_i, v_{i+1}) \in E$ pour tout $1 \leq i \leq k-1$. Autrement dit, les transmissions de données peuvent être routées via plusieurs bus

de communication depuis leur processeur source jusqu'à leur processeur destination.

S3 Pour tout cycle $v_1, \ldots, v_k, v_{k+1} = v_1$ de G, il existe $d \in D_A$ tel que pour tout $1 \leq i \leq k$, $\lambda(v_i) = d$. Autrement dit, les seuls cycles possibles dans G sont ceux étiquetés par le même symbole de D_A. Cette propriété modélise le fait que les seuls cycles possibles dans G sont ceux qui apparaissent du fait de la fusion des ordonnancements, comme le cycle dans la figure 3.38(g) dont les sommets portent la même dépendance données A▷B.

Enfin, un **ordre pseudo-topologique** d'un placement $G = (V, E, \lambda)$ est une bijection $\phi : V \mapsto [1..p]$ (où $p = |V|$), telle que pour tout $v \in V$:

S4 Si $\lambda(v) \in O_A$, alors pour tout $d \in D_A$ tel que $(d, \lambda(v)) \in E_A$, il existe $v' \in V$ tel que $\lambda(v') = d$, $(v', v) \in E$ et $\phi(v') < \phi(v)$.

S5 Si $\lambda(v) \in D_A$, alors il existe $v' \in V$ avec $(v', v) \in E$ et $\phi(v') < \phi(v)$, tel que, ou bien $\lambda(v') = \lambda(v)$, ou bien $(\lambda(v'), \lambda(v)) \in E_A$.

S6 Pour tout $v' \in V$, si $(v', v) \in E$, $(\lambda(v'), \lambda(v)) \notin E_A$, et $\lambda(v') \neq \lambda(v)$, alors $\phi(v') < \phi(v)$.

L'algorithme 3.2 prend en entrée un placement $G = (V, E, \lambda)$ d'un DAG $A = (O_A, D_A, E_A)$, et renvoie comme résultat une bijection $\phi : V \mapsto [1..p]$ représentant un ordre pseudo-topologique de G.

Algorithme 3.2 Construction d'un ordre pseudo-topologique

```
1   k := 1;
2   X_0 := ∅;
3   tant que k ≤ p faire
4       choisir un sommet w ∈ V tel que
5           ⌘ Pour tout k, w ∉ X_{k-1}
6           ⌘ Pour tout w'' ∈ V tel que (w'', w) ∈ E, λ(w'') ≠ λ(w),
              et (λ(w''), λ(w)) ∉ E_A, alors w'' ∈ X_{k-1} ;
7           ⌘ Si λ(w) ∈ D_A, alors ∃ w' ∈ X_{k-1} avec ou bien λ(w') = λ(w)
              ou bien (λ(w'), λ(w)) ∈ E_A et tel que (w', w) ∈ E ;
8           ⌘ Si λ(w) ∈ O_A, alors ∀ b ∈ D_A avec (b, λ(w)) ∈ E_A, ∃ w' ∈ X_{k-1}
              tel que λ(w') = b et (w', w) ∈ E ;
9       fin choisir
10      X_k := X_{k-1} ∪ {w} ;
11      φ(k) := w ;
12      k := k + 1;
13  fin tant que
fin
```

La preuve de l'existence d'un ordre pseudo-topologique pour tout placement G d'un DAG G_A se fait par contradiction : l'impossibilité de faire le choix de la ligne 4 implique l'existence d'un cycle dans G, dont les sommets seraient étiquetés par des symboles différents de $O_A \cup D_A$, ce qui contredit la propriété **S3**. Il en découle que l'algorithme 3.2 termine toujours et que $X_p = V$. Par conséquent, l'application $\phi : V \mapsto [1 \ldots p]$ définie par $\phi(w) = k$ si et seulement si w est le sommet choisi lors de la k-ième itération, représente un ordre pseudo-topologique sur V.

3.5.8 Calcul des dates de début d'exécution

Une fois cet ordre pseudo-topologique obtenu, il reste à calculer les dates de début et de fin d'exécution de chaque opération (resp. dépendance de données) sur chaque processeur (resp. bus de communication). Cette dernière étape permet d'obtenir un véritable ordonnancement statique et tolérant aux fautes, et non plus un simple placement des opérations et dépendances de données (comme par exemple celui de la figure 3.37(d)) ; il est pour cela nécessaire de prendre en compte les WCET et WCTT fournis

par l'utilisateur dans le tableau $\mathcal{E}xe$. Là encore se pose le problème que le graphe résultant de la fusion des placements simples peut contenir des cycles, ce qui complique le calcul. La solution repose une nouvelle fois sur l'ordre pseudo-topologique. Les détails sont dans l'article [27]. Par exemple, à partir du placement de la figure 3.37(d), j'obtiens l'ordonnancement de la figure 3.39.

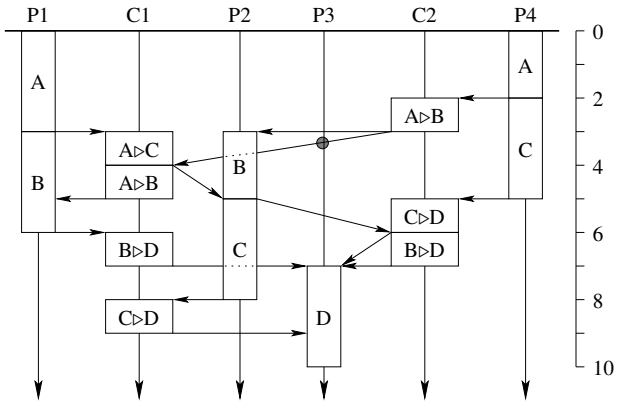


FIG. 3.39 – Ordonnancement final pour le placement de la figure 3.37(d).

3.5.9 Résultats d'exécution de l'heuristique HFOS et discussion

Les figures 3.40, 3.41 et 3.42 comparent la longueur moyenne des ordonnancements obtenus par les méthodes HFOS et AAA-TP, respectivement pour trois valeurs de CCR : 0.1, 1 et 10. Dans les trois cas, le graphe $\mathcal{A}rc$ quatre processeurs (P1, P2, P3 et P4) et est complet (six liens de communication point-à-point). Pour chaque test, 50 graphes $\mathcal{A}lg$ ont été générés aléatoirement, avec des tailles variant de 25 à 100 opérations (voir la section 3.3.7). Les paramètres de la méthode AAA-TP sont $\mathcal{N}pf$=1 et $\mathcal{N}lf$=0. Les patrons de fautes pour la méthode HFOS sont $\Pi_1 = \{\text{P1}\}$, $\Pi_2 = \{\text{P2}\}$, $\Pi_3 = \{\text{P3}\}$ et $\Pi_4 = \{\text{P4}\}$.

On constate que la méthode AAA-TP donne presque toujours un meilleur résultat que la méthode HFOS. La raison en est que le taux de réplication des opérations est plus élevé avec HFOS qu'avec AAA-TP. En effet, pour AAA-TP, $\mathcal{N}pf$=1 et $\mathcal{N}lf$=0 implique que le taux de réplication des opérations est exactement égal à 2. En revanche, pour HFOS, comme il y a quatre patrons de fautes, cela implique quatre placements simples, donc le taux maximal de réplication des opérations est égal à 4. En réalité, selon les graphes $\mathcal{A}lg$, il varie autour de 3. Cette différence explique le surcoût de HFOS par rapport à AAA-TP. En revanche, quand CCR=10, HFOS et AAA-TP ont des performances similaires, car le coût élevé des communications favorise les ordonnancements à forte localité, ce qui est le cas de ceux produits par HFOS puisque leur taux de réplication est supérieur.

En fait, la méthode HFOS n'est pas du tout faite pour tolérer un nombre fixe de fautes de composants matériels (au contraire de AAA-TP). Elle est faite pour tolérer les fautes de sous-ensembles précis de composants matériels. À ma connaissance, il n'existe aucun autre travail de recherche similaire à la méthode que je viens de présenter. Donc il n'est pas possible de tester, « mieux » que je ne l'ai fait dans cette section les performances de la méthodes HFOS (c'est-à-dire en comparant HFOS et AAA-TP), car il n'y a aucune méthode pouvant servir de point de comparaison.

Enfin, pour cette même raison, l'état de l'art est le même que pour les méthodes de génération d'ordonnancements statiques multiprocesseur et tolérants aux fautes (sections 3.4.2.7 et 3.4.3.7).

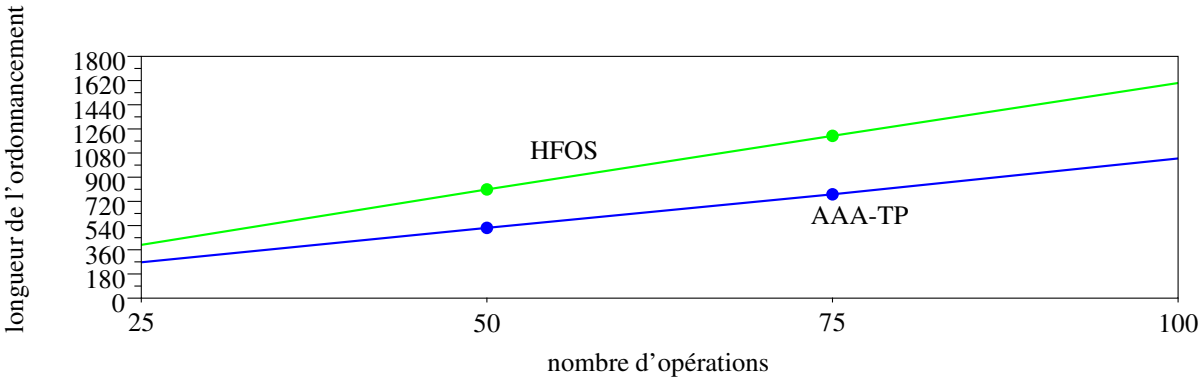


FIG. 3.40 – Comparaison de HFOS avec AAA-TP pour CCR=0.1.

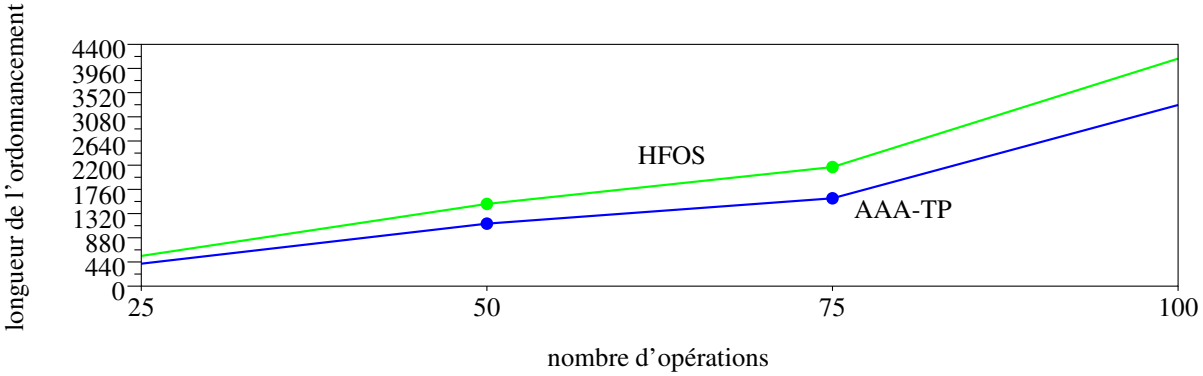


FIG. 3.41 – Comparaison de HFOS avec AAA-TP pour CCR=1.

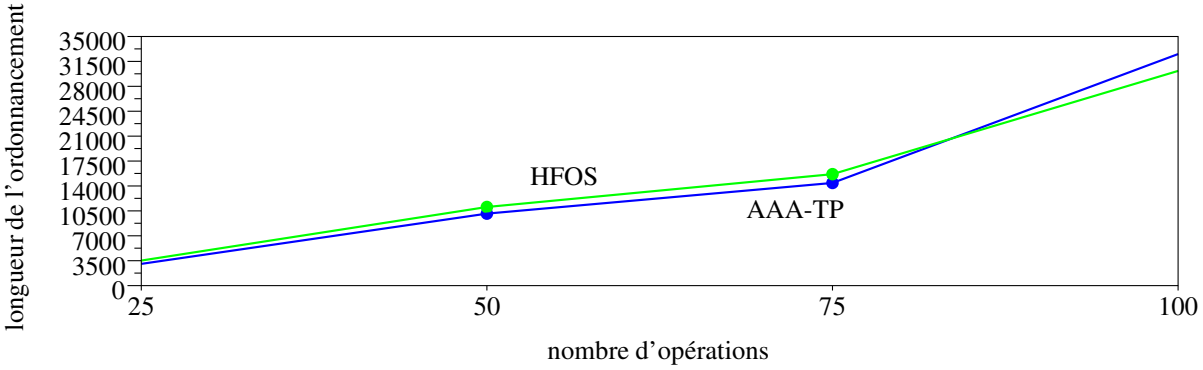


FIG. 3.42 – Comparaison de HFOS avec AAA-TP pour CCR=10.

3.6 Synthèse de contrôleurs pour la tolérance aux fautes

La synthèse de contrôleurs discrets est une approche formelle, apparentée au « model checking », très prometteuse dans le sens où elle permet d'obtenir automatiquement des systèmes satisfaisant par construction des propriétés spécifiées a priori. Comme je vais le montrer dans cette section, cette approche peut s'appliquer de façon élégante à la tolérance aux fautes, en rangeant les fautes dans la catégorie des événement incontrôlables. Les travaux présentés dans cette section ont été publiés dans les

articles [43, 31, 44] et dans les rapports de DEA de Mohamed Abdennebi [2], Safouan Taha [112], Huafeng Yu [123] et Nour Brinis [18].

3.6.1 Contexte

3.6.1.1 Synthèse de contrôleurs discrets

La **synthèse de contrôleurs discrets** (SCD) est une approche inventée par Ramadge et Wonham dans les années 80 [94], fondée sur la théorie des langages. Son but est, à partir de deux langages $\mathcal{U}$ et $\mathcal{D}$, d'obtenir un troisième langage $\mathcal{C}$ tel que :

$$\mathcal{U} \cap \mathcal{C} \subseteq \mathcal{D} \tag{3.28}$$

C'est donc une sorte de problème d'inversion, puisqu'il est possible de voir la solution recherchée comme étant égale à $\mathcal{C} = \mathcal{D}.\mathcal{U}^{-1}$, à condition de définir les opérations '.' et '$^{-1}$ sur les langages.

Les trois langages $\mathcal{U}$, $\mathcal{D}$ et $\mathcal{C}$ représentent respectivement le **procédé**, le **système désiré**, et le **contrôleur**. $\mathcal{U} \cap \mathcal{C}$ est quant à lui le **système contrôlé**. L'alphabet $\mathcal{E}$ du langage $\mathcal{U}$ est partitionné en deux sous-alphabets : l'ensemble $\mathcal{E}_C$ des événements **contrôlables** et l'ensemble $\mathcal{E}_I$ des événements **incontrôlables**. Le premier point clé de la SCD est que le contrôleur ne peut agir que sur les événements contrôlables du procédé. Le second point clé est que le contrôleur synthétisé est **le plus permissif possible**, c'est-à-dire que le langage $\mathcal{U} \cap \mathcal{C}$ doit être le plus grand possible inclus dans $\mathcal{D}$. Ramadge et Wonham ont réalisé une suite d'outils de SCD, baptisée TCT.[10]

La SCD peut échouer pour un objectif donné $\mathcal{D}$. Ceci signifie qu'il n'est pas possible de trouver un contrôleur $\mathcal{C}$ agissant uniquement sur $\mathcal{E}_C$ et tel que $\mathcal{U} \cap \mathcal{C} \subseteq \mathcal{D}$.

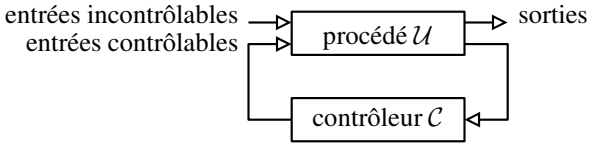


FIG. 3.43 – Système $\langle$procédé, contrôleur$\rangle$ en boucle fermée.

Enfin, il est classique de présenter le couple $\langle$procédé, contrôleur$\rangle$ comme un **système en boucle fermée**, où le contrôleur observe le procédé et agit sur son comportement (figure 3.43).

Plus récemment, plusieurs équipes ont appliqué et étendu ces techniques de théorie des langages aux **système de transitions étiquetées** (STE), tant dans le domaine de l'informatique que de l'automatique discrète. L'algorithmique liée à la SCD est la même que celle liée au model checking : c'est essentiellement du calcul d'atteignabilité et d'invariance d'ensembles d'états, ou bien en énumératif ou bien en symbolique avec des BDDs (« Binary Decision Diagram »). L'outil SIGALI [11] [85], que j'ai utilisé pour mes travaux propose ces deux techniques.

3.6.1.2 Systèmes de transitions étiquetées

Formellement, un STE est un n-uplet $S = \langle \mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, \mathcal{T} \rangle$, où $\mathcal{Q}$ est un ensemble fini d'états, q_0 est l'état initial de S, $\mathcal{I}$ est un ensemble fini de d'événements d'entrées (produits par l'environnement), $\mathcal{O}$ est une ensemble fini d'événements de sorties (émis vers l'environnement), et $\mathcal{T}$ est la relation de transition, c'est-à-dire un sous-ensemble de $\mathcal{Q} \times \mathcal{B}ool(\mathcal{I}) \times \mathcal{O}^* \times \mathcal{Q}$, où $\mathcal{B}ool(\mathcal{I})$ est l'ensemble des expressions booléennes sur $\mathcal{I}$ et $\mathcal{O}^*$ est l'ensemble des chaînes, éventuellement vides, d'éléments de $\mathcal{O}$.

[10] TCT : `http://www.control.utoronto.ca/people/profs/wonham`.
[11] SIGALI : `http://www.irisa.fr/vertecs/Logiciels/sigali.html`.

Chaque transition du STE a une **étiquette** de la forme g/a, où $g \in \mathcal{B}ool(\mathcal{I})$ doit être vrai pour que la transition soit prise (g est la **garde** de la transition), et où $a \in \mathcal{O}^*$ est une conjonction de sorties qui sont émises quand la transition est prise (a est l'**action** de la transition). L'état q est la **source** de la transition (q, g, a, q'), et l'état q' est sa **destination**. Une transition (q, g, a, q') sera représentée graphiquement par $q \xrightarrow{g/a} q'$.

Un STE est **déterministe** (resp. **réactif**) si et seulement si, pour tout état $q \in \mathcal{Q}$ et pour toute valuation des entrées, il existe au plus (resp. au moins) une transition partant de q dont la garde est vraie pour cette valuation.

Le mode de composition de deux STEs mis en parallèle est le **produit synchrone**, noté $||$, tel que défini par Milner [86]. Le produit synchrone est commutatif et associatif. Formellement :

$$\langle \mathcal{Q}_1, q_{0,1}, \mathcal{I}_1, \mathcal{O}_1, \mathcal{T}_1 \rangle \,||\, \langle \mathcal{Q}_2, q_{0,2}, \mathcal{I}_2, \mathcal{O}_2, \mathcal{T}_2 \rangle = \langle \mathcal{Q}_1 \times \mathcal{Q}_2, (q_{0,1}, q_{0,2}), \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{T} \rangle$$
$$\text{avec } \mathcal{T} = \{((q_1, q_2) \xrightarrow{g_1 \wedge g_2 / a_1 \wedge a_2} (q_1', q_2')) \mid q_1 \xrightarrow{g_1/a_1} q_1' \in \mathcal{T}_1, q_2 \xrightarrow{g_2/a_2} q_2' \in \mathcal{T}_2\}.$$

Comme tout opérateur de produit de STEs, le produit synchrone provoque l'*explosion combinatoire*, puisque le nombre d'états de $S_1||S_2$ est au pire égal au produit du nombre d'états de S_1 et de S_2. Toutefois, il limite cette explosion par rapport au produit asynchrone, grâce à la mise en commun d'états successeurs.

Un **chemin** dans le STE $\langle \mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, \mathcal{T} \rangle$ est une séquence de transitions $q_1 \xrightarrow{g_1/a_1} q_2 \xrightarrow{g_2/a_2} q_3 \cdots q_n \xrightarrow{g_n/a_n} q_{n+1}$. Un état q de $\mathcal{Q}$ est **atteignable** si et seulement si il existe un chemin de l'état initial q_0 à q. Un ensemble d'états E est **atteignable** si et seulement si tous ses états le sont. Un ensemble d'états E est **invariant** si et seulement si toute transition ayant pour source un état de E a pour destination un état de E.

J'utilise les Automates de Modes de Maraninchi et Rémond comme langage de programmation de STEs [83]. Un automate de modes est un STE dont chaque état représente un mode de fonctionnement du programme, spécifié par des équations flots de données entre les entrées et les sorties. Les Automates de Modes utilisent le produit synchrone pour composer plusieurs programmes en parallèle. Ceci permet à l'utilisateur de programmer de façon propre et modulaire. Le compilateur associé aux Automates de Modes, MATOU, compile un STE en équations polynomiales dans $\mathbb{Z}/3\mathbb{Z}$, qui est le format d'entrée de SIGALI ($\mathbb{Z}/3\mathbb{Z}$ étant le corps de Galois à trois éléments, $\{-1, 0, 1\}$).

3.6.1.3 Synthèse de contrôleur sur les systèmes de transitions étiquetées

Le procédé $\mathcal{U}$ est spécifié par un STE, plus exactement comme étant le résultat du produit synchrone de plusieurs STEs. L'ensemble $\mathcal{I}$ des entrées de $\mathcal{U}$ est partitionné en deux sous-ensembles : l'ensemble $\mathcal{I}_C$ des entrées contrôlables et l'ensemble $\mathcal{I}_I$ des entrées incontrôlables. Une transition est **contrôlable** si et seulement si il existe au moins une valuation des entrées contrôlables telle que sa garde soit fausse ; sinon elle est **incontrôlable**.

$\mathcal{D}$ est l'objectif que le système contrôlé doit satisfaire, typiquement *rendre invariant un sous-ensemble des états de* $\mathcal{U}$, ou *garder inatteignable un sous-ensemble des états de* $\mathcal{U}$. $\mathcal{D}$ est usuellement spécifié par un prédicat sur les états de $\mathcal{D}$. De façon équivalente, $\mathcal{D}$ peut être spécifié par un STE.

Le contrôleur $\mathcal{C}$ obtenu par SCD agit en restreignant les transitions de $\mathcal{U}$, c'est-à-dire en inhibant celles qui empêcheraient l'objectif $\mathcal{D}$ d'être satisfait. $\mathcal{C}$ ne peut inhiber que des transitions contrôlables. Pour cela, il choisit une valuation des entrées contrôlables telle que la garde de la transition à inhiber soit fausse. C'est le contrôleur le plus permissif possible : si dans un état donné il a le choix entre deux transitions contrôlables qui satisfont $\mathcal{D}$, alors le contrôleur ne choisit pas entre les deux, et donc le système contrôlé résultant est *indéterministe*.

Si la SCD échoue pour un objectif donné $\mathcal{D}$, puisque *tout* l'espace d'état est parcouru pendant la synthèse (que ce soit exhaustivement ou symboliquement), cela signifie qu'il est impossible de restreindre le procédé $\mathcal{D}$ en ne coupant que des transitions contrôlables.

3.6.1.4 Synthèse optimale

Il est également possible d'associer, à chaque transition et/ou à chaque état de $\mathcal{U}$, un **poids**, et de spécifier une **fonction de combinaison** des poids. Cette fonction est alors utilisée lors du calcul du produit synchrone, et on peut exiger qu'elle ne dépasse jamais une borne supérieure ou inférieure fournie, ou même qu'elle soit maximisée ou minimisée. C'est ce que fait la **synthèse optimale** [71, 114, 105, 84]. Une telle maximisation ou minimisation peut porter sur une seule transition ou sur des chemins de longueur finie. Remarquons que la synthèse optimale ne garantit pas que le système contrôlé sera déterministe, mais seulement que ce sera le plus permissif possible qui optimise la fonction de combinaison. Il est en effet possible d'avoir deux transitions contrôlables telles que le résultat de la fonction de combinaison soit le même dans les deux cas.

3.6.2 Principes généraux d'utilisation de la SCD pour la tolérance aux fautes

Dans l'optique de la tolérance aux fautes, il est naturel de considérer les événements représentant les fautes elles-mêmes comme des événements incontrôlables. Puis, il faut représenter dans le procédé $\mathcal{U}$ tous les comportements, aussi bien les bons (dans lesquels aucune faute ne se produit, ou celles qui se produisent sont masquées) que les mauvais (dans lesquels au moins une faute fait que le système ne fournit plus son service nominal). Enfin, il faut exprimer dans le système désiré $\mathcal{D}$ le fait qu'un certain nombre de fautes doivent être tolérées. En synthétisant un contrôleur $\mathcal{C}$ garantissant que $\mathcal{U} \cap \mathcal{C}$ satisfait les propriétés de $\mathcal{D}$, on obtient bien *automatiquement* un système tolérant aux fautes. C'est cette approche que j'ai explorée au cours des quatre dernières années, en collaboration avec Éric Rutten, et en encadrant quatre étudiants de DEA (Mohamed Abdennebi, Safouan Taha, Huafeng Yu et Nour Brinis) et un postdoc (Emil Dumitrescu).

Par ailleurs, l'hypothèse de faute sera modélisée par un STE qui sera ensuite composé en parallèle avec le reste de la spécification du procédé. Cette approche présente deux avantages, premièrement celui d'être flexible car il est possible de changer l'hypothèse de fautes sans modifier le reste de la spécification, et deuxièmement celui d'être formelle grâce à l'utilisation d'un STE.

Dans la suite de cette section, je présente en détails trois études de cas qui illustrent ces deux principes généraux :

- un système temps-réel, multiprocesseur et multitâche, tolérant aux fautes des processeurs : section 3.6.3 ;
- un système tolérant aux fautes en valeur des capteurs : section 3.6.4 ;
- et les généraux byzantins revisités : section 3.6.5.

Ces trois études de cas illustrent l'utilité et l'élégance de la SCD pour la tolérance aux fautes, pour différents types de fautes (crash, en valeur ou byzantines) appliqués à différents types de composants matériels (processeurs ou capteurs). Je termine par un état de l'art dans la section 3.6.6 et par une discussion dans la section 3.6.7.

3.6.3 Un système temps-réel tolérant aux fautes des processeurs

Une première idée a été de considérer un système réparti temps-réel constitué d'un ensemble de processeurs à silence sur défaillance, complètement connectés par un réseau de communication fiable, et d'un ensemble de tâches temps-réel périodiques s'exécutant sur ces processeurs. La figure 3.44(a) est une architecture répartie cible, comprenant trois processeurs complètement connectés, qui me servira d'exemple pour la suite. Chaque processeur P_i est modélisé par le STE de la figure 3.44(b) : l'état ERR_i

représente une défaillance, permanente puisqu'il n'y a pas de transition de ERR_i vers OK_i. Les événements f_i seront produits par un autre STE afin de tenir compte de l'hypothèse de faute (figure 3.45(a)). Enfin, chaque tâche τ_j est modélisée par le STE de la figure 3.44(c) : la tâche est d'abord en attente (état I^j), puis prête à débuter son exécution (état R^j suite à la requête d'exécution r^j), puis active sur un des trois processeurs P_1 (état A_1^j), P_2 (état A_2^j) ou P_3 (état A_3^j), et enfin terminée (état T^j suite à l'événement de terminaison t^j). Les événements r^j et t^j sont incontrôlables alors que les événements a_i^j sont contrôlables.

La transition de A_1^j vers A_2^j, étiquetée par a_2^j, modélise la **migration** de la tâche τ_j du processeur P_1 vers le processeur P_2. L'état de la tâche n'est pas transféré, donc sa migration consiste à la ré-activer sur le nouveau processeur P_2. Une telle migration sera décidée, par exemple, si le processeur P_1 est fautif. Pour une mise en œuvre en pratique, il faut utiliser les services d'un système d'exploitation afin d'exécuter plusieurs tâches sur un même processeur (en partage de temps) et afin de redémarrer une tâche sur son nouveau processeur suite à une migration. Du point de vue de la tolérance aux fautes, la migration d'une tâche suite à la défaillance de son processeur est de la réplication passive.

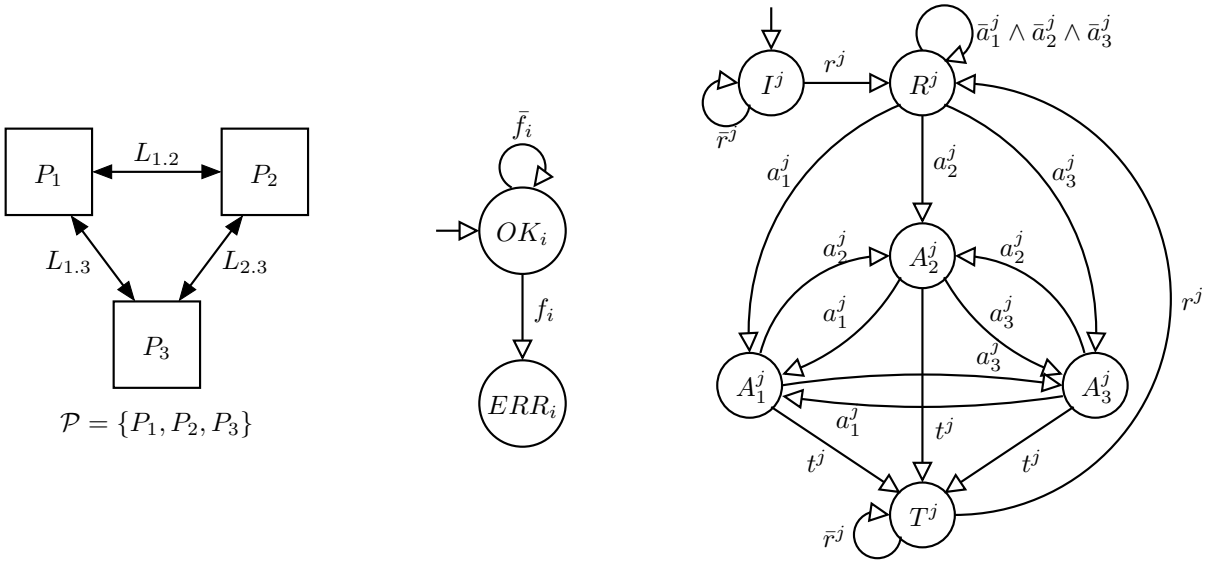
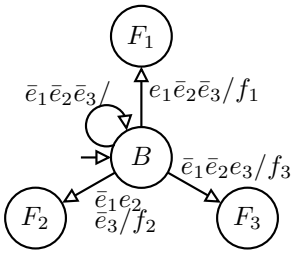


FIG. 3.44 – (a) Exemple d'architecture répartie cible ; (b) STE du processeur P_i ; (c) STE de la tâche temps-réel τ_j.

L'hypothèse de faute est que tous les processeurs sont à silence sur défaillance, qu'un seul processeur peut subir une faute, et que les défaillances sont permanentes. Ceci est représenté par le STE de la figure 3.45(a), qui produit les événements f_i en fonction des événements e_i, qui sont eux-même incontrôlables. Ce STE n'accepte que le premier événement e_i à se produire ; il produit alors l'événement f_i correspondant et va dans l'état F_i ; une fois dans cet état, il ne produit plus aucun événement f_i (la transition allant de l'état F_i à lui-même est implicitement étiquetée par $\bar{e}_1\bar{e}_2\bar{e}_3$, de telle sorte que le STE soit réactif). Par conséquent, un seul des STEs des processeurs (figure 3.44(b)) pourra passer de l'état OK_i à l'état ERR_i.

Dans ce modèle, tous les événements et toutes les variables d'état sont observables, donc le contrôleur détecte instantanément les défaillances des processeurs (passage à vrai de la variable d'état ERR_i). En pratique, cela peut être mis en œuvre au moyen de battements de cœur, chaque processeur envoyant périodiquement un message « je suis vivant » au contrôleur.



(a)

		consommation C par processeur			qualité Q par processeur		
		P_1	P_2	P_3	P_1	P_2	P_3
tâche	τ_1	4	4	2	3	5	3
	τ_2	2	2	3	2	2	5
	τ_3	2	3	4	2	2	5
	borne b	5	3	6			

(b)

FIG. 3.45 – (a) STE modélisant l'hypothèse de faute ; (b) Consommation énergétique C_i^j et qualité Q_i^j des tâches τ_j sur les processeurs P_i, avec les bornes b_i donnant la consommation énergétique maximale de chaque processeur.

Pour ne pas alourdir les dessins, certaines transitions ont été omises dans les STEs des figures 3.44(b), 3.44(c) et 3.45(a) : ce sont les transitions qui bouclent respectivement sur les états ERR_i, A_i^j et F_i de façon à rendre les STEs réactifs.

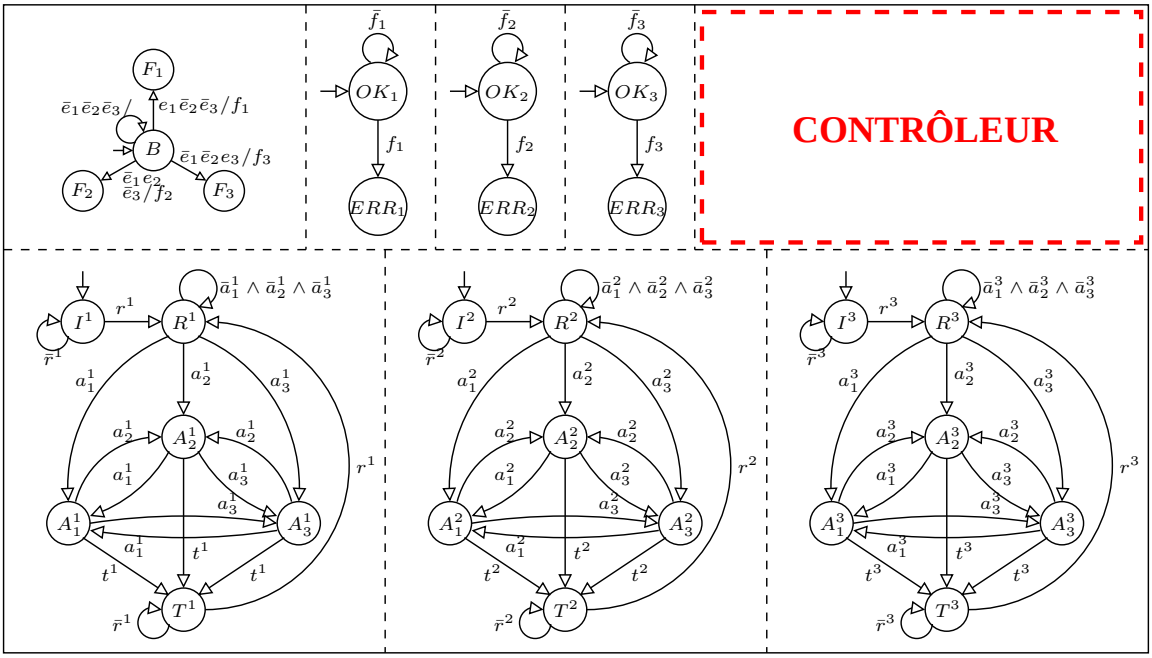


FIG. 3.46 – Système complet avec trois processeurs et trois tâches.

J'associe de plus deux poids à chaque état A_i^j : d'une part l'énergie consommée par la tâche τ_j quand elle est exécutée par le processeur P_i, et d'autre part la qualité du résultat calculé. La fonction de combinaison pour les deux poids est la somme, c'est-à-dire que la consommation énergétique cumulée (resp. la qualité cumulée) est la somme des consommations énergétiques (resp. des qualités) dans tous les états actifs du produit synchrone. En outre, chaque processeur dispose d'une consommation énergétique maximale qui ne doit pas être dépassée par les tâches qu'il exécute à un instant donné. Ces poids et ces bornes sont donnés dans la figure 3.45(b).

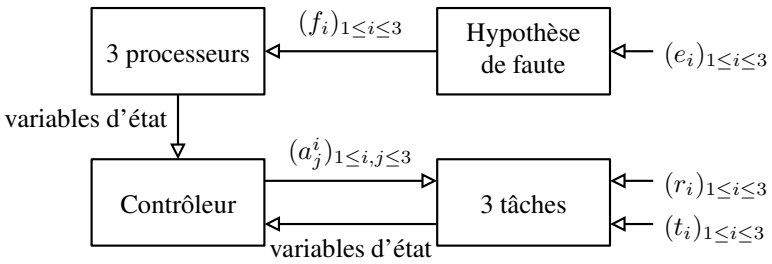


FIG. 3.47 – Schéma bloc du système complet en boucle fermée.

Le système complet est représenté dans les figures 3.46 et 3.47. Formellement, $\mathcal{U}$ est le résultat du produit synchrone des sept STEs de la figure 3.46. Une **configuration** du système est une affectation des tâches aux processeurs : par exemple, la configuration $\langle A^1_1|A^2_2|A^3_3\rangle$ indique que les tâches τ_1, τ_2 et τ_3 sont respectivement exécutées sur les processeurs P_1, P_2 et P_3, alors que la configuration $\langle A^2_1, A^3_1|\emptyset|A^1_3\rangle$ indique que τ_1 est sur P_3, que τ_2 et τ_3 sont sur P_1, et que P_2 n'exécute aucune tâche. Il y a 27 configurations différentes au total.

Le problème est alors de trouver un contrôleur garantissant les quatre propriétés suivantes :

SC1 Aucune tâche ne reste active sur un processeur défaillant : $\neg \bigvee_{j=1}^{3} \bigvee_{i=1}^{3} (A^j_i \wedge Err_i)$.

SC2 Le système réalise sa fonctionnalité : depuis tout état atteignable, les configurations terminales — dans lesquelles la propriété $\bigwedge_{i=1}^{3} T^i$ est vraie — sont atteignables.

SC3 Aucun processeur ne dépasse sa borne max de consommation énergétique : $\forall 1 \leq i \leq 3$, $\sum_{j=1}^{3} C^j_i \leq b_i$.

SC4 La qualité de chaque tâche est maximale, c'est-à-dire que la qualité cumulée $Q = \sum_{i=1}^{3} \sum_{j=1}^{3} Q^j_i$ est maximale.

Par exemple, la figure 3.48 illustre ce qui se produit en cas de défaillance de P_2 (événement incontrôlable e_2) quand le système est dans la configuration $\langle A^1_1|A^2_2|A^3_3\rangle$. Pour ne pas alourdir la figure, les négations d'événements ont été omises dans les étiquettes des transitions ; ainsi, l'étiquette de la transition du milieu est en réalité $\{e_2, \bar{a}^1_2, \bar{a}^1_3, \bar{a}^2_1, \bar{a}^2_3, \bar{a}^3_1, \bar{a}^3_2\}$. Grâce au positionnement à vrai de certains événements contrôlables a^j_i, trois transitions seront inhibées par le contrôleur, de façon à ne jamais atteindre les configurations interdites $\langle A^1_1, A^2_1|\emptyset|A^3_3\rangle$, $\langle A^1_1|A^2_2|A^3_3\rangle$ et $\langle A^1_1|\emptyset|A^2_3, A^3_3\rangle$.

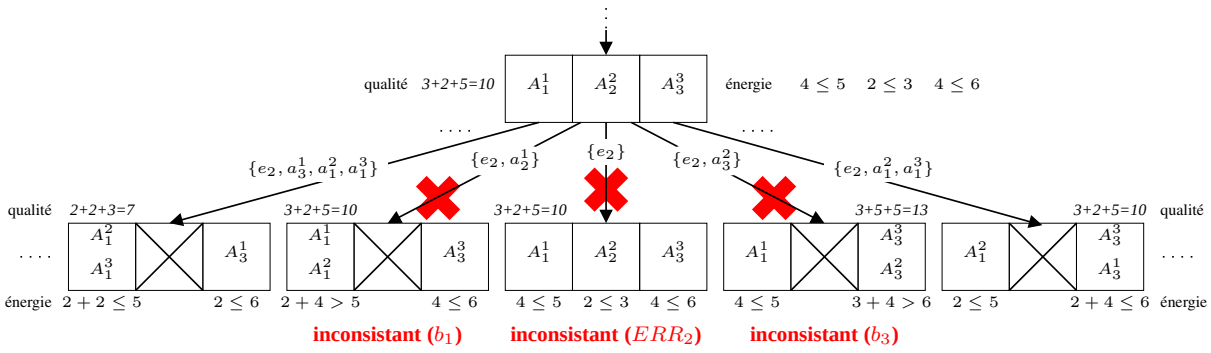


FIG. 3.48 – Trois configurations interdites par le contrôleur synthétisé.

Le contrôleur synthétisé à partir des seules propriétés **SC1** et **SC2** ne donne pas un système contrôlé déterministe, car il est *le plus permissif possible*. En revanche, en prenant en compte en plus les propriété-

tés **SC3** et **SC4**, la synthèse optimale garantit que le système contrôlé est *unique*, aux égalités numériques près.

Le système de la figure 3.46 a été programmé par Éric Rutten en MATOU, et, à partir des propriétés exprimées ci-dessus, SIGALI a synthétisé automatiquement un contrôleur rendant le système tolérant à une faute de processeur. À ma connaissance, c'était la première fois que la synthèse optimale de contrôleurs discrets était utilisée afin de rendre automatiquement un système tolérant aux fautes. Une telle approche est très séduisante de par la flexibilité qu'elle apporte : en effet, à partir d'un même modèle $\mathcal{U}$, la SCD permet de tester plusieurs objectifs de tolérance aux fautes et plusieurs hypothèses de fautes. De plus, quand la synthèse réussit, on obtient un système équipé d'un mécanisme *dynamique* de reconfiguration pour tolérer les fautes des processeurs, avec une garantie *statique* que toutes les fautes seront tolérées à l'exécution, et avec un borne connue sur le temps de réaction du système. En d'autres termes, on a les avantages de la réplication passive (reconfiguration dynamique et faible surcoût) et les avantages de la réplication active (garantie statique et borne du temps de réaction). Quand la synthèse échoue, dans la mesure où SIGALI explore *l'intégralité* de l'espace d'états, cela signifie qu'il n'existe pas de solution pour l'ensemble des fautes requises et pour les bornes imposées.

La SDC produit un système contrôlé qu'on peut voir comme un gestionnaire centralisé des tâches : ce système contrôlé (un STE) prend en entrée les événements incontrôlables et rend en sortie les événements de migration. On peut l'implanter sur un processeur dédié ou fabriquer à partir de lui une implantation répartie comme dans [31]. Dans les deux cas, le problème de la défaillance du ou des processeurs chargés d'exécuter le système contrôlé se pose. Une solution serait d'appliquer au système contrôlé des techniques classiques de tolérance aux fautes, par exemple la réplication active avec voteur, où le nombre de répliques est fonction du nombre de fautes pouvant affecter les processeurs.

L'inconvénient majeur vient du coût algorithmique : à l'heure actuelle, il n'est pas possible de traiter des systèmes plus gros que trois processeurs et trois tâches. Je reviendrai sur ce point dans la section 3.6.7.

3.6.4 Un système tolérant aux fautes en valeur des capteurs

La seconde idée que j'ai eue a consisté à tolérer les fautes en valeur des capteurs. La SCD étant limitée au domaine booléen, il a fallu se restreindre à des capteurs renvoyant une information booléenne. J'ai donc choisi de modéliser une cuve de liquide munie de quatre capteurs de niveau, sujets à des fautes en valeur, et de trois vannes quant à elles fiables. Le but est de contrôler les vannes de telles manière que la cuve ne soit jamais ni vide ni débordante, le niveau de la cuve étant inféré à partir des sorties renvoyées par les capteurs..

Les capteurs sont notés C_1, C_2, C_3 et C_4. Chacun est ou bien immergé ou bien sec, ce qui constitue bien une information booléenne. L'hypothèse de faute est que les quatre capteurs ont des fautes permanentes en valeur, et qu'un seul des quatre peut être fautif. Chaque capteur C_i peut ainsi être modélisé par le STE de la figure 3.49(a) : M_i est l'état immergé, S_i est l'état sec et ERR_i est l'état d'erreur. Le fait que les fautes soient permanentes est modélisé par l'absence de transition depuis l'état ERR_i. Selon le niveau initial du liquide dans la cuve, l'état initial du STE est S_i ou M_i. Par exemple, si le niveau initial est entre les capteurs C_2 et C_3, alors les états initiaux sont M_1, M_2, S_3 et S_4. En outre, le STE a une sortie qui vaut faux si le capteur est sec et vrai s'il est immergé ; je note c_i la sortie du capteur C_i. Par exemple, la transition $S_i \xrightarrow{\bar{f}_i m_i/1} M_i$ modélise le fait que le capteur C_i devient immergé et produit la sortie $c_i = 1$. Enfin, v_i est une entrée incontrôlable qui modélise précisément le fait que, quand le capteur C_i est fautif, sa sortie peut prendre n'importe quelle valeur. Par exemple, la transition $S_i \xrightarrow{f_i/v_i} ERR_i$ modélise le fait que le capteur C_i devient fautif et produit la sortie $c_i = v_i$. Le fait qu'un seul capteur peut être fautif est modélisé par le STE de la figure 3.49(b). Comme dans la section précédente, les événements e_i sont incontrôlables.

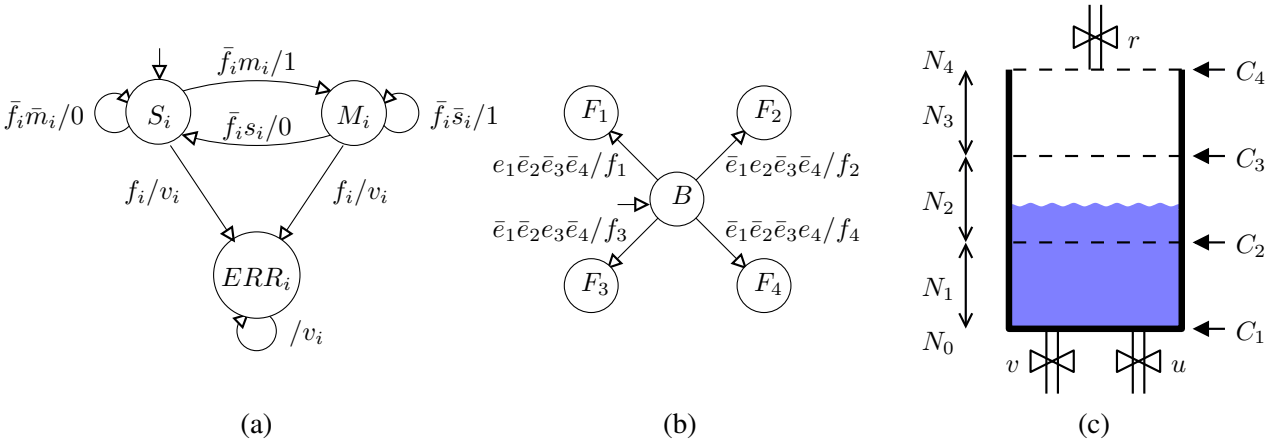


FIG. 3.49 – (a) STE modélisant le capteur C_i ; (b) STE modélisant l'hypothèse de faute ; (c) Cuve de liquide munie de quatre capteurs de niveau et de trois vannes.

Quand les quatre capteurs sont secs, la cuve est vide, niveau que je note N_0. Au contraire, quand les quatre capteurs sont immergés, la cuve déborde, niveau que je note N_4. Entre les deux, quand les capteurs C_1 à C_i sont immergés mais que les capteurs C_{i+1} à C_4 sont secs, je note ce niveau N_i. Les niveaux N_1 à N_3 sont donc des abstractions, puisqu'en réalité le niveau exact du liquide dans la cuve peut être n'importe où entre les capteurs C_i et C_{i+1}.

La cuve est en outre munie de trois vannes : une vanne de remplissage r (contrôlable), une vanne de vidange v (contrôlable) et une vanne pour l'utilisateur u (incontrôlable). L'événement r indique que la vanne de remplissage est ouverte, alors que l'événement $\bar{r}$ indique qu'elle est fermée (et de façon similaire pour les deux autres vannes). Je suppose enfin que les trois vannes ont exactement le même débit. Cette dernière hypothèse permet de simplifier grandement le STE qui modélise la cuve (figure 3.50).

La figure 3.49(c) illustre cette cuve. Le STE qui la modélise est présenté dans la figure 3.50. Il a cinq état, un pour chaque niveau possible du liquide. Ses entrées sont les événements r, v et u, tandis que ses sorties sont les événements m_i et s_i, $1 \leq i \leq 4$ qui entraînent, dans les STEs des capteurs, les changements d'état correspondant aux changements de niveau du liquide.

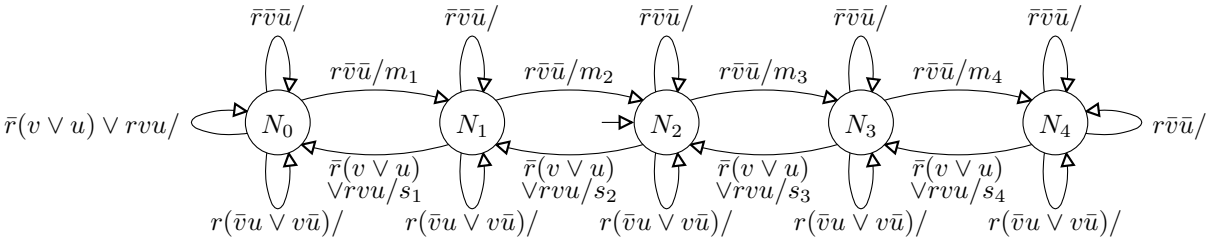


FIG. 3.50 – STE modélisant la cuve de la figure 3.49(c).

La figure 3.51 illustre le système complet constitué de la cuve, des quatre capteurs et de l'hypothèse de faute. Pour ne pas alourdir les dessins, les transitions qui bouclent sur les états F_i et B, et qui rendent le STE réactif, ont été omises dans la figure 3.49(b). Ainsi donc, le STE $\mathcal{U}$ du procédé est donc le résultat du produit synchrone du STE de la cuve, des quatre STEs des capteurs, et du STE de l'hypothèse de faute.

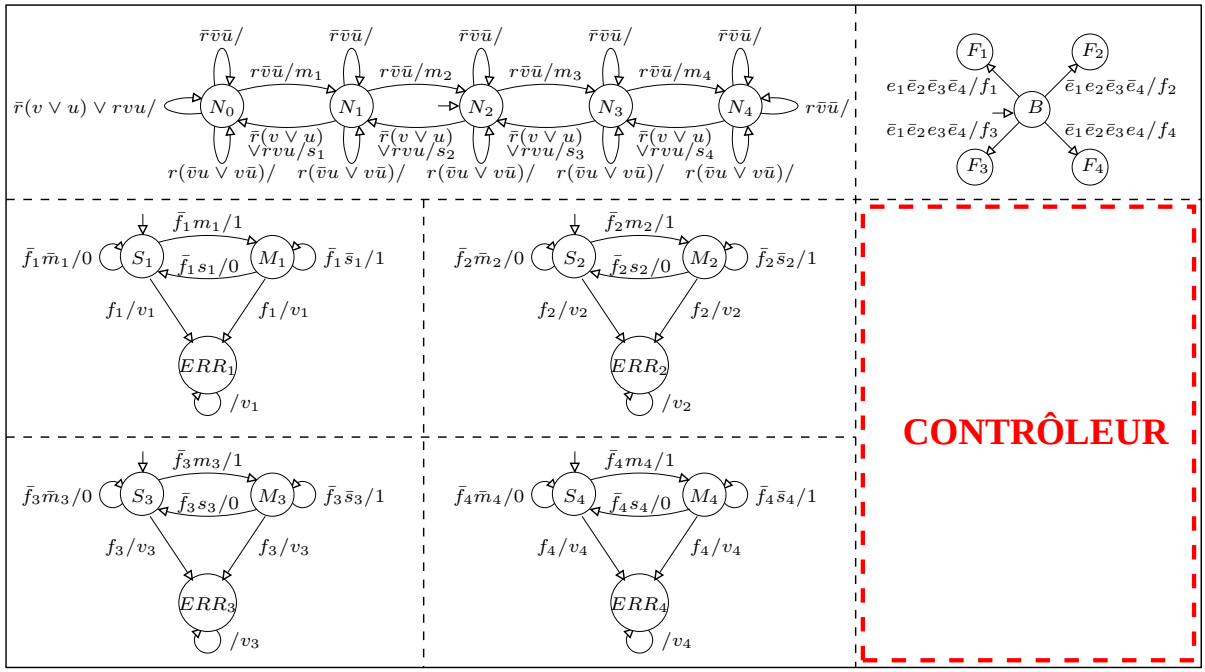


FIG. 3.51 – Système complet constitué de la cuve, des quatre capteurs et de l'hypothèse de faute.

Le but étant de contrôler les vannes de telle manière que la cuve ne soit jamais ni vide ni débordante, je cherche donc un contrôleur garantissant que les états N_0 et N_4 du STE de la figure 3.50 soient inatteignables. Ceci est formellement exprimé par la formule suivante :

$$\neg(N_0 \vee N_4) \tag{3.29}$$

Puisque la propriété (3.29) porte sur les variables d'état de la cuve (N_0 et N_4), cela signifie que le contrôleur synthétisé interagit avec la cuve en observant son état et en agissant sur les variables contrôlables r et v, et non pas en observant les sorties des quatre capteurs. Donc d'un côté la SCD garantit par construction que N_0 et N_4 sont inatteignables dans le système contrôlé (ce qui est le but recherché), mais d'un autre côté le contrôleur n'observe pas les sorties des quatre capteurs, donc il produit les événements r et v sans tenir compte des fautes susceptibles d'affecter les capteurs (ce qui n'est pas du tout ce qu'on recherche). Cette première approche naïve est illustré par la figure 3.52.

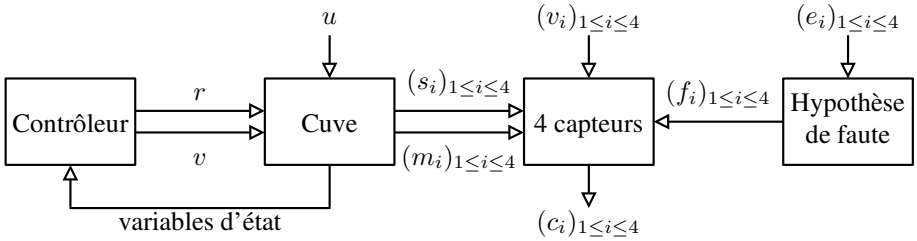


FIG. 3.52 – Contrôleur interagissant avec le procédé sans tenir compte des fautes des capteurs.

La solution proposée par Huafeng Yu consiste à ajouter un **observateur synchrone**, exactement comme pour le model checking [49] (et non pas comme les observateurs de Wong et Wonham qui sont des fonctions d'abstraction [119]) : c'est un STE dont le rôle est d'observer les sorties des capteurs c_i et

d'aller dans l'état BAD dès que ces sorties correspondent à la cuve vide (c'est-à-dire les quatre capteurs secs : $\bigwedge_{i=1}^{4} \bar{c}_i$) ou à la cuve débordante (c'est-à-dire les quatre capteurs immergés : $\bigwedge_{i=1}^{4} c_i$). Grâce à cet observateur, l'objectif de SCD devient tout simplement $\neg BAD$. Cette fois-ci, le contrôleur agit sur les entrées r et v en fonction de l'état des capteurs. Le STE de l'observateur est montré dans la figure 3.53(a), et le nouveau système contrôlé est montré dans la figure 3.53(b).

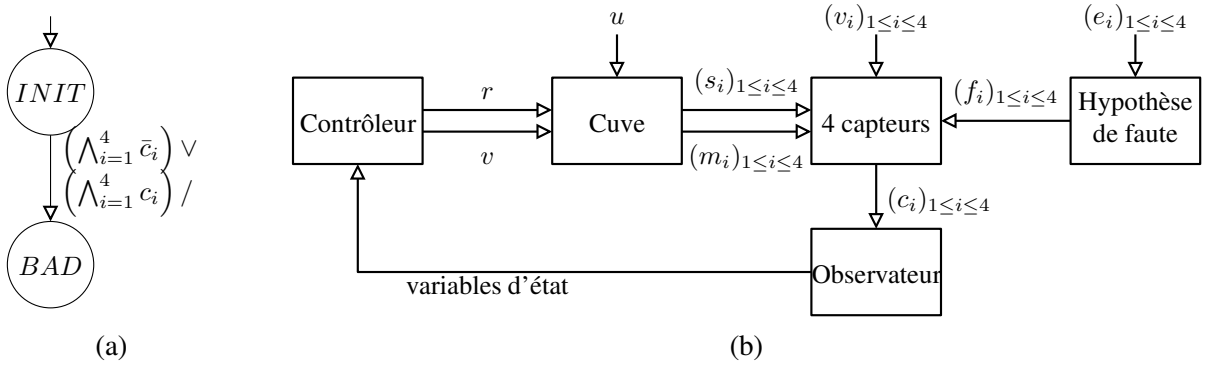


FIG. 3.53 – (a) Observateur synchrone pour le système de la cuve et des capteurs ; (b) Système contrôlé en boucle fermée.

Il reste alors à s'assurer que les états N_0 et N_4 sont bien inatteignables dans le système contrôlé, puisque maintenant cela n'est plus garanti directement par l'objectif de SCD, et cela peut se faire avec un outil de model checking, en vérifiant la propriété suivante sur le système contrôlé $\mathcal{U}||\mathcal{C}$:

$$\neg Reachable\left(\{N_0, N_4\}\right) \tag{3.30}$$

où le prédicat $Reachable\,(S)$ vérifie que l'ensemble d'états S est atteignable.

Le système de la figure 3.53(b) a été programmé par Huafeng Yu, ainsi que l'observateur synchrone de la figure 3.53(a), et SIGALI a synthétisé automatiquement un contrôleur rendant la cuve tolérante à une faute en valeur de capteur. Le système non contrôlé comporte 813 états et le système contrôlé 243, pour 47 variables d'état ; SIGALI permet d'isoler l'équation polynomiale dans $\mathbb{Z}/3\mathbb{Z}$ qui correspond au contrôleur seul $\mathcal{C}$, et de la transformer en un TDD (« Ternary Decision Diagram », équivalent ternaire des BDDs) ; dans le cas présent, la représentation ASCII du TDD ainsi obtenu fait 5,9 MO.

Par ailleurs, la propriété d'inatteignabilité des deux états de cuve vide et débordante a été prouvée par SIGALI. En revanche, si je modifie l'hypothèse de faute pour tenter de tolérer les fautes de deux capteurs, alors la SCD échoue ; ceci signifie que, dans le cas de fautes en valeur, deux capteurs défectueux parmi quatre rendent le système incontrôlable. En d'autres termes, ma cuve de liquide équipée de quatre capteurs peut tolérer exactement une faute en valeur. À ma connaissance, c'était la première fois que la SCD était utilisée pour tolérer des fautes en valeur, et qu'un observateur synchrone était utilisé conjointement avec la SCD. Comme dans la section 3.6.3, l'inconvénient majeur vient du coût algorithmique : à l'heure actuelle, il n'est pas possible de traiter des systèmes plus gros que quatre capteurs.

3.6.5 Les généraux byzantins revisités

Dans [72], Lamport et al. définissent le **problème des généraux byzantins** de la façon suivante : n divisions de l'armée byzantine, chacune commandée par son général, campent autour d'une cité ennemie. Les généraux doivent se mettre d'accord sur un plan de bataille commun, attaque ou retraite, en ne communicant que par des messagers. Le problème est que certains des généraux sont des **traîtres** qui essayent de perturber les généraux loyaux. Un des généraux est le **commandant** de l'armée, tandis que les $n-1$ autres sont ses **lieutenants**. Le général commandant envoie en premier un ordre (attaque ou retraite) à ses $n-1$ généraux lieutenants. Si le commandant est loyal, il envoie le même ordre à tous ses

lieutenants ; dans le cas contraire, il peut envoyer des ordres différents à ses lieutenants, c'est-à-dire des ordres incohérents. C'est suite à cette formulation que les fautes en valeurs et incohérentes sont appelées **fautes byzantines**. Puis, chaque lieutenant transmet l'ordre reçu à tous les autres lieutenants : là encore, un lieutenant loyal transmet l'ordre reçu de son commandant à tous les autres lieutenants ; en revanche, un lieutenant traître peut transmettre des ordres incohérents. Le but est de trouver un algorithme permettant aux généraux loyaux d'atteindre le consensus sur l'ordre de bataille. Formellement, les deux propriétés suivantes doivent être satisfaites :

IC1 Tous les lieutenants loyaux obéissent au même ordre.

IC2 Si le commandant est loyal, alors chaque lieutenant loyal obéit à l'ordre qui lui est envoyé.

Les algorithmes 3.3 et 3.4, proposés par Lamport et al., décrivent respectivement les actions du général commandant et des $n-1$ généraux lieutenants : m est le nombre de généraux pouvant être traîtres (le nombre effectif de traîtres n'est pas connu), v est l'ordre initial, et i est le numéro du lieutenant.

Algorithme 3.3 CommandantByzantin (m, v)
1 Envoyer mon ordre v aux $n-1$ lieutenants ;
fin

Algorithme 3.4 LieutenantByzantin (m, i)
1 v_i := valeur reçue du commandant ;
2 **si** $m = 0$ **alors**
3 Utiliser comme ordre la valeur v_i ;
4 **sinon**
5 Envoyer v_i aux $n-2$ autres lieutenants ;
6 **pourtout** $j \neq i$ **faire**
7 v_j := valeur reçue du lieutenant j ;
8 **fin faire**
9 Utiliser comme ordre la majorité $maj(v_1, v_2, \ldots, v_{n-1})$;
10 **fin si**
fin

Lamport et al. prouvent par induction sur m qu'il faut au moins $3m+1$ généraux pour garantir que tous les généraux loyaux prennent la même décision en présence d'au plus m traîtres. Trois hypothèses sur les messages échangés sont requises : chaque message envoyé est reçu correctement, le destinataire connaît l'envoyeur, et l'absence d'un message peut être détectée. En terme de système informatique, ces hypothèses sont satisfaites avec un réseau complètement connecté de liens point-à-point et avec des horloges synchronisées. De plus, la fonction $maj(v_1, v_2, \ldots, v_{n-1})$ est telle que si une majorité des valeurs v_i est égale à v, alors le résultat retourné est v.

La question à laquelle je désire répondre est la suivante : parmi les n généraux byzantins, combien au maximum peuvent être des traîtres ? Afin d'y répondre par SCD, je modélise l'environnement comme le STE *le plus permissif possible* recevant quatre entrées e_c, e_1, e_2 et e_3 (respectivement les traîtrises du commandant et des trois lieutenants) et produisant quatre sorties t_c, t_1, t_2 et t_3 (chacune de ces sorties sera utilisée en entrée respectivement par le LTS du commandant et des trois lieutenants ; voir la figure 3.54). Je note Loy_c et Tra_c respectivement l'état dans lequel le général commandant est loyal ou traître, et Loy_i et Tra_i l'état dans lequel le i-ième général lieutenant est loyal ou traître. Le STE suivant est donc l'environnement le plus permissif possible : $\langle Loy_c \xrightarrow{e_c/t_c} Tra_c\rangle \,||\, \langle Loy_1 \xrightarrow{e_1/t_1} Tra_1\rangle \,||\, \langle Loy_2 \xrightarrow{e_2/t_2} Tra_2\rangle \,||\, \langle Loy_3 \xrightarrow{e_3/t_3} Tra_3\rangle$. L'idée est que la SCD va *restreindre* ce modèle d'environnement en *empêchant* certains généraux d'être des traîtres, c'est-à-dire en *inhibant* certaines des transitions du modèle d'environnement. Autrement dit, je désire obtenir par SCD le STE le plus permissif *et* garantissant que les généraux vont atteindre le consensus en toutes circonstances.

Je note Att_c et $Retr_c$ respectivement l'état dans lequel le général commandant attaque ou fait retraite, at Att_i et $Retr_i$ l'état dans lequel le i-ième général lieutenant attaque ou fait retraite.

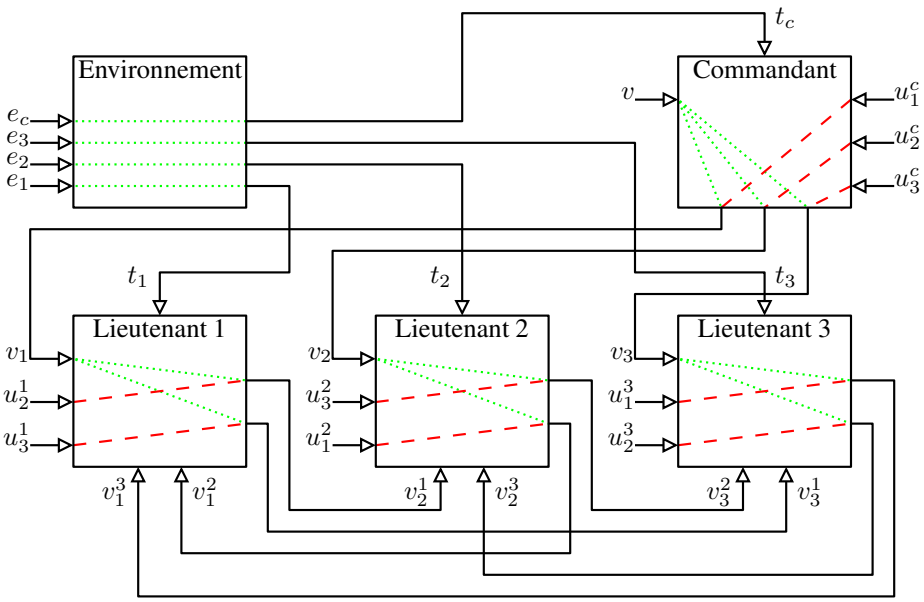


FIG. 3.54 – Système complet constitué du modèle de l'environnement, du commandant et des trois lieutenants.

Le STE du commandant a en entrée l'ordre initial v qu'il est censé transmettre à ses trois lieutenants. Pour modéliser le fait que, s'il est traître, alors il peut envoyer des messages incohérents, je lui ajoute trois entrées incontrôlables, u_1^c, u_2^c et u_3^c. Dans son mode de fonctionnement normal, ses trois sorties sont égales à v ; ceci est représenté dans la figure 3.54 par des traits pointillés verts. Mais quand il est traître, alors ses trois sorties sont chacune égale à une des trois entrées incontrôlables ; ceci est représenté par des traits interrompus rouges. De même, le STE du lieutenant 1 a en entrée l'ordre v_1, reçu de son commandant, et qu'il est censé transmettre aux deux autres lieutenants, par ses sorties v_2^1 et v_3^1 (et similairement pour les lieutenants 2 et 3). Pour modéliser le fait que, s'il est traître, alors il peut envoyer des messages incohérents, je lui ajoute deux entrées incontrôlables, u_2^1 et u_3^1. Là encore, les traits pointillés verts représentent le fonctionnement normal alors que les traits interrompus rouges représentent la traîtrise. Enfin, le STE du lieutenant 1 doit calculer la majorité des trois valeurs reçues, v_1, v_1^2 et v_1^3, afin de déterminer s'il doit aller dans l'état Att_1 ou $Retr_1$ (et similairement pour les lieutenants 2 et 3).

En terme de SCD, la propriété **IC1** se traduit en l'inatteignabilité de l'ensemble des états tels que le prédicat $\forall i \neq j, Loy_i \wedge Loy_j \wedge \big((Att_i \wedge Retr_j) \vee (Retr_i \wedge Att_j)\big)$ est vrai. Quant à la propriété **IC2**, elle se traduit en l'inatteignabilité de l'ensemble des états tels que le prédicat $\forall i, Loy_c \wedge Loy_i \wedge \big((Att_c \wedge Retr_i) \vee (Retr_c \wedge Att_i)\big)$ est vrai.

Nour Brinis a programmé en MATOU le système de la figure 3.54. En demandant à SIGALI de synthétiser un contrôleur à partir des deux propriétés ci-dessus, nous avons obtenu un système de quatre généraux byzantins tolérant la présence d'un traître parmi eux. Ce résultat est consistant avec le théorème de Lamport et al. L'originalité réside dans l'utilisation d'entrées incontrôlables pour modéliser des valeurs incohérentes, ainsi que dans l'utilisation de la SCD pour déterminer le nombre maximal de fautes byzantines admissibles (ceci en produisant le modèle de l'environnement le plus permissif possible et garantissant que les généraux vont atteindre le consensus en toutes circonstances). Comme dans la section 3.6.3, l'inconvénient majeur vient du coût algorithmique : à l'heure actuelle, il n'est pas possible de traiter des systèmes plus gros que quatre généraux byzantins.

3.6.6 État de l'art sur la synthèse de contrôleurs tolérants aux fautes

À ma connaissance, et bien qu'ils ne mentionnent aucun outil logiciel, Cho et Lim sont les premiers à avoir eu l'idée de rendre un système tolérant aux fautes grâce à la SCD, en considérant les fautes comme des événements incontrôlables [22]. Leurs travaux sont basés sur le contexte de travail de la SCD de Ramadge et Wonham [94]. Tout d'abord, l'ensemble des événements Σ est découpé en deux partitions : $\Sigma = \Sigma_c \cup \Sigma_{uc} = \Sigma_n \cup \Sigma_{an}$, respectivement les sous-ensembles des événements contrôlables, incontrôlables, normaux et anormaux ; de plus, $\Sigma_{an} \subset \Sigma_{uc}$. Par rapport à un ensemble d'états marqués Q_m (l'objectif du contrôle), ils définissent un **événement récurrent** comme étant tel que Q_m peut être atteint depuis l'état d'origine de cet événement, ou bien à travers des événements contrôlables, ou bien à travers d'autres événements eux-aussi récurrents. À partir de cette définition, une **faute** est un événement anormal qui n'empêche pas le système d'atteindre Q_m ; dans le cas contraire, c'est une **défaillance**. Leur idée est qu'une faute est juste un dysfonctionnement alors qu'une défaillance entraîne un arrêt grave du système. Enfin, un système est **tolérant aux fautes** par rapport à Q_m si, quand un événement anormal se produit durant son exécution, ou bien il existe une autre séquence d'événements conduisant à Q_m, ou bien le chemin jusqu'à cet événement anormal peut être éliminé. Une séquence d'événements composée uniquement d'événements normaux ou de fautes, et qui conduit de l'état initial à Q_m, est appelée une **séquence d'événements tolérant aux fautes** (SETF) si, pour tout événement normal, tous les événements possibles qui suivent l'état d'origine de cet événement, ou bien sont contrôlables, ou bien sont des événements récurrents. L'ensemble K de toutes les SETFs est alors choisi comme **langage légal**. Par construction, K est **réalisable**, c'est-à-dire à la fois contrôlable et observable. Enfin, le procédé est construit comme résultant de la composition parallèle de plusieurs automates d'états finis. Les différences par rapport à mes recherches résident dans :

- l'utilisation d'un ensemble d'états comme objectif de contrôle au lieu de propriétés d'invariance ou d'atteignabilité, qui sont plus faciles à utiliser en pratique, d'autant plus quand elles sont utilisées avec un observateur synchrone ;
- l'utilisation de la composition parallèle « standard » au lieu du produit synchrone : ce dernier limite l'explosion combinatoire, sans toutefois l'éviter complètement ;
- l'absence de définition claire de l'hypothèse de faute, alors que celle-ci est modélisée dans mes recherches par un STE, ce qui est à la fois formel et flexible ;
- l'utilisation de la synthèse optimale qui apporte de la richesse et qui permet de garantir l'unicité du contrôleur synthétisé, aux égalités numériques près ;
- enfin, l'utilisation d'un outil logiciel de SCD alors que Cho et Lim n'en mentionnent aucun.

Bien qu'ils ne fassent jamais référence à la SCD, la technique proposée par Kulkarni et Arora dans [69] et améliorée par Kulkarni et Ebnenasir dans [70] est très proche de ce que je présente dans cette section 3.6. Elle consiste à synthétiser automatiquement un programme tolérant aux fautes à partir d'un programme initial non tolérant aux fautes. Dans leur modèle, un programme est un ensemble d'états, chaque état représentant une valuation des variables du programme (représentation par un codage implicite), et un ensemble de transitions. Deux modèles d'exécution des programmes sont considérés : le modèle à **forte atomicité**, où un programme peut consulter et modifier toutes ses variables en une seule étape atomique (c'est-à-dire qu'il peut effectuer une transition depuis n'importe quel état vers n'importe quel autre état), et le modèle à **faible atomicité**, où il ne le peut pas. Le programme non tolérant aux fautes initial est supposé conforme à sa spécification en l'absence de fautes, mais aucune garantie n'est connue en présence de fautes. Ensuite, une faute est un sous-ensemble de l'ensemble des transitions du programme. Les auteurs considèrent trois niveaux de tolérance aux fautes :

la tolérance sûre : même en présence de fautes, le programme satisfait une propriété de sûreté (« fail-safe tolerance »),

la tolérance non-masquante : même en présence de fautes, le programme retourne à des états où il satisfait une propriété de sûreté et une propriété de vivacité (« non-masking tolerance »),

la tolérance masquante : conjonction des deux niveaux précédents (« masking tolerance »).

Afin de traiter leurs deux modèles d'atomicité et leurs trois niveaux de tolérance aux fautes, les auteurs proposent six algorithmes de transformation de programme. Dans le modèle à forte atomicité, leur algorithme est polynomial par rapport au nombre d'états du programme initial non tolérant aux fautes. Dans le modèle à faible atomicité, c'est exponentiel, et Kulkarni et Arora montrent même que le problème est NP-complet. Le principe de ces transformations de programmes est d'enlever récursivement des transitions, ce qui est très semblable à la SCD.

Pour prolonger le travail de Kulkarni et Arora, Gärtner and Jhumka proposent une méthode afin de traiter également les systèmes avec traces fermées par fusion [36]. Une spécification est **fermée par fusion** si et seulement si l'histoire entière de chaque trace est incluse dans chaque état de la trace (et donc que l'état suivant du système ne dépend que de son état courant et des entrées, c'est-à-dire pas des entrées précédentes). Pour transformer une spécification non fermée par fusion en une qui soit fermée par fusion, le moyen usuellement utilisé consiste à ajouter des **variables d'histoire** aux états, de façon à mémoriser la séquence des entrées précédentes. L'inconvénient est que c'est exponentiel en mémoire. Gärtner and Jhumka proposent une méthode polynomiale, qui revient à dissocier les traces de fusion (et donc à ajouter un nouvel état à chaque fois), puis à supprimer les mauvais états de fusion. Si n est le nombre d'états de la spécification non fermée par fusion, alors au pire le nombre d'états de la spécification équivalente fermée par fusion est $\mathcal{O}(n^2)$.

Attie et al. ont également proposé une méthode de synthèse automatique de programmes tolérants aux fautes [6]. Dans leur approche, un système est un ensemble de processus concurrents, consistant chacun en un graphe orienté, dont les transitions sont étiquetées par des commandes gardées. L'état du système est le vecteur contenant l'état de chaque processus concurrent. À chaque pas d'exécution, un processus est choisi au hasard pour tirer, à partir de son état courant, une transition valide. Afin de programmer de tels systèmes, la logique temporelle CTL est utilisée comme langage de spécification [32]. Ceci permet de distinguer entre la **partie sûreté** et la **partie vivacité** de la spécification (« safety » et « liveness »). Les fautes sont modélisées par des commandes gardées qui perturbent l'état du système. L'occurrence d'une faute est modélisée par un graphe orienté dont les transitions sont étiquetées par des commandes gardées de fautes. Attie et al. définissent les mêmes propriétés de tolérance aux fautes que Kulkarni et Arora : dans la **tolérance masquante**, à la fois la partie sûreté et la partie vivacité sont préservées ; dans la **tolérance non-masquante**, seule la partie sûreté est préservée, mais pas nécessairement la partie vivacité ; et dans la **tolérance sûre**, la partie vivacité est préservée mais pas nécessairement la partie sûreté.

Le point de départ du problème de synthèse avec tolérance aux fautes consiste en une spécification du problème (une formule CTL de la forme $init\_spec \wedge AG(global\_spec)$), une spécification de faute (une formule CTL F), une spécification de couplage problème-faute (une formule CTL $AG(coupling\_spec)$), et un type de tolérance TOL (ou bien masquante, sûre, ou non-masquante). Le but est de synthétiser un programme concurrent qui satisfasse $init\_spec \wedge AG(global\_spec)$ en absence de fautes, qui satisfasse $AG(coupling\_spec)$ en absence de fautes, et qui soit TOL-tolérant à F pour $init\_spec \wedge AG(global\_spec)$. Attie et al. utilisent la procédure de décision de [32] pour résoudre ce problème, c'est-à-dire pour synthétiser les comportements de recouvrement conformes aux propriétés de tolérance requises. La complexité de l'algorithme de synthèse proposé est exponentiel dans la taille de la spécification (problème plus couplage problème-faute).

Une différence importante entre mon approche et celles de Kulkarni et Arora et Attie et al. est que leur modèle de calcul (MdC) est l'entrelacement non-déterministe, alors que le mien est la composition parallèle synchrone. J'estime que, pour la conception de systèmes répartis à sûreté critique, un MdC déterministe est préférable à un MdC non-déterministe. Cette affirmation est motivée par les succès du MdC synchrone [13], en particulier dans l'avionique [17].

Kamach et al. ont appliqué la SCD à un système ayant plusieurs modes opératoires [58] (ce modèle est très similaire aux Automates de Modes de Maraninchi et Rémond [83]). Leur approche permet à

l'utilisateur de spécifier, par exemple, un **mode nominal** et un **mode dégradé** pour un sous-système, et de faire passer ce sous-système d'un mode à l'autre en fonction d'événements incontrôlables, appelés ici **événements de commutation**. Ils décrivent une étude de cas, une usine d'assemblage constituée de deux vérins hydrauliques et d'une ventouse. Le vérin horizontal a un mode dégradé, dans lequel il ne peut plus bouger. Les événements de commutation associés à ce vérin sont p (défaillance) et r (réparation). Du point de vue de la tolérance aux fautes, les conséquences de l'incontrôlabilité de p et r sont que les fautes sont d'une durée quelconque. Cette étude de cas a été réalisée à l'aide de l'outil TCT de Ramadge et Wonham.

Un autre domaine proche de la SCD est la **génération de plans**, technique issue de l'intelligence artificielle. Des travaux sur la génération de plans tolérants aux fautes existent [56], mais ils ne tolèrent qu'une seule faute, ce qui les rend, à mon avis, très limités.

Enfin, si on élargit le champ de recherches aux méthodes formelles, on trouve de nombreux travaux proposant la **vérification formelle** de propriétés de tolérance aux fautes sur des spécifications formelles produites manuellement, en particulier dans le contexte des algèbres de processus [101, 20, 14]. Ce qui distingue ces approches de la SCD est le fait que les propriétés de tolérance aux fautes sont vérifiées *a posteriori*, alors que les résultats que j'ai présentés dans cette section permettent de synthétiser un système vérifiant les propriétés requises de tolérance aux fautes *a priori*.

3.6.7 Discussion

J'ai présenté dans cette section trois solutions originales basées sur la SCD pour des problèmes de tolérance aux fautes : tolérance aux fautes des processeurs, tolérance aux fautes en valeur des capteurs, et tolérance aux fautes byzantines. J'ai également travaillé sur deux autres problèmes, eux-aussi résolus grâce à la SCD : avec Safouan Taha [112], tolérance aux fautes des actionneurs pour deux cuves de liquide communicantes munies de plusieurs vannes qui peuvent défaillir ; et enfin avec Emil Dumitrescu [31], tolérance aux fautes des liens de communication pour un arbitre d'accès à une ressource partagée. Les points forts d'une telle utilisation de la SCD sont :

l'automatisation car la SCD produit automatiquement un système tolérant aux fautes à partir d'un système non tolérant aux fautes ;

la séparation des préoccupations car le système non tolérant aux fautes peut être conçu indépendamment des exigences de tolérance aux fautes ;

la flexibilité car, une fois le système entièrement modélisé, il est facile de tester plusieurs hypothèses de fautes, plusieurs modèles de l'environnement, plusieurs objectifs de tolérance aux fautes...

la sûreté car, en cas de succès de la SCD, les propriétés requises de tolérance aux fautes sont garanties ;

l'optimalité quand la synthèse optimale est utilisée, modulo les éventuelles égalités numériques (donc une optimalité non stricte).

Le principal point faible est l'**explosion combinatoire**. C'est un inconvénient général de la SCD. Mon opinion est que la SCD est aujourd'hui au même stade que le model checking il y a 15 ans, c'est-à-dire que c'est une technique prometteuse mais d'usage limité à cause de sa complexité algorithmique. Concernant les travaux que je viens de présenter, deux points viennent encore aggraver cet inconvénient : d'une part la traduction du langage des STEs vers $\mathbb{Z}/3\mathbb{Z}$ qui est le formalisme d'entrée de SIGALI, et d'autre part l'exploration symbolique de l'espace d'états dans SIGALI qui est réalisée à l'aide de TDDs, équivalents ternaires des BDDs mais hélas moins efficaces. Mes espoirs pour le futur résident dans l'utilisation de l'interprétation abstraite [24], pour deux raisons : d'une part elle est utilisée avec succès pour l'exploration symbolique d'espaces d'états (par exemple dans l'outil NBAC [54]), et d'autre part elle permettra de traiter des systèmes avec valeurs numériques, donc les fautes en valeur.

3.7 Discussion et développements futurs

3.7.1 Comment choisir la bonne méthode pour un problème donné ?

J'ai présenté dans ce chapitre plusieurs méthodes pour améliorer la sûreté de fonctionnement d'un système sujet à des fautes matérielles : ce sont AAA-TP, AAA-TB, AAA-F et HFOS. Le premier critère pour choisir une de ces méthodes est le type de l'architecture répartie cible, qui exclue automatiquement certaines méthodes. Par exemple, si l'architecture comporte des bus, alors AAA-TP est exclue et il faut s'orienter vers AAA-TB ou HFOS. Le second critère est le niveau de sûreté de fonctionnement voulu. Par exemple, si l'utilisateur désire tolérer un nombre fixe de fautes, alors le choix se limite à AAA-TP ou AAA-TB. En revanche, si l'utilisateur désire atteindre un niveau donné de fiabilité, alors le choix peut se porter naturellement sur AAA-F, mais des méthodes comme AAA-TP peuvent toujours être utilisées à condition de calculer le nombre de fautes matérielles à tolérer afin d'atteindre la fiabilité requise ; toutefois, dans un tel cas, AAA-F sera probablement préférable car le surcoût engendré en longueur d'ordonnancement est moindre. Par ailleurs, si l'utilisateur désire tolérer non pas un nombre fixe de fautes matérielles mais les fautes de certains sous-ensembles des composants matériels de l'architecture répartie cible, alors HFOS est la méthode préférable. Pour un problème complexe, plusieurs méthodes différentes peuvent être appliquées aux divers sous-systèmes. Enfin, il ne faut pas oublier les méthodes visant à tolérer les fautes logicielles, même si ce chapitre ne les aborde pas du tout (programmation défensive, blocs de recouvrement, programmation N-versions, programmation N-autotestable...).

3.7.2 Ordonnancement & répartition bicritère longueur/fiabilité

Depuis septembre 2003, j'ai entrepris une collaboration fructueuse avec Denis Trystram (au laboratoire ID, Grenoble) sur les méthodes bicritères pour la sûreté de fonctionnement (co-encadrement de deux étudiants de DEA : Nicolas Leignel [78] et Érik Saule [100]). Ce sujet me passionne par ses fondements théoriques et ses applications potentielles. J'ai déjà obtenu des résultats prometteurs, à savoir la méthode AAA-F présentée à la section 3.4.5. L'axe vers lequel j'oriente mes recherches actuellement concerne d'une part le modèle de fiabilité, et d'autre part l'aspect théorique.

Concernant le modèle de fiabilité, il s'agit de trouver un modèle qui soit *indépendant du temps*. Dans le modèle le plus classiquement employé dans la littérature (y compris par moi), la probabilité de défaillance d'un composant suit une loi de Poisson à paramètre constant : la probabilité qu'un composant de taux de défaillance λ soit opérationnel pendant une durée d, c'est-à-dire sa fiabilité, est donc égale à $\exp^{-\lambda.d}$. Aussi cette fiabilité dépend-elle de deux facteurs : d'une part le taux de défaillance λ, et d'autre part la durée d. Quand on calcule la fiabilité d'un ordonnancement multiprocesseur, on retrouve ces deux facteurs : d'une part les taux de défaillance de tous les composants matériels de l'architecture cible, et d'autre part la longueur totale de l'ordonnancement. Par conséquent, quand on fait de l'ordonnancement bicritère longueur & fiabilité, on croit avoir à faire à deux critères antagonistes (dans le sens où améliorer la fiabilité nécessite plus de redondance, ce qui pénalise la longueur, et vice-versa), mais en réalité c'est plus compliqué puisque le critère longueur influe sur le critère fiabilité. Je me propose donc de réfléchir à utiliser comme critère une autre mesure représentative de la fiabilité globale mais qui serait indépendante de la longueur de l'ordonnancement, par exemple une sorte *taux de défaillance équivalent du système global*. Cela devrait aboutir à la mise au point de nouvelles méthodes bicritères d'ordonnancement & répartition qui soient plus efficaces.

Concernant l'aspect théorique, il s'agit de définir un cadre plus simple (par exemple en supposant que les liens de communication sont fiables) dans lequel des résultats théoriques d'ordonnancement pourront être trouvés, sous la forme d'un facteur maximal entre la longueur de l'ordonnancement obtenu au pire cas et la longueur de l'ordonnancement optimal. Là aussi, cela devrait aboutir à la mise au point de nouvelles méthodes bicritères d'ordonnancement & répartition qui soient plus efficaces. Les applications potentielles sont multiples, allant de la sûreté de fonctionnement (bicritère longueur & fiabilité) aux

logiciels embarqués (bicritère longueur & consommation électrique).

3.7.3 Transformation de programmes et programmation orientée aspects

Un second axe de recherches futures concerne l'obtention de programmes temps-réel tolérants aux fautes par des **transformations automatiques de programmes** et par la **programmation orientée aspects** [65], axe sur lequel je travaille avec Pascal Fradet et Tolga Ayav depuis septembre 2004 (tous deux à l'INRIA Rhône-Alpes). Le but recherché est la **séparation des préoccupations**, c'est-à-dire offrir une méthode de conception de systèmes tolérants aux fautes, où l'utilisateur spécifie *séparément*, d'une part la partie fonctionnelle de son système temps-réel (ce que fait le système), et d'autre part le niveau de tolérance aux fautes souhaité. À partir de ces deux données, nous cherchons à produire automatiquement un système ayant à la fois les fonctionnalités et le niveau de tolérance aux fautes spécifiés. L'originalité de notre approche réside dans la *formalisation* des transformations de programmes dans le cadre d'un langage de programmation simplifié, et dans la *preuve* que le système obtenu continue à satisfaire ses contraintes temps-réel quelles que soient les fautes qui se produisent (à condition qu'elles restent conformes à l'hypothèse de faute).

Les systèmes temps-réel considérés sont constitués de $n + 1$ processeurs à silence sur défaillance, reliés par un réseau de communication fiable, et d'une mémoire stable. Un seul des processeurs à la fois peut être sujet à une faute, temporaire ou permanente. De plus, n tâches périodiques indépendantes sont exécutées, chacune sur un processeur distinct, et avec une date d'échéance (« deadline » en anglais) égale à leur période. Un des processeurs est donc inutilisé : sur ce processeur, un programme moniteur est chargé de surveiller le fonctionnement correct des autres processeurs et de remplacer le premier qui défaille. En revanche, les tâches sont supposées être exemptes de fautes logicielles. Le moyen utilisé pour la détection des défaillances est le battement de cœur (« heart-beating ») : chaque tâche écrit périodiquement dans la mémoire stable pour indiquer que son processeur est toujours vivant, et en parallèle, le moniteur vérifie que tous les processeurs sont vivants. Pour le masquage des défaillances, le moyen utilisé est le point de sauvegarde (« checkpointing ») et la reprise (« rollback ») : chaque tâche sauvegarde périodiquement son contexte dans la mémoire stable, et dès que le moniteur détecte la défaillance d'un processeur, il restaure le contexte de la tâche qui vient d'être interrompue et reprend son exécution au dernier point sauvegardé.

Nous avons donc défini deux transformations automatiques sur le programme des tâches : la première pour insérer les battements de cœur, et la seconde pour insérer les points de sauvegarde. Mais pour prouver que les dates d'échéance continuent à être respectées quelles que soient les défaillances, il est impératif que, sur le programme *final* de chaque tâche τ_i, les battements de cœur soient périodiques (période T^i_{HB}), et que les points de sauvegarde le soient également (période T^i_{CP}). Cela soulève deux problèmes : d'une part l'insertion d'instructions pour faire un battement de cœur modifie le temps d'exécution du programme, et est donc susceptible de ne pas préserver la période des points de reprise, et d'autre part le respect des dates d'échéance dépend de la relation entre les deux périodes T^i_{HB} et T^i_{CP}. Les solutions auxquelles nous travaillons consistent à procéder à une transformation préalable du programme de chaque tâche τ_i, consistant à *égaliser* le temps d'exécution du code dans tous les chemins d'exécution (de telle sorte que son BCET soit égal à son WCET), et à calculer les valeurs *optimales* de T^i_{HB} et T^i_{CP} pour minimiser le temps d'exécution au pire cas, même en cas de défaillance du processeur sur lequel τ_i est exécutée. Nous avons également démontré *formellement* que ces transformations garantissent que chaque tâche respecte sa date d'échéance [9].

Enfin, de telles transformations de programme sont un premier pas vers une méthode de conception de systèmes tolérants aux fautes orientée aspects, dans laquelle les propriétés de tolérance aux fautes seront *tissées* dans la spécification fonctionnelle de l'utilisateur. La grande originalité d'une telle approche réside dans la définition d'un langage d'aspects permettant d'exprimer des propriétés *temporelles* sur les programmes à transformer.

Bibliographie

[1] A. Abd-allah. Extending reliability block diagrams to software architectures. Research report, Center for Software Engineering, Computer Science Department, University of Southern California, Los Angeles (CA), USA, 1997.

[2] M. Abdennebi. Synthèse de contrôleurs discrets pour systèmes embarqués tolérants aux pannes. Rapport de DEA, ESIA, Université de Savoie, Annecy, France, September 2003.

[3] I. Ahmad and Y.-K. Kwok. On exploiting task duplication in parallel program scheduling. In *IEEE Trans. on Parallel and Distributed Systems*, volume 9, pages 872–892, September 1998.

[4] I. Assayad. Heuristique d'ordonnancement fiable pour systèmes embarqués temps-réel. Rapport de DEA, EDMI, UJF, Grenoble, France, June 2003.

[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*, pages 347–356, Firenze, Italy, June 2004. IEEE.

[6] P.C. Attie, A. Arora, and E.A. Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Trans. on Programming Languages and Systems*, 26(1) :125–185, 2004.

[7] A. Avizienis. Design of fault-tolerant computers. In *Fall Joint Computer Conference*, volume 31, pages 733–743, 1967.

[8] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. on Dependable and Secure Computing*, 1(1) :11–33, January 2004.

[9] T. Ayav, P. Fradet, and A. Girault. Implementing fault-tolerance in real-time systems by automatic program transformations. In S.L. Min and W. Yi, editors, *International Conference on Embedded Software, EMSOFT'06*, pages 205–214, Seoul, South Korea, October 2006. ACM. INRIA Research report 5919.

[10] G. Baille, P. Garnier, H. Mathieu, and R. Pissard-Gibollet. Le CYCAB de l'Inria Rhône-Alpes. Rapport technique 0229, Inria, Rocquencourt, France, April 1999.

[11] M. Baleani, A. Ferrari, L. Mangeruca, M. Peri, S. Pezzini, and A. Sangiovanni-Vincentelli. Fault-tolerant platforms for automotive safety-critical applications. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES'03*, San Jose (CA), USA, November 2003. ACM.

[12] M.O. Ball. Computational complexity of network reliability analysis : An overview. *IEEE Trans. on Reliability*, 35 :230–239, August 1986.

[13] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1) :64–83, January 2003. Special issue on embedded systems.

[14] C. Bernardeschi, A. Fantechi, and L. Simoncini. Formally verifying fault tolerant system designs. *The Computer Journal*, 43(3), 2000.

[15] A.A. Bertossi, L.V. Mancini, and F. Rossini. Fault-tolerant rate-monotonic first-fit scheduling in hard real-time systems. *IEEE Trans. on Parallel and Distributed Systems*, 10 :934–945, 1999.

[16] F. Brasileiro, P. Ezhilchelvan, S. Shrivastava, N. Speirs, and S. Tao. Implementing fail-silent nodes for distributed systems. *IEEE Trans. on Computers*, 45(11) :1226–1238, November 1996.

[17] D. Brière, D. Ribot, D. Pilaud, and J.-L. Camus. Methods and specifications tools for Airbus on-board systems. In *Avionics Conference and Exhibition*, London, UK, December 1994. ERA Technology.

[18] N. Brinis. Synthèse d'un contrôleur pour le problème des généraux byzantins. Rapport de DEA, École Nationale des Sciences de l'Informatique, La Manouba, Tunisie, July 2005.

[19] J. Bruno, E.G. Coffman, and R. Sethi. Scheduling independant tasks to reduce mean finishing time. *Communication of the ACM*, 17 :382–387, 1974.

[20] G. Bruns and I. Sutherland. Model checking and fault tolerance. In *International Conference on Algebraic Methodology and Software Technology, AMAST'97*, Sidney, Australia, 1997.

[21] Y.G. Chen and M.C. Yuang. A cut-based method for terminal-pair reliability. *IEEE Trans. on Reliability*, 45 :413–416, September 1996.

[22] K.-H. Cho and J.-T. Lim. Synthesis of fault-tolerant supervisor for automated manufacturing systems : A case study on photolothographic process. *IEEE Trans. on Robotics and Automation*, 14(2) :348–351, April 1998.

[23] W.W. Chu, M.-T. Lang, and J. Hellerstein. Estimation of inter-module communication (IMC) and its applications in distributed processing systems. *IEEE Trans. on Computers*, C-33 :691–699, August 1984.

[24] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Symposium on Principles of Programming Languages*, Los Angeles (CA), USA, January 1977.

[25] F. Cristian. Understanding fault-tolerant distributed systems. *Communication of the ACM*, 34(2) :56–78, February 1991.

[26] C. Dima, A. Girault, C. Lavarenne, and Y. Sorel. Off-line real-time fault-tolerant scheduling. In *9th Euromicro Workshop on Parallel and Distributed Processing, PDP'01*, pages 410–417, Mantova, Italy, February 2001.

[27] C. Dima, A. Girault, and Y. Sorel. Static fault-tolerant scheduling with "pseudo-topological" orders. In *Joint Conference on Formal Modelling and Analysis of Timed Systems and Formal Techniques in Real-Time and Fault Tolerant System, FORMATS-FTRTFT'04*, volume 3253 of *LNCS*, Grenoble, France, September 2004. Springer-Verlag.

[28] A. Dogan and F. Özgüner. Matching and scheduling algorithms for minimizing execution time and failure probability of applications in heterogeneous computing. *IEEE Trans. on Parallel and Distributed Systems*, 13(3) :308–323, March 2002.

[29] A. Dogan and F. Özgüner. Biobjective scheduling algorithms for execution time-reliability trade-off in heterogeneous computing systems. *The Computer Journal*, 48(3) :300–314, 2005.

[30] 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.

[31] E. Dumitrescu, A. Girault, and E. Rutten. Validating fault-tolerant behaviors of synchronous system specifications by discrete controller synthesis. In *IFAC Workshop on Discrete Event Systems, WODES'04*, Reims, France, September 2004.

[32] E.A. Emerson and E.M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2 :241–266, 1982.

[33] G. Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *IEEE Real-Time Systems Symposium, RTSS'95*, pages 152–161, Pisa, Italy, 1995.

[34] G. Fohler. Adaptive fault-tolerance with statically scheduled real-time systems. In *Euromicro Workshop on Real-Time Systems, EWRTS'97*, Toledo, Spain, June 1997. IEEE.

[35] F. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1) :1–26, March 1999.

[36] F. Gärtner and A. Jhumka. Automating the addition of fail-safe fault-tolerance : Beyond fusion-closed specifications. In *Joint Conference on Formal Modelling and Analysis of Timed Systems and Formal Techniques in Real-Time and Fault Tolerant System, FORMATS-FTRTFT'04*, volume 3253 of *LNCS*, Grenoble, France, September 2004. Springer-Verlag.

[37] A. Girault, H. Kalla, M. Sighireanu, and Y. Sorel. An algorithm for automatically obtaining distributed and fault-tolerant static schedules. In *International Conference on Dependable Systems and Networks, DSN'03*, San-Francisco (CA), USA, June 2003. IEEE.

[38] 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.

[39] 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 Publishers.

[40] A. Girault, H. Kalla, and Y. Sorel. A scheduling heuristics for distributed real-time embedded systems tolerant to processor and communication media failures. *International Journal of Production Research*, 42(14) :2877–2898, July 2004.

[41] A. Girault, H. Kalla, and Y. Sorel. Transient processor/bus fault tolerance for embedded systems. In *IFIP Working Conference on Distributed and Parallel Embedded Systems, DIPES'06*, pages 135–144, Braga, Portugal, October 2006. Springer.

[42] 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, Phoenix (AZ), USA, April 2001. IEEE. Extended abstract.

[43] A. Girault and E. Rutten. Discrete controller synthesis for fault-tolerant distributed systems. In *International Workshop on Formal Methods for Industrial Critical Systems, FMICS'04*, volume 133 of *ENTCS*, pages 81–100, Linz, Austria, September 2004. Eslevier Science.

[44] A. Girault and H. Yu. A flexible method to tolerate value sensor failures. In *International Conference on Emerging Technologies and Factory Automation, ETFA'06*, pages 86–93, Prague, Czech Republic, September 2006. IEEE.

[45] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnoy Kan. Optimization and approximation in deterministic sequencing and scheduling. *Annals of Discrete Mathematics*, 5 :287–326, 1979.

[46] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *7th International Workshop on Hardware/Software Co-Design, CODES'99*, Rome, Italy, May 1999. ACM.

[47] T. Grandpierre and Y. Sorel. From algorithm and architecture specifications to automatic generation of distributed real-time executives : A seamless flow of graphs transformations. In *International Conference on Formal Methods and Models for Codesign, MEMOCODE'03*, Mont Saint-Michel, France, June 2003. IEEE.

[48] R. Gupta, S. Pande, K. Psarris, and V. Sarkar. Compilation techniques for parallel systems. *Parallel Computing*, 25(13) :1741–1783, 1999.

[49] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *International Conference on Algebraic Methodology and Software Technology, AMAST'93*, Twente, NL, June 1993. Springer-Verlag.

[50] K. Hashimoto, T. Tsuchiya, and T. Kikuno. Effective scheduling of duplicated tasks for fault-tolerance in multiprocessor systems. *IEICE Trans. on Information and Systems*, E85-D(3) :525–534, March 2002.

[51] C.-C. Hsieh and Y.-C. Hsieh. Reliability and cost optimization in distributed computing systems. *Computers and Operations Research*, 30 :1103–1119, 2003.

[52] M.A. Iverson. *Dynamic Mapping and Scheduling Algorithms for a Multi-User Heterogeneous Computing Environement*. Thèse de doctorat, Ohio State University, Columbus (OH), USA, 1999.

[53] P. Jalote. *Fault-Tolerance in Distributed Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1994.

[54] B. Jeannet. Dynamic partitioning in linear relation analysis. Application to the verification of reactive systems. *Formal Methods in System Design*, 23(1) :5–37, July 2003.

[55] P.A. Jensen and M. Bellmore. An algorithm to determine the reliability of a complex system. *IEEE Trans. on Reliability*, 18 :169–174, November 1969.

[56] R. Jensen, M. Veloso, and R. Bryant. Synthesis of fault-tolerant plans for non-deterministic domains. In *Workshop on Planning under Uncertainty and Incomplete Information*, Trento, Italy, June 2003.

[57] H. Kalla. *Génération automatique de distributions/ordonnancements temps-réel, fiables et tolérants aux fautes*. Thèse de doctorat, INPG, Grenoble, France, December 2004.

[58] O. Kamach, L. Pietrac, and E. Niel. Approche multi-modèle pour les systèmes à événements discrets : application à un préhenseur pneumatique. In *Modélisation des Systèmes Réactifs, MSR'05*, pages 159–174, Autrans, France, September 2005. Hermes.

[59] N. Kandasamy, J.P. Hayes, and B.T. Murray. Dependable communication synthesis for distributed embedded systems. In *International Conference on Computer Safety, Reliability and Security, SAFECOMP'03*, Edinburgh, UK, September 2003.

[60] B. Kao, H. Garcia-Molina, and D. Barbara. Aggressive transmissions of short messages over redundant paths. *IEEE Trans. on Parallel and Distributed Systems*, 5(1) :102–109, January 1994.

[61] K.C. Kapur and L.R. Lamberson. *Reliability in Engineering Design*, chapter 2 and 3. John Wiley & Sons, 1977.

[62] S. Kartik and C.S.R. Murthy. Improved task allocation algorithms to maximize reliability of redundant distributed computing systems. *IEEE Trans. on Reliability*, 44(4) :575–586, December 1995.

[63] S. Kartik and C.S.R. Murthy. Task allocation algorithms for maximising reliability of distributed computing systems. *IEEE Trans. on Computers*, 46(6) :719–724, June 1997.

[64] R.M. Keichafer, C.J. Walter, A.M. Finn, and P.M. Thambidurai. The MAFT architecture for distributed fault tolerance. *IEEE Trans. on Computers*, 37(4) :398–404, April 1988.

[65] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming, ECOOP'97*, volume 1241 of *LNCS*, pages 220–242, Jyväskylä, Finland, June 1997. Springer-Verlag.

[66] J.C. Knight and N.G. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Trans. on Software Engineering*, 12(1) :96–109, 1986.

[67] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1) :112–126, October 2003.

[68] H.-K. Ku and J.P. Hayes. Systematic design of fault-tolerant multiprocessors with shared buses. *IEEE Trans. on Computers*, 46(4) :439–455, April 1997.

[69] S.S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In M. Joseph, editor, *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT'00*, volume 1926 of *LNCS*, pages 82–93, Pune, India, September 2000. Springer-Verlag.

[70] S.S. Kulkarni and A. Ebnenasir. Automated synthesis of multitolerance. In *International Conference on Dependable Systems and Networks, DSN'04*, Firenze, Italy, June 2004. IEEE.

[71] R. Kumar and V.K. Garg. Optimal supervisory control of discrete event dynamic systems. *SIAM J. Control Optim.*, 33(2) :419–439, 1995.

[72] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. on Programming Languages and Systems*, 4(3) :382–401, July 1982.

[73] J.-C. Laprie. Sûreté de fonctionnement informatique : concepts de base et terminologie. Rapport technique, LAAS-CNRS, Toulouse, France, 2004.

[74] J.-C. Laprie, J. Arlat, C. Béounes, and K. Kanoun. Definition and analysis of hardware-and-software fault-tolerant architectures. *IEEE Computer*, 23(7) :39–51, 1990.

[75] J.-C. Laprie et al. *Dependability : Basic Concepts and Terminology*. Dependable Computing and Fault-Tolerant Systems. Springer-Verlag, 1992.

[76] 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*, pages 1684–1689. Hermès, July 1991.

[77] C. Lavarenne and Y. Sorel. Modèle unifié pour la conception conjointe logiciel-matériel. *Traitement du Signal*, 14(6) :569–578, 1997.

[78] N. Leignel. Ordonnancement fiable pour la génération de code temps-réel embarqué. Rapport de DEA, INPG, Grenoble, France, June 2004.

[79] T. Lévêque. Fault tolerance adequation in SynDEx. Rapport de stage, Inria Rhône-Alpes, Montbonnot, France, September 2004.

[80] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in hard real-time environnement. *Journal of the ACM*, 20(1) :46–61, January 1973.

[81] D. Lloyd and M. Lipow. *Reliability : Management, Methods, and Mathematics*, chapter 9. Prentice-Hall, 1962.

[82] G.K. Manacher. Production and stabilization of real-time task schedules. *Journal of the ACM*, 14(3) :439–465, July 1967.

[83] F. Maraninchi and Y. Rémond. Mode-automata : a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, 46(3) :219–254, 2003.

[84] H. Marchand, O. Boivineau, and S. Lafortune. On the synthesis of optimal schedulers in discrete event control problems with multiple goals. *SIAM J. Control Optim.*, 39(2) :512–532, 2000.

[85] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the Signal environment. *Discrete Event Dynamic System : Theory and Applications*, 10(4) :325–346, October 2000.

[86] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice-Hall, 1989.

[87] J. Musa, A. Iannino, and K. Okumuto. *Software Reliability : Measurement, Prediction, Application*, chapter 4.2. McGraw-Hill, 1990. Professional Edition.

[88] C. Pinello, L. Carloni, and A. Sangiovanni-Vincentelli. Fault-tolerant deployment of embedded software for cost-sensitive real-time feedback-control applications. In *Design, Automation and Test in Europe, DATE'04*, Paris, France, February 2004. IEEE.

[89] D. Powell. Failure mode assumption and assumption coverage. In *International Symposium on Fault-Tolerant Computing, FTCS-22*, pages 386–395, Boston (MA), USA, July 1992. IEEE. Research report LAAS 91462.

[90] D. Powell. Distributed fault tolerance — lessons from Delta-4. *IEEE Micro*, 14(1) :36–47, February 1994.

[91] D. Powell, J. Arlat, L. Beus-Dukic, A. Bondavalli, P. Coppola, A. Fantechi, E. Jenn, C. Rabéjac, and A. Wellings. GUARDS : a generic upgradable architecture for real-time dependable systems. *IEEE Trans. on Parallel and Distributed Systems*, 10(6) :580–599, 1999. Special Issue on Dependable Real-Time Systems.

[92] D. Powell et al. The Delta-4 approach to dependability in open distributed systems. In *International Symposium on Fault-Tolerant Computing, FTCS-18*, pages 246–251, Tokyo, Japan, June 1988. IEEE.

[93] 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 *International Conference on Parallel Processing, ICPP'02*, pages 360–386, Vancouver, Canada, August 2002. Rapport technique No. TR-UNL-CSE 2002-0501.

[94] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1) :206–230, January 1987.

[95] K. Ramamritham. Allocation and scheduling of precedence-related periodic tasks. *IEEE Trans. on Parallel and Distributed Systems*, 6(4) :412–420, April 1995.

[96] P. Richards. Timing properties of multiprocessor systems. Rapport technique TDB60-27, Tech. Operations Inc., Burlington, USA, August 1960.

[97] J. Rose, M.D. Hutton, J.P. Grossman, and D.G. Corneil. Characterization and parameterized random generation of digital circuits. In *Design Automation Conference, DAC'06*, pages 94–99, Las Vegas (NV), USA, June 1996.

[98] J. Rushby. Critical system properties : Survey and taxonomy. *Reliability Engineering and Systems Safety*, 43(2) :189–219, 1994. Research report CSL-93-01.

[99] J. Rushby. Bus architectures for safety-critical embedded systems. In *International Workshop on Embedded Systems, EMSOFT'01*, volume 2211 of *LNCS*, Tahoe City (CA), USA, October 2001. Springer-Verlag.

[100] E. Saule. Ordonnancement fiable pour la génération de code temps-réel embarqué. Rapport de DEA, INPG, Grenoble, France, June 2005.

[101] H. Schepers and J. Hooman. Trace-based compositional proof theory for fault tolerant distributed systems. *Theoretical Computer Science*, 128, 1994.

[102] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 1997.

[103] F.B. Schneider. Implementing fault-tolerant services using the state machine approach : A tutorial. *ACM Computing Surveys*, 22(4) :299–319, December 1990.

[104] S. Sekhavat and J. Hermosillo. The Cycab robot : A differentially flat system. In *IEEE Intelligent Robots and Systems, IROS'00*, Takamatsu, Japan, November 2000. IEEE.

[105] R. Sengupta and S. Lafortune. An optimal control theory for discrete event systems. *SIAM J. Control Optim.*, 36(2) :488–541, March 1998.

[106] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols : An approach to real-time synchronization. *IEEE Trans. on Computers*, 39 :1175–1185, September 1990.

[107] S.M. Shatz and J.-P. Wang. Models and algorithms for reliability-oriented task-allocation in redundant distributed-computer systems. *IEEE Trans. on Reliability*, 38(1) :16–26, April 1989.

[108] S.M. Shatz, J.-P. Wang, and M. Goto. Task allocation for maximizing reliability of distributed computer systems. *IEEE Trans. on Computers*, 41(9) :1156–1168, September 1992.

[109] G.C. Sih and E.A. Lee. A compile-time scheduling heuristic for interconnection constraint heterogeneous processor architectures. *IEEE Trans. on Parallel and Distributed Systems*, 4(2) :175–187, February 1993.

[110] Y. Sorel. Massively parallel computing systems with real time constraints, the "algorithm architecture adequation" methodology. In *Massively Parallel Computing Systems Conference*, pages 44–53, Ischia, Italy, May 1994.

[111] S. Srinivasan and N.K. Jha. Safety and reliability driven task allocation in distributed systems. *IEEE Trans. on Parallel and Distributed Systems*, 10(3) :238–251, March 1999.

[112] S. Taha. Synthèse de contrôleurs discrets pour systèmes embarqués tolérants aux pannes. Rapport de DEA, Institut National Polytechnique de Grenoble, Grenoble, France, June 2004.

[113] V. T'kindt and J.-C. Billaut. *Multicriteria Scheduling : Theory, Models and Algorithms*. Springer, 2006.

[114] E. Tronci. Optimal finite state supervisory control. In *IEEE Conference on Decision and Control, CDC'96*, Kobe, Japan, December 1996. IEEE.

[115] B.P. Upender and P.J. Koopman. Communication protocols for embedded systems. *Embedded Systems Programming*, 7 :46–58, 1994.

[116] A. Vicard. *Formalisation et Optimisation des Systèmes Informatiques Distribués Temps-Réel Embarqués*. Thèse de doctorat, Université de Paris XIII, July 1999.

[117] B. Waxman. Routing of multipoint connections. *IEEE J. on Selected Areas in Communications*, 6(9) :1617–1622, December 1988.

[118] C.B. Weinstock. SIFT : System design and implementation. In *International Fault-Tolerant Computing Symposium, FTCS-10*, pages 75–77, 1980.

[119] K.C. Wong and M.W. Wonham. On the computation of observers in discrete-event systems. *Discrete Event Dynamic System : Theory and Applications*, 14(1) :55–107, January 2004.

[120] J. Xu and D.L. Parnas. On satisfying timing constraints in hard real-time systems. In *Software for Critical Systems, SIGSOFT'91*, New-Orleans (LA), USA, December 1991. Published as ACM SIGSOFT Engineering Notes, Volume 16, Number 5.

[121] T. Yang and A. Gerasoulis. List scheduling with and without communication delays. *Parallel Computing*, 19(12) :1321–1344, 1993.

[122] V. Yodaiken. Against priority inheritance. FSM Labs Technical Report. `http://www.yodaiken.com/papers/inherit.pdf`, September 2004.

[123] H. Yu. Synthèse de contrôleurs pour la tolérance aux fautes des capteurs. Rapport de DEA, Institut National Polytechnique de Grenoble, Grenoble, France, June 2005.

Chapitre 4

Commande longitudinale de véhicules autonomes

Lors de mon séjour post-doctoral dans le projet PATH de UC Berkeley en 1997, j'ai commencé à m'intéresser aux autoroutes automatisées. J'ai travaillé sur leur compilateur SHIFT pour réseaux dynamiques de systèmes hybrides ainsi que sur une modélisation complète d'une **autoroute automatisée** avec plusieurs voies d'accélération. Les véhicules évoluant sur cette autoroute sont **autonomes**, c'est-à-dire que toute l'intelligence est concentrée dans les véhicules : ceux-ci incorporent des capteurs, calculateurs et actionneurs afin de contrôler leur vitesse, de décider les changements de voies, de conserver une distance minimale avec le véhicule devant... C'est ce qu'on appelle en anglais « Autonomous Intelligent Cruise Control » (AICC). D'une part, j'ai défini une nouvelle **loi de commande longitudinale en accélération** pour un système constitué de deux véhicules, un leader et un suiveur, et j'ai étudié sous quelles conditions cette loi de commande permet de réguler ce système (section 4.2). D'autre part, j'ai mis au point un **contrôleur hybride** (au sens continu/discret) embarqué dans tous les véhicules d'une autoroute ; j'ai démontré formellement qu'il empêchait tout accident (section 4.3) ; j'ai aussi montré par simulation qu'il permettait de diminuer la distance nécessaire à l'insertion des véhicules, d'augmenter le trafic maximum par voie, et de diminuer la congestion (section 4.4).

Mes travaux sur la commande longitudinale de véhicules autonomes ont été effectués en collaboration avec Marco Antoniotti (New-York University, USA), Akash Deshpande (Teja Technologies, USA), Sergio Yovine (VERIMAG, Grenoble), Jean-Philippe Roussel (stagiaire de l'ESSAIM, Mulhouse, que j'ai encadré) et Fethi Bouziani (étudiant du DEA EEATS, Grenoble, que j'ai encadré).

4.1 Contexte

L'autoroute que je considère est modélisée à partir de l'Interstate 10 dans la région de Houston (Texas, USA). Elle comporte une voie principale, trois jonctions d'entrées et trois jonctions de sortie :

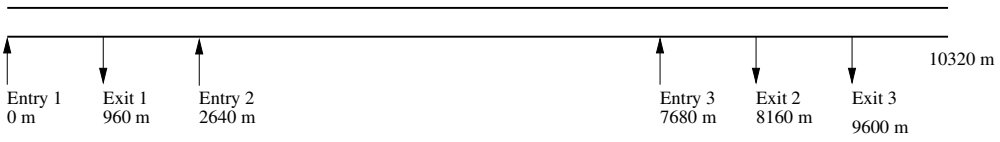


FIG. 4.1 – Plan de l'autoroute.

Les jonctions d'entrée et de sortie ont la topographie suivante :

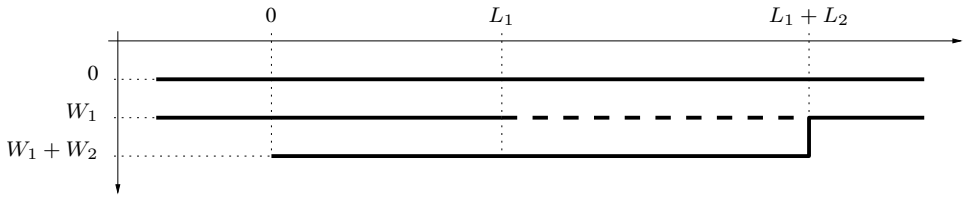


FIG. 4.2 – Topographie d'une jonction d'entrée.

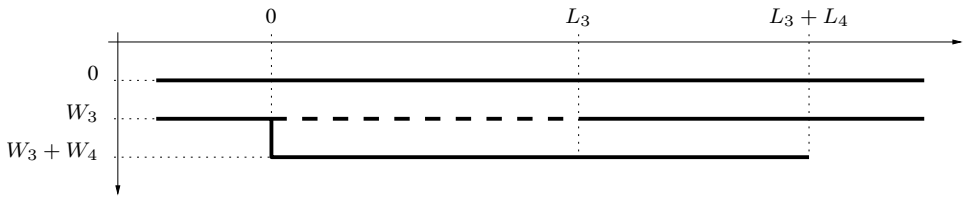


FIG. 4.3 – Topographie d'une jonction de sortie.

Les paramètres L_i et W_i sont respectivement les longueurs et les largeurs des voies. Les véhicules peuvent changer de voie uniquement à l'endroit des pointillés. Donc la distance maximale autorisée pour s'insérer dans la voie principale est L_2. C'est le paramètre crucial car il influence directement la facilité à s'insérer (plus L_2 est grand, plus c'est facile), et également car il joue un grand rôle dans le coût de construction des autoroutes (plus L_2 est grand, plus l'autoroute est encombrante et chère à construire).

Enfin les capacités d'accélération et de freinage des véhicules sont limitées par la technologie, c'est-à-dire que l'accélération est comprise dans un intervalle $[a_{min}, a_{max}]$. Je prends $a_{min} = -0.5g = -4.905\ ms^{-2}$ et $a_{max} = 0.2g = 1.962\ ms^{-2}$ (avec $g = 9.81\ ms^{-2}$). Ces chiffres sont compatibles avec la technologie actuelle pour les voitures. Ils sont en effet inférieurs à ceux cités dans [19], eux-mêmes basés sur [11] et sur des discussions personnelles avec des ingénieurs de Ford Motor Company (chiffres qui indiquent $a_{max} = 0.4g$ et $a_{min} = -0.8g$). À titre indicatif, pour s'arrêter, une voiture démarrant à $28\ ms^{-1}$ et freinant à $-4.905\ ms^{-2}$ met $79.92\ m$.

4.2 Lois de commande longitudinales

Les véhicules doivent faire face à deux situations : ou bien il y a un véhicule devant, ou bien il n'y en n'a pas (du moins pas à portée de capteur). Il faut donc deux lois de commande, une afin de suivre un véhicule (appelé le leader) en évitant les collisions (section 4.2.2), et une autre pour atteindre une vitesse désirée et s'y maintenir en l'absence de véhicule devant (section 4.2.1). Par ailleurs, des recherches ont montré que, pour les véhicules, on peut *découpler* la commande longitudinale de la commande latérale [16, 26, 17]. C'est cette approche que j'ai adoptée, en me concentrant sur la commande longitudinale.

4.2.1 Loi d'accélération pour consigne en vitesse

Je note x la **position** du véhicule, $\dot{x}$ sa **vitesse** et $\ddot{x}$ son **accélération**. La vitesse désirée est v_{max} ; elle est finie et peut dépendre de la voie et de la portion dans laquelle il se trouve. Je néglige la dynamique interne du véhicule et je choisis de le commander par son accélération. Une solution qui permettrait de tenir compte de la dynamique interne du véhicule serait de concevoir une commande à deux niveaux, un premier niveau pour calculer l'accélération du véhicule, et un second niveau pour en déduire la

commande à appliquer aux pédales de frein et d'accélérateur [27]. Bref, dans mon cas, la commande est :

$$\ddot{x} = u(x, \dot{x}, v_{max}) \tag{4.1}$$

Je choisis une loi de commande proportionnelle classique :

$$\ddot{x}(t) = u(t) = \mu(v_{max} - \dot{x}) \tag{4.2}$$

où le gain μ est de dimension s^{-1}. Il en résulte :

$$\dot{x}(t) = v_{max} + (v_0 - v_{max})e^{-\mu t} \tag{4.3}$$

La loi de commande $u(t)$ garantit que le véhicule atteint la vitesse désirée v_{max}. Si l'accélération du véhicule doit rester dans les limites imposées par la technologie $[-4.905\ ms^{-2}, 1.962\ ms^{-2}]$, alors j'applique la commande saturée suivante :

$$\begin{aligned} u(t) &= \mu(v_{max} - \dot{x}(t)) \\ \ddot{x}(t) = u_{sat}(t) &= \text{if } u(t) > 1.962\ ms^{-2} \text{ then } 1.962\ ms^{-2} \\ &\quad \text{else if } u(t) < -4.905\ ms^{-2} \text{ then } -4.905\ ms^{-2} \\ &\quad \text{else } u(t) \end{aligned} \tag{4.4}$$

Puisque la vitesse désirée v_{max} est finie, la loi de commande saturée $u_{sat}(t)$ garantit aussi que le véhicule va atteindre v_{max}.

4.2.2 Loi d'accélération pour suivi du leader

4.2.2.1 Variables d'état du système

Je considère à présent un système constitué d'un **véhicule leader** (position z, vitesse $\dot{z}$, accélération $\ddot{z}$) et d'un **véhicule suiveur** (position x, vitesse $\dot{x}$, accélération $\ddot{x}$), conduisant sur une voie rectiligne d'autoroute. Comme dans la section précédente, je néglige la dynamique interne et je choisis de commander le véhicule suiveur par son accélération :

$$\ddot{x} = u(x, \dot{x}, z, \dot{z}) \tag{4.5}$$

Je suppose que l'accélération du leader $\ddot{z}$ est **continue par morceaux**. Je suppose également que les deux véhicules se déplacent **en avant** et ne sont jamais **immobiles**, que le suiveur est toujours **derrière** le leader, et que la vitesse du leader est **bornée** par v_{max}. Enfin, l'accélération du leader doit rester dans les limites de la technologie $[-4.905\ ms^{-2}, 1.962\ ms^{-2}]$. En revanche, je ne suppose rien sur les saccades $\dddot{z}$ et $\dddot{x}$ (« jerk » en anglais).

4.2.2.2 Politiques de commande

Trois politiques existent pour réaliser une telle loi de commande, dépendant de la politique d'espacement entre le suiveur et le leader [32] :

1. Maintenir un **espacement constant** est particulièrement bien adapté aux distances courtes, par exemple quand les véhicules sont groupés en train virtuels (« platoons »). Toutefois, le leader doit communiquer aux autres véhicules son accélération afin d'éviter les problèmes d'instabilité.
2. Maintenir un **temps inter-véhicules constant** revient à maintenir un espacement proportionnel à la vitesse. Cela ne nécessite que des capteurs de distance et de vitesse.

3. Maintenir un **facteur de sécurité constant** permet de garantir que l'arrêt instantané du leader n'entraîne pas une collision avec le suiveur. Cela revient à maintenir un espacement proportionnel au carré de la vitesse.

L'espacement constant est trop restrictif à cause de la communication inter-véhicules. Le facteur de sécurité constant est trop pénalisant pour le trafic maximum par voie. Aussi, c'est le temps inter-véhicules constant qui est le mieux adapté à mon système.

4.2.2.3 Commande non linéaire

Je commence par calculer le **rapport du temps inter-véhicules** (l'espace inter-véhicules divisé par la vitesse du suiveur) **par le temps désiré h** (en secondes). Ce rapport $r(t)$ est sans dimension :

$$r(t) = \frac{z(t) - x(t)}{h\dot{x}(t)} \tag{4.6}$$

Le rapport $r(t)$ est dans l'intervalle $[0, +\infty[$. Il rend compte du positionnement entre le suiveur et le leader. Entre 0 et 1 le suiveur est **trop proche** et doit freiner, au delà de 1 le suiveur est **trop loin** et doit accélérer, à 1 le suiveur est **bien positionné** puisqu'il est exactement à h secondes du leader. Afin de garder le temps inter-véhicules égal à h, il faut réguler $r(t)$ à 1. Je choisis donc comme sortie $y(t) = r(t)$, avec 1 comme consigne. La dynamique de la sortie est :

$$\begin{aligned}
\dot{y}(t) \;=\; \dot{r}(t) \;&=\; \frac{h\dot{x}(t)(\dot{z}(t) - \dot{x}(t)) - h\ddot{x}(t)(z(t) - x(t))}{h^2\dot{x}^2(t)} \\
&=\; \frac{\dot{z}(t) - \dot{x}(t)}{h\dot{x}(t)} - \frac{\ddot{x}(t)}{\dot{x}(t)} \cdot \frac{z(t) - x(t)}{h\dot{x}(t)} \\
&=\; \frac{\dot{z}(t) - \dot{x}(t)}{h\dot{x}(t)} - \frac{\ddot{x}(t)}{\dot{x}(t)} \cdot r(t) \\
&=\; \frac{\dot{z}(t) - \dot{x}(t) - h\ddot{x}(t)r(t)}{h\dot{x}(t)}
\end{aligned} \tag{4.7}$$

Le **point d'équilibre** est atteint lorsque $\dot{y}(t) = \dot{r}(t) = 0$ et $y(t) = r(t) = 1$, et la commande est alors $\hat{u}(t)$. D'après l'équation (4.7), cela implique $\dot{z}(t) - \dot{x}(t) - hu_{sat}(t) = 0$, et donc :

$$u_{sat}(t) = \frac{\dot{z}(t) - \dot{x}(t)}{h} \tag{4.8}$$

Enfin, j'ajoute un **terme de rétro-action** (« feedback ») :

$$\begin{aligned}
\ddot{x}(t) = u(t) \;&=\; u_{sat}(t) + \lambda(r(t) - 1) \\
&=\; \frac{\dot{z}(t) - \dot{x}(t)}{h} + \lambda(r(t) - 1)
\end{aligned} \tag{4.9}$$

où le gain λ est de dimension ms^{-2}. S'il existe t_0 tel que $r(t_0) = 1$, alors la commande de l'équation (4.9) est la même que celle de l'équation (4.8), qui maintient $r(t)$ égal à 1 par construction.

4.2.3 Régulation autour du point d'équilibre

Afin d'étudier la stabilité de mon système autour du point d'équilibre, je peux calculer une équation différentielle sur $r(t)$ et l'intégrer. J'obtiens après quelques calculs :

$$\begin{aligned}
&\lambda + \lambda r(t)^2 - 2\lambda r(t) - 2h\dot{r}(t)^2\dot{z}(t) + 2\lambda h^2\dot{r}(t)^2 + \ddot{z}(t) \\
&- 3\lambda h^2 r(t)\dot{r}(t)^2 + 3\lambda h\dot{r}(t) - 5\lambda h r(t)\dot{r}(t) + h\dot{r}(t)\ddot{z}(t) \\
&+ 2\lambda h r(t)^2\dot{r}(t) + \lambda h^2 r(t)^2\ddot{r}(t) - \dot{r}(t)\dot{z}(t) - 2r(t)\ddot{z}(t) \\
&+ r(t)^2\ddot{z}(t) + hr(t)\ddot{r}(t)\dot{z}(t) - h\ddot{r}(t)\dot{z}(t) - \lambda h^2 r(t)\ddot{r}(t) \\
&+ r(t)\dot{r}(t)\dot{z}(t) - hr(t)\dot{r}(t)\ddot{z}(t) \;=\; 0
\end{aligned}$$

Cette équation différentielle ne semble pas être intégrable. Mon idée est par conséquent d'étudier la fonction $r(t)$ analytiquement, et de démontrer qu'elle converge vers 1 quand $t \to +\infty$.

4.2.3.1 Étude des variations de $r(t)$

Sous mes hypothèses (section 4.2.2.1), $r(t)$ est continue et dérivable, et sa dérivée première est aussi continue :

$$\begin{aligned}
\dot{r}(t) &= \frac{\dot{z}(t) - \dot{x}(t) - h\ddot{x}(t)r(t)}{h\dot{x}(t)} \\
&= \frac{\dot{z}(t) - \dot{x}(t) - \lambda h(r(t)-1) - r(t)(\dot{z}(t) - \dot{x}(t))}{h\dot{x}(t)} \\
&= \frac{(\dot{z}(t) - \dot{x}(t))(1 - r(t)) + \lambda h r(t)(1 - r(t))}{h\dot{x}(t)} \\
&= \frac{(1 - r(t))(\dot{z}(t) - \dot{x}(t) + \lambda h r(t))}{h\dot{x}(t)} \\
&= \frac{(1 - r(t))(\ddot{x}(t) + \lambda)}{\dot{x}(t)} \qquad (4.10)
\end{aligned}$$

J'étudie les variations de $r(t)$. Puisque $\dot{x}$ est positive, j'obtiens :

$$\text{Si } \forall t,\ r(t) < 1,\ \text{alors } \forall t,\ \ddot{x}(t) > -\lambda \Leftrightarrow \dot{r}(t) > 0. \qquad (4.11)$$

$$\text{Si } \forall t,\ r(t) > 1,\ \text{alors } \forall t,\ \ddot{x}(t) > -\lambda \Leftrightarrow \dot{r}(t) < 0. \qquad (4.12)$$

Ceci établit la proposition suivante sur les variations de $r(t)$:

Proposition 4.1 *Si $\forall t,\ \ddot{x}(t) > -\lambda$, alors :*
- *$\forall t,\ r(t) \in [0, 1)$ et r est strictement croissante, ou*
- *$\forall t,\ r(t) = 1$, ou*
- *$\forall t,\ r(t) \in (1, +\infty)$ et r est strictement décroissante.*

Ce résultat est important car, à condition que $\ddot{x}(t) > -\lambda$, il garantit que $r(t)$ évolue dans le bon sens : si le suiveur est trop proche, c'est-à-dire $r(t) < 1$, alors $r(t)$ augmente, alors que si le suiveur est trop loin, c'est-à-dire $r(t) > 1$, alors $r(t)$ décroît. De plus, $r(t)$ n'oscille jamais autour de 1.

4.2.3.2 Étude de la limite de $r(t)$

Proposition 4.2 *Si $\exists t_0$ tel que $\forall t \geq t_0,\ \ddot{x}(t) > -\lambda$, alors $\lim_{t \to +\infty} r(t) = 1$.*

Preuve. C'est une conséquence directe de la proposition 4.1. Dans le cas où $r(t) \in [0, 1[$, puisque $r(t)$ est décroissant et minoré, il a une limite l. Puisque $\lim_{t \to +\infty} r(t)$ est finie, c'est une limite asymptotique, et donc $\lim_{t \to +\infty} \dot{r}(t) = 0^+$. Il est alors facile de montrer que $l = 1$. La preuve est similaire quand $r(t) \in]1, +\infty[$. □

Ce résultat est important car, à condition que $\ddot{x}(t) > -\lambda$, le suiveur n'entre jamais en collision avec le leader (ce qui se produirait si $r(t)$ devenait négatif).

4.2.3.3 Étude de la condition $\ddot{x}(t) > -\lambda$

Je suppose dans tout ce paragraphe que $r(t) \in [0, 1[$. Le cas où $r(t) \in]1, +\infty[$ est analogue. Les propositions suivantes établissent que la **condition invariante** sur l'accélération du suiveur $\ddot{x} > -\lambda$ est satisfaite si la **condition invariante** sur l'accélération du leader $\ddot{z} > -\lambda$ est satisfaite, et si la **condition initiale** sur l'accélération du suiveur $\ddot{x}(t_0) > -\lambda$ est satisfaite pour un certain t_0.

Proposition 4.3 *Si $\exists t_0$ tel que $\ddot{x}(t_0) > -\lambda$ et $\forall t \geq t_0,\ \ddot{z}(t) > -\lambda$, alors $\forall t \geq t_0,\ \ddot{x}(t) > -\lambda$.*

Preuve. Sans perte de généralité, je prends $t_0 = 0$. Mes hypothèses sont $\ddot{x}(0) > -\lambda, \forall t,\ \ddot{z}(t) > -\lambda$, et $\forall t,\ r(t) < 1$. Je distingue deux cas exclusifs, selon que $\ddot{x}(t) \geq \ddot{z}(t)$ ou $\ddot{x}(t) < \ddot{z}(t)$ sur un intervalle $[0, \delta]$ où $\ddot{z}(t)$ est continue :

1. Supposons que $\forall t \in [0, \delta],\ \ddot{x}(t) \geq \ddot{z}(t)$. Par hypothèse, $\forall t \in [0, \delta]$, $\ddot{z}(t) > -\lambda$. Donc $\forall t \in [0, \delta]$, $\ddot{x}(t) > -\lambda$.
2. Supposons que $\forall t \in [0, \delta],\ \ddot{x}(t) < \ddot{z}(t)$. Prouvons que $\dot{r}(t) > 0$ sur $[0, \delta]$. Je rappelle l'équation (4.10) :
$$\dot{r}(t) = \frac{(1 - r(t))(\ddot{x}(t) + \lambda)}{\dot{x}(t)}$$
Par hypothèse, $1 - r(0) > 0$ et $\ddot{x}(0) + \lambda > 0$. Puisque $\dot{x}(0) > 0$, alors $\dot{r}(0) > 0$. Supposons que $\dot{r}$ ne soit pas strictement positive sur $[0, \delta]$: alors $\exists \varepsilon > 0$ tel que $\dot{r}(\varepsilon) = 0$ et $\forall t \in [0, \varepsilon),\ \dot{r}(t) > 0$. Maintenant, $\dot{r}(\varepsilon) = 0$ implique $\ddot{x}(\varepsilon) = -\lambda$. D'après l'équation (4.9), $\dddot{x}$ existe :
$$\dddot{x}(t) = \lambda \dot{r}(t) + \frac{\dddot{z}(t) - \dddot{x}(t)}{h} \tag{4.13}$$
Par hypothèse, $\forall t \in [0, \delta],\ \dddot{z}(t) - \dddot{x}(t) \geq 0$, donc $\forall t \in [0, \varepsilon[,\ \dddot{x}(t) > 0$. Donc $\ddot{x}$ est strictement croissante $[0, \varepsilon[$. Puisque $\ddot{x}$ est continue, $\ddot{x}$ est strictement croissante sur $[0, \varepsilon]$ et $\ddot{x}(\varepsilon) > \ddot{x}(0)$. Mais par hypothèse, $\ddot{x}(0) > -\lambda$, donc $\ddot{x}(\varepsilon) > -\lambda$. D'où il vient $\ddot{x}(\varepsilon) = -\lambda > -\lambda$ qui est une contradiction. Par conséquent, il n'existe pas $\varepsilon > 0$ tel que $\dot{r}(\varepsilon) = 0$ et $\forall t \in [0, \varepsilon),\ \dot{r}(t) > 0$. Je conclus que $\forall t \in [0, \delta],\ \dot{r}(t) > 0$, et donc, $\forall t \in [0, \delta],\ \ddot{x}(t) > -\lambda$. □

La conséquence de la proposition 4.3 est que, pour éviter les collisions, il faut que le gain λ de la loi de commande (4.9) soit supérieur à l'opposé de la borne inférieure de l'accélération du leader. Autrement dit, $\lambda > -a_{min}$.

En combinant les propositions 4.1, 4.2 et 4.3, j'obtiens le théorème suivant :

Théorème 4.1 (régulation par la commande (4.9)) *Si $\exists t_0$ tel que $\ddot{x}(t_0) > -\lambda$ et $\forall t \geq t_0,\ \ddot{z}(t) > -\lambda$, alors l'une de ces trois propositions est vraie :*

- *$\forall t \geq t_0,\ r(t) \in [0, 1)$, r est strictement croissante, et $\lim_{t \to +\infty} r(t) = 1$, ou*
- *$\forall t \geq t_0,\ r(t) = 1$, ou*
- *$\forall t \geq t_0,\ r(t) \in (1, +\infty)$, r est strictement décroissante, et $\lim_{t \to +\infty} r(t) = 1$.*

Ceci démontre que le véhicule suiveur n'entre jamais en collision avec le véhicule leader, à condition que $\ddot{x}(0) > -\lambda$ et que pour tout t, $\ddot{z}(t) > -\lambda$. La commande donnée par l'équation (4.9) permet donc de réguler le système pour que sa sortie, $y(t) = r(t)$, converge asymptotiquement vers sa consigne qui est 1.

Concernant les variables d'état du système, $x(t)$ et $\dot{x}(t)$, puisqu'on est dans une situation de poursuite, elles peuvent diverger. D'ailleurs si le véhicule leader roule à vitesse constante non nulle, alors $\lim_{t \to +\infty} z(t) = +\infty$ et je peux démontrer que $\lim_{t \to +\infty} x(t) = +\infty$ aussi. En effet, en se plaçant au point d'équilibre $r(t) = 1$, il vient :
$$z(t) - x(t) = h\dot{x}(t)$$

Supposons de plus que le véhicule leader roule à la vitesse constante v_{const}. On a alors $z(t) = z(0) + v_{const}t$ et il en découle :

$$h\dot{x}(t) + x(t) = z(0) + v_{const}t$$

C'est une équation différentielle linéaire du premier ordre. Les solutions de l'équation homogène associée (où le second membre est remplacé par 0) sont de la forme (k étant une constante positive) :

$$x(t) = ke^{-t/h}$$

Tandis qu'une solution particulière de l'équation différentielle est :

$$x(t) = z(0) + v_{const}(t - h)$$

Donc les solutions générales sont de la forme (k étant une constante positive) :

$$x(t) = z(0) + v_{const}(t - h) + ke^{-t/h}$$

Quand t tend vers $+\infty$, le terme $ke^{-t/h}$ tend vers 0 et on retrouve la solution particulière. Comme on s'y attendait, le véhicule suiveur roule à la même vitesse que le véhicule leader ($\dot{x}(t) = v_{const}$), tout en maintenant un écart égal à hv_{const}, c'est-à-dire h secondes. De plus, $\lim_{t\to+\infty} z(t) = \lim_{t\to+\infty} x(t) = +\infty$.

4.2.4 Étude des bornes

Je traite à présent le problème posé par la saturation due aux limites technologiques. L'accélération du suiveur est donc définie par la commande u_{sat} donnée dans l'équation suivante :

$$\begin{aligned} u(t) &= \frac{\dot{z}(t) - \dot{x}(t)}{h} + \lambda\left(\frac{z(t) - x(t)}{h\dot{x}(t)} - 1\right) \\ \ddot{x}(t) = u_{sat}(t) &= \text{if } u(t) \geq 1.962\ ms^{-2} \text{ then } 1.962\ ms^{-2} \\ &\quad \text{else if } u(t) \leq -4.905\ ms^{-2} \text{ then } -4.905\ ms^{-2} \\ &\quad \text{else } u(t) \end{aligned} \tag{4.14}$$

Je montre d'abord que, sous certaines conditions initiales, l'accélération du suiveur est bornée ; puis je généralise ce résultat à une file quelconque de véhicules. Par ailleurs, en collaboration avec Jean-Philippe Roussel [29], cette loi de commande a été implantée dans le logiciel ORCCAD [1][33] et testée avec succès sur les véhicules électriques CYCAB [5, 30].

4.2.4.1 Bornes sur $\ddot{x}(t)$

Théorème 4.2 (régulation par la commande (4.14)) *Si* $\exists t_0$ *tel que* $\forall t \geq t_0,\ -\lambda < a_{min} \leq \ddot{z}(t) \leq a_{max}$ *et* $a_{min} \leq \ddot{x}(t_0) \leq a_{max}$ *et* $a_{min} \leq \frac{\dot{z}(t_0)-\dot{x}(t_0)}{h} \leq a_{max}$*, alors l'une de ces trois propositions est vraie :*

- $\forall t \geq t_0,\ r(t) \in [0, 1)$*, r est strictement croissante, et* $\lim_{t\to+\infty} r(t) = 1$*, ou*
- $\forall t \geq t_0,\ r(t) = 1$*, ou*
- $\forall t \geq t_0,\ r(t) \in (1, +\infty)$*, r est strictement décroissante, et* $\lim_{t\to+\infty} r(t) = 1$.

De plus $\forall t \geq t_0,\ a_{min} \leq \ddot{x}(t) \leq a_{max}$.

[1] ORCCAD = « Open Robot Controller Computer Aided Design ». `http://www.inrialpes.fr/sed/Orccad`.

Preuve. Les variations et la limite de $r(t)$ sont une conséquence du théorème 4.1. Concernant les bornes de $\ddot{x}(t)$, je me limite au cas où $r(t_0) < 1$. Les cas $r(t_0) = 1$ et $r(t_0) > 1$ sont analogues. Sans perte de généralité, je prends $t_0 = 0$:

$$r(0) < 1 \Longrightarrow r(0) - 1 < 0 \Longrightarrow \ddot{x}(0) < \frac{\dot{z}(0) - \dot{x}(0)}{h}$$

Mes hypothèses sont $a_{min} \leq \ddot{x}(0) < \frac{\dot{z}(0)-\dot{x}(0)}{h} \leq a_{max}$, et $\forall t,\ -\lambda < a_{min} \leq \ddot{z}(t) \leq a_{max}$. Je distingue deux cas exclusifs, selon que $\ddot{x}(t) \geq \ddot{z}(t)$ ou $\ddot{x}(t) < \ddot{z}(t)$ sur un intervalle $[0, \delta]$ où $\ddot{z}(t)$ est continue :

1. Supposons que $\forall t \in [0, \delta]$, $\ddot{x}(t) \geq \ddot{z}(t)$:
   - Par hypothèse, $\forall t \in [0, \delta]$, $\ddot{z}(t) \geq a_{min}$. Donc $\forall t \in [0, \delta]$, $\ddot{x}(t) \geq a_{min}$.
   - $\forall t \in [0, \delta], \ddot{x}(t) \geq \ddot{z}(t) \Longrightarrow \forall t \in [0, \delta], \ddot{z}(t) - \ddot{x}(t) \leq 0 \Longrightarrow \dot{z}(t) - \dot{x}(t)$ décroît sur $[0, \delta]$. Aussi $\forall t \in [0, \delta]$, $\frac{\dot{z}(t)-\dot{x}(t)}{h} \leq \frac{\dot{z}(0)-\dot{x}(0)}{h}$ qui est par hypothèse plus petit que a_{max}. De plus, $r(t) < 1 \Longrightarrow \lambda(r(t) - 1) < 0$, donc :

     $$\forall t \in [0, \delta],\ \lambda(r(t) - 1) + \frac{\dot{z}(t) - \dot{x}(t)}{h} \leq a_{max}$$

     Par conséquent $\forall t \in [0, \delta], \ddot{x}(t) \leq a_{max}$.
2. Supposons que $\forall t \in [0, \delta]$, $\ddot{x}(t) < \ddot{z}(t)$:
   - Par hypothèse, $\forall t \in [0, \delta]$, $\ddot{z}(t) \leq a_{max}$. Donc $\forall t \in [0, \delta]$, $\ddot{x}(t) \leq a_{max}$.
   - D'après le théorème 4.1, $\forall t \in [0, \delta], \dot{r}(t) > 0$. Par hypothèse $\forall t \in [0, \delta], \ddot{z}(t) - \ddot{x}(t) > 0$. Donc, d'après l'équation (4.13), $\forall t \in [0, \delta], \dddot{x}(t) > 0$, ce qui implique que $\ddot{x}(t)$ décroît strictement sur $[0, \delta]$. Aussi, $\forall t \in [0, \delta], \ddot{x}(t) \geq \ddot{x}(0)$ qui est par hypothèse plus grand que a_{min}. Donc $\forall t \in [0, \delta], \ddot{x}(t) \geq a_{min}$.

   Dans les deux cas, $\forall t \in [0, \delta], \ddot{x}(t) \in [a_{min}, a_{max}]$. □

4.2.4.2 Stabilité d'une file

Je généralise maintenant mes résultats à une **file de n véhicules**. Le véhicule i suit le véhicule $i - 1$ et ses variables d'état sont x_i, $\dot{x}_i$ et $\ddot{x}_i$. Par définition, $r_i = \frac{x_{i-1}-x_i}{h\dot{x}_i}$. Je définis trois prédicats sur les véhicules :

- $\textsc{Inv}_i(t_0) \triangleq\ \forall t \geq t_0,\ -\lambda < a_{min} \leq \ddot{x}_i(t) \leq a_{max}$
- $\textsc{InitAcc}_i(t_0) \triangleq\ a_{min} \leq \ddot{x}_i(t_0) \leq a_{max}$
- $\textsc{InitVel}_i(t_0) \triangleq\ a_{min} \leq \frac{\dot{x}_{i-1}(t_0) - \dot{x}_i(t_0)}{h} \leq a_{max}$

Théorème 4.3 (stabilité de file) *Si* $\exists t_0$ *tel que* $\textsc{Inv}_1(t_0)$ *et* $\bigwedge_{i=2}^{n}\big(\textsc{InitAcc}_i(t_0) \wedge \textsc{InitVel}_i(t_0)\big)$*, alors* $\bigwedge_{i=1}^{n} \textsc{Inv}_i(t_0)$.

Preuve. D'après le théorème 4.2, pour tout i j'ai : $\textsc{Inv}_i(t_0) \wedge \textsc{InitAcc}_{i+1}(t_0) \wedge \textsc{InitVel}_{i+1}(t_0) \Longrightarrow \textsc{Inv}_{i+1}(t_0)$. Donc, par induction sur i, j'ai : $\bigwedge_{i=1}^{n} \textsc{Inv}_i(t_0)$. □

4.2.5 Automates hybrides

Avant de présenter mes simulations, cette section introduit la notion d'automate hybride. Depuis une dizaine d'années, de nombreuses recherches ont été menées sur les systèmes hybrides continu/discret. On peut par exemple se référer au récent numéro spécial [3] ou à la série de conférences LNCS « Hybrid Systems Computation and Control ». Je présente ici brièvement un modèle utilisé couramment pour spécifier les systèmes hybrides, à savoir les automates hybrides.

Un **automate hybride** est un automate d'états fini avec un ensemble de variables continues. Le système passe du temps dans les états, permettant ainsi aux variables continues d'évoluer, et effectue des transitions discrètes entre les états. La dynamique associée aux variables continues est typiquement spécifiée sous la forme d'un ensemble d'**équations différentielles**. Chaque transition discrète est étiquetée par une garde et une action. Une **garde** est une contrainte (égalité ou inégalité) sur les variables continues, et une transition ne peut être prise que si sa garde est vraie. Quand cela se produit, les variables continues sont mises à jour en accord avec l'**action** de la transition. De plus, un automate hybride peut avoir des **invariants** associés à ses états discrets, typiquement spécifiés sous la forme de contraintes sur ses variables continues. Enfin, l'**état hybride** d'un tel automate est constitué à la fois de son état discret courant et de la valeur courante de ses variables continues.

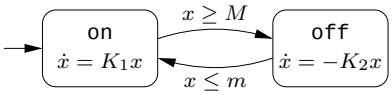


FIG. 4.4 – Un exemple d'automate hybride.

La figure 4.4 est un exemple archi-classique d'un automate hybride décrivant le comportement d'un thermostat. Il a deux états discrets, `on` et `off`, et une variable continue x, la température de l'eau. Dans l'état `on`, la chaudière fonctionne donc la température augmente, alors que dans l'état `off`, la température diminue. Les transitions entre les états `on` et `off` sont discrètes ; elles dépendent de si la température atteint le seuil bas m ou le seuil haut M. Enfin, K_1 et K_2 sont deux constantes.

Il existe plusieurs sémantiques qui gouvernent la façon dont sont tirées les transitions discrètes. Puisque les gardes des transitions et les invariants des états sont des *inégalités*, l'instant auquel une transition peut être tirée est dans un *intervalle* de temps. Une transition peut être :

- **urgente** (« eager » en anglais) : elle doit être prise dès que sa garde le lui permet ;
- **tardive** (« late ») : elle doit être prise au dernier instant auquel l'invariant de son état source est encore vrai ;
- ou **retardable** (« delayable ») : elle peut être prise à n'importe quel instant dans son intervalle valide.

Enfin, de nombreuses variantes des automates hybrides ont été définies, selon le type de dynamique continue. Par exemple, dans les **automates temporisés**, les équations différentielles sont toutes de la forme $\dot{x} = 1$. Dans les **automates hybrides linéaires**, elles sont toutes de la forme forme $\dot{x} = k$ où k est une constante. Et dans les **automates hybrides rectangulaires**, elles sont toutes de la forme $\dot{x} \in [a, b]$, où a et b sont deux constantes.

4.2.6 Simulations

Voici à présent plusieurs simulations de ma loi de commande en accélération pour suivi du leader (commande 4.14). Elles illustrent l'importance des conditions initiales. J'ai utilisé le simulateur SHIFT de l'Université de Berkeley [9] [2], au développement duquel j'ai d'ailleurs contribué. SHIFT utilise un intégrateur Runge-Kutta à pas variable. Toutes les simulations ont été faites avec $h = 0.6\ s$, et le comportement du leader est donné par l'automate hybride de la figure 4.5. Les transitions discrètes sont notées sous la forme « garde/action ». Ce comportement est parmi les pires possibles : le leader alterne les freinages et les accélérations, rendant difficile, a priori, l'ajustement de la position du suiveur.

[2] SHIFT : `http://www-shift.eecs.berkeley.edu`

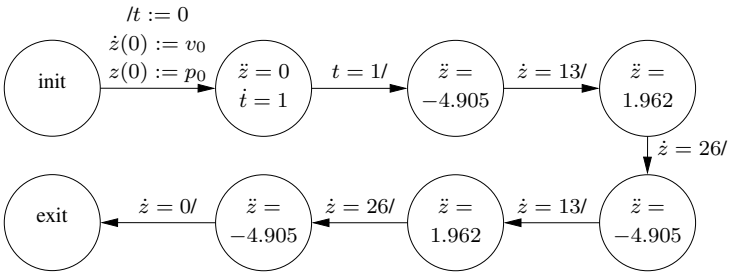


FIG. 4.5 – Automate hybride du comportement du leader.

4.2.6.1 Cas non saturé

L'accélération du suiveur est donnée par l'équation (4.9). Les conditions initiales sont $z(0) = 10\,m$, $\dot{z}(0) = 22\ ms^{-1}$, $x(0) = 0\ m$ et $\dot{x}(0) = 26\ ms^{-1}$. Donc $r(0) = \frac{10-0}{0.6\times 26} = 0.641$, c'est-à-dire que le suiveur est trop proche.

λ	3	5	7	12	17
$\ddot{x}(0)$	-7.744	-8.462	-9.180	-10.975	-12.770

Pour $\lambda \in \{3, 5, 7\}$, $\ddot{x}(0) \not> -\lambda$, donc $r(t)$ décroît initialement. Pour $\lambda \in \{12, 17\}$, $\ddot{x}(0) > -\lambda$ et toutes les hypothèses du théorème 4.1 sont satisfaites, donc $r(t)$ est strictement décroissante sur $[0, +\infty[$ et sa limite est 1 :

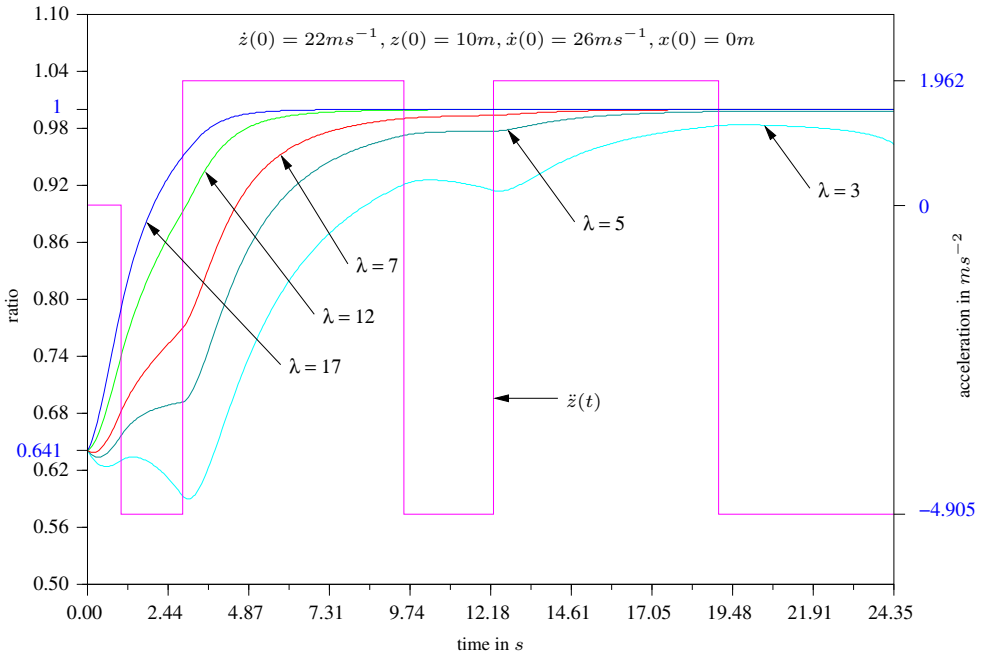


FIG. 4.6 – Cas non saturé : $r(t) < 1$.

Pour $\lambda = 3$, le sens de variation de $r(t)$ change plusieurs fois. D'après l'équation (4.10), $\dot{r}(t)$ change de signe quand $\ddot{x}(t) = -\lambda$. Aussi, chaque fois que $\ddot{x}(t)$ croise la ligne horizontale $-\lambda$, il y a un maximum ou un minimum de $r(t)$ ainsi qu'un changement de signe de $\dot{r}(t)$.

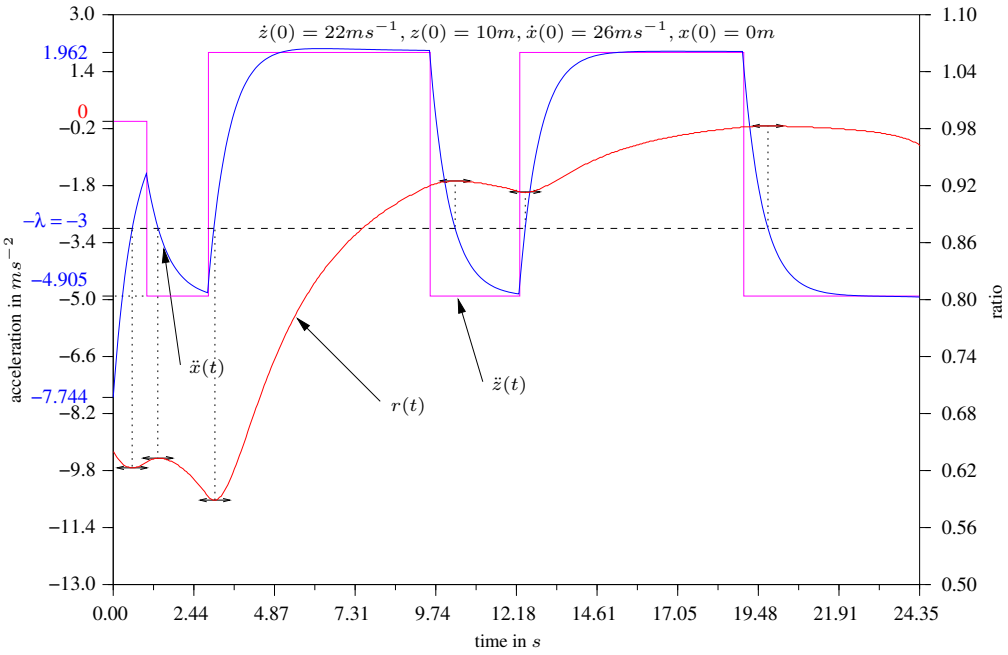


FIG. 4.7 – Maxima et minima de $r(t)$ pour $\lambda = 3$.

4.2.6.2 Cas saturé

J'étudie à présent l'influence des saturations, c'est-à-dire $\ddot{x}(t) \in [-4.905, 1.962]$. L'accélération du suiveur est donnée par l'équation (4.14). Je me concentre sur le cas $r(t) < 1$ qui est plus critique vis à vis des accidents. La figure 4.8 montre l'accélération du suiveur avec les mêmes conditions initiales que pour la figure 4.6 :

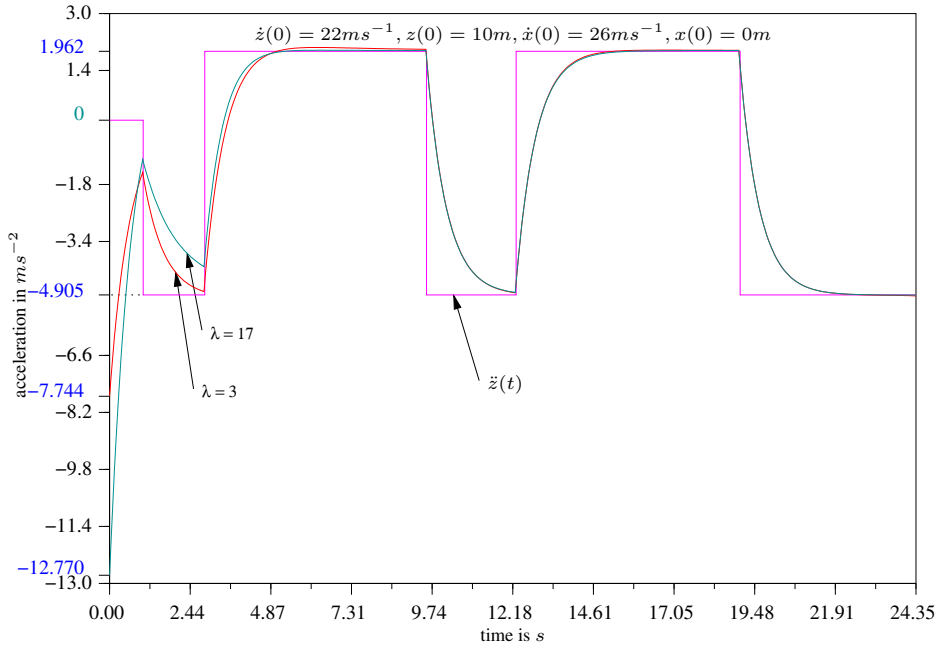


FIG. 4.8 – Accélération du suiveur pour $\lambda \in \{3, 17\}$.

Pour $\lambda = 17$ et $\lambda = 3$, l'accélération est bien au delà des limites technologiques $[-4.905, 1.962]$! La figure 4.9 montre les cas pour λ allant de 3 à 30 :

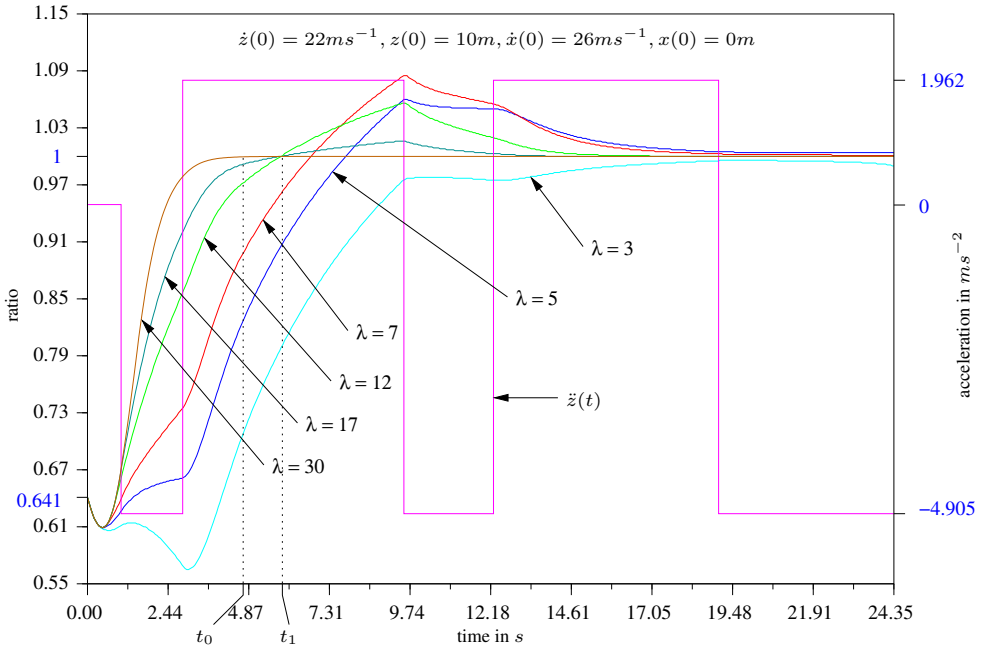


FIG. 4.9 – Cas saturé : $r(t) < 1$.

Les valeurs initiales de l'accélération sont indiquées dans la table suivante :

λ	3	7	12	17	30
$\ddot{x}(0)$	-7.744	-9.180	-10.975	-12.770	-17.437

Dans tous les cas, $\ddot{x}(0) \not\geq a_{min}$, donc la condition invariante du théorème 4.2 n'est pas vérifiée, ce qui explique pourquoi $r(t)$ décroît initialement (figure 4.9). Pour $\lambda \in \{5, 7, 12, 17\}$, $r(t)$ devient supérieur à 1, alors que pour $\lambda = 30$ ça n'est pas le cas :

- $\lambda = 30$: À $t_0 = 4.80\ s$, on a $r(t_0) \simeq 1$ et $\frac{\dot{z}(t_0)-\dot{x}(t_0)}{h} \simeq 1.827\ ms^{-2} \leq 1.962\ ms^{-2} = a_{max}$. Les conditions initiales du théorème 4.2 sont vérifiées à t_0 et le système peut être régulé autour de son point d'équilibre $r(t) = 1$. De plus, $\forall t \geq t_0, u_{sat}(t) = u(t)$.
- $\lambda = 17$: À $t_1 = 5.85\ s$, on a $r(t_1) \simeq 1$ et $\frac{\dot{z}(t_1)-\dot{x}(t_1)}{h} \simeq 2.072\ ms^{-2} \not\leq 1.962\ ms^{-2} = a_{max}$. Donc les conditions initiales du théorème 4.2 ne sont pas satisfaites à t_1. Si la loi de commande était $\ddot{x}(t) = u(t)$, alors le système pourrait être régulé et $r(t)$ resterait dans $[0, 1[$, mais alors $\ddot{x}(t)$ serait en dehors de $[-4.905, 1.962]$. Puisque j'applique la loi de commande $\ddot{x}(t) = u_{sat}(t)$, $r(t)$ devient supérieur à 1.

Par conséquent, quand $u_{sat}(t) \neq u(t)$, le système n'est pas garanti de pouvoir être régulé. Cela dépend des conditions initiales.

4.3 Contrôleur hybride

Maintenant que je dispose de deux lois de commande longitudinale pour les véhicules, je présente le contrôleur qui équipe tous les véhicules circulant sur l'autoroute. Ce contrôleur est un automate hybride (voir la section 4.2.5). Ses variables continues sont principalement la position longitudinale et latérale du véhicule (c'est un modèle uniquement 2D). Quant à ses transitions discrètes, elles sont toutes urgentes.

Je rappelle que la commande longitudinale est en accélération, alors que la commande latérale est en vitesse. Je note γ_v la loi d'accélération pour consigne en vitesse donnée par l'équation (4.4), et γ_f la loi d'accélération pour suivi du leader donnée par l'équation (4.14). Pour mémoire :

$$
\begin{aligned}
u_v(t) &= \mu(v_{max} - \dot{x}(t)) \\
\gamma_v(t) &= \text{if } u_v(t) > 1.962\ ms^{-2} \text{ then } 1.962\ ms^{-2} \\
&\quad \text{else if } u_v(t) < -4.905\ ms^{-2} \text{ then } -4.905\ ms^{-2} \\
&\quad \text{else } u_v(t)
\end{aligned}
$$

$$
\begin{aligned}
u_f(t) &= \frac{\dot{z}(t) - \dot{x}(t)}{h} + \lambda \left(\frac{z(t) - x(t)}{h\dot{x}(t)} - 1 \right) \\
\gamma_f(t) &= \text{if } u_f(t) \geq 1.962\ ms^{-2} \text{ then } 1.962\ ms^{-2} \\
&\quad \text{else if } u_f(t) \leq -4.905\ ms^{-2} \text{ then } -4.905\ ms^{-2} \\
&\quad \text{else } u_f(t)
\end{aligned}
$$

4.3.1 Capteurs

Chaque véhicule est équipé de **capteurs** dont la **portée** est limitée et égale à d_{max}. Ils permettent de connaître à tout instant la distance le séparant des véhicules voisins et les vitesses relatives.

Quand un véhicule roule sur une portion à une seule voie, les notations utilisées sont données dans la figure 4.10. Le véhicule concerné y est représenté par un rectangle épais alors que ses voisins le sont par des rectangles fins.

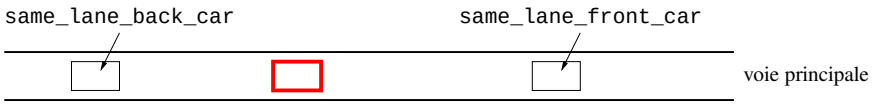


FIG. 4.10 – Véhicules voisins dans une portion à une voie.

De même, quand un véhicule roule sur la voie principale d'une portion à deux voies, les notations utilisées sont données dans la figure 4.11. Quand il roule sur la voie latérale, la figure est symétrique.

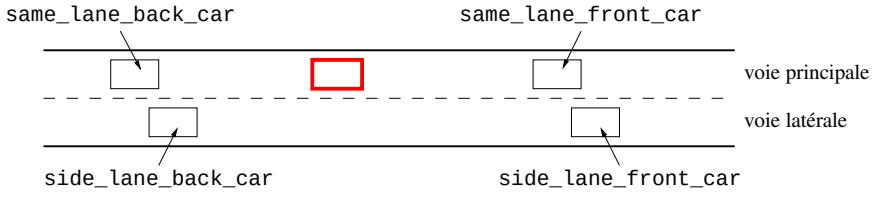


FIG. 4.11 – Véhicules voisins dans une portion à deux voies.

Concernant l'implémentation en SHIFT, chaque véhicule maintient des **pointeurs** vers les véhicules voisins. Ces pointeurs sont automatiquement mis à jour grâce à du code SHIFT qui surveille tous les véhicules dans la même portion d'autoroute. À chaque fois que `same_lane_front_car` est trop loin (ou quand il n'y a aucun véhicule devant), le pointeur `same_lane_front_car` est égal à `nil`.

La position longitudinale du véhicule est notée x, sa vitesse longitudinale $\dot{x}$, et son accélération $\ddot{x}$. Chaque fois que `same_lane_front_car` existe, sa position longitudinale est notée z_1, sa vitesse $\dot{z_1}$, et son accélération $\ddot{z_1}$ (resp. $z_2/\dot{z_2}/\ddot{z_2}$ pour `side_lane_front_car`, $z_3/\dot{z_3}/\ddot{z_3}$ pour

`same_lane_back_car` et $z_4/\dot{z}_4/\ddot{z}_4$ pour `side_lane_back_car`). Enfin, sa position latérale est notée y et sa vitesse latérale $\dot{y}$.

4.3.2 Principe de fonctionnement

Tous les véhicules démarrent au point d'entrée de la voie latérale de l'une des trois jonctions d'entrée, avec une vitesse initiale égale à v_{init}, et terminent leur parcours à la fin de la voie latérale de la jonction de sortie qui leur a été affectée. Ainsi, chaque véhicule passe par les sept phases suivantes : `accelerate`, `merge`, `drop-out`, `cruise`, `yield`, `exit` et `end`, et éventuellement par la phase `collision`. Chaque phase est représentée par un état de l'automate du contrôleur hybride. Dans chaque état, il faut établir les équations régissant son accélération longitudinale et sa vitesse latérale. Il faut de plus soigneusement établir les gardes des transitions, car ce sont elles qui vont permettre de garantir que les conditions nécessaires des théorèmes de stabilité sont satisfaites.

4.3.3 Phases du contrôleur hybride

Tout véhicule démarre dans la voie d'entrée de sa jonction d'entrée. Or à l'instant du démarrage, il peut y avoir déjà un véhicule devant dans la voie d'entrée et à portée de capteur (tout simplement le véhicule qui a démarré juste avant). Afin d'assurer qu'il n'y aura pas de collision, je vérifie que les conditions du théorème 4.3 sont bien satisfaites. La garde de la transition initiale vers `accelerate` est donc $\text{INITVEL}_{i-1}(t_d) \wedge \text{INITACC}_{i-1}(t_d)$, où i est le véhicule courant, $i-1$ est son `same_lane_front_car` et t_d est l'instant de démarrage. Les huit phases sont les suivantes :

1. La première phase est `accelerate`, pendant laquelle le véhicule est dans la voie latérale mais ne peut pas encore s'insérer (c'est-à-dire $x < L_1$) ; il démarre avec une vitesse v_{init} et tente d'atteindre v_{max}, sans entrer en collision avec un éventuel `same_lane_front_car`. Si `same_lane_front_car` $\neq$ `nil`, alors $\ddot{x} = \min(\gamma_v(\dot{x}, v_{max}), \gamma_f(\dot{x}, z_1 - x, \dot{z}_1))$, sinon $\ddot{x} = \gamma_v(\dot{x}, v_{max})$. Sa vitesse latérale est $\dot{y} = 0\ ms^{-1}$. Quand $x = L_1$, il va dans la phase `merge`.
2. La phase `merge` est celle qui implémente la stratégie d'insertion. La stratégie usuelle consiste à viser un trou entre deux véhicules successifs de la voie principale, à s'aligner sur ce trou et à s'insérer dedans [10, 23, 6]. La stratégie que je présente ici est différente et basée sur la **coopération des véhicules de la voie principale**. Puisque toute communication est interdite, la coopération est implémentée par le contrôleur même de chaque véhicule. La phase `merge` est divisée en deux sous-phases : `align-to-gap` et `go-to-main` :

   (a) Dans la sous-phase `align-to-gap`, si `same_lane_front_car` et `side_lane_front_car` sont définis, alors $\ddot{x} = \min(\gamma_v(\dot{x}, v_{max}), \gamma_f(\dot{x}, z_1 - x, \dot{z}_1), \gamma_f(\dot{x}, z_2 - x, \dot{z}_2))$. Si `same_lane_front_car` et/ou `side_lane_front_car` sont `nil`, alors le terme correspondant du $\min$ disparaît. Sa vitesse latérale est $\dot{y} = -1\ ms^{-1}$. Ainsi, le véhicule se comporte exactement comme si `side_lane_front_car` était dans sa propre voie. De même, `side_lane_back_car`, s'il est défini, doit **céder le passage** au véhicule, c'est-à-dire se comporter comme si son `side_lane_front_car` à lui, qui est le véhicule dont on calcule actuellement l'accélération, était dans sa propre voie (voir la phase `yield` ci-dessous). Cette double action de céder le passage va automatiquement augmenter la longueur du trou entre `side_lane_back_car` et `side_lane_front_car`, permettant ainsi au véhicule de s'insérer entre eux deux !

   Si la différence de vitesse entre les véhicules de la voie latérale et la voie principale est trop grande, ou si, quand le véhicule pénètre dans la portion $[L_1, L_1 + L_2]$, son `side_lane_front_car` est trop proche (c'est-à-dire $z_2 - x$ est trop petit), alors le véhicule va freiner le plus fort possible afin de rester derrière. Mais comme

l'accélération est bornée, $z_2 - x$ peut devenir négatif. Dans ce cas, le trou est raté et le véhicule va maintenant prendre comme nouveau `side_lane_front_car` le `same_lane_front_car` de son précédent `side_lane_front_car`, s'il existe, et comme nouveau `side_lane_back_car` son précédent `side_lane_front_car`. Les véhicules voisins vont mettre à jour leurs pointeurs de façon similaire. Ainsi, si un trou est raté, alors le véhicule essayera automatiquement de s'aligner sur le trou suivant.

Quand le véhicule est prêt à s'insérer, il va dans la sous-phase `go-to-main`. Pour déterminer quand cela est possible, on vérifie à tout moment les conditions initiales du théorème 4.3, où i est le véhicule courant, $i-1$ est son `side_lane_front_car` et $i+1$ est son `side_lane_back_car` (cf la figure 4.12).

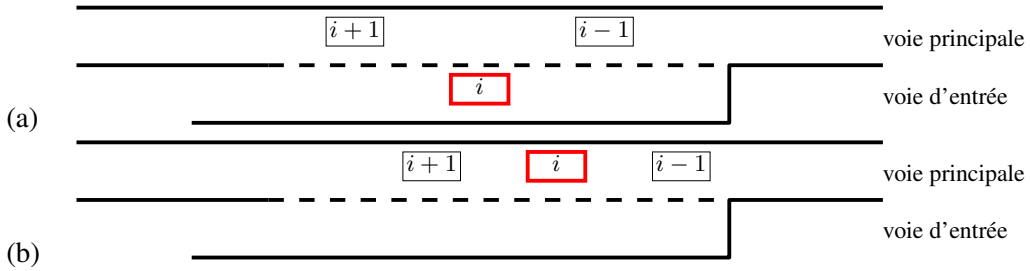


FIG. 4.12 – (a) Avant l'insertion du véhicule i ; (b) Après l'insertion.

Si les conditions ne sont jamais satisfaites jusqu'à ce que $x = L_1 + L_2$, alors l'insertion est impossible : le véhicule est alors rejeté et va dans la phase `drop-out`. Sinon, soit t_m l'instant auquel les conditions sont satisfaites. $\text{INV}_{i-1}(t_m)$ et $\text{INV}_i(t_m)$ sont satisfaites parce que les accélérations sont bornées par les limites technologiques. La garde de la transition de `align-to-gap` à `go-to-main` est donc $\text{INITVEL}_i(t_m) \wedge \text{INITACC}_i(t_m) \wedge \text{INITVEL}_{i+1}(t_m) \wedge \text{INITACC}_{i+1}(t_m)$.

$\text{INV}_{i-1}(t_m) \wedge \text{INITVEL}_i(t_m) \wedge \text{INITACC}_i(t_m)$ garantit que le véhicule i n'entre pas en collision avec le véhicule $i-1$, tandis que $\text{INV}_i(t_m) \wedge \text{INITVEL}_{i+1}(t_m) \wedge \text{INITACC}_{i+1}(t_m)$ garantit que le véhicule $i+1$ n'entre pas en collision avec le véhicule i. Puisque ce sont les collisions qui me préoccupent, je n'utilise que la partie $\geq a_{min}$ des inégalités. En effet, $\ddot{x} \not\leq a_{max}$ signifie que le véhicule sur le point de s'insérer est trop loin du véhicule de devant pour pouvoir le rattraper, ce qui n'est pas un problème pour la sécurité.

(b) La sous-phase `go-to-main` est divisée en deux parties. La première partie est utilisée tant que le véhicule est dans la voie latérale, alors que la seconde partie est utilisée à partir du moment où le véhicule franchit la ligne entre les deux voies et entre donc dans la voie principale. Dans la voie latérale, si `same_lane_front_car` et `side_lane_front_car` sont définis, alors $\ddot{x} = \min(\gamma_v(\dot{x}, v_{max}), \gamma_f(\dot{x}, z_1 - x, \dot{z_1}), \gamma_f(\dot{x}, z_2 - x, \dot{z_2}))$. Si `same_lane_front_car` et/ou `side_lane_front_car` sont `nil`, alors le terme correspondant du min disparaît. Par ailleurs, sa vitesse latérale est $\dot{y} = -1\ ms^{-1}$.

Dès que le véhicule franchit la ligne entre la voie latérale et la voie principale (c'est-à-dire $y = W_1$), ses pointeurs `side_lane_front_car` et `same_lane_front_car` sont échangés. Les pointeurs vers les véhicules voisins sont également mis à jour. Dans la voie principale, c'est pareil sauf que les pointeurs sont inversés. Jusqu'à ce le véhicule atteigne le milieu de la voie principale (c'est-à-dire $y = W_1/2$), sa vitesse latérale est $\dot{y} = -1\ ms^{-1}$; à cet instant, il va dans la phase `cruise`.

Si le véhicule est incapable d'aller dans la voie principale avant $L_1 + L_2$, alors il est rejeté et va dans la phase `drop-out`.

3. Dans la phase `drop-out`, le véhicule quitte l'autoroute parce qu'il a été incapable de s'insérer sur la voie principale avant $L_1 + L_2$; donc il est enlevé de la simulation et le compteur `nb_miss` est incrémenté.

4. Dans la phase `cruise`, le véhicule roule dans la voie principale sans aucun véhicule à portée de capteur dans la voie latérale (quand elle existe). Il doit rouler à la vitesse v_{max} si possible, en évitant les collisions avec son éventuel `same_lane_front_car`. Donc $\ddot{x} = \min(\gamma_v(\dot{x}, v_{max}), \gamma_f(\dot{x}, z_1 - x, \dot{z_1}))$. Sa vitesse latérale est $\dot{y} = 0\, ms^{-1}$. Si `same_lane_front_car` = `nil`, alors le terme γ_f du $\min$ disparaît. Quand il pénètre dans la portion $[L_1, L_1 + L_2]$ d'une jonction d'entrée, s'il y a un véhicule à portée de capteur dans la voie latérale, alors il va dans la phase `yield` et ce véhicule devient son `side_lane_front_car`. Dans le cas contraire, il n'y a aucun véhicule à qui céder le passage, donc il reste dans la phase `cruise`. Quand le véhicule pénètre dans la portion $[0, L_3]$ de sa jonction de sortie, il va dans la phase `exit`.

5. Dans la phase `yield`, le véhicule roule dans la portion $[L_1, L_1 + L_2]$ de la voie principale, *et* il y a un véhicule dans la voie latérale devant lui et à portée de ses capteurs ; il doit rouler à la vitesse v_{max} si possible, sans entrer en collision avec un éventuel `same_lane_front_car` ni avec le `side_lane_front_car` qui va tenter de s'insérer dans la voie principale. Donc $\ddot{x} = \min(\gamma_v(\dot{x}, v_{max}), \gamma_f(\dot{x}, z_1 - x, \dot{z_1}), \gamma_f(\dot{x}, z_2 - x, \dot{z_2}))$. Sa vitesse latérale est $\dot{y} = 0\, ms^{-1}$. Si `side_lane_front_car` et/ou `same_lane_front_car` sont `nil`, alors le terme correspondant du $\min$ disparaît. Avec une telle accélération, le véhicule se comporte exactement comme si son `side_lane_front_car` était dans la même voie que lui. Mes résultats de micro-simulation (voir la section 4.4) montrent que cette coopération entre les véhicules de la voie principale et de la voie d'entrée permet à tous les véhicules de s'insérer, qui plus est sans risque de collision. Quand $x = L_1 + L_2$ ou quand `side_lane_front_car` est `nil`, le véhicule va dans la phase `cruise`. Le pointeur `side_lane_front_car` peut devenir `nil` pour plusieurs raisons : soit le véhicule est hors de portée de capteur, ou il s'est inséré dans la voie principale, ou il a été dépassé.

6. Dans la phase `exit`, le véhicule quitte la voie principale pour aller dans la voie latérale de sa jonction de sortie (c'est-à-dire $x < L_3$) ; il doit rouler à v_{max} si possible, sans entrer en collision avec ses éventuels `same_lane_front_car` ou `side_lane_front_car`. Sa vitesse latérale est $1\ ms^{-1}$. Cette phase est traitée exactement comme la phase `merge`, en considérant que le véhicule doit s'insérer dans la voie latérale, sauf qu'il n'y a pas de `side_lane_back_car`. La phase `exit` est divisée en deux sous-phases : `prepare-exit` et `go-to-exit`. Je ne les détaille pas autant car elles sont très similaires à `align-to-gap` et `go-to-main`. Les seules différences sont d'une part que la garde pour la transition de `prepare-exit` et `go-to-exit` ne porte que sur le véhicule et son éventuel `side_lane_front_car`, et d'autre part que dans la vitesse latérale est $\dot{y} = 1\, ms^{-1}$. Dès que le véhicule franchit la ligne entre la voie principale et la voie latérale (c'est-à-dire $y = W_3$), ses pointeurs `side_lane_front_car` et `same_lane_front_car` sont échangés. Les pointeurs vers les véhicules voisins sont également mis à jour. Dans la voie latérale, on se s'occupe plus du `side_lane_front_car`, donc l'accélération est $\ddot{x} = \min(\gamma_v(\dot{x}, v_{max}), \gamma_f(\dot{x}, z_1 - x, \dot{z_1}))$. Jusqu´à ce que le véhicule atteigne le milieu de la voie latérale (c'est-à-dire $y = W_3 + W_4/2$), sa vitesse latérale est $\dot{y} = 1\ ms^{-1}$; à cet instant, il va dans la phase `end`.

7. Dans la phase `end`, le véhicule quitte l'autoroute (c'est-à-dire $L_3 < x < L_3 + L_4$) ; il doit rouler à v_{max} si possible, sans entrer en collision avec son éventuel `same_lane_front_car`. Sa vitesse latérale est $\dot{y} = 0\, ms^{-1}$. Si `same_lane_front_car` $\neq$ `nil`, alors $\ddot{x} = \min(\gamma_v(\dot{x}, v_{max}), \gamma_f(\dot{x}, z_1 - x, \dot{z_1}))$, sinon $\ddot{x} = \gamma_v(\dot{x}, v_{max})$. Quand $x = L_3 + L_4$, le véhicule est enlevé de la simulation.

8. Le véhicule va dans la phase `collision` dès que $z_1 - x$ devient négatif (collision avec l'arrière du `same_lane_front_car`) ou $x - z_3$ devient négatif (collision avec l'avant du `same_lane_back_car`). Il doit alors freiner le plus possible et atteindre le côté de l'autoroute : donc $\ddot{x} = -4.905\ ms^{-2}$ et $\dot{y} = 2\ ms^{-1}$. De plus, je force les deux véhicules à rouler à la même vitesse. Donc quand $z_1 - x < 0$, j'affecte à $\dot{x}$ la valeur de $\dot{z_1}$ au moment de la collision. Au moment où le véhicule atteint le côté de l'autoroute (c'est-à-dire $y = W_1$ ou W_3 dans une portion à une voie, $y = W_1 + W_2$ ou $W_3 + W_4$ dans une portion à deux voies), il est enlevé de la simulation. J'incrémente ici le compteur `nb_coll`.

La figure 4.13 montre l'automate hybride du contrôleur que je viens de décrire. Je l'ai implémenté dans le langage SHIFT : 13.000 lignes de code SHIFT dont 3.000 lignes pour le seul contrôleur hybride, plus 2.000 lignes de code C pour les fonctions auxiliaires (toutes celles qui sont plus faciles à programmer dans un langage de style impératif que dans un langage de style fonctionnel).

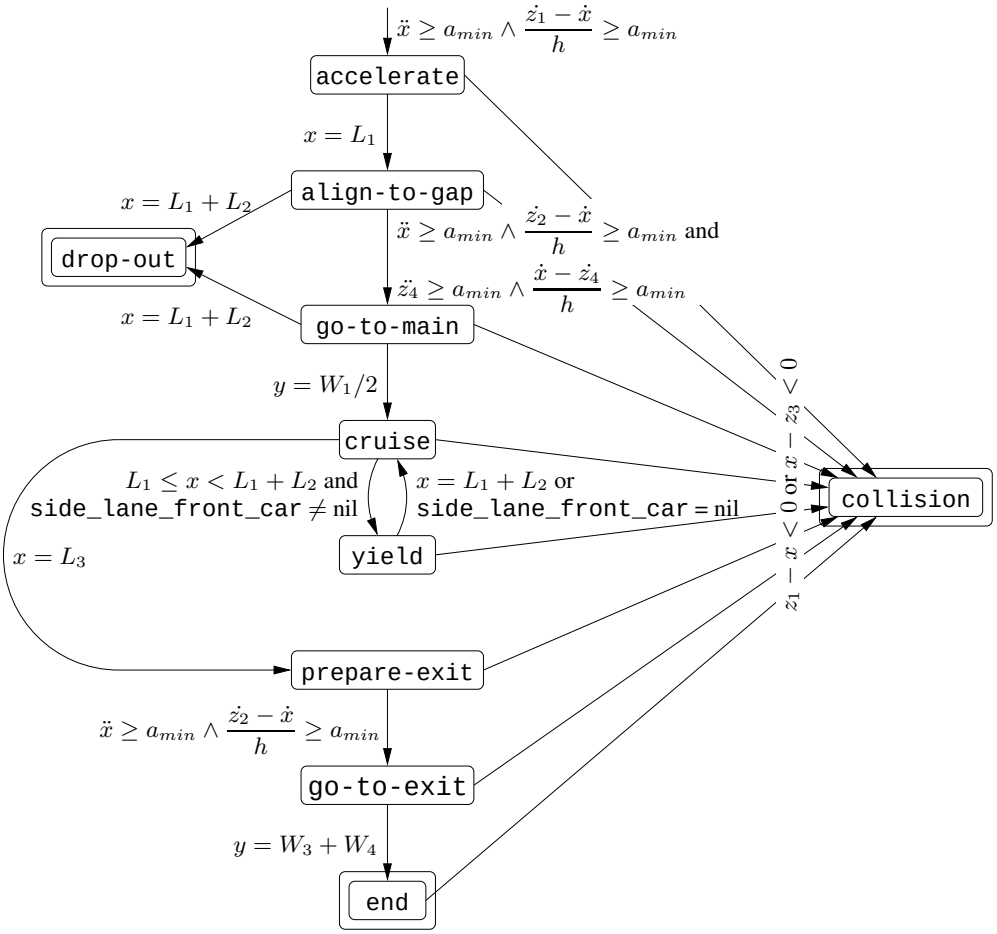


FIG. 4.13 – Automate hybride du contrôleur.

4.3.4 Preuve de l'absence de collisions sur l'autoroute

Théorème 4.4 (stabilité de l'autoroute) *Si tous les véhicules roulant sur l'autoroute de la figure 4.1 sont contrôlés par le contrôleur hybride de la figure 4.13, alors il n'y a aucune collision.*

Preuve. L'autoroute de la figure 4.1 comporte une voie principale, trois voies d'entrée et trois voies de sortie. Tous les véhicules sont créés à l'une des trois voies d'entrée. Chaque véhicule i pénètre dans sa voie d'entrée à l'instant t_i quand son contrôleur prend la transition initiale vers l'état `accelerate`. Cette transition est gardée par le prédicat « $\text{INITVEL}_i(t_i)$ and $\text{INITACC}_i(t_i)$ ». Quand le véhicule 1 est créé, il n'y a aucun véhicule devant lui, donc son accélération est γ_v qui est bornée par les limites technologiques : $\text{INV}_1(t_1)$ est donc vrai. Par conséquent, d'après le théorème 4.3, $\text{INV}_i(t_i)$ est vrai $\forall\, i \in [1, n]$. Il en résulte que, d'après le théorème 4.2, $\forall\, i \in [2, n+1]$, il n'y a pas de collision entre le véhicule i et le véhicule $i-1$ son leader. Ceci garantit qu'il n'y a aucune collision sur les trois voies d'entrée.

Puis, chaque véhicule j pénètre dans la voie principale à l'instant t_j quand son contrôleur prend la transition de `align-to-gap` vers `go-to-main`. Cette transition est gardée par le prédicat « $\text{INITVEL}_j(t'_j)$ and $\text{INITACC}_j(t'_j)$ ». À la première jonction d'entrée, comme il n'y a aucun véhicule sur la voie principale, les véhicules s'insèrent librement. Tout se passe comme s'il n'y avait qu'une seule file de véhicules, donc le théorème 4.2 s'applique directement. Aux deux autres jonctions d'entrée, je distingue deux étapes : dans la première, il n'y a encore aucun véhicule dans la voie principale et le raisonnement est le même que pour la première jonction ; dans la seconde il y a des véhicules dans la voie principale. Dès que le véhicule j est dans la phase `align-to-gap`, les véhicules dans la voie principale (ils sont dans la phase `yield`) considèrent j comme étant également dans la voie principale pour le calcul de leur accélération, même si les collisions sont virtuelles à ce stade. Puis, dès que le véhicule j est dans la phase `go-to-main`, d'après le théorème 4.3, $\text{INV}_j(t'_j)$ est vrai $\forall\, j \in [1, n]$. Donc, d'après le théorème 4.2, il vient que $\forall\, j \in [2, n+1]$ il n'y a pas de collision entre le véhicule j et son leader $j-1$. Ceci démontre qu'il n'y a pas de collisions dans la voie principale.

Quelle que soit la jonction de sortie, un véhicule sortant k pénètre dans la voie de sortie à l'instant t_k quand il prend la transition de `prepare-exit` vers `go-to-exit`. Cette transition est gardée par le prédicat « $\text{INITVEL}_k(t'_k)$ and $\text{INITACC}_k(t'_k)$ ». Puisqu'il n'y a aucun véhicule dans la voie de sortie avant la jonction, tout se passe comme s'il n'y avait qu'une seule file de véhicules, donc le théorème 4.2 s'applique directement.

En définitive, il n'y a aucune collision sur la totalité de l'autoroute. □

4.3.5 Stratégies d'insertion alternatives

Plusieurs stratégies d'insertion peuvent être implémentées dans ce contrôleur hybride. Le point important est que, tant que les gardes sur les transitions sont maintenues et que les lois de commandes en accélération sont conservées, le théorème 4.4 continue à s'appliquer. Les critères permettant de comparer différentes stratégies d'insertion sont, triés par ordre décroissant d'importance : l'**absence de collision**, l'**absence de rejet de véhicules**, et la **congestion du trafic**.

Ainsi, avec Fethi Bouziani, j'ai mis au point une nouvelle méthode d'insertion provoquant encore moins de perturbation dans la voie principale [6]. Elle consiste à diviser la portion où les véhicules peuvent s'insérer (l'intervalle $[L_1, L_1 + L_2]$ de la figure 4.2) en deux régions : une région $[L_1, L_1 + L]$ dans laquelle les véhicules de la voie d'accélération doivent chercher un **trou convenable** entre deux véhicules successifs dans la voie principale et s'aligner dessus, mais sans avoir le droit de s'insérer, suivie d'une région $[L_1 + L, L_1 + L_2]$ dans laquelle les véhicules sont autorisés à s'insérer, si les conditions de sécurité sont satisfaites. Nous avons donc ajouté une phase `reach-point` dans le contrôleur hybride de la figure 4.13, entre les états `accelerate` et `align-to-gap` :

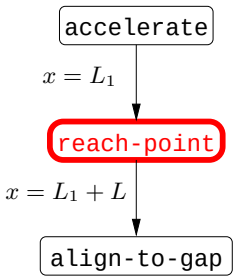


FIG. 4.14 – Le nouvel état `reach-point`.

Le principe est que, dans la phase `reach-point`, le véhicule examine tous les trous inter-véhicules dans la portion $[L_1, L_1 + L]$ de la voie principale entre son `same_lane_back_car` et son `same_lane_front_car`, en choisit un, et s'aligne dessus, de façon à être en situation idéale au moment d'effectuer sa transition vers la phase `align-to-gap`.

En accord avec ce principe, afin de diminuer les perturbations dans la voie principale, la transition de `cruise` à `yield` est désormais gardée par :

$$L_1 + L_2 > x \geq L_1 + L \text{ and } \texttt{same\_lane\_front\_car} \neq \text{nil}$$

La figure 4.15 illustre un exemple de détection des trous : trois trous ont été identifiés, G_1, G_2 et G_3.

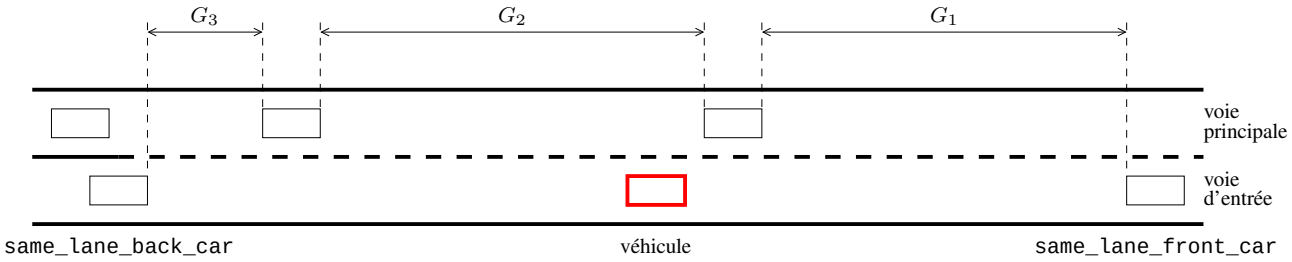


FIG. 4.15 – Exemple de détection des trous.

Pour chaque trou G_i, je note b_i le véhicule délimitant l'arrière du trou, f_i celui délimitant l'avant du trou, et l_i la longueur du trou. Je détermine alors le **nombre théorique de véhicules** qui pourraient tenir dedans, à la vitesse du véhicule b_i, et en respectant le temps inter-véhicules h. Ce nombre m_i est calculé par la formule suivante :

$$m_i = \frac{l_i - hv_{b_i}}{l + hv_{b_i}} \tag{4.15}$$

où la distance inter-véhicule est hv_{b_i} et la longueur du véhicule est l. Je suppose ici que les véhicules ont tous la même longueur l. Si le véhicule b_i n'existe pas, alors je prends v_{f_i} à la place de v_{b_i}. Ensuite, pour chaque trou G_i, si $\lfloor m_i \rfloor \geq 1$, alors je crée $\lfloor m_i \rfloor$ **points virtuels**, séparés chacun de hv_{b_i} mètres. Si au contraire $\lfloor m_i \rfloor < 1$, alors je crée un seul point virtuel situé au milieu du trou. Chaque point virtuel a une longueur égale à l, la même que les véhicules. La figure 4.16 illustre ce processus.

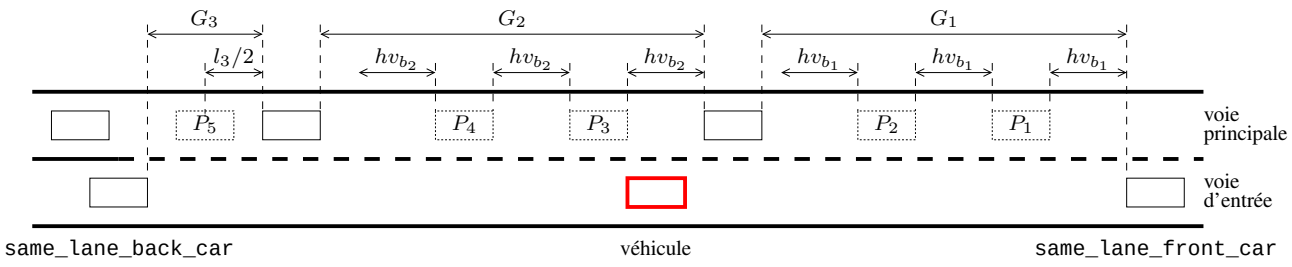


FIG. 4.16 – Exemple de création des points virtuels dans les trous.

En même temps, à chaque point virtuel P_j de chaque trou G_i, j'attribue une **note $\mathbf{n_j}$** égale à 1 si $m_i \geq 1$ et à m_i sinon. Cette note me permettra de **classer** les points atteignables afin de choisir celui sur lequel le véhicule s'alignera.

De plus, je calcule la distance d_j nécessaire au véhicule qui veut s'insérer pour s'aligner sur le point virtuel P_j ; ce calcul suppose que les véhicules dans la voie principale roulent à une vitesse constante et que le véhicule suit une loi d'accélération longitudinale de type « bang-bang ». Cette distance me permet de supprimer de la liste les points virtuels non atteignables : ce sont tous ceux tels que $x + d_i > L_1 + L$, c'est-à-dire tels qu'une fois aligné dessus, le véhicule ne sera plus dans la portion de l'autoroute où il a le droit de s'insérer. Par exemple, dans la figure 4.17, le point virtuel P_1 est éliminé pour cette raison.

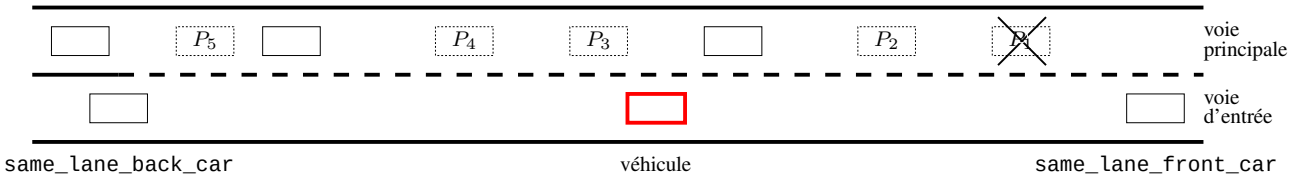


FIG. 4.17 – Exemple d'élimination d'un point virtuel non atteignable.

Puis, les points restants sont ordonnés selon leur note n_j, et seuls ceux qui maximisent cette note sont conservés. Si plusieurs points P_j sont tels que $n_j = 1$ alors ceux-la sont conservés et les autres sont éliminés. C'est ce qui se passe dans la figure 4.18 où les notes n_2, n_3 et n_4 sont égales à 1 alors que n_5 est inférieure à 1, et donc le point P_5 est éliminé. Si aucun point virtuel n'est dans ce cas, alors c'est que tous ont une note inférieure à 1, et seul celui qui maximise la note n_j est conservé.

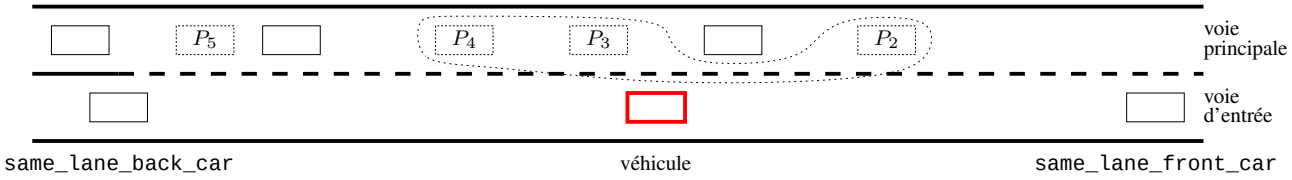


FIG. 4.18 – Exemple de sélection des points virtuels qui maximisent la note n_j.

En dernier lieu, le point virtuel parmi ceux restants et dont la distance d_j est la plus petite est choisi comme étant le **meilleur point virtuel**. Par exemple, dans la figure 4.19, le point P_3 est choisi.

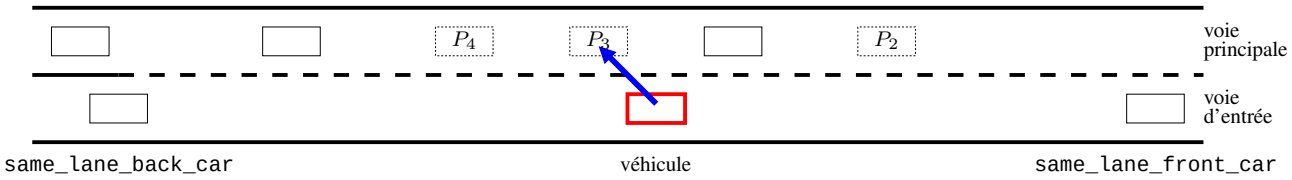


FIG. 4.19 – Exemple de sélection du meilleur point virtuel.

Concernant la loi d'accélération « bang-bang », elle est donnée par l'équation suivante, où x_p et $\dot{x_p}$ sont respectivement la position et la vitesse du point virtuel visé, et x et $\dot{x}$ sont respectivement la position et la vitesse du véhicule qui doit s'aligner avec ce point virtuel :

$$
\begin{array}{l}
u(t) = \text{if } \dot{x}(t) \leq \dot{x_p}(t) \text{ then} \\
\qquad \text{if } x(t) - x_p(t) \geq -\dfrac{(\dot{x_p}(t) - \dot{x}(t))^2}{2 \times a_{max}} \text{ then } a_{max} \\
\qquad \text{else if } \dot{x}(t) \leq 0\ ms^{-1} \text{ then } 0\ ms^{-2} \\
\qquad\qquad \text{else } a_{min} \\
\qquad \text{else if } x(t) - x_p(t) \leq -\dfrac{(\dot{x_p}(t) - \dot{x}(t))^2}{2 \times a_{max}} \text{ then } a_{min} \\
\qquad \text{else if } \dot{x}(t) \geq v_{max} \text{ then } 0\ ms^{-2} \\
\qquad\qquad \text{else } a_{max}
\end{array}
\tag{4.16}
$$

Cette accélération offre le grand avantage de permettre de calculer facilement le temps nécessaire au véhicule pour s'aligner avec le point virtuel choisi. L'inconvénient est que le véhicule dans la voie d'entrée alterne accélération maximale et freinage maximal, ce qui n'est certes pas très confortable.

Si le véhicule dans la voie d'entrée atteint la position $L_1 + L$ sans avoir réussi à s'aligner avec un point virtuel, alors il passe dans la phase `align-to-gap` et tout se déroule comme dans le contrôleur présenté dans la section 4.3, mais avec une moindre distance pour s'insérer. Dans un tel cas, il y a donc un risque que le véhicule ne réussisse jamais à s'insérer. En réalité, les résultats de micro-simulation effectués avec le nouveau contrôleur et dans les mêmes conditions expérimentales montrent le contraire. Non seulement tous les véhicule s'insèrent avec succès, mais en plus le trafic dans la voie principale est bien moins perturbé.

Je présente dans la section suivante le résultat des simulations des deux stratégies d'insertion, celle du contrôleur de la figure 4.13, et celle avec le nouvel état `reach-point` et la nouvelle loi d'accélération (4.16).

4.4 Micro-simulations

4.4.1 Données de travail

Comme je l'ai dit à la section 4.1, je prends comme bornes de l'accélération $[a_{min}, a_{max}] = [-4.905\ ms^{-2}, 1.962\ ms^{-2}]$. De plus, je prends comme vitesse maximale $v_{max} = 28\ ms^{-1}$, soit $100\ kmh^{-1}$. Enfin, je prends $d_{max} = 100\ m$ comme portée des capteurs, ce qui est une distance suffisante pour s'arrêter au cas où un obstacle immobile sur la voie de l'autoroute serait détecté. En effet, à $28\ ms^{-1}$ et avec $a_{min} = -4.905\ ms^{-2}$, il suffit de $80\ m$ pour s'arrêter.

Quant aux conditions de trafic sur mon autoroute, elles sont indiquées par la table 4.1.

	Entrée 1	Entrée 2	Entrée 3
Flux moyen	2000/hr	1000/hr	1000/hr
Probabilité de temps inter-véhicules (s)	Uniforme [1.3,2.3]	Uniforme [3.1,4.1]	Uniforme [3.1,4.1]
Probabilité de sortie	Sortie 1 5% Sortie 2 24% Sortie 3 71%	Sortie 1 0% Sortie 2 25.6% Sortie 3 74.4%	Sortie 1 0% Sortie 2 25.6% Sortie 3 74.4%
Vitesse initiale	11 ms^{-1}	22 ms^{-1}	22 ms^{-1}

TAB. 4.1 – Conditions de trafic.

Les intervalles des lois uniformes sont calculés en divisant 3600 secondes (une heure) par le nombre de véhicules à créer. Pour l'entrée 1, cela fait un véhicule chaque 1.8 seconde. L'intervalle de création est alors de 0.5 seconde avant et après cet instant. Pour l'entrée 1, cela donne donc l'intervalle [1.3,2.3].

La table 4.1 indique la probabilité qu'un véhicule quitte l'autoroute à la sortie i, si il ne l'a pas déjà quittée. Par exemple, un véhicule démarrant à l'entrée 2 a 25.6% de chances de sortir à la sortie 2 et 74.4% à la sortie 3. Étant donné que la sortie 1 est *avant* l'entrée 2, le véhicule ne peut pas sortir à la sortie 1, d'où le 0%.

4.4.2 Résultats

J'ai simulé mon autoroute durant 60 minutes. Le gain λ de la loi de commande était fixé à 7 ms^{-2} (valeur qui est, comme cela est requis par la proposition 4.3, supérieure à $-a_{min} = 4.905\ ms^{-2}$). Le gain μ était fixé à 7 ms^{-2}, et le temps inter-véhicules était fixé à 0.6 seconde. La table 4.2 résume les statistiques à la fin de la simulation.

	distance moyenne de début d'insertion	distance maximale de fin d'insertion	nombre de véhicules insérés	nombre de véhicules rejetés
Entrée 1	41.6 m	153.6 m	2085	0
Entrée 2	56.8 m	168.8 m	1020	0
Entrée 3	73.9 m	185.9 m	1021	0

TAB. 4.2 – Statistiques de la simulation après 60 minutes.

Comme le laissait supposer le théorème 4.4, la simulation s'est déroulée sans une seule collision. Également, il n'y a aucun véhicule rejeté, parce que la **distance maximale** d'insertion est 185.9 m alors que la **distance allouée** est 480 m (voir la section 4.1).

Il reste à évaluer la congestion du trafic. La figure 4.20 montre le profil de vitesse dans la voie principale durant la simulation. Les portions où se déroulent les insertions dans les jonctions d'entrée 2 et 3 sont respectivement les intervalles $[2880\ m, 3360\ m]$ et $[7920\ m, 8400\ m]$. La vitesse minimale dans ces portions est 21.11 ms^{-1}, ce qui est seulement 24.6% de moins que la vitesse nominale (28 ms^{-1}). Plus important, il n'y a que très peu de congestion, et la congestion ne se propage pas *en dehors* des deux jonctions d'entrée. Autrement dit, la réduction de vitesse en amont et en aval des deux jonctions d'entrée est de 0%. Ce résultat est comparable à ceux de [28], pour des paramètres de trafic similaires [3],

[3] Vitesse nominale dans la voie principale = 30.5 ms^{-1}, vitesse nominale dans la voie d'entrée = 23 ms^{-1}, flux de véhicules = 2500 par heure.

mais avec une politique d'espacement constant entre les véhicules (c'est-à-dire des trains de véhicules).

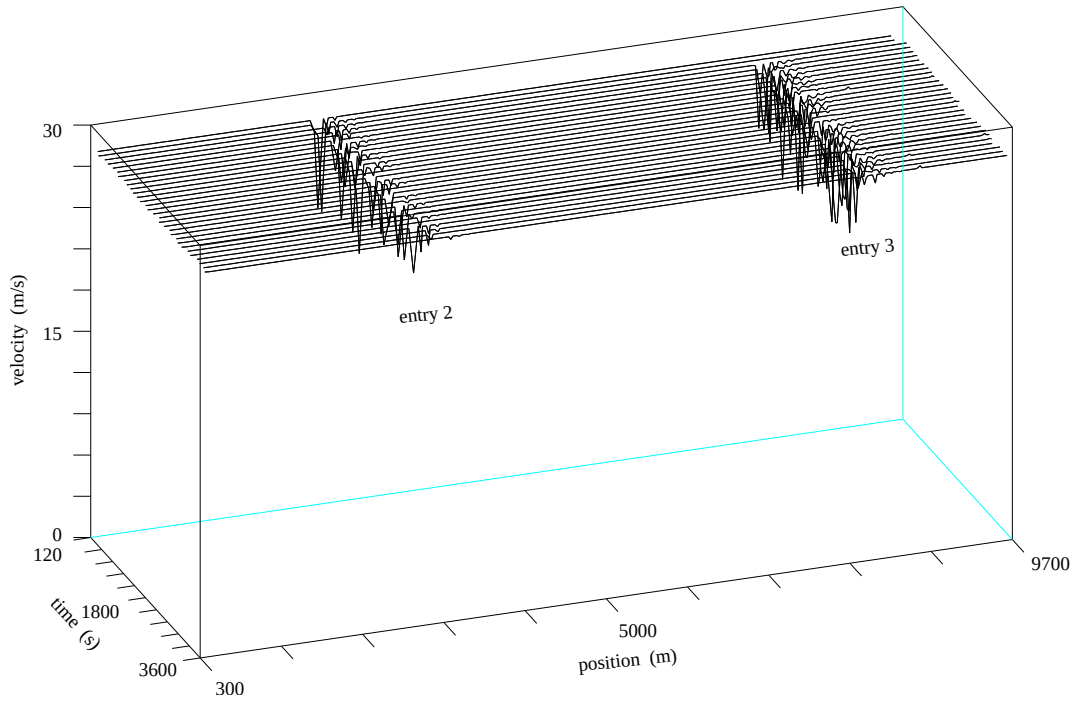


FIG. 4.20 – Profil de vitesse dans la voie principale.

Quant à la stratégie d'insertion alternative développée avec Fethi Bouziani, elle a été simulée avec les mêmes paramètres (conditions de trafic, gains des lois de commande, temps inter-véhicules...), pendant 45 minutes. La distance L qui délimite la zone où les véhicules de la voie d'accélération sont dans la phase `reach-point` a été fixée à 200 m. La table 4.3 résume les statistiques à la fin de la simulation.

	distance moyenne de début d'insertion	distance maximale de fin d'insertion	nombre de véhicules insérés	nombre de véhicules rejetés
Entrée 1	202.5 m	314.7 m	1529	0
Entrée 2	203.6 m	319.4 m	753	0
Entrée 3	203.8 m	321.4 m	755	0

TAB. 4.3 – Statistiques de la simulation après 45 minutes.

Comme on le constate, les nombres de collision et de véhicules rejetés sont toujours égaux à 0. En outre, les véhicules commencent à s'insérer presque immédiatement dans la portion $[L_1 + L, L_1 + L_2]$. Autrement dit ils ne restent quasiment pas, ni dans la phase `align-to-gap` (pour ceux de la voie d'accélération), ni dans la phase `yield` (pour ceux de la voie principale). En conséquence, le trafic dans la voie principale est encore plus fluide. La figure 4.21 illustre le nouveau profil de vitesse dans la voie principale. La comparaison avec la figure 4.20 montre que le trafic dans la voie principale y est bien moins perturbé.

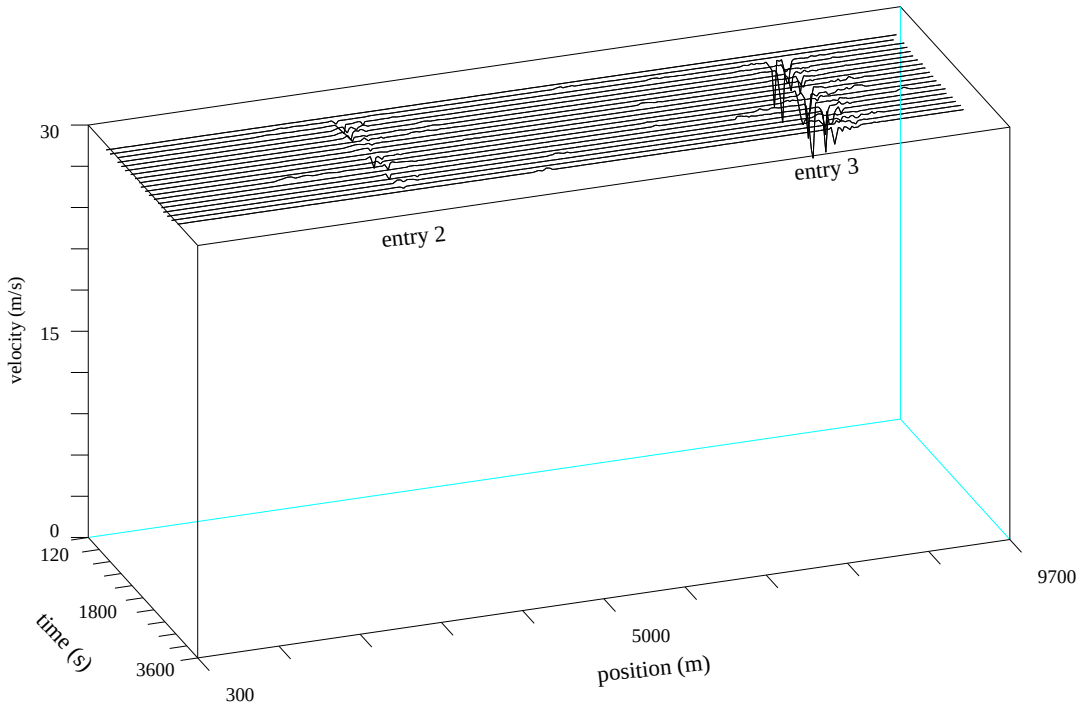


FIG. 4.21 – Nouveau profil de vitesse dans la voie principale.

La raison pour laquelle cette simulation n'a duré que 45 minutes est que je n'ai pas eu à ma disposition de machine SUN SOLARIS avec suffisamment de mémoire vive pour simuler l'autoroute pendant 60 minutes. Par conséquent, la simulation s'est déroulée sur les machines de UC Berkeley. Or, le temps de simulation nécessaire pour 60 minutes dépasse les 72 heures, et la liaison INTERNET avec Berkeley n'étant pas suffisamment fiable, il m'a été impossible d'atteindre le terme de 60 minutes.

4.5 Discussion et développements futurs

4.5.1 Bilan

J'ai présenté dans ce chapitre un contrôleur hybride pour véhicules autonomes roulant sur des autoroutes automatisées. Ce contrôleur est un automate hybride qui régule à la fois le comportement discret des véhicules (les transitions de changement d'état indiquent quand s'insérer, quand céder le passage...) et leur comportement continu (les équations différentielles sont les lois de commande pour l'évolution des variables continues, telles que l'accélération longitudinale, la vitesse latérale...).

J'ai démontré des résultats de régulation par les lois de commande, qui établissent les conditions sous lesquelles il n'y a pas de collision entre les véhicules. Puis j'ai montré qu'en choisissant pour gardes des transitions de l'automate hybride justement les conditions initiales de ces théorèmes de stabilité des lois de commande, le contrôleur hybride garantit qu'il n'y a aucune collision dans toute l'autoroute.

En outre, j'ai montré qu'on pouvait implanter d'autres stratégies d'insertion dans ce contrôleur, tout en préservant le résultat ci-dessus. Ainsi, avec Fethi Bouziani, j'ai étudié et implémenté une autre stratégie d'insertion plus traditionnelle, basée sur l'évaluation des trous inter-véhicules dans la voie principale et la sélection du meilleur trou, c'est-à-dire celui pour lequel l'insertion sera la plus facile et la plus rapide.

De plus cette approche est très flexible et modulaire. En particulier, mon protocole d'insertion a été repris par d'autres équipes de recherche pour d'autres scénarios tels que des autoroutes automatisées

avec camions [10], la consommation d'essence et la pollution [12], l'évitement d'obstacles, l'insertion avec coopération entre les véhicules,...

J'ai effectué plusieurs simulations, à la fois des lois de commande d'accélération individuellement et des autoroutes complètes. Les résultats de ces simulations soulignent la justesse de mon approche et la très bonne qualité du contrôleur hybride (très peu de congestion dans la voie principale au niveau des jonctions d'entrée et des distances d'insertion très faibles).

Enfin, les travaux présentés dans ce chapitre ont été publiés dans les articles suivants : [13, 14, 2, 1].

4.5.2 État de l'art

Les travaux pertinents par rapport à ce travail peuvent être divisés en trois catégories : la commande longitudinale de véhicules automatisés et l'analyse de la régulation ; les stratégies d'insertion pour véhicules automatisés désirant changer de voie ; et le contrôle/commande hybride de véhicules automatisés.

4.5.2.1 Sur la commande longitudinale

Les travaux sur l'analyse de régulation de systèmes dynamiques avec contraintes de saturation incluent [36] et [4]. Toutefois, ces travaux concernent la commande de systèmes avec une dynamique linéaire du premier ou du second ordre, donc leurs résultats ne sont pas transposables ici.

[20] et [8] ont aussi effectué des travaux sur l'AICC avec temps inter-véhicules constant. Leur modèle cinématique est plus complexe que le mien, incluant par exemple un modèle dynamique du moteur. Les auteurs ont réalisé des simulations afin d'affiner les paramètres de leurs lois de commande. Toutefois, ils n'ont démontré aucun résultat de stabilité.

[19] propose une loi de commande longitudinale pour systèmes d'AICC avec temps inter-véhicules constant : la distance inter-véhicules est de la forme $\lambda_2 v_i + \lambda_3$, où λ_2 et λ_3 sont des constantes, et v_i est la vitesse du véhicule suiveur. Leur contrôleur automatisé est comparé avec plusieurs modèles de contrôle humain, et il est montré qu'il évite totalement l'effet accordéon [31], qu'il aboutit à un trafic plus fluide et un flux plus important, grâce à une distance inter-véhicules plus courte et l'élimination du temps de réaction de l'humain.

[21] étudie également la politique de commande avec temps inter-véhicules constant. L'accélération du véhicule suiveur est obtenue en commandant à la fois la pédale d'accélérateur et de frein. Le moteur, la transmission et l'embrayage sont également modélisés. Le premier système étudié consiste en un véhicule leader et un suiveur. Des gains garantissant la régulabilité sont proposés (ce que les auteurs appellent « la stabilité asymptotique du système »). Toutefois, ceci n'est valable que dans le cas non saturé ; quand l'accélération est bornée, les auteurs ne disent pas si leur système est stable ou pas. Le second système étudié consiste en un groupe de véhicules (« platoon of vehicles »), séparés chacun par un temps constant. Le contrôleur proposé garantit la stabilité du groupe de véhicules (c'est-à-dire qu'il n'y a pas d'effet accordéon), à condition que tous les véhicules roulent à une vitesse proche d'une vitesse constante fournie en consigne.

[34] propose deux lois de commande longitudinale, la première avec un objectif d'espacement inter-véhicules constant, et la seconde avec un objectif de temps inter-véhicules constant. Le modèle de véhicule est basé sur [7] et est plus complexe que le mien. L'auteur considère des groupes de véhicules roulant sur une voie rectiligne simple, donc sans changement de voie. La loi de commande pour espacement constant implique la communication entre les véhicules du groupe (le leader diffuse son accélération à tous les autres véhicules) ; elle garantit la stabilité du groupe de véhicules. L'auteur compare le flux maximal avec les deux lois de commande, et comme prévu constate qu'il est meilleur avec l'espacement constant qu'avec le temps constant.

Enfin, beaucoup de recherches ont été faites sur la politique de commande avec temps inter-véhicules constant (par exemple [35] et [25]). Ces travaux se placent dans le cadre des autoroutes automatisées où les véhicules roulent en groupes. Ce type de loi de commande augmente très significativement le

flux maximal des autoroutes, mais il requiert des infrastructures coûteuses, par exemple des rampes, du contrôle d'accès, de la communication... En particulier, il faut une couche pour coordonner les actions des véhicules : insertion, sortie, séparation d'un groupe en deux, fusion de deux groupes...

4.5.2.2 Sur les stratégies d'insertion

[23] propose une stratégie d'insertion reposant, sur le freinage ou l'accélération du véhicule s'insérant afin de créer devant lui un espace inter-véhicules suffisamment grand, puis sur son freinage ou son accélération afin de rendre sa vitesse égale à celle de la voie visée. L'hypothèse fondamentale est qu'à l'exception du véhicule s'insérant, les vitesses de tous les autres véhicules, aussi bien dans la voie d'origine que dans la voie de destination, doivent rester *constantes*. Ce n'est pas le cas dans mon travail, donc ces résultats ne sont pas réutilisables.

[22] traite le problème de l'évitement d'obstacle pendant la manœuvre de changement de voie. De même que [23], les auteurs supposent qu'à l'exception du véhicule s'insérant, les vitesses de tous les autres véhicules, aussi bien dans la voie d'origine que dans la voie de destination, doivent rester *constantes*.

[28] présente une autoroute automatisée constituée d'une voie principale et d'une voie d'entrée avec une rampe d'accès et d'une portion d'insertion. La stratégie d'insertion consiste à viser un trou dans la voie principale et à s'aligner dessus avant de s'insérer. Les trous sont crées dans la voie principale selon des ordres donnés par l'infrastructure. Si aucun trou n'est disponible, alors le véhicule s'arrête dans la rampe d'accès et ne redémarre que quand un trou est enfin disponible.

[24] s'attaque au problème de la commande longitudinale pour des véhicules sur autoroute automatisée. Une trajectoire de référence est produite pour le véhicule qui veut s'insérer, en fonction de la position et de la vitesse des véhicules dans la voie principale, de telle sorte que le véhicule, une fois inséré, forme un groupe avec ceux-ci. Les informations nécessaires à la production de cette trajectoire sont fournies par la couche de coordination de l'autoroute automatisée. La couche de coordination requiert aussi que le groupe de véhicules dans la voie principale de sépare en deux afin de créer un trou acceptable pour le véhicule désirant s'insérer.

4.5.2.3 Sur le contrôle/commande hybride

Le système d'autoroute automatisée présenté dans [28] est muni d'un contrôleur hybride. La partie discrète s'occupe des communications entre l'infrastructure et les véhicules automatisés. Par exemple, l'infrastructure peut envoyer des ordres afin de créer des trous dans la voie principale. La partie continue s'occupe du suivi de véhicule, de la formation des groupes de véhicules et de l'insertion. Toutefois, les auteurs ne démontrent aucun résultat sur la correction de leur contrôleur hybride.

[27] décrit un système de contrôle/commande longitudinal et latéral pour véhicules automatisés. L'ensemble est intégré afin de former un système hybride : chaque véhicule est muni de plusieurs superviseurs modélisés par des automates d'états finis et servant aux différentes manœuvres qu'il veut exécuter. Dans les états discrets, l'accélération du véhicule est déterminée par une équation différentielle appropriée. Là encore, aucun résultat de correction du contrôleur hybride n'est démontré par les auteurs.

Enfin, [15] et [18] décrivent en détails les cinq couches de l'architecture développée par PATH pour les autoroutes automatisées. En particulier, la couche de coordination communique avec les autres véhicules afin de choisir quelle manœuvre exécuter. Elle est modélisée sous la forme d'un automate d'états fini. La couche de régulation exécute les manœuvres telles que la fusion de groupes de véhicules, la séparation en deux, le changement de voie... Elle est modélisée sous la forme de lois de commande avec rétro-contrôle (« feedback ») basées sur des modèles linéaires. L'ensemble constitue donc bien un modèle hybride.

4.5.3 Prise en compte des contraintes sur la vitesse

J'ai écrit dans la section 4.2.2 que ma loi de commande saturée de l'accélération longitudinale (équation (4.14)) garantit que le véhicule suiveur n'entre jamais en collision avec le véhicule leader : c'est le théorème 4.2. Les conditions de ce théorème sont que l'accélération du leader $\ddot{z}$ soit *toujours* dans l'intervalle $[a_{min}, a_{max}]$, que l'accélération du suiveur $\ddot{x}$ soit *initialement* dans le même intervalle $[a_{min}, a_{max}]$, et que la différence entre la vitesse du leader et du suiveur $\dot{z} - \dot{x}$, divisée par le temps inter-véhicule désiré h, soit elle aussi *initialement* dans le même intervalle $[a_{min}, a_{max}]$. Or ce théorème ne tient pas compte d'une autre contrainte de ma route automatisée, qui est que la vitesse du suiveur $\dot{x}$ doit *toujours* être dans l'intervalle $[0, v_{max}]$. Autrement dit, la loi de commande de l'accélération longitudinale ne devrait pas être celle de l'équation (4.14), mais plutôt :

$$\begin{aligned} u(t) &= \frac{\dot{z}(t) - \dot{x}(t)}{h} + \lambda \left(\frac{z(t) - x(t)}{h\dot{x}(t)} - 1 \right) \\ \ddot{x}(t) = u_{sat}(t) &= \text{if } \dot{x}(t) \geq v_{max} \text{ or } \dot{x}(t) \leq 0\ ms^{-1} \text{ then } 0\ ms^{-2} \\ &\quad \text{else if } u(t) \geq 1.962\ ms^{-2} \text{ then } 1.962\ ms^{-2} \\ &\quad \text{else if } u(t) \leq -4.905\ ms^{-2} \text{ then } -4.905\ ms^{-2} \\ &\quad \text{else } u(t) \end{aligned} \tag{4.17}$$

Je trouve intéressant d'étudier le système constitué de deux véhicules, un leader (position z, vitesse $\dot{z}$, accélération $\ddot{z}$) et un suiveur (position x, vitesse $\dot{x}$, accélération $\ddot{x}$) régit par la commande longitudinale de l'équation (4.17), sous la forme d'un **système de contraintes** :

$$\begin{cases} \ddot{x}(t) &= \dfrac{\dot{z}(t) - \dot{x}(t)}{h} + \lambda \left(\dfrac{z(t) - x(t)}{h\dot{x}(t)} - 1 \right) \\ \dot{x} &\geq 0 \\ \dot{x} &\leq v_{max} \\ \ddot{x} &\leq a_{max} \\ \ddot{x} &\geq a_{min} \end{cases} \tag{4.18}$$

La particularité d'un tel système est de mélanger des contraintes **bilatérales** (=) et des contraintes **unilatérales** ($\geq$ et $\leq$). Le but de ces recherches sera d'étudier sous quelles conditions on peut garantir qu'un tel système est stable, autrement dit n'entre jamais en collision.

Afin de comprendre comment se comporte le système (4.18), on peut tracer son diagramme de phase dans le plan $(\dot{x}, \ddot{x})$. Prenons par exemple :

leader

$z(0)$	$133.5\ m$
$\dot{z}(0)$	$0\ ms^{-1}$
$\ddot{z}(0)$	$0\ ms^{-2}$

suiveur

$x(0)$	$0\ m$
$\dot{x}(0)$	$28\ ms^{-1}$

Le calcul donne $\frac{\dot{z}(0)-\dot{x}(0)}{h} = \frac{0-28}{0.6} = -46.667\ ms^{-2}$ et $\lambda(r(0) - 1) = 7 \times (\frac{133.5}{0.6 \times 28} - 1) = 48.625\ ms^{-2}$, d'où $\ddot{x}(0) = u(t) = 48.625 - 46.667 = 1.958\ ms^{-2}$. Autrement dit, l'accélération initiale du véhicule suiveur est très proche de a_{max}. Notez bien que toutes les conditions du théorème 4.2 sont satisfaites. La figure 4.22 illustre le comportement du véhicule suiveur, respectivement avec la loi de commande (4.17) en rouge et avec la loi (4.14) en bleu.

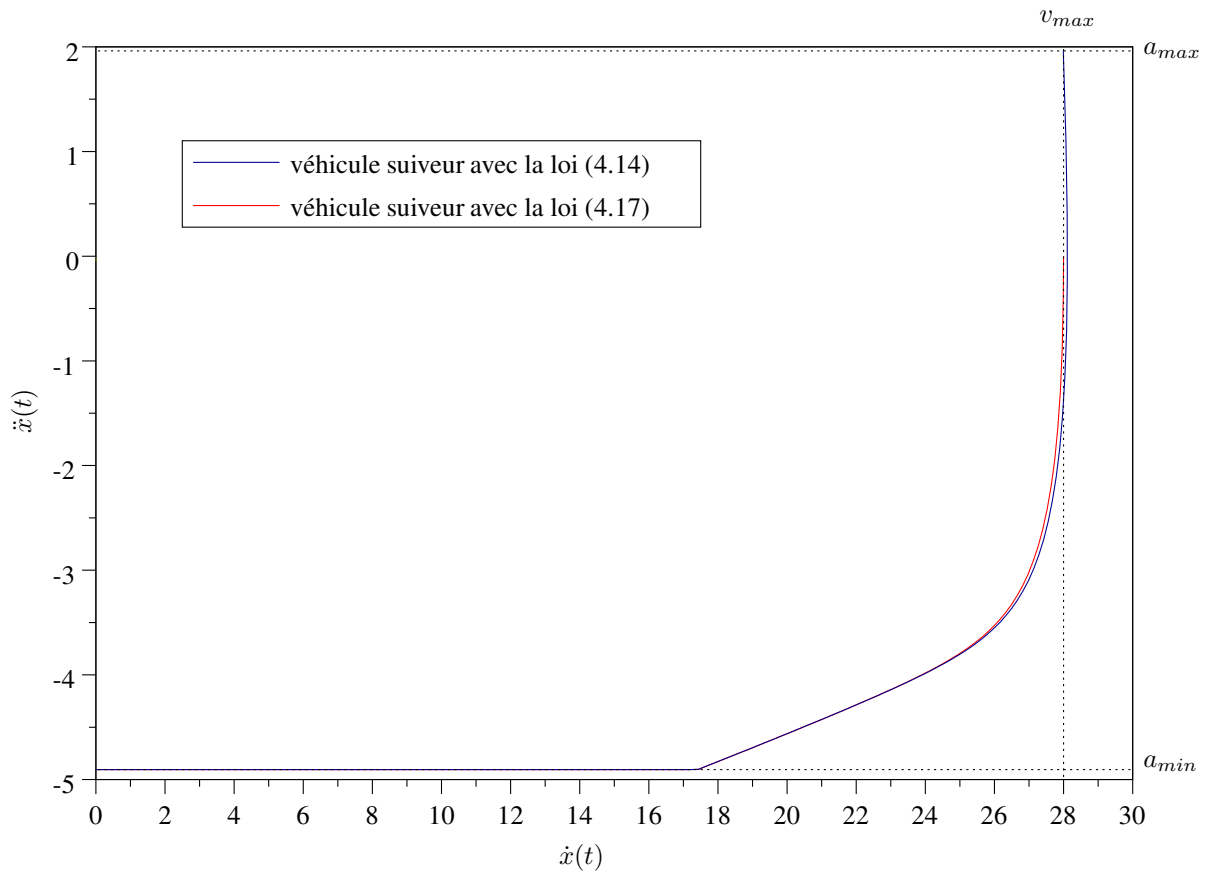


FIG. 4.22 – Deux exemples de diagrammes de phase dans le plan $(\dot{x}, \ddot{x})$

On constate qu'avec la loi (4.14), la vitesse du véhicule suiveur dépasse v_{max}, et c'est justement l'influence de ce dépassement qu'il faudra étudier.

Bibliographie

[1] M. Antoniotti, A. Deshpande, and A. Girault. Microsimulation analysis of a hybrid system model of multiple merge junction highways and semi-automated vehicles. In *IEEE International Conference on Systems Man and Cybernetics, SMC'97*, Orlando, USA, October 1997. IEEE.

[2] M. Antoniotti, A. Deshpande, and A. Girault. Microsimulation analysis of multiple merge junctions under autonomous AHS operation. In *IEEE Conference on Intelligent Transportation Systems, ITSC'97*, Boston, USA, November 1997. IEEE.

[3] P. Antsaklis. Hybrid systems : Theory and applications. *Proceedings of the IEEE*, 88(7), July 2000. Special Issue.

[4] K.J. Åström and S. Brufani. Manual control of an unstable system with a saturating actuator. In *IEEE Conference on Decision and Control, CDC'97*, pages 964–965, San Diego, USA, December 1997. IEEE.

[5] G. Baille, P. Garnier, H. Mathieu, and R. Pissard-Gibollet. Le CYCAB de l'Inria Rhône-Alpes. Rapport technique 0229, Inria, Rocquencourt, France, April 1999.

[6] F. Bouziani. Étude et implémentation de stratégies d'insertion pour une autoroute automatisée. Rapport de DEA, EEATS, INPG, Grenoble, France, June 2003.

[7] D. Cho and J.K. Hedrick. Automotive powertrain modelling for control. *Journal of Dynamic Systems, Measurement, and Control*, 111(4), December 1989.

[8] P. Daviet and M. Parent. Platooning techniques for empty vehicle distribution in the PRAXITÈLE project. In *4th Mediterranean Symposium on New Directions in Control and Automation*, Maleme, Greece, June 1996.

[9] A. Deshpande, A. Göllü, and L. Semenzato. The Shift programming language for dynamic networks of hybrid automata. *IEEE Trans. on Automatic Control*, 43(4) :584–588, April 1998. Research report UCB-ITS-PRR-97-7.

[10] F. Dufraisse and F. Hertschuh. Développement de simulations et analyse de faisabilité dans le cadre du projet de route automatisée. Rapport de stage ingénieur, École des Mines de Paris, Paris, France, June 1998.

[11] R.E. Fenton. A headway safety policy for automated highway operations. *IEEE Trans. on Vehicular Technology*, VT-28, 1979.

[12] C.H. Gibson III. The effectiveness of automated highway merging protocols on highway operations and vehicle performance. Master of Science, UC Berkeley, Berkeley, USA, 1997. `http://citeseer.ist.psu.edu/45841.html`.

[13] A. Girault. A hybrid controller for autonomous vehicles driving on automated highways. *Transportation Research Part C : Emerging Technologies*, 12(6) :421–452, December 2004. Research report Inria 4286.

[14] A. Girault and S. Yovine. Stability analysis of a longitudinal control law for autonomous vehicules. In *IEEE Conference on Decision and Control, CDC'99*, Phœnix, USA, December 1999. IEEE. Research report Inria 3498.

[15] D. Godbole, J. Lygeros, and S. Sastry. Hierarchical hybrid control : A case study. In *Conference on Decision and Control, CDC'94*, pages 1592–1597, Orlando, USA, December 1994. IEEE.

[16] J.K. Hedrick, M. Tomizuka, and P. Varaiya. Control issues in automated highway systems. *IEEE Control Systems Mag.*, 14(6) :21–32, December 1994.

[17] P. Hingwe and M. Tomizuka. Experimental evaluation of a chatter free sliding mode control for lateral control in AHS. In *American Control Conference, ACC'97*, Albuquerque, USA, 1997. IEEE.

[18] R. Horowitz and P. Varaiya. Control design of an automated highway system. *Proceedings of the IEEE*, 88(7) :913–925, July 2000.

[19] P. Ioannou and C.C. Chien. Autonomous intelligent cruise control. *IEEE Trans. on Vehicular Technology*, 42 :657–672, November 1993.

[20] P.A. Ioannou, F. Ahmed-Zaid, and D.H. Wuh. A time headway autonomous intelligent cruise controller : Design and simulation. Rapport de recherche UCB-ITS-PWP-94-07, University of Southern California, April 1994.

[21] P.A. Ioannou and Z. Xu. Throttle and brake control systems for automatic vehicle following. *IVHS Journal*, 1(4) :345–377, 1994.

[22] H. Jula, E.B. Kosmatopoulos, and P.A. Ioannou. Collision avoidance analysis for lane changing and merging. *IEEE Trans. on Vehicular Technology*, 49(6) :2295–2308, November 2000.

[23] A. Kanaris, E.B. Kosmatopoulos, and P.A. Ioannou. Strategies and spacing requirements for lane changing and merging in automated highway systems. *IEEE Trans. on Vehicular Technology*, 50(6) :1568–1581, November 2001.

[24] X.-Y. Lu and K.J. Hedrick. Longitudinal control algorithm for automated vehicle merging. In *IEEE Conference on Decision and Control, CDC'00*, Sydney, Australia, December 2000. IEEE.

[25] T.S. No, K.-T. Chong, and D.-H. Roh. A Lyapunov function approach to longitudinal control of vehicles in a platoon. *IEEE Trans. on Vehicular Technology*, 50(1) :116–124, January 2001.

[26] H.A. Pham, J.K. Hedrick, and M. Tomizuka. Combined lateral and longitudinal control of vehicles for AHS. In *American Control Conference, ACC'94*, Baltimore, USA, 1994. IEEE.

[27] R. Rajamani, H.-S. Tan, B.K. Law, and W.-B. Zhang. Demonstration of integrated longitudinal and lateral control for the operation of automated vehicles in platoons. *IEEE Trans. on Control Systems Technology*, 8(4) :695–708, 2000.

[28] B. Ran, S. Leight, and B. Chang. A microscopic simulation model for merging control on a dedicated-lane automated highway system. *Transportation Research Part C : Emerging Technologies*, 7(6) :369–388, 1999.

[29] J.-P. Roussel. Réalisation d'une loi de commande pour le contrôle longitudinal d'un véhicule automatique. Rapport de stage ingénieur, ESSAIM, Mulhouse, France, July 2001.

[30] S. Sekhavat and J. Hermosillo. The Cycab robot : A differentially flat system. In *IEEE Intelligent Robots and Systems, IROS'00*, Takamatsu, Japan, November 2000. IEEE.

[31] S. Sheikholelslam and C.A. Desoer. A system level study of the longitudinal control of a platoon of vehicles. *Journal of Dynamic Systems, Measurement, and Control*, 114(2) :286–292, June 1992.

[32] S.E. Shladover. Review of the state of development of advanced vehicle control systems (AVCS). *Vehicle System Dynamics*, 24 :551–595, 1995.

[33] D. Simon, B. Espiau, E. Castillo, and K. Kapellos. Computer-aided design of a generic robot controller handling reactivity and real-time control issues. *IEEE Trans. on Control Systems Technology*, 1(4), December 1993.

[34] D. Swaroop, J.K. Hedrick, C.C. Chien, and P.A. Ioannou. A comparison of spacing and headway control laws for automatically controlled vehicles. *Vehicle System Dynamics*, 23(8) :597–626, November 1994.

[35] D. Swaroop, J.K. Hedrick, and S.B. Choi. Direct adaptative longitudinal control of vehicle platoons. *IEEE Trans. on Vehicular Technology*, 50(1) :150–161, January 2001.

[36] H. Ye, A.N. Michel, and L. Hou. Stability theory for hybrid dynamical systems. *IEEE Trans. on Automatic Control*, 43(4) :461–474, April 1998.

Chapitre 5

Conclusion

J'ai envie de terminer ce document par un peu de prospective sur les systèmes informatiques embarqués. Je vais prendre comme exemple l'automobile, un domaine grand public où les volumes produits rendent nécessaire d'atteindre des niveaux de sûreté de fonctionnement équivalents à l'avionique civile. La fin des années 1990 a vu une explosion du nombre de systèmes informatiques embarqués dans les automobiles. Une voiture haut de gamme a ainsi pu en compter jusqu'à 80 : assistance au freinage, direction assistée, déclenchement des airbags, injection électronique, vitres électriques, climatisation, mais aussi tout ce qu'on appelle l'« infotainment », GPS, téléphonie, autoradio... Certains sont donc à sûreté critique, et d'autres non. Cette tendance va très certainement se poursuivre à l'avenir. En particulier, les constructeurs automobiles travaillent actuellement sur ce qu'on appelle le « drive-by-wire » (comme c'est déjà le cas dans l'avionique avec le « fly-by-wire ») : direction électronique (sans colonne de direction mécanique) et freinage électronique. Toutefois, de nombreuses défaillances, qui se sont produites justement sur ces voitures haut de gamme (en particulier affectant l'injection et le freinage), ont entraîné une prise de conscience des difficultés qu'il y a à produire de façon sûre ce type de systèmes :

- Une première difficulté vient du mode de fonctionnement des constructeurs automobiles et de leurs sous-traitants : un système est tout d'abord spécifié par le constructeur, puis il est conçu et développé par le sous-traitant, avant d'être intégré dans l'automobile par le constructeur. Or bien souvent, la spécification se fait sans méthode formelle, ce qui est une source bien connue de fautes de développement.
- Une seconde difficulté vient du fait que beaucoup de systèmes sont déployés sur un même processeurs, alors qu'ils ont des niveaux de sûreté de fonctionnement très différents. Par conséquent, une défaillance dans un système peu critique peut entraîner une défaillance de l'ensemble des systèmes exécutés sur le même processeur, entraînant ainsi une défaillance grave.

Si on examine le cas de l'avionique civile, il est frappant de constater que ces difficultés y ont été parfaitement comprises, il y a déjà très longtemps, et que des solutions efficaces ont été trouvées : par exemple, l'utilisation de méthodes de spécification formelle est intégrée au processus de développement (par exemple SCADE chez EADS, outil qui permet non seulement la spécification formelle mais aussi la génération de code embarquable certifié), et les systèmes qui gèrent les commandes de vol (ce qui est le plus critique dans un avion civil) sont complètement séparées des systèmes d'infotainment. C'est loin d'être le cas aujourd'hui dans l'automobile, mais je crois que les pratiques industrielles sont en train de changer. Par exemple, DASSAULT SYSTÈME est en train de mettre au point, pour le domaine automobile, des outils de développement de systèmes informatiques embarqués utilisant des méthodes formelles (il s'agit d'une variante d'ESTEREL), permettant tout comme SCADE de spécifier formellement un système et de générer automatiquement du code embarquable conforme à la spécification.

Du côté de la recherche, de nombreuses équipes travaillent à l'application de méthodes formelles aux problèmes spécifiques de la conception des systèmes informatiques embarqués : langages de programmation, compilation, model-checking, synthèse de contrôleurs discrets et hybrides, répartition au-

tomatique, tolérance aux fautes, interprétation abstraite, raffinement... Je contribue moi-même à ces recherches.

Cette convergence de vue entre les sociétés éditrices de logiciels et les équipes de recherche est, à mon avis, une bonne chose pour l'avenir. Mais le coût très élevé de ces logiciels risque de limiter leur utilisation à certains domaines critiques, comme l'avionique, le spatial, l'automobile... En revanche, pour beaucoup de produits électroniques grand public, la pression de la concurrence et leur faible coût va probablement empêcher l'utilisation de ces logiciels. Concernant les systèmes peu complexes, cela ne devrait pas avoir de conséquences trop graves ; en revanche, pour les gros systèmes qui nécessitent de nombreux ingénieurs pendant de nombreux mois de développement, des fautes de développement seront inévitables en raison de leur très grande complexité. Il reste à espérer que cela se limitera aux systèmes peu critiques.

Résumé

Je présente dans ce document mes résultats de recherche sur la conception sûre de systèmes embarqués sûrs. La première partie concerne la répartition automatique de programmes synchrones. Le caractère automatique de la répartition apporte un réel degré de sûreté dans la conception de systèmes répartis car c'est la partie la plus délicate de la spécification qui est automatisée. Grâce à cela, l'absence d'inter-blocage et l'équivalence fonctionnelle entre le programme source centralisé et le programme final réparti peuvent être formellement démontrées. La deuxième partie traite le sujet de l'ordonnancement et de la répartition de graphes de tâches flots-de-données sur des architectures à mémoire répartie, avec contraintes de tolérance aux fautes et de fiabilité. Je présente principalement des heuristiques d'ordonnancement statique multiprocesseur avec pour but la tolérance aux fautes et la fiabilité des systèmes, mais également l'utilisation de méthodes formelles telles que la synthèse de contrôleurs discrets ou les transformations automatiques de programmes. Enfin, la troisième partie concerne les autoroutes automatisées, avec deux volets : la commande longitudinale de véhicules autonomes et les stratégies d'insertion dans les autoroutes automatisées.

Summary

In this document, I present my research results on the safe design of safe embedded systems. The first part concerns the automatic distribution of synchronous programs. The automatic aspect provides an essential safety level in the design of distributed systems because it is the most delicate part of the specification that is being automated. Thanks to it, the absence of deadlock and the functional equivalence between the centralized source program and the distributed final program can be formally proved. The second part deals with the scheduling and the distribution of dataflow task graphs onto distributed memory architectures, with fault-tolerance and reliability constrains. I mainly present multiprocessor static scheduling heuristics with fault-tolerance and reliability goals, but also the use of formal methods such as discrete controller synthesis and automatic program transformations. Finally, the third part concerns automated highways, with two aspects : the longitudinal control of autonomous vehicles, and the insertion protocols in automated highways.