THÈSE

présentée à

l'université de ROUEN
U.F.R de Sciences

pour obtenir le titre de

Docteur en sciences

Spécialité
INFORMATIQUE INDUSTRIELLE

par

Rémy KOCIK

SUR L'OPTIMISATION DES SYSTÈMES DISTRIBUÉS TEMPS RÉEL EMBARQUÉS : APPLICATION AU PROTOTYPAGE RAPIDE D'UN VÉHICULE ÉLECTRIQUE AUTONOME

Soutenue le 22 mars 2000 devant le jury composé de :

Jean-Pierre ELLOY	Rapporteur
Bernard ESPIAU	Rapporteur
Samuel BOUTIN	Examinateur
Pierre MICHÉ	Examinateur
Michel PARENT	Examinateur
Jacques LABICHE	Directeur de thèse
Yves SOREL	Responsable de recherche

REMERCIEMENTS

Je tiens tout particulièrement à remercier Yves Sorel, Directeur de Recherche à l'INRIA, qui m'a encadré tout au long de cette thèse. Je lui suis reconnaissant pour la disponibilité dont il a fait preuve et pour la confiance qu'il m'a accordée.

Ma reconnaissance va également à Michel Parent qui m'a accueilli au sein de son projet, ainsi qu'à Jacques Labiche qui m'a régulièrement encouragé et m'a permis de m'inscrire en thèse.

Je tiens aussi à remercier vivement :

- Monsieur Jean-Pierre Elloy, Directeur de Recherche à l'IRCyN,
- Monsieur Bernard Espiau, Directeur de Recherche à l'INRIA,

pour m'avoir fait l'honneur d'être les rapporteurs de cette thèse,

- Monsieur Samuel Boutin, Ingénieur de Recherche chez Renault,
- Monsieur Pierre Miché, Professeur à l'université de Rouen,

pour avoir accepté de participer au jury.

Un grand merci également à tous les autres membres de l'équipe SynDEx, Thierry Grandpierre, Christophe Lavarenne et Annie Vicard qui, à travers les discussions que nous avons eues, m'ont apporté de nombreuses idées qui m'ont permis d'avancer dans mon travail.

Je remercie Chantal Chazelas, pour m'avoir soulagé de nombreuses tâches administratives et Georges Ouanounou pour l'aide et les nombreux services qu'il m'a rendu.

Que ma famille soit remerciée pour m'avoir soutenue et m'avoir permis de réaliser mes études dans les meilleures conditions.

Enfin, je tiens particulièrement à remercier Catherine, ma femme, qui m'a supporté tout au long de cette thèse et qui m'a délesté de nombreuses tâches quotidiennes afin que je puisse me consacrer pleinement à ma thèse, souvent au détriment de ses propres études. Je remercie aussi Héléna pour "avoir fait ses nuits" très rapidement et surtout pour son sourire charmeur.

TABLE DES MATIÈRES

INTRODUCTION

Les systèmes automatisés ont été conçus, à l'origine, pour remplacer l'homme dans l'accomplissement de tâches fastidieuses, dangereuses ou dépassant ses capacités physiques. De nos jours, le domaine d'application de ces systèmes a été largement étendu puisque leur utilisation s'est démocratisée au point d'envahir notre vie quotidienne. On ne les trouve plus seulement dans les systèmes industriels complexes ; ils ont fait leur apparition dans les produits de consommation les plus courants (électroménager, domotique, jouets . . .). Dans ce nouveau contexte leur but est souvent d'améliorer le confort et la sécurité de leurs utilisateurs. Le domaine de l'automobile est un bon exemple de cette évolution. Hier, les systèmes automatisés étaient surtout utilisés dans les ateliers de montage des véhicules. Aujourd'hui le véhicule lui-même intègre de nombreux systèmes automatisés tels que, par exemple, le contrôle moteur, le freinage ABS, la boîte de vitesses automatique ou bien encore la direction assistée. Prenons par exemple cette dernière, il y a quelques années réalisée mécaniquement, elle peut aujourd'hui être électrique et pilotée par microprocesseur. Elle permettra, dans les prochaines générations de véhicule, un contrôle automatique de la conduite (fonction de pilote automatique). La tendance actuelle est au remplacement des commandes mécaniques par des systèmes d'asservissement électronique.

Cette évolution a été rendue possible grâce aux progrès réalisés dans le domaine de la conception des circuits électroniques permettant de construire des microprocesseurs de plus en plus puissants. Un système automatisé est composé d'un système contrôlé et d'un système de contrôle. Le système de contrôle étant conçu pour assujettir le système contrôlé à suivre un "certain comportement" pour lequel le système automatisé est défini. Les puissances de calcul de plus en plus importantes apportées par les microprocesseurs permettent d'une part à l'automaticien de réaliser des outils de calculs formels et de simulations nécessaires à une modélisation fine des phénomènes physiques qui régissent le comportement d'un système à contrôler ; d'autre part, les microprocesseurs intégrés au sein du système de contrôle autorisent l'implantation de ces modèles fins puisqu'ils rendent possible l'exécution des calculs de plus en plus nombreux et de plus en plus complexes qu'ils nécessitent (algorithmes de traitement du signal, de traitement d'images, de contrôle-commande...). Ces systèmes de contrôle numériques dits systèmes temps réel confèrent aux systèmes automatisés auxquels ils appartiennent une plus grande précision, un comportement plus sûr et plus "intelligent".

Le cycle de conception et de développement de ces systèmes nécessite une collaboration forte et permanente entre automaticiens et informaticiens. Dans une première phase, l'automaticien est chargé de définir un modèle mathématique du système. Il doit prendre en compte les problèmes de discrétisation liés à une réalisation numérique du système de contrôle. Ce passage d'un modèle

continu à un modèle discret se traduit par des contraintes de temps (liées à l'échantillonnage des signaux) sur la fréquence et le temps d'exécution des calculs que devra effectuer le système temps réel pour suivre le modèle défini. Dans une deuxième phase, l'informaticien réalise les programmes qui doivent implanter ces calculs sur le système temps réel. Lors de cette phase d'implantation, l'informaticien doit être capable de garantir que ses programmes satisfont les contraintes de temps d'exécution définies par l'automaticien. La dernière phase du cycle de conception consiste à relier physiquement le système temps réel au système à contrôler. Il faut alors vérifier que le système automatisé se comporte comme prévu. Dans la majorité des cas, il est nécessaire de tester les programmes afin que le système de contrôle se comporte comme le modèle défini. Il est aussi nécessaire d'observer le comportement du système automatisé. Ces observations permettront d'établir les écarts qu'il y a entre le fonctionnement réel et le fonctionnement voulu du système automatisé. Si cet écart est acceptable le système est validé. Si ce n'est pas le cas, on retourne à la première phase de développement afin de modifier ou d'affiner les modèles. Ce cycle de développement est effectué autant de fois que nécessaire jusqu'à la validation du système automatisé.

La réalisation d'un système temps réel nécessite une bonne maîtrise des outils fournis par la théorie de l'automatique lors de la phase de modélisation et de simulation, ainsi qu'une bonne maîtrise de l'informatique temps réel lors de la phase d'implantation. La recherche sur la programmation des systèmes temps réel tente de fournir des méthodes de développement permettant de gérer au mieux cette grande complexité et d'aboutir à la réalisation de systèmes sûrs et efficaces. C'est dans ce but qu'ont été développées les méthodes dites "formelles". Ces méthodes cherchent à définir et à regrouper des outils de spécification (description haut niveau -la plus proche du modèle- du système), des outils de vérification permettant de simuler et de vérifier les propriétés de la spécification ainsi que des outils de génération automatique des programmes temps réel implantant la spécification. Réunir sous un même environnement de développement, un langage de spécifications, des outils de vérification et de génération automatique de programmes temps réel capables de prendre en compte différents types d'architectures (notamment multiprocesseurs) et capables de garantir que le code généré est conforme à la spécification, constituerait l'outil de conception idéal capable d'aboutir rapidement à la réalisation d'un système sûr et optimisé. Réaliser un tel outil est l'un des challenges actuels de la recherche sur la programmation des systèmes temps réel.

Cette thèse s'inscrit dans le cadre des approches formelles. Elle aborde les problèmes de la spécification et de l'implantation de cette spécification dans les systèmes temps réel distribués embarqués. La méthodologie AAA[1] que nous présentons, a été conçue et utilisée avec succès pour la réalisation de systèmes temps réel dont l'architecture matérielle est de type multiprocesseurs. Elle autorise la spécification des algorithmes à l'aide de graphes flots de données ce qui la rend bien adaptée à la description des lois de commandes qui sont généralement décrits par des schémas-bloc. En revanche, les discontinuité dans le comportement du système sont généralement décrites par des graphes d'automate. La méthodologie AAA ne permets pas actuellement de prendre en compte ce type de spécification. Elle ne permet pas non plus de prendre en compte plusieurs contraintes temporelles dans une même application. Dans cette thèse, nous cherchons à adapter cette méthodologie aux besoins des concepteurs de systèmes informatiques embarqués nécessitant la prise en compte de fortes contraintes de coûts (coût financier, encombrement, consommation ...) rencontrées notamment dans les applications de type automobile. Nous chercherons donc à étendre la méthodologie afin de fournir dans un environnement de développement cohérent la possibilité de spécifier les applications avec les deux types de graphes utilisés par l'automaticien (flot de données et flot de contrôle), de permettre la spécification de contraintes temporelles multiples et de générer auto-

1. Adéquation Algorithme Architecture

matiquement une implantation efficace sur une architecture distribuée prenant en compte tous les aspects de cette spécification.

Dans la première partie nous présentons de manière détaillée les contraintes de la programmation de ces systèmes temps réel embarqués. La deuxième partie est consacrée à l'extension du langage de spécification de la méthodologie AAA afin qu'il prenne mieux en compte les besoins de l'automaticien. On améliore l'expressivité de cette spécification, d'une part, en fournissant la possibilité de séparer et de décrire les aspects fonctionnels (lois de commande) et les aspects événementiels (changements de mode de fonctionnement) du système à l'aide de deux formalismes différents (flot de données, flot de contrôle), et d'autre part, en ajoutant aussi la possibilité de spécifier différentes contraintes temporelles.

La troisième partie traite la prise en compte des différentes contraintes (matérielles, algorithmiques et temporelles) par la méthodologie AAA lors de la phase d'implantation. Nous proposons notamment une extension permettant une implantation optimisée d'une spécification comportant plusieurs contraintes temporelles, AAA n'optimisant jusqu'alors qu'une seule contrainte de temps, la durée totale d'exécution de tous les calculs.

Dans la dernière partie, nous montrons comment utiliser la méthodologie AAA à travers une application concrète, le prototype d'un véhicule autonome.

Première partie

Problématique des systèmes temps réel embarqués

CHAPITRE 1

LOIS DE COMMANDES DISCRÈTES

La conception d'un système temps réel passe nécessairement par une étape de modélisation. Cette étape nécessite une bonne maîtrise des outils mathématiques issus de la théorie de l'automatique. C'est sur la base de cette théorie que seront ensuite définis les algorithmes des calculs chargés de conférer au système un comportement proche du modèle défini, ainsi que les contraintes de temps d'exécution sur ces algorithmes. La grosse difficulté sera ensuite de pouvoir réaliser une implantation garantissant que ces contraintes de temps seront toujours respectées à l'exécution. Une bonne connaissance de ces contraintes est nécessaire pour réaliser une implantation efficace des algorithmes, c'est pourquoi, afin de bien comprendre leur nature et leur provenance, il nous semble utile de présenter quelques notions d'automatique. Ce chapitre n'a la prétention d'être ni un cours ni un rappel de cours d'automatique, nous présentons ici simplement les notions qui nous permettent de mettre en évidence certaines propriétés qui conditionneront l'implantation.

1.1 Système automatisé

1.1.1 Système

" Un système est un objet complexe, formé de composants distincts reliés entre eux par un certain nombre de relations. Les composants sont considérés comme des sous-systèmes, ce qui signifie qu'ils entrent dans la même catégorie d'entités que les ensembles auxquels ils appartiennent. . . . L'idée essentielle est que le système possède un degré de complexité plus grand que ces parties, autrement dit qu'il possède des propriétés irréductibles à celles des composants. . . . Le but de la théorie des systèmes est de déterminer ce qu'on peut dire de l'évolution d'un système quand on possède à son sujet telles ou telles informations (par exemple, structure interne, lois d'interaction entre composants, perturbations d'origine externe, état à un certain instant, évolution au cours d'une certaine période, etc.) "

(Encyclopédie Universalis)

Nous nous intéressons ici aux systèmes matériels. Les composants qui les constituent sont des éléments mécaniques, électriques et/ou chimiques liés physiquement entre eux [61].

1.1.2 Processus

L'automaticien cherche à maîtriser, en fonction d'un but défini, l'évolution dans le temps d'un système matériel appelé *processus* (ou *procédé*). Son travail consiste, dans une première phase de *modélisation*, à décrire le processus sous la forme d'équations mathématiques. Le *modèle* qu'il construit ainsi, lui permet de prédire l'évolution du processus lorsque ses composants sont soumis à différents phénomènes physiques (ou *actions*) externes.

1.1.3 Lois de commande, système de commande

A l'issue de l'étude du processus, l'automaticien cherche à définir un ensemble d'actions, les *commandes*, qui permettront de le faire évoluer selon un but prédéfini. Il en donne un modèle mathématique, les *lois de commande* qui permettent alors de concevoir un système matériel, le *système de commande*, dont la fonction est de réaliser physiquement les commandes. Le processus (système dont on cherche à maîtriser l'évolution) et le système de commande (système qui doit agir sur le processus) forment ensemble le *système automatisé*.

La figure 1.1 présente les deux grandes étapes qui conduisent à la réalisation du système automatisé ; dans un premier temps, à partir du processus - système matériel que l'on cherche à commander- on réalise un système mathématique, le modèle du système automatisé composé d'un modèle mathématique du processus et de lois de commande, modèle mathématique du système de commande. La transformation qui réalise le passage d'un espace matériel à un espace mathématique est la *modélisation*. Dans un deuxième temps, on réalise le système automatisé, système matériel conforme à son modèle mathématique défini lors de la modélisation. Cette réalisation - transformation d'un modèle mathématique en système matériel- consiste à concevoir un système matériel de commande conformément à son modèle mathématique et à assembler celui-ci avec le processus de départ.

1.2 Modèle de loi de commande

1.2.1 Décomposition du processus en sous-processus

Afin de faciliter la conception du système de contrôle-commande, l'automaticien cherche à décomposer un processus complexe en sous-processus plus simples, le plus possible autonomes, c'est-à-dire ayant le moins d'interactions possibles entre eux. Le système de contrôle-commande peut alors être conçu comme un ensemble de sous-systèmes construits sur la base de lois de commande plus simples [20].

Cette décomposition a pour but de réduire la complexité des lois de commande. Elle tend à leur donner, autant que possible, une structure monovariable (1 entrée, 1 sortie). Les interactions entre ces systèmes monovariables, si elles ne sont pas trop fortes, peuvent être considérées comme des perturbations externes. Lorsque ces interactions sont fortes, elles doivent être modélisées et intégrées en entrée/sortie des lois de commande. Dans ce cas, les lois de commande qui en résultent ont une structure multivariable.

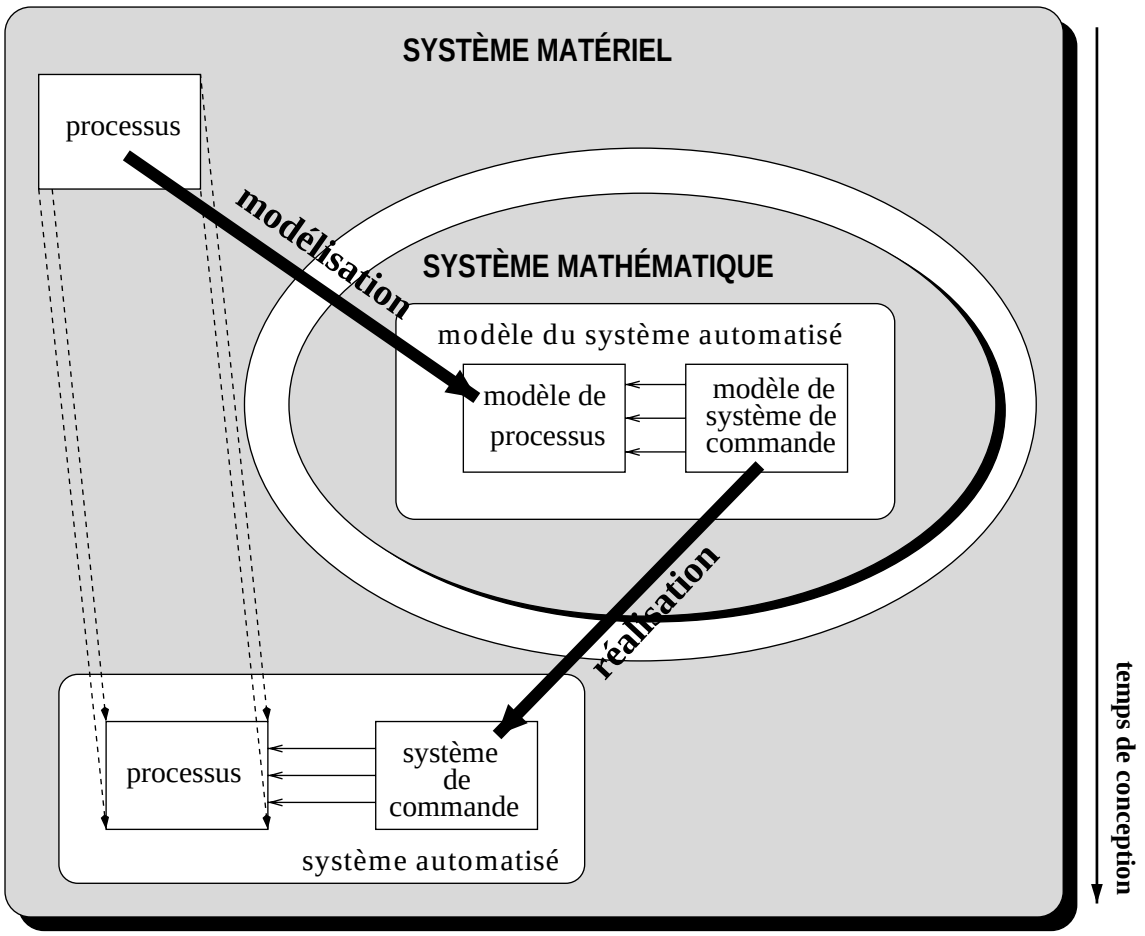


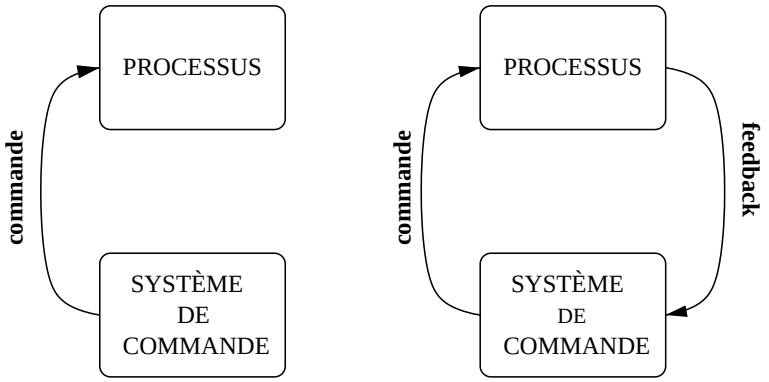
FIG. 1.1 – *Conception d'un système automatisé*

L'entrée d'un système monovariable est appelée *consigne* ; elle représente la valeur que l'on désire qu'un paramètre physique du processus atteigne. La sortie du système est la *commande*; elle représente l'ampleur d'une action physique appliquée au processus pour que le paramètre physique du processus atteigne la valeur de la consigne. Par exemple, un système de régulation de la température d'un four est un système monovariable dont la consigne est la valeur désirée de la température du four. La commande peut être l'intensité du courant électrique à fournir à la résistance chauffante pour que la température du four atteigne le plus vite possible et avec une certaine précision la valeur de température définie par la consigne. Le terme *régulateur* est souvent employé pour désigner un système dont la consigne varie très lentement ou pas du tout, en opposition au terme *asservissement* qui désigne un système pour lequel la consigne varie constamment.

1.2.2 Structure boucle fermée d'une loi de commande

On peut distinguer deux types de système de commande ; lorsque le système "observe" le processus qu'il doit faire évoluer, le système est de type *boucle fermé* (ou *bouclé*), dans le cas contraire, il est de type *boucle ouverte* (ou *non-bouclé*) (Fig. 1.2).

Traditionnellement les lois de commande sont représentées sous la forme de graphes appelés



(a) Boucle ouverte (b) Boucle fermée

FIG. 1.2 – *Structure générale d'un système automatisé*

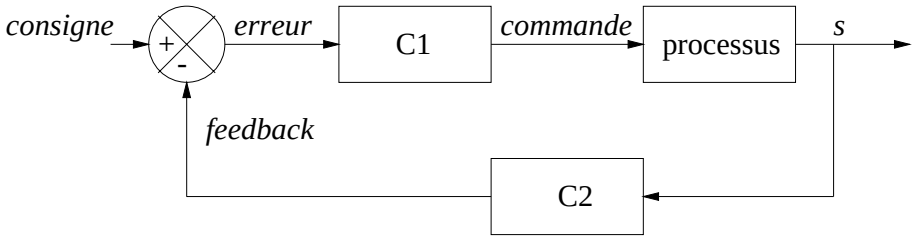
schéma-blocs[28] dans lesquels chaque bloc représente la description mathématique, appelée *fonction de transfert*, d'une transformation appliquée à un signal -suite de valeurs représentant l'évolution dans le temps de l'amplitude d'un phénomène physique- représenté sous la forme d'un arc orienté. La fin (resp. l'origine) d'un arc relié à une fonction de transfert signifie que le signal considéré est une entrée (resp. une sortie) de cette fonction de transfert. A l'aide de cette représentation il devient très facile de visualiser globalement toutes les transformations (gain, filtre, etc.) nécessaires à la commande d'un processus.

La figure 1.3 représente la structure générale des deux types de loi de commande (bouclé, non-bouclé) sous la forme d'un schéma-blocs.

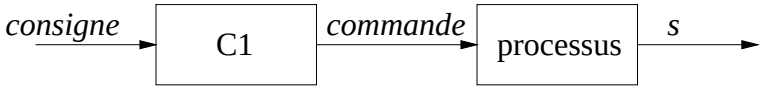
Dans un système de type boucle ouverte, la loi de commande est uniquement fonction de la consigne ($commande = f(consigne)$) alors que dans un système en boucle fermée, elle est fonction de l'écart entre la consigne et la sortie s du processus représentant le phénomène physique que l'on cherche à commander ($commande = f(consigne - s)$).

L'automaticien utilise autant que possible des systèmes bouclés qui autorisent une commande plus fine du processus. Ils permettent, d'une part, de corriger grâce à l'observation de l'évolution du processus l'erreur entre le processus réel et son modèle, ils permettent d'autre part, de prendre en compte l'influence de perturbations externes sur l'évolution du processus.

Une loi de commande de type bouclée (Fig.1.3(a)) est constituée d'un *comparateur* dont la sortie représente l'*erreur* entre la *consigne* (valeur souhaitée pour le phénomène physique que l'on cherche à commander) et le *feedback* (valeur corrigée de la valeur de la sortie *s* du processus), $erreur = consigne - feedback$. La chaîne principale de traitement est appelée *chaîne d'action*; elle relie la sortie du comparateur à la sortie du processus commandé. Elle intègre un bloc $C1$, souvent appelé *correcteur*, chargé de produire la *commande* à appliquer à l'entrée du processus. $C1$ est généralement composé d'éléments de filtrage, de gain, etc. Le bouclage est assuré par la *chaîne de réaction* qui est la chaîne fonctionnelle reliant la sortie s du processus à l'entrée du comparateur. Dans cette chaîne, $C2$ est un élément qui produit le *feedback* à partir de la mesure de s. Il n'est pas toujours possible de mesurer directement la valeur de sortie du processus qu'on cherche à contrôler,



(a) Boucle fermée



(b) Boucle ouverte

FIG. 1.3 – *Structures générales de lois de commande*

dans ce cas le rôle de $C2$ est principalement de réaliser une estimation de cette valeur à partir de la mesure d'un autre paramètre du processus. On peut par exemple estimer une accélération par dérivation de la mesure d'une vitesse, etc. Le rôle de $C2$ est aussi, si besoin est, de mettre en forme, de filtrer, et d'amener le signal s de la mesure dans une plage de dynamique équivalente à celle de la consigne afin de permettre la comparaison nécessaire au calcul de l'erreur [61].

Pour certaines applications, il n'est pas possible de mettre en place une chaîne de réaction, dans ce cas le système de contrôle-commande est un système de type boucle ouverte. C'est le cas par exemple d'une application de transmission d'un signal hertzien. Il n'est pas possible de mettre en œuvre une chaîne de retour permettant de mesurer la qualité du signal reçu et ainsi d'adapter la transmission en fonction des modifications physiques (température, pression, perturbations électromagnétiques) du milieu, puisque cette chaîne de retour serait soumise aux mêmes perturbations. La mesure ne serait donc pas fiable.

Dans la suite de ce document, afin de simplifier le discours, nous supposerons que les systèmes sont bouclés et monovariables, les propriétés auxquelles nous nous intéresserons restant valables pour les systèmes en boucle ouverte et les systèmes multivariables.

1.2.3 Stabilité

1.3 Système de commande à calculateur

Lorsque le processus que l'on cherche à maîtriser est complexe, il est intéressant, voir même nécessaire, d'utiliser un système à calculateur pour mettre en œuvre les lois de commande. Cette solution numérique apporte différents avantages par rapport à une approche analogique [64]:

– **gain en termes de coût et de productivité** : Dans un système à calculateur, les lois de commande sont implantées par des programmes. En phase de mise au point, elles peuvent donc

être rapidement paramétrées et facilement modifiées. Dans un système analogique, les lois de commande sont réalisées par des dispositifs mécaniques et électroniques. Dans le cas le plus favorable, la modification de ces lois implique un changement de composants. Dans le pire des cas, elle peut aller jusqu'à un changement complet de la structure matérielle conduisant à reconcevoir le système.

– **gain de performance** : les lois de commande des systèmes à calculateur profitent naturellement de la puissance de calcul des processeurs. Ainsi celles-ci peuvent être plus complexes que celles utilisées dans les systèmes analogiques. De plus, les systèmes de contrôle-commande numériques sont supérieurs à leurs équivalents analogiques du point de vue de leur immunité au bruit interne et à la dérive de leurs caractéristiques.

1.3.1 Structure d'un système bouclé à calculateur

Généralement, dans les systèmes complexes l'approche utilisée par l'automaticien pour intégrer un calculateur dans la chaîne de traitement automatique consiste à discrétiser le processus [47]. Ainsi, le comparateur et les blocs de traitement $C1$ et $C2$ de notre système bouclé décrit précédemment sont réalisés en numérique par le calculateur. Les interactions calculateur-processus sont concrétisées par des *transducteurs* qui réalisent l'interface entre le système et le processus. Par définition, ce sont des éléments qui permettent de transformer l'ampleur d'un phénomène physique en un phénomène physique d'une autre nature. Dans les systèmes qui nous intéressent on utilise des *capteurs* et des *actionneurs* qui permettent de mesurer l'ampleur d'un phénomène physique en la convertissant en signal électrique. Inversement, les capteurs transforment un signal électrique en un autre phénomène physique. Ainsi les capteurs permettent au système d'observer le processus (acquisition de s) alors que les actionneurs permettent de le modifier (réalisation physique de la $commande$). Un convertisseur numérique analogique (C.N.A) permet de convertir le signal numérique de commande produit par le calculateur $Commande_n$ en un signal analogique $Commande$ appliqué à l'actionneur qui pilote le processus. Le convertisseur analogique numérique (C.A.N) permet de numériser le signal de sortie s issu du capteur (Fig.1.4)[9][20].

Remarque : Le signal de $consigne$ peut être un signal issu d'un calcul ; dans le cadre de la régulation de température d'un four utilisée pour le traitement thermique de pièces métalliques, on peut imaginer qu'une fonction du temps génère une consigne dont le but est d'amener le four à des températures précises selon un cycle préétabli. Le signal de $consigne$ peut aussi être un signal numérisé issu d'un capteur. La température désirée est alors fixée par un opérateur à l'aide d'un potentiomètre.

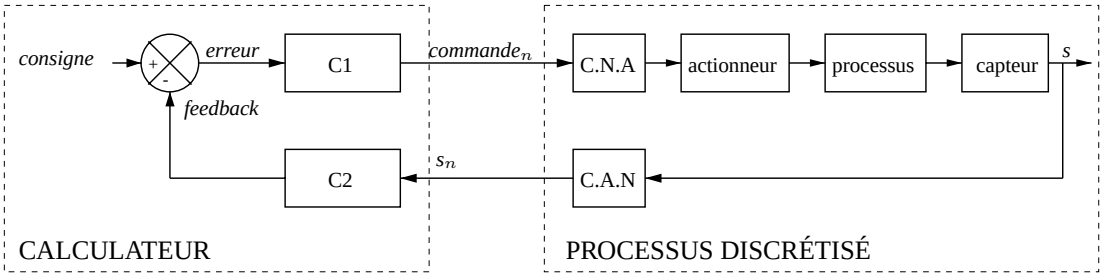


FIG. 1.4 – *Système de commande bouclé à calculateur*

1.3.2 Conséquences de la discrétisation du processus

Dans les systèmes numériques de commande à calculateur les opérations d'échantillonnage et de quantification nécessaires à la discrétisation des signaux analogiques peuvent apporter des erreurs qui tendent à dégrader les performances du système. Ainsi, pour un niveau de performance équivalent, si on veut tenir compte de ces erreurs, il est plus complexe de concevoir des lois de commande pour un système numérique que pour un système analogique. Néanmoins, des outils de calcul formel et de simulation apportent une aide efficace pour résoudre ces problèmes.

1.3.2.1 Choix d'une période d'échantillonnage

Afin de simplifier considérablement les modèles, l'échantillonnage de la sortie s du processus doit être périodique. Entre deux échantillons le processus ne doit pas évoluer de manière incontrôlable, ce qui implique que la période d'échantillonnage T_e de s ne doit pas dépasser certaines valeurs limites maximales admissibles. Pour l'automaticien, il est intéressant de considérer, toujours pour les mêmes raisons de simplification, que tous les calculs nécessaires à la commande sont réalisés entre deux instants d'échantillonnage de s. Ainsi le calculateur produit une valeur de commande pour chaque échantillon de s, le calculateur, le C.A.N et le C.N.A étant cadencés à la même horloge [64].

Le choix de la période d'échantillonnage T_e est essentiel pour garantir le bon comportement du système. Dans de telles conditions d'exécution, plus Te est choisie grande, plus le processeur dispose de temps pour effectuer les calculs qui doivent produire une valeur de commande. Inversement, plus cette période est petite, plus la puissance de calcul nécessaire pour garantir que la commande sera produite avant l'échantillon suivant de s doit être élevée.

La fréquence d'échantillonnage ($Fe = \frac{1}{T_e}$) minimale nous est imposée par le théorème de Shannon : si un signal ne contient pas de fréquences supérieures à F_c alors théoriquement le signal original peut être reconstruit sans distorsion s'il est échantillonné à la fréquence $2F_c$.

En pratique, l'automaticien choisit une fréquence bien supérieure à F_c (souvent de l'ordre de 8 à 10 fois F_c). La lecture de différents ouvrages sur la théorie de l'automatique montre que ce choix est souvent empirique et repose généralement sur le savoir-faire du concepteur du système. Toutefois, on trouve quelques règles théoriques ; elles montrent toutes que le choix de la fréquence d'échantillonnage est lié à la constante de temps ($\tau = \frac{1}{2\pi F_c}$) du processus à commander et que ce choix n'est pas unique puisque ces règles définissent une plage de valeurs acceptables pour Te [19][47][64].

Néanmoins, un compromis est nécessaire entre la puissance de calcul et les performances requises. La meilleure sélection pour la fréquence d'échantillonnage est celle qui est la plus faible (nécessitant donc le moins de puissance) et qui donne un comportement satisfaisant au système selon des critères de précision, stabilité, etc., que l'on souhaite apporter au système.

1.3.2.2 Modèle pseudo-continu de la loi de commande échantillonnée

La modélisation de systèmes discrets est beaucoup moins facile à réaliser que la modélisation des systèmes continus pour lesquels la théorie de l'automatique fournit un ensemble de méthodes et d'outils mathématiques beaucoup plus important. C'est pourquoi, afin de simplifier les calculs, l'au-

tomaticien cherche plutôt à utiliser un modèle *pseudo-continu* de son système plutôt qu'un modèle discret [19][80][20][68].

Le modèle pseudo-continu peut être utilisé à condition que la période d'échantillonnage T_e ne soit pas élevée par rapport à la plus petite des constantes de temps dominantes τ du processus, soit : $T_e < \tau$

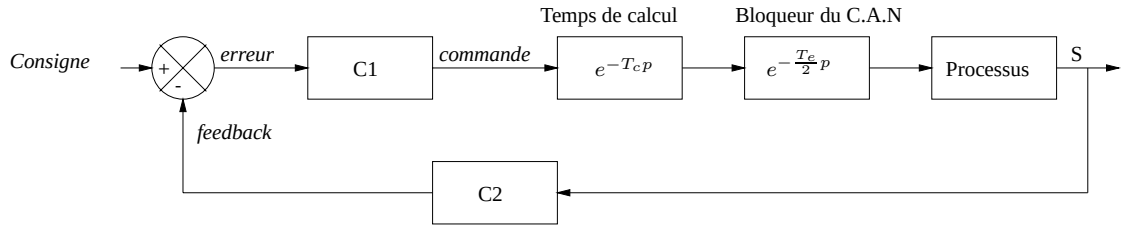


FIG. 1.5 – *Modèle pseudo-continu d'une loi de commande*

Dans ce cas, le convertisseur numérique-analogique qui fournit au processus un signal de commande maintenu pendant une période d'échantillonnage (bloqueur) peut être modélisé par un retard pur $e^{\frac{-T_e}{2}p}$.

De même, le temps de calcul T_c, intervalle de temps borné par l'instant où l'échantillonnage de s est effectué et l'instant où le signal de $commande$ est mis à jour (cf. §1.2.2) peut aussi être modélisé par un retard pur $e^{-T_c p}$.

Le modèle de loi de commande devient alors celui présenté figure 1.5.

1.3.2.3 Influence du temps de calcul sur le comportement du système automatisé

Il peut être montré, pour les systèmes continus, que le *temps de réponse* du système -critère de rapidité d'un asservissement- est proportionnel à la constante de temps T_p donnée par la somme des constantes de temps du processus. Dans un système pseudo-continu il faut remplacer cette constante de temps T_p par une constante $T_{p_e} = T_p + \frac{T_e}{2} + T_c$ prenant en compte les retards introduits par le bloqueur et par le temps de calcul.

Pour une même loi de commande, un système analogique sera donc plus rapide qu'un système échantillonné. Pour approcher la rapidité du système continu avec un système échantillonné, il est nécessaire de choisir des périodes d'échantillonnage les plus faibles possibles et de minimiser les temps de calcul.

1.3.2.4 Influence de la période d'échantillonnage sur le dimensionnement des lois de commande

Pour comprendre comment le choix de la période d'échantillonnage influence le calcul des paramètres d'une loi de commande, nous reprenons ici un exemple tiré d'un ouvrage de Hansruedi Buhler [20]. Cet exemple se rapporte au calcul des coefficients d'un correcteur à action proportionnelle et intégrale (PI) très utilisé pour améliorer la précision et la rapidité de la commande. Les conditions décrites dans cet exemple conduisent au choix suivant pour les paramètres K_i de l'action intégrale et K_p de l'action proportionnelle :

$K_i = \frac{T_e}{2K(T_p + \frac{T_e}{2})}$

$K_p = \frac{T_1 - \frac{T_e}{2}}{2K(T_p + \frac{T_e}{2})}$

, avec T_1 constante de temps dominante du processus

Si on choisi T_e très faible, alors K_i prend aussi des valeurs très faibles. Ainsi pour une faible valeur du signal d'*erreur* (cf. §1.2.2) auquel est appliqué le correcteur, l'action intégrale risque de disparaître à cause de la représentation finie des signaux digitaux. De même, lorsque $T_e \to 0$ alors $K_p \to \frac{T_1}{2KT_p}$. Si T_1 est élevée par rapport à T_p alors K_p peut être très élevée. Une variation d'un bit du signal d'*erreur* peut alors entraîner des variations brutales de l'action proportionnelle.

1.3.2.5 Borne minimale sur la période d'échantillonnage

Il existe au moins deux raisons qui imposent une borne minimale sur la période d'échantillonnage. La première est purement informatique : le calculateur doit avoir le temps de calculer la commande. La seconde est physique : plus la période d'échantillonnage est petite plus les paramètres de gain des lois de commande devront être élevés ce qui a pour effet d'augmenter la valeur maximale du signal de commande et par conséquent l'énergie que doit fournir l'organe de commande [19].

CHAPITRE 2

SYSTÈMES TEMPS RÉEL EMBARQUÉS

Après avoir présenté, dans le chapitre précédent, le point de vue de l'automaticien nous adopterons à partir de maintenant le point de vue de l'informaticien. Nous chercherons autant que possible à établir le lien entre les terminologies rencontrées dans ces deux disciplines.

2.1 Application, système informatique, environnement

Une *application* est composée d'un *système informatique*[1] et d'un *environnement* physique avec lequel il interagit. Le système est composé d'un *calculateur* et d'un ensemble de programmes qu'il doit exécuter appelé *logiciel* (Fig.2.1).

Le système informatique, considéré par l'automaticien comme un simple maillon du système automatisé constitue l'unique centre d'intérêt de l'informaticien. L'environnement est défini comme l'ensemble de tous les éléments physiques qui sont extérieurs au système informatique et qui sont en interaction avec lui. La frontière entre l'environnement et le système informatique est parfois difficile à établir, c'est pourquoi nous avons choisi ici de considérer que l'ensemble des composants physiques qui peuvent être programmés (microprocesseurs, bus de communication, mémoire, interface E/S, etc.) fait partie du système informatique, alors que tous les composants physiques non-programmables constituent l'environnement. Il représente, du point de vue de l'automaticien, l'ensemble des éléments qui constituent le processus discrétisé à l'exception des convertisseurs analogique-numérique et numérique-analogique (cf. figure 1.4).

1. lorsqu'il n'y aura pas de confusion possible nous l'appellerons simplement système

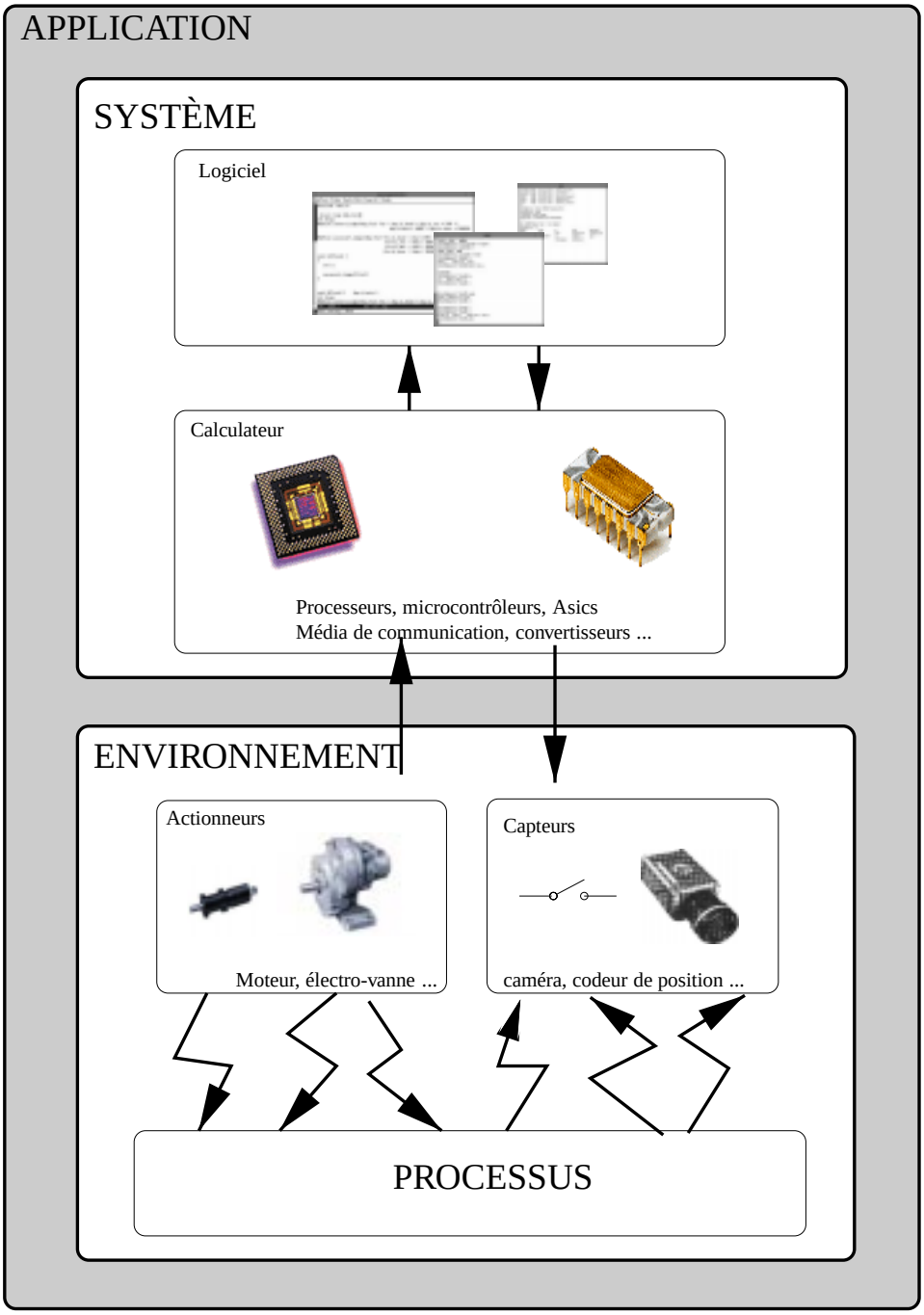


FIG. 2.1 – *Application temps-réel*

2.1.1 Système réactif, système contrôlé

Comme nous l'avons vu dans le chapitre précédent, un système automatisé peut être vu comme un ensemble de sous-systèmes dont la structure, généralement bouclée, est plus ou moins complexe. Cette structure bouclée est constituée d'une chaîne d'action produisant un signal de commande et d'une chaîne de réaction produisant un signal de feedback. La valeur de chacun des signaux issus des capteurs permet de connaître l'évolution du processus. Cet ensemble de valeurs est appelé *état* du processus. Dans le cas d'un système automatique à calculateur, le processus est discrétisé, les signaux issus des capteurs sont donc numérisés par les convertisseurs C.A.N. L'ensemble des valeurs que peut prendre un signal numérique est fini (2^n valeurs possibles avec n = nombre de bits de résolution du convertisseur analogique-numérique). L'ensemble des états possibles dans lesquels peut se trouver un processus discrétisé est donc un ensemble fini.

Le rôle du système informatique est de s'assurer que le processus reste toujours dans les états prédéfinis par l'application. Pour cela, il doit constamment être en interaction avec l'environnement de manière à détecter ses changements d'états (*événements d'entrée*), et à réagir en réalisant des opérations de calcul pour produire un changement d'état (*événements de sortie*) de l'environnement afin de contrôler son comportement. C'est pourquoi ce type de système est appelé *système réactif* [39]. L'environnement est parfois aussi appelé *système contrôlé*.

2.1.2 Exemple de système réactif

Prenons l'exemple d'un distributeur de billets. Il est constitué d'un système informatique qui interagit avec un environnement. L'environnement est ici constitué d'un clavier, d'un lecteur de carte, d'un mécanisme de distribution de billets, d'un écran, d'un stock de billets de banque et des utilisateurs. Ainsi, le clavier et le lecteur de carte permettent de connaître les désirs de l'utilisateur (en quelque sorte l'état de l'utilisateur), l'écran et le tiroir à billets sont les actionneurs permettant de satisfaire les désirs de l'utilisateur suivant la validité de sa carte, de son code et de la disponibilité des billets. Les événements sont produits par les utilisateurs (introduction de carte magnétique, composition d'un code, etc.). Les actions à mettre en œuvre sont l'affichage des informations sur l'écran, la lecture de la carte et la distribution du nombre de billets désirés par l'utilisateur. De ce point de vue, le système informatique qui est chargé de contrôler cet environnement est un système réactif. Il réagit aux événements d'entrée en produisant les actions de sortie adéquates.

2.1.3 Système temps réel

Dans certains systèmes réactifs, la validité d'une action dépend du temps qui s'écoule entre la réalisation de cette action et l'événement qui l'a déclenchée. En d'autres termes, sur ce type de systèmes, il est nécessaire que les réactions du système aux événements soient effectuées en un temps inférieur à une borne maximale. Les applications soumises à ce type de contraintes sont qualifiées de *temps réel* et les bornes maximales sur le temps de réalisation des actions sont appelées *contraintes temps réel*.

" Un système temps réel est un système dont l'exactitude des résultats ne dépend pas seulement de l'exactitude logique des calculs mais aussi de la date à laquelle le résultat est produit. Si les

contraintes temporelles ne sont pas satisfaites, on dit qu'une défaillance système s'est produite."

([4])

Prenons l'exemple d'une application de chronomètre électronique, l'environnement est constitué par des boutons de mise en marche, d'arrêt, de remise à zéro et d'un afficheur. Le système informatique qui contrôle cet environnement est un système temps réel. En effet, l'action "arrêter le comptage du chronomètre" déclenchée par l'événement "stop" (pression de l'utilisateur sur un bouton) doit être accomplie en un temps inférieur à la précision recherchée sur la mesure sous peine d'aboutir à un résultat erroné. Ici, un temps de réaction trop long conduit à un mauvais fonctionnement de l'application.

Par contre, dans l'exemple du distributeur de billets décrit précédemment on peut considérer que le fonctionnement de l'application est correct quel que soit le temps mis par le système pour prendre en compte et satisfaire la demande de l'utilisateur. Néanmoins, l'utilisateur du distributeur risque de s'impatienter si le délai d'attente pour obtenir les billets est trop long. Ce délai peut varier dans de grandes proportions, mais on peut considérer que l'utilisateur est satisfait si ce délai reste très inférieur au temps qu'il aurait à attendre à un guichet pour effectuer la même opération. Ici, la contrainte temps réel que l'on pourrait considérer (délai inférieur au temps d'attente au guichet, c'est-à-dire quelques minutes) est vague et subjective (elle dépend aussi de la patience et de l'humeur de l'utilisateur). C'est pourquoi, ce type d'application pour lequel les contraintes temporelles ne sont pas imposées par l'environnement est plutôt qualifié de *système interactif* plutôt que de système temps réel.

Dans le cadre d'applications temps réel, le non-respect d'une contrainte temporelle peut avoir des conséquences plus ou moins catastrophiques. C'est par exemple le cas dans une application de robot mobile autonome capable de détecter des obstacles ; la détection tardive d'un obstacle se trouvant sur sa trajectoire peut ici aboutir à une collision susceptible d'endommager le robot.

Suivant la nature de l'application, il est possible de tolérer de temps en temps et avec une certaine marge le non respect de certaines contraintes. Ce type de contraintes est appelé *contraintes relatives* en opposition aux contraintes dites *strictes* qui doivent impérativement être respectées.

Toute la difficulté est ici de concevoir un système *prédictible* de manière à pouvoir garantir que les contraintes strictes seront toujours satisfaites et que le dépassement des contraintes relatives restera borné et occasionnel [50][85].

Dans ce document, nous nous plaçons dans le cadre d'applications temps réel pour lesquelles la majorité des contraintes est de type strict.

2.2 Architecture matérielle d'une application temps réel

Nous appelons *architecture matérielle*, l'ensemble des composants physiques qu'il est nécessaire d'ajouter à un processus pour réaliser l'application temps réel. L'architecture matérielle est donc composée d'un calculateur, de capteurs et d'actionneurs.

2.2.1 Calculateur

Le calculateur peut être composé d'un ou plusieurs *microprocesseurs*, de circuits intégrés spécialisés tels que des *ASIC* ou des *FPGA* ainsi que des *médias de communication*.

2.2.1.1 ASIC et FPGA

Les ASIC et les FPGA sont des circuits intégrés que l'on peut configurer pour réaliser à bas coût des fonctions "câblées" simples dont les temps de réaction sont extrêmement courts. La programmation de ces circuits n'est pas évidente car elle nécessite l'utilisation de programmateurs ou de procédures spécialisés. Ces circuits n'intègrent pas de séquenceur d'instructions, ils ne sont donc capables que d'exécuter une seule fonction.

2.2.1.2 Microprocesseurs, microcontrôleurs

Un microprocesseur est composé d'un CPU [2] et d'unités de communication pour communiquer avec des périphériques externes ou d'autres microprocesseurs.

Le *CPU* est une machine séquentielle constituée généralement d'un *séquenceur d'instruction* (SI), d'une *unité de traitement* (UT) et d'une *mémoire*. Les dernières générations de microprocesseurs peuvent aussi intégrer une *unité de calcul flottant* (FPU) dont le but est de considérablement accélérer certains calculs mathématiques (multiplication, division, sinus, arrondi, etc.).

Un *microcontrôleur* est un microprocesseur intégrant un certain nombre d'interfaces supplémentaires (mémoires, timers, PIO [3], décodeurs d'adresse, etc.). Ces nombreuses entrées-sorties garantissent un interfaçage aisé avec un environnement extérieur tout en nécessitant un minimum de circuits périphériques ce qui les rend particulièrement bien adaptés aux applications temps réel embarquées.

Du fait de leur forte intégration en périphérique (certains microcontrôleurs vont jusqu'à intégrer des fonctions spécialisées dans la commande des moteurs), les microcontrôleurs sont souvent moins puissants que les microprocesseurs; le CPU qu'ils intègrent est généralement en retard d'une ou même deux générations.

Dans la suite du document, nous utiliserons le terme de *processeur* pour désigner indifféremment un microprocesseur ou un microcontrôleur. De ce point de vue le processeur est simplement un circuit qui intègre un CPU.

2.2.1.3 Médias de communication

Lorsque le calculateur intègre plusieurs processeurs on dit que l'architecture matérielle du système est *multiprocesseur* ou *parallèle*. Les *médias de communications* sont les éléments qui permettent aux processeurs d'un calculateur multiprocesseur d'échanger des données. On peut classer

2. Central Processing Unit
3. Parallel Input Output

ces médias de communication selon trois catégories principales[90] :

- *média point à point SAM*[4] aussi appelée *lien* : c'est une liaison bidirectionnelle entre deux mémoires SAM de deux processeurs différents ;
- *média multipoint SAM* ou *bus SAM* : c'est une liaison multidirectionnelle qui relie les mémoires SAM de plus de deux processeurs ;
- *média multipoint RAM* ou *bus RAM* : c'est aussi une liaison multidirectionnelle, mais ici contrairement aux médias précédents la communication est réalisée à travers une mémoire commune de type RAM[5]

2.2.2 Transducteurs

Le calculateur interagit avec le processus par l'intermédiaire de transducteurs. Ceux qui permettent d'observer le processus sont appelés *capteurs* et ceux qui permettent d'agir sur le processus sont appelés *actionneurs* (cf. §1.3.1).

2.2.2.1 Capteur

Un actionneur est un dispositif conçu pour mesurer une grandeur physique (température, pression, accélération, etc.) en la convertissant en une tension ou un courant électrique. Le signal électrique issu d'un capteur est un signal continu qui doit être discrétisé pour pouvoir être traîté par le calculateur. Cette discrétisation ou *numérisation* est réalisée par un circuit appelé Convertisseur Analogique-Numérique (C.A.N). Il est utilisé pour *échantillonner* le signal électrique issu du capteur, c'est-à-dire mesurer -le plus souvent à des intervalles réguliers- la valeur de ce signal électrique et ainsi produire une suite de valeurs binaires qui constituent le signal discrétisé ou *signal numérique*. L'opération de codage de la valeur échantillonnée en un nombre binaire codé sur n bits est appelée quantification. La résolution de la quantification et donc la précision de la mesure dépend du nombre de bits utilisés pour coder la valeur (la valeur peut être codée sur 2^n niveaux) (fig. 2.2).

Dans le cas particulier où le codage est réalisé à l'aide d'un seul bit, le capteur est appelé *capteur binaire*, le signal qu'il délivre est un signal *booléen* qui par définition ne peut prendre que deux valeurs (0 ou 1). Ce type de capteur est utilisé pour déterminer l'état d'un phénomène modélisé par deux états exclusifs (exemple : présence/non-présence d'un objet, seuil de température atteint/non-atteint, récipient vide/non-vide, etc.).

2.2.2.2 Actionneur

Le capteur est un dispositif qui convertit un signal électrique en un phénomène physique (moteur, vérin électrique, voyant, haut-parleur, etc.) censé modifier l'état courant du processus. Le signal de commande fourni par le calculateur est un signal numérique qu'il faut convertir en signal électrique

4. Sequential Access Memory. L'accès à ce type de liaison est de type FIFO -First In First Out- pour lequel l'ordre d'émission des données est aussi l'ordre de réception.

5. Random Access Memory. L'accès à ce type de liaison est en général CREW, c'est-à-dire lecture concurrente (CR) et écriture exclusive (EW) : l'écriture d'une donnée précède les lectures.

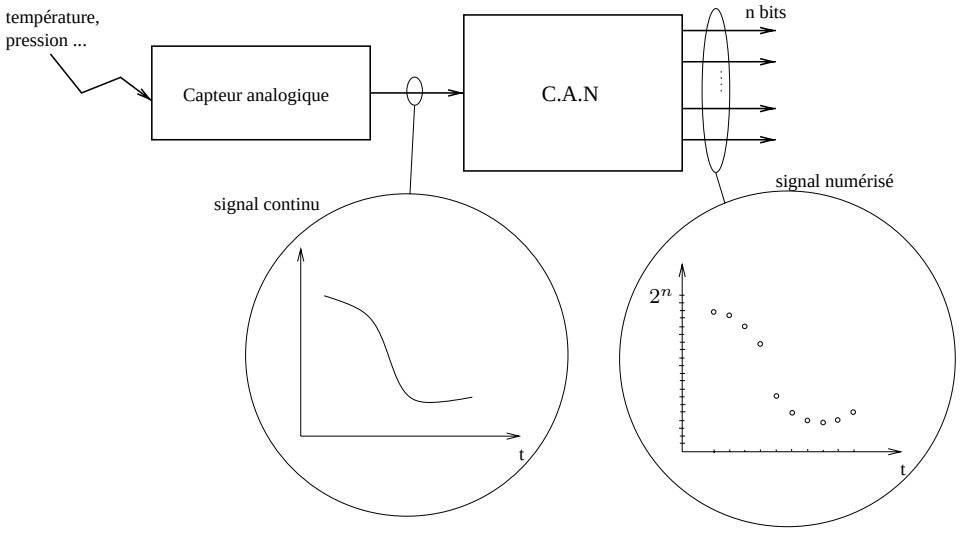


FIG. 2.2 – *Capteur analogique*

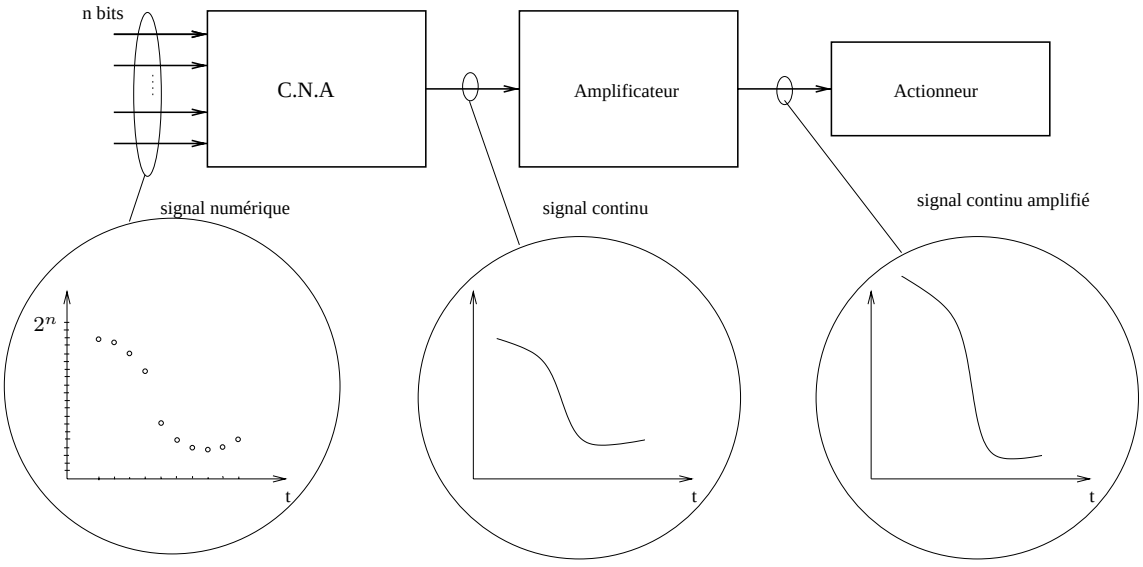


FIG. 2.3 – *Actionneur*

analogique à l'aide d'un Convertisseur Numérique Analogique (C.N.A). Pour reconstruire un signal analogique à partir de la suite des valeurs numériques qui constituent le signal numérique de commande, le C.N.A maintient grâce à un *bloqueur*, la valeur du signal analogique à la même valeur pendant le laps de temps nécessaire au calculateur pour produire la valeur suivante. Le signal obtenu est ensuite lissé par un filtre. Il est à noter que ce signal analogique de commande ne peut être appliqué directement à l'entrée de l'actionneur. Il est nécessaire d'intercaler un amplificateur entre le C.N.A et l'actionneur qui fournira à ce dernier la puissance nécessaire à la création du phénomène physique pour lequel il est conçu.

Certains actionneurs ne nécessitent pas l'utilisation d'un C.N.A car ils peuvent directement être commandés par des signaux numériques, c'est le cas par exemple :

- d'une vanne hydraulique qui peut être commandée par un signal booléen (ouverte/fermée) ;
- d'un moteur à courant continu piloté par un pont de transistors (rôle d'amplificateur de puissance) commandé par un signal booléen dont le rapport cyclique déterminera la tension moyenne appliquée au moteur.

CHAPITRE 3

CONCEPTION DES SYSTÈMES TEMPS RÉEL EMBARQUÉS

3.1 Spécificité des systèmes temps réel embarqués

Lorsque le système temps réel est physiquement intégré à l'environnement qu'il contrôle et qu'il est soumis aux mêmes contraintes physiques (température, pression, etc.) que son environnement, il est dit *embarqué*.

Un système temps réel intégré dans un robot mobile est un système embarqué, alors qu'un système de contrôle de bras de robot n'en est pas un. Dans le premier cas, le système temps réel fait partie intégrale du robot, il se déplace avec lui, il est soumis aux mêmes contraintes physiques externes. Dans le deuxième cas, le système peut être dans une armoire électrique placée dans une pièce différente de la pièce où se situe le robot. Ce système temps réel est donc indépendant du bras manipulateur, il n'est pas intégré à son environnement (bras + objets manipulés).

Les systèmes embarqués sont généralement soumis à des contraintes spécifiques de *coûts* pris au sens large du terme. Ces contraintes sont dues, d'une part, au type d'applications couvertes par ce type de système, d'autre part, à l'intégration du système à son environnement. Ces contraintes particulières de coût sont de plusieurs types : encombrement, consommation d'énergie, prix, etc.

Prenons l'exemple d'un système temps réel embarqué dans un robot dont la mission est d'explorer des canalisations. Ce système sera soumis à des contraintes d'autonomie car il devra intégrer et gérer au mieux sa propre source d'énergie et sera soumis à de fortes contraintes d'encombrement. D'autres applications telles que les applications automobiles nécessitent l'utilisation de systèmes embarqués dont les contraintes sont principalement des contraintes de coût financier (forte compétitivité du secteur commercial), d'encombrement (habitabilité du véhicule) et de consommation d'énergie.

Concevoir un système temps réel embarqué nécessite une bonne maîtrise des coûts matériels mais aussi une bonne maîtrise des coûts de développement car ces derniers représentent une grande

part du coût financier d'une application.

3.2 Maîtrise des coûts matériels

Les performances des systèmes temps réel ne cessent de s'améliorer. Ce gain en performance se traduit par un accroissement de la complexité matérielle : des microprocesseurs de plus en plus puissants doivent être utilisés non seulement pour garantir les contraintes temps réel mais aussi pour piloter efficacement les capteurs et actionneurs dont le nombre tend à croître avec le niveau de performance recherché. Le coût induit par le système temps réel sur le coût d'un système automatisé devient donc de plus en plus important. La consommation électrique et l'encombrement augmentent aussi de manière significative avec le nombre de composants utilisés. La réalisation de tels systèmes impose donc de concevoir un système devant satisfaire des contraintes souvent contradictoires de coût et de puissance.

En effet, de fortes contraintes temps réel imposeront l'utilisation de calculateurs puissants basés sur des processeurs cadencés à des fréquences élevées. Or en micro-électronique, l'utilisation de fréquences élevées est synonyme de consommation électrique plus élevée ce qui va à l'encontre des contraintes de consommation d'énergie. De la même manière, réduire la consommation électrique pourra nécessiter d'utiliser des composants spécifiques à faible consommation qui sont souvent plus coûteux que leur équivalent "forte consommation". C'est le cas des ordinateurs personnels ; à puissance égale un ordinateur portable est beaucoup plus coûteux qu'un ordinateur de bureau.

Réaliser un système temps réel optimal pour toutes ces contraintes relève de l'utopie. Prendre en compte toutes les contraintes en même temps nécessite de réaliser des compromis permettant de définir une solution non optimale mais globalement satisfaisante pour toutes les contraintes.

Un bon moyen de satisfaire globalement toutes les contraintes est de minimiser le nombre de composants nécessaires à l'application. Nous analysons ici brièvement différents moyens d'y arriver :

Limiter le nombre de capteurs Les capteurs sont utilisés pour réaliser soit l'acquisition d'une consigne, soit, dans un système bouclé, pour réaliser le feedback. On pourrait faire l'économie de capteurs en utilisant des systèmes non bouclés, c'est-à-dire sans feedback (cf §.1.2.2), mais ces systèmes sont moins performants et nécessitent un modèle fin du processus. Dans la plupart des cas, l'utilisation d'un système bouclé est obligatoire car il n'est pas possible de réaliser un modèle assez fin du processus. C'est le cas par exemple, d'un système d'asservissement en vitesse d'un véhicule dit "cruise control", pour lequel le conducteur fixe une consigne de vitesse grâce à deux boutons positionnés sur le volant. Le véhicule doit alors maintenir cette vitesse sans l'intervention du conducteur. Le système temps réel chargé de réaliser cette fonction doit, à partir de la consigne, calculer une commande à appliquer au carburateur. Il est ici impossible de définir une relation simple entre le débit d'essence et la vitesse du véhicule, car cette vitesse est fonction de nombreux paramètres tels que la charge du véhicule, le vent, la pente de la route, etc., un capteur de vitesse est donc ici indispensable. Bien que l'utilisation de systèmes en boucle ouverte n'est généralement pas envisageable, il est quand même parfois possible de limiter le nombre de capteurs en les partageant entre plusieurs fonctionnalités que doit assurer le système. Par exemple, dans le cadre d'une automobile, un seul capteur de vitesse pourrait être utilisé pour l'implantation du cruise control et pour la commande efficace d'une boîte de vitesse automatique et d'un système de freinage ABS.

Limiter le nombre d'actionneurs L'actionneur est l'élément qui agit physiquement sur le processus, supprimer un actionneur revient donc à supprimer une fonctionnalité du système, ce n'est donc pas ce que l'on recherche.

Réduire les câblages Le câblage peut représenter une part importante du coût global d'une application. C'était le cas il y a quelques années dans les applications automobiles. Certains véhicules bien équipés pouvaient intégrer jusqu'à quatre ou cinq kilomètres de câble électrique, et plus de 1000 interconnexions [57]. Le système temps réel était alors construit autour d'un calculateur central sur lequel était connecté tous les capteurs et actionneurs. Cette architecture dite centralisée imposait un câblage en étoile pour relier tous ces capteurs et actionneurs au calculateur. On peut minimiser ce câblage en rapprochant le calculateur des actionneurs grâce à l'utilisation d'un calculateur multiprocesseur, dont chacun des microprocesseurs est positionné le plus près possible d'un ensemble de capteurs et actionneurs.

Limiter le nombre de composants du calculateur Il existe au moins deux moyens de réduire le coût induit par le calculateur. Tout d'abord on va chercher à utiliser des microcontrôleurs, plutôt que des microprocesseurs. Pour une puissance de calcul et un ensemble d'interfaces E/S équivalents, un microcontrôleur nécessitera beaucoup moins de composants externes qu'un microprocesseur. On cherchera ensuite à réduire la puissance de calcul nécessaire à l'application qui peut être exprimée sous la forme du rapport (nombre de calculs à exécuter)/(contrainte de temps d'exécution des calculs). Quelle que soit l'architecture matérielle retenue, les contraintes temps réel doivent être satisfaites, la seule manière de diminuer la puissance de calcul nécessaire à la réalisation d'une application donnée consiste donc à réduire le nombre de calculs à exécuter. Il faut donc utiliser au mieux le calculateur en concevant des programmes optimisés.

Le domaine des transports développe un nombre croissant de systèmes temps réel embarqués pour lesquels les contraintes de coût sont particulièrement sévères. Ceci est particulièrement vrai dans le domaine de l'automobile où la forte compétitivité a incité les industriels à rechercher des technologies permettant de construire des systèmes temps réel de plus en plus performants à moindre coût. Ainsi, l'architecture matérielle du système temps réel a évolué en prenant en compte les différentes solutions visant à minimiser le nombre de composants que nous venons d'énumérer. C'est à travers l'exemple significatif de l'automobile que nous présentons ici chronologiquement les différents types d'architecture témoins de cette évolution. Ce domaine est un bon exemple pour illustrer la complexité de conception des systèmes temps réel embarqués car il regroupe a peu près toutes les contraintes que l'on trouve dans d'autres domaines : sécurité (comme pour les applications médicales et avioniques), gros volume de production et sensibilité aux coûts (comme pour les produits de consommation courants et les télécommunications), acquisition de données et contrôle (comme pour le domaine de la robotique), traitement du signal et communications (comme pour le domaine des télécommunications), fortes contraintes temps réel (comme pour le domaine militaire et aéronautique).

3.2.1 Architecture centralisée

La figure 3.1 présente le premier type d'architecture utilisée. Elle est composée d'un calculateur qui peut être monoprocesseur ou multiprocesseur à mémoire partagée et d'un ensemble de capteurs

et actionneurs, tous reliés au calculateur. Ce type d'architecture conduit à un câblage de type "étoile" souvent important et coûteux.

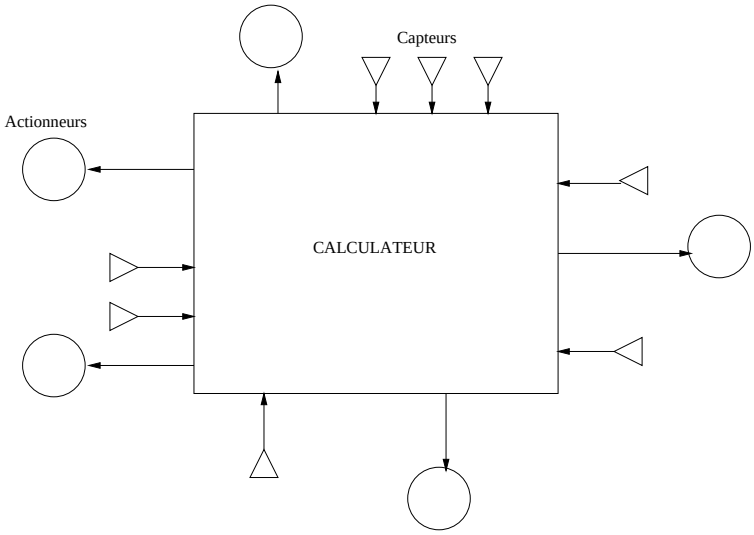


FIG. 3.1 – *Architecture centralisée*

3.2.2 Architecture multi-calculateur

L'architecture multi-calculateur (fig.3.2) est un premier pas dans la recherche du moindre coût. Cette architecture est composée d'un ensemble de calculateurs non reliés entre eux. L'application est morcelée et chaque calculateur implante un ensemble de fonctionnalités. Ici, le principal intérêt est de permettre un rapprochement entre les calculateurs et les transducteurs. La réduction des câblages ainsi obtenue permet non seulement de limiter les coûts, mais aussi d'augmenter la fiabilité du système : si un processeur est défaillant toutes les fonctionnalités ne sont pas hors service, les câbles plus courts sont moins sensibles aux perturbations électromagnétiques externes.

La conception de ce type de système est simplifiée car elle revient à réaliser plusieurs systèmes temps réel plus simples. Le principal inconvénient de cette architecture est de ne pas pouvoir garantir une cohérence globale dans le comportement de l'ensemble de ces systèmes qui ne communiquent pas entre eux.

3.2.3 Architecture faiblement distribuée

Au début des années 80, des bus de terrain ont été développés [57] par les constructeurs automobiles et équipementiers : bus CAN par Bosh, Van par PSA et Renault, J1850 par les constructeurs américains, A-Bus par Volkswagen et K-Bus par BMW. Ces bus bifilaires dédiés aux environnements perturbés tels que les automobiles permettent de relier entre eux les calculateurs. Le bus le plus utilisé actuellement est le bus CAN, il est devenu un standard dans les applications automobiles. Grâce à ces nouveaux bus, sont apparues dans les applications temps réel embarquées des architectures que l'on qualifie ici de "faiblement distribuées" (fig.3.3). Elles sont constituées d'un ensemble de calculateurs reliés entre eux par un bus. Par rapport à l'approche multi-calculateur le

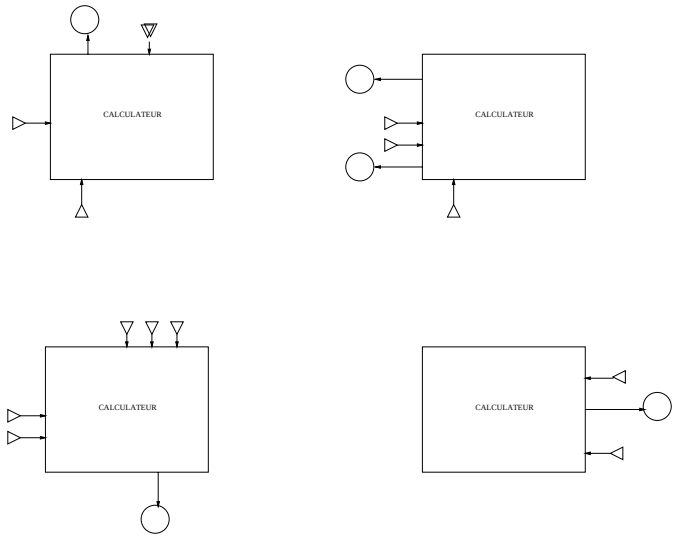


FIG. 3.2 – *Architecture multi-calculateur*

gain est indéniable : il est possible de contrôler le comportement global de l'ensemble de tous les calculateurs et il est possible de partager des capteurs entre les calculateurs.

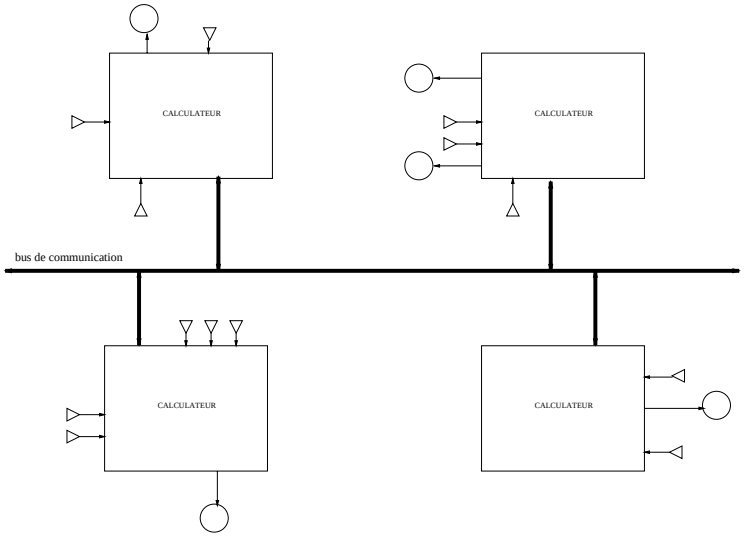


FIG. 3.3 – *Architecture faiblement distribuée*

3.2.4 Architecture fortement distribuée

Cette architecture est la plus aboutie. Les capteurs et actionneurs deviennent intelligents, ils peuvent directement être connectés sur le bus (fig 3.4). Cette approche permet de réduire considérablement tous les câblages, car tous les organes électriques du véhicule peuvent être reliés au bus. C'est aussi l'approche la plus complexe à mettre en œuvre au niveau logiciel. Toute la difficulté consiste à gérer efficacement le multiplexage des données issues des capteurs et des calculateurs sur le bus de telle sorte que les contraintes temporelles de chacun des signaux soient satisfaites (cf.

§4.2).

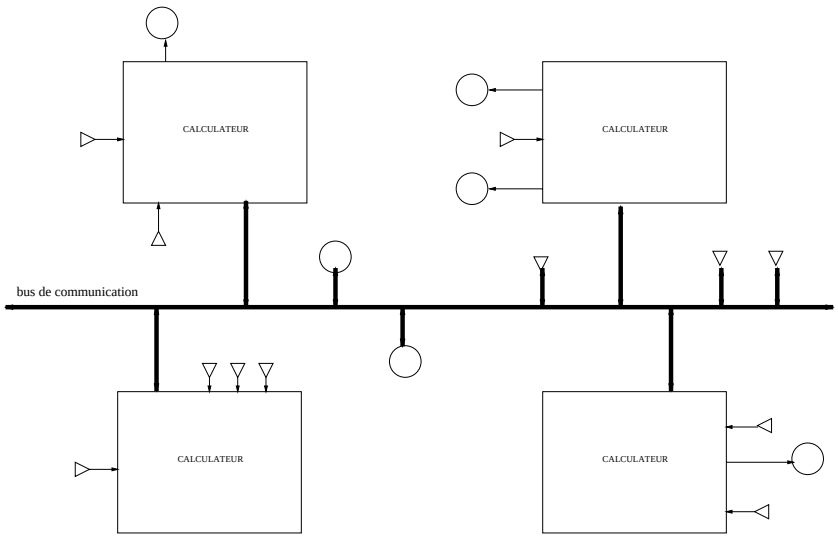


FIG. 3.4 – *Architecture fortement distribuée*

3.2.5 Hétérogénéité

Les architectures temps réel embarqués sont actuellement non seulement fortement distribuées, mais elles sont aussi hétérogènes. Une même application peut intégrer une dizaine de calculateurs. Pour garantir une exécution temps réel des algorithmes il est parfois nécessaire de disposer d'une grande puissance de calcul fournie par exemple par des DSP [1] ou des microprocesseurs performants. Certaines fonctions peuvent aussi être programmées sur des FPGA ou des ASIC. Des microcontrôleurs sont aussi utilisés pour leur capacité à gérer un grand nombre d'entrées et de sorties avec un minimum de composants périphériques. Enfin, une application peut aussi intégrer plusieurs médias de communication différents.

3.3 Maîtrise des coûts liés au développement du logiciel

" ... ce qui est cher n'est plus le composant, mais le temps passé par les ingénieurs à concevoir puis utiliser le système, et éventuellement les erreurs et le temps passé à les corriger."

(Encyclopédie Universalis)

L'intégration de lois de commande complexes et nombreuses, d'algorithmes de traitement du signal et des images conjointement à l'utilisation d'architectures distribuées hétérogènes rend de plus en plus difficile la conception du logiciel. Cette complexité a conduit les ingénieurs à utiliser une méthode de développement en étapes, le cycle en V.

1. Digital Signal Processor

Il est un fait établi que le nombre de lignes de code des applications ne cesse d'augmenter. Le temps consacré par les ingénieurs et techniciens à l'écriture de ces lignes de code et à leur débogage représente la part la plus importante du budget consacré à la réalisation d'une application.

Dans un premier temps, Nous décrivons le cycle en V utilisé par de nombreux ingénieurs. Afin de tenir compte des contraintes de coût de plus en plus fortes, nous présentons ensuite une méthode de réduction de ce cycle aboutissant à des temps de développement considérablement réduits.

3.3.1 Cycle de développement d'un système temps réel

La conception des systèmes temps réel nécessite la résolution d'un grand nombre de problèmes matériels et logiciels. Afin de gérer au mieux cette complexité les méthodes de développement sont généralement basées sur un principe de hiérarchisation permettant de décrire le système comme un ensemble de sous-systèmes plus simples donc plus faciles à concevoir. Ce principe est appliqué dans le cycle de développement le plus utilisé actuellement pour la conception des systèmes temps réel. Ce cycle de développement appelé cycle en V permet selon une hiérarchisation descendante par étape, d'aboutir à la conception détaillée d'une application à partir d'une description abstraite. A chacune des étapes est associée une étape de validation. Le cycle de développement va donc consister à réaliser, une par une, toutes les étapes de la branche descendante, puis à valider chacune de ces étapes en remontant dans la hiérarchie. La validation d'une étape de conception entraîne la remontée vers une étape de validation de niveau hiérarchique supérieur. La non validation d'une étape de conception entraîne la réexécution de cette étape et de toutes les étapes inférieures ainsi que leur revalidation [31][52][67].

Pour la réalisation du logiciel temps réel on peut distinguer trois grandes étapes de conception : la modélisation, la spécification et l'implantation. A ces trois étapes correspondent respectivement trois étapes de validation : les tests unitaires, l'intégration et la validation du système(fig.3.5).

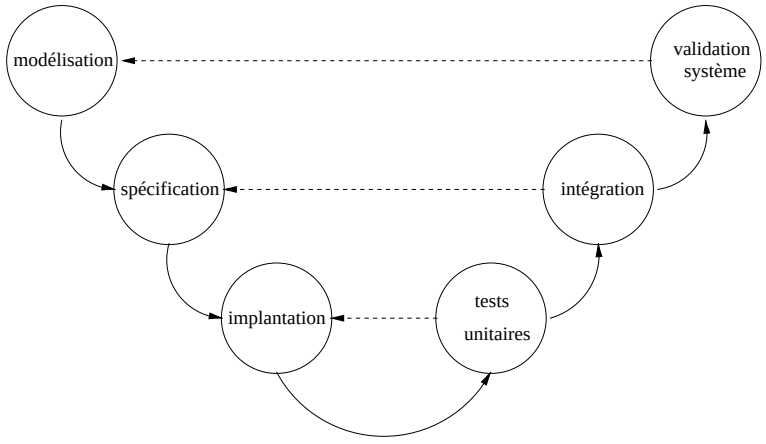


FIG. 3.5 – *Cycle de développement d'un système temps réel*

Modélisation : Cette première étape réalisée par l'automaticien consiste à déterminer et à décrire sous forme d'équations mathématiques, le comportement que devra avoir le système.

Spécification : C'est une description "haut niveau" du logiciel. Elle permet de définir les algorithmes à mettre en œuvre et définir les contraintes temporelles sur les opérations qui constituent les algorithmes.

Implantation : C'est la réalisation proprement dite du logiciel. Elle consiste à écrire les programmes qui doivent implanter la spécification sur l'architecture matérielle.

Tests unitaires : Cette première étape de validation est souvent longue et fastidieuse. Il s'agit ici de déboguer et de s'assurer que chacune des opérations, que les fonctions de communication inter-processeur, etc. fonctionnent correctement. Cette étape est souvent réalisée conjointement avec l'implantation : une opération est souvent testée dès qu'elle a été écrite.

Intégration : Ici, tous les éléments du logiciel sont assemblés. On vérifie alors que le système se comporte comme défini par le modèle. Si ce n'est pas le cas, la spécification est remise en cause, l'implantation est modifiée.

Validation système : Cette étape consiste à vérifier que le comportement de l'application est conforme au cahier des charges. Si cette étape ne valide pas le système, le modèle est remis en cause, il doit être modifié, affiné. Le cycle de développement entier est réexécuté.

3.3.2 Réduction du cycle de développement

Le cycle en V, grâce à la hiérarchisation des problèmes à résoudre, a montré son efficacité dans de nombreuses applications. Néanmoins, la complexité croissante des applications induit un coût de développement élevé. Afin de garantir la compétitivité de telles applications, il est nécessaire de minimiser les temps de développement, c'est-à-dire réduire le cycle de développement en V.

Concevoir un système temps réel en suivant un cycle de développement en V consiste à raffiner un cahier des charges lors d'itérations successives du cycle jusqu'à obtenir le système automatisé recherché (fig. 3.6). Le temps de développement d'une application temps réel est donc proportionnel au temps de réalisation de chaque étape du cycle, mais aussi du nombre de réitérations du cycle.

L'utilisation d'outils de développement issus de méthodes formelles est un bon moyen pour réduire à la fois le nombre de réitérations du cycle en V et le temps de réalisation d'un cycle. En effet, l'intérêt de ces méthodes est de reposer sur des langages de spécification rigoureux fondés sur un ensemble de règles mathématiques qui autorisent des vérifications et de la génération automatique d'un code sain conforme à la spécification.

Les gains que l'on peut espérer de tels méthodes et outils se situent à plusieurs niveaux du cycle de développement :

– la sémantique des langages de spécification des systèmes temps réel tend à se rapprocher le plus possible de la sémantique des outils de modélisation. Ainsi l'étape de spécification est grandement simplifiée, les erreurs dues au passage du modèle à la spécification sont réduites. La validation de la spécification lors de l'étape d'intégration est accélérée car la spécification

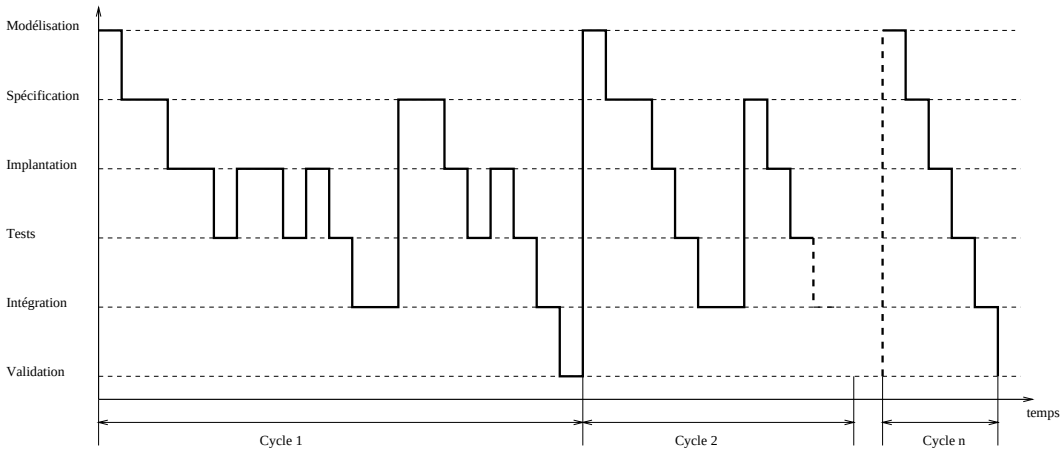


FIG. 3.6 – *Itérations du cycle en V*

converge plus rapidement vers le modèle en nécessitant moins d'itérations du cycle spécification - implantation - tests unitaires - intégration ;

– des outils de vérification permettent de détecter des erreurs de spécification. Ce point est particulièrement important car plus une erreur est détectée tôt moins elle engendrera de réitérations du cycle ;

– la génération automatique de code permet de s'affranchir de l'écriture d'un grand nombre de lignes de code et de réutiliser les développements d'autres applications. On réduit considérablement les tests unitaires. La spécification ayant été vérifiée, l'intégration n'est plus nécessaire ;

– la trçabilité de l'application est améliorée car généralement les méthodes formelles intègrent des outils de génération automatique de documentation, le code bas niveau est souvent simple et commenté ; il est alors très facile de retrouver une erreur sur la spécification à partir d'une erreur sur le code bas niveau.

L'utilisation d'outils de simulation réduit aussi considérablement le nombre d'itérations du cycle en V en permettant de détecter des erreurs éventuelles dans les lois de commande, avant toutes spécification et implantation. La figure 3.7 représente le nouveau cycle de développement réduit obtenu en utilisant ces outils.

Le langage de spécification intègre des méthodes de simulation autorisant leur utilisation lors de la modélisation. Ainsi, les étapes de modélisation et de spécification peuvent être réunies en une seule étape. Les vérifications et les simulations que l'on peut alors réaliser permettent de réduire les erreurs de modèle. On espère ainsi obtenir la validation du système dès la première itération du cycle de développement.

La génération du code est automatique et s'appuie sur un ensemble minimal de librairie de fonctions de très bas niveau portables sur différentes architectures cibles. Les tests unitaires sont à réaliser une seule fois pour chaque type d'architecture. Le coût de développement et de tests unitaires est donc partagé entre toutes les applications qui utilisent la librairie. La génération de code et les librairies peuvent être dans le meilleur des cas certifiées. On apporte ainsi la preuve que l'implantation est conforme à la spécification, qui a elle-même été vérifiée.

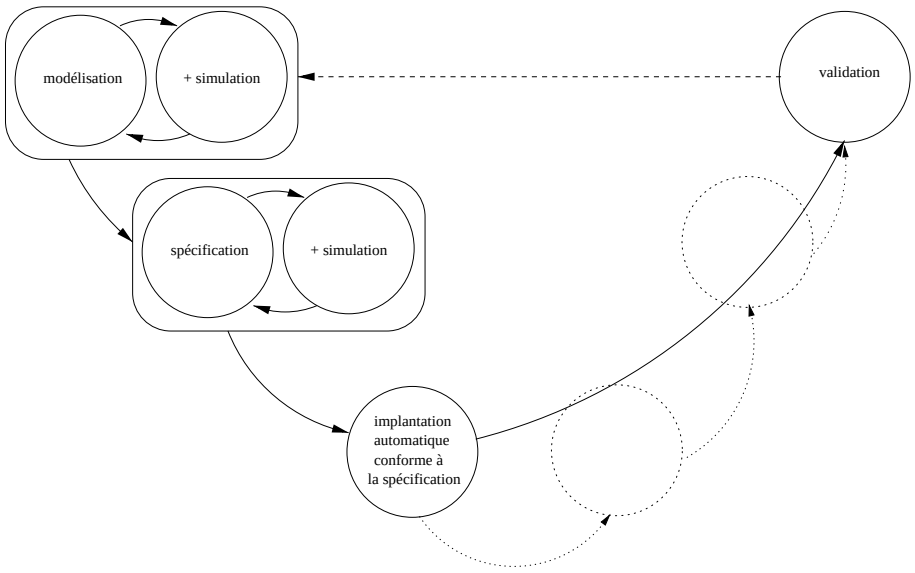


FIG. 3.7 – *Réduction du cycle de développement d'un système temps réel*

Ce nouveau cycle en V réduit doit permettre d'aboutir dans le meilleur des cas à une seule itération du cycle (fig. 3.8) comparable à un cycle de développement en cascade sans remontée [31] pour lequel chaque étape est validée avant de passer à la suivante (fig. 3.9).

L'objectif de cette thèse est de fournir une méthode de conception de systèmes temps réel embarqués se rapprochant le plus possible de ce cycle en V réduit.

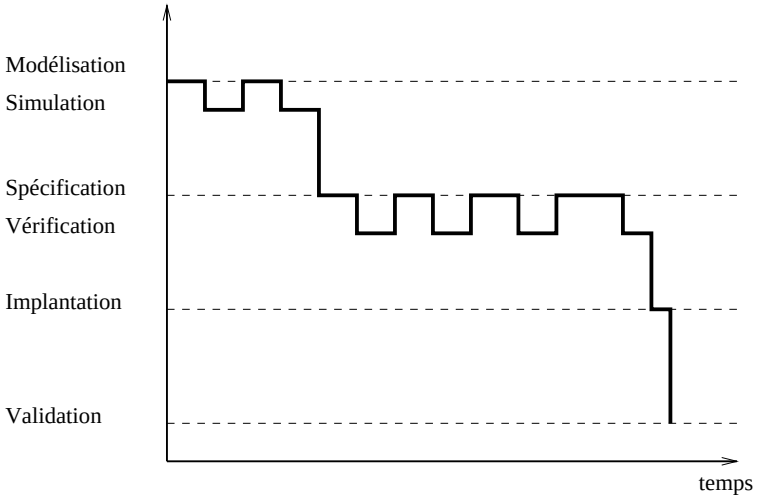


FIG. 3.8 – *Itération du cycle en V réduit idéal*

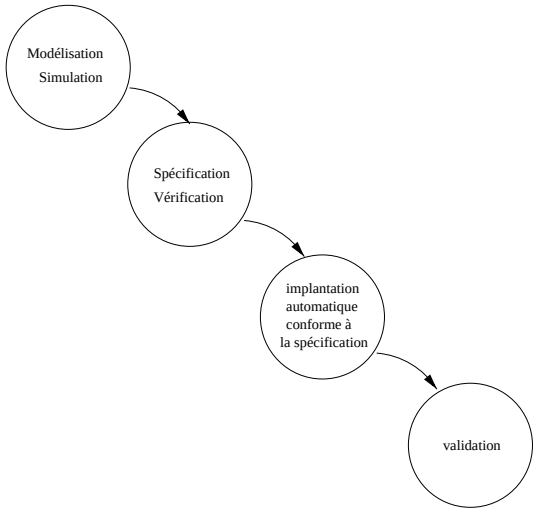


FIG. 3.9 – *Cycle en cascade correspondant au cycle en V réduit idéal*

Deuxième partie

Spécification des applications

Dans le cycle de développement d'un système temps réel, c'est au moment de la spécification qu'une collaboration s'établie entre les automaticiens qui définissent le modèle du système et les informaticiens qui sont chargés de son implantation. C'est l'étape de conception charnière qui consiste à traduire le modèle mathématique en un modèle informatique. La spécification doit être la plus proche possible des outils utilisés par l'automaticien de manière à minimiser les erreurs lors de cette traduction. Le formalisme utilisé doit être clair et simple afin d'être compréhensible à la fois par les automaticiens et par les informaticiens ; il doit être rigoureux et précis pour permettre une description complète du système et ne pas introduire d'ambiguïtés susceptibles de générer plusieurs interprétations possibles lors de la phase d'implantation. La spécification doit être de haut niveau de manière à faire au maximum abstraction des détails de l'implantation et permettre une vue globale du système. En ce sens elle doit être indépendante du langage de programmation et de l'architecture matérielle cible sur laquelle elle sera implantée [62][67].

Dans cette partie, nous recensons différents outils et langages utilisés généralement pour la conception de systèmes temps réel. Nous classons ces approches dans deux catégories ; les approches "classiques", les plus répandues dans l'industrie, et les approches formelles. Nous présentons, au sein des approches formelles, la méthodologie AAA fondée sur la sémantique des langages synchrones conçue pour la spécification de systèmes temps réel distribués. A l'issue de cet état de l'art nous cherchons à déterminer les caractéristiques essentielles que doit posséder un outil de conception des systèmes temps réel distribués embarqués. Enfin, nous proposons une extension de la méthodologie AAA, base de ce travail, afin qu'elle intègre ces caractéristiques.

CHAPITRE 4

BESOINS DE SPÉCIFICATION

La réalisation d'un système temps réel nécessite la conception d'une architecture matérielle composée d'un calculateur, de transducteurs et d'un logiciel qui les exploite (cf.§2.1).

Le cahier des charges décrit à l'aide d'un langage naturel est sujet à différentes interprétations qui dépendent du bagage culturel (formation, expérience, etc.) de chacun. C'est pourquoi la spécification est une étape importante de la réalisation d'un système temps réel car elle consiste à traduire le cahier des charges dans un langage non ambigu : le langage de spécification. Écrire une spécification revient à interpréter le cahier des charges et à fixer les principaux choix de l'architecture matérielle et du logiciel. Ainsi, elle doit décrire au plus haut niveau les aspects logiciels et matériels essentiels du système sous une forme simplifiée sans en donner les détails[67]. Cette spécification sera ensuite raffinée jusqu'à l'obtention d'une architecture matérielle opérationnelle ainsi que d'exécutables implantant le logiciel sur cette architecture matérielle.

Dans cette thèse nous nous intéressons essentiellement à la conception du logiciel dont le calculateur est de type multiprocesseurs (cf.§3.2).

La figure 4.1 montre les différentes étapes du raffinement de la spécification du logiciel : plus on détaille la spécification, plus on se rapproche du code exécutable et donc plus il est nécessaire d'utiliser des langages proches du matériel. La conception du logiciel consiste à traduire une spécification réalisée à l'aide de langages proches du langage naturel de l'homme, dans un langage "compréhensible"par la machine : le code exécutable.

Dans l'esprit de Turing et Post[89], un *algorithme* est une séquence finie de calcul exécutable par une machine à états finie. La notion d'algorithme nous permet de désigner indépendement de tout langage et de toute implantation, l'ensemble des calculs qui constituent le logiciel.

On peut décomposer le logiciel en deux parties : les *algorithmes applicatifs* qui décrivent le comportement du système et l'*exécutif* dont le rôle est d'allouer les ressources matérielles (distribution et ordonnancement) aux *opérations* qui constituent les algorithmes applicatifs, de fournir des services tels que la gestion de la mémoire, les accès aux fichiers, le chargement des programmes, etc. Nous

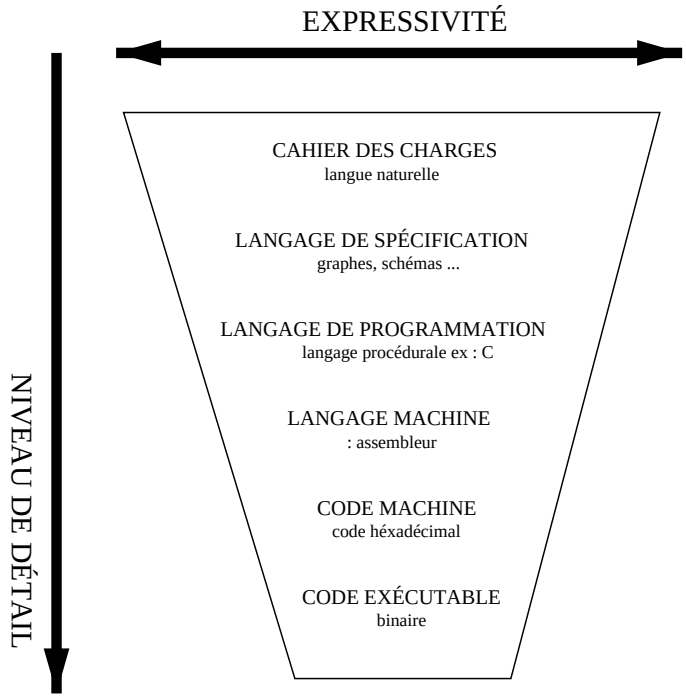


FIG. 4.1 – *Raffinement d'une spécification logicielle*

traitons ici les aspects algorithmes applicatifs[1], les aspects exécutifs et allocations des ressources font l'objet de la partie suivante de ce document.

Dans un système temps réel, les aspects matériel et logiciel sont fortement liés. Pour réaliser l'architecture matérielle du système, il est nécessaire de connaître les algorithmes devant y être implantés afin de la dimensionner correctement (choix des capteurs et actionneurs, précision des convertisseurs, puissance des processeurs, etc.). De même, pour réaliser le logiciel il est nécessaire de connaître non seulement les algorithmes ainsi que les contraintes temporelles sur l'exécution de ces algorithmes, mais il est aussi nécessaire de connaître l'architecture matérielle afin de générer du code exécutable adapté à l'architecture (jeu d'instructions des processeurs, accès capteurs et actionneurs, protocoles des médias de communication, etc.). La spécification est la base servant de point de départ à la conception du matériel et du logiciel. Elle doit donc décrire à un haut niveau les algorithmes, l'architecture matérielle et les contraintes temporelles d'exécution des algorithmes sur le calculateur.

4.1 Spécification des Algorithmes

Un système automatisé est composé d'un processus et d'un système temps réel en perpétuelle interaction avec celui-ci (cf.§2.1). Pour prendre en compte l'aspect infiniment répétitif de ces intéractions (le nombre d'interactions ne peut être borné, il peut donc être considéré comme infini), il est nécessaire d'étendre la notion classique d'algorithme. Ainsi dans le cadre de la conception des applications temps réel, un algorithme est une séquence finie d'opérations, réalisables en un

1. lorsque il n'y aura pas de confusion possible nous les appellerons simplement algorithmes

temps fini sur un support matériel fini, répétée infiniment et dont l'exécution est déclenchée par les événements d'entrée.

Dans un système temps réel complexe deux types d'algorithmes applicatifs distincts sont à prendre en compte : la *commande* et le *contrôle*. Les algorithmes de commande décrivent les actions qui doivent être appliquées au processus. Ils sont constitués des fonctions de transformations des signaux analogiques échantillonnés définies par les lois de commande (correcteurs PID, gain, filtrage numérique, traitement du signal et des images...). Les algorithmes de contrôle déterminent, en fonction des événements d'entrées, quelles sont les actions à appliquer au processus à un instant donné. Ce sont soit des algorithmes de logique séquentielle, soit des algorithmes de logique combinatoire. Les premiers sont principalement utilisés pour réaliser du *changement de modes* de fonctionnement, c'est à dire pour décrire les différentes évolutions possibles de l'application et associer à chacune de ces évolutions un ensemble d'actions à appliquer au processus. Ils sont aussi utilisés pour décrire, dans un mode de fonctionnement donné, le *séquencement et le parallélisme entre les actions*. Enfin, les algorithmes de logique combinatoire sont utilisés pour décrire des *changements de paramètres* internes aux lois de commande qui définissent les actions, afin de prendre en compte les discontinuités et les non-linéarités des modèles qu'elles implantent.

Dans certains systèmes, seul l'aspect commande est présent. Dans d'autres, l'application nécessite simplement la prise en compte de calculs purement événementiels. Généralement ces aspects commande et contrôle sont présents simultanément, c'est pourquoi le système temps réel est souvent appelé *système de contrôle-commande*. La répartition des calculs liés à chacun de ces deux aspects dépend du type d'application envisagée.

4.1.1 Algorithmes de commande

Les algorithmes de commande sont généralement constitués d'un ensemble de fonctions généralement issues de la théorie du signal, des images ou de l'automatique et dont la finalité est de réaliser des transformations sur des signaux. Les calculs impliqués dans ces algorithmes peuvent être très simples (addition et soustraction de nombres entiers) aussi bien que très complexes (fonctions trigonométriques, logarithme, transformée de Fourrier appliquées à des nombres en représentation à virgule flottante). Quelle que soit cette complexité, les algorithmes de commande nécessite souvent la mémorisation du passé de certains signaux. C'est par exemple le cas de la sortie d'un filtre numérique défini par une fonction de la forme :

$$y_n = Ax_n + \sum_{i=0}^{m} B_i y_{n-i} \quad \text{avec} \quad A, B_i \in I\!R \ \text{ et } \ n > m \geq 0 \ \in I\!N,$$

ou bien encore d'un intégrateur de la forme :

$$y_{n+1} = \frac{x_{n+1} - x_n}{t_{n+1} - t_n}$$

avec t_{n+1} et t_n dates respectives d'échantillonnage de x_{n+1} et x_n.

4.1.2 Algorithmes de contrôle

Le contrôle est nécessaire pour répondre principalement à trois besoins : le changement de paramètres de certaines lois de commande, le séquencement et le parallélisme entre actions et le chan-

gement de modes de fonctionnement.

4.1.2.1 Modification de paramètres

Tout au long de la vie d'une application temps réel, l'état du processus évolue. Dans certaines applications, il est nécessaire d'adapter les paramètres de certaines lois de commande en fonction de l'état dans lequel se trouve le processus. C'est par exemple le cas lorsque l'on cherche à commander un processus correspondant à un phénomène physique complexe dont les caractéristiques nécessitent d'être linéarisées pour pouvoir être modélisé. En fait, lors de la modélisation le processus est décrit sous forme d'équations mathématiques. Le modèle réalisé est un ensemble de fonctions dont les entrées reflètent des actions physiques que l'on peut appliquer sur le processus et les sorties représentent l'évolution des propriétés physiques du processus lorsqu'il est soumis à ces actions (cf. §1.1.2). Pour certains phénomènes physiques il est difficile d'identifier cette fonction. L'automaticien cherche à approcher le phénomène physique par une ou plusieurs fonctions simples, c'est la linéarisation. Modéliser de tels phénomènes physiques revient dans ce cas à réaliser un ensemble de modèles valides chacun sur une plage d'évolution du phénomène.

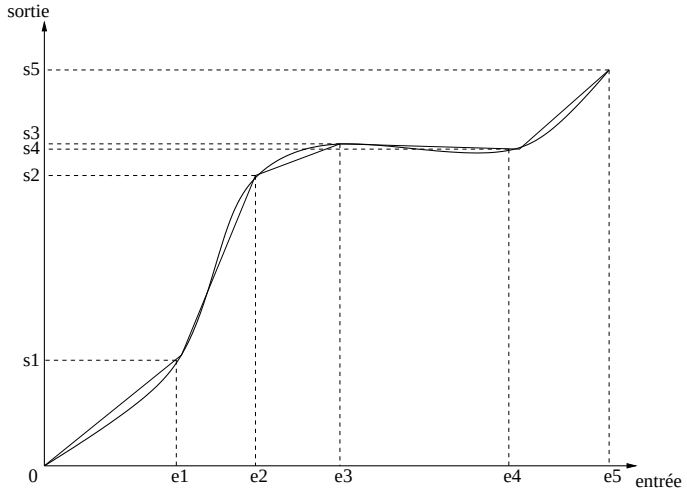


FIG. 4.2 – *Linéarisation*

A chacun des modèles linéaires ainsi définis, il est alors nécessaire d'associer une loi de commande, chaque loi de commande étant valide pour une plage de fonctionnement du système. L'exemple présenté par la figure 4.2 montre la fonction de transfert recherchée pour la loi de commande d'un phénomène physique complexe linéarisé par un ensemble de cinq fonctions affines $y = ax + b$. Lors du fonctionnement du système, la partie contrôle se chargera de sélectionner les coefficients a et b adaptés à la plage de fonctionnement dans laquelle le système se trouve. La loi de commande présentée figure 12.3 dans la partie application de ce document illustre aussi parfaitement ce concept.

4.1.2.2 Séquencement

Dans certaines applications il est nécessaire de séquencer des actions afin d'amener le processus d'un état à un autre en passant par d'autres états bien spécifiques selon une séquence déterminée.

C'est par exemple le cas pour l'automatisation d'une rame de métro. Le démarrage de la rame peut être vu comme le séquencement de plusieurs actions : fermer les portes des wagons, vérifier que toutes les portes sont fermées, desserrer les freins et enfin commander le moteur du wagon tracteur.

4.1.2.3 Changement de modes

Comme pour le séquencement, le changement de mode de fonctionnement sous-entend généralement un changement de comportement du système de contrôle-commande. Le séquencement d'actions répond à un besoin d'atteindre un ensemble de buts successifs. Le changement de mode répond à un besoin d'adapter le comportement du système à l'évolution du processus. Les changements de mode ne sont pas planifiés, ils dépendent de l'évolution de l'environnement. Le programmeur doit tout de même définir les règles d'évolution d'un mode à un autre, c'est-à-dire quels sont les événements qui conditionnent le changement de modes et les différents modes autorisés.

4.2 Spécification des contraintes temporelles

4.2.1 Contrainte de cadence

Par définition, un système temps réel est un système informatique dont le logiciel est soumis à des contraintes spécifiques de temps, les contraintes temps réel. Il est nécessaire de pouvoir spécifier ces contraintes. Leur prise en compte à l'implantation influe sur l'ordre d'exécution des calculs qui constituent le logiciel que doit exécuter le calculateur[2].

Nous avons vu dans la première partie de ce document que pour un système bouclé élémentaire, la période d'échantillonnage T_e admissible dépend fortement de la nature du processus et en particulier de son comportement dynamique (cf.§1.3.2.1). De plus, le processus peut être décomposé en un ensemble de sous-processus possédant chacun ses propres caractéristiques dynamiques.

Pour un système temps réel, une contrainte sur la période d'échantillonnage permet de fixer l'intervalle de temps minimum qui sépare l'arrivée de deux événements d'entrée. Cette contrainte permet d'imposer au système réactif un rythme minimal d'entrée des événements (cf.§2.1.1). Le système réactif doit être capable de traiter tous les événements dont l'intervalle entre les dates d'arrivée est au moins égale à la période d'échantillonnage. Ce type de contrainte est appelée *contrainte de cadence* (fig. 4.3).

A chacun des sous-processus qui composent le processus on associe un algorithme de contrôle-commande. L'exécution de chacun de ces sous-algorithmes doit respecter sa propre contrainte de cadence correspondant à la fréquence d'échantillonnage définie pour ses entrées (feedback et consigne). Ainsi dans les systèmes complexes il est généralement nécessaire de pouvoir spécifier plusieurs contraintes de cadence.

4.2.2 Contrainte de latence

Les opérations d'échantillonnage et de blocage peuvent être modélisées par un simple retard et les algorithmes de commande peuvent être modélisés par une fonction de transfert à laquelle est

2. Cette notion d'ordonnancement sera reprise et détaillée dans la partie implantation de ce document.

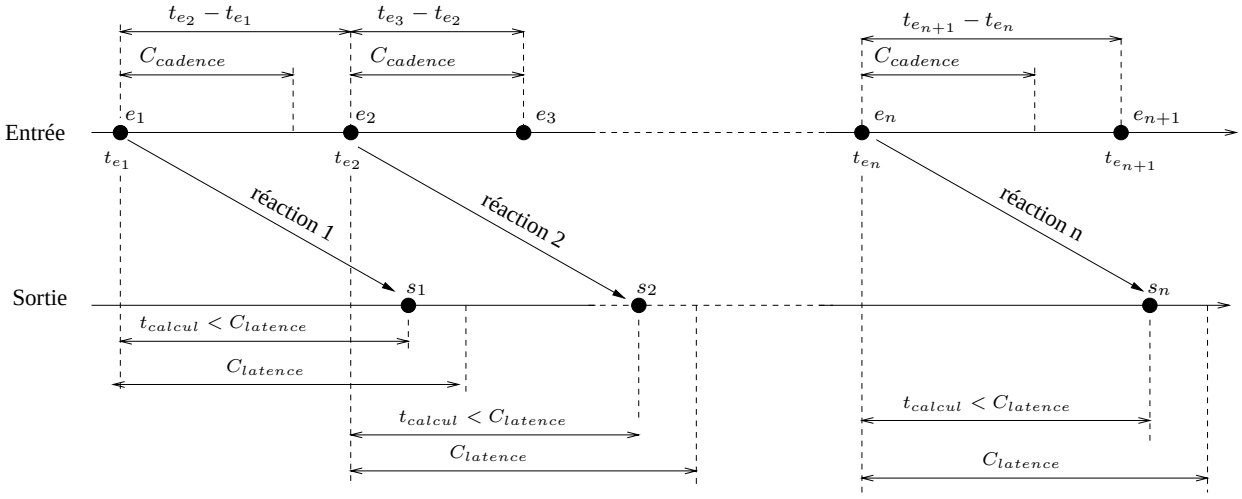


FIG. 4.3 – *Latence*

associée un retard pur équivalent au temps de calcul de l'algorithme (cf. §1.3.2.3). La somme de ces deux retards est le temps de réponse du système temps réel. Pour garantir que ce temps de réponse ne dépassera jamais une certaine valeur critique à partir de laquelle les performances du système ne seraient plus satisfaisantes ou à partir de laquelle le système serait instable, il est nécessaire de borner le temps de calcul. Cette borne est appelée *contrainte de latence*. De manière générale, elle correspond à l'intervalle de temps maximum pendant lequel le système réactif doit produire les actions engendrées par les événements d'entrée (fig. 4.3).

4.3 Spécification des contraintes matérielles

La spécification du logiciel ne peut être complètement indépendante de l'architecture matérielle car les lois de commande ont été définies pour certains capteurs et certains actionneurs. L'utilisation d'une architecture disposant d'autres capteurs et d'autres actionneurs entraînerait obligatoirement une modification des lois de commande et donc des algorithmes. Dans le cadre d'un cycle de développement en V réduit (cf. 3.3.2) il est nécessaire de spécifier à la fois les algorithmes et l'architecture de manière à permettre une implantation automatique des algorithmes sur l'architecture. Lors de cette implantation sur une architecture distribuée les opérations de l'algorithme doivent être affectées à un processeur, c'est la *distribution.*

La distribution des opérations de gestion des capteurs et actionneurs ne peut être arbitraire car chaque capteur et chaque actionneur sont gérés par un microprocesseur. Il est donc nécessaire de spécifier ce lien afin que l'opération de gestion de chaque capteur et de chaque actionneur soit affectée au processeur sur lequel le capteur et l'actionneur correspondant sont reliés. De plus, pour pouvoir générer les communications inter-processeur, il est nécessaire de connaître la topologie du réseau du calculateur, c'est-à-dire le type de média de communications (lien, bus SAM, bus RAM) ainsi que les connexions des processeurs à ces médias. Enfin, pour pouvoir générer un code exécutable optimisé, il est nécessaire de connaître les principales caractéristiques des composants utilisés (temps de calcul des opérations sur les processeurs, taux de transfert sur les médias de

communication, etc.).

CHAPITRE 5

MÉTHODES ET OUTILS DE SPÉCIFICATION EXISTANTS

La programmation des systèmes temps réel est un domaine de l'informatique en forte expansion. On peut expliquer ce phénomène, d'une part, par le nombre et la complexité croissants des applications et, d'autre part, par la jeunesse de cette science. De nombreux laboratoires de recherche ede nombreuses entreprises s'emploient activement à combler les lacunes des méthodes de conception des systèmes temps réel. C'est pourquoi durant ces dernières années un grand nombre d'outils d'aide à la spécification est apparu sur le marché. Néanmoins, ceux-ci ne répondent pas à tous les besoins.

Dans ce chapitre, nous décrivons tout d'abord les caractéristiques principales des langages de spécification sur lesquels sont basés les outils de conception des systèmes temps réel. Nous présentons ensuite un état de l'art non exhaustif des outils existants cherchant à dégager les principales approches de conception en vogue actuellement.

Nous avons choisi de découper cet état de l'art afin de séparer les deux principaux types d'outils : la première partie est consacrée aux outils dits "classiques" basés sur une approche non-formelle de la spécification, la deuxième partie est consacrée aux outils utilisant des langages de spécification formels caractérisés par une sémantique mathématique rigoureuse permettant de démontrer les propriétés du système ainsi spécifié. Les vérifications sont très importantes lorsqu'on cherche à réduire les coûts liés au temps de développement car elles permettent de détecter le plus tôt possible dans le cycle de développement les erreurs de conception - elles sont d'autant plus coûteuses qu'elles sont découvertes tardivement - et ainsi réduire considérablement les phases de tests et de débogage. Ce sont ces approches qui doivent être mises en œuvre dans le cadre de la stratégie de réduction du cycle de développement en V présentée dans la première partie de ce document. Néanmoins, ces approches issues de la recherche sont récentes et les systèmes temps réel conçus à l'aide d'une approche classique sont encore largement majoritaires. Nous nous devons donc de présenter ici ces deux types d'approche [52].

Les outils de conception reposent généralement sur des langages de programmation de haut ni-

veau orientés métiers. Il n'existe pas d'outil universel permettant de décrire avec une même efficacité tous les aspects de la programmation temps réel (contrôle, commande, parallélisme, contraintes temporelles). Chaque outil à ses propres spécificités lui conférant une aptitude à décrire une certaine classe de problème. A l'issue de l'état de l'art, nous tentons donc de définir, dans les grandes lignes, l'outil idéal basé sur une approche formelle permettant de répondre aux attentes des ingénieurs chargés de concevoir des systèmes temps réels embarqués.

5.1 Caractéristiques des langages de spécification

Nous avons déjà évoqué le fait qu'il existe deux grandes familles de langages : les langages formels et les autres. Qu'un langage de spécification soit formel ou non, il possède au moins deux caractéristiques importantes dans le cadre de la spécification des applications temps réel embarquées.

La première reflète la manière dont il permet de décrire les algorithmes. On peut ainsi classer les langages en quatre catégories : *impératif, déclaratif, logique* et *objet*[6].

La *programmation logique* est très différente des autres types de programmation, son but est de permettre la résolution d'expressions logiques. Le langage le plus connu basé sur ce type de programmation est Prolog. Ce type de langage est peu adapté pour décrire les algorithmes mis en jeu dans les applications temps réel. La *programmation objet* consiste à définir des classes et des méthodes applicables à chaque classe. L'exécution des programmes définis sur ce mode consiste à créer des instances d'objets - basés sur les classes - qui interagissent par des appels aux méthodes. Les langages objets récents les plus connus sont Smaltalk et C++. La sémantique des langages objets autorise parfois des pratiques de programmation dangereuses. Il existe des langages objets adaptés à la spécification des systèmes temps réel, néanmoins les langages objets utilisent des techniques d'allocation mémoire dynamique qui rendent les implantations peu efficaces. Nous considérons donc que ces langages sont peu adaptés à la conception d'applications temps réel embarquées pour lesquelles le déterminisme, la sécurité et l'optimisation sont des critères de conception très importants. Pour ces raisons, nous présentons donc ici essentiellement les différences entre les langages dits impératifs et les langages dits déclaratifs.

La deuxième caractéristique importante est la manière dont ces langages intègrent la notion de temps. Selon ce critère on peut classer les langages en deux grandes familles : les langages synchrones et les langages asynchrones.

5.1.1 Impératif versus déclaratif

Que ce soit avec un langage impératif ou avec un langage déclaratif il est toujours possible de décrire l'algorithme que l'on cherche à implanter sur le calculateur, mais les différences de sémantique qui existent entre ces types de langages leur confèrent une expressivité différente. Certaines classes de problèmes seront plus facilement décrites avec un langage impératif qu'avec un langage déclaratif et vice-versa.

5.1.1.1 Langages impératifs, flot de contrôle

Les langages impératifs sont basés sur la notion d'état du système représenté par les valeurs associées aux variables du programme. Un langage textuel impératif peut être représenté graphiquement sous la forme d'un graphe flot de contrôle. Dans ce type de programmation on identifie des blocs constitués d'une suite d'instructions. Le point d'entrée d'un bloc représente son début, le point de sortie d'un bloc représente sa fin. On exprime une relation d'ordre d'exécution entre blocs en connectant la sortie d'un bloc à l'entrée d'autres blocs. Il est possible d'étiqueter ces arcs de manière à exprimer des conditions sur "l'enchaînement d'exécution des blocs". L'exécution d'un bloc provoque un changement d'état du système par l'affectation de valeurs aux variables du programme. Cette relation d'ordre impose un ordre total d'exécution à tous les blocs du graphe.

Il est possible de faire une analogie entre les variables d'un programme et les paramètres physiques d'un système, et l'analogie entre l'état d'un programme reflétant la valeur de l'ensemble de ces variables et l'état d'un système reflétant la valeur de ces paramètres physiques (cf.§2.1.1). On peut donc associer un bloc d'un graphe flot de contrôle à un état particulier d'un système et on peut associer aux arcs du graphe une condition reflétant une modification de certains paramètres du système. Ainsi, un graphe flot de contrôle permet de décrire aisément l'évolution possible du système (suite des états par lesquels doit "passer" le système pour aboutir à un état recherché). Un graphe flot de contrôle, et plus généralement un langage impératif, est bien adapté à la description des aspects contrôle d'un système temps réel [6]. Par contre l'ordre total qu'il impose sur l'exécution de tous les blocs du graphe rend difficile l'exploitation du parallélisme présent dans les algorithmes (parallélisme potentiel).

5.1.1.2 Langages déclaratifs, flot de données

Dans les langages déclaratifs[1], il n'existe ni notion de variable ni notion d'état. Ce type de programmation est basée sur la notion de fonction au sens mathématique du terme.

La forme graphique d'un langage déclaratif est appelé graphe *flot de données*. Dans un graphe flot de données une fonction est représentée par un bloc dont les entrées sont les arguments et les sorties sont les résultats du calcul de la fonction sur ses arguments. Les arcs représentent des dépendances de données entre les fonctions et donc des transferts de données entre les blocs. L'exécution de chaque bloc est répétitive et déclenchée par l'arrivée de données d'entrée, d'où la notion de *flots* définissant une séquence ordonnée de données. Lorsqu'un bloc a besoin lors de sa n-ième exécution de consommer une donnée produite lors de la $(n-1)$ième exécution d'un autre bloc, il faut intercaler entre ces deux blocs un bloc appelée "retard", qui consomme une donnée sur son arc d'entrée après avoir produit sur son arc de sortie la donnée lue sur son arc d'entrée lors de son exécution précédente. Un graphe flot de données n'impose qu'un ordre partiel de type "s'exécute avant" entre les blocs.

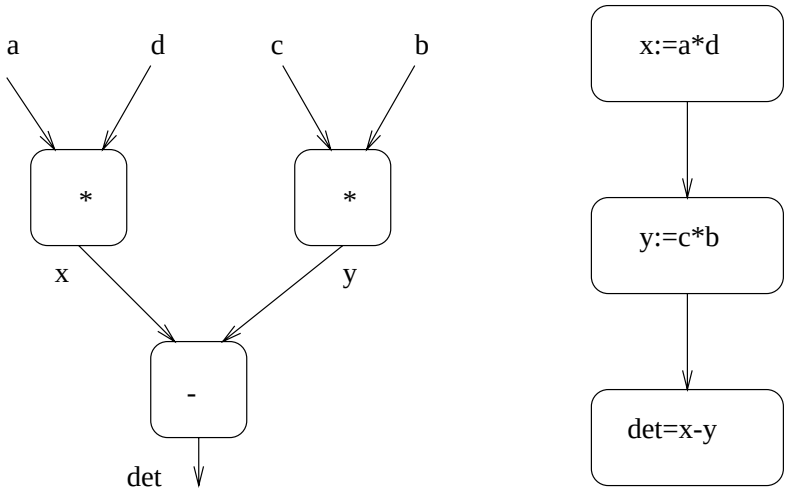
Un graphe flot de données permet de décrire facilement la composition d'un ensemble de fonctions qui appliquées à une séquence d'événements (flot) permet de délivrer le résultat recherché. Ce type de langage est donc bien adapté pour décrire par exemple les équations différentielles discrétisées du système de commande car, d'une part celui-ci est modélisé par l'automaticien comme la composition d'un ensemble de sous-processus, et d'autre part les algorithmes de commande prenant en compte le passé des signaux (exemple du filtre récursif) peuvent être facilement décrits grâce à l'utilisation de blocs "retard". Enfin, les signaux issus des capteurs forment après discrétisation des

1. appelés aussi langages fonctionnels

flots de données. Chacune des valeurs de ces flots est un événement d'entrée déclenchant les calculs qui produisent des événements définissant des flots de sortie, ce qui correspond bien au but du graphe flot de donné : décrire des transformations sur des flots d'entrée pour produire des flots de sortie. L'intérêt principal d'un graphe flot de données, grâce à l'ordre partiel qu'il induit sur l'ordre d'exécution des blocs, est, contrairement au graphe flot de contrôle, de rendre explicite le parallélisme potentiel des algorithmes. Ce type de graphe est donc bien adapté pour l'implantation distribuée d'algorithmes.

5.1.1.3 Exemple

La figure 5.1 représente l'expression d'un déterminant d'une matrice $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$, avec $det\,[A] = ad - bc$ selon une spécification flot de données (fig.5.1(a)) ou flot de contrôle (fig.5.1(b)).



(a) graphe flot de données

(b) graphe flot de contrôle

FIG. 5.1 – *Expression du déterminant d'une matrice 2-2*

5.1.2 Synchrone versus Asynchrone

Afin de répondre aux besoins de déterminisme du comportement et de sûreté de conception des systèmes temps réel, une nouvelle famille de langages a été développée : les *langages synchrones*. Les langages synchrones se distinguent principalement par une modélisation différente du temps qui conduit, non plus à s'intéresser comme dans les langages asynchrones à la durée des événements et des réactions, mais seulement à leur ordre d'apparition et d'exécution. Ce modèle conduit à une spécification et à un schéma d'implantation plus simple donc plus facile à comprendre, à vérifier et à optimiser [32].

5.1.2.1 Langages Synchrones

Dans les langages synchrones le temps est modélisé comme une suite d'instants logiques. L'hypothèse synchrone consiste à considérer que les actions du système temps réel sont produites dans le même instant que les événements qui les ont déclenchées. Cette hypothèse permet de définir un temps logique où il est fait abstraction des temps d'exécution, ce qui rend la spécification du logiciel indépendante de l'architecture matérielle. Ainsi, dans les langages synchrones, on ne s'intéresse qu'à l'ordre des événements d'entrée et de sortie. La sémantique de ces langages est basée sur des modèles mathématiques rigoureux. La compilation des programmes produit un automate sur lequel on peut effectuer des vérifications, par exemple de vivacité (l'automate ne comporte pas d'état dans lequel il ne pourrait sortir), d'atteignabilité (un état peut être ou ne pas être atteint à partir d'un autre état), etc., reflétant différentes propriétés du système (il n'existe pas d'interblocage, un événement peut ou ne peut se produire, etc.). L'automate ainsi vérifié peut être traduit automatiquement en code exécutable séquentiel (C, assembleur, FORTRAN ...) [13][56][55].

Le modèle synchrone, surprenant au premier abord, est une transposition à l'informatique du modèle utilisé par l'électronicien lors de la conception des circuits numériques : l'établissement des potentiels électriques dans les circuits est considéré comme non observable à l'échelle de l'horloge du système, le basculement de toutes les portes logiques est considéré comme instantané aux instants définis par l'horloge.

Ce modèle conduit principalement à deux notions :

- la *simultanéité* exprimant le fait que deux événements sont présents en même temps ;
- l'*atomicité* des réactions exprimant le fait que les actions sont produites dans le même instant que l'apparition des événements qui les ont déclenchées ;

Pour que l'implantation d'une spécification à partir d'un langage synchrone soit valide, l'hypothèse synchrone doit être vérifiée par cette implantation. Pour que les hypothèses de simultanéité et d'atomicité soient réalistes, il est théoriquement nécessaire que la machine qui calcule les réactions soit infiniment rapide. En pratique, on peut simplement considérer un intervalle de temps maximum pendant lequel l'état du processus n'a pas le temps d'évoluer. Cet intervalle définit en quelque sorte une échelle de temps logique représentant les instants d'évolution du processus, c'est-à-dire les événements d'entrée du système temps réel. Il suffit alors que les actions du système temps réel soient produites en un temps inférieur à cet intervalle pour que, si on se place sur l'échelle de temps définissant les instants d'évolution du processus, les réactions puissent être considérées comme atomiques et donc que les actions sont simultanées avec les événements d'entrée.

5.1.2.2 Langages asynchrones

Dans les langages asynchrones le temps est considéré comme continu, il n'existe donc pas de notion de simultanéité. Il n'existe pas non plus de notions d'atomicité. On s'intéresse ici à la durée des calculs, il est donc possible qu'un événement déclenche une réaction alors que la réaction déclenchée par un événement survenu plutôt n'est pas encore achevée. Ceci conduit à un chevauchement temporel des deux réactions. Ce problème d'exécution simultanée est résolu à l'implantation par des techniques de *préemption* des calculs. La préemption peut poser des problèmes d'indéterminisme (l'occurence d'un événement peut conduire à plusieurs exécutions différentes du même programme)

et il peut être difficile de borner les temps d'exécution et donc impossible de garantir le respect des contraintes temps réel.

Remarque : les techniques d'implantation des spécifications synchrones et asynchrones seront détaillées et comparées dans la troisième partie de la thèse consacrée à l'implantation.

5.2 Outils basés sur une approche non-formelle de la spécification

Les *approches classiques* ou non-formelles de la conception des systèmes temps réel embarqués sont encore largement utilisés dans les laboratoires et entreprises du monde entier. Elles consistent à réaliser une spécification "papier" du système temps réel à partir de laquelle la programmation est réalisée dans un langage haut niveau (de type C, FORTRAN, ADA). Ces méthodes, satisfaisantes pour concevoir des systèmes transformationnels, montrent leur limites dès que l'on cherche à garantir la sûreté de conception des systèmes temps réel (où des vies humaines sont parfois mises en jeu). Elle ne permettent pas de gérer correctement la complexité des applications temps réel embarquées ; une vérification non exhaustive est possible mais elle doit être réalisée "à la main". Elle est donc, comme la mise point qui nécessite de très longues phases de tests et de déboguages, coûteuse en ressources humaines.

Pour combler l'écart entre ces méthodes et les nouvelles approches formelles, des outils de spécifications et de simulations ont été développés. Ces outils, basés sur une approche que l'on appellera non-formelle, constituent un bon support pour faire évoluer en douceur les mentalités habituées à utiliser les approches classiques et à leur faire accepter la rigueur des approches formelles et le coût d'une formation initiale à ces concepts nouveaux. Le succès de ces outils repose sur une spécification intuitive à l'aide de langages -le plus souvent graphiques- proches des modèles réalisés par l'automaticien, ainsi que sur de grandes facilités de simulation, de génération de code et de déboguage. Nous présentons ici un échantillon de ces outils.

5.2.1 MathWorks

La société Scientific Software propose depuis quelques années l'environnement MathWorks dont le but est de permettre à l'utilisateur de développer des algorithmes, concevoir des systèmes et générer du code. Cet environnement est constitué d'un langage textuel, Matlab, auquel ont été ajoutés deux outils, Simulink et StateFlow, afin de prendre en compte les spécificités de la conception des systèmes temps réel[43].

5.2.1.1 Matlab

MATLAB est, à l'origine, un environnement de calcul scientifique. C'est avant tout un langage interprété de haut niveau qui permet de manipuler simplement des structures de données telles que des tableaux, des fonctions, des matrices. La convivialité de ce langage est due :

– à une sémantique très proche des règles du calcul scientifique ;

– aux nombreuses fonctions mathématiques (sinus, opérations de manipulation des matrices et des complexes . . .) fournies en standard ;

– aux nombreuses commandes de visualisation graphique 2D et 3D ;

– aux nombreuses bibliothèques optionnelles, chacune adaptée à un domaine scientifique particulier (traitement du signal, traitement d'images, réseaux de neurones, automatique . . .);

– à la possibilité d'interfacer du code C ou Fortran avec les fonctions MATLAB.

5.2.1.2 Simulink

Simulink est une interface graphique permettant de décrire des graphes flots de données dont les blocs sont des fonctions décrites avec Matlab. Ces graphes sont proches de la représentation schéma-bloc utilisé généralement par l'automaticien pour représenter les lois de commande d'un système automatisé (fig. 5.2). Il permet de spécifier et paramétrer rapidement un prototype grâce à des simulations prenant à la fois en compte le comportement continu du processus et le comportement discret du système temps réel évitant ainsi de longues phases de tests. Son utilisation est proche de celle des outils de C.A.O pour la conception des systèmes électroniques :

– il intègre une importante bibliothèque de blocs prédéfinis (gain, intégrateur, filtre . . .) comme les outils de C.A.O intègrent des bibliothèques de composants ;

– comme pour les outils de C.A.O, il est possible de construire hiérarchiquement de nouveaux blocs, soit en utilisant des blocs existants, soit en créant de nouveaux blocs à partir de fonctions Matlab ;

– enfin, les nouveaux blocs créés par les utilisateurs peuvent être regroupés dans des librairies pouvant être réutilisées dans d'autres applications.

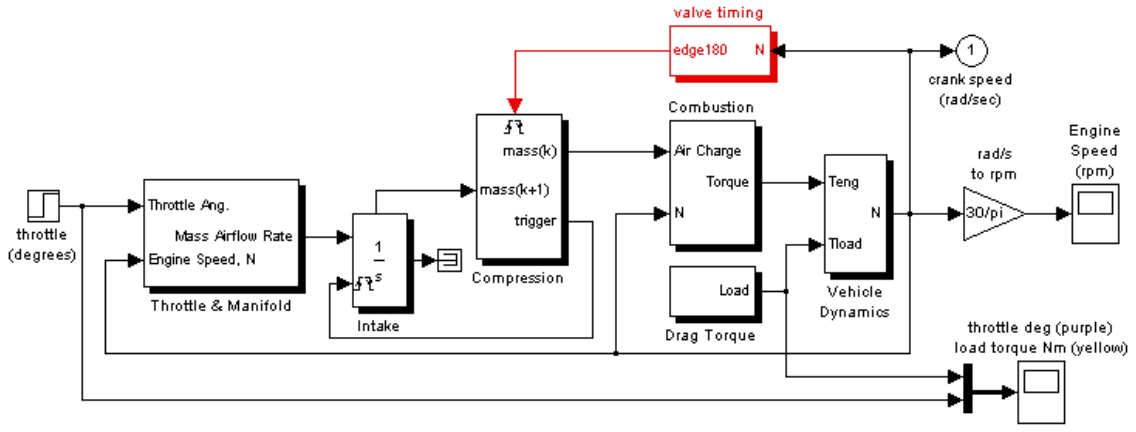


FIG. 5.2 – *Exemple de graphe Simulink*

Les dernières versions de Simulink intègrent une nouvelle fonctionnalité importante pour la spécification des systèmes temps réel : les *triggers*. L'association d'un trigger à un bloc Simulink ajoute

une entrée à celui-ci. L'exécution du bloc peut alors être contrôlée par un signal connecté à cette nouvelle entrée. Selon la configuration du trigger l'exécution peut être déclenchée soit sur un front montant (la valeur du signal passe d'une valeur négative à une valeur positive), soit sur un front descendant (la valeur du signal passe d'une valeur positive à une valeur négative), soit sur un front (changement de signe du signal). Une option spécifique du trigger permet de déclencher l'exécution d'un bloc sur un signal de type *function call*. Cette option permet à un bloc (celui qui émet le signal) de demander l'exécution du bloc triggé comme pour un appel de sous-programme.

Le trigger est un apport important des dernières versions de Simulink car, par ce biais, il est possible de spécifier la fréquence d'exécution d'un bloc en connectant son entrée trigger à la sortie d'un générateur d'impulsions *discrete pulse generator* configuré sur la fréquence recherchée : c'est une manière de spécifier des contraintes de cadence sur les blocs.

5.2.1.3 Stateflow

Stateflow est un outil intégré récemment à Simulink. C'est une interface graphique qui permet de décrire des automates de contrôle selon une sémantique proche du langage Statecharts (fig. 5.3). L'automate est édité avec Stateflow et compilé sous la forme d'une fonction C qui peut être intégrée dans un schéma Simulink sous la forme d'un bloc dont les entrées sont des conditions associées aux transitions de l'automate et les sorties sont des actions associées aux états et/ou aux transitions. Les sorties de l'automate permettent de contrôler l'exécution des blocs par le biais des entrées trigger. Les changements de mode peuvent être décrits par un automate. Les lois de commande sont décrites par un graphe Simulink. L'automate contrôle l'exécution des lois de commande.

5.2.1.4 Génération automatique de code

L'environnement MathWorks peut être connecté à différents outils de génération de code notamment Real Time Worshop qui réalise une analyse automatique d'une spécification Simulink et génère du code C ou ADA95 à partir de cette analyse. Le code généré peut s'appuyer sur différents exécutifs temps réel tels que VxWorks/Tornado de la société Wind River pour des architectures de bus VME, PCI, ISA, PC104 ainsi que des cartes spécifiques à base de DSP.

Une telle chaîne d'outils permet de couvrir toutes les étapes de conception du cycle en V. C'est pourquoi Simulink est très utilisé pour la réalisation rapide de prototypes qui permettent généralement de valider la faisabilité du système. A partir du prototype mis au point, la conception d'un produit industriel prenant en compte les contraintes de coût peut alors être lancée. Le programme Simulink du prototype doit alors être recodé "à la main" pour être adapté et optimisé pour l'architecture cible multiprocesseur envisagée qui n'est généralement pas prise en compte par les outils de génération de code disponible pour Simulink.

5.2.2 Autres outils

Il existe d'autres outils de simulation et de spécification des systèmes temps réel embarqués. Nous présentons ici sommairement quelques uns de ces outils car les concepts qu'ils mettent en œuvre sont similaires à ceux que l'on rencontre dans l'environnement MathWorks. Nous nous contenterons donc d'énoncer les différences entre ces outils.

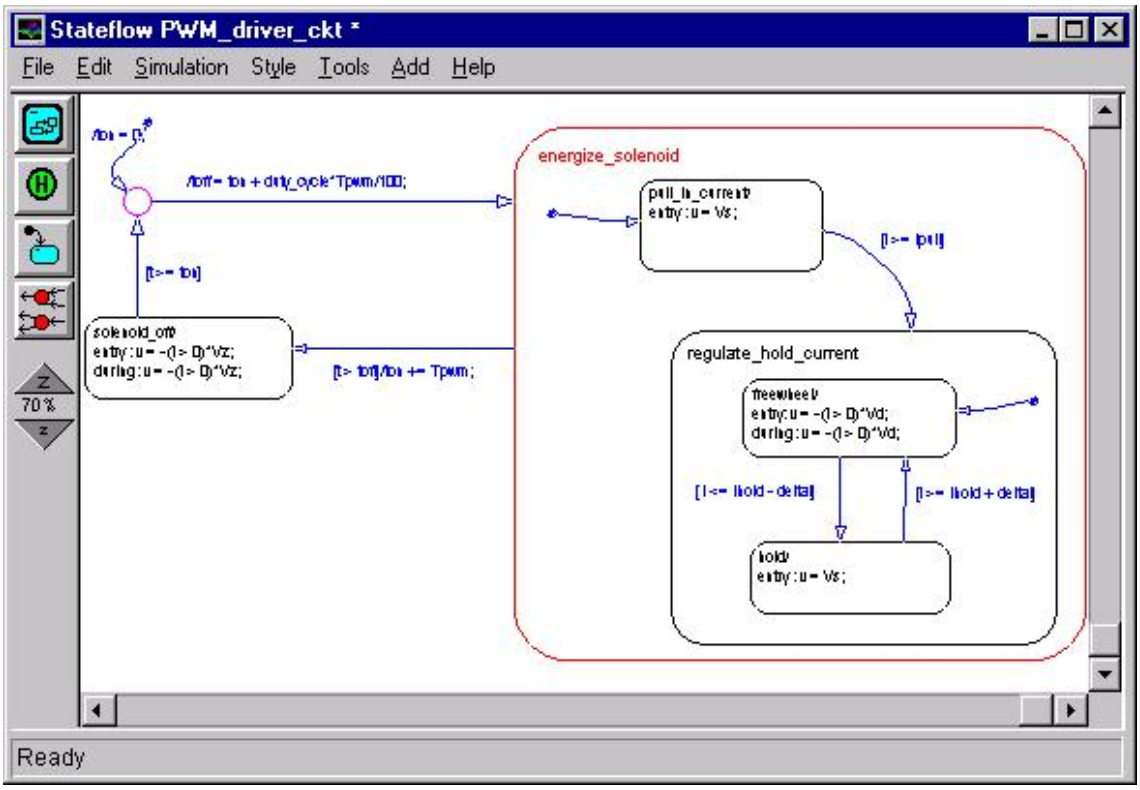


FIG. 5.3 – *Exemple de graphe stateflow*

5.2.2.1 MATRIXx

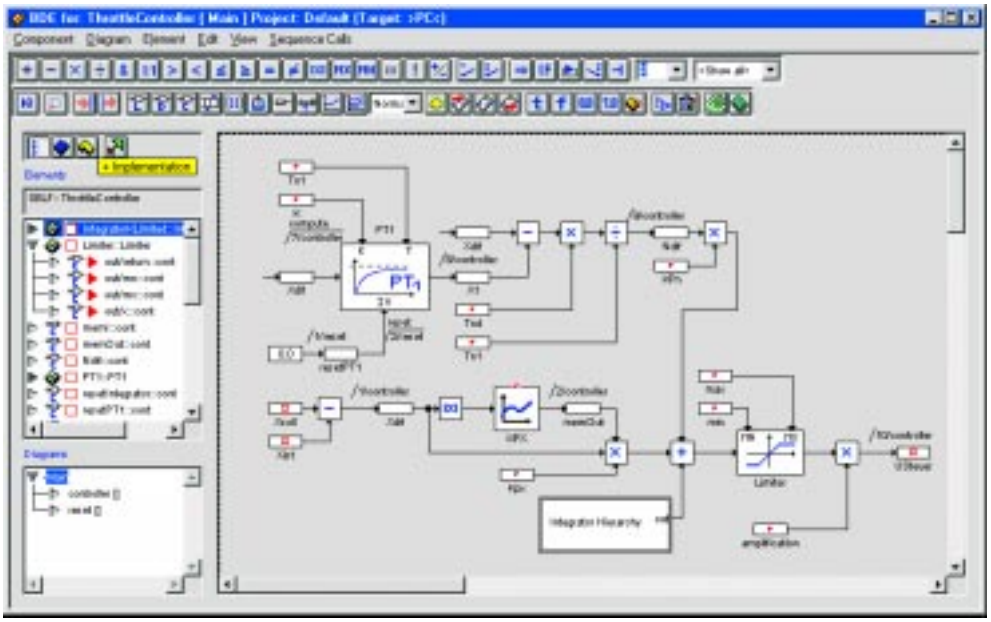
L'environnement MATRIXx est un ensemble d'outils graphiques développés et commercialisés par la société Integrated systems[44]. Il regroupe principalement deux outils de spécification :

- Système Build qui est l'équivalent de Simulink ;
- BetterState qui est l'équivalent de Stateflow. Il permet de décrire les systèmes à événements discrets à l'aide de graphes flot de contrôle orientés automate dont la sémantique rappelle celle de Statecharts et à l'aide de graphe flot de contrôle de type organigramme.

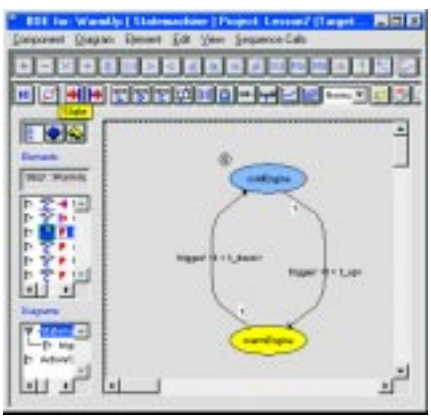
Cette approche est essentiellement graphique, mais il est possible d'utiliser le langage de spécification textuel fournit par l'outil mathématique Xmath proche de Matlab. La conception des systèmes repose sur l'utilisation des bibliothèques fournies et sur des bibliothèques utilisateur écrites dans le langage cible. Il offre des outils de documentation automatique *DocumentIt* et de génération de code *AutoCode*. Ce code peut s'appuyer sur les exécutifs commercialisés par la même société, pSOS et pOSEK, ou sur des API personnalisables.

5.2.2.2 ASCET-SD

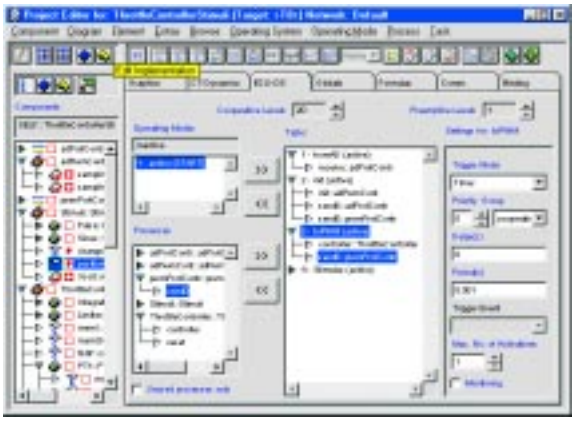
ASCET-SD est un produit de la société ETAS[41]. Il a été conçu particulièrement pour la spécification des applications automobiles. La philosophie est proche de celle de Mathworks. Dans un



(a) Schéma-bloc



(b) Automate



(c) Contraintes temporelles

FIG. 5.4 – *Outil ASCET-SD*

même environnement, ASCET-SD offre une interface graphique de spécification flot de données de type Simulink (fig. 5.4(a)) et une interface graphique de spécification d'automates (fig. 5.4(b)). La spécification des contraintes temporelles ne se fait pas ici de manière graphique sur le graphe flot de données comme dans Simulink, mais par le biais d'une interface spécifique où l'on peut définir les priorités, les périodes d'exécution des blocs et d'autres paramètres de l'application (fig. 5.4(c)).

La génération de code s'appuie sur l'exécutif ERCOS (compatible OSEK) développé par ETAS.

5.2.2.3 SCILAB, SCICOS

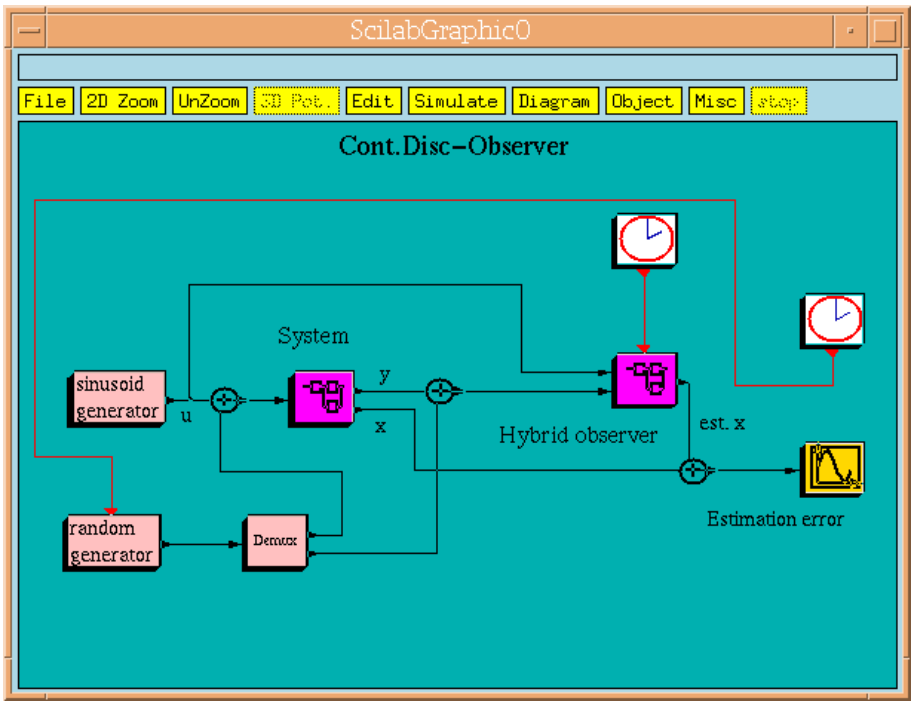


FIG. 5.5 – *Diagramme scicos d'un modèle hybride de système continu avec observateur discret*

Scilab est un outil développé par l'INRIA pour la spécification et la simulation d'applications de contrôle-commande et de traitement du signal et des images[42]. Scilab est l'homologue purement recherche de Matlab. Il offre des caractéristiques sensiblement identiques. Comme Matlab, il permet la manipulation de différentes structures de données, scalaires, vecteurs et matrices, et fournit un ensemble de librairies de fonctions de calculs et de visualisations graphiques. Scilab supporte la hiérarchie et peut être interfacé avec des fonctions FORTRAN ou C. Scilab est un produit distribué gratuitement sous forme de code source (licence GPL) et profite donc de toute la communauté du logiciel libre qui le fait évoluer. Une interface Maple-Scilab permet de combiner les avantages du calcul symbolique de Maple avec le calcul numérique réalisé par Scilab.

Scicos est une boîte à outils de Scilab qui permet de définir graphiquement et de simuler des systèmes hybrides complexes (prise en compte de systèmes discrets et continus). L'éditeur graphique de Scicos permet de décrire le système sous la forme de schémas-bloc (figure 5.5). Cette spécification est proche de celle de Simulink. Néanmoins, la sémantique de Scicos est plus rigoureuse que celle de Simulink. En Scicos, à chaque signal est associé un ensemble d'intervalles temporels

appelés “intervalles d’activation” pendant lesquels le signal peut évoluer. Hors de ces intervalles d’activation, le signal n’est pas calculé. Les signaux sont produits par les blocs eux-mêmes pilotés par des signaux d’activation reliés à l’entrée d’activation du bloc. Le calcul du bloc n’est effectué que pendant les intervalles d’activations définis par le signal d’activation qui conditionne son exécution. Cette fonctionnalité est l’équivalent des triggers de Simulink, mais ici le calcul des instants d’activation possède une sémantique claire et définie : les signaux de sortie d’un bloc héritent des intervalles d’activation du bloc et peuvent être utilisés pour conditionner l’exécution d’autres blocs. Lorsque l’entrée d’activation d’un bloc n’est pas reliée à un signal, l’intervalle d’activation du bloc est l’union des intervalles d’activation de ces signaux d’entrée.

Des travaux sont en cours pour interfacer Scicos avec SynDEx (cf. §5.3.7) un outil de génération de code pour applications temps réel distribuées basé sur une approche formelle.

5.3 Outils basés sur une approche formelle de la spécification, méthodologie AAA

5.3.1 Langage synchrones

La famille des langages synchrone s’inscrit clairement dans le cadre des approches formelles de spécification des systèmes temps réels puisqu’ils ont été conçus dans ce but [13][14]. Chacun de ces langages peut être considéré à part entière comme un outil de spécification et de conception des systèmes temps réel. Il existe principalement 4 langages synchrones :

– *Esterel* est le plus ancien des langages synchrones, il est développé par l’INRIA en collaboration avec l’ENSMP[2]. C’est un langage impératif dont les principales primitives permettent de décrire l’exécution parallèle, le séquencement d’actions et l’attente d’événements[17][18]. Le langage Esterel est bien adapté à la spécification des machines à états finies décrivant les algorithmes de contrôle ;

– *Statecharts*, développé par D.Harel et Pnueli est aussi un langage de type impératif. C’est un langage graphique flot de contrôle orienté automate. Il permet de donner une description comportementale d’un système en termes d’états, d’événements et de transitions. Le formalisme autorise la hiérarchisation et le parallélisme d’automates ainsi que la diffusion des événements ;

– *Lustre* est un langage déclaratif. Il est développé à L’IMAG[3]. Un programme *Lustre* est un ensemble d’*équations* (au sens mathématique du terme) et d’*assertions* (utilisées pour spécifier au compilateur certaines propriétés utiles sur le programme en vue de réaliser des vérifications et des optimisations). Ces équations et assertions permettent de définir des transformations appliquées à des *flots* définis par une séquence de valeurs à laquelle est associée une horloge représentant une séquence d’instants. L’horloge de chaque flot est fonction d’une horloge de base ;

2. Ecole Nationale Supérieure des Mines de Paris
3. Institut de Mathématiques Appliquées de Grenoble

– *Signal* est développé à l'IRISA [4][51][49][7]. C'est un langage déclaratif très proche de Lustre. Comme Lustre les primitives du langage permettent de manipuler les valeurs et les horloges de flots. En revanche, en Signal il n'existe pas d'horloge de base. Les opérateurs du langage permettent de décrire des relations entre les horloges des différents flots. C'est le langage qui synthétise une horloge de base à partir de ces relations. Signal et Lustre sont des langages flots de données, ils sont donc bien adaptés pour spécifier les algorithmes de commande.

Tous les compilateurs de ces langages traduisent la spécification d'une part en un automate sur lequel peut être réalisé des vérifications comportementales et d'autre part en un programme séquentiel généralement basé sur le langage C.

Pour plus de détails sur les différents langages synchrones, le lecteur pourra se reporter à l'ouvrage de N. Halbawchs [38] consacré à ce sujet.

5.3.2 Electre

A notre connaissance, Electre est le seul langage formel asynchrone. C'est un langage de type impératif développé à l'IRCyN [5]. Electre manipule des tâches et des événements. Les opérateurs Electre permettent d'exprimer le séquencement, la répétition, le parallélisme, la préemption des tâches en fonction de l'occurence d'événements ainsi que l'exclusion entre tâches[72][73]. Différents travaux ont conduit à réaliser un environnement de développement appelé Atride (Atelier Temps Réel pour l'Implémentation et le Développement en Electre) regroupant des outils d'édition, de compilation, de simulation (Silex), d'exécution (Exile) et de vérification temporelle (Valet) ; la génération de code s'appuyant sur le système d'exploitation temps réel répartie Chorus.

5.3.3 Statemate Magnum

Statemate Magnum est un outil graphique de simulation et de développement conçu par la société I-Logix. La spécification comportementale est décrite principalement à l'aide du langage Statecharts ce qui autorise la vérification de la spécification. Les aspects commande sont décrits avec le langage flot de données Activity-chart. Ces deux langages sont combinés à l'aide d'un troisième langage, Module-chart permettant de décrire la structure de l'application, c'est-à-dire les liens qui existent entre les différents modules décrits à l'aide de Statecharts et Activity-chart.

Un effort tout particulier a été réalisé pour permettre la connexion de Statemate Magnum avec de nombreux autres outils :

– pour la simulation : Simulink (Mathworks) et SystemBuild (Matrixx) ;

– pour la réalisation d'interfaces graphiques : VAPS de Virtual Prototype ;

– pour la génération de code : AutoCode (Matrixx), il peut aussi générer du code basé sur l'exécutif VxWorks de Wind River.

– pour le débogage : DOORS de Quality Systems and Software et RTM de Marconi Systems Technology

4. Institut de Recherche en Informatique et Système Aléatoire
5. Institut de Recherche en Cybernétique de Nantes

5.3.4 SyncCharts/Esterel

SyncCharts/Esterel est un produit développé par Simulog[15][82]. Il utilise un formalisme graphique flot de contrôle orienté automate proche de Statecharts. A partir de cette spécification graphique il est possible de générer un code source Esterel et ainsi profiter des outils de simulation comportementale de vérification et de génération de code d'Esterel.

5.3.5 ORCCAD

ORCCAD est un environnement de conception de systèmes temps réel embarqués orienté robotique[81]. La spécification des applications est organisée selon une hiérarchie de trois niveaux :

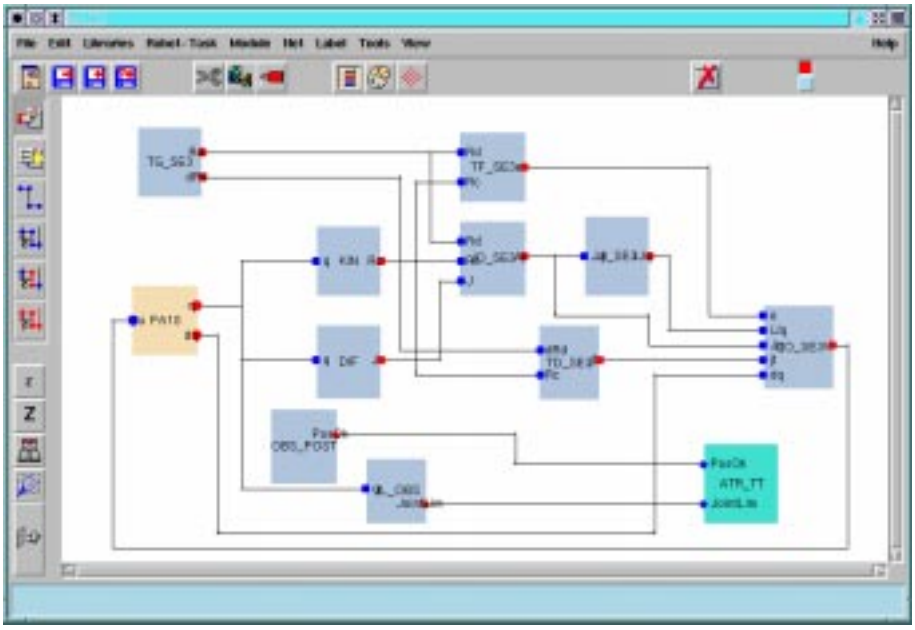
- Tâche-Module : c'est l'entité de base la plus fine. Elle est utilisée pour spécifier une partie d'une loi de commande. Chaque Tâche-Module représente un ensemble ordonné d'opérations ainsi qu'une contrainte de période d'exécution sur ces opérations.
- Tâche-Robot : c'est un ensemble de Tâche-modules qui communiquent afin de réaliser une loi de commande ;
- Procédure-Robot : c'est l'entité de plus haut niveau. C'est un automate décrivant le séquencement, la mise en parallèle de Tâche-Robot.

Les niveaux Tâche-Robot et Tâche-module définissent le comportement dynamique du robot. Ils sont mis en œuvre par les automaticiens. En revanche le niveau Procedure-Robot permet de spécifier les missions du robot et s'adresse donc aux ingénieurs chargés de programmer le robot conçu par les automaticiens.

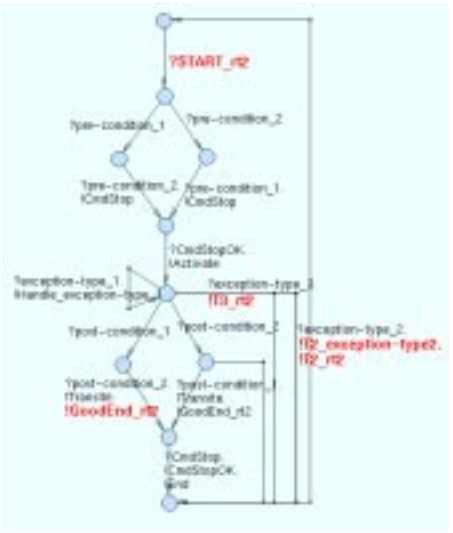
ORCCAD utilise une approche graphique flot de données pour décrire les tâches-Robot. Une Tâche-Module est écrite dans un langage de programmation (généralement C). Le niveau Procédure-Robot est décrit à l'aide du Langage Esterel, ce qui autorise sa vérification comportementale. La génération de code s'appuie sur l'OS temps réel VxWorks de WindRiver systems.

5.3.6 SILDEX

Sildex[87] est un environnement graphique de conception des systèmes temps réel développé par la société TNI basé sur le langage Signal. Bien que Sildex ne soit pas un outil de simulation hybride, la philosophie de spécification est proche de celle utilisée dans les approches de type Mathworks. Les algorithmes sont spécifiés à l'aide d'un graphe flot de données rappelant les schémas-bloc de l'automaticien : chaque composant du graphe apparaît comme un symbole graphique reflétant sa fonction qui est décrite soit directement à l'aide du langage Signal soit en combinant d'autres composants (hiérarchie)(fig. 5.7). Un système de librairie permet de créer de nouveaux composants réutilisables dans d'autres applications ou d'utiliser ceux fournis avec le logiciel. La validation de la spécification peut être faite à la fois par simulation en positionnant des "sondes" sur le graphe permettant la visualisation de certains signaux lors de l'exécution, et par vérification formelle. Sildex dispose d'un éditeur qui permet aussi de spécifier des automates, des tables de vérité et du GRAFCET. Ces spécifications sont automatiquement traduites en langage Signal et intégrées au graphe Sildex sous la forme d'un composant.



(a) Éditeur Tâche-Robot



(b) Visualisation d'Automates

(c) Éditeur Procédure-Robot

FIG. 5.6 – *Outil ORCCAD*

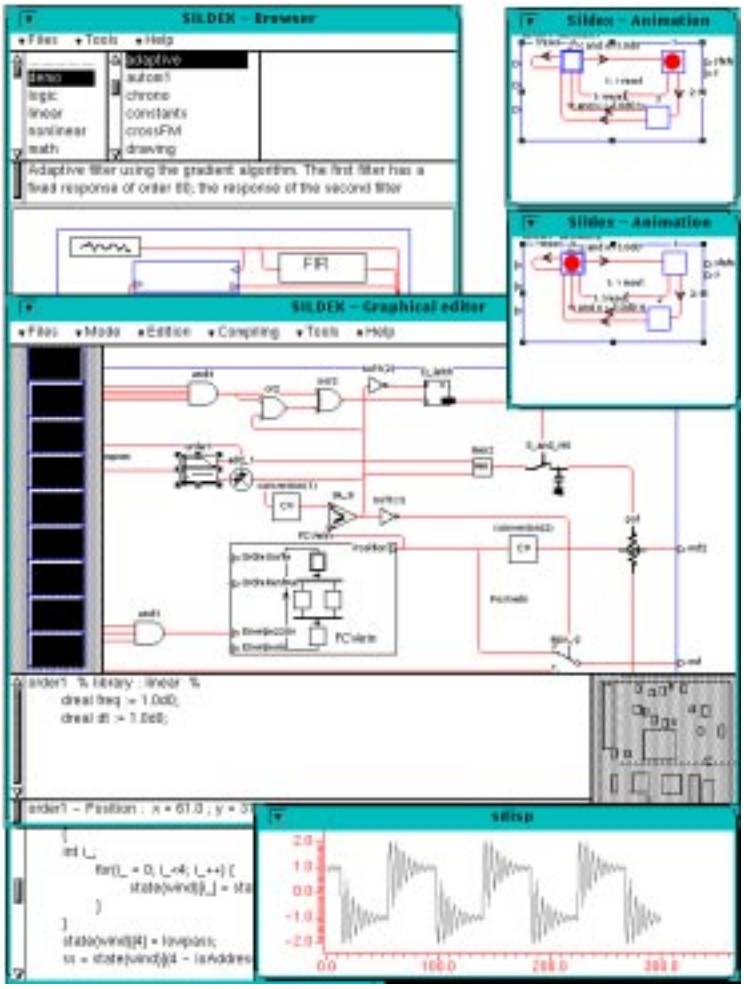
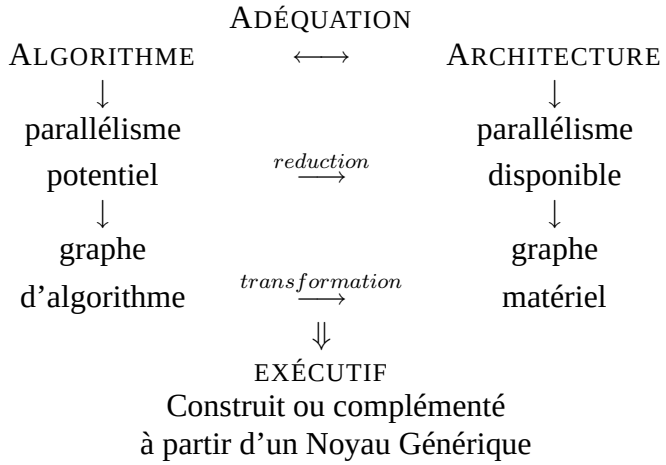


FIG. 5.7 – *Outil Sildex*

Les dernières versions de Sildex permettent d'importer un graphe Simulink-Stateflow, de générer le code C ou JAVA monoprocesseur et de générer automatiquement de la documentation concernant le code et la spécification de l'application.

5.3.7 Méthodologie AAA, SynDEx

La méthodologie AAA [6] développée à l'INRIA a pour but de réaliser une implantation optimisée sur des architectures distribuées d'algorithmes temps réel spécifiés et vérifiés à l'aide de langages reposant sur la sémantique des langages Synchrones [83] . Elle s'appuie sur un formalisme de graphes pour transformer de manière automatique une spécification (graphe d'algorithme et graphe d'architecture) en un code optimisé multiprocesseur. Les graphes ont été choisis car ils sont bien adaptés à décrire aussi bien du matériel que du logiciel et par conséquent bien adaptés à décrire les problèmes d'optimisation d'implantation et de génération de code automatique qui en découlent.



Un algorithme tel que défini par Turing et Post est une séquence (ordre total) finie d'opérations directement exécutable par une machine à états finie. Cette définition doit être étendue afin de permettre d'une part la prise en compte du parallélisme disponible dans les architectures distribuées, composées de plusieurs machines à états finies interconnectées, et d'autre part la prise en compte de l'interaction infiniment répétitive de l'application avec son environnement. Pour cela notre modèle d'algorithme est un *graphe de dépendances factorisé* : c'est un hypergraphe orienté acyclique (DAG) [75], dont les sommets sont des *opérations* partiellement ordonnées [69] (parallélisme potentiel) par leurs dépendances de données (hyperarcs orientés pouvant avoir plusieurs extrémités pour une seule origine, "diffusion"), et dont l'exécution est conditionnée par une dépendance d'entrée particulière "de conditionnement" (l'exécution n'a lieu que lorsque la dépendance porte une valeur particulière, booléenne ou même entière). À chaque interaction avec l'environnement, concrétisée par un ensemble d'événements d'entrée issus des capteurs, les valeurs des arcs de conditionnement déterminent l'ensemble des opérations à exécuter pour obtenir les événements de sortie pour les actionneurs, à partir des valeurs d'entrée acquises par les capteurs. L'algorithme est donc modélisé par un graphe de dépendances, infiniment large mais répétitif, réduit par factorisation à son motif répétitif [48], généralement appelé *graphe flot de données*. De plus, chaque partie répétitive du graphe flot de données ("nid de boucles", répétitions finies) est aussi réduite par factorisation à son

6. Adéquation Algorithme Architecture

motif répétitif. Le graphe de l'algorithme peut être soit directement spécifié comme tel, ou bien déduit d'une spécification séquentielle ou CSP (Communicating Sequential Processes de Hoare) par analyse de dépendances, ou encore produit par les compilateurs des langages synchrones (Esterel, Lustre, Signal, à travers leur format commun "DC") [38] qui présentent l'intérêt de faire des vérifications formelles en termes d'ordre sur les événements.

Les modèles les plus classiquement utilisés pour spécifier des architectures parallèles ou distribuées sont les PRAM ("Parallel Random Access Machines") et les DRAM ("Distributed Random Access Machines") [92]. Le premier modèle correspond à un ensemble de processeurs communicant par mémoire partagée alors que le second correspond à un ensemble de processeurs à mémoire distribuée communicant par passage de messages. Si ces modèles sont suffisants pour décrire, sur une architecture homogène, la distribution et l'ordonnancement des opérations de calcul de l'algorithme, ils ne permettent pas de prendre en compte des architectures hétérogènes ni de décrire précisément la distribution et l'ordonnancement des opérations de communication inter-processeurs qui sont souvent critiques pour les performances temps réel.

Pour cela notre modèle d'*architecture multicomposant* hétérogène est un hypergraphe non orienté, dont chaque sommet est une machine à états finie [33], séquenceur d'opérations de calcul ou séquenceur d'opérations de communication, que nous appelons respectivement *opérateur de calcul* et *opérateur de communication*, et dont chaque hyperarc est une ressource partagée entre séquenceurs (mémoire RAM et/ou FIFO, son bus, le multiplexeur d'accès au bus, et l'arbitre du multiplexeur) que nous appelons respectivement *média RAM* et *média FIFO*. Un opérateur de calcul ne peut être connecté qu'à un (des) média(s) RAM, alors qu'un opérateur de communication peut être connecté à des médias RAM et/ou FIFO. L'hétérogénéité ne signifie pas seulement que les opérateurs et les médias peuvent avoir chacun des caractéristiques différentes (durée d'exécution des opérations et taille mémoire des dépendances de données), mais aussi que certaines opérations ne peuvent être exécutées que par certains opérateurs, ce qui permet de décrire aussi bien des composants programmables (processeurs) que des composants spécialisés (ASIC ou FPGA) [27].

En termes plus concrets, notre modèle d'architecture est une extension du modèle classique RTL ("Register Transfer Level", niveau transfert de registres) [63], que nous qualifions de *Macro-RTL*. Une opération du graphe de l'algorithme est une *macro-instruction* (une séquence d'instructions ou un circuit combinatoire) ; une dépendance de données est un *macro-registre* (des cellules mémoire contiguës ou des conducteurs interconnectant des circuits combinatoires). Ce modèle encapsule les détails liés au jeu d'instructions, aux micro-programmes, au pipe-line, au cache, et lisse ainsi ces caractéristiques de l'architecture, qui seraient sans cela trop délicates à prendre en compte lors de l'optimisation. Il présente une complexité réduite adaptée aux algorithmes d'optimisation rapides tout en permettant des résultats d'optimisation relativement (mais suffisamment) précis.

L'*adéquation* consiste à mettre en correspondance de manière efficace l'algorithme sur l'architecture pour réaliser une implantation optimisée, c'est-à-dire allouer spatialement (distribution) et temporellement (ordonnancement) les ressources matérielles de l'architecture aux opérations de l'algorithme en minimisant les ressources. Ce type de problème est connu comme étant NP-complet ; on doit donc se contenter, si on veut un résultat dans un temps raisonnable, d'une solution sous-optimale calculée à l'aide d'heuristiques. Le critère d'optimisation choisi est un critère de latence c'est-à-dire que l'on cherche à réduire le temps total d'exécution du graphe.

Cette méthodologie est supportée par le logiciel SynDEx[7] lui aussi développé à l'INRIA[91].

7. **Syn**chronized **D**istributed **Ex**ecutive

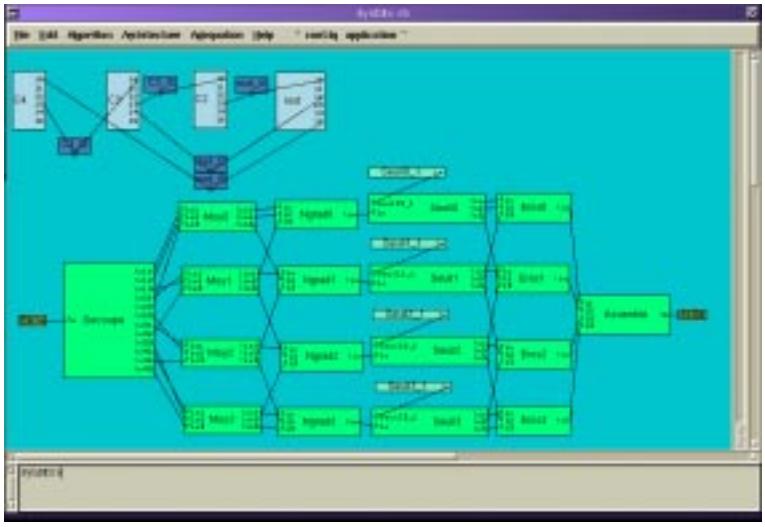
La figure 5.8(a) présente un graphe d'architecture (dans la partie supérieure de la fenêtre) composé d'un réseau de 4 microprocesseurs reliés par des liaisons SAM ainsi qu'un graphe d'algorithme décrits dans l'interface de SynDEx v5. A l'issue de l'adéquation SynDEx réalise une simulation de la séquence d'exécution des opérations sur chaque processeur en prenant en compte les temps de communications (fig. 5.8(b)). Cette simulation temporelle permet de vérifier si les contraintes temps réel sont satisfaites, mais elle permet aussi de dimensionner l'architecture matérielle en vérifiant l'impact d'une modification de l'architecture (ajout, suppression, changement d'opérateur ou de media de communication) sur le temps d'exécution du graphe. A l'issue de cette simulation, SynDEx peut générer un code optimisé intégrant un exécutif taillé sur mesure pour l'application (communications inter-processeurs, initialisations, ordonnancement etc.) dont le surcoût est très faible comparé au surcoût généralement engendré par les exécutifs commerciaux. Ce code est garanti sans interblocage et conforme à la spécification.

5.4 Synthèse de l'état de l'art : vers l'outil idéal

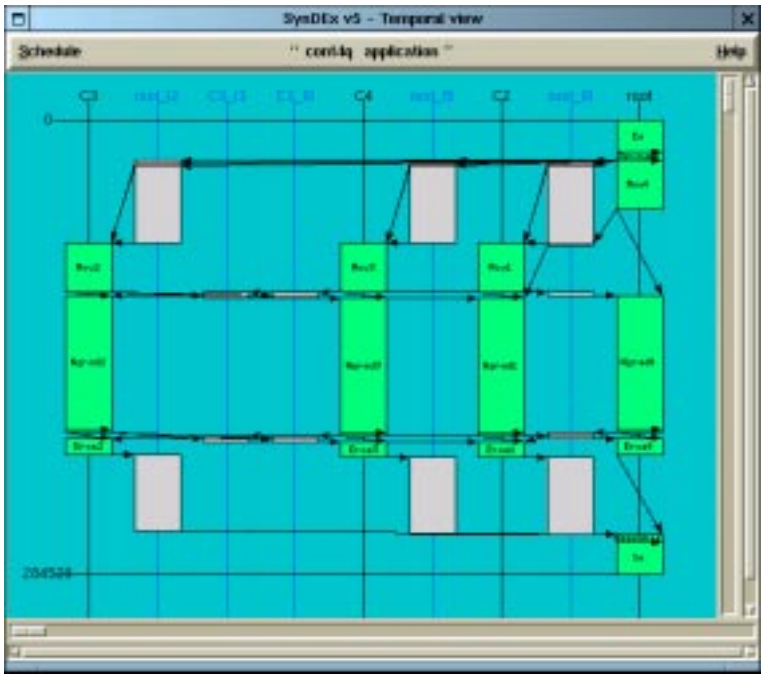
On peut imaginer un outil idéal qui, à partir du modèle mathématique défini par l'automaticien, réaliserait la conception complète du système temps réel identique en tout point au modèle et respectant les contraintes de coût. Il est bien évidemment utopique de croire que cet outil puisse exister car il devrait être capable de résoudre des problèmes d'optimisation connus comme étant NP-complets et dont le nombre de paramètres est important. Par exemple, ne serait-ce que pour choisir une architecture matérielle, et en considérant pour simplifier le problème que le choix de capteurs et actionneurs ait été réalisé par l'automaticien, un tel outil devrait disposer d'une base de connaissances énorme sur les caractéristiques physiques des composants (consommation, encombrement) et logique (type combinatoire ou séquentielle, nombre d'entrées/sorties, espace mémoire adressable, horloge ...) des composants existants ; il nécessiterait aussi de disposer d'algorithmes permettant de choisir l'architecture optimale parmi un espace de solutions gigantesque. Il est par contre raisonnable de penser réaliser des environnements de conception regroupant des outils chacun bien adapté à certaines classes de problèmes, permettant de réaliser et d'implanter des choix orientés par le savoir faire des ingénieurs. A partir des points forts de chacun des outils et langages décrits précédemment, nous donnons les spécificités que devrait, à notre avis, posséder un environnement d'aide à la conception des systèmes temps réel "quasi idéal" autorisant un cycle de conception proche du cycle en cascade sans retour (cf. §3.3.2).

5.4.1 Approche formelle

Le cycle de développement en cascade sans retour suppose que chaque étape de conception est validée avant de réaliser l'étape suivante et qu'aucune erreur n'est introduite lors du passage d'une étape à une autre. On aboutit ainsi à la phase de conception finale sans avoir à remettre en cause les étapes précédentes. A la fin du cycle de conception, le système est conforme au cahier des charges et il n'est pas nécessaire de remettre en cause certaines étapes du cycle. Il est évident que pour prétendre aboutir à un tel cycle de conception, l'approche choisie doit être très rigoureuse et donc nécessairement formelle. Elle doit, de plus, couvrir toutes les étapes du cycle de conception (modélisation, spécification, implantation, validation) et fournir des outils et méthodes nécessaires pour valider chacune de ces étapes.



(a) Graphe d'architecture et graphe d'algorithme



(b) Graphe temporel

FIG. 5.8 – *Logiciel SynDEx*

5.4.2 Spécification hiérarchisée graphique et textuelle

Nous avons vu que le principe de la spécification est de décrire de manière non-ambiguë le système, qu'elle doit en présenter les aspects essentiels sans en montrer les détails et qu'elle doit être assez fine pour pouvoir exprimer toutes les subtilités du cahier des charges.

Pour répondre à ces besoins, la spécification doit être essentiellement graphique et doit pouvoir supporter la hiérarchie. La hiérarchie est, pour la spécification graphique, l'équivalent de la procédure, de la fonction ou du sous-programme des langages textuels. Un graphe, mieux que du texte, permettra d'un simple coup d'oeil de comprendre le fonctionnement global du système, la hiérarchie permettant d'encapsuler les détails dans des éléments de haut niveau du graphe.

Une approche graphique hiérarchique est l'approche la plus ergonomique pour l'utilisateur. L'œil et le cerveau utilise une approche hiérarchique par exemple pour rechercher une ville sur une carte routière ; un premier examen rapide et non détaillé de la carte permet de repérer la région dans laquelle se situe la ville. L' œil se focalise ensuite sur les détails de cette région pour y repérer la ville en question.

De plus, ce type d'approche correspond naturellement à la stratégie de conception d'un système qui consiste à raffiner une spécification pour réaliser une implantation : chacun des éléments de plus haut niveau de spécification peut encapsuler un sous-graphe décrivant plus finement sa fonction. Elle permet aussi de simplifier la conception de l'application en présentant le système comme un assemblage d'éléments plus simples, ce qui correspond à la démarche de l'automaticien lorsqu'il décompose le processus en sous-processus plus simples.

Pour renforcer l'expressivité de la spécification graphique, il peut être intéressant d'offrir à l'utilisateur la possibilité de remplacer certains éléments du graphe par des icônes. Par exemple, dans un graphe flot de données, un bloc décrivant la fonction d'acquisition d'un capteur de température pourra être avantageusement associé à une icone représentant un thermomètre. La lisibilité générale du graphe peut en être grandement améliorée.

Pour être efficace et d'utilisation aisée, l'outil de spécification doit aussi offrir la possibilité de spécifier textuellement certains éléments car plus on cherchera à raffiner la spécification, plus on mettra en évidence les détails de l'implantation. Pour représenter tous ces détails, il sera alors nécessaire d'utiliser un nombre important d'éléments graphiques. Dans un tel niveau de raffinement plutôt proche de la programmation, il est plus judicieux d'utiliser un langage textuel qui permet de transcrire plus rapidement les algorithmes mis en jeu.

5.4.3 Multi-formalisme

Les langages impératifs (ou flots de contrôle) sont bien adaptés pour décrire les aspects contrôle d'un système temps réel. En revanche, lorsqu'il s'agit de décrire les aspects commande, les langages déclaratifs (ou flots de données) sont mieux adaptés. C'est pourquoi les automaticiens utilisent généralement des schéma-bloc (flots de données) pour décrire leurs lois de commande, et lorsqu'ils cherchent à décrire le séquencement, les changements de modes de fonctionnement d'une machine de production ils utilisent plutôt des langages de type flots de contrôle tels que le GRAFCET, les réseaux de Petri, les automates etc.

La spécification des systèmes temps réel embarqués nécessite la prise en compte à la fois des aspects commande et des aspects contrôle qui sont généralement tous deux présents dans ce type

de système. Puisqu'on cherche à réduire l'écart entre le modèle et la spécification, l'outil de spécification quasi idéal doit permettre à l'automaticien d'utiliser le même formalisme pour décrire le modèle et spécifier les algorithmes correspondant. Le langage de spécification ne doit pas nécessiter d'effort d'adaptation particulier de la part de l'automaticien et doit donc être proche des méthodes et outils qu'il a l'habitude de manipuler. Ainsi, pour spécifier la commande du système temps réel il est nécessaire d'utiliser un langage graphique flot de données afin de rester proche des schémas-bloc. Pour spécifier la commande on privilégiera plutôt un langage flot de contrôle de type automate.

Nous montrons ici qu'il est indispensable qu'un outil de spécification ne soit pas basé uniquement sur un seul formalisme, mais qu'il fournisse à l'utilisateur la possibilité de spécifier chacun de ces algorithmes avec un formalisme adapté. En l'occurence, un outil de conception de systèmes temps réel doit permettre de spécifier les algorithmes de commandes avec un langage flot de données et les algorithmes de contrôle avec un langage flot de contrôle.

La plupart des outils que nous avons présentés permettent de spécifier le système de contrôle-commande à l'aide de ces deux types de formalismes. Néanmoins, dans ces outils, il existe toujours une prédominance de l'un par rapport à l'autre. Certains outils privilégient une approche flot de données dans laquelle il est possible d'introduire du flot de contrôle. C'est le cas des outils tels que Mathworks, Matrixx, ASCET-SD, Sildex qui permettent de spécifier certains blocs (ou sommets) du graphe flot de données à l'aide d'un langage de type automate. D'autres outils sont plus orientés flot de contrôle et permettent de décrire les actions déclenchées par les sorties de l'automate à l'aide d'un graphe flot de données (Statemate Magnum, Orccad). Cette deuxième approche nous semble préférable car les algorithmes de contrôle sont généralement les algorithmes de plus haut niveau dans la spécification, alors que les algorithmes de commande sont les plus proches du matériel, il est donc plus logique de raffiner une spécification de type flot de contrôle à l'aide d'un formalisme flot de données que l'inverse.

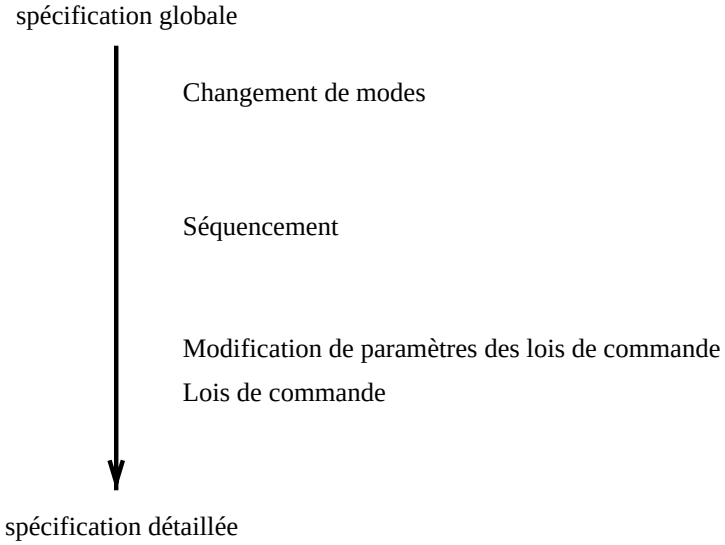


FIG. 5.9 – *Niveaux de détail de spécification des algorithmes*

La figure 5.9 montre la hiérarchie que l'on peut établir entre les différents types d'algorithmes présents dans un système temps réel en fonction du niveau de détails que leur spécification nécessite. Au plus haut niveau de la spécification, correspondant au niveau le plus éloigné du matériel,

on trouve les algorithmes de contrôle qui gèrent les modes de fonctionnement du système. Puis on trouve des algorithmes qui gèrent le séquencement des actions à mettre en œuvre dans chacun des modes de fonctionnement. Enfin, au plus bas niveau, les algorithmes de commande et de modifications des paramètres de ces lois de commande décrivent les actions.

Cette hiérarchie de niveau de spécification rejoint celle utilisée par les ingénieurs pour concevoir des systèmes de production automatisés. Au plus haut niveau, les modes de marche et d'arrêt des machines sont souvent décrits à l'aide d'un document appelé GEMMA [8]. Pour raffiner cette spécification, ils utilisent un langage de type GRAFCET qui leurs permet de définir les séquences d'actions devant être réalisées dans chacun des modes de fonctionnement. Enfin, les actions sont le plus souvent réalisées matériellement et sont donc décrites à l'aide de schémas-bloc, de logigrammes et de schémas électroniques ou pneumatiques.

Cette hiérarchie telle que nous l'avons définie doit se retrouver dans les outils de spécification des systèmes temps réel. L'outil idéal doit, au plus haut niveau, être orienté flot de contrôle pour décrire les changements de modes et le séquencement qui sont raffinés à l'aide d'une spécification flot de donnée pour décrire les lois de commande. C'est pourquoi l'approche choisie par des outils tels que Statemate Magnum et Orccad nous semble mieux adaptée. Néanmoins il est indispensable de pouvoir introduire du flot de contrôle dans un graphe flot de donnée comme cela est réalisé par les outils Mathworks, Matrixx, ASCET-SD et Sildex afin de pouvoir exprimer facilement le contrôle de changement des paramètres dans les lois de commande.

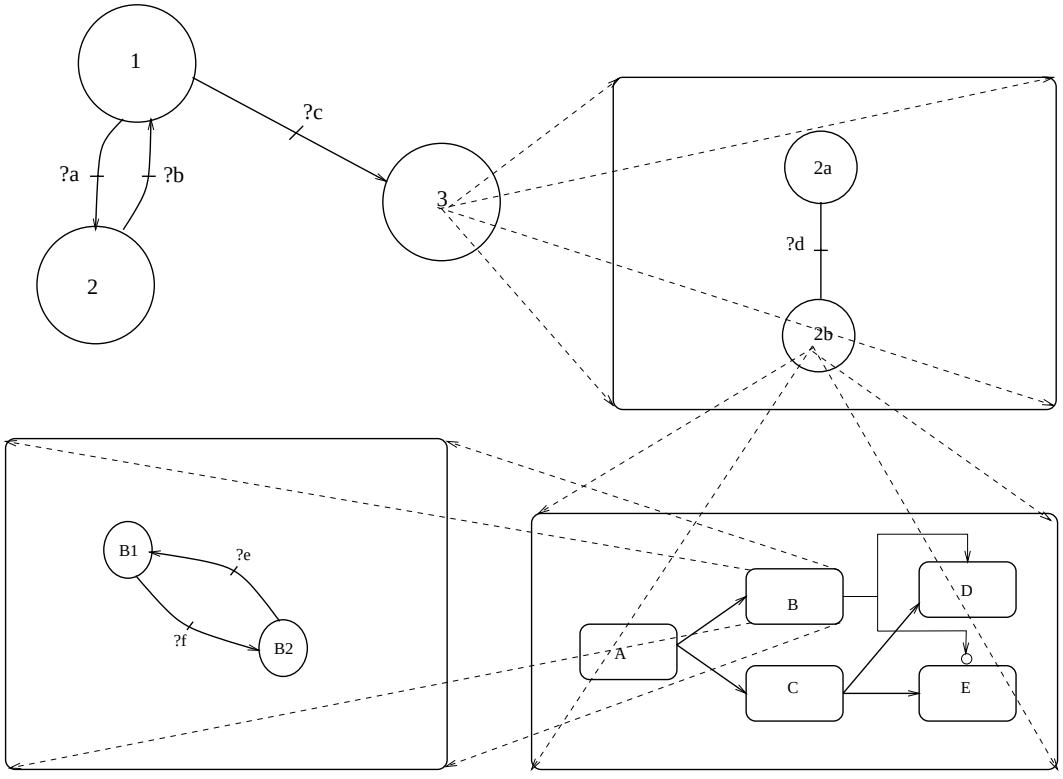


FIG. 5.10 – *Hiérarchie de spécifications multi-formalismes*

La figure 5.10 donne un exemple de spécification hiérarchique multi-formalisme. Un automate

8. Guide d'Etude des Modes de Marche et d'Arrêt

décrit trois principaux modes de fonctionnement d'un système. Il est possible de spécifier le fonctionnement du système dans chacun de ces modes. Ici le mode de fonctionnement "3"est raffiné par une autre spécification de type automate décrivant le séquencement des actions associées aux états de l'automate. Ces actions sont décrites par un graphe flot de donné pour lequel l'exécution de certains sommets est conditionné par la valeur de signaux booléens. Ce type de graphe est appelé *graphe flot de données conditionné*. Sur cet exemple c'est le sommet "B" qui produit le booléen qui conditionne l'exécution des sommets "D" et "E". La fonction de calcul de ce booléen représentée par le sommet "B" est ici décrite à l'aide d'un automate.

5.4.4 Spécification des contraintes temporelles

La spécification des contraintes temporelles est l'un des points faibles de tous les outils. Généralement il est possible de spécifier des contraintes de cadence en imposant des fréquences d'exécutions sur les calculs. Sur certains outils, comme par exemple Simulink, cette contrainte de cadence est décrite graphiquement : l'entrée trigger d'un bloc est relié à un générateur d'impulsion réglé sur la fréquence d'exécution désirée. Dans d'autres outils tels que ORCCAD, la fréquence d'exécution d'un bloc est un paramètre que l'utilisateur doit fournir pour décrire le bloc.

Dans aucun des outils présentés ici il n'est possible de spécifier de contrainte de latence : cette contrainte est implicite.

Implicitement, on associe à ces calculs une contrainte de latence correspondant à leur période d'exécution. Sur certains outils, comme par exemple Simulink, cette contrainte de cadence est décrite graphiquement : l'entrée trigger d'un bloc est relié à un générateur d'impulsion réglé sur la fréquence d'exécution désirée.

5.4.5 Simulation

La modélisation de systèmes discrets est beaucoup moins facile à réaliser que la modélisation des systèmes continus pour lesquels la théorie de l'automatique fournit un ensemble de méthodes et d'outils mathématiques beaucoup plus important. C'est pourquoi, afin de simplifier les calculs, l'automaticien cherche à utiliser un modèle *pseudo-continu* de son système de commande plutôt qu'un modèle discret [19][80][20][68]. La simulation hybride qui consiste à simuler l'ensemble du système automatisé (simulation d'un processus continu relié au système temps réel discret) est indispensable pour valider le modèle à partir duquel le système temps réel doit être conçu. Elle permet de paramétrer les lois de commande afin de compenser les approximations réalisées par la modélisation pseudo-continue. Ce sont les erreurs de modélisation qui ont le plus d'impact sur le temps de développement car elles sont situées à la première étape du cycle de conception. Il faut alors attendre la dernière étape de validation pour les détecter, ce qui entrainera alors la ré-exécution complète du cycle de conception.

Cette simulation, si elle est suffisamment fiable, permet de valider une fois pour toute le modèle du système temps réel et d'évaluer les propriétés dynamiques du système. Ainsi la simulation hybride devrait être systématiquement associée à l'étape de modélisation car c'est seulement par ce biais qu'il est possible de choisir la loi de commande adéquate et de la paramétrer. Grâce à elle, il est, par exemple, possible de vérifier si la période d'échantillonnage du feedback et le gain d'un asservissement permettent de garantir la stabilité du système.

Cette simulation hybride, pour être la plus proche de la réalité, doit nécessairement prendre en compte les retards induits par les temps de calcul car le temps de réponse d'une loi de commande est lié au retard introduit par la latence des calculs qui la composent (cf §1.3.2.3). Elle doit essentiellement prendre en compte les algorithmes de commande, mais il peut aussi être nécessaire de simuler l'ensemble des algorithmes afin de connaître l'influence sur le comportement dynamique du système des discontinuités dûes aux changements de mode.

5.4.6 Vérification

La spécification doit pouvoir être vérifiée formellement du point de vue logique et du point de vue comportemental. La vérification logique a pour but de s'assurer que la spécification réalisée est cohérente du point de vue de la sémantique du langage employé. Elle consiste par exemple dans un graphe flot de données à s'assurer que les horloges des signaux d'entrées d'un sommet du graphe sont identiques et que le graphe ne comporte pas de boucles génératrices d'inter-blocages à l'exécution. La vérification comportementale cherche à garantir que la spécification est conforme au cahier des charges. Elle permet de prouver, entre autre, que certains comportements du système ne seront jamais possibles. Par exemple, si on spécifie un système de commande d'aiguillage d'un système ferroviaire, la vérification comportementale doit permettre de vérifiér qu'aucun train ne pourra accéder à une portion de voie déjà occupée par un autre train. Il pourra aussi être montré, sur un système de boîte de vitesses automatique d'un véhicule, que l'enclenchement de la vitesse la plus faible ne sera jamais déclenchée alors que le véhicule roule à une vitesse élevée. Ce type de vérification est aussi très important car c'est grace à elle que l'on peut garantir la fiabilité du système (à condition que l'implantation soit conforme à la spécification).

Dans le cadre d'un outil de spécification multi-formalisme, la vérification logique doit être réalisée pour chacun des formalismes et à tous les niveaux de la spécification. Dans le cadre d'une interface graphique, cette vérification pourra être réalisée en ligne au fur et à mesure de la construction des graphes. Mieux, elle devra interdire certaines erreurs, par exemple en autorisant seulement des connexions cohérentes entre les éléments du graphe. La vérification comportementale ne s'intéresse qu'au contrôle du système, elle ne sera donc appliquée qu'aux formalismes décrivant cet aspect.

Seule une approche formelle autorise la vérification. Certains outils basés sur ce type d'approche (Statemate Magnum, Sildex et Orccad) permettent d'exporter la spécification vers des outils autorisant la simulation hybride tels que Simulink ou SystemBuild. Orccad peut être associé à un outil de simulation SIMPARC prenant en compte l'architecture matérielle sur laquelle est implantée la spécification afin de pouvoir intégrer les temps de calcul dans la simulation.

5.4.7 Implantation

Un outil de spécification disposant de toutes les fonctionnalités citées précédemment ne sert pas à grand chose si la spécification doit être implantée "à la main" par les ingénieurs, car dans ce cas, il n'est pas possible de garantir que l'implantation est conforme à la spécification et donc que le système possède les propriétés montrées par la vérification logique et comportementale. De plus, l'implantation "à la main" des algorithmes induit un coût financier élevé lié à des temps de développement et de déboguage importants. Ce coût sera d'autant plus élevé que l'on aura cherché à optimiser le code. La maintenance et les modifications ultérieures du système seront très coûteuses.

L'outil de spécification doit donc impérativement pouvoir être connecté à un outil d'implantation ou en intégrer un. Celui-ci doit non seulement prendre en compte les architectures distribuées hétérogènes, mais il doit aussi garantir que l'implantation réalisée est conforme à la spécification, aussi bien du point de vue logique que temporel, et que cette implantation utilise au mieux les ressources matérielles disponibles. Ainsi, le surcoût engendré par l'exécutif doit être minimal et le parallélisme potentiel des algorithmes doit être exploité.

5.4.8 Validation

La validation du système consiste à connecter le système temps réel au processus qu'on cherche à commander, et à observer que les propriétés et le comportement du système automatisé sont conformes au cahier des charges. Cette étape nécessite donc de mesurer et d'observer l'évolution des différents paramètres du processus que le système temps réel cherche à commander. Dans le cadre de la conception de sytèmes embarqués, il est souvent difficile d'intégrer au système les instruments de mesure adéquats. De plus, la structure matérielle intrinsèque du système fournit déjà ces instruments de mesure : c'est le rôle d'un capteur fournissant le signal de feedback à un asservissement que d'observer un paramètre du processus. Ainsi, idéalement, il doit pouvoir être possible, pour faciliter le débogage et valider le système, de pouvoir visualiser l'évolution de certains signaux pendant l'évolution du système automatisé ou de pouvoir la mémoriser afin de l'analyser par la suite. Ces visualisations ou mémorisations doivent pouvoir être intégrées au code de l'application au gré du concepteur du système.

CHAPITRE 6

ÉVOLUTION DE LA MÉTHODOLOGIE AAA

La réalisation d'un outil de conception idéal, ou du moins d'un outil permettant de couvrir avec succès toutes les étapes du cycle de conception, demande de la part des équipes qui le développe, la connaissance et la maîtrise d'un large domaine de compétences. La résolution des problèmes complexes que pose ce type de réalisation nécessite, pour avoir des chances d'aboutir, d'être à la pointe de la recherche dans des disciplines telles que la théorie des langages, la vérification formelle, l'automatique, le parallélisme, etc. Aucune équipe au monde ne regroupe cet éventail de connaissance. C'est sans doute ce qui explique que ce type d'outil n'existe actuellement pas. Chaque outil a ses points forts et ses lacunes le prédisposant à être utilisé seulement pour certaines étapes du cycle de conception. Pour contourner ce problème, les équipes de recherche et de développement collaborent et créent des passerelles entre des outils complémentaires qu'ils développent chacun de leur coté. Pour illustrer cette tendance, on peut par exemple citer le format commun des langages synchrones qui vise à mettre en commun les différents outils de vérifications, de simulation, d'implantation, etc., qui ont été développés pour chaque langage. La plupart des outils que nous avons présentés ont des passerelles vers d'autres outils.

La méthodologie AAA et le logiciel SynDEx qui la supporte n'échappe pas à cette règle. SynDEx est un outil plutôt orienté implantation. Nous cherchons donc à le connecter avec d'autres outils, de spécification, de simulation, etc. L'implantation est généralement le point faible des outils de conception des systèmes temps réel. La génération de code intégré à ces outils est souvent limité à des architectures bien particulières, le plus souvent monoprocesseur. Leur utilisation se limite donc le plus souvent aux applications dont l'architecture matérielle est simple (monoprocesseur), ou bien encore au prototypes coûteux conçus dans les laboratoires de recherche. Le passage du prototype au produit industriel est alors réalisé en implantant et optimisant "à la main" sur une architecture minimale les algorithmes mis au point sur le prototype. SynDEx est conçu pour réaliser un prototypage rapide optimisé des applications : il est capable de générer un code efficace, et est adaptable à un large éventail d'architectures distribuées hétérogènes, ce qui autorise la réalisation

de prototypes complexes dont l'architecture est très proche du produit final.

Par rapport aux propriétés définies pour un outil "quasi-idéal" de conception des applications temps réel embarqués, SynDEx présente plusieurs lacunes qui se situent principalement au niveau de la spécification. Des travaux ont été réalisés pour connecter SynDEx à d'autres outils. Il existe notamment une interface Signal-SynDEx, une interface DC [1][23] en SynDEx. Des travaux sont en cours pour la réalisation d'une interface Scicos-SynDEx[25]. Néanmoins, pour faciliter la connectivité avec des outils de spécification de haut niveau, il est nécessaire d'étendre le formalisme de SynDEx afin de prendre en compte le multi-formalisme, la hiérarchie et la possibilité de faire appel à des librairies. Actuellement les contraintes temporelles ne sont pas prises en compte par SynDEx, il cherche dans tout les cas à trouver une implantation minimisant le chemin critique c'est-à-dire la durée nécessaire à l'exécution de l'ensemble des calculs qui composent l'algorithme. Nous chercherons donc à intégrer à SynDEx un support pour la spécifications des contraintes temporelles ainsi que la prise en compte de ces contraintes lors de la distribution, de l'ordonnancement. Ces derniers aspects font l'objet de la partie suivante de ce document.

6.1 Spécification hiérarchique à l'aide de librairies

Les travaux en cours sur le sujet ont abouttis à la conception d'une nouvelle interface graphique et d'un langage baptisé HGDL [2] qui permet une description de la hiérarchie et de l'utilisation de bibliothèques d'opérations et d'opérateurs sous une forme textuelle. Cette nouvelle interface graphique utilise HGDL pour la sauvegarde fichier du graphe décrit par l'utilisateur. Il est donc possible de décrire la spécification soit à l'aide de l'interface graphique, soit directement sous forme textuelle.

6.2 Spécification d'un ensemble de contraintes temporelles

Le modèle de graphe de d'épendances factorisés décrit §5.3.7 permet de spécifier aussi bien la répétition infinie induite par les interactions de l'algorithme applicatif avec l'environnement, que la répétition infinie de motifs de graphe. Cela correspond à la version sous une forme de graphe, des spécifications textuelles d'algorithmes utilisant des nids de boucles [30][8][45] de type Fortran. Ces graphes peuvent être utilisés pour spécifier des contraintes temporelles multiples.

6.2.1 Graphes factorisés

Les graphes factorisés sont basés sur le même principe que celui de la factorisation d'équations mathématiques. Par exemple, le produit $S = C \cdot e$ d'une matrice C par un vecteur e peut s'écrire de la manière suivante :

$$\begin{cases} S_1 = C_{11} \cdot e_1 + C_{12} \cdot e_2 + ... + C_{1n} \cdot e_n \\ S_2 = C_{21} \cdot e_1 + C_{22} \cdot e_2 + ... + C_{2n} \cdot e_n \\ \vdots \\ S_n = C_{n1} \cdot e_1 + C_{n2} \cdot e_2 + ... + C_{nn} \cdot e_n \end{cases}$$

1. Declarative Code, format commun des langages Synchrones
2. Hierarchical Graph Description Language

mais aussi sous une forme factorisée :

$$(S_i)_{1\leq i\leq n} = \sum_{j=1}^{n} C_{ij} \cdot e_j$$

Les graphes factorisés permettent le même type de simplification. La figure 6.1 représente le produit d'une matrice c de dimension 3x3 par un vecteur e de dimension 3 :

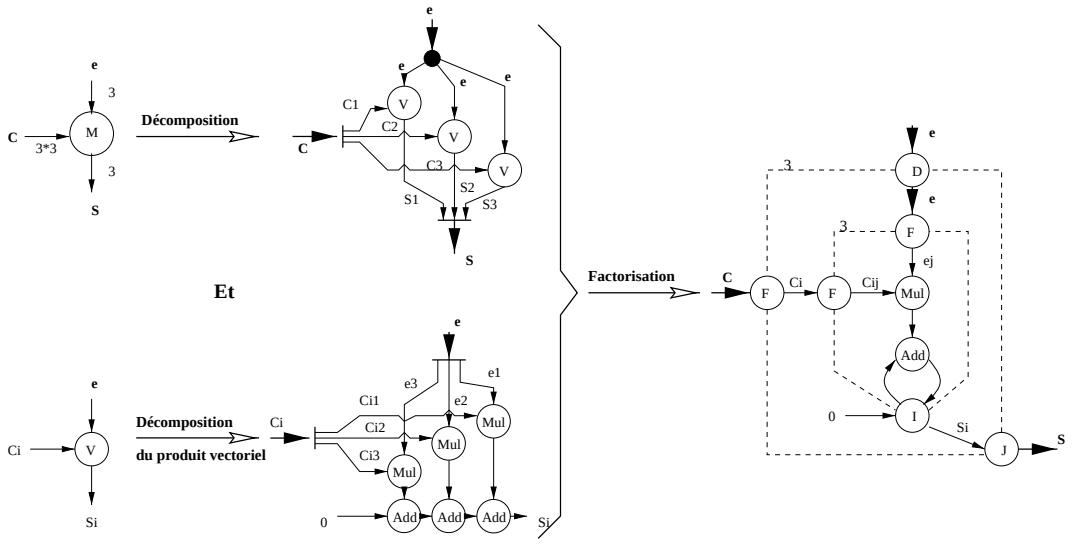


FIG. 6.1 – *Décomposition et factorisation d'un produit matrice-vecteur*

Cette factorisation, tout en gardant la sémantique opératoire et l'ordre partiel du graphe permet de réduire la taille de la spécification en mettant en évidence ses parties régulières.

Les graphes factorisés font appel à quatre nouveaux types de sommets, les *macro-ports*, qui délimitent les frontières, mises en évidence par les pointillés, entourant le motif de la factorisation pour le séparer du reste du graphe :

- le macro-port **Diffuse** "entre" en diffusant le vecteur e à tous les produits scalaires V,
- le macro-port **Fork** "entre" en factorisant un groupe d'arcs d'entrée,
- le macro-port **Join** "sort" en factorisant un groupe d'arcs de sortie,
- le macro-port **Iterate** "entre et sort" (par 0 et Si) en factorisant un groupe d'arcs inter-motifs.

6.2.2 Spécification de contraintes temporelles à l'aide de graphes factorisés

6.2.2.1 Contraintes de cadence

Les systèmes temps réels complexes sont soumis à différentes contraintes temporelles de type latence et cadence afin de respecter les fréquences d'échantillonnage et les temps de réponses acceptables définis par l'automaticien de manière à garantir la stabilité et les performances requises du

système. En réalité, nous avons vu dans la première partie de ce document que l'automaticien a une grande latitude sur le choix de la fréquence d'échantillonnage d'un signal. En théorie, dans la plupart des cas, il est possible de choisir la même fréquence d'échantillonnage pour tous les capteurs de l'application. Cette solution présente l'intérêt de simplifier considérablement la modélisation des lois de commandes, elle a par contre un désavantage majeur, celui de nécessiter une architecture puissante pour que tous les calculs des lois de commandes puissent être effectués avec cette contrainte. Pour minimiser l'architecture matérielle, il faut minimiser la puissance de calcul nécessaire à l'exécution du logiciel et donc minimiser le volume de calcul qui doit être réalisé dans l'intervalle de temps donné par les contraintes temps réel. Dans ce cas, la stratégie consiste à définir des fréquences d'échantillonnages minimales pour chaque signaux et d'exécuter les calculs qui traitent ces échantillons le moins souvent possible. Ainsi, on est amené à définir des fréquences d'exécutions pour chacune des opérations qui constituent l'algorithme. En fait, le terme de fréquence d'exécution des calculs, bien qu'utilisé par toute la communauté de l'informatique temps réel, nous semble mal adapté, car cette notion de fréquence induit une notion de périodicité d'exécution. Or, la périodicité d'exécution ne se justifie que sur les opérations d'échantillonnage, et de mise à jour des signaux, puisque l'automaticien à défini les lois de commandes en posant cette hypothèse. Les opérations de calcul ne réalisant pas d'échantillonnage doivent être exécutées autant de fois qu'il y a de données à traiter, mais pas nécessairement de façon périodique. C'est pourquoi, de manière générale, nous préférons parler de répétition des opérations plutôt que de fréquence d'exécution. Les opérations d'entrées-sorties qui réalisent les échantillonnages sont des opérations particulières qui doivent être répétées périodiquement, on peut donc ici parler de fréquence d'exécution.

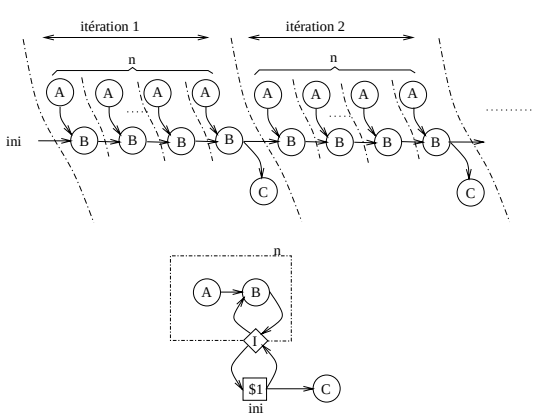
L'intérêt de contraintes temporelles multiples est donc de permettre une implantation pour laquelle certains sous-ensembles d'opérations du graphe d'algorithme (ou motifs) pourront être exécutés moins souvent que d'autres de manière à réduire le volume de calcul moyen que le calculateur devra réaliser et ainsi minimiser la puissance de calcul nécessaire à la réalisation du système.

Le but des graphes factorisés est justement de permettre la spécification de répétition de motifs dans un graphe. C'est pourquoi elle est bien adaptée pour la spécification des contraintes de cadences. En effet, à chaque opération du graphe peut être associé un nombre de répétition par rapport à une itération du graphe complet. Ainsi, si on associe une contrainte de cadence sur une entrée ou sur une sortie du graphe par le biais d'une période d'exécution de celle-ci, on peut déterminer la période d'exécution de toutes les autres entrées-sorties et donc les contraintes de cadences sur leur exécution.

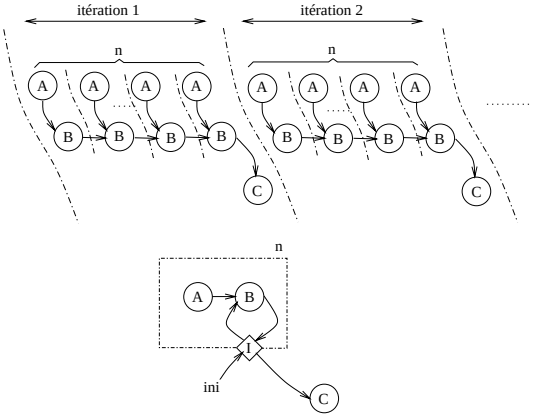
Nous présentons figure 6.2 et 6.3 des exemples de différents schémas d'exécution possible que l'automaticien peut chercher à obtenir entre des opérations devant être exécutées à des rythmes différents (sous-échantillonnage, interpolation, etc.). Pour chacun de ces schémas d'exécution, nous donnons le graphe factorisé correspondant.

6.2.2.2 Contraintes de latence

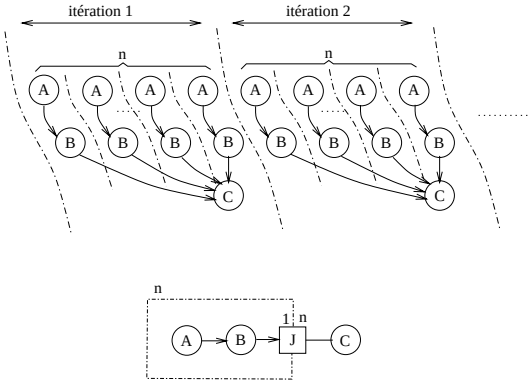
Chaque sortie du graphe factorisé est fonction d'une ou plusieurs entrées. La latence définie le temps qui s'écoule entre la date de début d'exécution de la première entrée et la date de fin d'exécution de la sortie. On cherche à garantir que le temps qui s'écoule entre l'instant où on réalise une entrée et l'instant où on réalise une sortie est borné par la contrainte de latence, c'est à dire que toutes les opérations qui interviennent dans le calcul de la sortie sont toujours réalisées en un temps inférieur à la contrainte de latence. Dans les graphes factorisés on peut définir trois types de latence



(a) Sous-échantillonnage d'un filtre récursif

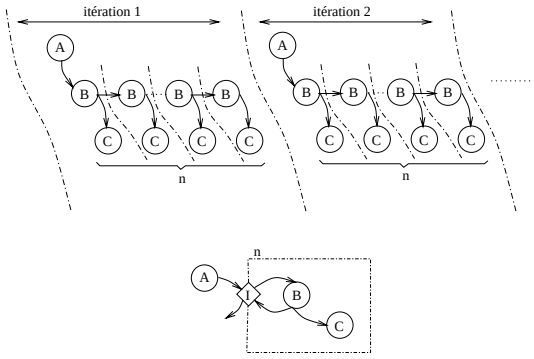


(b) Sous-échantillonnage d'un filtre non récursif

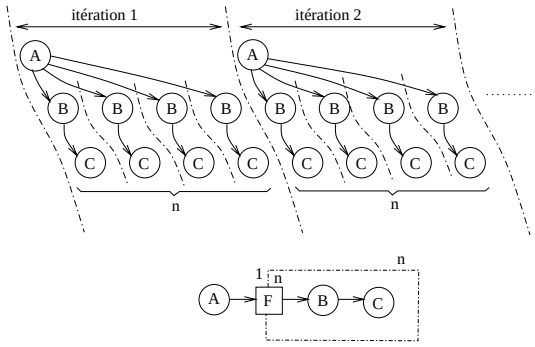


(c) Traitement de données par bloc

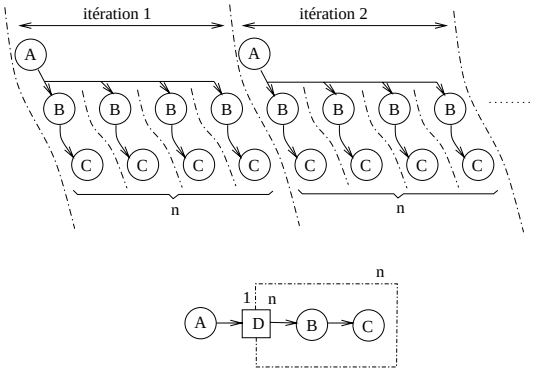
FIG. 6.2 – *Exemple de représentation factorisée, rythme d'exécution rapide → lent*



(a) Interpolation



(b) Traitement par blocs



(c) Sur-échantillonage

FIG. 6.3 – *Exemple de représentation factorisée, rythme d'exécution lent → rapide*

que nous présentons sur l'exemple d'un filtre récursif (fig. 6.4). L'opération E est l'échantillonnage d'un signal, l'opération C calcule pour chaque entrée E une sortie filtrée qui est aussi fonction de la sortie précédente. L'opération S sous-échantillonne la sortie du filtre.

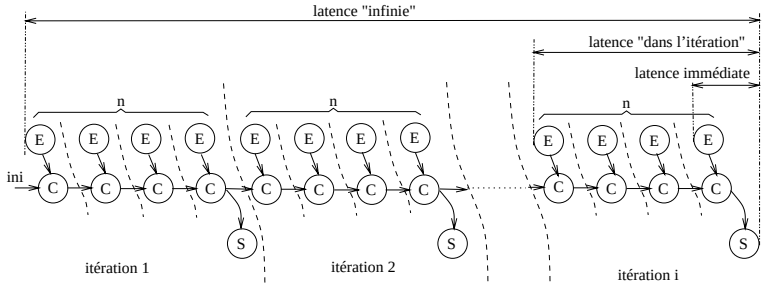


FIG. 6.4 – *Trois type de latence*

Latence "infinie" : l'opération S produit une sortie fonction de deux entrées, la valeur produite par E et sa sortie précédente. La m^{ime} sortie de C dépend de l'entrée E_m et de sa $m-1^{ime}$ sortie qui est elle-même fonction de l'entrée E_{m-1} et de sa $m-2^{ime}$ sortie, etc. Ainsi la première entrée qui intervient dans le calcul d'une sortiee S xiste entre est la toute première entrée E_1. Cet exemple montre que la latence entre une sortie et la première entrée qui intervient dans le calcul de cette sortie peut croître à chaque nouvelle itération du graphe. Dans ce cas, la latence ne peut être bornée, c'est pourquoi nous ne nous intéresserons pas à cette latence que nous avons baptisé "latence infinie".

Latence "dans l'itération" : la latence infinie prend en compte le passé lointain du signal, mais plus une valeur d'entrée est ancienne, moins son influence sur le résultat de sortie se fait ressentir. C'est une autre raison pour laquelle la latence "infinie" présente peut d'intérêt. Pour prendre en compte le passé du signal dans la latence, et pour éviter de prendre en compte un passé trop lointain, il est préférable de s'intéresser à une latence qui représente l'intervalle de temps qui s'écoule entre la première entrée depuis la sortie précédente et la sortie calculée à partir de ces entrées. Cette latence "dans l'itération" ne prend ainsi en compte que les n dernières entrées, c'est à dire toutes les entrées d'une itération du graphe.

Latence "immédiate" : il est aussi possible de définir une latence entre la dernière entrée et la sortie. A partir de cette latence ont peut déterminer la valeur des deux autres latences. La latence infinie est égale à $n * i$ la latence immédiate, et la latence "dans l'itération" est égale à n fois la latence "immédiate". C'est cette latence que nous spécifierons.

La latence représente le temps qui s'écoule entre l'arrivée d'un événememement d'entrée et la production de l'événement de sortie déclenchée par l'arrivée de l'événement d'entrée. Dans le langage SynDEx, il existe une opération, `memory`, qui permet de retarder un signal d'une ou plusieurs itérations du graphe. Il est donc possible de spécifier un retard entre une entrée et une sortie en intercalant entre les deux une opération `memory`. Lorsque le calcul d'un signal de sortie dépend de plusieurs entrées il faut, pour spécifier une latence par ce biais, intercaler le même retard entre toutes les entrées et la sortie afin de ne pas décorréler temporellement les signaux. Afin d'éviter ce problème et pour faciliter la spécification de la latence d'une sortie, nous proposons d'ajouter aux opérations de sortie, en plus de son nombre d'itérations et de sa durée d'exécution, un attribut représentant un

retard identique sur toutes ses entrées. Cet attribut permet alors de spécifier une latence immédiate multiple de la période d'exécution de la sortie.

6.3 Multi-formalisme de spécification

L'état de l'art a montré qu'un outil de spécification devait être multi-formalisme pour faciliter la spécification des différents aspects du système temps réel. Ainsi un langage flot de contrôle facilite la description des algorithmes de contrôle et un langage flot de données favorise la description des algorithmes de commande [67][36]. Le multi-formalisme facilite la spécification, mais il devient un handicap lorsque l'on cherche à réaliser une implantation efficace de celle-ci : le formalisme de type flot de données permet de mettre en évidence le parallélisme potentiel des algorithmes qu'il faudra mettre en correspondance avec le parallélisme disponible de l'architecture matérielle. C'est pourquoi, la méthodologie AAA et le logiciel SynDEx sont basés exclusivement sur ce formalisme.

Afin de permettre un interfaçage de SynDEx avec d'autres outils de spécification orientés flot de contrôle (ce pourrait par exemple être une interface ORCCAD-SynDEx), il est nécessaire d'introduire le multi-formalisme dans SynDEx. Pour satisfaire les contraintes liées à la spécification et celles liées à l'implantation efficace de cette spécification, nous proposons ici l'étude de la transformation d'une spécification flot de contrôle vers une spécification flot de donnée. Cette transformation permettra l'élaboration d'une interface autorisant la spécification des aspects contrôle d'une application avec un formalisme de type flot de contrôle, et des aspects commande de cette même application à l'aide d'un formalisme flot de données. Un traducteur automatique basé sur cette transformation et intégré à l'interface permettra alors d'unifier ces deux formalismes en un seul de type flot de données afin d'exhiber le parallélisme potentiel des algorithmes ainsi spécifiés, en vu de réaliser une implantation efficace (fig. 6.5).

Il existe différentes approches permettant de mixer une spécification flot de contrôle avec une spécification flot de donnée :

- la plus simple, celle choisie dans des outils tels que Mathworks ou Matrixx, consiste à compiler une spécification flot de contrôle de type automate en une fonction décrite dans un langage de programmation (généralement langage C) et d'encapsuler cette fonction dans un bloc d'un graphe flot de données ;

- une autre approche consiste à étendre la sémantique d'un langage de spécification flot de données en définissant des macro-primitives basées sur les primitives du langage, macros-primitives permettant de décrire facilement des automates. C'est l'approche qui a été retenue par Éric Rutten avec SignalGti, une extension du langage Signal ;

- enfin, une dernière approche consiste à transformer la spécification flot de contrôle en une spécification flot de données. Cette spécification flot de donnée décrivant ainsi les aspects contrôle est alors mixée avec la spécification flot de donnée des aspects commande réalisée par l'informaticien. Cette approche est celle choisie dans l'outil Sildex. Il existe aussi une transformation du langage Argos en langage DC[3] et de Statecharts et Activitycharts en Signal[11].

3. Declarative Code, format commun des langages synchrones

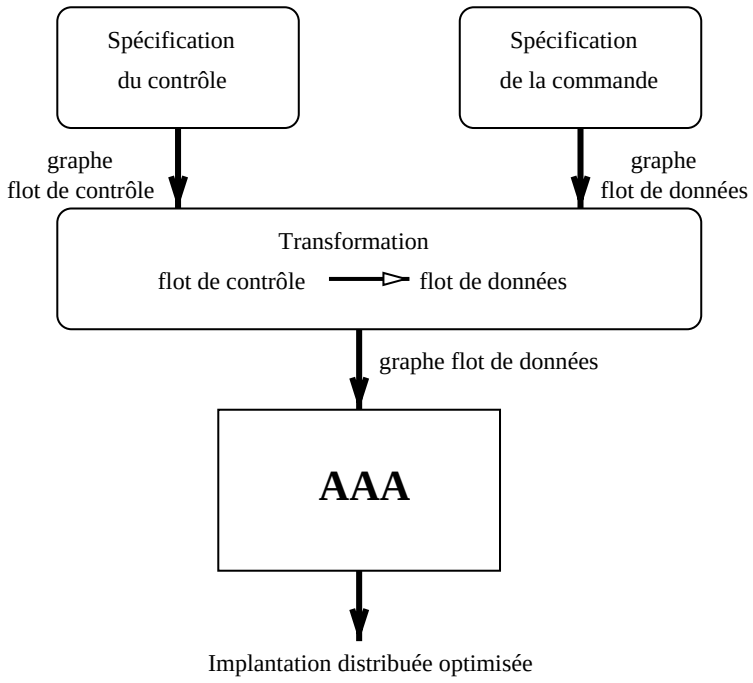


FIG. 6.5 – *Unification d'une spécification multi-formalisme en vue d'une implantation optimisée*

Dans le cadre de notre étude, la première approche ne nous semble pas satisfaisante essentiellement pour deux raisons ; d'une part, elle ne permet pas de réaliser de vérification globale sur la spécification. Des vérifications formelles peuvent être réalisées sur la spécification flot de données, d'autres vérifications peuvent être faites sur la spécification flot de contrôle, mais cela ne garantit pas que la composition des deux spécifications soit correcte. D'autre part, il n'est pas possible de tirer au mieux partie de l'architecture distribuée du calculateur, car l'ensemble des calculs liés à la spécification du contrôle est encapsulé dans une fonction qui ne peut alors pas être distribuée. Nous présentons ici les travaux relatifs aux deux autres types d'approches, après un bref rappel sur les automates qui facilitent la description des aspects contrôle des systèmes. Nous proposons ensuite une transformation flot de contrôle-flot de donnée autorisant une distribution et un ordonnancement efficace des calculs et des communications sur un calculateur multiprocesseur.

6.3.1 Rappels sur les automates

Le comportement d'un système discret peut être décrit par une *machine séquentielle à états finie* ou *automate à états fini*. Un automate à états fini est constitué d'un ensemble fini d'états S, d'un état initial s_0, d'un ensemble fini de signaux d'entrée I, d'un ensemble fini de signaux de sortie O, d'une fonction de transition δ définissant la valeur de l'état suivant en fonction de la valeur des entrées et de l'état courant, ainsi que d'une fonction ω de calcul des sorties. Un automate particulier est donc décrit par le sextuplet (S,s_0,I,O,δ,ω).

Il existe deux types d'automates, les automates dits de *Moore* et ceux dit de *Mealy*[2][21]. La différence réside dans la fonction de sortie ω. Pour un automate de type Mealy les sorties de l'automate sont fonctions des entrées et de l'état courant $\omega : I \times S \rightarrow O$, alors que dans un automate de type Moore les sorties sont fonctions des états, $\omega : S \rightarrow O$

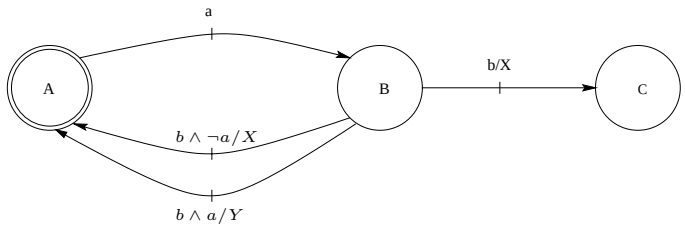


FIG. 6.6 – *Représentation d'une machine de Mealy*

La figure 6.6 est la représentation graphique d'un automate de type Mealy. Chaque sommet du graphe représente un état. Les flèches représentent les transitions possibles entre un état de départ et un état d'arrivée. A chaque flèche peut être associé un couple $B(I)/o$, où $B(I)$ est une équation booléenne à variables dans I et o un ensemble de signaux appartenant à O. $B(I)$ définit une condition de franchissement de la transition. L'ensemble des signaux de sorties o est émis lorsque la transition est franchie. L'état initial s_0 peut être repéré soit par une flèche représentant une transition sans état de départ, soit comme ici par un double cercle.

La représentation d'un automate de Moore (Fig. 6.7) diffère de celle d'un automate de Mealy seulement par la position des signaux de sorties ; ceux-ci ne sont plus associés à une transition, mais à un état.

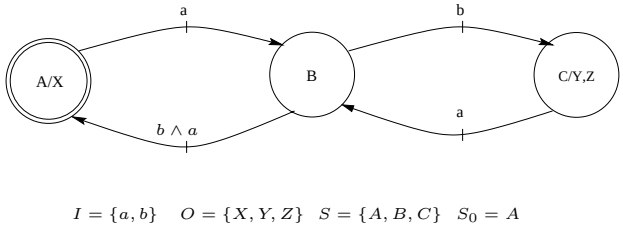


FIG. 6.7 – *Représentation d'une machine de Moore*

Il existe toujours l'équivalent Moore d'une machine de Mealy et réciproquement [22]. A comportement équivalent, un automate de Moore utilise une fonction ω de calcul des sorties moins complexe, mais nécessite généralement plus d'états que l'automate de Mealy.

6.3.2 Unification dans un formalisme flot de données, d'une spécification multi-formalisme : état de l'art

Différents travaux ont été réalisés sur le sujet. Nous présentons ici un échantillon représentatif de trois approches différentes.

6.3.2.1 Signal *GTi*

L'approche choisie par E.Rutten [74][60][11] est différente des autres approches. Plutôt que de transformer un formalisme flot de contrôle afin de l'intégrer dans une spécification flot de donnée, il propose une extension du langage Signal en construisant sur les primitives du langage de nouvelles

primitives permettant de définir des intervalles de temps dans lesquels des processus Signal peuvent être exécutés.

Cette extension de Signal appelée Signal *Gti* fournit entre-autres trois primitives qui peuvent être utilisées pour spécifier des machines à états finies :

- la primitive `I:=]A,B]` définit un signal I présent sur l'intervalle de temps délimité par l'occurence d'un signal A et l'occurence d'un signal B ;
- `P on I` définit une tâche comme l'exécution d'un processus Signal P sur l'intervalle I ;
- `C in I` définit un signal comme restriction du signal C sur l'intervalle I. Dans l'intervalle de temps I ce signal recopie les valeurs de C, hors de l'intervalle l'horloge du signal n'est pas présente, le signal n'est donc pas défini.

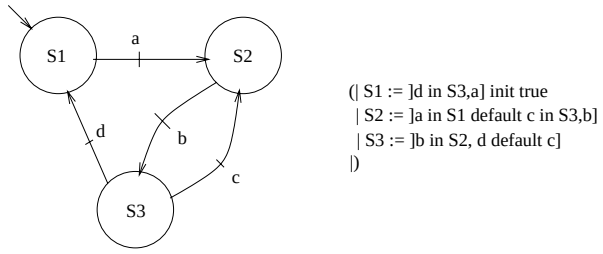


FIG. 6.8 – *Spécification d'un automate en Signal* Gti

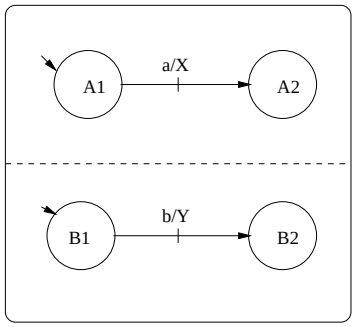
La figure 6.8 représente un automate et sa description en Signal *Gti* à l'aide de ces trois primitives. Le principe est de représenter chaque état de l'automate par un signal booléen défini par un intervalle. Ainsi à chaque état est associé un intervalle de temps sur lequel des calcul pourront être réalisés. Le début de l'intervalle représentant un état est défini par la validation d'une transition entrant dans l'état, c'est à dire par un `default` sur le calcul de toutes les transitions arrivant dans l'état. Le calcul des transitions valides consiste pour chaque transition à définir un signal booléen dont l'horloge est définie lorsque l'état de départ de la transition est actif. Le calcul de cette transition peut s'écrire `cond in Idépart` où `cond` est la condition de franchissement de la transition et `Idépart` est l'intervalle représentant l'état de départ de la transition. La fin de l'intervalle représentant l'état est défini par un `default` sur le calcul des transitions sortantes de l'état. Enfin, l'état initial doit être initialisé à la valeur `vrai`.

Cette approche à l'avantage de permettre la spécification des aspects contrôle et des aspects commande avec un seul formalisme, ce qui facilitera la vérification comportementale.

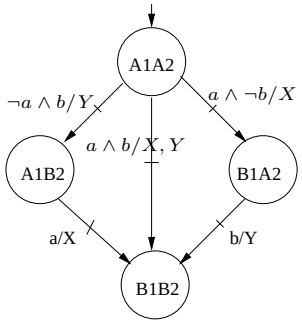
6.3.2.2 Transformation d'Argos en DC

Dans [59] F. Maraninchi et N. Halbwachs décrivent les règles de transformation du langage Argos en un ensemble d'équations booléennes du langage DC. *Argos*[58] peut être considéré comme une variante de Statecharts. C'est un langage graphique hiérarchique flot de contrôle permettant de décrire un système réactif par composition d'automates de types Mealy. Les opérateurs de composition sont le produit synchrone, l'encapsulation et le raffinement.

Le *produit synchrone* permet de définir le comportement d'un système à l'aide de plusieurs automates parallèles. La figure 6.9(a) représente un exemple de description d'un système à l'aide de 2 automates de Mealy mis en parallèle. Le produit synchrone permet de construire la machine de Mealy équivalente (fig. 6.9(b)).



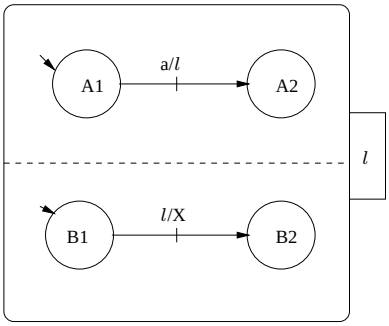
(a) Produit synchrone



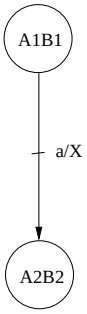
(b) Comportement du produit synchrone

FIG. 6.9 – *Produit synchrone d'automates de Mealy en Argos*

L'*encapsulation* consiste à décrire le comportement du système à l'aide d'automates parallèles synchronisés par des signaux locaux. La figure 6.10 montre le comportement d'un système décrit par 2 automates parallèles synchronisés par un signal local l.



(a) Encapsulation

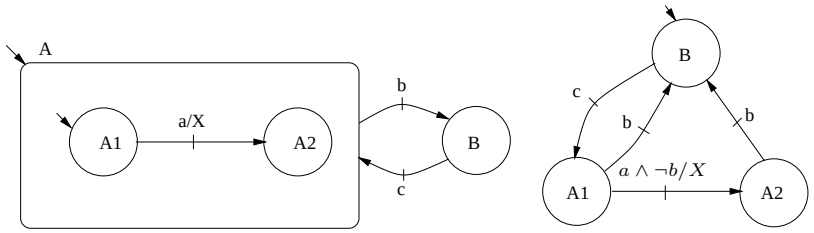


(b) Comportement de l'encapsulation

FIG. 6.10 – *Encapsulation d'un automate de Mealy en Argos*

Le *raffinement* des états d'un automate est présenté figure 6.11. Sur cet exemple le comportement du système est décrit par deux états : A et B. Le raffinement permet ici de décrire plus finement le comportement du système lorsqu'il est dans l'état A.

DC est un langage de manipulation de flots. En DC un *flot* est un objet qui maintient une valeur à chaque instant du programme. Si x est un flot alors x_n est la valeur de x à la n^{ime} réaction (ou n^{ime}instant) du programme. La traduction décrite dans le document s'appuie sur deux opérateurs



(a) Raffinement

(b) Comportement du raffinement

FIG. 6.11 – *Raffinement d'un automate de Mealy en Argos*

DC de définition de flots : les *équations* et les *mémorisations.* Une équation $EQU(i, b, a, v)$ définit un flot i résultant d'une expression booléenne b, actualisée lorsque l'expression a est vrai soit :

$$i_0 = \begin{cases} b_0 \text{ si } a_0 \\ v \text{ sinon} \end{cases}, \; i_n = \begin{cases} b_n \text{ si } a_n \\ i_{n-1} \text{ sinon} \end{cases} \quad \text{avec n} > 0$$

Une mémorisation $MEM(i, b, a, v)$ définit un flot i résultant de la mémorisation de l'expression booléenne b, rafraichie lorsque a est vrai, la valeur initiale de la mémoire étant donnée par v, soit :

$$i_0 = \text{v}, i_{n+1} = \begin{cases} b_n \text{ si } a_n \\ i_n \text{ sinon} \end{cases}$$

Sans entrer dans les détails, le principe de la traduction définie par Maraninchi et Halbwachs consiste à associer à chaque état de l'automate hiérarchique un flot défini par une mémorisation $MEM(S, rp, TRUE, initv)$. Lorsqu'un état est actif (resp. inactif) le flot booléen S correspondant est $vrai$ (resp. $faux$). rp est une fonction booléenne prenant en compte les transitions de l'automate de manière à ce que S soit $vrai$ lorsqu'il n'existe pas de transition sortante valide de l'état correspondant alors que l'automate est déjà dans cet état, ou qu'il existe une transition valide entrant dans cet état. Chaque sortie O est définie par une équation $EQU(o, rp, TRUE, FALSE)$. rp est une équation booléenne qui prend la valeur $vrai$ lorsque l'une des transitions auquel appartient o est valide et que l'état de départ de cette transition est actif. Les auteurs prouvent que cette transformation permet de prendre en compte l'encapsulation, le raffinement et le produit synchrone, et que le programme DC obtenu conserve les propriétés de la spécification ARGOS.

6.3.2.3 Transformation d'un automate de Mealy en processus Signal dans l'outil Sildex

L'outil Sildex[87] fournit une interface permettant de décrire graphiquement des automates traduits ensuite automatiquement en Signal. Les définitions Signal de l'automate sont alors encapsulées dans une "boite" sildex pour laquelle les entrées et sorties correspondent aux entrées et sorties de l'automate.

Les automates qu'il est possible de décrire avec Sildex sont des automates de Mealy. Les entrées de l'automate sont des signaux de n'importe quel type. La condition de franchissement d'une

transition est le résultat d'un calcul sur les entrées exprimé sous la forme d'une expression Signal. Il est ainsi possible de valider une transition par exemple sur le dépassement d'une valeur de seuil de la valeur d'un signal d'entrée. Les sorties sont de type *event* ou *boolean*. La validation (resp. l'invalidation) d'une sortie de l'automate consiste à rendre *présent* (resp. *absent*) le signal de sortie si celle-ci est de type *event* ou a lui assigner la valeur *vrai* (resp. faux) si celle-ci est de type *boolean*.

La spécification d'un automate peut être indéterministe. Supposons par exemple que d'un même état S partent deux transitions T_1 et T_2 arrivant chacune à un état différent et que la valeur d'un signal booléen d'entrée i_1 soit associé à T_1 et i_2 à T_2. Si i_1 et i_2 sont *vrai* simultanément alors les deux transitions peuvent être validées simultanément. Dans un automate de Mealy (ou de Moore) un seul état est actif à la fois, dans ce cas il n'est pas possible de prévoir quelle transition sera franchie et donc dans quel état sera le système. Pour éviter de telles indéterminations Sildex demande à l'utilisateur d'associer des priorités aux transitions partant d'un même état. Dans une situation d'indéterminisme, c'est la transition de plus haute priorité qui est franchie.

Le principe de la transformation de l'automate en processus Signal consiste à coder l'état courant de l'automate par un signal de type *entier*, chaque état possible de l'automate correspondant à une valeur particulière de ce signal. A chaque transition de l'automate est associé un Signal de type `event` dont la présence traduit son franchissement. Celui-ci s'effectue lorsque l'état de départ est actif, que la condition est valide et qu'aucune transition de priorité supérieure partant du même état n'est franchie. Chaque sortie est le résultat de l'union ($default$ du langage Signal) des signaux de toutes les transitions auxquelles est associée la sortie.

6.3.3 Etude d'une transformation autorisant une implantation efficace sur un calculateur distribué

Nous avons vu dans les transformations présentées précédemment que les calculs impliqués dans les aspects contrôle d'un système sont principalement des opérations logiques sur des signaux booléens. Ceux-ci sont donc généralement négligeables comparés au fort volume de calcul que représente l'aspect commande (traitement du signal et des images, loi de commande) d'un système. C'est pourquoi les articles relatif à l'état de l'art présenté ici ne s'intéressent pas aux temps d'exécution des spécifications obtenues par les transformations décrites. Néanmoins, lorsqu'on cherche à implanter de tels algorithmes de contrôle sur une architecture distribuée multi-processeurs il n'est plus possible de négliger ces temps d'exécutions car ils peuvent donner lieu à des communications inter-processeurs que l'on ne peut négliger.

En effet, les transformations sont généralement toutes basées sur le même principe. A chaque état de l'automate et à chaque transition est associé un signal. A chaque instant (au sens instant logique d'une spécification synchrone) tous ces signaux sont calculés. Par exemple, un signal correspondant à un état est une fonction des signaux de transition sortants de cet état. Lorsqu'une condition de transition de sortie de cet état est valide (signal de transition vrai) et que l'état est actif alors l'état est désactivé (le signal correspondant devient faux) et l'état d'arrivée de la transition est activé. Cette méthode revient à calculer la valeur de toutes les transitions et de tous les états de l'automate à chaque instant d'exécution. Autrement dit, à chaque itération du programme on parcourt entièrement l'automate. Or, les conditions de transition sont souvent fonction de résultats de calcul réalisés sur des valeurs issues des capteurs. Dans une application distribuée temps réel les capteurs et les calculs sont distribués sur les différents processeurs de l'architecture. Pour pouvoir calculer l'automate il faut donc communiquer à la fonction de calcul de l'automate la valeur de chacun des

signaux intervenant dans le calcul des conditions des transitions ce qui peut être très coûteux en communications inter-processeurs.

Nous proposons ici une transformation permettant de générer un graphe flot de données à partir d'un automate de Moore. Cette transformation consiste à ne calculer qu'une partie de l'automate à chaque instant et donc à ne communiquer que les valeurs des signaux liés à ces calculs. Ainsi, au lieu de calculer toutes les transitions et tous les états de l'automate on ne calcule que les transitions sortant de l'état actif courant. On se repose ici sur le fait que dans un automate un seul état n'est actif à la fois, il n'est donc pas nécessaire de calculer les états et les transitions qui ne sont pas reliées à l'état actif courant. Nous visons avec cette technique à réduire les communications inter-processeurs.

6.3.3.1 Calcul de l'état courant

Plutôt que de réaliser séquentiellement les calculs du nouvel état de l'automate puis les lois de commandes à réaliser dans l'état calculé il est préférable, afin d'exploiter au mieux le parallélisme, de calculer la valeur du prochain état en même temps que les calculs des lois de commandes de l'état courant. Dans ce cas, la valeur de l'état courant est la valeur calculée dans l'itération précédente du graphe. Notre transformation va donc consister à définir l'état courant par une opération mémoire. Dans la transformation Argos-DC, pour chaque état de l'automate est associé un signal booléen représentant l'activation/non-activation de l'état. En partant de l'hypothèse qu'un seul état ne peut être actif à la fois, pour simplifier le calcul du nouvel état, l'état sera représenté dans notre transformation par un entier plutôt que par n booléen. Cet entier pourra prendre n valeurs représentant chacune un des n états de l'automate.

Le calcul du prochain état consiste à n'évaluer que les transitions sortantes de l'état courant, ainsi à chaque état de l'algorithme est associé un calcul de transition. Il n'est activé à l'exécution que lorsque l'état courant du système est l'état de départ de la transition. Le résultat du calcul est réinjecté dans la mémoire représentant l'état du système par l'intermédiaire d'un opérateur `Default` à n entrées.

Un opérateur `default` possède deux entrées et une sortie. La première entrée est prioritaire par rapport à la seconde. Ainsi, si un signal est présent sur la première entrée alors la sortie recopie la valeur de cette première entrée. Si par contre, le signal n'est pas présent sur l'entrée prioritaire et si un signal est présent sur la seconde entrée alors la sortie recopie la valeur de la seconde entrée. Lorsque les deux signaux sont présents simultanément c'est la valeur de la première entrée prioritaire qui est recopiée. Le `default` produit donc un signal dont l'horloge est l'union des horloges de ces signaux d'entrées. Le `ndefault` est la généralisation à n entrées du `default`. Les entrées ont des priorités croissantes, c'est à dire que l'entrée i est prioritaire sur l'entrée $i + 1$.

Dans notre transformation un seul état est actif à la fois et il y a toujours un état actif. Lorsque aucune transition n'est valide il n'y a pas de transition d'état. La valeur du prochain état doit être la même que la valeur de l'état courant. C'est pourquoi nous ajoutons une entrée supplémentaire de moindre priorité au `ndefault`. Cette entrée est reliée à la sortie de la mémoire d'état afin de prendre en compte la valeur de l'état courant. Ainsi la sortie du `ndefault` définit un signal possédant une valeur à chaque itération, valeur définissant la valeur du prochain état du système.

La figure 6.12 illustre la structure générale du graphe flot de donnée codant le calcul d'un automate à n états obtenus par notre transformation.

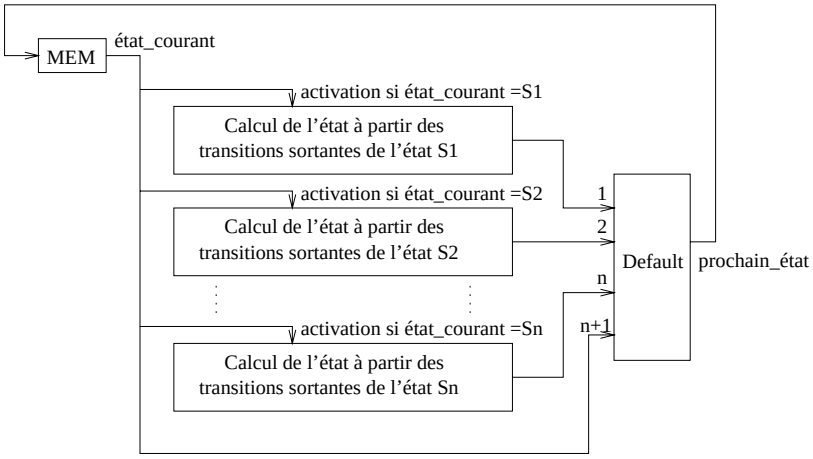


FIG. 6.12 – *Principe de codage du calcul de l'état actif d'un automate à n états*

6.3.3.2 Calcul des transitions sortantes d'un état

La valeur du nouvel état de l'automate est fonction de l'état courant et des transitions associées à l'état courant. Un état de l'automate ne devient actif que lorsqu'une transition entrant dans cet état est validée. Pour valider une transition 3 conditions doivent être réunies :

- l'état de départ de la transition doit être actif ;
- la condition sur la transition doit être satisfaite ;
- enfin, pour éviter les situations d'indéterminisme pour laquelle plusieur transitions respectent les 2 conditions précédentes, une priorité est associée à chacune des transitions sortant d'un état. Ainsi pour être validée une transition doit, parmis les transitions valides, être celle qui à la plus haute priorité.

En n'activant que l'évaluation des transitions sortant de l'état courant, notre transformation permet de ne sélectionner à l'exécution que l'évaluation des transitions qui respectent la première condition. Le calcul d'une transition doit produire un signal dont la valeur est l'état d'arrivée de la transition, lorsque la condition associée à la transition est satisfaite. Lorsque la condition sur la transition n'est pas satisfaite, la valeur de ce signal n'est pas définie (signal non présent). Cette fonction peut simplement être réalisée par une opération dont la sortie est constante et dont l'exécution n'est activée que lorsque le calcul de la condition est vrai.

Une opération de type `ndefault` dont les entrées sont reliées à la sortie de chacune des fonctions de calcul de transitions permet de ne sélectionner que la valeur de l'état d'arrivée de la transition la plus prioritaire. C'est cette valeur d'état d'arrivée qui définie la valeur du prochain état.

6.3.3.3 Calcul des sorties

Dans un automate de Moore, les sorties sont associées aux états. Les sorties de l'automate sont des signaux booléens qui doivent conditionner l'exécution de certaines parties du graphe d'algorithme, un signal de sortie doit donc toujours être défini (horloge toujours présente). Ainsi, lors-

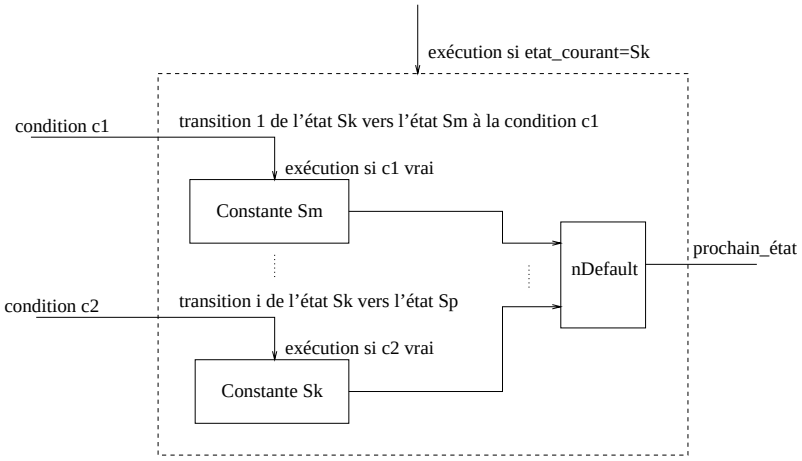


FIG. 6.13 – *Calcul du nouvel état à partir des transitions sortantes de l'état courant*

qu'un état auquel est associé une sortie est actif, le signal booléen de sortie doit prendre la valeur *vrai*. Si aucun des états auquel est associée une sortie n'est actif alors la sortie doit prendre la valeur *faux*. Les sorties doivent toutes être calculées à chaque itération du graphe.

6.3.3.4 Implantation de la transformation dans SynDEx v5

La transformation présentée ici à été implantée dans la dernière version de SynDEx disponible à l'époque où ont été réalisés ces travaux, c'est à dire la version 5 écrite en C++ et TCL-TK [37]. L'idéal pour implanter cette transformation aurait été de fournir à l'utilisateur la possibilité d'insérer dans un graphe flot de donnée une macro-opération spécifique de calcul d'automate, un double-clic sur cette opération ouvrant alors une interface permettant de spécifier un automate et de le transformer en un graphe flot de donnée encapsulé dans cette macro-opération. À l'époque à laquelle ont été réalisés ces travaux, l'interface graphique de SynDEx ne supportait pas encore la spécification hiérarchique qui était en cours de développement. Nous avons donc réalisé, sur la base de l'interface graphique existante, une interface graphique permettant de spécifier un automate et de générer automatiquement un fichier au format SynDEx décrivant le graphe flot de données de calcul de l'automate. Celui-ci peut ainsi être "chargé" dans SynDEx et inséré dans une application par un simple copier-coller.

D'autre travaux présentés dans [90] ont été intégrés à la même époque dans cette version de SynDEx. Ces travaux ont consistés à ajouter aux opérations du graphe d'algorithme un port d'entrée de conditionnement de l'exécution. Ce port d'entrée auquel est associé une polarité permet de définir sur la valeur *vrai* (polarité positive) ou *faux* (polarité négative) du signal booléen auquel elle est reliée, les instants d'exécution de l'opération. Il est ainsi possible de spécifier l'exécution exclusive de deux opérations dont les ports d'activation sont reliés au même signal booléen mais avec des polarités différentes. Une analyse des polarités sur ces ports de conditionnement permet de construire l'arborescence de conditionnement et ainsi de chercher un ordonnancement permettant de réduire le temps d'exécution de la branche la plus longue. C'est sur cette version que nous avons basé notre transformation.

L'activation d'une opération ne peut être définie que sur un signal booléen, il faut donc générer

pour chaque état un signal booléen permettant de conditionner le calcul des transitions associées à un état. Chacun de ces n signaux booléens doit être calculé par une fonction qui produit une valeur `vrai` lorsque l'état courant est l'état que représente le signal booléen et une valeur `faux` sinon. La figure 6.14 présente la structure du graphe générant à partir de la mémoire d'état ces n signaux booléens. La fonction qui permet de calculer la valeur d'un signal booléen qui représente un état est une fonction de comparaison. Elle est dupliquée pour chacun des n signaux. La première entrée de chacune de ces fonctions est reliée à la sortie de la mémoire d'état. La seconde entrée de chacune de ces fonctions est reliée à une opération générant un signal constant qui a la valeur de l'état représenté par le signal booléen.

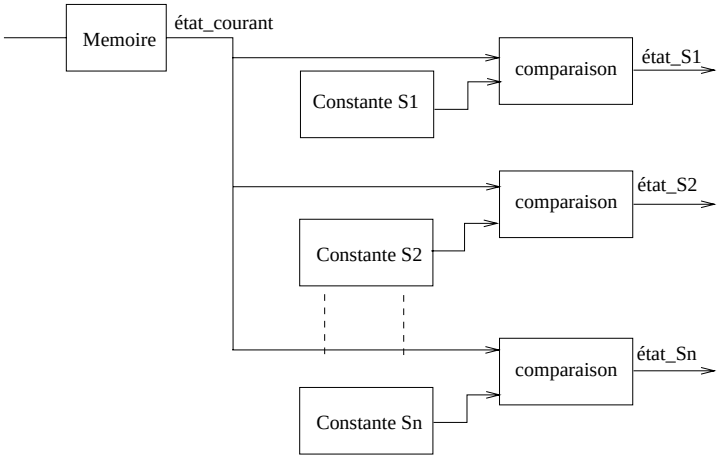


FIG. 6.14 – *Calcul des n booléens d'état un automate à n états*

Chaque booléen d'état doit conditionner l'exécution des calculs des transitions de sortie de l'état qu'il représente. Puisqu'il n'est pas possible d'encapsuler dans une macro-opération ces calculs, il est nécessaire de conditionner par le booléen d'état toutes les opérations qui composent le calcul. Dans le calcul d'une transition, l'opération qui fournit la valeur de l'état d'arrivée de la transition est déjà conditionnée par la condition de franchissement de la transition. Il faut donc un ET logique entre la condition de franchissement de la transition et le booléen d'activation du calcul.

Pour chaque sortie, il faut générer un OU logique sur les signaux booléens des états auxquels la sortie est associée.

Il peut paraître surprenant de coder l'état par un entier et de recréer ensuite n signaux booléen représentant chacun un état du système plutôt que de coder directement l'état par les n signaux booléen. En fait le codage de l'état sur un entier permet de simplifier le calcul du prochain état, il évite d'utiliser n mémoires pour mémoriser l'état et d'avoir à calculer à chaque itérations les n valeurs à stocker dans ces mémoires.

6.3.3.5 Exemple

La figure 6.15 présente un automate de Moore définissant, par exemple, les modes de fonctionnement d'un robot mobile. Cet automate est composé de 4 états représentant l'attente de démarrage du robot (`init`), le fonctionnement en pilotage manuel (`manu`), le fonctionnement en déplacement automatique (`auto`) et enfin l'arrêt de l'application (`arrêt`) pendant lequel toutes les données collectées pendant le fonctionnement de l'application sont sauvegardées sur une mémoire de masse. Le

passage du mode de fonctionnement manuel en mode automatique ainsi que l'arrêt de l'application sont déclenchés par des événements (`mode_manu`, `mode_auto`, `stop`) générés par l'opérateur lorsqu'il appuie sur les boutons d'un panneau de commande. Les sorties de l'automate doivent conditionner les calculs de l'asservissement des moteurs (`mot`), le calcul des consignes à appliquer aux asservissements des moteurs en mode manuel (`m`) et en automatique (`a`) et la sauvegarde des données (`s`).

La transformation que nous avons présentée consiste à produire un graphe flot de données dont les entrées (`go`,`mode_manu`, `mode_auto`, `stop`) sont les conditions de changement de mode de fonctionnement et les sorties sont les sorties de l'automate (`m`,`a`, `s`,`mot`).

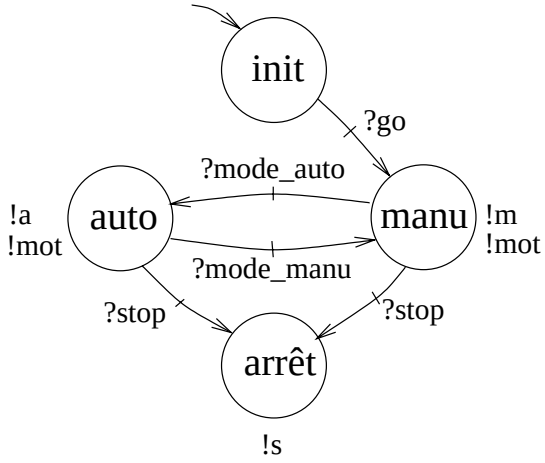


FIG. 6.15 – *Transformation d'une machine de Moore*

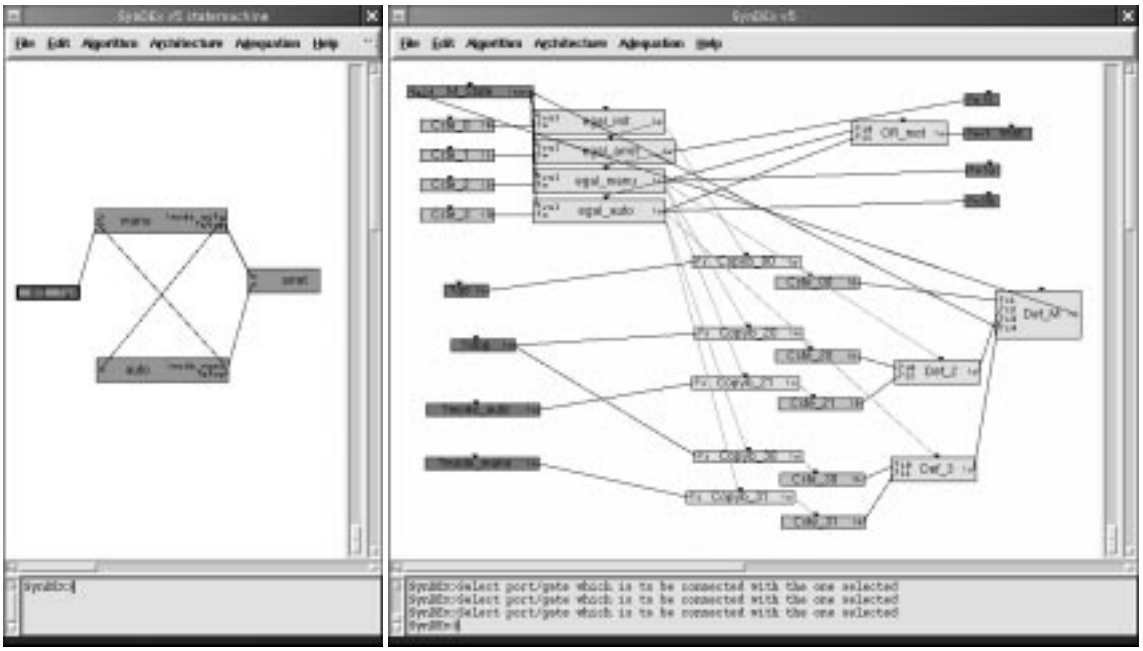
La figure 6.16(a) est une copie d'écran de l'automate spécifié dans l'interface graphique de SynDEx. Chaque état est représenté par un sommet du graphe, les conditions sur les transitions sont les noms des ports de sortie des états.

6.3.4 Implantation multi-processeurs du graphe flot de données obtenu par notre transformation

La figure 6.16(b) est une copie d'écran du graphe flot de données obtenu automatiquement par la transformation de l'automate présenté précédemment. Pour chaque valeur de l'état courant, nous avons obtenu avec SynDEx la simulation temporelle des temps d'exécution des calculs du prochain état pour une implantation sur une architecture composée de deux processeurs (`OPR1` et `OPR2`) reliés par un bus de communication de type CAN (`CAN`) (fig. 6.17).

Sur ces simulations, on peut remarquer que dans un mode de fonctionnement donné, seules les valeurs des entrées intervenant dans le calcul des transitions partant de l'état courant sont communiquées entre les processeurs. Par exemple, l'entrée `stop` qui déclenche les transitions de l'état `auto` dans l'état `arrêt` et de l'état `manu` dans l'état `arrêt` ne sont communiquées que dans les séquences d'exécution correspondant à l'état `manu` (fig. 6.17(c)) et `auto` (fig. 6.17(d)). La transformation présentée ici permet donc bien d'éviter les communications de toutes les entrées pour tous les états du système.

Par contre, dans tous les états du système on peut remarquer qu'il est nécéssaire de communiquer



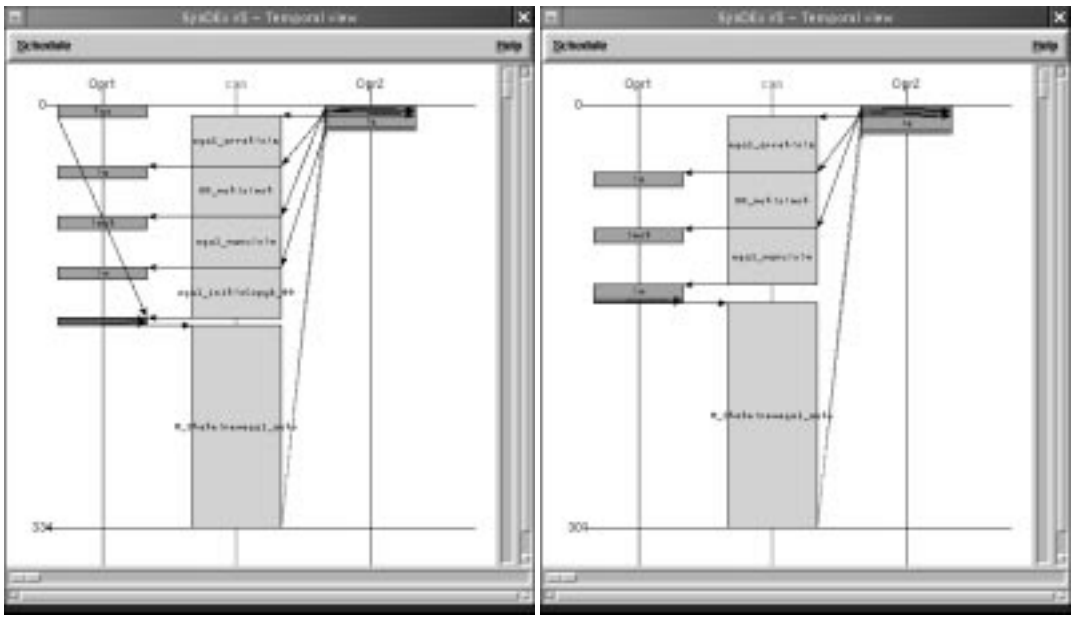
(a) Spécification de l'automate (b) Graphe flot de données généré

FIG. 6.16 – *Transformation d'un automate de Moore en graphe flot de données*

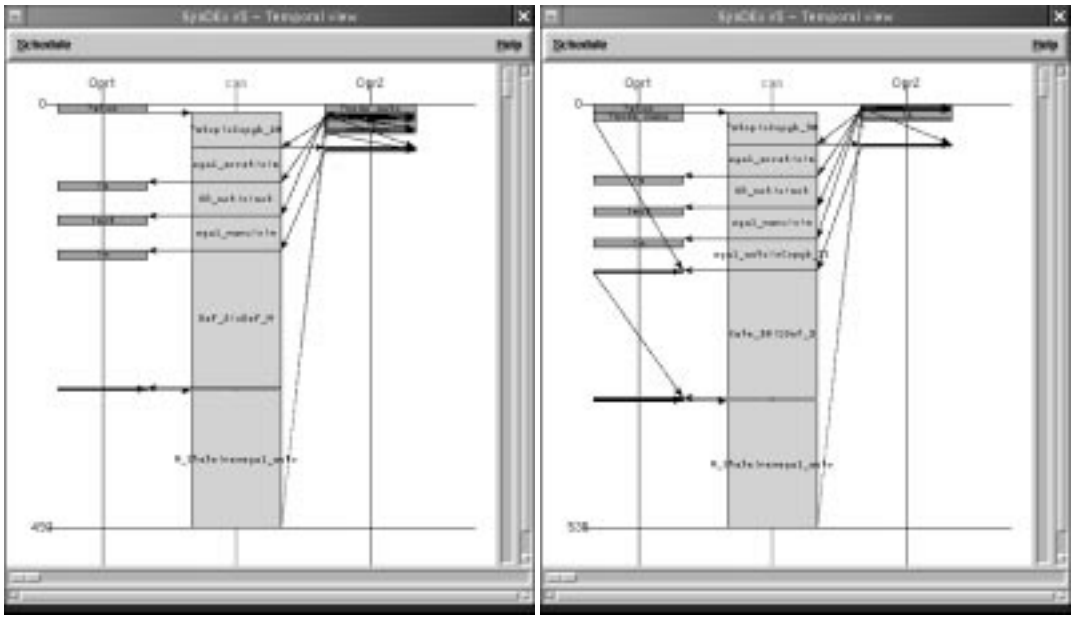
entre les processeurs les valeurs de chacun des signaux booléens représentant les états. En effet, dans SynDEx la synchronisation des séquences de calcul exécutées sur chaque processeur de l'architecture matérielle est réaliséee par les communications. Ainsi la séquence des communications sur un bus est connue de tous les processeus accédant au bus. En fonction des calculs à réaliser dans un état et des communications que ces calculs engendrent, la séquence de communication est différente. Il est donc nécessaire de distribuer sur tous les processeurs la valeur des booléens qui conditionnent l'activation des calculs, c'est à dire chaque booléen codant un état de l'automate. C'est pourquoi il faut à chaque itération du graphe transmettre les n booléens relatifs aux n états de l'automate.

Il existe une solution pour éviter ce surcoût. Plutôt que de communiquer chaque valeurs des booléens il vaut mieux communiquer à tous les processeurs la valeur de l'état codé par un `entier` fourni par l'opération `memory` et recalculer sur chaque processeurs la valeur des n booléens. On remplace ainsi n communications par une seule communication, on ajoute n comparaisons sur chaque processeurs. Le temps de calcul lié à une comparaison (moins d'une dizaine d'instructions processeur) étant bien inférieur au temps d'une communication, on réduit le temps d'exécution global de l'algorithme.

Des travaux sont en cours afin de spécifier le conditionnement d'une opération non plus par un booléen, mais directement par un entier. L'opération n'est alors exécutée que lorsque la valeur du signal correspond à la valeur spécifiée sur l'opération. Dans ce cas, la transformation est simplifiée, il n'est plus nécessaire de générer les booléens représentant chaque état. Ainsi, seule la communication de l'entier de conditionnement, c'est à dire la valeur entière de l'état est nécessaire.



(a) état courant = init (b) état courant = arrêt



(c) état courant = manu (d) état courant = auto

FIG. 6.17 – *Simulation temporelle des calculs de l'automate*

Troisième partie

Implantation, mise en œuvre des applications

Lors de la conception d'un système temps réel, la phase d'implantation est la plus délicate : elle consiste à *distribuer* et *à ordonnancer* les opérations de calcul de l'algorithme et les opérations de communication sur l'ensemble des composants qui constituent le calculateur (microprocesseurs, asics, bus de communication ...). La distribution est une *allocation spatiale* d'une opération à une ressource, alors que l'ordonnancement est une *allocation temporelle* d'une opération sur une ressource. Le but étant de distribuer et d'ordonnancer toutes les opérations pour former un programme, ensemble d'opérations qu'un calculateur doit exécuter dans un certain ordre avec certaines contraintes. Il existe, pour une architecture donnée, un grand nombre de possibilités d'ordonnancement et de distribution mais seulement un certain nombre d'entre elles permet de satisfaire à la fois aux contraintes liées à l'algorithme (dépendances de données) et aux contraintes temps réel (aspect temporel) mais aussi aux contraintes de l'architecture (aspect matériel). Cet espace de solutions valides est d'autant plus restreint que les contraintes sont fortes ; il est d'autant plus dur d'atteindre une de ces solutions. C'est le cas des systèmes temps réel embarqués qui sont des applications souvent complexes (algorithmes complexes, contraintes de temps fortes) et dont les contraintes de coût se traduisent généralement par de fortes contraintes matérielles.

CHAPITRE 7

PROBLÉMATIQUE DE L'IMPLANTATION

7.1 Contraintes d'implantation

7.1.1 Exemple didactique

Prenons l'exemple d'un système temps réel chargé de gérer deux témoins lumineux $L1$ et $L2$ et deux cellules photoélectriques $C1$ et $C2$. On peut imaginer que le but de cette application est de comptabiliser un nombre bien précis de passages d'objets devant les cellules et de les signaler par l'allumage des témoins lumineux correspondants. Bien entendu, cet exemple est fictif car trop simple pour nécessiter un système temps-réel à microprocesseur. Dans la réalité, le problème posé serait résolu simplement par l'utilisation de quelques portes logiques (compteurs ...). Cet exemple n'a donc qu'un intérêt pédagogique.

On suppose que le cahier des charges définit le comportement suivant de notre application (fig.7.1) :

- à la mise en route du système, les témoins lumineux sont éteints;
- 5 passages successifs devant la cellule $C1$ provoquent l'allumage de $L1$;
- lorsque $L1$ est allumée, un nouveau passage devant $C1$ provoque son extinction, et est comptabilisé comme premier passage;
- $L2$ et $C2$ doivent se comporter de la même manière que $L1$ et $C1$, mais seulement pour 2 passages successifs devant $C2$.

Le système temps réel que l'on considère ici est donc un calculateur auquel est relié deux capteurs $C1$ et $C2$. Les actionneurs seront constitués de deux relais $R1$ et $R2$ chargés de contrôler

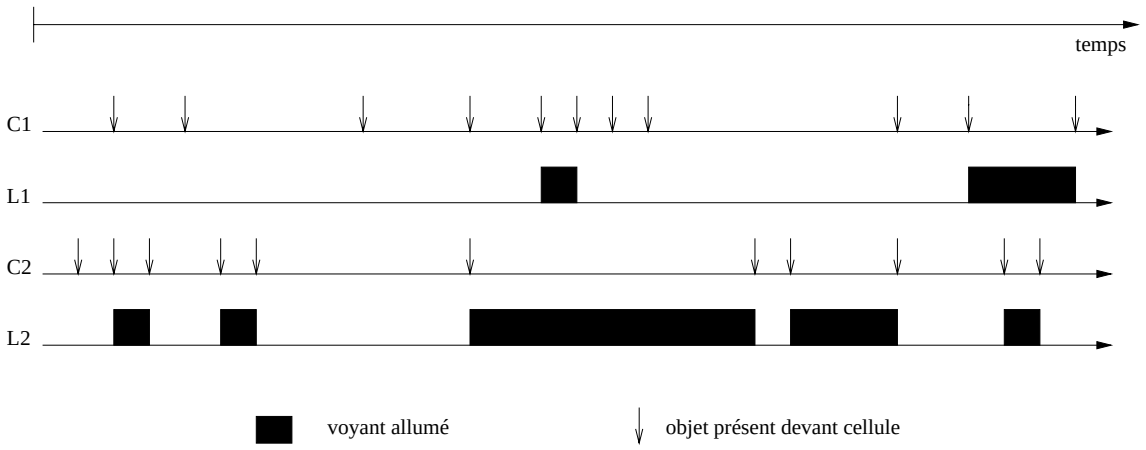


FIG. 7.1 – *Comportement du compteur d'objets*

respectivement les témoins lumineux $L1$ et $L2$. L'environnement du système est constitué par les objets qui passent devant les celulles ainsi que par $L1$ et $L2$.

Cet exemple simple illustre naturellement le comportement en trois étapes d'un système réactif. La première étape consiste à observer l'environnement, c'est-à-dire lire les signaux issus des capteurs. Ces données constituent en quelque sorte une "photographie" de l'environnement. Dans la deuxième étape on réalise des calculs sur cette "photographie" afin de déterminer, en fonction de l'état de l'environnement, la commande à appliquer aux actionneurs afin de produire une réaction qui fera évoluer l'environnement dans un état recherché. Enfin, la troisième étape consiste à produire physiquement (par le biais des actionneurs) les réactions précédemment calculées. Nous décomposerons ici l'algorithme de cette application en 6 opérations : $Lit1$ (Lecture état de $C1$), $Lit2$ (Lecture état de $C2$), $Etat1$ (calcul du nouvel état de $L1$), $Etat2$ (calcul du nouvel état de $L2$), $Cmd1$ (commande de $R1$), $Cmd2$ (commande de $R2$).

7.1.2 Contraintes d'ordonnancement

7.1.2.1 Cyclicité

Une implantation consistant à exécuter simplement dans un certain ordre les 6 opérations que l'on a définies ne permettrait pas de conférer à notre application le comportement souhaité car le programme qui matérialise cette implantation doit "perpétuellement" faire évoluer l'état des voyants en fonction de l'évolution dans le temps des signaux issus des cellules. On met ici en évidence une des propriétés importantes que doit posséder un système réactif : les intéractions entre l'environnement et le système temps réel ne doivent pas être ponctuelles, elles doivent perpétuellement se reproduire dans le temps tout au long de la vie de l'application.

Cette propriété impose un comportement itératif au programme, c'est-à-dire une reproduction cyclique de la séquence lecture-calcul-réaction (fig.7.1.2.1).

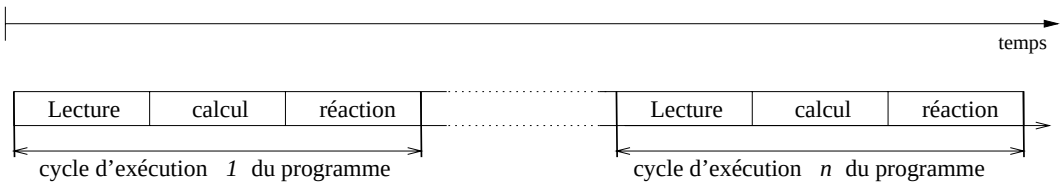


FIG. 7.2 – *cyclicité d'un programme temps réel*

7.1.2.2 Dépendances de données

Il existe !6 soit (720) possibilités d'ordonnancer les 6 opérations de notre exemple ($Lit1$, $Etat1$, $Cmd1$, $Lit2$, $Etat2$ et $Cmd2$) pour former une séquence exécutée cycliquement par un microprocesseur. Cependant, toutes ces solutions n'aboutissent pas forcement à un comportement correct de l'application. Par exemple, l'opération $Cmd1$ ne pourra être réalisée que si le nouvel état du voyant a été calculé, c'est-à-dire après l'exécution de $Etat1$. De même $Etat1$, nécessite la connaissance de l'état du capteur $C1$ qui lui est fourni par $Lit1$. Ces contraintes d'ordonnancement sont appelées *dépendances de données orientées producteur-consommateur* entre les opérations. De la même manière, sur notre exemple, les dépendances de données citées précédemment sont aussi valables pour les opérations $Lit2$, $Etat2$ et $Cmd2$. Dans la partie précédente de ce document, nous avons montré qu'un graphe flot de données permettait de mettre en évidence les dépendances de données (arcs du graphe) entre les opérations (sommets du graphe). La figure 7.3 représente le graphe flot de données de l'algorithme de notre application exemple.

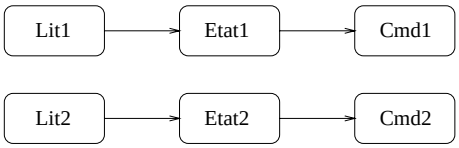


FIG. 7.3 – *Diagramme flot de données du compteur d'objets*

Pour déterminer un ordonnancement valide il faut éliminer toutes les solutions pour lesquelles les dépendances de données ne sont pas respectées, il reste alors 20 possibilités d'ordonnancer ces 6 opérations sur un microprocesseur. (fig.7.4).

7.1.2.3 Contraintes temporelles : latence, cadence

Choisir un ordonnancement satisfaisant les dépendances de données entre les opérations est une condition nécessaire mais non suffisante pour garantir le bon fonctionnement d'une application temps réel (cf. §2.1.3). Notre exemple le montre : pour que le comportement de notre système satisfasse son cahier des charges, il est nécessaire, d'une part, que tout passage d'un objet devant une des deux cellules soit bien comptabilisé. En d'autres termes, il ne faut pas perdre de changements d'état de l'environnement lié à la transition "non-présence d'objet"- "présence d'objet" et d'autre part, les relais devront pouvoir être commandés avant l'arrivée éventuelle d'un nouvel événement "présence objet" car un délai trop long conduirait à un comportement erroné du système. On illustre ici une propriété essentielle qui caractérise les programmes temps réel : les résultats des calculs ne sont valables que s'ils sont produits dans des laps de temps définis par la nature même de l'application,

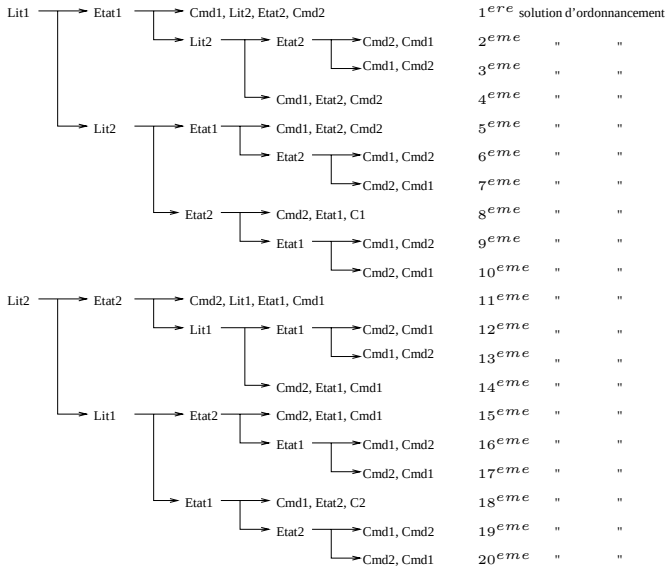


FIG. 7.4 – *Possibilités d'ordonnancement satisfaisant les dépendances de données*

les contraintes temps réel.

Sur notre système, on peut définir deux contraintes de cadence correspondant aux fréquences d'acquisition de l'état des cellules. Pour être certain de comptabiliser toutes les occurences d'un objet, en considérant que la cellule permet de mémoriser l'arrivée d'un objet, les périodes d'acquisition doivent être choisies de manière à être inférieures à l'intervalle de temps minimal qui sépare l'occurence de deux objets devant les cellules. On peut aussi définir deux contraintes de latence correspondant au temps écoulé entre la lecture de changement d'état d'une cellule (arrivée d'un objet) et la commande du voyant correspondant. Ces contraintes imposent un retard maximum entre l'instant où un objet arrive devant la cellule et l'instant où on commande le voyant.

Le chronogramme de la figure 7.5 met en évidence une séquence d'activation erronée pour le voyant $L2$ lorsque la contrainte de cadence n'est pas satisfaite. Pour une même séquence temporelle d'arrivée d'objets devant la cellule $C2$, on montre (partie haute de la figure) la séquence d'activation et de désactivation du voyant $L2$ pour une cadence d'exécution trop faible du programme. Pour une cadence d'exécution adéquate (partie basse de la figure) la séquence d'activation et de désactivation du voyant est correcte.

Les contraintes de cadence et de latence influencent directement le choix de l'ordonnancement des opérations car deux ordonnancements différents confèrent au programme des propriétés temporelles différentes. La figure 7.6 montre deux ordonnancements possibles pour les six opérations de notre graphe. Ces deux ordonnancements sont équivalents en termes de durée totale d'exécution, ils sont tous les deux logiquement corrects (dépendances de données respectées dans les deux cas), pourtant la deuxième solution est la meilleure selon un critère d'optimisation des latences.

7.1.3 Contraintes de distribution

Un microprocesseur est une machine capable d'exécuter séquentiellement des instructions. L'implantation d'un algorithme sur un calculateur monoprocesseur consiste donc à trouver un ordonnan-

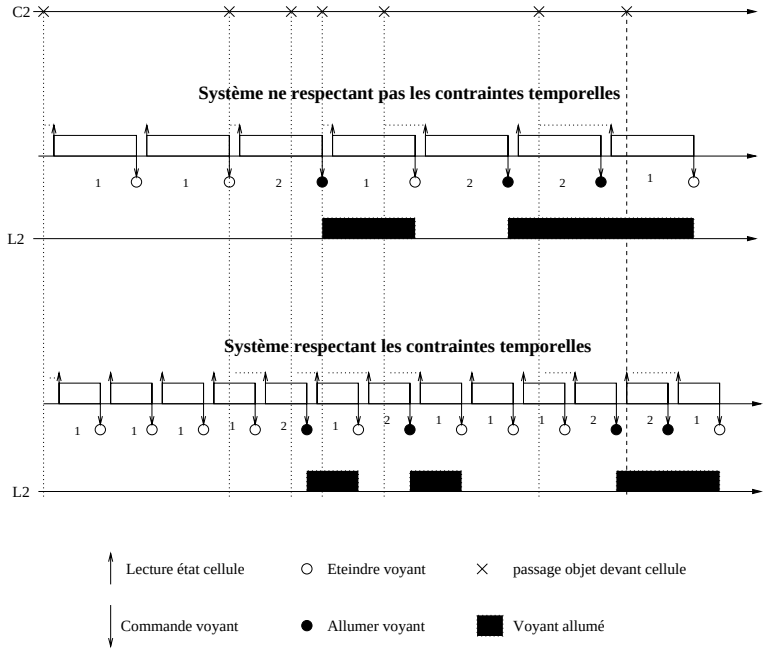


FIG. 7.5 – *Respect des contraintes temps réel*

cement qui respecte les contraintes d'algorithme et les contraintes temporelles de l'application. L'implantation sur un calculateur multiprocesseur est beaucoup plus complexe car elle ne consiste plus seulement à ordonnancer les opérations, il faut aussi les distribuer sur les différents processeurs de l'application. En fait, il faut non seulement faire un choix sur le placement temporel des opérations mais il faut aussi faire un choix sur le placement physique de celles-ci.

En théorie, il existe n^m possibilités de distribuer m opérations sur n processeurs. En pratique, toutes ces possibilités ne sont pas valides. En effet, certaines opérations accèdent à des ressources matérielles, il faut donc que ces opérations soient exécutées sur le microprocesseur qui gère ces ressources. Supposons que l'architecture matérielle de notre exemple soit celle présentée figure 7.7, dans ce cas l'opération $Lit1$ doit être exécutée sur le processeur auquel est attaché la cellule $C1$, c'est-à-dire le processeur 1. Pour la même raison, $L2$ $Cmd1$ et $Cmd2$ doivent être exécutées sur le processeur 2.

En tenant compte de ces contraintes, il reste quatre possibilités de distribution pour les opérations $Etat1$ et $Etat2$. Ce choix de distribution est très important : il a une influence directe sur les propriétés temporelles de l'implantation car, d'une part, une opération n'a pas la même durée d'exécution sur des processeurs de types différents et, d'autre part, lorsqu'il existe une dépendance de données entre deux opérations placées sur des processeur différents, il faut ajouter une opération de communication entre les deux processeurs. Toutes les opérations de communication devront être ordonnancées sur le média de communication. Les propriétés temporelles de l'implantation dépendront donc aussi des caractéristiques des médias de communication.

Dans un système multiprocesseur, afin de réduire les temps de latence des calculs, on cherchera à exploiter le parallélisme de l'algorithme. Le nombre de solutions d'ordonnancement des opérations est donc plus élevé que pour un calculateur monoprocesseur puisqu'on ne se limite plus à choisir un ordre séquentiel des opérations et car il faut aussi tenir compte d'opérations de communication.

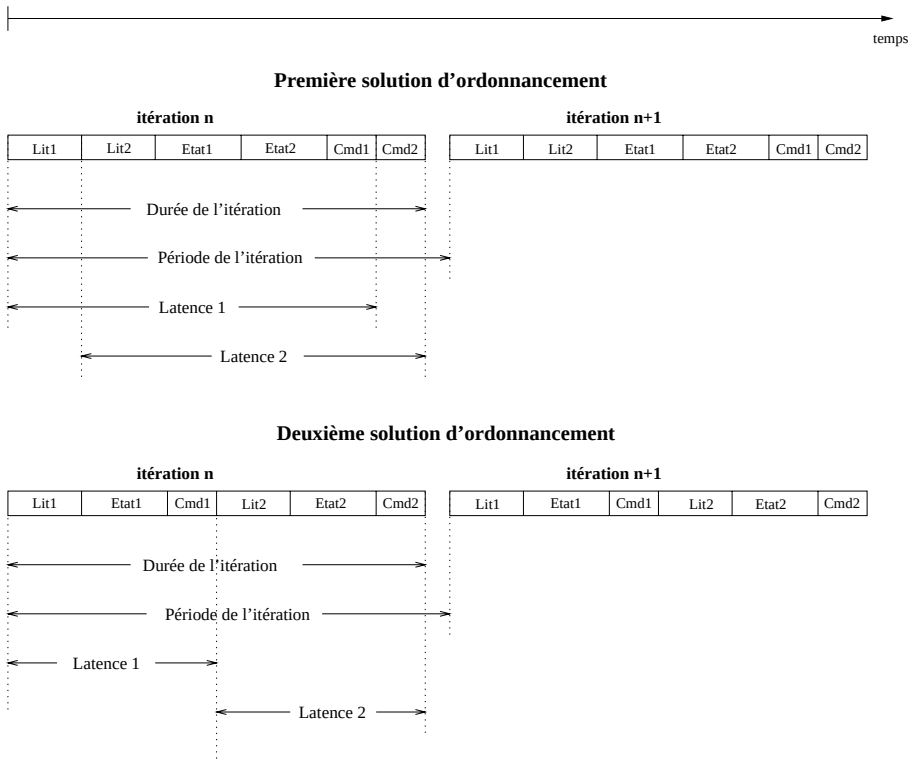


FIG. 7.6 – *Dépendance Latence - ordonnancement*

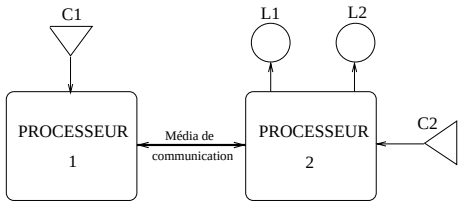


FIG. 7.7 – *Architecture matérielle du compteur d'objet*

7.2 Stratégies d'implantation

Nous l'avons vu, l'implantation d'une spécification temps réel sur une architecture distribuée consiste à réaliser un choix de distribution et d'ordonnancement des opérations qui, d'une part, permet de satisfaire les contraintes temps réel et d'autre part utilise au mieux les ressources matérielles disponibles. Nous donnons ici un aperçu de différentes stratégies permettant de mener à bien ces choix[40][62].

7.2.1 Stratégies d'ordonnancement

7.2.1.1 Modèles de tâche

La majorité des stratégies d'ordonnancement fait appel à la notion de *tâche* qui peut être définie comme une séquence ordonnée indivisible d'instructions à laquelle on associe des propriétés temporelles devant être respectées à l'exécution. On trouve dans la littérature les définitions de plusieurs modèles de tâche [35].

Tâche périodique : Une tâche périodique, comme son nom l'indique, doit être exécutée à des intervalles de temps fixés (ou périodes). Dans sa version la plus générale, une tâche périodique est définie par le quadruplet $t_i(R, C, D, T)$ où R est la date de premier réveil de la tâche, C son temps d'exécution maximal, D son échéance relative, c'est-à-dire le temps maximal dont on dispose pour exécuter la tâche, et T sa période d'exécution. La période d'exécution de la tâche correspondra à la contrainte de cadence qui est imposée à la tâche.

Tâche apériodique, sporadique : Les tâches apériodiques et sporadiques sont des tâches exécutées à des instants à priori non connus. L'exécution de ces tâches est généralement déclenchée par un événement provoqué par le processus. La période d'exécution n'intervient donc pas dans le modèle de ces tâches. On distingue une tâche sporadique d'une tâche apériodique par la connaissance d'un intervalle de temps minimum entre deux exécutions.

Remarque : Les différentes définitions d'une tâche ne font pas apparaître la notion de dépendance de données. Ainsi les tâches sont souvent considérées commme indépendantes. Dans ce document, nous utilisons le terme "opération" lorsque nous voulons désigner des tâches ayant des dépendances de données. C'est ce terme qui est utilisé dans la méthodologie AAA et le logiciel SynDEx qui la supporte.

7.2.1.2 Ordonnancement préemptif, non-préemptif

Lorsque l'exécution d'une tâche peut être interrompue par l'exécution d'une autre tâche, l'ordonnancement est dit *préemptif*. Il est dit *non-préemptif* lorsqu'aucune tâche ne peut interrompre l'exécution d'une autre tâche. Un ordonnancement préemptif apporte un surcoût sur le temps de calcul. En effet, au moment de la préemption, le contexte de la tâche préemptée doit être sauvegardé. Ce contexte doit être restauré lorsque la préemption s'achève, c'est-à-dire au moment où l'exécution de la tâche préemptée doit reprendre. Par contre, un ordonnancement préemptif permet dans certains cas de garantir des contraintes temporelles impossibles à garantir avec un ordonnancement non-préemptif. On peut distinguer deux types de préemption, celle déclenchée par des interruptions matérielles et celle déclenchée par les algorithmes d'ordonnancement.

7.2.1.3 Ordonnancement en ligne, hors ligne

L'ordonnancement peut être calculé lors de la conception du logiciel, dans ce cas, l'ordonnancement est dit *hors ligne*. Il consiste à analyser l'algorithme et ses contraintes temporelles afin de

définir la séquence d'exécution des tâches qui permet de respecter les contraintes d'algorithme, temporelles et matérielles (cf. §7.1). Dans ce cas, l'exécutif qui gère l'application est simple car le code à exécuter est composé d'une séquence d'opérations insérée dans une boucle assurant la réitération de l'algorithme. Cette approche de l'ordonnancement nécessite une spécification rigoureuse des algorithmes pour pouvoir être réalisée automatiquement par des outils logiciels. C'est généralement celle qui est utilisée quand l'algorithme est spécifié avec les langages synchrones car la réalisation d'une séquence d'exécution des opérations à l'avantage de permettre facilement de conserver et de s'assurer que l'ordre des événements défini par la spécification est conservée à l'exécution et donc que les propriétés mises en évidence par la vérification sont conservées par l'implantation.

Lorsque l'ordonnancement est calculé pendant l'exécution, on parle d'ordonnancement *en ligne*. Dans ce type d'approche, l'exécutif est construit autour d'un ordonnanceur chargé de gérer l'activation, la mise en attente et l'arrêt des *tâches* [29][77][78]. Un ordonnancement en ligne est plus coûteux à l'exécution qu'un ordonnancement hors ligne car en plus des calculs liés aux algorithmes de l'application, le calculateur doit aussi réaliser les calculs liés à l'ordonnanceur. De plus, un ordonnancement hors ligne est prédictible car la séquence de calcul est connue avant l'exécution alors que dans un ordonnancement en ligne l'ordre d'exécution est déterminé par l'ordonnanceur, il n'est donc pas toujours possible de connaître à l'avance l'ordre précis d'exécution des tâches. Un ordonnancement en ligne sera par contre plus flexible qu'un ordonnancement hors ligne, en ce sens qu'il pourra plus facilement prendre en compte les événements dont l'occurence est imprévisible (tâches apériodiques ou sporadiques).

7.2.1.4 Ordonnanceur à priorités statiques, à priorités dynamiques

L'ordonnanceur est la fonction de l'exécutif chargée de choisir l'ordre d'exécution des tâches dans un ordonnancement en ligne. Différentes hypothèses simplificatrices sont généralement faites :

- les propriétés temporelles d'une tâche sont constantes tout au long de la durée de vie de l'application ;
- les tâches sont indépendantes, ce qui revient à ignorer les dépendances de données. Le découpage de l'application en un ensemble de tâches doit donc être judicieux.

Le choix de l'ordonnancement est basé sur une priorité associée à chacune des tâches de l'application. A des intervalles de temps réguliers, l'ordonnanceur détermine la tâche à exécuter parmi une liste. Ce choix est réalisé en comparant la priorité d'exécution de la tâche en cours à la priorité de chacune des tâches de la liste. Généralement les priorités des tâches sont déterminées à la compilation et sont invariantes pendant l'exécution de l'application, on parle alors d'ordonnancement *à priorités statiques* (par abus de langage ce type d'ordonnancement est parfois appelé "ordonnancement statique"). Dans certains exécutifs les priorités peuvent varier pendant l'exécution, dans ce cas l'ordonnancement est dit à priorités dynamiques. L'assignation des priorités aux tâches est parfois empirique, il repose sur le savoir-faire des programmeurs de l'application. Généralement il repose sur des règles définies par la *politique d'ordonnancement* [62][35].

Politique d'ordonnancement Rate Monotonic : Dans cette approche, on considère que l'échéance R de la tâche est égale à sa période T et que la priorité associée à l'exécution de la tâche est inversement proportionnelle à sa période d'exécution T. Il a été démontré qu'avec ce type d'ordonnance-

ment, pour qu'une implantation soit acceptable (les échéances de toutes les tâches seront satisfaites) il suffit que l'ensemble des tâches respecte la condition suivante [53]:

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

C'est la politique d'ordonnancement la plus utilisée car c'est un ordonnancement à priorité statique, simple à mettre en œuvre.

Politique d'ordonnancement Earliest Deadline First : La priorité est ici déterminée en fonction de l'échéance relative R de la tâche [54][84][70]. C'est un ordonnancement à priorités dynamiques car la priorité de la tâche est constamment réévaluée au cours du temps. Le critère d'acceptabilité est le suivant :

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \text{ et } \sum_{i=1}^{n} \frac{C_i}{R_i} \leq 1$$

Cette politique d'ordonnancement est plus dificile à mettre en œuvre que celle du Rate Monotonic, car il est parfois difficile de fixer les échéances des tâches. Par contre, grâce à l'exploitation de l'échéance, paramètre commun à tous les types de tâches, la prise en compte des tâches sporadiques et apériodiques est facilitée [26].

Gestion des ressources Dans le cadre d'un ordonnancement préemptif, il est nécessaire de garantir qu'une ressource n'est accédée que par une seule tâche à la fois. L'accès en exclusion mutuelle à une ressource peut engendrer un phénomène d'*inversion de priorité* au cours duquel l'exécution d'une tâche haute priorité voulant accéder à une ressource est retardée par l'exécution de tâches de plus basse priorité, dont l'une est rendue non-préemptible pendant l'accès à cette même ressource [79]. Il existe des protocoles permettant d'éviter ce problème. Le plus utilisé est le *protocole de la priorité plafonnée* qui consiste à assigner temporairement une priorité maximale à la tâche accédant à la ressource.

7.2.2 Stratégies de distribution

Il existe différentes façons d'utiliser les ressources d'une architecture multiprocesseur. De celles-ci dépend la distribution des opérations. Dans les systèmes temps réel on pourra distinguer trois manières différentes de considérer l'architecture.

7.2.2.1 Architecture distribuée faiblement synchronisée

Dans ce type d'architecture, chaque processeur est dédié à la commande d'une partie bien définie du processus. Tout les algorithmes de contrôle et de commande relatifs à une partie sont regroupés dans un même processeur. Cette approche est très utilisée dans les applications automobiles. Ici le choix de la distribution est fonctionnel, il ne permet pas d'utiliser au mieux les ressources des processeurs : certains processeurs peuvent être très chargés, d'autres très peu. Dans cette approche, le couplage entre les processeurs est faible, l'implantation revient quasimment à ordonnancer n systèmes temps réel monoprocesseurs indépendants. Il n'y a pas d'horloges globale, sur chaque processeur les communications sont traîtées comme des événements externes[46].

7.2.2.2 Architecture distribuée fortement synchronisée

Dans ce type d'approche, les calculs réalisés sur chacun des processeurs sont fortement liés. Les communications inter-processeurs permettent de créer une horloge globale permettant de synchroniser l'exécution des calculs de manière à garantir leur cohérence. Il existe principalement deux manières de concevoir ce type de système :

Architecture de type maître - esclave Dans certaines applications, un processeur particulier (maître) est désigné pour réaliser les calculs "de plus haut niveau", c'est lui qui est notamment chargé d'exécuter les algorithmes de contrôle. Les autres processeurs (esclaves) sont considérés comme des capteurs et actionneurs intelligents controlés par le processeur maître. Ici, toutes les opérations sont distribuées à la main sur chacun des processeurs. Cette approche, comme la précédente, à moins qu'elle utilise des techniques de migration de code, a le désavantage de ne pas permettre d'utiliser au mieux les ressources de tous les processeurs.

Architecture de type distribuée Ici tous les calculs de contrôle et de commande sont distribuées sur les processeurs. Le découpage de l'algorithme (parllélisme potentiel) est effectué de manière à tirer au mieux partie du parallélisme disponible de l'architecture. Chaque processeur exécute une partie des calculs, la cohérence globale des calculs est assurée par les communications. Il a été montré que le problème de l'implantation optimale d'un algorithme sur une architecture distribuée est un problème NP-complet. Trouver la solution qui permet d'utiliser au mieux les ressources est, dans un temps de calcul raisonnable impossible, on se contente donc généralement d'une solution sous-optimale obtenue à l'aide d'heuristiques. C'est dans ce type d'approche que se situe la méthodologie AAA.

CHAPITRE 8

OUTILS D'IMPLANTATION

On peut recenser principalement trois approches différentes pour réaliser l'implantation dans un système temps réel. La première approche consiste à réaliser à la main un exécutif sur mesure adapté à l'application. La deuxième approche consiste à utiliser un exécutif commercial (OS) temps réel. Enfin la troisième approche consiste à générer automatiquement à partir de la spécification un exécutif taillé sur mesure pour l'application.

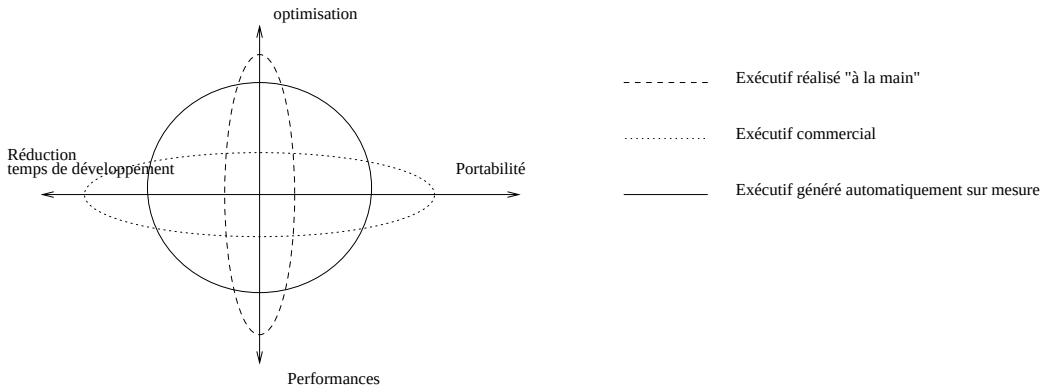


FIG. 8.1 – *Comparaison des méthodes d'implantation*

La figure 8.1 montre les points forts et faibles de ces méthodes. Celle consistant à réaliser à la main l'exécutif permet généralement de réaliser des systèmes plus optimisés, souvent plus performants, au prix de temps de développement plus long. De plus, il n'est pas possible de rentabiliser ces temps de développement en réutilisant le code écrit dans d'autres applications. Cette solution est souvent retenue dans le cadre d'applications pour lesquelles il n'existe pas d'exécutifs capables de prendre en compte l'architecture matérielle ou dans le cas où l'architecture matérielle est très fortement soumise à des contraintes de coût et nécessite une très forte optimisation des programmes (limitation mémoire, rapidité d'exécution ...). L'utilisation d'un OS commercial offre principalement l'avantage de réduire les temps de développement puisqu'elle évite les phases de programmation et

de déboguage de l'exécutif et permet plus facilement de porter les programmes sur d'autres architectures dans la mesure où l'OS utilisé est disponible sur ces différentes architectures. La troisième approche est la plus récente elle cherche à cumuler les avantages des deux premières. L'exécutif est généré automatiquement et n'inclut que les fonctionnalités nécessaires à l'application, ainsi les temps de développement sont réduits, et les performances sont proches d'une implantation à la main.

Une récente étude [86] a montrée que 64% des implantations sont réalisées à l'aide d'un OS commercial. La génération automatique d'exécutif est très prometteuse, elle est de plus en plus utilisée. Nous présentons ici ces deux approches.

8.1 Implantation avec un OS temps réel

8.1.1 Normalisation d'OS

Afin d'améliorer la portabilité et réduire les coûts des implantations, différentes tentatives ont été mises en œuvre pour normaliser les OS temps réel.

8.1.1.1 Norme SCEPTRE

Le projet SCEPTRE [1] auquel a participé un groupe d'industriels et d'universitaires a délivré en 1980 un rapport se définissant comme "un guide standard pour les ingénieurs en temps réel"[76]. Son but est de réaliser "une synthèse de l'expérience française en matière de construction des Exécutifs Temps Réel. Il cherche à montrer comment structurer les mécanismes de coopération entre activités parallèles dans un Exécutif Temps Réel, à fixer les concepts et un vocabulaire, et à spécifier un ensemble d'opérations appelé noyau SCEPTRE permettant de concevoir et construire un Exécutif Temps Réel portable et performant". Selon SCEPTRE, un exécutif temps réel doit cacher les propriétés spécifiques des machines utilisées pour n'en laisser voir que les propriétés logiques afin d'améliorer la compréhension des programmeurs, de permettre l'emploi de langages évolués et de faciliter l'adaptabilité et la portabilité des applications. Il doit fournir une interface unique entre les différentes parties d'une application écrite dans des langages différents, assurer la gestion des entrées/sorties et gérer les ressources (mémoires, processeurs, bus de communication ...). Pour pallier aux inconvénients de l'utilisation d'un exécutif, SCEPTRE préconise une approche intermédiaire consistant à construire un exécutif à la carte à partir d'un ensemble de modules que l'on assemble en fonction des besoins. SCEPTRE définit l'ensemble des services que doit fournir un exécutif :

- la communication,
- la synchronisation,
- la gestion et l'ordonnancement des tâches,
- la gestion de la mémoire,
- la gestion des interruptions et les E/S physiques,

1. Standardisation du Cœur des Exécutifs des Produits Temps Réel Européens

- les E/S logiques et la gestion des périphériques,
- la gestion des fichiers et des supports,
- le gestion des programmes,
- la gestion des travaux et des transactions,
- le traitement des erreurs et des exceptions,
- la gestion du temps.

SCEPTRE définit une “machine abstraite” englobant tous les mécanismes bas niveau (interruption, adressage des interfaces d’E/S...), le noyau sur lequel peut s’appuyer des fonctions de plus haut niveau écrites en langage évolué. Ainsi, dans cette approche on trouve principalement trois niveaux :

- le noyau, réalise l’interface entre le hardware et l’exécutif ;
- le niveau exécutif sur lequel s’appuie l’application ;
- le niveau application.

SCEPTRE définit et manipule quatre entités de base :

- la machine : c’est l’environnement matériel qui assure l’exécution de processus cablés, microprogrammés ou programmés qui partagent en commun une mémoire et/ou des registres. Une machine peut intégrer un ou plusieurs processeurs ;
- la tâche : c’est un agent actif responsable de l’exécution par une machine d’un programme composé à partir du répertoire d’instructions de cette machine ;
- l’ordonnanceur : module chargé de gérer un ensemble de processeurs identiques pour le compte d’une famille de tâches ;
- l’agence : fournit une collection d’opérations spécialisées aux programmes utilisateurs ;

Il décrit d’autre part les relations entre ces différentes entités.

Noyau SCEPTRE Le noyau SCEPTRE a pour but de définir un ensemble d’opérations élémentaires (ou primitives) “bas niveau”sur lesquelles doivent s’appuyer les fonctions de l’exécutif. Ce noyau doit être programmé en fonction des caractéristiques du calculateur cible pour l’utiliser au mieux, il n’est donc pas portable. Par contre, il doit permettre d’assurer une portabilité maximale aux applications qui l’utilisent.

La conception du noyau SCEPTRE répond principalement à 6 critères :

- Minimalité : nombre de mécanismes aussi restreint que possible,
- Généralité : prise en compte de configurations matérielles diverses,
- Performance : meilleur compromis généralité/performance,

- Simplicité : éviter les ambiguïtés sémantiques,
- Constructibilité : permettre la construction de mécanismes de plus haut niveau ayant des fonctions analogues mais présentant une plus grande sécurité d'emploi,
- Indépendance : les mécanismes du noyau doivent être indépendants de tout langage de programmation et de toute machine.

Le noyau se limite aux aspects communication, synchronisation, gestion et ordonnancement des tâches et gestion des interruptions et des E/S physiques. Il fournit donc des opérations élémentaires permettant de réaliser efficacement l'ordonnancement, la signalisation, l'exclusion mutuelle et la communication. A chacun de ces types de fonctions est associé un type d'objet, respectivement la tâche (pour identifier les tâches), l'événement (pour mémoriser le signal marquant qu'une condition est devenue vraie), la région (pour caractériser la possession exclusive d'un objet partagé) et la file (pour ranger les informations provenant de différentes tâches, et les prendre en compte dans leur ordre d'arrivée). Chaque noyau se base sur un ordonnanceur propre qui peut être préemptif ou non.

Opérations de gestion de tâches démarrer(tâche), arrêter(tâche), se terminer, changer-priorité(tâche, nouvelle-priorité), ETAT(tâche), PRIORITE(tâche), TACHE-COURANTE

Opérations de manipulation d'événements ATTENDRE(liste d'événements appartenant à TACHE-COURANTE), SIGNALER(événement, tâche),

8.1.1.2 OSEK/VDX

OSEK/VDX est une proposition de standardisation de système d'exploitation pour applications automobiles embarquées. Il regroupe le savoir-faire de constructeurs, d'équimentiers et d'universitaires allemands (BMW, Daimler-Benz/Mercedes-Benz, Opel Volkswagen, Bosch, Siemens et le IIIT de l'université de Karlsruhe) pour OSEK [2], et de constructeurs automobiles français (Peugeot et Renault) qui ont apporté l'approche VDX [3][3].

Les applications automobiles sont caractérisées par des contraintes temps réel rigoureuses. Les architectures utilisées sont de type fortement réparties et hétérogènes. Les calculateurs sont appelés ECU [4]. OSEK/VDX est un système d'exploitation monoprocesseur destiné à la programmation de ces ECU. Il définit un OS temps réel, ainsi que des services de communication et de gestion de tâches en réseau.

Objectifs d'OSEK Le but de la standardisation est d'assurer une portabilité du code sur différents ECU et de permettre la réutilisation de code d'une application à une autre afin de réduire les coûts de développement.

OSEK/VDX cherche à définir une architecture logicielle modulaire et ouverte ainsi que leurs interfaces associées pour relier l'ensemble des ECU d'un véhicule.

2. abbréviation allemande de "open systems and their corresponding interfaces for automotive electronics"
3. Vehicle Distributed eXecutive
4. Electronic Control Unit

Les fonctionnalités décrites s'apparentent à trois domaines distincts :

- **OSEK/VDX communication** : définit les services pour la communication entre les ECUs et à l'intérieur des ECUs ;
- **OSEK/VDX network management** : définit des services de configuration du réseau d'ECU et de monitoring ;
- **OSEK/VDX operating system** : définit les services d'un OS temps réel pour la gestion des tâches.

OSEK/VDX Operating System Le cahier des charges de l'OS a été défini de manière à prendre différentes contraintes propres aux applications embarquées et plus particulièrement automobiles[1]. Ces contraintes sont :

- OS temps réel implanté très efficacement au vue du temps d'exécution ;
- il doit pouvoir "tourner" en environnement embarqué avec une mémoire réduite (ROM, RAM) ;
- il doit être conçu de façon modulaire de manière à pouvoir être configuré et dimensionné en fonction des besoins de chaque application.

Ainsi OSEK/VDX Operating System définit les fonctionnalités et le comportement d'un OS temps réel ainsi qu'une interface standard entre les applications et les services fournis par l'OS. Cette interface est définie par des services systèmes. Elle doit être identique pour toutes les implantations de l'OS sur les différentes familles de processeurs. Les services définis dans l'OS sont des fonctions de gestion de tâches, de synchronisation entre tâches, de gestion d'interruptions et d'alarmes ainsi que des fonctions de traitement des erreurs.

OSEK/VDX OS supporte des implantations en ROM, c'est-à-dire que le code peut être exécuté à partir d'une mémoire ROM. Il est conçu pour nécessiter un minimum de ressources matérielles (RAM, ROM, temps d'exécution CPU) et peut être implanté sur des microcontrôleurs 8 bits.

Pour garantir l'utilisation d'un minimum de ressource, OSEK/VDX est conçu pour pouvoir être adapté au besoin de chaque application. Pour répondre à ce besoin la notion de *classe de conformance* (CC) a été introduite. Les classes définies sont basées les unes sur les autres avec une compatibilité ascendante. L'utilisateur peut choisir le niveau de classe qui convient à son application et ainsi adapter l'OS en fonction de ses besoins. Chaque classe a été définie en fonction des propriétés des tâches et de caractéristiques d'ordonnancement de ces tâches.

Modèle de tâche Dans OSEK/VDX-OS on peut manipuler des *tâches simples* ou des *tâches étendues*. Une tâche simple peut être dans un état *suspendue*, *prête à être exécutée* ou *active* (en cours d'exécution). Une tâche étendue est une tâche simple à laquelle un état d'*attente* a été rajouté. Dans cet état une tâche attend l'arrivée d'au moins un événement qui peut être émis par une autre tâche, un compteur ou une alarme. L'arrivée de cet événement fait passer la tâche dans l'état *prête à être exécutée*. A un instant donné, une seule tâche peut être dans l'état *active*.

Politique d'ordonnancement Le choix de l'ordonnancement des tâches est réalisé à l'exécution par l'*ordonnanceur*. Néanmoins, le développeur de l'application décide de la séquence d'exécution en configurant les priorités des tâches et en sélectionnant un mécanisme d'ordonnancement (full-, non- or mixed-preemptif) :

- ordonnancement non-préemptif : l'ordonnanceur est appelé pour choisir l'exécution d'une tâche soit après la terminaison d'une tâche, soit après la terminaison correcte d'une tâche avec une activation explicite d'une tâche successeur, soit par un appel explicite à l'ordonnanceur ou lors d'un appel explicite à un wait (attente d'un événement) ;
- ordonnancement préemptif : une tâche en cours d'exécution peut être interrompue à chacune de ces instructions par l'ordonnanceur qui peut alors faire exécuter une autre tâche. Le contexte d'une tâche interrompue est sauvegardé ce qui autorise sa réexécution à l'endroit où elle a été arrêtée ;
- ordonnancement mixte : ce mode est un compromis entre les deux précédents. Certaines tâches pourront être interrompues, d'autres non. Le choix est fait par le développeur en fixant à la compilation l'attribut préemptif/non-préemptif de chaque tâche ;

Le plus haut niveau de priorité d'exécution est réservé aux interruptions matérielles. Le niveau de priorité juste en dessous est réservé aux fonctions de l'OS. Les niveaux inférieurs sont les priorités que l'utilisateur peut assigner aux tâches.

Gestion des ressources Les ressources matérielles qui sont gérées par des services de l'OS ne peuvent être appelées que par une interface unique afin de garantir une portabilité maximum des applications.

Afin d'éviter le phénomène d'inversion de priorité bien connu dans les OS préemptifs et afin d'éviter que deux tâches accèdent en même temps à la même ressource, l'ordonnanceur utilise le *Priority Ceiling Protocol* : une tâche accédant à une ressource se voit attribuer momentanément un niveau de priorité maximum afin qu'aucune autre tâche ne puisse la préempter et accéder à la même ressource.

Classes de conformance Les classes de conformance ont été définies en fonction des propriétés des taches et de caractéristiques d'ordonnancement de ces tâches.

Il y a deux types de classes de conformance : les Basiques CC (BCC) et les Etendues (ECC). Il existe cinq niveaux de classe que l'on peut énumérer par compatibilité ascendante :

- BCC1 : à ce niveau toutes les tâches sont de type simple. Les demandes d'activation d'une tâche déjà active sont ignorées. Chaque tâche à un niveau de priorité différent. Cette classe de conformance est celle qui requiert le minimun de ressources ;
- BCC2 : identique à BCC1, mais elle autorise plus d'une tâche par niveau de priorité ;
- BCC3 : identique à BCC2, mais mémorise toutes les demandes d'activation d'une tâche lorsque celle-ci est déjà active. La tâche sera réactivée plus tard autant de fois qu'il y a eu de requêtes mémorisées ;

– ECC1 : identique à BCC3, mais ici les tâches sont de type étendu. De plus, pendant l'exécution de la tâche une seule requête d'activation est mémorisée ;

– ECC2 : identique à ECC2, mais comme pour BCC2 toutes les requêtes d'activation sont mémorisées.

Traitement des erreurs OSEK/VDX-OS intègre deux types de service de gestion d'erreur. Le premier offre la possibilité d'intégrer à l'OS des fonctions définies par l'utilisateur pour faciliter le débogage des applications. Le second est un service de report d'erreurs et de procédure de recouvrement. Toujours dans le souci de garantir une utilisation minimale des ressources, il existe deux niveaux de gestion des erreurs. La gestion avancée qui procure un grand confort de gestion d'erreurs mais qui est gourmande en ressource peut être utilisée dans les phases de test des applications ou dans les applications peu critiques. La gestion réduite moins complète libère des ressources pour les applications critiques.

OSEK/VDX Communication OSEK/VDX-COM est un standard de communications entre les ECUs du véhicule, entre les ECUs et les périphériques et entre les tâches internes aux ECUs. Ces communications inter-ECU ou intra-ECU sont réalisées par les mêmes services, la position dans le réseau de l'émetteur et du récepteur d'une communication n'a pas d'importance. OSEK/VDX-COM est construit sur un modèle composé de trois couches de communications :

– la couche data est constituée des composants qui gèrent matériellement le protocole de communication du bus, ainsi que des drivers logiciels de gestion de ces composants ;

– la couche transport gère le découpage et la reconstitution des messages longs qui ne peuvent être transportés en un seul message ;

– la couche interaction est l'interface avec l'application. Cette couche rend les communications indépendantes du bus et du protocole choisi.

Les communications sont réalisées par deux types de messages. Les messages de type "état" représentent la valeur d'une variable du système. Ces messages ne sont pas mémorisés. La valeur est écrasée par une valeur plus récente. Si la valeur n'est pas lue avant écrasement alors cette valeur n'est pas consommée. Par contre les messages de type "événement" sont mémorisés et doivent être consommés.

Les communications peuvent être point à point ou multipoints (broadcast).

OSEK/VDX Network Management Le but de OSEK/VDX-NM est de réaliser de manière cohérente le démarrage (start-up) et l'arrêt (shut-down) de tous les ECUs, la vérification et la surveillance de la configuration et de l'aide au diagnostic.

Implantation et certification d'OSEK/VDX OSEK/VDX n'est pas en lui-même un OS temps réel, c'est une proposition de standardisation d'un OS temps réel pour applications automobiles. Pour être utilisée dans une application, une implantation particulière d'OSEK/VDX doit être utilisée pour chaque type de processeurs utilisés. Différentes sociétés fournissent des implantations,

chacune adaptée à une architecture cible. Pour que la mise en réseau de différents ECUs basés sur des architectures matérielles différentes et donc des implémentations d'OSEK/VDX différentes puissent fonctionner correctement et pour garantir une portabilité maximale, il est nécessaire d'être certain que chaque implantation d'OSEK/VDX est conforme au standard. C'est pourquoi le projet européen MODISTARC a été chargé de définir des jeux de tests permettant d'aboutir à la certification des implantations. Pour être certifiée conforme à OSEK/VDX, une implantation doit, d'une part, avoir passé avec succès ces tests et, d'autre part, les performances (latence d'interruption, temps de changement de contexte ...) de cette implantation doivent être fournies par le concepteur.

Pour une meilleur portabilité, le langage OIL a été conçu pour spécifier la configuration d'une application (classe de conformances utilisées, type d'ordonnancement, configuration des tâches ... utilisés par chacun des ECUs de l'application).

8.1.2 Exécutifs commerciaux

L'implantation d'une spécification avec un exécutif commercial consiste à découper l'algorithme en tâches. La liste de tâches obtenues est gérée à l'exécution par l'ordonnanceur de l'OS. Les OS commerciaux sont généralement basés sur une politique d'ordonnancement à priorités statiques. Certains OS implantent directement une politique de type Rate Monotonic, dans ce cas à l'exécution la priorité de la tâche est déterminée par sa période d'exécution, dans d'autres cas l'utilisateur doit fixer une priorité.

L'offre commerciale des OS est très abondante. Il est difficile de comparer les caractéristiques temporelles des OS car les valeurs, si elles sont fournies par le constructeur, dépendent fortement des conditions dans lesquelles elles ont été mesurées. Le choix d'un OS est donc plus souvent dicté par la gamme de processeurs cibles, par les services offerts par celui-ci, par la taille mémoire occupée, etc.

Actuellement, une quinzaine d'OS se partagent la quasi-totalité du marché[86]. Parmi ceux-ci on pourra remarquer :

VxWORKS Commercialisé par la société WindRiver Systems, il a été conçu pour le développement en environnement Unix. Il offre la possibilité de traîter des architectures multiprocesseurs à mémoire partagé (VME bus). L'offre en terme de processeurs cibles est très importante. C'est l'OS le plus prisé actuellement, 12% des systèmes temps réel l'utilisent ;

Chorus, LynxOS, QNX Ce sont des OS répartis qui offrent des services de gestion de communications entre tâches situés sur des processeur différents. La structure de ces OS est proche de la norme POSIX (UNIX temps réel). Chorus est développé par la société Chorus Systems, LynxOs par Lynx Real-time Systems et QNX par QNX Systems ;

OSEK/VDX Le marché des OS compatibles avec la norme OSEK/VDX ne cessent de croître. Actuellement, de nombreux développeurs fournissent un OS basé sur cette norme ;

Windows NT Bien qu'il n'ait pas été conçu à l'origine pour les applications temps réel, on remarque que de nombreuses applications sont réalisées avec cet OS. La part de marché qu'il occupe

tend à s'approcher de celle de VxWorks.

8.2 Génération automatique d'exécutif

8.2.1 Langages synchrones

Les langages synchrones déjà présentés §5.1.2.1 disposent généralement d'un compilateur capable de transformer la spécification en un automate à états fini sur lequel il est possible de vérifier des propriétés. Ce compilateur définit aussi des règles permettant de traduire cet automate en un code séquentiel garantissant la conservation des propriétés montrées sur la spécification. L'ordonnancement réalisé est ainsi généralement off-line non-préemptif. Le code utilise un langage de haut niveau tel que le langage C, il est composé d'une boucle dans laquelle le code des opérations est appelé en séquence. Cette approche est hautement prédictible, elle permet de concevoir des applications sures et de réduire le cycle de développement.

8.2.2 Méthodologie AAA

La méthodologie AAA s'inscrit dans le cadre des langages synchrones. Elle vise l'optimisation de l'implantation d'un algorithme sur une architecture. Le calcul de l'implantation consiste à réduire le parallélisme potentiel de l'algorithme mis en évidence par le graphe d'algorithme au parallélisme disponible de l'architecture matérielle décrit par le graphe d'architecture. Ces transformations ont été formalisées dans [90]. Parmi toutes les distributions et ordonnancements possibles de l'algorithme sur l'architecture on cherche à l'aide d'une heuristique une solution optimale sous-optimale (car le problème est NP complet) selon un critère de minimisation de la durée d'exécution d'une itération du graphe complet de l'algorithme. La méthodologie permet donc de ne prendre en compte qu'une seule contrainte de latence. Cette latence est le chemin critique, appelé R, du graphe, il est déterminé à partir du calcul des durées d'exécution associés à chaque opérations, en tenant compte des durées de communications entre ces opérations (Cf figure 8.2). Pour cela, à chaque opération du graphe logiciel sont associées des dates qui bornent le début et la fin de l'opération, elles sont du type "début au plus tôt depuis le début " (notée $S(opration)$), "début au plus tard depuis la fin" (notée $\bar{S}(opration)$). La durée d'exécution de l'opération dépend du processeur sur lequel elle est exécutée. Avant distribution, ont défini cette durée comme étant la moyenne des durées d'exécution sur chacun des opérateurs du graphe matériel. La valeur des dates est calculée, dans un premier temps, à partir du graphe de dépendance que constitue le graphe logiciel. En effet, une opération ne peut être exécutée qu'à partir du moment où l'exécution de toutes les opérations qui doivent lui transmettre des données (prédecesseurs), est terminé. Dans un deuxième temps, les dates associées à une opération sont recalculées lorsque l'opération est placée sur un opérateur. La date de début au plus tôtlorc'est à dire après distribution sur un processeur, le calcul de la date de début au plus tôt dépend de la date de fin de la dernière opération placée sur l'opérateur, mais aussi de la date de fin de ces prédecesseurs.

Le problème d'optimisation consiste à minimiser R. L'heuristique utilisée dans SynDEx pour approcher la solution optimale de ce problème NP complet est basée sur la "flexibilité d'ordonnancement[5]"(notée $F(opration)$) : chaque opération possède une flexibilité d'ordonnancement calcu-

5. en Anglais "Schedule-flexibility"

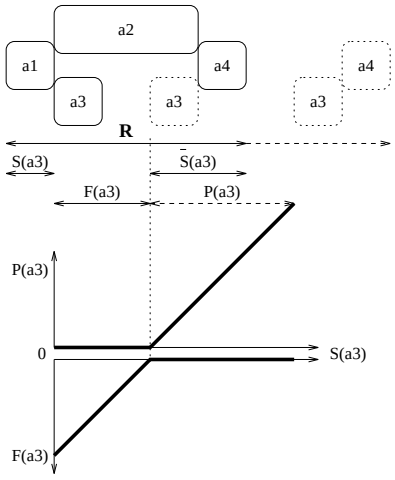


FIG. 8.2 – *Calcul de la pression d'ordonnancement*

lée à partir de R moins la différence entre la date de début au plus tôt depuis le début et celle de début au plus tard depuis la fin. C'est une heuristique "gloutonne" (par opposition au principe de "retour-arrière") : les choix effectués ne sont en effet pas remis en cause par la suite; lors de chaque itération, l'heuristique effectue la distribution d'une action sur un processeur. L'action est choisie parmi un ensemble d'actions que l'on appelle les candidats. Un candidat est une action dont tous les prédécesseurs sont distribués. La flexibilité d'ordonnancement est alors utilisée comme une fonction de coût qui permet de déterminer la meilleure alliance action/processeur. Pour SynDEx, une nouvelle heuristique basée sur la flexibilité d'ordonnancement a été développée pour tenir compte des retards engendrés par les communications. La fonction de coût a été étendue de façon à tenir compte de l'allongement éventuel du chemin critique engendré par la distribution d'une action et de ses communications. Ainsi, selon le processeur choisi, la date d'exécution de l'action peut varier, ce qui se traduit par différentes dates de début au plus tôt. Plus une action est exécutée tardivement, moins elle a de flexibilité d'ordonnancement. Cette flexibilité, qui est une fonction continue par intervalle, devient nulle quand la date d'exécution atteint ou dépasse la date de début au plus tard déterminée (ce qui induit l'allongement du chemin critique). Une nouvelle fonction a été introduite : la "pénalité d'ordonnancement[6]" (notée $P(action)$)', elle aussi continue par intervalle, elle représente l'allongement du chemin critique (elle est donc nulle tant que ce chemin n'est pas allongé). La différence entre la pénalité d'ordonnancement et la flexibilité d'ordonnancement est une nouvelle fonction, continue, appelée "pression d'ordonnancement[7]" (notée $S(action)$)). Plus la pénalité d'ordonnancement d'une action augmente, plus la pression d'ordonnancement de cette action augmente aussi, plus il est donc urgent de distribuer cette action.

L'heuristique consiste donc à effectuer, pas à pas, la distribution et l'ordonnancement : a chaque pas de l'heuristique, pour chaque action candidate, il faut rechercher le meilleur processeur, c'est à dire celui qui induit la plus faible pression d'ordonnancement. Quand la paire action/processeur est déterminée pour chacun des candidats, il faut distribuer l'action qui possède la plus élevées des pressions d'ordonnancement sur le meilleur processeur qui lui a été trouvé. Ensuite il suffit d'ajouter les éventuels nouveaux candidats successeurs de l'action distribuée, et réitérer le processus. Le

6. en Anglais "Schedule penalty"
7. en Anglais "Schedule pressure"

nombre de pas de l'heuristique est donc égal au nombre d'actions.

Lorsque toutes les opérations du graphe d'algorithme sont distribuées et ordonnancées, SynDEx affiche une prédiction temporelle de l'exécution de l'algorithme sur l'architecture matérielle. L'utilisateur peut ainsi vérifier si l'application satisfait les contraintes temps-réel. Lorsqu'il est satisfait par cette implantation celui-ci peut demander à SynDEx de générer automatiquement l'exécutif garanti sans interblocage qui supporte l'exécution temps réel de l'application sur l'architecture distribuée. Cet exécutif est décrit par un macro-code indépendant du jeu d'instruction des processeurs. Celui-ci est ensuite traduit en code compilable pour chacun des processeurs de l'architecture au moyen d'un jeu de macros extensible et facilement portable constituant "l'exécutif générique".

CHAPITRE 9

EVOLUTION DE AAA

Nous avons vu qu'un système temps réel doit implanter un ensemble de lois de commandes ayant chacune des contraintes de cadences et de latences définies en fonction des constantes de temps des sous-processus qu'elles commandent ainsi que des performances recherchées (temps de réponse, etc.). La méthodologie AAA ne permet jusqu'ici de spécifier et d'implanter qu'une seule contrainte de cadence égale à la latence d'exécution de toutes les opérations du graphe d'algorithme. Dans cette approche, pour garantir que toutes les contraintes sont satisfaites, il faut garantir que la cadence et donc la latence d'exécution du graphe complet (ensemble de toutes les lois de commandes) soit inférieure à la contrainte temporelle la plus sévère, c'est à dire que la durée d'exécution de tous les calculs d'une itération soit inférieure à cette contrainte. Cette condition est contraignante, car il n'est alors pas possible, par exemple, de réaliser un ordonnancement dès qu'une opération du graphe à une durée d'exécution supérieure à la contrainte de cadence la plus élevée. Dans ce chapitre, nous proposons une extension de la méthodologie AAA devant permettre d'implanter des algorithmes à contraintes temporelles multiples en s'affranchissant de cette condition contraignante.

Pour valider le système de contrôle-commande il est nécessaire de fournir à l'utilisateur la possibilité d'introduire dans le code de l'application des "sondes" ou "espions" permettant de mémoriser les données calculées pendant l'exécution et de les comparer hors exécution aux valeurs recherchées. La fin de ce chapitre présente des techniques d'implantation efficaces de ces espions permettant de minimiser les perturbations apportées par les calculs et les communications supplémentaires qu'elles nécessitent.

9.1 Implantation multi-contraintes

9.1.1 Contraintes d'implantation d'un algorithme spécifié par un graphe factorisé

Nous avons décrit §6.2 une extension de la sémantique de spécification du graphe d'algorithme afin de permettre la spécification de contraintes temporelles multiples, nous considérons ici que les

algorithmes que nous cherchons à implanter sont spécifiés à l'aide de cette sémantique.

9.1.1.1 Dépendances de données

Un graphe de dépendance factorisé est constitué d'un ensemble d'opérations reliées par des dépendances de données. Une dépendance de données entre deux opérations se traduit à l'implantation par une relation d'ordre d'exécution entre les deux opérations. Cette relation d'ordre d'exécution est très importante, elle doit être respectée par l'implantation sous peine de produire à l'exécution des résultats de calcul erronnés, ce qui conduirait à un mauvais fonctionnement du système de contrôle-commande pouvant avoir des conséquences catastrophiques. Pour respecter l'ordre d'exécution, il faut garantir que l'exécution d'une opération consommatrice d'une donnée ne commence qu'après la fin de l'exécution de l'opération productrice de la donnée. Cette relation d'ordre d'exécution peut être traduite par la notion de *prédécesseur* et de *successeur*. On appelle *prédécesseur* d'une opération toute opération ayant avec cette opération une relation d'ordre de type "doit s'exécuter avant". Le *successeur* d'une opération est une opération ayant avec cette opération une relation de type "doit s'exécuter après".

Pour implanter une spécification décrite à l'aide de graphes factorisés, il faut préciser cette notion de prédecesseur. En effet, dans ce type de graphe on peut exprimer des répétitions temporelles traduisant la mise en séquence de n instances d'une opération. On montre pour les différents cas de spécification possibles que s'il existe une dépendance de donnée entre deux opérations alors cette dépendances de données s'applique aux n instances des opérations répétées.

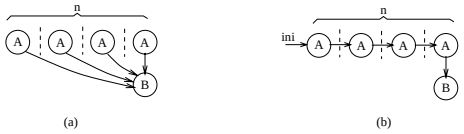


FIG. 9.1 – *Répétition temporelle d'une opération productrice*

Cas de la répétition temporelle d'une opération productrice (fig. 9.1) Dans le cas (a) de la figure 9.1 chaque itération successive de A produit une donnée consommée par B, chaque instance de A est donc un prédécesseur de B. Dans le cas (b) seule la dernière instance de B produit une donnée consommée par B, mais la répétition temporelle induit un ordre total sur l'exécution des instances de A. Ainsi l'instance i de A est prédécesseur de l'instance $i+1$. Par transitivité et puisque l'instance n de A est prédécesseur de B, alors chaque instance de A est prédécesseur de B.

FIG. 9.2 – *Répétition temporelle d'une opération consommatrice*

Cas de la répétition temporelle d'une opération consommatrice (fig. 9.2) Dans tous les cas de la figure 9.2 chaque instance de l'opération consommatrice B consomme une donnée de A qui est donc prédécesseur de chacune des instances de B.

L'implantation d'un graphe factorisé doit respecter les dépendances de données entre toutes les instances d'une opération et toutes les instances de ces prédécesseurs.

9.1.1.2 Contraintes de cadence

Les graphes factorisés permettent de spécifier des contraintes de cadences d'exécution sur les opérations. Plus précisemment, on spécifie la répétition temporelle des opérations et donc des rapports de rythme d'exécution entre les opérations. Dans ce contexte, il suffit de spécifier la cadence d'exécution d'une seule entrée ou sortie, ou bien encore la cadence d'exécution du graphe complet pour en déduire les cadences d'exécution de toutes les entrées et sorties. Dans les applications de contrôle-commande il y a généralement plusieurs entrées et plusieurs sorties pouvant être exécutées à des rythmes différents, ont dit que l'application est multi-cadencée. Ces contraintes de cadences sont les fréquences d'échantillonnages fixées par l'automaticien.

L'implantation d'un graphe factorisé doit respecter les contraintes de cadences d'acquisition des entrées et de mise à jour des sorties afin de respecter les fréquences d'échantillonnage fixées par l'automaticien.

9.1.1.3 Contraintes de latence

Chaque sortie est fonction d'un certain nombre d'entrées. La latence reflète le temps qui s'écoule entre la date de début d'exécution de la première entrée et la date de fin d'exécution de cette sortie. En d'autres termes la contrainte de latence fixe une borne sur le temps qui s'écoule entre l'instant où une entrée est acquise et l'instant où une sortie qui est calculée à partir de l'entrée est mise à jour. Cette latence est définie par l'automatitien, elle dépend du temps de réponse recherché pour l'asservissement. Dans la méthodologie AAA elle peut être spécifiée par un retard inséré sur toutes les entrées d'une opération de sortie (cf. §6.2.2.2).

L'implantation doit respecter les contraintes de latences spécifiées sur les sorties afin de garantir les temps de réponse et la stabilité des asservissements.

9.1.1.4 Contraintes de périodicité, jitter

L'automaticien, lorsqu'il conçoit une loi de commande discrète, fait l'hypothèse que les opérations d'échantillonnage et de reconstitution des signaux sont réalisées périodiquement. Des travaux [88] ont montrés que du jitter (variations temporelles) sur les instants d'échantillonnage pouvaient modifier les performances du système de contrôle-commande, voire même engendrer une instabilité.

Pour réaliser une implantation correcte des lois de commandes sur le calculateur il est donc indispensable de respecter au mieux la périodicité d'acquisition et de reconstitution des signaux continus. Il faut minimiser le jitter d'acquisition des entrées, le jitter de mise à jour des sorties, ainsi que le jitter sur la latence entre une entrée et une sortie.

L'exécution des opérations de calcul doit respecter les contraintes d'ordonnancement imposées par les dépendances de données, elles doivent aussi garantir des échéances imposées par les instants où les opérations de sorties doivent être exécutées, par contre l'exécution d'une opération de calcul

ne nécessite pas de respecter des contraintes de périodicité puisqu'elles ne réalisent pas d'échantillonnage.

L'implantation doit minimiser le jitter sur la période d'exécution des entrées et sorties et sur la latence des sorties. Les opérations de calcul ne sont pas nécessairement exécutées périodiquement.

9.1.2 Proposition d'une méthode d'ordonnancement hors ligne pour l'implantation d'une spécification multi-contrainte

Un ordonnancement hors ligne non préemptif permet de construire des exécutifs prédictibles très peu coûteux en utilisation mémoire et en temps de calcul. Un ordonnancement préemptif est moins prédictible, il apporte un surcoût sur le temps d'exécution plus important qu'un ordonnancement hors ligne non préemptif car il nécessite généralement d'intégrer à l'exécutif un ordonnanceur chargé de prendre des décisions d'ordonnancement à l'exécution. Néanmoins, la préemption est indispensable pour que l'implantation puisse satisfaire certaines contraintes temporelles. Nous cherchons ici à proposer une méthode d'ordonnancement à mis chemin entre un ordonnancement hors ligne non préemptif et un ordonnancement préemptif en espérant ainsi bénéficier des avantages des deux méthodes d'ordonnancement. Pour cela, nous avons introduit un minimum de préemption dans la méthode d'ordonnancement utilisée jusqu'alors dans la méthodologie AAA pour l'implantation d'une spécification mono-contrainte.

9.1.2.1 Respect des dépendances de données

Afin de respecter les dépendances de données, le principe de distribution et d'ordonnancement de l'heuristique de la méthodologie AAA qui calcule l'implantation d'une spécification mono-contrainte consiste à n'ordonnancer une opération que lorsque ces prédécesseurs sont déjà ordonnancés (cf. §8.2.2). Pour l'implantation d'un graphe factorisé ce principe reste valable, il suffit simplement de considérer que l'ordonnancement d'une opération itéré n fois consiste à ordonnancer les n instances de l'opération avant d'ordonnancer son successeur.

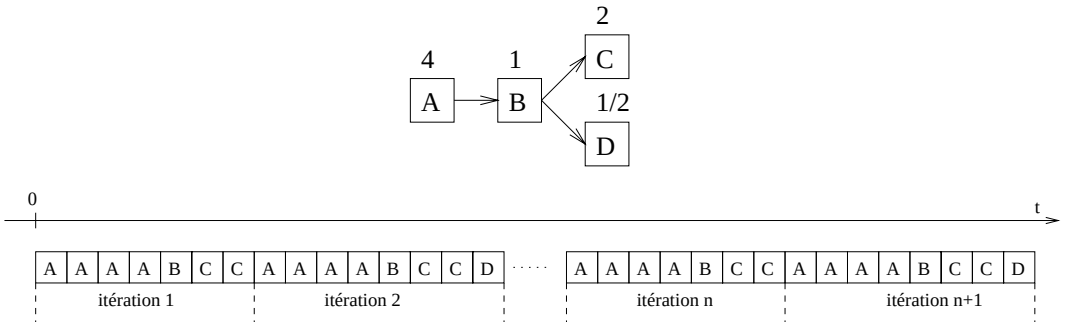


FIG. 9.3 – *Ordonnancement hors ligne non préemptif respectant les dépendances de données d'un graphe factorisé*

La figure 9.3 présente un exemple d'ordonancement basé sur ce principe. Le graphe définit une opération d'entrée A devant être exécutée 4 fois, une opération de calcul B exécutée une seule fois, et deux opérations de sorties C et D devant être exécutées respectivement 2 fois et une fois sur deux

dans une itération du graphe. L'opération A sans prédécesseur est ordonnancée, c'est à dire que l'on place en séquence les quatres instances de cette opération. Il est ensuite possible d'ordonnancer B puisque tous ces prédecesseurs (ici A) sont déjà ordonnancés. C est ensuite ordonnancé deux fois dans l'itération et D est ordonnancé une itération sur deux. Pour simplifier l'exemple nous avons considéré ici que toutes les opérations ont la même durée d'exécution. La séquence d'exécution ainsi définie est réitérée toutes les 40ms.

On remarque sur cet ordonnancement que l'itération n et l'itération $n+1$ n'ont pas tout à fais la même séquence. Ceci est dû au fait que D est exécutée une itération sur deux. De manière générale lorsque des opérations sont exécutées à des rythmes différents on peut observer ce phénomène. Le nombre de séquences différentes que l'on peut observer dépend des itérations spécifiées pour chaque opération. Sur une durée d'exécution appelée *hyperpériode* définie par le PPCM des périodes d'exécution de toutes les opérations on trouve toutes les séquences d'éxécution différentes. Cette hyperpériode définit ainsi un "motif" d'exécution répété tout au long de l'exécution de l'application. Sur notre exemple la période d'exécution d'une itération du graphe est de 40ms, on peut en déduire les périodes d'exécution de chacune des opérations soit $T_A = 10ms$, $T_B = 40ms$, $T_C = 20ms$, $T_D = 80ms$, soit une hyperpériode de 80ms, soit deux itérations du graphe. L'ordonnancement ainsi réalisé respecte les dépendances de données et le nombre d'itérations de chaque opération.

L'implantation d'une spécification multi-contraintes selon une méthode d'ordonnancement hors ligne non préemptive consiste à construire la séquence d'exécution des opérations et à exécuter itérativement cette séquence. Nous avons vu sur l'exemple précédent que la longueur de la séquence d'exécution qui doit être construite est égale au PPCM des périodes d'exécution de toutes les opérations. La séquence d'exécution qui doit être décrite peut être très longue et nécessiter un espace mémoire important pour être stockée dans les processeurs qui devront l'exécuter. Le choix de période d'exécutions multiples les unes des autres permettra de réduire la taille de cette séquence à la plus grande des périodes d'exécution.

Afin de limiter la taille de la séquence d'exécution nous faisons ici l'hypothèse que les nombres de répétitions des opérations sont multiples les uns des autres et donc que les périodes d'exécutions des opérations sont multiples les unes des autres. Cette hypothèse est raisonnable, car nous avons vu §1.3.2.1 que l'automaticien à une grande latitude sur le choix des fréquences d'échantillonnage. On doit donc, dans la majorité des cas, pouvoir se ramener à une spécification qui satisfait cette hypothèse.

Remarque : l'exécution d'une opération de calcul ne doit pas nécessairement être périodique. C'est donc un abus de language que d'utiliser le terme de "période d'exécution" sur une opération de calcul. En fait, la période d'exécution d'une opération de calcul, non nécessairement périodique, peut être définie par le nombre d'itérations de l'opération dans un intervalle de temps divisé par cet intervalle de temps. On définit ainsi une "période d'exécution moyenne". C'est à cette définition que l'on fait référence lorsque l'on parle de période d'exécution à propos d'une opération de calcul.

9.1.2.2 Condition d'ordonnançabilité

Soit un graphe d'algorithme factorisé A composé d'un ensemble O de n opérations, $O = \{OP_1, OP_2, ..., OP_n\}$. Nous considérons ici qu'une opération est caractérisée par une durée d'exécution maximale D_i et un nombre de répétition N_i dans la période d'exécution T du graphe.

Le graphe d'algorithme factorisé est ordonnançable sur un processeur si la durée d'exécution de tous les calculs qui compose le graphe est inférieure ou égale à la période d'exécution du graphe T.

Une opération Op_i est exécutée N_i fois dans la période T, sa durée d'exécution est D_i il faut donc un temps $N_i \times D_i$ pour exécuter les N_i itérations de Op_i dans la période T.

Ainsi il faut un temps $N_1 \times D_1 + ... + N_n \times D_n$ pour exécuter tous les calculs nécessaires à l'exécution complète du graphe factorisé. Une condition nécessaire et suffisante pour que le graphe factorisé soit ordonnançable sur un processeur est :

$$\sum_{i=1}^{n} N_i \times D_i \leq T$$

9.1.2.3 Respect des contraintes de périodicité sur les entrées-sorties

Sur l'ordonnancement présenté figure 9.3, on constate que l'entrée A et les sorties C et D ne sont pas exécutées à des intervalles de temps réguliers. Cet ordonnancement ne respecte donc pas les contraintes de périodicité sur les entrées/sorties et n'est donc pas satisfaisant pour l'implantation de lois de commande.

Pour garantir cette périodicité, il faut déclencher l'exécution des entrées/sorties par un événement dont l'occurence est périodique. En pratique, la technique la plus simple pour garantir cette périodicité est de déclencher une interruption sur l'occurence d'une horloge physique, le "timer" : horloge temps réel du processeur. Cette interruption déclenchera alors l'exécution d'une routine d'interruption dans laquelle les entrées/sorties sont exécutées en séquence. Ainsi, nous définissons une séquence d'exécution des opérations d'entrées/sorties (E/S) et une séquence d'exécution des opérations de calcul qui peut être préemptée par la séquence d'E/S.

La période de l'interruption devra être égale à la plus petite période d'exécution spécifée dans le graphe. Puisque nous avons fait l'hypothèse que les périodes d'exécution sont multiples les unes des autres, les opérations d'entrées/sorties auront des périodes d'exécution multiples de la période d'interruption. Ainsi, une opération dont la période T_i est égale à $k * T_{it}$ avec k entier, ne sera exécutée que toutes les k interruptions. Etant donné que les entrées/sorties ont des périodes d'exécution différentes, la séquence d'E/S ne sera pas la même d'une exécution sur l'autre. Pour décrire complètement l'exécution des E/S il faut décrire toutes les séquences d'E/S sur une période de temps équivalente à l'hyperpériode.

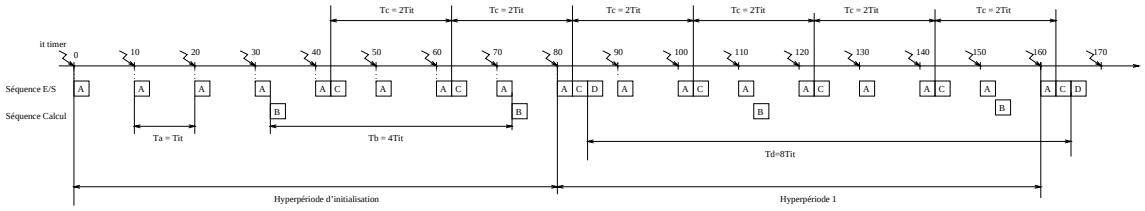


FIG. 9.4 – *Prise en compte de la contrainte de périodicité*

La figure 9.4 représente l'ordonnancement obtenu sur notre exemple. La période d'interruption Tit = PGCD(Ta, Tc, Td) est ici de 10ms. Les entrées/sorties sont ordonnancées dans les séquences d'E/S déclenchées par l'interruption. L'opération de calcul B est exécutée dans la séquence de calcul.

Avec cet ordonnancement toutes les entrées/sorties sont périodiques.

Sur cet exemple, on montre que la séquence d'exécution de la première hyperpériode est différente des autres hyperpériodes, car pour prendre en compte les dépendances de données il faut déphaser les sorties par rapport aux entrées. Ici la sortie C doit être exécutée toutes les 4 interruptions pour respecter sa fréquence d'exécution, mais elle doit aussi être exécutée après son prédécesseur, l'opération B. Pour respecter les dépendances de données B doit être exécutée après quatres exécutions de A. La séquence de calcul est donc déclenchée sur l'occurence de la quatrième interruption. C ne doit pas étre ordonnancée dans cette quatrième interruption car son exécution préempterait l'opération de calcul B qui est son prédecesseur, c'est pourquoi la première occurence de C est exécutée lors de la cinquième interruption.

9.1.2.4 Minimisation du jitter

La date d'exécution d'une opération d'entrée/sortie dépend du temps d'exécution des opérations qui sont ordonnancées avant elle dans la séquence d'E/S. Des variations sur les temps d'exécution de ses prédécesseurs provoqueront donc des variations sur la date de début d'exécution de l'opération considérée et donc du jitter sur sa période d'exécution.

Il existe deux causes possibles de variations du temps d'exécution des prédécesseurs. La première est que, d'une interruption sur l'autre, le nombre de prédécesseurs de l'opération peut être différent à cause des périodes d'exécutions différentes pour chaque entrée/sortie. Dans notre étude, nous avons émis l'hypothèse que les périodes d'exécution des opérations devaient être des multiples les unes des autres. Dans ce cas si on ordonnance les entrées/sorties selon la règle Rate Monotonic (périodes d'exécution croissantes), une entrée/sortie est exécutée à une période multiple de son prédécesseur. Ainsi, si une entrée/sortie doit être exécutée lors d'une interruption, son prédécesseur le sera. De cette manière on garantit que le nombre d'opérations d'entrée/sortie à exécuter avant une autre opération d'entrée/sortie est toujours constant. La deuxième cause de jitter sur la période d'exécution d'une opération d'entrée/sortie est due à la variation, d'une interruption sur l'autre, de la durée d'exécution de chaque prédécesseur. En effet, les opérations sont caractérisées par une durée d'exécution maximale permettant de calculer un ordonnancement garantissant que dans le pire cas les contraintes temporelles sont respectées. Dans certaines opérations, notamment celles qui intègrent des instructions de branchement conditionnel, le nombre d'instructions à exécuter par le processeur pour réaliser les calculs qui définissent l'opération peut varier d'une exécution sur l'autre. Cette variation sur le nombre d'instructions à exécuter ce traduit par une variation sur le temps d'exécution de l'opération. Ainsi, lorsqu'une entrée ou une sortie est exécutée après une ou plusieurs opérations, la période d'exécution de celle-ci est soumise à un jitter égale à la somme des jitter sur le temps d'exécution des opérations qui la précède. Réduire le jitter sur la période d'exécution d'une opération d'entrée/sortie nécessite donc de réduire le jitter sur le temps de calcul de chaque opération d'entrée/sortie. Une opération d'entrée/sortie doit donc être la plus simple possible. Nous préconisons le découpage des opérations d'entrées/sorties complexes en deux parties : une opération d'entrée/sortie minimum qui réalise seulement les quelques instructions assembleur nécessaire pour la lecture/écriture dans les registres des convertisseurs C.A.N et C.N.A, et une opération qui réalise du calcul sur la valeur qui est acquise par la première opération (cas d'une entrée), ou qui fournira à l'autre opération la valeur de la sortie à mettre à jour.

Dans la suite de ce document, nous considérerons systématiquement que dans la spécification que l'on cherche à implanter, les entrées/sorties sont réduites au minimum d'instructions

nécessaire pour lire ou écrire dans les registres des convertisseurs.

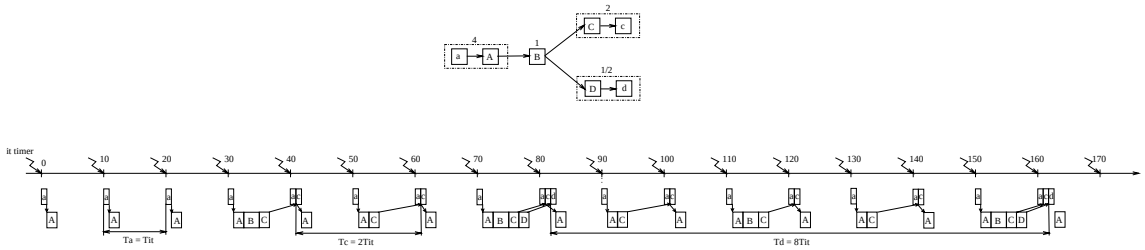


FIG. 9.5 – *Minimisation du jitter*

La figure 9.5 reprend l'exemple précédent. Le graphe d'algorithme a été modifié : le calcul de l'entrée a été décomposé en deux opérations a et A, il a été fait de même pour les sorties. Seule le calcul B n'a pas été modifié.

9.1.2.5 Prise en compte des temps d'exécution

Jusqu'alors nous n'avons pas pris en compte les temps de calcul des opérations, ou plutôt, dans l'exemple précédent, nous nous sommes placés dans un cas de figure favorable pour lequel la somme des durées des opérations était inférieur à la période d'interruption. Puisque la période d'interruption est égale à la plus petite période d'exécution spécifiée dans le graphe, entre deux interruptions, il ne peut y avoir au plus qu'une seule instance d'une opération donnée qui est exécutée. Ainsi, si la somme des durées d'exécution de toutes les opérations est inférieure à la période d'interruption, il est possible d'ordonnancer toutes les opérations entre deux interruptions. Nous sommes ici dans le cas favorable où la durée d'exécution de tous les calculs respecte la contrainte de cadence la plus sévère.

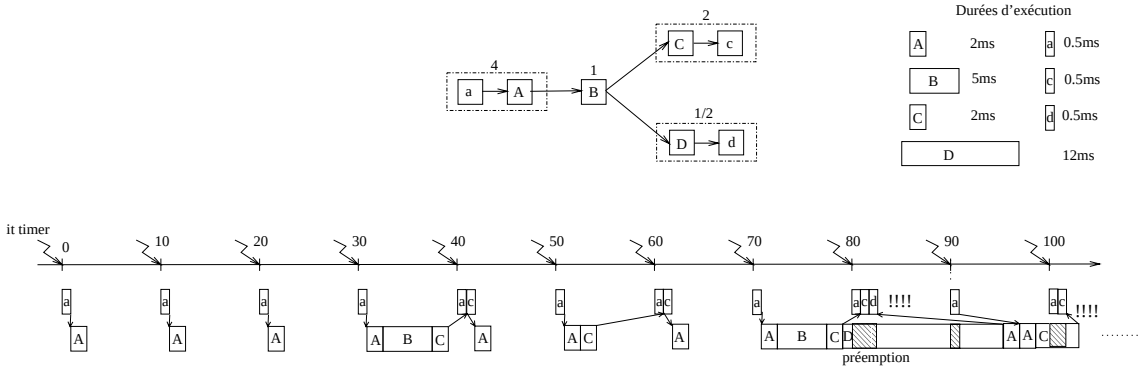


FIG. 9.6 – *Prise en compte des durées d'exécution : exemple d'ordonnancement impossible*

Reprenons l'exemple précédent, nous attribuons maintenant des durées d'exécutions aux opérations soit C_A=2ms, C_B=5ms, C_C=2ms, C_D=12ms, C_a=0.5ms, C_c=0.5ms et C_d=0.5ms. L'ordonnancement réalisé avec ces temps d'exécution ne permet pas de respecter les périodes d'exécution de c et de d. En effet, l'opération D ne peut être terminée avant l'occurence de la neuvième interruption qui l'interrompt donc. La sortie d est exécutée lors de cette interruption alors que l'exécution de son prédécesseur D n'est pas terminée. De plus, les calculs de A et C sont retardés et l'exécution de C ne peut plus être terminée avant l'exécution de l'instance suivante de c.

Pour remédier à ce problème il suffit d'ajouter une période d'interruption à la phase d'exécution de c et de d. Ainsi les sorties c et d n'interromprons plus le calcul de leurs prédécesseurs.

9.1.2.6 Prise en compte des contraintes de latence

Lorsque la somme des durées des opérations est supérieure à la période d'interruption nous avons vu qu'il pouvait être nécessaire de retarder l'exécution des sorties. Ce retard sera d'autant plus grand qu'une opération aura une durée importante. Par exemple, si sur le graphe précédent les opérations D et d ont une période d'exécution de 320ms et que D à une durée d'exécution de l'ordre de 50ms soit 5 périodes d'interruption, il faudra décaler les sorties c et d d'au moins cinq interruptions pour respecter leur périodicité. La latence est défine comme l'intervalle de temps qui sépare le début d'exécution d'une entrée et la fin d'exécution d'une sortie calculée à partir de la valeur de l'entrée. Donc, si on retarde les sorties par rapport aux entrées on rallonge la latence, on risque ainsi de ne plus satisfaire les contraintes de latences sur les sorties.

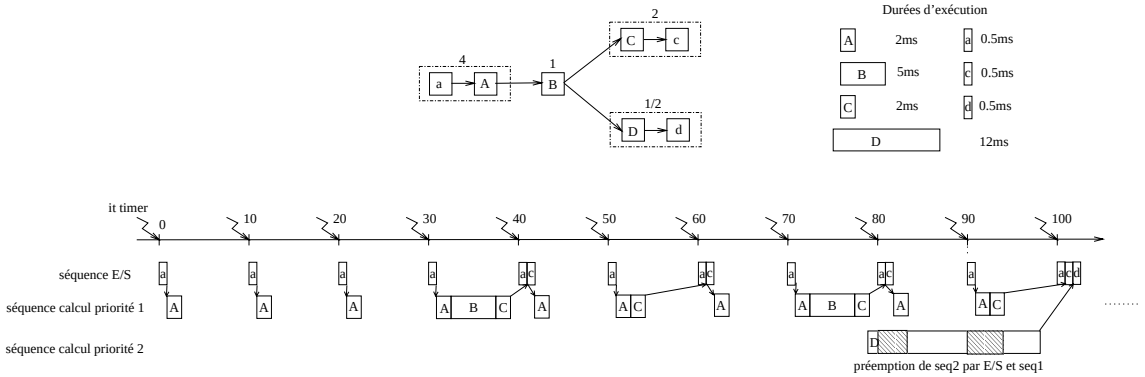


FIG. 9.7 – *Prise en compte des latences d'exécution : préemption entre plusieurs séquences de calcul*

Afin d'éviter d'alloger les latences des sorties, nous proposons d'ordonnancer les calculs dans plusieurs séquences de calcul. Chaque séquence de calcul aura une priorité d'exécution, une séquence pouvant être interrompue par une séquence de plus haute priorité. Bien entendu, la séquence d'E/S ne pourra être interrompue par aucune autre séquence. Le principe d'ordonnancement reste le même que celui définie précédemment à la différence que lorsqu'une opération est ordonnancée dans une séquence de calcul, on vérifie que si la date de fin oblige à retarder les sorties pour pouvoir respecter les cadences, alors une nouvelle séquence de calcul est créée et l'opération est ordonnancée dans cette nouvelle séquence. A partir de ce moment, afin d'éviter que cette opération ne puisse être interrompue par un successeur, toutes les autres opérations restant à ordonnancer le seront dans cette nouvelle séquence de calcul ou dans des séquences de calcul de priorités inférieures. La figure 9.7 présente l'ordonnancement obtenue avec cette technique. On remarque que seule la sortie d a du être retardée, la sortie c n'a pas été pénalisée par un retard supplémentaire.

9.1.2.7 Algorithme d'ordonnancement proposé

A partir des principes énoncés précédemment, il est possible de proposer un algorithme d'ordonnancement dont les principes sont relativement proches de ceux définis dans l'heuristique d'ordon-

nancement de la méthodologie AAA pour l'implantation de spécifications mono-contrainte. L'ordonnancement sera calculé sur une hyperpériode qui sera répétée tout au long de l'exécution de l'application.

Notations

- Soit un graphe factorisé comportant un ensemble de p opérations OP, on note T la période de l'itération du graphe complet.
- On note $OP_e(n, D)$ une opération d'entrée de durée D itérée n fois dans la période T. Sa période d'exécution est T/n.
- On note $OP_s(n, D, Rt)$ une opération de sortie de durée D itérée n fois dans la période T. Sa période d'exécution est T/n. Rt est le retard appliqué à tous ses signaux d'entrées définissant la latence $L = Rt*$ période de la sortie, soit $L = Rt * (T/n)$.
- On note $OP_c(n, D)$ une opération de calcul de durée D itérée n fois.
- T_{it} est la période d'interruption définie par $T_{it} = T/max$(nombre de répétition n de chaque opération).
- $ESFS_i$ [1] est la date de début d'exécution au plus tôt depuis le début associée à l'opération OPi.
- $EEFS_i$ [2] est la date de fin d'exécution au plus tôt depuis le début associée à l'opération OPi.
- $LSFS_i$ [3] est la date de début d'exécution au plus tard depuis le début associée à l'opération OPi.
- $LEFS_i$ [4] est la date de fin d'exécution au plus tard depuis le début associée à l'opération OPi.

Principe de l'ordonnancement

1. **Ordonnancement des E/S suivant la règle Rate Monotonic**
   - une entrée i est ordonnancée avec une phase nulle c'est à dire que sa première occurence est ordonnancée lors de la première interruption, les autres occurences sont ordonnancées toutes les n interruptions, avec n = période de l'entrée / période de l'interruption. On calcul les $ESFS_i$ et les $EEFS_i$ de tous les successeurs à partir de la date de fin de l'opération.
   - Une sortie est ordonnancée sur le même principe, mais avec une phase égale à sa contrainte de latence. On calcule les $LSFS_i$ et les $LEFS_i$ de tous les prédécesseurs de l'opération en considérant que la date de début de l'opération est L.

1. Earliest Start From Start
2. Earliest End From Start
3. Last Start From Start
4. Last End From Start

2. **Ordonnancement des opérations de calcul**

   Tant qu'il reste des opérations non ordonnancées faire :

   – Construction de la liste des candidats ordonnançables, c'est à dire des opérations dont les prédécesseurs sont ordonnancés.

   – Ordonnancement du candidat ayant le plus faible $LESFS_i$: pour chaque période d'exécution on crée une liste représentant une séquence de calcul qui peut "potentiellement" contenir une séquence d'opérations à exécuter. Lorsqu'une opération est ordonnancée (placement dans une liste) on vérifie que la somme des durées des calculs contenus dans la séquence est inférieure à sa période d'exécution. Si ce n'est pas le cas l'opération est ordonnancée dans la liste correspondant à la séquence suivante de priorité inférieure. La liste dans laquelle l'opération est placée est alors la liste courante dans laquelle on cherchera à ordonnancer les successeurs. Au début de l'ordonnancement la liste courante est celle qui correspond à la séquence de calcul de plus haute priorité. Lorsque l'on ordonnance une opération dans une séquence d'exécution correspondant à une période T_i on calcule le nombre d'instances $Ti/(T/n)$ de l'opération qui doivent être placées en séquence dans la liste.

   – Calcul des $ESFS_i$ et $LSFS_i$ de tous les successeurs de l'opération à partir de la date de fin d'exécution de l'opération.

9.1.2.8 Génération de code

L'algorithme d'ordonnancement appliqué à notre exemple produit cinq listes dont chaque élément est une paire (nom de l'opération, nombre d'instances par période d'exécution de la séquence) :

– **liste de la séquence E/S :** { (a,1), (c,1/2), (d,1/8) }, période d'exécution T_{it}

– **liste de la séquence de calcul 1 :** { (A,1), (B,1/4), (C,1/2) }, période d'exécution T_{it}

– **liste de la séquence de calcul 2 :** { (D,1/4)}, période d'exécution $T_{it}/2$

– **liste de la séquence de calcul 3 :** { Ø}, période d'exécution $T_{it}/4$

– **liste de la séquence de calcul 4 :** { Ø}, période d'exécution $T_{it}/8$

Sur cet exemple on considèrera que l'ordonnancement a été calculé pour un retard $Rt = 1$ spécifié sur c et un retard $RT = 2$ spécifié sur d.

La génération de code permettant d'implanter cet ordonnancement est très simple :

```
# Pour chaque opération de la liste E/S on initialise un compteur
#################################################################
# compt_a = 1;  inutile car phase=0 est exécutée à chaque interruption
compt_c = 5;  # phase + 1
compt_d = 10; # phase +1
```

```
# Pour chaque séquence d'exécution on initialise un compteur à 1
################################################################
compt_seq1 = 1;
compt_seq2 = 1;
# compt_seq3 et compt_seq4 inutile car vide

# Pour chaque opération de calcul on initialise un compteur
################################################################
# compt_A inutile, phase = 0, exécuté dans chaque occurence de seq1
compt_B = 4; # phase + 1
compt_C = 4; #phase + 1
compt_D = 1; # phase par rapport au début de seq2 = 0 + 1

# Description de la séquence d'interruption
################################################################
IT {
    # séquence d'E/S
    ################
       disable interrupt;
       a;                                    #opération a
       if(--compt_c == 0) {compt_c=2; c;}  #opération c, période 1/2 => compt_c =2
       if(--compt_d == 0) {compt_d=8; d;}  #opération d
       enable interrupt;

   # séquence calc 1, période IT/2
   ###############################
     if(--compt_seq1 == 0) {
                compt_seq1=2;
        A;                                 #opération A
       if(--compt_B==0){compt_B=4; B;} #opération B
if(--compt_C==0){compt_C=2; C;} #opération C

# séquence calc 2, période IT/4
                ################################
if{--compt_seq2 == 0) {
compt_seq2 = 2;
if(--compt_D==0) {compt_D=2; D} #operation D
}
      }
     rti
}
```

9.1.2.9 Prise en compte de la distribution

L'extension de la méthode d'ordonnancement à la prise en compte de la distribution des opérations sur un ensemble de processeurs ne change rien du point de vue de l'ordonnancement des entrées/sorties car une entrée/sortie doit obligatoirement être exécutée sur le processeur auquel est relié le capteur ou l'actionneur géré par l'opération d'entrée/sortie. Ainsi l'ordonnancement des E/S est réalisé séparement sur chacun des processeurs.

Par contre lorsque l'on cherchera à ordonnancer une opération de calcul, il faudra la placer sur chacun des processeurs de l'architecture, puis comparer la date de fin d'exécution au plus tôt obtenue pour chacun de ces placements. L'opération est alors placée définitivement sur le processeur donnant la date de fin d'exécution au plus tard la plus faible. Ce calcul de date au plus tard devra tenir compte des éventuelles opérations de communications engendrées par ce placement, opérations de communications qui seront ordonnancées sur les médias de communication comme cela est fait actuellement en mono-contrainte.

Les séquences d'entrées/sorties et de calcul sont exécutées sur interruption sur chaque processeur, interruptions générées par des horloges locales à chaque processeur qui risquent de se déphaser dans le temps. Ce déphasage peut créer sur les communications un retard d'une période d'interruption. Dans ce cas les contraintes temporelles risquent de ne plus être satisfaites. Il faut donc prendre en compte ce déphasage dans les temps de communication. Afin d'éviter un retard trop important, il est nécessaire de maîtriser la dérive des horloges en les resynchronisant régulièrement (par exemple à chaque fin d'hyperpériode).

9.2 Mise au point des lois de commandes à l'aide d'espions

Lorsqu'on cherche à mettre au point les lois de commande d'une application temps réel, il est intéressant de disposer d'outils permettant "d'espionner" certains signaux. On veut, par exemple, observer la réponse du système à une entrée échelon (temps de réponse, stabilité ...) afin d'affiner les paramètres des lois de commande. Pour réaliser cet espionnage il est nécessaire d'intégrer des fonctions de mémorisation au code de l'application. L'exécution de ces fonctions doit perturber le moins possible le comportement de l'application. Ces fonctions ne doivent donc pas être coûteuses en temps d'exécution afin d'introduire des retards minimaux dans les lois de commande.

Le logiciel SynDEx permet facilement d'ajouter de telles fonctions à une application, il suffit pour cela d'ajouter au graphe d'algorithme des opérations réalisant cette fonction, puis de réaliser une adéquation et enfin de regénérer le code de l'application. Les dépendances de données entre les opérations sont conservées, l'application avec ou sans espions se comporte de façon identique. Néanmoins certains processeurs de l'architecture peuvent ne pas disposer de suffisamment de mémoire pour stocker les valeurs des signaux espionnés. Ces valeurs doivent donc être stockées sur d'autres processeurs. Le coût des communications peut alors sérieusement perturber le comportement temporel de l'application. Nous apportons ici des techniques permettant de réduire ces coûts en partageant de la bande passante de communication entre les espions. On peut par exemple définir plusieurs espions qui communiqueront une valeur chacun leur tour. L'espionnage de chacun des signaux est moins fréquent mais l'espionnage d'un ensemble de signaux perturbe beaucoup moins l'application. De même certains signaux à espionner peuvent être conditionnés. C'est-à-dire qu'à certains moments ces signaux ne sont pas calculés. L'espionnage de ces signaux de sera donc réalisé

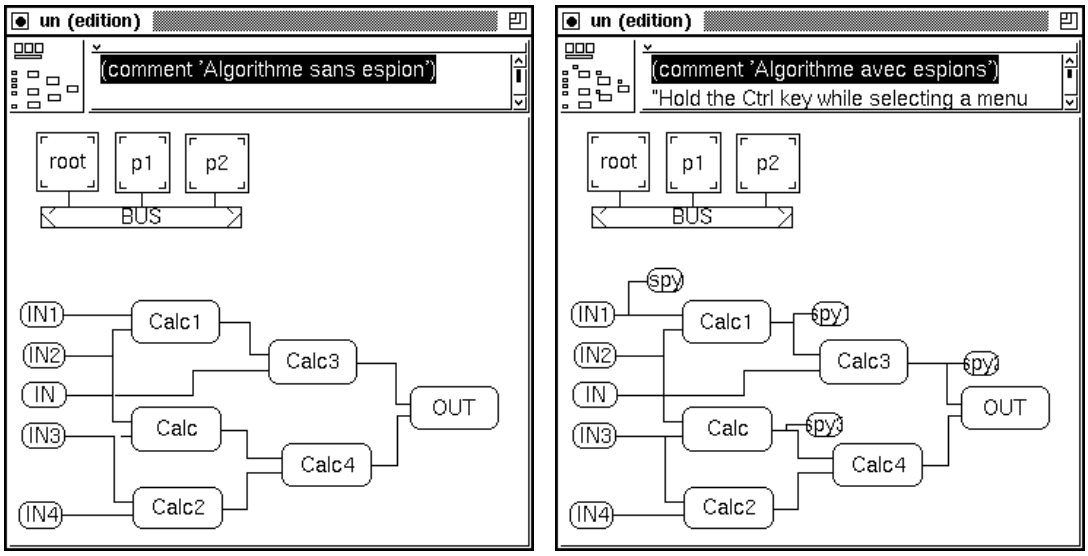
qu'aux instants où ces signaux sont calculés. La connaissance du conditionnement nous permet à la sauvegarde sur disque des espions, c'est à dire lorsque l'application est arrêtée, de dater chacune des valeurs mémorisées afin de reconstruire un signal temporellement correct.

Pour mieux prendre en compte les besoins de l'automaticien dans SynDEx, nous cherchons à ajouter de telles fonctions "d'espionnage" sur un graphe flot de données. Le but est de trouver une manière simple et efficace de les intégrer dans le générateur de code de SynDEx afin de procurer à l'utilisateur un moyen rapide et simple d'espionnage sans que celui-ci n'ait à écrire de code.

9.2.1 Principe de l'espionnage

Ce problème "d'espionnage" de données est à rapprocher des problèmes rencontrés en métrologie : il faut réaliser des mesures sur un système physique sans que celles-ci perturbent ou modifient le fonctionnement de celui-ci. Sachant que ce n'est bien évidemment physiquement pas réalisable, nous chercherons donc à minimiser les perturbations apportées à l'exécution par cette fonctionnalité.

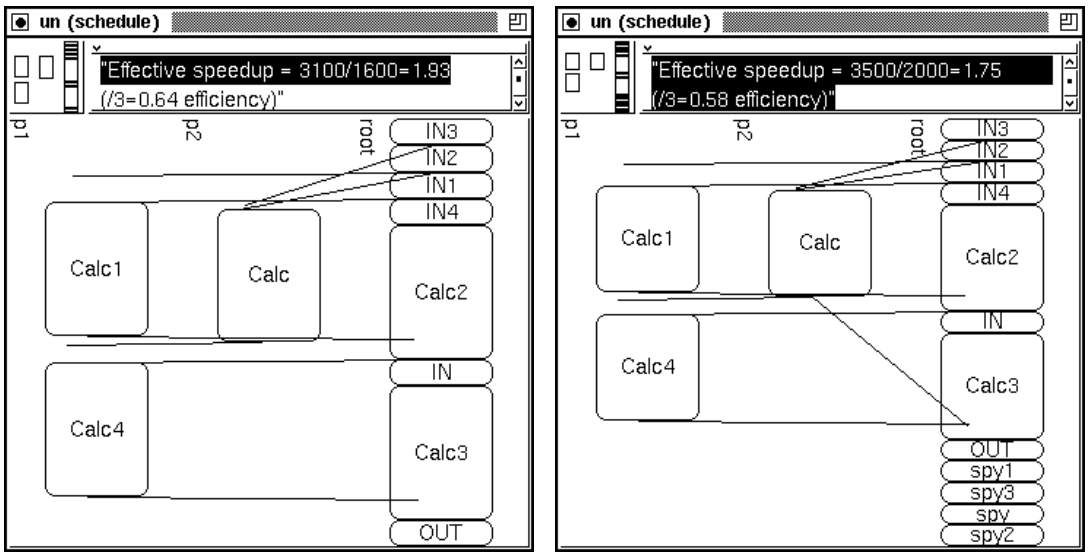
La manière la plus simple de réaliser de l'espionnage sur un graphe flot de données SynDEx consiste à ajouter des fonctions de sorties prennant en entrée les signaux à espionner. Ainsi ces fonctions se chargeront de réaliser la sauvegarde des données espionnées à chaque exécution de la boucle temps-réelle.(fig.9.8)



(a) Sans Espion (b) Avec Espions

FIG. 9.8 – *Principe de l'espionnage*

Lorsqu'on intègre au graphe de tels sommets, ceux-ci vont venir s'ajouter à la liste des opérations à exécuter dans la boucle temps-réelle et seront donc prises en compte par l'heuristique A^3 de SynDEx. Ainsi l'ordonnancement et la distribution des opérations de l'algorithme seront modifiées (fig.9.9). Ceci est la première perturbation qu'apportera l'ajout d'espions à l'application. En fait, ceci ne pose pas de problème particulier car SynDEx garanti que les dépendances de données entre



(a) Sans Espion (b) Avec Espions

FIG. 9.9 – *Influence de l'espionnage sur l'ordonnancement*

les opérations seront respectées quelque soit l'ordonnancement des opérations.

La deuxième perturbation sera un allongement de la boucle temps-réelle du à un surcout de calcul et de communications. Cet inconvénient est beaucoup plus génant puisqu'il pourra aboutir d'une part à un dépassement des contraintes temps-réelles d'autre part à des erreurs de calcul (cas des filtres récursifs dont les caractéristiques dépendent de la période temps-réelle).

En conséquence, la réduction des perturbations apportées par les espions consistera donc simplement à minimiser le surcout d'exécution de l'espionnage sur le temps d'exécution de la boucle temps réelle.

Dans cette optique, quelque soit l'architecture matérielle du système sur lequel est programmée l'application, la sauvegarde des données sera effectuée en deux temps : pendant l'exécution de la boucle temps-réelle les données seront mises en mémoire, à la fin de l'application le processeur ayant un accès à une mémoire de masse enregistrera dans des fichiers les valeurs mémorisées. On fait donc ici l'hypothèse qu'au moins un processeur dispose d'une mémoire suffisante à ces mémorisations. L'espionnage n'étant utile que pendant la phase de conception des systèmes, ce surplus de mémoire n'est nécessaire que sur le prototype de mise au point. Grâce à cette méthode les enregistrements sur disque ne s'effectueront jamais pendant l'éxecution de la boucle temps-réelle car les temps d'accès sont longs et variables.

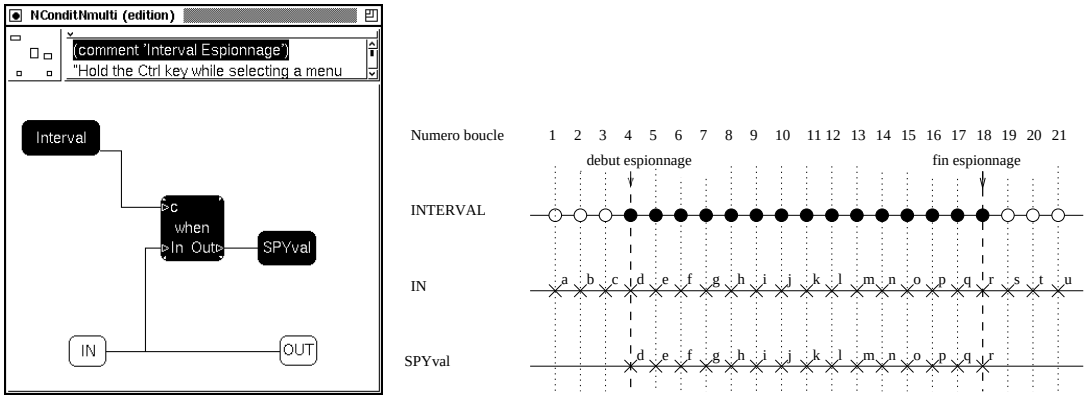
Afin de pouvoir au mieux exploiter les données espionnées, nous enregistrerons dans les fichiers des couples (date_acquisition, valeur). Cette datation sera réalisée par l'espionnage des valeurs du timer du processeur 'root' à chaque occurence de l'horloge la plus fine du graphe (c.a.d à chaque itération de la boucle-temps réelle). La position dans le tableau des valeurs acquises de la donnée

permettra ainsi de connaître la date d'espionnage à une période temps-réelle près. L'écriture d'un couple (date_acquisition,valeur) plutot qu'un couple (numero_itération_boucle, valeur) donnera des informations supplémentaires en terme de durée et de variations de durée de la période temps-réelle et permettra d'effectuer la correspondance entre le temps logique exprimé dans le graphe et le temps physique écoulé à l'exécution.

9.2.2 Intervalle d'espionnage

Ne connaissant pas a priori la durée de vie de l'application et ne disposant pas de ressources mémoires infinies, nous devrons limiter l'espionnage à une plage de temps que pourra définir l'utilisateur en fonction de son architecture et de ces besoins. Cette plage de temps sera décrite sous la forme d'un intervalle ayant une date de début par rapport à l'instant de la première exécution de la boucle temps-réelle et une longueur exprimée en nombre d'itération de celle-ci. Le fait de pouvoir spécifier la date de début permets à l'utilisateur d'adapter l'espionnage au phénomène physique qu'il veut étudier (transitoire, régime permanent...).

Sur un graphe, cette fonctionnalité peut être exprimée en conditionnant des signaux, c'est à dire en modifiant les horloges de ces signaux.



(a) Principe

(b) Diagramme temporel logique

FIG. 9.10 – *Espionnage sur un intervalle de temps*

La figure 9.10(a) donne le principe de cette restriction de l'espionnage sur un intervalle de temps. Sur tous les graphes de ce document, les sommets blancs représenteront l'algorithme de l'utilisateur, les sommets noirs seront les sommêts rajoutés afin de réaliser de l'espionnage.

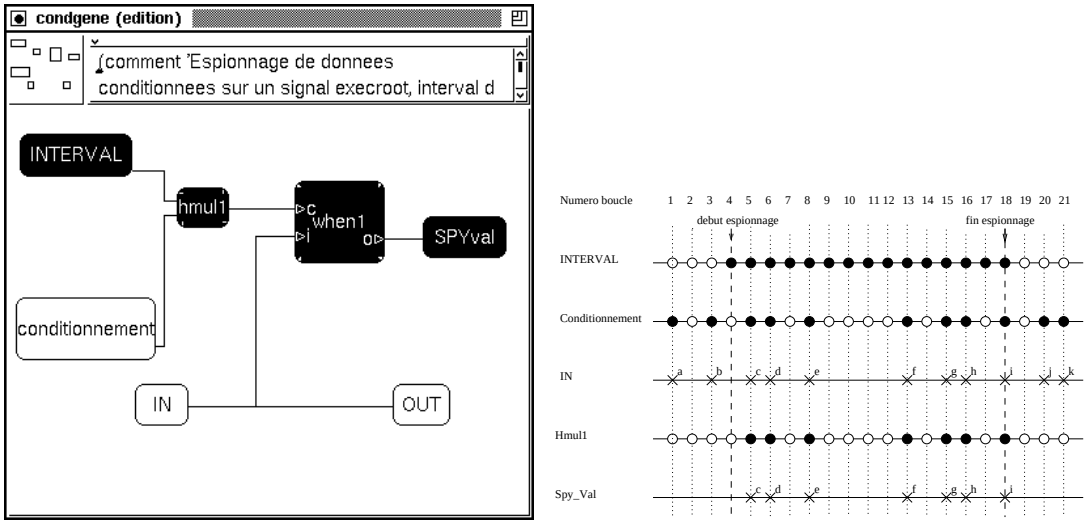
A ce graphe nous avons associé un diagramme temporel logique (fig.9.10(b)) représentant la valeur de certains flots à chaque itération de la boucle temps-réelle. Les conventions de représentation sont les suivantes : les flots de type booléens sont représentés par des cercles blancs (resp. noir) lorsque leur valeur est *vrai* (resp. *faux*). Les croix représentent l'horloge des flots non booléens, la valeur étant indiquée à coté de chacune d'elles. L'absence (resp. la présence) de croix ou de cercle dénote l'absence (resp. la présence) de valeur.

Sur le graphe 9.10(a) l'opération `Interval` génère un booléen. La valeur de ce booléen est *vrai* lorsque la date courante est dans l'intervalle de temps d'espionnage, *faux* sinon. L' opération

d'espionnage `SPYval` est précédée d'un sommet `when` dont l'entrée logique *c* est reliée à la sortie booléene d' `Interval`. Le `when` ne fournit des valeurs à `SPYval` que lorsque sont entrée logique est *vrai* c'est à dire pendant l'intervalle de temps d'espionnage. Ainsi, le sommet `SPYval` n'est exécuté que lorsque ce même booléen est *vrai.* L'application n'est ainsi perturbée que pendant l'intervalle d'espionnage.

9.2.3 Cas de l'espionnage de signaux conditionnés

Dans un graphe SynDEx, certains sommets peuvent ne pas être exécutés à chaque itération de la boucle temps-réelle. L'horloge d'exécution de ces opérations est donc un sous-echantillonnage de l'horloge d'exécution du graphe (qui nous le rappelons est l'horloge la plus fine, elle est appelée *execroot* dans SynDEx v4). On dit alors que ce sommet est conditionné, il n'est exécuté que lorsque la valeur de la sortie booléenne du sommet qui calcule cette valeur est *vrai.* Lorsque l'on veut espionner une donnée produite par un sommet conditionné, le principe d'espionnage décrit précédemment n'est pas utilisable tel quel.



(a) Principe

(b) Diagramme temporel logique

FIG. 9.11 – *Espionnage d'un signal conditionné*

En effet, lorsque le flot à espionner n'est pas produit par une opération conditionnée, celui-ci à pour horloge l'horloge la plus fine du graphe (*execroot*). Ainsi, le `when` introduit précédemment entre l'opération productrice du flot espionné et l' opération `SPYval` permet de sous-échantilloner le flot à espionner et de produire un flot dont l'horloge est exprimée par le booléen de la fonction `Interval` (nous l'appellerons *horloge_interval*).

Supposons maintenant que la fréquence d'horloge du flot de données *horloge_condit* à espionner soit plus basse que *execroot,* il se peut alors que l'espion soit exécuté (*horloge_interval*) alors que la donnée n'est pas produite. Ceci met en évidence la nécessité d'adapter l'horloge d'exécution de l'espion à l'horloge du flot espionné. En d'autre termes, on cherche à ne réaliser l'espionnage que pendant l'intervalle de temps défini, seulement si une donnée est produite. Il suffit pour cela d'utiliser l'opération `hmul` qui permet de générer l'intersection deux horloges (ici *horloge_condit* et

horloge_interval). C'est sur cette nouvelle horloge que le `when` échantillonera les données et c'est aussi sur celle-ci que `SPYval` sera exécuté. La figure 9.11(a)) illustre cette technique; les sommets `IN` et `OUT` sont conditionnés par la sortie booléene du sommet `condition`. Le diagramme temporel logique présenté à la figure 9.11(b) montre que le sommet `SPYval` ne mémorise que les données calculées pendant l'intervalle de temps d'espionnage (c.a.d ici les valeurs (*c,d,e,f,g,h,i*) du signal produit par `IN`).

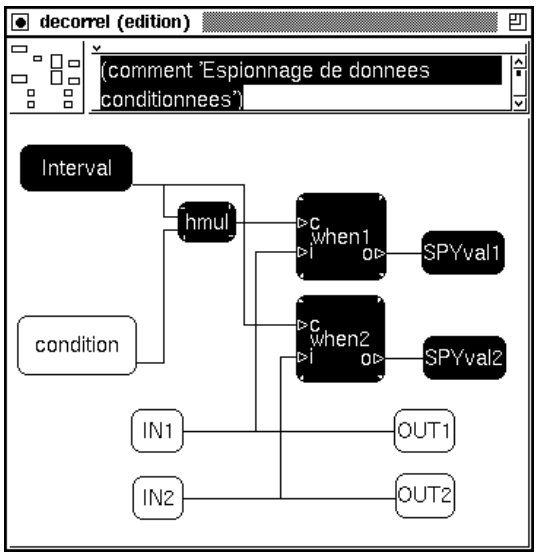
Cette technique permets donc bien d'adapter l'exécution de l'espion à l'horloge du signal à espionner. Par contre, elle ne prend pas en compte les problèmes "d'exploitation" des données receuillies. Si dans une même application, nous voulons effectuer l'espionnage de données conditionnées en même temps que l'espionnage de données non conditionnées, ou l'espionnage de données conditionnées sur des horloges différentes, il y aura décorrélation des données espionnées au moment de l'écriture sur disque. En effet, selon le principe décrit précédemment, la valeur espionnée ne sera enregistrée que dans l'intervalle de temps d'espionnage, que lorsque celle-ci est produite. Dès lors, les données calculées à des horloges différentes seront enregistrées à des instants différents. De même, certaines valeurs pourront être enregistrées moins souvent que d'autres. Les signaux enregistrés dans les fichiers seront donc temporellement décorrélés.

Sur le graphe 9.12(a) l'algorithme est composé de 2 sommets (`IN1`,`OUT1`) exécutés à l'horloge *horloge_condition* générée par le sommet `condition` ainsi que de 2 sommets (`IN2`,`OUT2`) exécutés inconditionnellement, donc à l'horloge *execroot*. La fonction `SPYval1` réalise l'espionnage de la sortie de `IN1`, le `hmul` produisant l'horloge commune à `interval` et `condition` nécessaire à l'espionnage de données conditionnées. La fonction `SPYval2` réalise l'espionnage de la sortie de `IN2`.

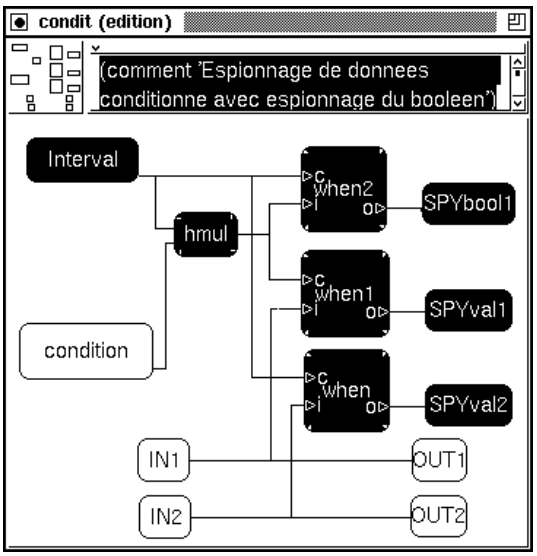
Le diagramme temporel 9.12(c) montre que l'opération *condition* génère le booléen d'horloge d'exécution des sommets `IN1` et `OUT1`, ce qui explique que `IN1` ne produise des valeurs que lorsque cette donnée est `vrai`. `IN2` produit des valeurs à chaque boucle. `hmul` produit un booléen *vrai* lorsque les sorties d'*Interval* et de *condition* le sont. `SPYval1` n'enregistre les données de *IN1* que lorsque `hmul1` est *vrai*. `SPYval2` espionne `IN2` lorsque `interval` est *vrai*.

La figure 9.12(e) visualise les fichiers d'espionnages génères par cette application. `SPYval1` ayant été exécutée moins souvent que `SPYval2` le fichier IN1.esp contient moins de valeurs que le fichier IN2.esp. La visualisation de IN2.esp est correcte, chaque point (valeur,date) est correcte. La visualisation de IN1.esp est incorrecte les valeurs sont associées à des dates fausses (cf. fig.9.12(c)).

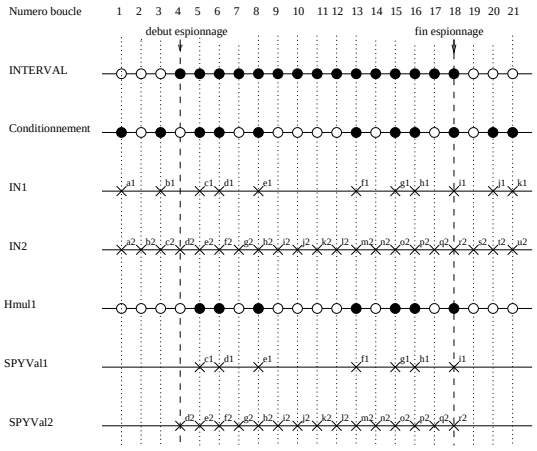
Pour pallier à ce problème, il faut trouver une solution qui permette de faire apparaître dans le fichier que l'espion n'a pas été exécuté. Pour cela, il suffit de dater l'acquisition des espions conditionnés afin de pouvoir reconstituer un signal temporellement correcte à l'écriture dans le fichier. Afin de limiter les surcouts d'occupation mémoire nous n'enregistrerons pas l'instant auquel à été espionnée la donnée, mais nous enregistrerons l'horloge de l'opération productrice de la donnée à espionner (c.a.d l'état du booléen conditionnant cette opération). Au moment de l'écriture sur disque nous pourrons ainsi reconstituer les couples (date,valeur). Seuls ceux correspondant à un état *vrai* du booléen conditionant le calcul de la donnée espionnée seront enregistrés. L'utilisateur pourra ainsi comparer facilement entre eux les signaux espionnés, et pourra aussi aisément connaître à quelles dates on été calculées ces valeurs.



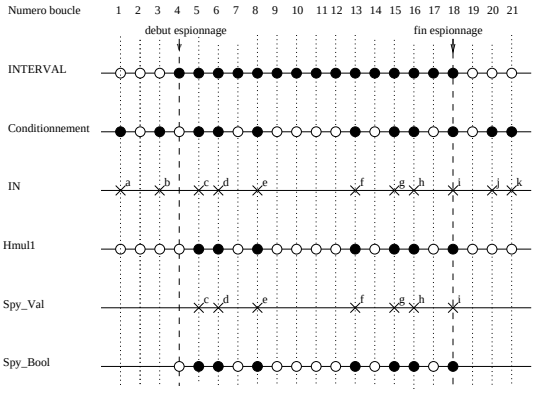
(a)



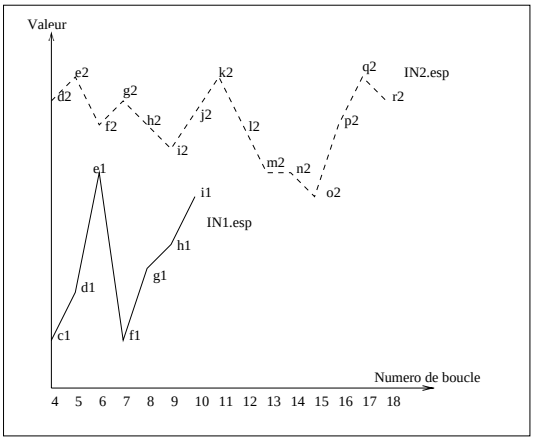
(b)



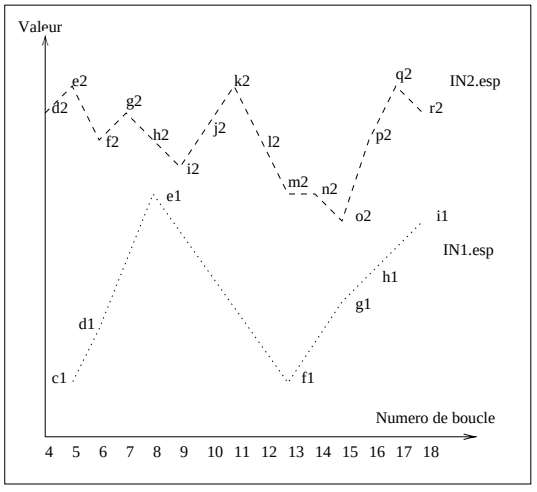
(c) Simulation des flots de données



(d) Simulation des flots de données



(e) Allure des fichiers



(f) Allure des fichiers

FIG. 9.12 – *Problème de corrélation temporelle des données espionnées*

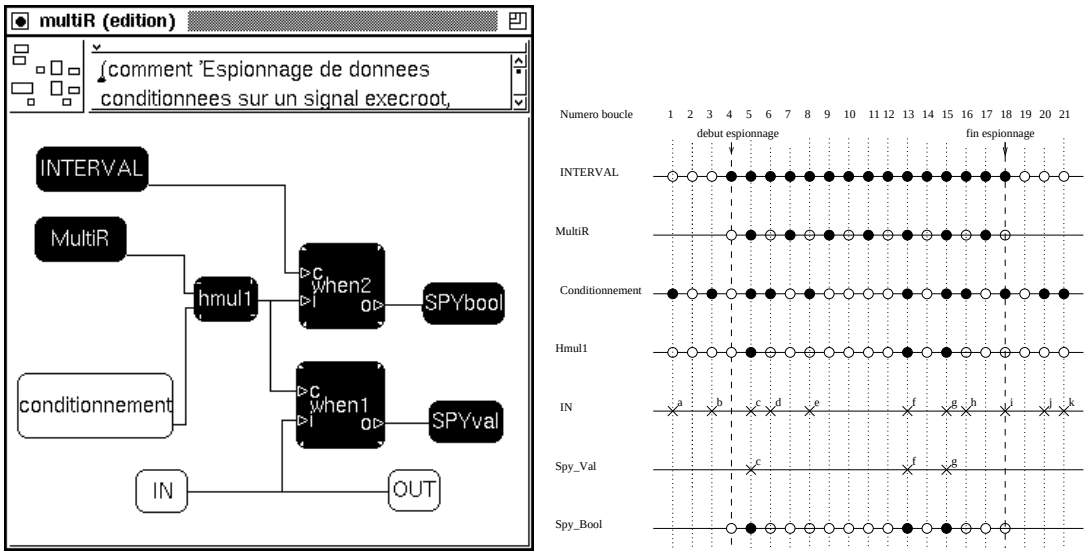
La figure 9.12(b) montre les modifications ainsi apportées à l'espionnage `IN1`. On rajoute un espion `SPYBool1`, exécuté sur l'horloge *horloge_nterval,* celui-ci enregistrera les valeurs de la sortie du `hmul` (c.a.d les instants auxquels `SPYVal1` est exécutée. Le diagrame temporel logique 9.12(d) montre que les flots de sorties des sommêts `Interval`, `condition`, `IN1`, `hmul`, `SPYval1`, `IN2`, `SPYval2` sont les même que précedemment. On a simplement ajouté l'espionnage de la sortie du `hmul1` seulement lorsque *horloge_interval* est *vrai.* La représentation des fichiers générés (fig. 9.12(f)) montre que la valeur du booléen espionné permets d'associer correctement une date à chaque valeur.

9.2.4 Prise en compte des ressources mémoire

La limitation des capacités mémoire de l'architecture matérielle peut induire deux problèmes:

– certains processeurs peuvent ne pas posséder assez de mémoire pour pouvoir stocker les données espionnées pendant l'exécution. Dans ce cas, le stockage de ces valeurs doit s'effectuer sur un autre processeur que celui sur lequel l'opération productrice de la donnée à espionnée est exécutée. Ces valeurs doivent donc être communiquées à ce processeur. Les communications étant souvent lentes par rapport aux calculs, l'influence de tels espions risque de ne pas être négligeable.

– l'utilisateur peut avoir besoin d'espionner un certain nombre de données sur un grand laps de temps, ce qui peut aboutir à des dépassement de capacité mémoire.



(a) (b) Simulation des flots de données

FIG. 9.13 – *Espionnage Sous echantilloné*

La solution à ces 2 problèmes consiste simplement à ne pas espionner systématiquement les données à chaque fois qu'elles sont générées. En effet, pour résoudre le premier problème on peut

répartir l'espionnage des données sur plusieurs périodes et ainsi réduire le nombre de communications à effectuer à chaque boucle. Si par exemple, une application contient 3 espions, on peut les exécuter à tour de rôle sur 3 boucles, on obtient ainsi une seule communication à chaque boucle. Pour résoudre le deuxième problème, un espionnage moins fréquent que le calcul permettra de réduire les besoins mémoires. L'inconvénient de cette solution est qu'elle génère une perte d'information due au sous-échantillonnage des données. C'est à l'utilisateur de vérifier que la fréquence des informations qu'il veut étudier n'est pas supérieure à la fréquence d'exécution de l'espion. Il devra donc dans certains cas trouver un équilibre entre la fréquence des espions, la durée de l'espionnage, le nombre d'espions, et les perturbations qu'il peut accepter sur l'exécution de son algorithme.

La figure 9.13(a) montre comment on peut implanter cette fonctionnalité dans un graphe flot de données SynDEx. Le principe s'inspire de celui retenu pour la restriction de l'espionnage sur un intervalle de temps. En fait, un sommêt `MultiR` produit un booléen *vrai* lorsque l'on veut espionner *faux* sinon. Cette opération exécutée sur l'horloge *horloge_interval* réalise un compteur modulo T (T étant la période d'espionnage exprimée en nombre de boucles temps-réelles). L'horloge *horloge_multir* ainsi produite est donc *vrai* à chaque remize à zéro de ce compteur. C'est maintenant *horloge_multir* qui doît être connecté à l'entrée du `hmul` à la place de *horloge_interval*. `SPYval` n'est exécuté que dans l'intervalle d'espionnage, que si la donnée à espionnée est calculée et que si le compteur modulo produit un booléen *vrai*. `SPYbool` enregistre toujours la sortie du hmul afin de connaitre les dates d'exécution de l'espion. Sur notre exemple 9.13(b) `MultiR` génere un booléen *vrai* toutes les 2 boucles temps-réelles. Par rapport à l'exemple précedent on remarque que toutes les valeurs d'espionnage correspondant à une valeur de la sortie de `multir` *faux* n'ont pas été enregistrées.

Quatrième partie

Application

Cette partie traîte de la conception d'un prototype de véhicule électrique "intelligent" baptisé CyCab. Le challenge posé par cette conception n'était pas de résoudre des problèmes de recherche en Automatique, mais d'apporter des solutions aux problèmes de la conception d'applications temps réel complexes soumises à de fortes contraintes industrielles faisant intervenir du contrôle-commande. Nous cherchions en effet à concevoir un prototype de véhicule industrialisable possédant des fonctionnalités déjà testées et développées sur des prototypes de laboratoire coûteux et utilisable en environnement urbain.

Habituellement, les architectures matérielles électroniques embarquées dans les prototypes de laboratoire ont une structure monoprocesseur ou multiprocesseur à mémoire partagée, dont la puissance de calcul et la taille mémoire qu'elles intègrent permettent de réaliser des algorithmes complexes sans souci d'optimisation de l'implantation. Pour programmer rapidement ce types d'architectures "standard", il existe de nombreux langages et outils d'implantations. Dans un premier temps, c'est généralement ce type d'architecture que les projets de recherche automobile utilisent pour développer leurs prototypes : ils ajoutent à un véhicule de série une architecture de type centralisée dont le calculateur est généralement composée d'un PC auquel peut être ajoutées des cartes DSP en fonction de la puissance de calcul recherchée et pour laquelle il existe des générateurs de code reliés à des outils de spécification et de simulation tels que Simulink. Dans un deuxième temps, lorsque la faisabilité du prototype a été démontrée et qu'il a été décidé de réaliser son industrialisation, une architecture matérielle faible coût est réalisée à base de microcontrôleurs et de bus de terrain. Actuellement, pour ce type d'architecture, il n'existe pas d'outils commerciaux permettant de couvrir toute la chaîne de conception depuis la spécification jusqu'à la génération de code. C'est donc à partir de la spécification Simulink réalisée pour le prototype que les ingénieurs produisent et optimisent à la main le logiciel de l'application.

Dans la mesure où nous cherchions à réaliser un prototype le plus proche possible du produit industriel, l'utilisation d'une architecture distribuée à microcontrôleur nous était imposée. Pour éviter une laborieuse phase de codage à la main des algorithmes, nous cherchions à utiliser une approche de conception générique valable quelque soit l'architecture utilisée. C'est pourquoi nous avons choisi la méthodologie AAA pour réaliser la spécification et l'implantation optimisée des lois de commande du CyCab. Nous espérions ainsi, d'une part, mener à bien et à moindre coût cette réalisation ambitieuse, et d'autre part, nous espérions retrouver la même facilité et fiabilité d'implantation que l'on peut généralement rencontrer sur les prototypes de laboratoire en vue de réduire au minimum les temps de développement. Enfin, cette réalisation permettait aussi de valider la méthodologie dans le cadre de l'implantation des algorithmes de contrôle-commande mis en jeux dans les systèmes temps réel embarqués, validation permettant alors de proposer cette méthodologie comme solution au problème de la conception des applications temps réel complexes soumises à de fortes contraintes industrielles.

Bien plus qu'une simple application, la réalisation de ce prototype a été la source d'inspirations et de réflexions qui a nourri les travaux effectués tout au long de cette thèse. La réalisation de ce prototype a été effectuée en parallèle avec les recherches sur la spécification et l'implantation présentées dans les trois premières parties de ce manuscrit. C'est pourquoi nous avons dû, pour réaliser le prototype, utiliser la méthodologie AAA telle qu'elle existait à l'époque, c'est-à-dire sans les améliorations proposées dans cette thèse.

Le contexte et les utilisations envisagées, présentées dans le premier chapitre de cette partie, nous ont permis de définir les applications que nous cherchions à réaliser. La conception de ces applications a d'abord consisté à définir un modèle du CyCab décrivant les aspects matériels (cap-

teurs et actionneurs) ainsi que les lois de commande. Bien que ces deux aspects aient été réalisés conjointement - les lois de commande ayant déterminées certains choix matériels et certains choix matériel ayant déterminés la structure et les paramètres de certaines lois de commande - nous avons choisi de présenter ces deux aspects séparément afin de faciliter la lecture de ce document. Nous décrivons d'abord, dans le chapitre 11, l'architecture matérielle commune à toutes les applications. Dans le chapitre suivant, nous présentons chacunes des applications réalisées sur le CyCab, c'est-à-dire leur but, les lois de commande mises en jeu et les spécificités matérielles qu'elles ont nécessité. Enfin, nous décrivons comment le logiciel SynDEx nous a permis de mener à bien les étapes de spécification et d'implantation des différentes applications du CyCab.

CHAPITRE 10

CONTEXTE DU PROJET

10.1 La voiture individuelle publique

La congestion du trafic automobile urbain et la pollution qu'il engendre poussent les grandes agglomérations à définir des stratégies visant à dissuader l'usage des véhicules personnels. Les moyens mis en œuvre sont principalement des mesures tarifaires (péages, horodateurs, etc.), des réglementations (limitations de la vitesse les jours de pics de pollution, autorisation de circulation alternée en fonction du numéro d'immatriculation, etc.), ainsi que le développement de zones piétonnières et de pistes cyclables. En parallèle, le transport public est actuellement en train de se renouveler et de se donner une image de modernité respectueuse de l'écologie et de la qualité de l'environnement urbain. Économiquement, le transport public se justifie parfaitement dès lors qu'une demande importante de déplacement existe sur un certain axe et sur certaines tranches horaires. Dans ce cas, le transport en commun peut apporter un service de qualité grâce à des fréquences élevées, tout en étant particulièrement économe en termes d'utilisation de l'espace par rapport aux débits possibles. En revanche, dès que la demande de transport public descend au-dessous d'un certain seuil, les gains indirects du transport collectif ne compensent plus les dépenses et ce service doit être abandonné.

Dans ce contexte, le concept de voitures individuelles publiques est proposé comme le chaînon manquant de l'offre de transport, entre les solutions actuelles de transport individuel et collectif. Des expériences ont été tentées par le passé sans beaucoup de succès. Cependant, les progrès effectués en matière d'automatisation des transports et d'électrification des véhicules permettent d'espérer de nouveaux développements de ce concept. Le principe commun de ces projets est de disposer d'un très grand nombre de petits véhicules publics à usage individuel (deux à quatre places) en fonctionnement normal sur la voirie traditionnelle et, selon les cas, dans un futur plus éloigné, en fonctionnement automatique sur un réseau propre. L'intérêt économique et écologique de ces systèmes repose sur une combinaison d'avantages : mode de propulsion électrique, petite taille, densité de parking (il n'est besoin que d'avoir accès à la première voiture d'une file) et aussi la réutilisation d'un même véhicule en n'employant qu'une main-d'oeuvre réduite.

10.2 Le programme Praxitèle

En 1993, un accord de collaboration a été signé entre la CGFTE [1], Renault, EDF, Dassault Automatismes et Télécommunications, l'INRETS [2] et l'INRIA pour lancer un programme baptisé PRAXITÈLE[65]. L'objectif de ce nouveau projet était d'arriver rapidement à une expérimentation grandeur nature pour valider l'intérêt du système décrit précédemment auprès des utilisateurs mais aussi de préparer l'avenir en développant les technologies d'analyse des futurs systèmes et en lançant des recherches sur leur évolution technologique, en particulier pour préparer les futures versions qui devraient être plus performantes et apporter de meilleurs services. Les développements techniques concernent l'ensemble des points organisationnels et reglementaires qu'il est nécessaire d'étudier pour arriver à des expérimentations en vraie grandeur du système de transport urbain public individuel dans ses différentes versions. Ces versions sont définies dans [66], chaque version inclut les fonctionnalités de la version précédente :

- **Version 1 :** les véhicules sont en conduite manuelle par les usagers ou éventuellement par des conducteurs professionnels (fonction taxi), occasionnels ou permanents. Le retour à vide des véhicules est assuré par du personnel qui déplace les véhicules sur des remorques ou en formant des trains de voitures avec accrochage mécanique. Chaque véhicule est localisé par le système de gestion qui commande les déplacements à vide en fonction de la demande client anticipée. L'accès, le paiement et la réservation se font par carte intelligente. Les voitures sont garées dans des parkings réservés où elles sont rechargées automatiquement. Des parkings automatiques peuvent être utilisés. Le co-voiturage peut éventuellement être instauré pour réduire la congestion,
- **Version 2 :** les véhicules peuvent se déplacer en pelotons avec accrochage électronique. La voiture de tête du peloton est conduite manuellement. Les voitures suiveuses sont en conduite automatique. Les pelotons utilisent de préférence des voies propres ou des couloirs réservés. La rentrée et la sortie d'un peloton se fait en automatique dans des stations équipées pour cela et qui font office de parking. Cette version apporte des débits importants et une automatisation de la conduite pendant les périodes de pointe sur certaines voies. L'accrochage automatique immatériel simplifie la formation des trains pour le retour à vide des véhicules. La mise en parking est assurée automatiquement et enclenche la recharge des batteries. Le déplacement dans les grands parkings est automatisé,
- **Version 3 :** les véhicules se déplacent en automatique individuellement ou en pelotons sur des sites propres protégés et équipés. L'entrée et la sortie des sites propres sont effectuées dans des stations qui font aussi office de parking et qui peuvent s'échanger des véhicules vides sans intervention humaine sur commande du système de gestion. Cette version résout le problème du déplacement automatique des véhicules vides entre les parkings qui sont sur le site propre (il peut y avoir d'autres parkings),
- **Version 4 :** Les véhicules peuvent se déplacer automatiquement (à vitesse réduite) sur la voie publique légèrement aménagée. Les usagers peuvent choisir une conduite manuelle ou une conduite automatique sur ces sites. Les voitures peuvent être livrées à domicile et rapportées aux parkings après une course quelconque. Cette version très futuriste n'est envisageable que dans le cadre d'une modification profonde du partage de la voirie avec les véhicules privés.

1. filiale transports publics de la Compagnie Générale des Eaux
2. Institut National de recherche sur les transports et leur sécurité

10.3 Le CyCab

Dans les grandes agglomérations, la sécurité physique des citadins, la diminution des pollutions atmosphériques et sonores, ou encore la préservation du cadre de vie et des sites touristiques sont autant de préoccupations qui conduisent à la création et à l'extension de zones piétonnes. Dans ces espaces urbains, les déplacements pédestres est la règle générale, ce qui pose des difficultés pour certains usagers (les hommes d'affaires dont le temps est compté, les personnes trop chargées ou à mobilité réduite, etc...) et dans certaines circonstances (pluie, etc...).

FIG. 10.1 – *Des véhicules en libre service*

L'objectif est de favoriser le développement des zones interdites aux voitures particulières en proposant une alternative aux déplacements pédestres dans ces espaces (Fig. 10.1). S'appuyant sur les dernières innovations technologiques développées dans le cadre du projet Praxitèle, l'INRIA est entièrement à l'origine de la conception des CyCab, très petits véhicules urbains d'une longueur égale à celle d'un vélo (Fig. 10.2) destinés à une utilisation en libre-service.

FIG. 10.2 – *CyCab : un véhicule de la taille d'un vélo*

Les CyCab seront implantés en flotte (Fig. 10.3) dans des zones délimitées, partout où il est nécessaire de circuler ponctuellement et librement sur de courtes distances. Ce système vient compléter les transports en commun classiques dans des lieux où ces derniers seraient inadéquats en raison de leur mise en œuvre difficile ou de leur faible rentabilité. Néanmoins, ils n'autorisent pas

de gros débits de voyageurs, mais permettent, par exemple, d'effectuer le trajet terminal entre la station de transport en commun et la destination finale.

FIG. 10.3 – *Une flotte de véhicules en libre service*

10.4 Utilisations envisagées

La configuration du véhicule CyCab pourra être adaptée en fonction des spécificités de chaque site. Parmi les applications possibles du nouveau système, on peut citer l'aménagement d'espaces tels que :

- les pôles d'activité (exemple: le quartier de La Défense), les technopôles ;
- les sites privés industriels ou de recherche ;
- les parcs et les sites touristiques, notamment les lieux fréquentés par les visiteurs étrangers disposant de peu de temps. Dans ce cas, le CyCab sert à la fois de moyen de transport et de guide ;
- les zones piétonnes en centre ville, aménagées pour le commerce ou le tourisme ;
- les parcs d'exposition, les parcs animaliers, les parcs naturels ;
- les centres hospitaliers ou de convalescence, les stations thermales, les stations balnéaires ;
- les campus universitaires ;
- les zones de loisirs.

Pour satisfaire aux besoins de ces différentes utilisations (souplesse d'utilisation, sécurité, ergonomie, etc.) le CyCab offre de nouvelles fonctionnalités telles que la conduite sécurisée, la conduite automatique et radioguidée que nous présentons dans le chapitre 12.

CHAPITRE 11

ARCHITECTURE MATÉRIELLE DU CYCAB

Le concept du CyCab a été imaginé par l'INRIA Rocquencourt. La mécanique a été réalisée par la société Andruet selon un cahier des charges défini par l'INRIA Rocquencourt. L'électronique a été conçue et réalisée par l'INRIA Rhônes-Alpes en collaboration avec l'INRIA Rocquencourt. Pour pouvoir définir les lois de commande des différentes applications envisagées, il a fallu réunir et comprendre les connaissances de chaque équipe engagée dans cette réalisation et ainsi définir les caractéristiques mécaniques et électroniques du véhicule. De même, une connaissance approfondie de l'architecture informatique (interfaces électroniques, microcontrôleurs, mapping mémoire, etc.) est nécessaire pour réaliser l'implantation des algorithmes sur l'architecture. Nous présentons ici une synthèse de ces caractéristiques matérielles.

Le CyCab a été conçu en tenant compte de contraintes d'utilisation publique et de production en série : coût réduit, faibles dimensions, robustesse, maintenance facile. Toute la conception a été orientée dans ce sens depuis la mécanique jusqu'au système informatique. Il est composé d'un châssis métallique sur lequel est fixé une coque en matériau composite. C'est un véhicule électrique disposant de 4 roues motrices et directrices pilotées par le système informatique embarqué. La figure 11.1 présente les principaux éléments qui permettent de le piloter en manuel (joystick et terminal informatique) ou en automatique (caméra et cible infrarouge, capteurs à ultrasons etc.).

11.1 Mécanique

11.1.1 Coque

La coque a été conçue pour accueillir deux passagers ainsi que leurs bagages (approximativement le contenu d'un caddie). Elle est constituée de deux parties moulées dans un matériau composite peint qui à l'avantage d'être léger, robuste, facile à produire en série et à entretenir.

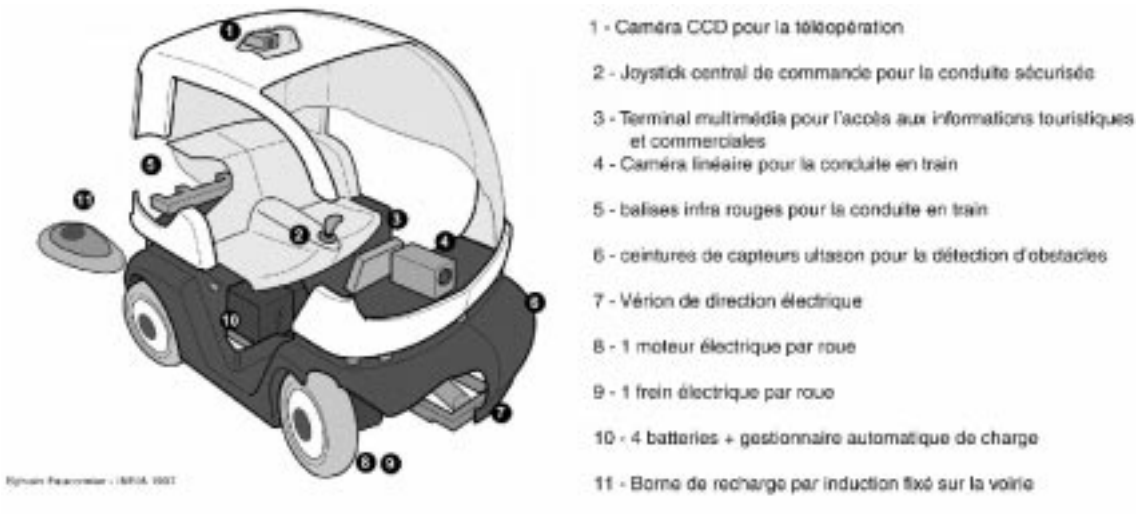


FIG. 11.1 – *Véhicule électrique semi-autonome*

11.1.2 Châssis

La mécanique du CyCab est dérivée d'un châssis tubulaire de voitures de golf électriques produites en petites séries et commercialisées par la société Andruet SA. Il supporte les 4 batteries de 12 Volts (qui lui donnent en théorie une autonomie de 2 heures à une vitesse maximale de 20 Km/h sur des pentes n'excédant pas 10%), les 4 blocs roue (roue + moteur + frein mécanique à tambour), les 4 moteurs de frein de parking ainsi que le vérin électrique actionnant la mécanique de direction.

11.1.2.1 Motorisation

La motorisation du CyCab est réalisée par les 4 blocs roue qui sont identiques. Cette solution a permis de réduire les coûts et le volume : 4 petits moteurs 48V à courant continu pilotés par des petits contrôleurs de puissance sont plus facilement intégrables dans un espace réduit qu'un seul gros moteur piloté par un contrôleur de forte puissance. Les blocs sont interchangeables et produits en plus grande quantité, la maintenance en est facilitée.

11.1.2.2 Direction

Les 4 roues du CyCab sont directrices, mais elles ne sont pas indépendantes. Elles sont reliées par une tringlerie (Fig. 11.2) actionnée par le vérin électrique.

La figure 11.3 représente la liaison entre le vérin de direction et la tringlerie. Les relations métriques dans le triangle permettent de déterminer l'angle donné aux roues en fonction de la longueur du vérin :

$$S = \sqrt{d_1^2 + d_2^2 - 2 d_1 d_2 \sin(\alpha)}$$

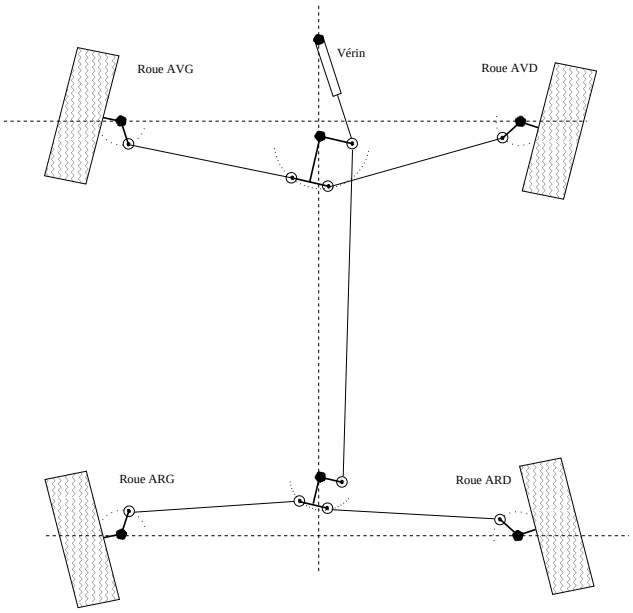


FIG. 11.2 – *Mécanique de direction*

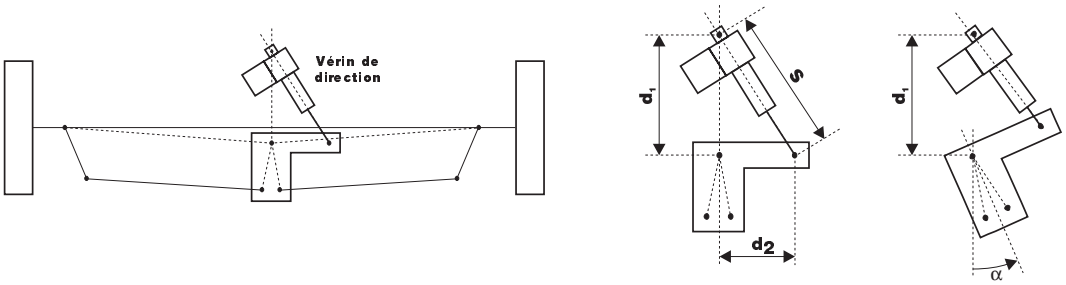


FIG. 11.3 – *Commande de la tringlerie de direction par vérin*

11.1.2.3 Freinage

Chaque bloc roue est doté d'un frein à tambour mécanique commandé par câble. Un moteur de type essuie-glace de voiture tire ou relâche le câble. Sur le CyCab de Rocquencourt, la commande de ces moteurs est réalisée par l'entrée tout ou rien d'un boîtier électronique.

11.2 Architecture électronique embarquée

L'électronique embarquée dans le CyCab est composée principalement d'un boîtier d'alimentation et de protection qui produit différentes tensions (12, 24 et 48 Volts) à partir de la tension fournie par les 4 batteries 12 Volts, d'un boîtier de commande des 4 moteurs de frein et d'un calculateur distribué composé d'un PC et de 5 boîtiers baptisés "nœud" (N_i) chacun dédié à la commande d'un moteur de direction et d'un moteur de frein ou du vérin de direction (Fig. 11.4).

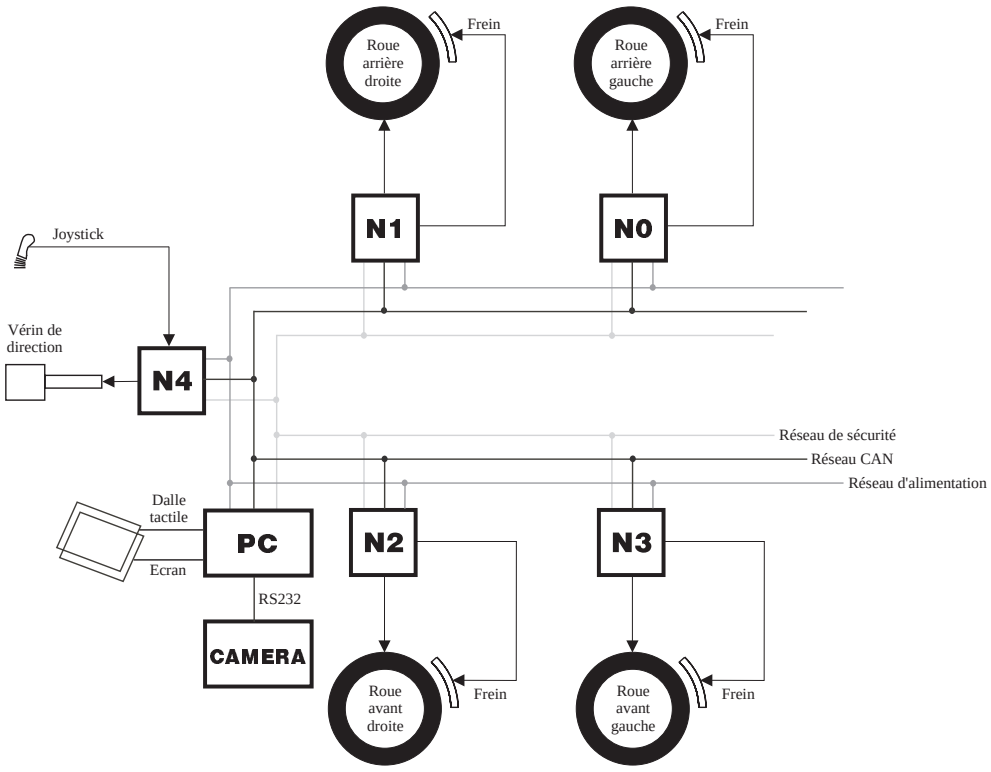


FIG. 11.4 – *Architecture du calculateur embarqué*

11.2.1 Processeurs

Dans sa version de base le calculateur embarqué est composé de 6 processeurs :

- **1x Intel 486DXII 66Mhz :** le PC embarqué est architecturé autour d'un microprocesseur 486DXII cadencé à 66Mhz disposant de 4Mo de mémoire RAM. Son rôle principal est de gérer un écran LCD couleur auquel est adjointe une dalle tactile, ainsi que de fournir l'accès à une liaison Ethernet et à un disque dur pour faciliter le chargement des programmes et la mise au point du prototype.
- **5x Micro-contrôleur MC68332 20Mhz :** Chaque nœud intègre un micro-contrôleur MC68332 de la famille Motorola, chargé de gérer les capteurs et actionneurs nécessaires à la commande du véhicule.

Dans sa version la plus complexe, un micro-contrôleur Intel 80KC196 intégré dans une caméra linéaire infrarouge est rajouté à l'architecture. Des capteurs ultrasons intégrant des Motorola 68HC11 sont en cours de développement.

11.2.2 Médias de communication

Le calculateur intègre principalement 2 types de médias de communication. Une liaison point à point SAM de type RS232, une liaison multipoint SAM de type CAN. La liaison RS232 est utilisée pour faire communiquer le 80C196 de la caméra et le 486DXII66 du PC. Le bus CAN relie tous les nœuds et le PC.

Le CAN[16] est un bus série bi-filaire qui a été conçu spécialement pour les applications automobiles afin d'assurer des communications fiables en environnement perturbé avec un débit de transmission pouvant atteindre 1Mbits/s. Son protocole de communication est basé sur une structure multi-maître où l'arbitrage orienté priorité est basé sur un état récessif et un état dominant du bus ; l'état haut du bus est récessif c'est-à-dire que l'émission simultanée sur le bus d'un état logique haut et d'un état logique bas par deux émetteurs conduit à un état logique bas effectif sur le bus. Pour réaliser l'arbitrage, chaque émetteur est aussi récepteur du message qu'il envoie. Il peut ainsi comparer pendant l'émission, l'état du bus avec le message qu'il émet. Lorsqu'il émet un bit recessif et qu'un bit dominant est présent sur le bus, c'est qu'un autre émetteur est en train de transmettre un bit dominant, il perd l'arbitrage du bus et réémet son message ultérieurement. Ce sont les identificateurs des messages qui sont les premiers bits émis qui permettent d'arbitrer le bus. La position des bits dominants et récessifs dans l'identificateur détermineront la priorité du message (priorité inversement proportionnelle à la valeur de l'identificateur). Les messages que le bus CAN véhicule (trames) sont constitués d'au maximum 8 octets de données et d'environ 8 octets de contrôle d'erreur et d'identification. Le surcoût induit par ces bits de contrôle réduit donc d'au moins 50% la bande passante utile.

11.2.3 Capteurs

A chaque bloc roue est associé un capteur de température du moteur de locomotion, d'un codeur optique incrémental 2048 tops/tr qui donne la position angulaire du rotor et d'un capteur de courant d'induit du moteur de locomotion et de 2 capteurs fin de course sur le moteur de frein.

Le nœud de direction gère un codeur optique incrémental 2048 tops/tr relié au moteur électrique du vérin, un capteur de déplacement rectiligne du vérin (potentiomètre linéaire) et 2 capteurs fin de course.

Enfin, divers capteurs spécifiques à chacune des applications ont été intégrés à l'architecture du CyCab : joystick, dalle tactile, gyromètre, caméra linéaire infrarouge, capteur à effet hall, capteurs à ultrasons.

11.2.4 Actionneurs

Les actionneurs du CyCab sont les 4 moteurs de traction 48V à courant continu 1kW, les 4 moteurs de type essuie-glace de voiture qui tirent les câbles des freins à tambour, le vérin électrique qui pilote la direction et l'écran LCD qui permet d'afficher des informations au conducteur.

11.3 Architecture d'un noeud

La commande en puissance de chaque moteur de traction et du vérin de direction est réalisée par le *noeud* qui lui est associé. Ces noeuds intelligents, tous identiques, sont constitués de trois modules électroniques couplés entre eux et logés dans un même boîtier (Fig. 11.5). Chaque nœud de roue contrôle un moteur de locomotion et un moteur de frein avec tous leurs capteurs (codeur incrémental, capteur de température, etc.). Le cinquième nœud gère le vérin de direction et le joystick.

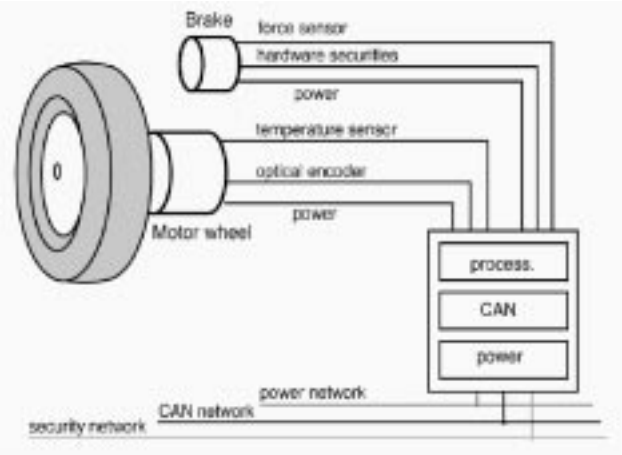


FIG. 11.5 – *architecture d'un "nœud"*

Un nœud est composé de trois couches (ou modules). La plus basse fournit la puissance aux moteurs. La deuxième réalise les communications avec les autres nœuds et l'acquisition des capteurs. La dernière, la couche calcul, supporte un micro-contrôleur MC68332 (famille 68000 de Motorola).

11.3.1 Module de calcul

Le module de calcul est conçu sur le modèle d'une carte Motorola BCC32. Elle supporte un micro-contrôleur Motorola MC68332, une EPROM 64k x 16bits contenant un moniteur-déboggeur (CPU32bug), une mémoire RAM 32k x 16bits, une interface RS232 et une mémoire FLASH bootable de 4Mbits (fig.11.6)

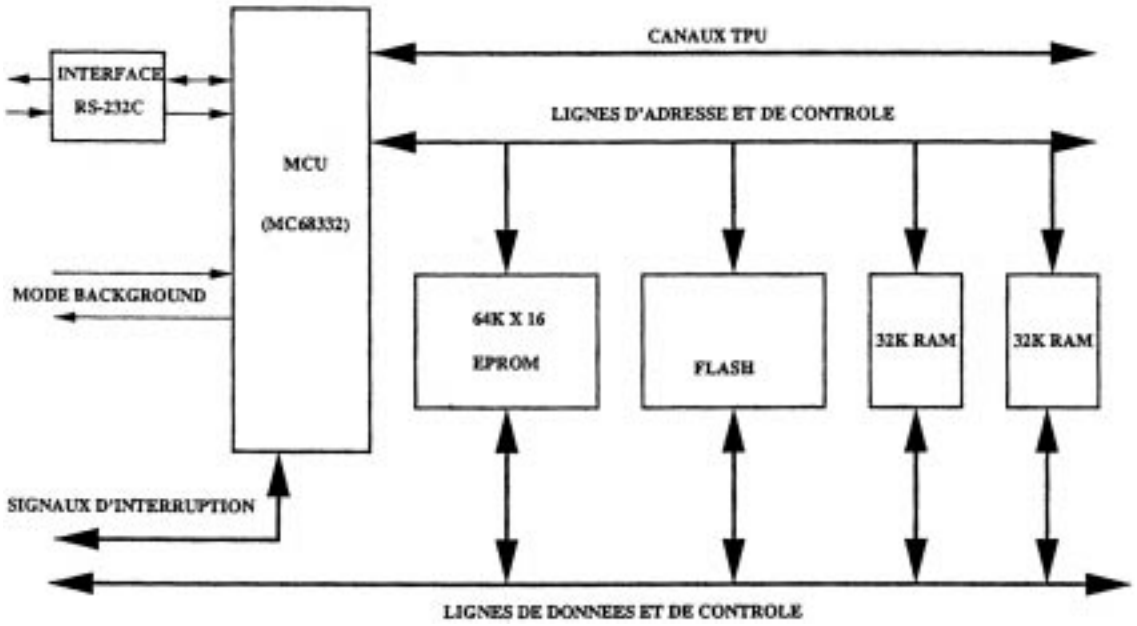


FIG. 11.6 – *Synoptique du module de calcul*

L'architecture interne du MC68332 est composée de cinq modules : une unité centrale de calcul (CPU), une unité centrale de calcul temporel (TPU), un module série avec gestion de FIFO (QSM), une RAM statique et un module d'intégration système :

- **CPU :** Le CPU intégré dans le MC68332 se situe dans la gamme des microprocesseurs 68xxx. Il dispose en interne de registres accessibles en mode utilisateur (8 registres de données 32bits D0-D7, 8 registres d'adresse 32bits A0-A6, un pointeur de pile 32bits A7, 1 compteur programme 32bits) ou en mode super-utilisateur (1 registre de pointeur de pile 32bits SSP, 1 registre de status 16bits incluant le CCR 8bits). Le jeu d'instructions et les modes d'adressages sont compatibles avec ceux du 68000, et pour la plupart avec ceux du 68020,

- **TPU :** Le TPU est un micro-contrôleur semi-autonome dédié aux opérations de contrôle temporel, de capture et de comparaison. Il opère en parallèle avec le CPU et est capable d'ordonnancer des tâches, d'effectuer des instructions écrites en ROM, d'accéder à des données partagées avec le CPU et de réaliser des entrées-sorties. Il fournit 2 bases de temps 16bits, 16 canaux timer indépendants, un scheduler de tâches et intègre en ROM un ensemble de fonctions de capture et de comparaison de signaux : capture d'entrée/compteur de transition d'entrées, comparaison d'entrées-sorties, PWM[1], mesure de période avec détection de transition, contrôle de moteur pas à pas, générateur d'impulsions, etc. L'utilisateur a la possibilité

1. Modulation à largeur d'impulsions

de charger en RAM d'autres fonctions qu'il aura soit lui-même programmé (il existe un compilateur permettant de générer du code pour le TPU) soit acheté,

- **Module série QSM :** Il est constitué de 2 interfaces (QSPI et SCI) de communication série. Le QSPI est une interface série full-duplex offrant la possibilité de communiquer avec des périphériques où des microprocesseurs sélectionnables parmi un ensemble de 16 adresses. Il dispose d'un DMA de 256bits autorisant 16 transferts série de 8bits, 8 transferts 16bits ou une transmission de 256bits sans intervention CPU. Le SPI est une interface série full ou half-duplex sans DMA proche d'une interface RS232,

- **Module d'intégration système :** Il est chargé de la configuration et de la protection du 68332. Il contient principalement tous les registres de configuration de tous les modules soit 4kbits logeables en $7FF000-$7FFFFF ou en$FFF000-$FFFFFF, un arbitre d'interruptions internes et externes, un timer d'interruption périodique programmable, une interface de contrôle du bus externe pour l'adressage aux périphériques et à la mémoire externe, ainsi qu'un décodeur d'adresse offrant la possibilité de définir sans adjonction de circuit périphérique, le mapping mémoire (chip-select) et le type de transfert entre le CPU et les périphériques (async/sync, R/W, wait state, etc.),

- **Module RAM statique :** Cette RAM statique permet le stockage et l'accès rapide à 2Ko de données. Elle est utilisable aussi bien par le CPU que par le TPU pour lequel il sera possible d'y stocker de nouvelles fonctions non proposées en standard dans sa mémoire ROM.

11.3.2 Module interface entrées-sorties

Ce module est constitué, mis à part les sorties puissance, de toute l'électronique nécessaire à la gestion des entrées/sorties capteurs et actionneurs. Elle réalise la mise en forme et la protection des entrées-sorties du 68332 (isolation galvanique par opto-couplage) ainsi que les convertions analogiques numériques à l'aide d'un convertisseur 10bits à 8 canaux d'entrées. Un circuit logique programmable de type EPLD[2] génère les signaux logiques de contrôle des composants et notamment la logique de décodage d'inhibition ou d'autorisation de la commande en puissance des moteurs en fonction de la valeur de signaux de dysfonctionnement éventuellement émis par les autres nœuds (arrêt d'urgence cablé). Ce module intègre aussi une interface CAN construite autour du composant Philips 82C200. Comme l'interface RS232, cette interface dispose d'une FIFO qui ne permet pas de mémoriser plus d'un message, l'arrivée d'un nouveau message écrase la valeur du dernier message reçu.

11.3.3 Module de puissance

Le module de puissance est composé de deux amplificateurs de puissance. L'architecture de ces amplificateurs est de type pont en H utilisant des transistors de puissance MOS-FET pour la commande de moteurs à courant continu. La commande du pont en H est réalisée par des transistors pilotés par une entrée logique : à chaque état logique du signal qui est appliqué à cette entrée, correspond une polarité pour le courant traversant le moteur. Ainsi par une commande en PWM de cette entrée, consistant à générer un signal périodique de période T et dont l'intervalle de temps

2. Erasable programmable logical Device

t^+_{pwm} pour lequel il est à l'état logique haut est variable, on peut contrôler la tension moyenne du courant qui traverse le moteur et donc le sens et la vitesse de rotation du rotor. Sur l'intervalle de temps T le moteur est donc alimenté par une tension $U_{batterie}$ (48V) pendant t^+_{pwm} et par une tension $-U_{batterie}$ pendant $T - t^+_{pwm}$ ($t^+_{pwm} \leq T$), soit une tension moyenne $U - moy$ sur T :

$$\begin{aligned} U_{moy} &= \frac{1}{T}\int_0^T u(t)\,dt \\ &= \frac{1}{T}\left(\int_0^{t^+_{pwm}} U_{batterie}\,dt + \int_{t^+_{pwm}}^{T} -U_{batterie}\,dt\right) \\ &= \frac{U_{batterie}(2t^+_{pwm} - T)}{T} \end{aligned}$$

Ainsi, la tension moyenne sur un intervalle de temps T aux bornes du moteur est $-U_{batterie}$ pour $t^+_{pwm} = 0$, $+U_{batterie}$ pour $t^+_{pwm} = T$ et 0 pour $t^+_{pwm} = \frac{T}{2}$.

Les entrées de commande PWM des deux amplificateurs sont reliées, par l'intermédiaire de l'EPLD de la carte d'entrée-sortie, à des broches de sortie TPU du 68332 pilotées par une fonction PWM en ROM du TPU. Ainsi l'ensemble TPU + amplificateur de puissance peut être assimilé à un hacheur commandé en PWM : pour appliquer une tension aux bornes du moteur, il suffit de positionner la valeur t^+_{pwm} dans un registre du TPU. Un signal d'inhibition de la puissance permet de bloquer l'ensemble des transitors du pont en H afin d'isoler le moteur de l'alimentation et donc de forcer un courant nul dans le moteur quelle que soit la valeur du rapport cyclique du signal PWM. Cette fonctionnalité est utile notamment dans les phases d'initialisation du nœud, (on ne connait pas a priori l'état de la sortie TPU), en cas d'arrêt d'urgence où pour économiser l'énergie en cas d'arrêt prolongé du véhicule.

11.4 Remarques

– Il existe actuellement (janvier 2000) deux versions du Cycab avec quelques petites différences au niveau de l'électronique. Un rapport de recherche [10] décrit de manière très détaillée l'électronique du CyCab de l'INRIA Rhônes-Alpes. À quelques petites différences près on retrouve la même électronique dans le CyCab Rocquencourt sur lequel a été réalisé les applications présentées dans le chapitre suivant.

– L'architecture retenue pour le CyCab est composée d'un ensemble hétérogène de processeurs reliés par un bus de communication. Les capteurs et actionneurs sont reliés aux processeurs. Nous avons qualifié ce type d'architecture de "faiblement distribué" (cf. §3.2.3). Nous rappelons que son intérêt réside dans le rapprochement des capteurs et actionneurs des processeurs ainsi que dans la factorisation des capteurs afin de réduire les coûts (câblage, nombre de capteurs, perturbations électromagnétiques etc.)

– Pour la programmation de cette architecture, une attention particulière doit être portée à l'optimisation des algorithmes de commande nécessitant des calculs complexes car le 68332 ne possède pas de FPU (il faut donc autant que possible éviter de réaliser des calculs sur des

nombres à virgules flottantes, construire des tables pour le calcul des fonctions trigonométriques, etc.). Il faut aussi que les primitives de gestion de l'interface CAN et de la RS232 soient optimisées car, pour éviter la perte de messages avec de hauts débits de transmission, le processeur devra très fréquemment lire le registre dans lequel la donnée est reçue (pas de FIFO) et devra aussi fréquemment mettre à jour le registre d'émission des composants qui gèrent ces interfaces.

CHAPITRE 12

DESCRIPTION DES APPLICATIONS

Lors de la modélisation, l'automaticien décrit les lois de commande qui doivent permettre au système de contrôle-commande d'évoluer selon un but défini par l'application. Sur le CyCab de Rocquencourt quatre applications différentes ont été réalisées : "conduite manuelle sécurisée", "suivi de véhicule", "téléopération" et "localisation". Ce chapitre décrit pour chacune de ces applications, les principes et les lois de commande définies dans l'étape de modélisation.

12.1 Conduite manuelle sécurisée

La conduite manuelle sécurisée est l'application la plus simple qui a servi de base pour construire les trois autres applications. Le but de celle-ci est d'asservir le cap et la vitesse du CyCab aux consignes du conducteur. Elle correspond à une conduite manuelle d'un véhicule classique à ceci près qu'elle est réalisée à l'aide d'un joystick plutôt qu'à l'aide d'un volant et de pédales et qu'il n'y a pas de liaison mécanique entre le joystick et la direction et entre le joystick et les moteurs de traction. Il est a noter que la suppression de cette liaison mécanique (X-by-wire) rejoint la grande préocupation des concepteurs automobile déjà mise en œuvre dans les applications d'avionique. Le CyCab est donc entièrement piloté par le calculateur embarqué qui calcule les tensions à appliquer aux moteurs de traction, de commande de direction et de commande de frein en fonction de la position du joystick. La position latérale du joystick détermine la direction du véhicule, la position longitudinale détermine sa vitesse. Lorsque le joystick est amené en position longitudinale centrale, le véhicule freine, avec une décélération proportionnelle à la vitesse (freinage doux). Un freinage plus soutenu est obtenu en positionnant le joystick dans la direction opposée au sens de déplacement. En cas de problème, un arrêt quasi immédiat du véhicule est obtenu par appui sur un bouton "coup de poing" (arrêt d'urgence). Celui-ci déclenche les freins mécaniques et coupe la puissance des moteurs. Cette conduite manuelle est dite "sécurisée" car il est possible de limiter certains paramètres de l'évolution dynamique du véhicule (vitesse maximale, accélération latérale, angle de braquage, etc.).

12.1.1 Loi de commande longitudinale

Les lois de commande en PWM des moteurs ont été définies à partir du modèle simplifié de moteur à courant continu suivant :

$$\begin{cases} C_m = k_c I \\ U = E + (R + Lp)I \\ E = k_e \Omega \end{cases}$$

où C_m est le couple moteur,
I et U le courant et la tension d'induit,
k_c et k_e les constantes de couple et de f.e.m,
E la force électromotrice,
R et L la résistance et l'inductance,
Ω la vitesse de rotation.

La figure 12.1 représente, sous la forme d'un schéma-bloc, le principe d'une loi de commande d'un moteur de traction du CyCab. À partir de la consigne C_{cons} (proportionnelle à la position longitudinale du joystick), on calcule une consigne en courant I_{cons} qu'on sature afin de limiter le courant dans l'induit. La mesure de la position angulaire α_{mes}, donnée par le capteur odométrique relié à l'axe du moteur permet de calculer (par dérivation) la vitesse de rotation Ω_{calc} . Après multiplication par la constante de couple k_c, on obtient la force électromotrice E_{calc} qui nous permet de calculer la commande en tension $U_{cmd} = RI_{cons} + E_{calc}$ (on néglige ici l'inductance L qui est souvent très faible). Enfin, U_{cmd} est transformée en la valeur t^+_{pwm} prise en consigne de la fonction PWM du TPU (cf. §11.3.3).

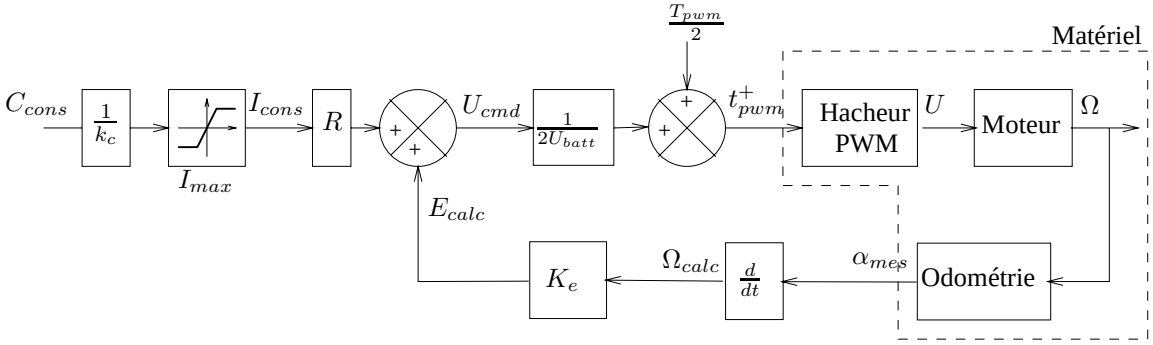


FIG. 12.1 – *Loi de commande d'un moteur de traction*

Pour prendre en compte la nature distribuée (4 moteurs) de la motorisation du CyCab, cette loi de commande doit être découpée en une première partie de calcul de la consigne en courant I_{cons} et en une deuxième partie, dupliquée pour chacun des moteurs, de calcul de la commande PWM t^+_{pwm}(fig. 12.2).

Le calcul de la consigne en courant doit permettre l'obtention d'un freinage doux lorsque le joystick est relaché. C'est pourquoi la fonction qui relie la position longitudinale du joystick Joy_{tra} n'est pas linéaire. Tout d'abord, un seuillage sur la position est nécessaire pour délimiter une petite plage centrale pour laquelle le joystick est considéré centré longitudinalement ce qui permet

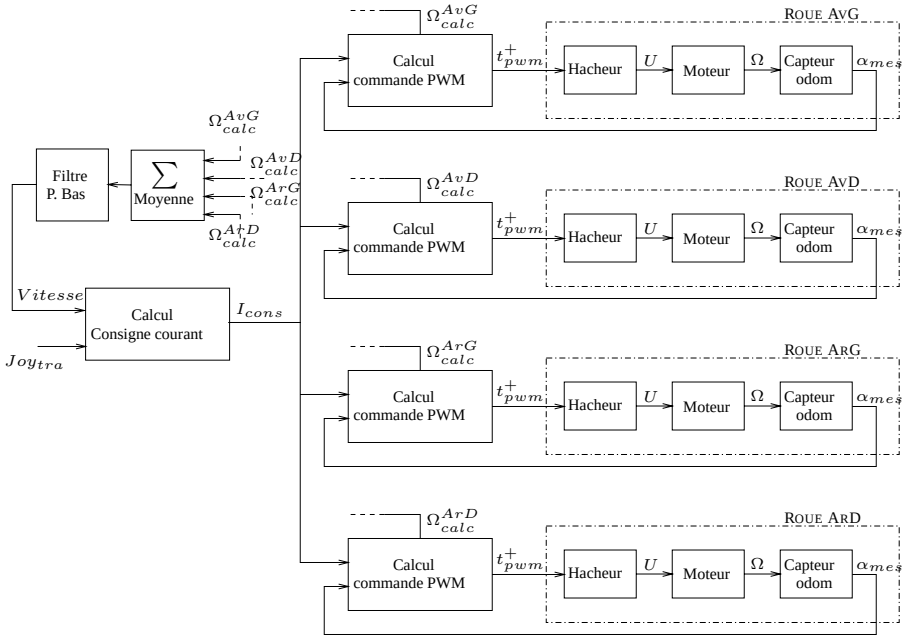


FIG. 12.2 – *Structure générale de la commande de traction du CyCab*

d'obtenir un "zéro" franc. Deux cas peuvent alors se présenter :

- **le joystick est positionné dans la plage centrale** : on calcule alors une consigne en courant proportionnelle ($-K_1$) à la vitesse du véhicule et saturée à $Ifrein_{max}$ afin de réaliser un freinage progressif ;
- **le joystick n'est pas positionné dans la plage centrale** : on calcule une consigne en courant proportionnelle (K_2) à l'écart entre la position du joystick et la plage centrale et saturée à I_{max}.

Pour adoucir la conduite, la consigne en courant est ensuite filtrée. La figure 12.3 représente la chaîne complète de calcul de la consigne en courant I_{cmd}.

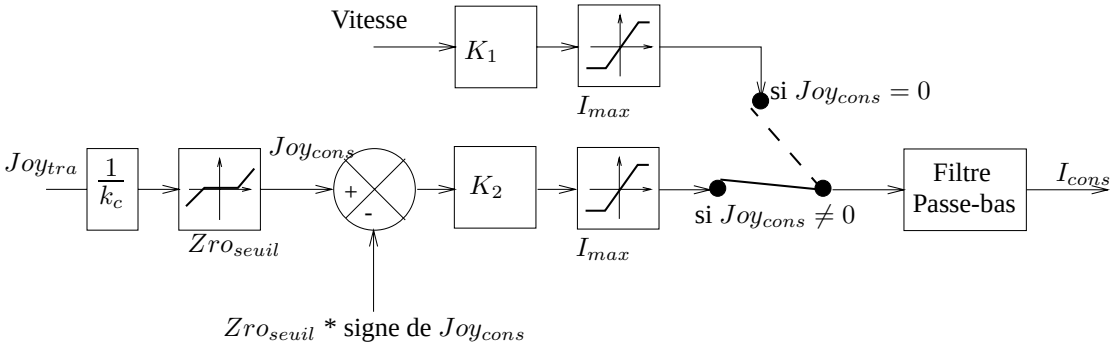


FIG. 12.3 – *Loi de commande d'un moteur de traction*

12.1.2 Loi de commande latérale

La loi de commande latérale réalise un asservissement de position. La consigne d'entrée est donnée par la mesure de la position latérale du joystick Joy_{dir} représentant la position angulaire des roues α_{dir} par rapport à l'axe longitudinal du véhicule désirée par le conducteur. La commande de sortie est la t^+_{pwm} d'entrée du hacheur qui pilote le moteur à courant continu du vérin. Le calcul de t^+_{pwm} à partir d'une consigne en courant I_{cons} est réalisé par une loi de commande identique (à quelques valeurs de gain et de seuil près) à celle présentée figure 12.1 pour la commande des moteurs de traction.

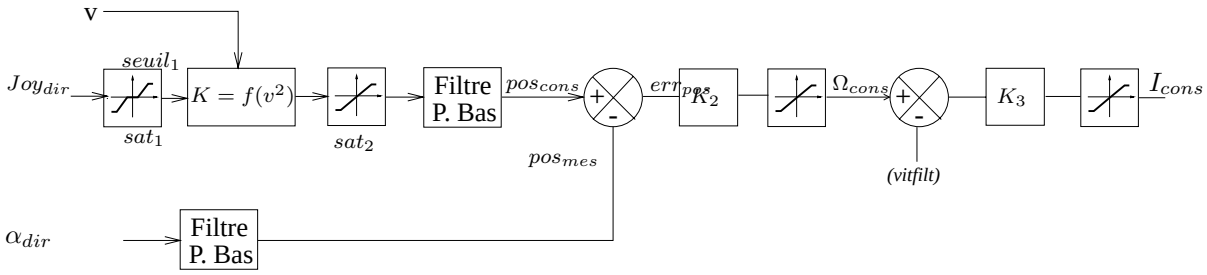


FIG. 12.4 – *Structure générale de la commande de traction du CyCab*

La figure 12.4 représente la loi de commande qui calcule I_{cons} à partir de Joy_{dir}. Une erreur de position err_{pos} est calculée par différence de la position mesurée pos_{mes} par le capteur de position absolue du vérin (potentiomètre) et de la consigne du conducteur Joy_{dir}. On remarquera que Joy_{dir} est préalablement multipliée par un facteur variant avec la vitesse *v* du véhicule afin de limiter l'accélération latérale. Le seuillage sur Joy_{dir} permet d'obtenir un "zéro franc" sur la position centrale du joystick. L'erreur de position err_{pos} est transformée en consigne de vitesse Ω_{cons} par multiplication par un facteur K_2. À partir de cette consigne Ω_{cons} et de la vitesse de déplacement du moteur $vitfiltr$, on calcule une erreur de vitesse err_Ω. Celle-ci est multipliée par un gain K_3 pour obtenir la consigne en courant I_{cons}. La vitesse de déplacement $vitfiltr$ est calculée dans l'asservissement en PWM. On rappelle que cette vitesse est obtenue par dérivation de la position angulaire du rotor donnée par le capteur odométrique. Les saturations jouent le rôle de "butées logicielles", dont le rôle est de limiter le courant dans le vérin électrique ainsi que sa course.

12.2 Suivi de véhicule

Le but de l'application de suivi de véhicule est de réaliser des trains de véhicules avec accrochage immatériel : le véhicule de tête est conduit manuellement, les autres véhicules suivent automatiquement. Cette application a été imaginée à l'origine dans le cadre de la voiture publique individuelle pour apporter une solution à la redistribution des véhicules dans les stations. Le principe du suivi de véhicule[24][5] est basé sur l'utilisation d'une caméra placée à l'avant de chaque véhicule et d'une cible active placée à l'arrière. Sur le véhicule suiveur, à partir de la géométrie connue de la cible du véhicule précédent, on détermine par triangulation la position relative des deux véhicules dans le plan horizontal.

12.2.1 La cible active

Elle est constituée de balises à leds infrarouges disposées en triangle. En théorie trois balises suffisent pour déterminer la position du véhicule, mais en pratique nous en avons utilisé cinq afin de rendre la détection plus robuste. Trois balises centrales clignotent en alternance avec deux balises extrêmes. Ce clignotement d'une période de 5ms permet à la caméra d'extraire l'image utile (les balises) de l'image de fond (reflets, lumières etc.) en effectuant une différence entre deux images consécutives. Pour permettre le calcul de distance entre les véhicules, la balise centrale est décalée de 10cm en avant par rapport aux quatre autres balises disposées dans le plan arrière du véhicule (fig. §12.5).

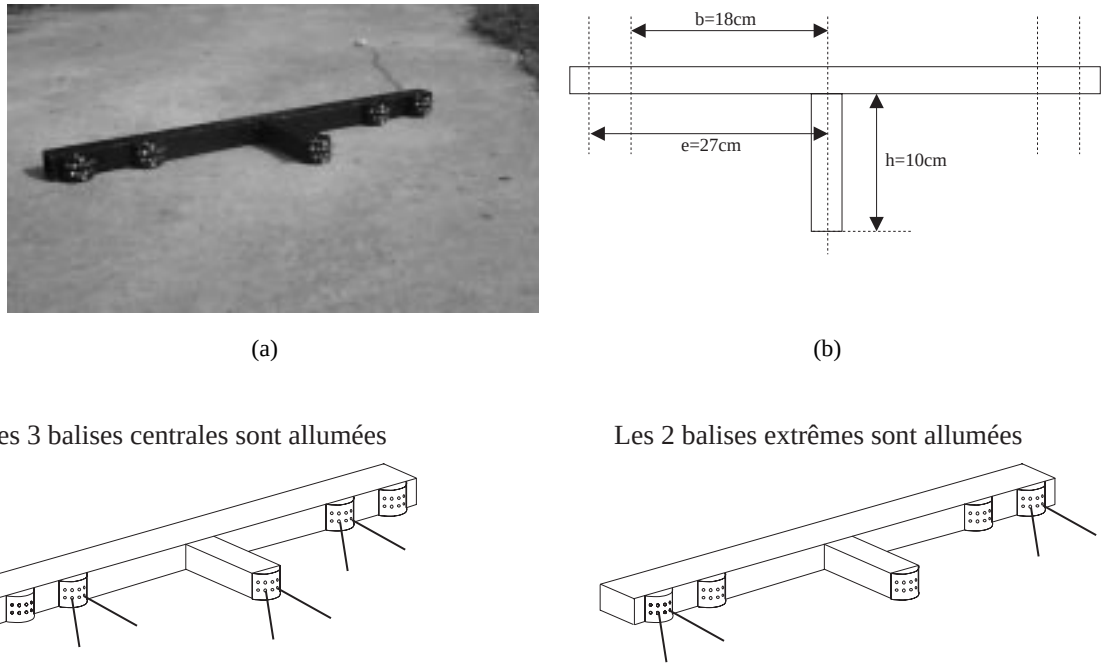
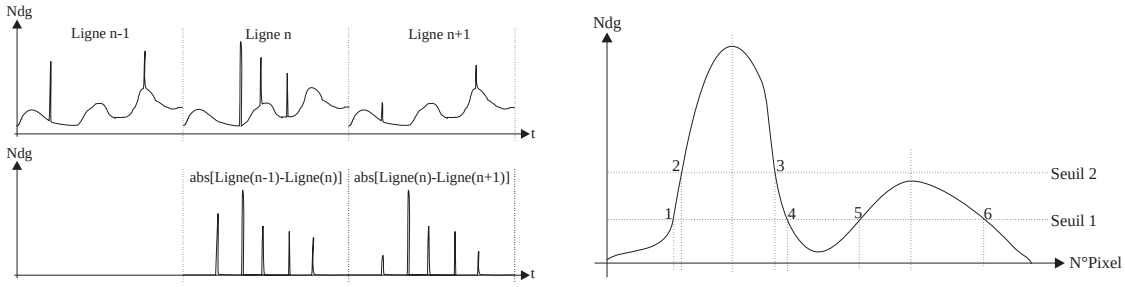


FIG. 12.5 – *Cible active*

12.2.2 La caméra

C'est une caméra linéaire développée spécialement pour le projet Praxitèle par la société COSE (startup INRIA). Elle est composée d'un capteur CCD 2048 pixels, d'une électronique de commande et de mise en forme du signal de sortie du CCD et d'un microcontrôleur Intel 80KC196 qui gère le fonctionnement de la caméra, calcule la position du véhicule précédent et la transmet sur une liaison RS232. Pour éviter de surcharger le microcontrôleur, la synchronisation de la capture de l'image avec les changement d'état de la cible (clignotement) et la différence de deux images consécutives (fig. 12.6(a)) sont réalisées matériellement par la carte électronique de commande de

la caméra. De même cette électronique réalise un seuillage permettant de récupérer uniquement la position et les niveaux de gris des pics qui franchissent des seuils définis ainsi que la position et le niveau de gris des points correspondant aux franchissements des seuils (fig 12.6(b)). Le temps d'exposition à la lumière du capteur CCD peut être contrôlé logiciellement afin de l'adapter aux conditions externes (luminosité, éloignement de la cible, pluie etc.).



(a) Mesure différentielle (b) Seuillage

FIG. 12.6 – *Traitement hardware du signal issu du capteur CCD*

12.2.3 Calculs de positionnement

La figure 12.7 présente les données géométriques de l'ensemble cible-capteur CCD nécessaire au calcul du déport latéral `Dep`, de l'écart d'angle α et de la distance `Dist` entre les deux véhicules utilisés pour la commande du véhicule.

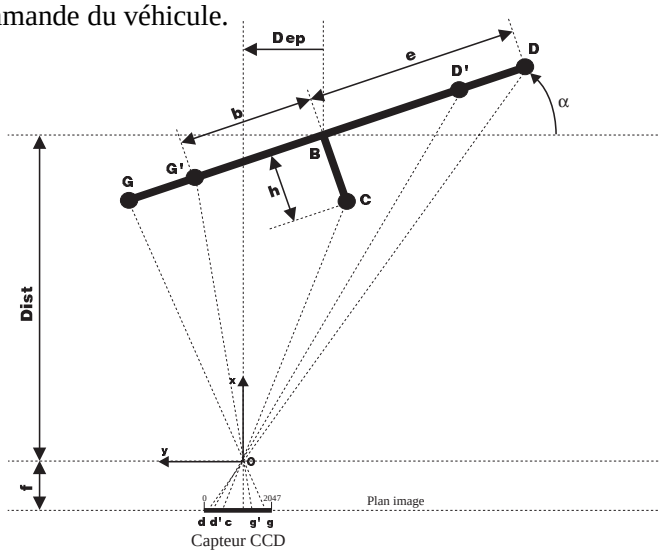


FIG. 12.7 – *Géométrie de l'ensemble capteur CCD et cible pour le calcul de positionnement*

Il a été montré dans [34] que pour connaître ainsi la position du véhicule précédent, la caméra doit effectuer les calculs suivant :

$$\tan(\alpha) = \frac{ef(g+d-2c) + hc(g-d)}{e[c(g+d) - 2gd] + hf(g-d)}$$

$$Dist = \frac{e\cos(\alpha)}{g-d}[2f + (g+d)\tan(\alpha)]$$

$$Dep = h\cos(\alpha)[\tan(\alpha) - \frac{c}{f}] + Dist\frac{c}{f}$$

Un filtre de Kalman doit aussi être calculé par la caméra. Son rôle est de rendre la mesure de position plus stable grâce à une prédiction de cette position lorsque le CCD ne perçoit plus la cible pendant quelques instants, de filtrer de manière optimale les bruits de mesure et enfin d'estimer la vitesse relative entre les deux véhicules.

12.2.4 Calcul de la consigne longitudinale

La consigne longitudinale à appliquer au CyCab a été présentée pour d'autres types de véhicules dans [5][24]. Le principe est de garder une distance entre les véhicules proportionnelle à la vitesse de déplacement du véhicule suiveur soit :

$$X_1 - X_2 = d_{min} + hV_2$$

où X_1 et V_1 sont la position et la vitesse du véhicule suivi, X_2 et V_2 la position et la vitesse du véhicule suiveur, d_{min} est une distance minimale de sécurité et h une constante de temps qui dépend du temps de réponse des moteurs électriques du véhicule. Ceci conduit à une consigne longitudinale en accélération (démonstration dans [5]) de la forme :

$$A_2 = h^{-1}(d_v + C_p(dx - hV_2 - d_{min}))$$

où dx est la distance $Dist$ donnée par la caméra, d_v est la vitesse relative estimée par le filtrage de Kalman, V_2 est estimée par odométrie, h^{-1} est le temps désiré qui sépare les véhicules et $C_p = h^{-1}$.

12.2.5 Calcul de la consigne latérale

Il existe différentes versions du calcul de la consigne latérale permettant d'obtenir avec des calculs plus ou moins complexes une précision plus ou moins grande sur l'écart entre la trajectoire empruntée par le véhicule suivi et le véhicule suiveur. La version la plus simple consiste à réaliser un suivi de type "remorque" pour lequel on fait pointer les roues du véhicule vers les roues arrières du véhicule de tête. La consigne d'angle des roues α à appliquer à la direction est la suivante :

$$\alpha = \tan^{-1}\left(\frac{d_y}{d_x}\right)$$

où d_x et d_y représentent l'écart de position entre les deux véhicules (fig. 12.8) prédit par le filtrage de Kalman appliqué aux données issues de la caméra.

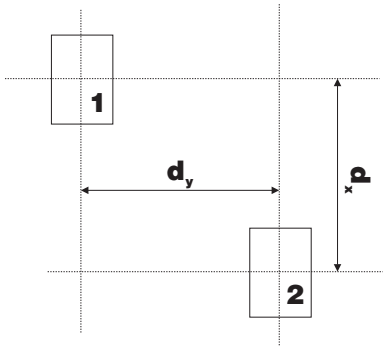


FIG. 12.8 – *Écart de position*

12.3 Localisation

Le but de la localisation est de connaître à chaque instant la position du véhicule dans un environnement a priori connu sans faire appel à un système GPS [1].

Le principe repose en partie sur l'utilisation des capteurs odométriques associés à chacun des moteurs de locomotion et au vérin de direction qui permettent de connaître la position angulaire des roues par rapport à leur axe de rotation (locomotion) et par rapport à l'axe du véhicule (direction). Grâce à ces informations, on peut reconstituer la trajectoire parcourue par le véhicule depuis un instant donné. Connaissant la position de départ du véhicule, on peut ainsi théoriquement connaître à chaque instant la position du véhicule. La figure 12.9 présente une trajectoire localisée uniquement par l'odométrie. On constate en pratique que la position dérive peu à peu jusqu'à devenir très peu précise.

Pour remédier à cette imprécision, on fait l'hypothèse que le conducteur reste toujours sur la route pour recaler la position estimée par l'odométrie. C'est le principe essentiel du *Map Matching*. En utilisant l'estimation obtenue par l'odométrie, on peut savoir sur quelle(s) route(s) le véhicule est susceptible de se trouver. Dès lors, sachant que la voiture se trouve sur la route, on peut corriger son cap en le mettant à la valeur du cap de la route et on peut corriger la position en la projetant sur la route (fig. 12.10). Cependant cette projection n'apporte une correction que sur la position latérale. C'est dans les virages que la position longitudinale est recalée.

Cette approche repose essentiellement sur l'utilisation d'un filtrage de Kalman qui permet de connaître l'erreur commise par l'odométrie. On sait ainsi avec certitude dans quelle zone se trouve le véhicule. Si une seule route se trouve dans la zone, le recalage est immédiat, il suffit de projeter la position estimée sur cette route. Lorsque plusieurs routes se trouvent dans la zone d'erreur, on envisage les différentes positions possibles jusqu'à ce que se révèlent les positions impossibles qui

1. Global Positionning System

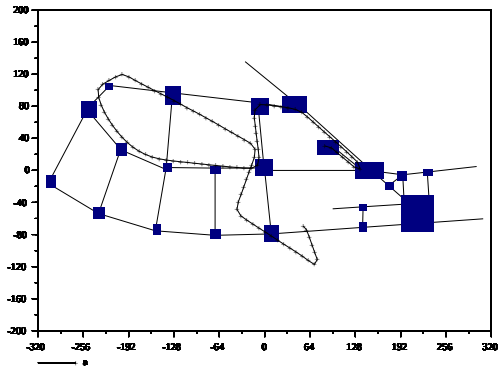


FIG. 12.9 – *Localisation en odométrie pure*

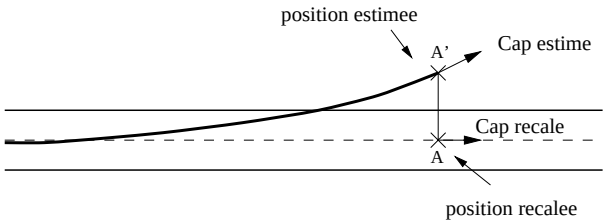


FIG. 12.10 – *Recalage de l'odométrie*

seront alors éliminées, c'est-à-dire lorsque la route envisagée finit par ne plus être dans la zone d'incertitude.

Par exemple sur la figure 12.11, la voiture estime être en A' lorsqu'elle est en A avec la zone d'incertitude grisée. Les deux routes possibles sur lesquelles elle peut se trouver sont les routes A et B. Lorsque la voiture va prendre le virage de la route B, elle va éliminer la route A qui ne possède pas de virage et qui va alors sortir de la zone d'incertitude.

Le lecteur pourra se rapporter à [34] pour une description détaillée du modèle d'erreur et sur le paramétrage du filtre de Kalman. La figure 12.12 illustre les bons résultats obtenus avec cette technique de map matching. Elle représente la trajectoire estimée recalée du CyCab sur le site de l'INRIA Rocquencourt.

12.4 Téléopération

La mise en place du projet Praxitèle repose en partie, comme nous l'avons déjà évoqué, sur une gestion intelligente de la flotte de véhicules et de leur répartition. Il est donc nécessaire d'imaginer des systèmes performants, pratiques et économiques pour le déplacement des CyCab à vide.

Dans ce contexte, la *téléopération*[12] peut être une alternative à la formation de trains de véhicules vides suivant un unique véhicule conduit manuellement (application de suivi de véhicule). Cette nouvelle application consiste à commander à distance les déplacements du véhicule. Ainsi un

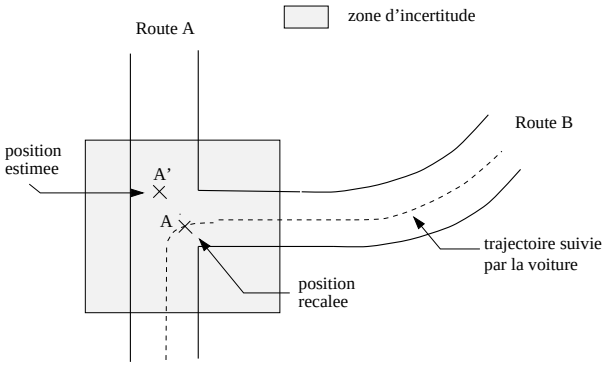


FIG. 12.11 – *Choix d'une route*

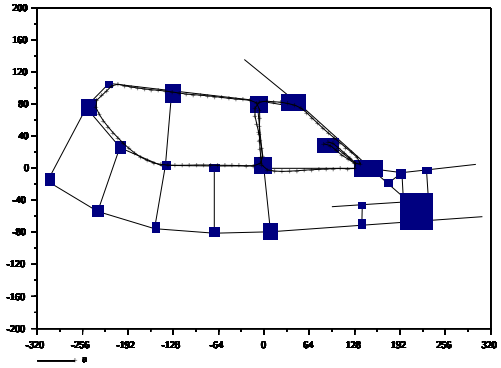


FIG. 12.12 – *Trajectoire estimée à l'aide du Map Matching*

opérateur installé dans un poste de commande fixe reçoit des images envoyées par une caméra située dans le CyCab et commande le véhicule à l'aide d'un joystick. L'inconvénient de cette méthode, par rapport au train de véhicules, est de mobiliser l'opérateur pour un seul véhicule. Cependant, il faut noter que dès le déplacement d'un véhicule effectué, l'opérateur est immédiatement disponible pour le déplacement d'un autre CyCab.

12.4.1 Principe

Le principe de cette application consiste, d'une part, à fournir à l'opérateur un retour visuel de l'environnement dans lequel évolue le CyCab ainsi qu'un ensemble d'informations utiles sur son état (vitesse, angle des roues, localisation, etc.), d'autre part, à fournir au CyCab les consignes de déplacements générées par l'opérateur à l'aide d'un joystick (fig. 12.13).

Pour supporter cette communication radio bidirectionnelle entre le poste fixe et le CyCab nous avons intégré au PC embarqué danc le CyCab une carte réseau Ethernet Radio de type WaveLan distribuée par Lucent Technologies. Néanmoins, au moment où nous avons développé cette application, aucun constructeur n'était en mesure de nous fournir, à un prix raisonnable, une bonne carte

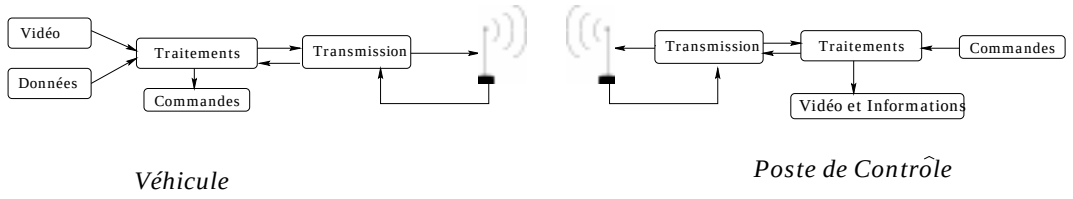


FIG. 12.13 – *Synoptique de la téléopération*

de numérisation et de compression vidéo accompagnée d'une documentation suffisamment précise permettant de la programmer en assembleur ou en C. En effet, tous les fabriquants de bonnes cartes graphiques refusent de fournir autre chose que des drivers déjà compilés, fonctionnant sous Windows et donc inutilisables dans une application fonctionnant sous DOS. Nous avons donc adopté une solution moins élégante utilisant une transmission analogique de l'image, parallèlement à la transmission numérique des données réalisée par la carte Wavelan. La figure 12.14 présente le principe et l'architecture de cette application.

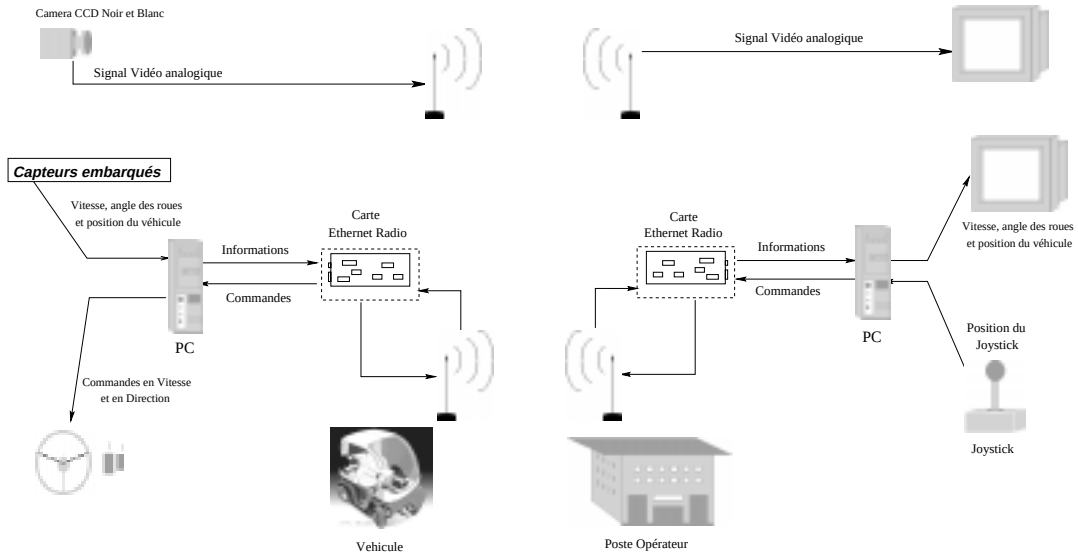


FIG. 12.14 – *Principe et architecture matérielle de la téléopération*

12.4.2 Chaîne de traitement vidéo

Deux petites caméras bas coût grand angle de type webcam ont été positionnées sur le CyCab de manière à fournir des images de l'environnement avant et de l'environnement arrière de celui-ci. Les sorties vidéo de ces caméras sont reliées à un émetteur vidéo analogique (2,4Ghz, couverture théorique d'1km en vue directe). Un relais électrique piloté par une sortie numérique d'un nœud du CyCab permet de sélectionner le flux vidéo à émettre. L'opérateur peut ainsi choisir à distance la vue avant ou la vue arrière du véhicule en fonction des manœuvres qu'il effectue. Du coté poste opérateur, un récepteur vidéo analogique fournit à un moniteur les images reçues.

12.4.3 Chaîne de traitement des données numériques

Le PC embarqué dans le CyCab gère la carte Wavelan de transmission numérique. Les données qu'il envoie au poste fixe sont principalement la position angulaire des roues par rapport à l'axe du véhicule (direction), la vitesse de déplacement et la position du véhicule dans le site de l'INRIA Rocquencourt.

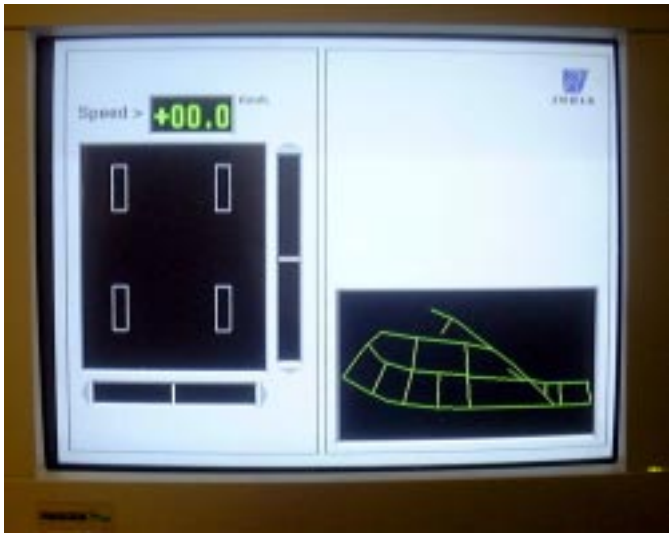


FIG. 12.15 – *L'écran du poste de contrôle.*

L'écran du poste fixe a été divisé en plusieurs champs (Fig 12.15) :

- Un premier champ représente un afficheur digital. Il indique en temps réel la vitesse du véhicule en $km.h^{-1}$;
- un deuxième champ est consacré à la représentation vue du haut de la position des roues du véhicule. Elle permet de visualiser en temps réel l'angle des roues. Dans une version future, elle permettra d'afficher des informations de distances, fournies par des capteurs ultrasonores, entre le CyCab et d'éventuels obstacles situés dans son environnement proche ;
- deux autres champs représentant des baragraphes indiquent la valeur de la vitesse de déplacement et de l'angle des roues par rapport aux valeurs maximales qu'il est possible d'atteindre ;
- enfin, la position du véhicule est repérée par une croix sur une carte du site de l'INRIA Rocquencourt.

Un joystick connecté au PC du poste opérateur permet de commander à distance le CyCab. L'amplitude de déplacement horizontal du joystick est convertie en consigne d'angle des roues. L'amplitude de déplacement vertical était, dans un premier temps, convertie en une consigne d'accélération. Quelques tests de conduite ont très vite mis en évidence la difficulté de maintenir constante la vitesse. Nous sommes alors passés à une relation linéaire transformant la position du joystick en

une vitesse. Ceci a amélioré assez sensiblement la conduite du véhicule mais en laissant toutefois quelques problèmes :

1. les faibles vitesses sont difficiles à obtenir et à maintenir ;
2. le véhicule ne peut réagir aussi vite que la commande.

Pour remédier à ces inconvénients nous avons décidé de changer le profil de vitesse : à la place du profil linéaire nous avons choisi un profil parabolique pour les faibles vitesses, puis un profil linéaire pour les vitesses plus élevées (fig. 12.16).

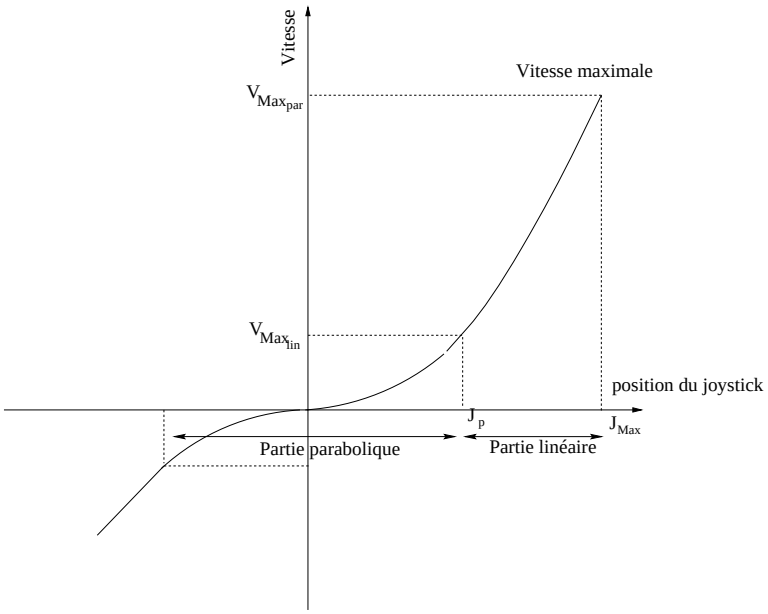


FIG. 12.16 – *Profil de vitesse*

Afin de ne pas envoyer au véhicule des commandes trop “impulsives”, nous avons ensuite filtré la commande en vitesse par un filtre du premier ordre. Après avoir réalisé ceci, nous nous sommes rendus compte que le joystick ne renvoyait pas constamment zéro lorsqu’il était au centre, ce qui risquait de faire trembler le véhicule à l’arrêt. Nous avons donc rajouté un léger seuil afin de renvoyer un “zéro franc” lorsque le joystick est au centre.

Une dernière possibilité a été ajoutée au niveau du poste opérateur : en appuyant sur une touche l’opérateur envoie des données vers le véhicule qui passe alors en “mode téléopération”. En appuyant sur une autre touche le CyCab passe en “mode manuel”.

CHAPITRE 13

RÉALISATION DU LOGICIEL AVEC SYNDEX

Dans ce chapitre nous montrons en détail, principalement sur l'exemple de la conduite manuelle sécurisée du CyCab de Rocquencourt, comment les lois de commande définies pour chacune des applications ont été spécifiées et implantées sur l'architecture matérielle à l'aide de SynDEx v4.

13.1 Spécification de la conduite manuelle sécurisée

L'interface graphique et textuelle de SynDEx v4 permet de décrire les aspects matériels et les aspects logiciels du système temps réel. Ainsi, le matériel est décrit par un graphe appelé *graphe d'architecture* et le logiciel par un graphe appelé *graphe d'algorithme*.

13.1.1 Graphe d'architecture

13.1.1.1 Sémantique du graphe d'architecture

Un graphe d'architecture peut être constitué de deux types de sommets, des sommets opérateurs et des sommets bus de communication (medias SAM). Le sommet opérateur possède des ports de type PBL, PPL ou PSL représentant des interfaces de communications bidirectionnelles. Un port PBL permet de connecter l'opérateur à un bus afin de réaliser une liaison bus SAM. Un port PPL d'un opérateur peut être connecté à un port PPL d'un autre opérateur afin de représenter une liaison point à point SAM. Un port de type PSL représente une liaison particulière autorisant l'accès à la ressource mémoire (disque dur, liaison Ethernet, etc.) par laquelle les exécutables sont chargés sur les processeurs. Dans SynDEx v4 ce graphe est décrit à l'aide de trois fonctions du langage de spécification (`processor` pour la création d'un opérateur, `bus` pour la création d'un bus SAM et `connect` pour réaliser les connections entre les ports).

13.1.1.2 Graphe d'architecture de l'application suivi de véhicule CyCab

La figure 13.1 montre une copie d'écran du graphe matériel SynDEx représentant l'architecture embarquée dans le CyCab. Elle est composée de cinq opérateurs représentant les cinq microcontrôleurs MC68332 (`AvG332`, `AvD332`, `ArG332`, `ArD332`, `Dir332`), d'un opérateur `root` représentant le 486DXII66 et d'un opérateur `CAM196` représentant le 80KC196 de la caméra linéaire spécialement conçue pour l'application de suivi de véhicule. Les cinq MC68332 et le 486DXII66 sont tous reliés par un port PBL à un sommet `CAN` représentant le bus de communication CAN. Le 80KC196 et le 486DXII66 sont reliés par une liaison point à point SAM représentant une liaison RS232.

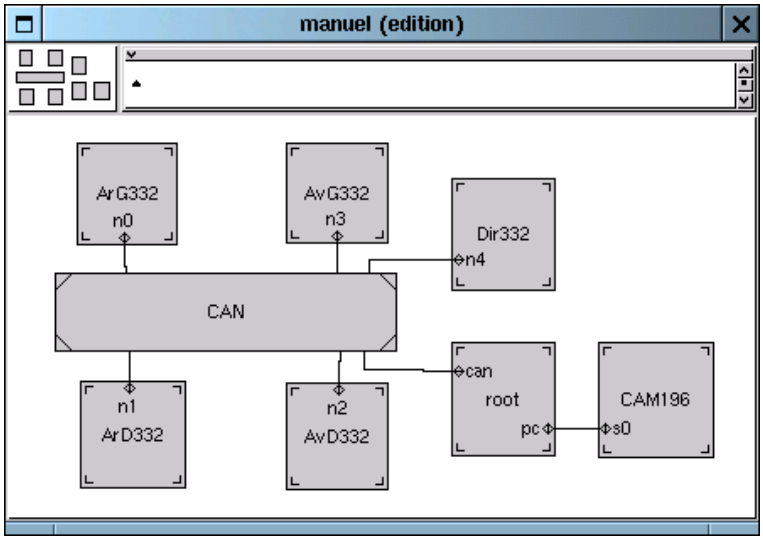


FIG. 13.1 – *Graphe Matériel*

13.1.2 Graphe d'algorithme

13.1.2.1 Sémantique du graphe d'algorithme

Le langage de spécification d'algorithme de SynDEx v4 est un langage graphique basé sur la sémantique du langage SIGNAL.

Les sommets du graphe sont des opérations à exécuter sur des flots de données matérialisées sur le graphe par des arcs. Ces opérations sont des fonctions (au sens mathématique du terme) de transformation de flots et les arcs traduisent des dépendances de données entre ces fonctions.

Dans un graphe SynDEx v4 on peut distinguer trois types de sommets : des sommets d'entrée qui produisent uniquement des données, des sommets de calcul qui produisent et consomment des données et des sommets de sortie qui consomment uniquement des données. Typiquement, les sommets d'entrée représentent des accès capteurs et ceux de sortie représentent des accès actionneurs. L'exécution du graphe est répétée infiniment ; les données le traversent de la gauche vers la droite c'est-à-dire des sommets d'entrée vers les sommets de sortie. Le temps d'exécution d'une itération du graphe correspond à la période temps-réel.

Une opération est créée dans le graphe à l'aide de la primitive textuelle du langage `function` qui permet de spécifier son nom, le nom de la procédure que le linker devra appeler lors de la génération de code, sa durée d'exécution, le nom et le type des flots de données qu'elle consomme et produit. Ce type représente le type des données. Dans SynDEx v4 quatre types sont autorisés : `logical`, `integer`, `real`, `dpreal`. Il est aussi possible de définir des tableaux de données basées sur l'un de ces quatre types.

Les opérations de calcul ne doivent pas réaliser d'effets de bord. Il existe un opérateur particulier, `memory` qui permet de mémoriser des données d'une itération sur l'autre du graphe. Ce sommet `memory` pour lequel il est possible de définir les valeurs initiales et le nombre de valeurs précédentes à mémoriser correspond au "retard" et aux "fenêtres glissantes"des traîteurs de signal.

Il existe, comme en SIGNAL, des opérations de calculs particulières qui permettent de modifier l'horloge des flots de données :

- `when` : il est utilisé pour définir un sous-échantillonnage d'un flot. Le `when` possède une entrée de type `logical`, et une entrée et une sortie de même type. Lorsque la valeur de l'entrée booléenne est $vrai$ la sortie recopie l'entrée. Lorsque la valeur est $faux$, la valeur du flot de sortie n'est pas définie, son horloge n'est pas présente à cet instant,
- `default` : ce type de sommet possède deux entrées et une sortie toutes de même type. L'une des entrées baptisée *pri* est prioritaire par rapport à l'autre. La sortie recopiera la valeur de l'entrée *pri* si elle est présente, que l'autre entrée soit présente ou non, sinon elle recopiera la valeur de l'autre entrée,
- `hmul` : c'est une implantation spéciale du `when` pour laquelle les 2 entrées et la sortie sont de type booléen. Il permet de calculer l'intersection (occurence simultanée) de deux flots booléens.
- `hadd` : c'est une implantation spéciale du `default` pour laquelle les 2 entrées et la sortie sont de type booléen. Il permet de calculer l'union des occurences de deux signaux booléens,

Les instructions `execroot` et `exec` permettent de définir l'horloge d'exécution des opérations. Ainsi l'instruction `exec` spécifie les sommets dont l'exécution est conditionnée par un booléen calculé par un autre sommet. L'instruction `execroot` spécifie les sommets dont l'exécution n'est pas conditionnée.

Enfin, l'instruction `constrain` permet de spécifier des contraintes de placement pour certains sommets du graphe d'algorithme. En règle générale, on évite de contraindre le placement des sommets de calcul pour laisser à SynDEx le maximum de latitude pour trouver une solution de placement et d'ordonnancement optimale. Par contre, les sommets d'entrées et de sorties doivent obligatoirement être contraints à être exécutés sur un opérateur particulier car ces sommets réalisent des accès à des ressources matérielles (lecture ou écriture dans un fichier, accès à un capteur ou à un actionneur) et doivent donc être placés sur l'opérateur qui gère cette ressource.

13.1.2.2 Graphe d'algorithme de l'application conduite manuelle sécurisée

Pour spécifier le graphe d'algorithme, les lois de commande doivent être découpées en opérations. Ce découpage est nécessaire pour tenir compte de la nature distribuée de l'application. Par

exemple, les calculs nécessaire à la gestion de deux transducteurs liés à des processeurs différents doivent être divisés en deux opérations distinctes qui seront chacunes exécutées sur le processeur qui les concerne. Le découpage devra autant que possible permettre la parallélisation des calculs. La figure 13.2 présente le découpage que nous avons choisi pour la loi de commande latérale de l'application de conduite manuelle sécurisée, le graphe d'algorithme complet de cette application est présenté figure 13.3(a).

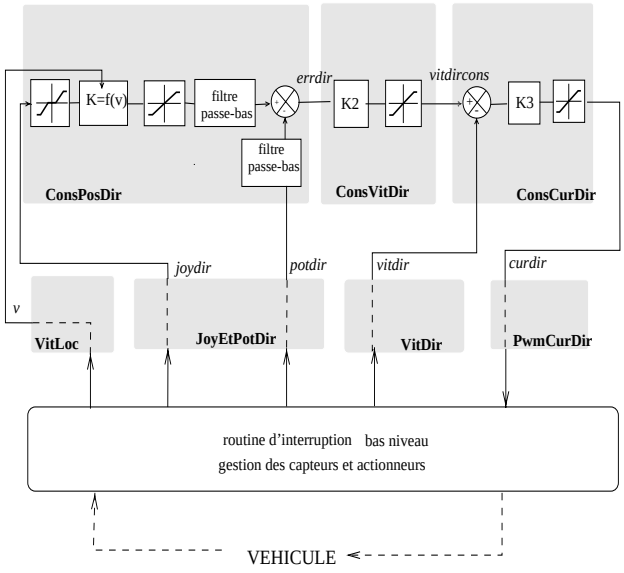
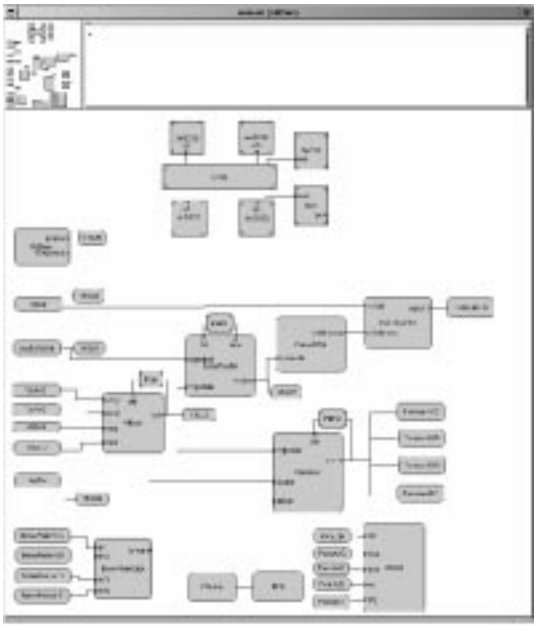
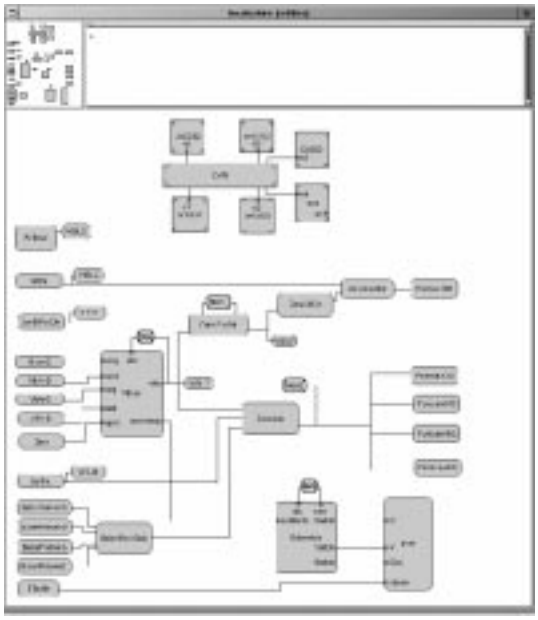


FIG. 13.2 – *Asservissement de controle latéral*

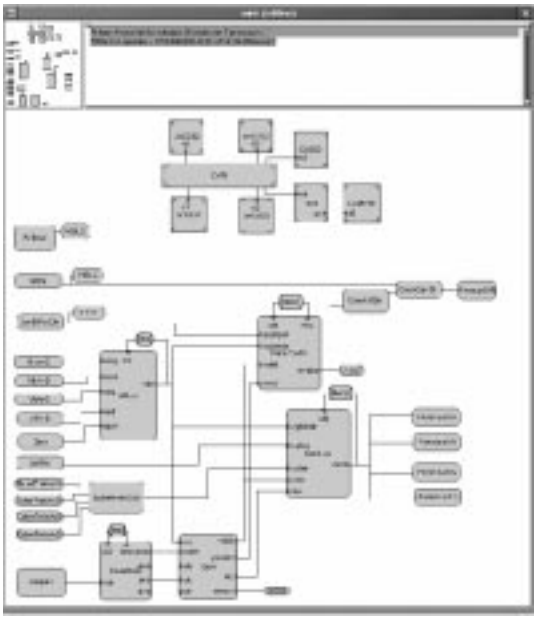
L'opération `ConsPosDir` réalise le calcul de l'erreur en position du vérin `errdir`. `ConsVitDir` transforme cette erreur de position en consigne de vitesse `vitdircons` qui est elle-même transformée en consigne de courant `curdir`. Le sommet `PWMcurDIR` implante le calcul de la commande en PWM présentée §12.1. `VitDir` fournit la vitesse de déplacement du vérin ($vitfiltr$) alors que `JoyEtPotDir` fournit la position du joystick et la position du vérin. L'acquisition de ces deux données a été réunie dans ce même sommet, car elles sont toutes les deux consommées par l'opération `ConsVitDir` pour laquelle on ne connaît pas a priori le processeur sur lequel elle sera exécutée. Ainsi si `JoyEtPotDir` et `ConsVitDir` sont placées sur des processeurs différents, une communication via le bus CAN des deux données sera réalisée. Les avoir regroupées dans un même tableau permettra de les envoyer dans la même trame CAN plutôt que dans deux trames différentes. On économise ainsi de la bande passante CAN. La vitesse du véhicule est calculée par `VitLoc`, l'acquisition de la vitesse de chacune des roues est réalisée par les sommets `VitAvG`, `VitAvG`, `VitAvG`, `VitAvG` (fig. 13.3(a)). Le calcul de la consigne en courant appliquée à chacun des moteurs de locomotion est calculée par `ConsLoc`. Les opérations `PwmcurAVG`, `PwmcurAVG`, `PwmcurAVG` et `PwmcurAVG` réalisent, comme pour la loi de commande latérale, le calcul en PWM de la commande de chacun des moteurs. Les sommets `VISU`*num* ont été rajoutés au graphe représentant la commande du véhicule. Ils permettent "d'espionner" les données calculées en les affichant ou en les sauvegardant sur disque. Cette facilité est très utile pendant la période de test et de deboggage de l'application. Les sommets d'entrée et de sorties baptisés *nom*`ARG` (respectivement *nom*`ARD`, *nom*`AvG`, *nom*`AVD` et *nom*`DIR`) sont contraints à être exécutés sur le processeur 68332ARG (respectivement 68332ARD, 68332AVG, 68332AVD et 68332DIR). Tous les sommets `VISU`*num* sont contraints à être exécutés sur le processeur root qui gère le disque dur et l'écran.



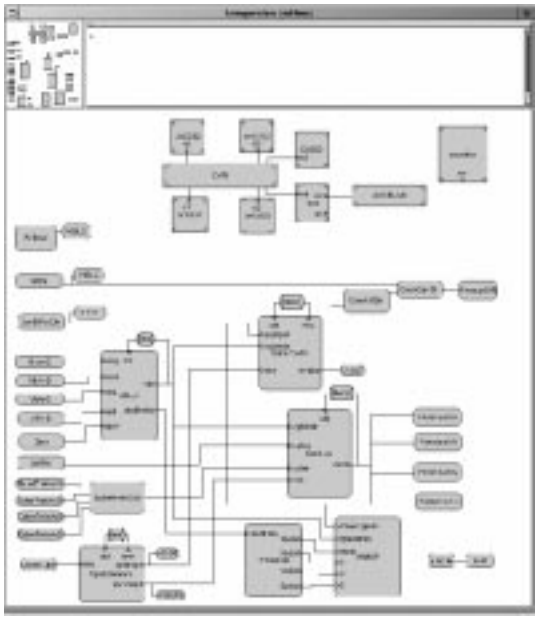
(a) Conduite manuelle



(b) Localisation



(c) Suivi de véhicule



(d) Téléopération

FIG. 13.3 – *Spécification SynDEx des applications du CyCab*

Le code généré par SynDEx implante une exécution autocadencée du graphe, c'est-à-dire que l'exécution de l'itération suivante du graphe débute dès que l'exécution de l'itération courante est terminée. Ainsi, la période d'exécution du graphe peut varier d'une itération à l'autre en fonction de la variation de durée totale d'exécution d'une itération, ce qui a pour conséquence d'introduire du jitter sur l'échantillonnage de signaux pouvant rendre le système instable. Pour remédier à ce problème, nous avons intégré au graphe le sommet `PcTimer` qui génère un booléen $vrai$ toutes les 10ms sur interruption timer du PC. Tous les autres sommets du graphe sont exécutés sur la valeur $vrai$ de ce booléen (`exec`). Le graphe est ainsi exécuté périodiquement toutes les 10ms.

En réalité, les sommets `PwmcurDIR`, `PwmcurAVG`, `PwmcurAVD`, `PwmcurARG` et `PwmcurARD` n'implantent pas directement le calcul de t^+_{pwm} en fonction de I_{cons}. En fait, ces calculs doivent être exécutés à une fréquence plus élevée (1ms) que les autres calculs. SynDEx v4 ne permettant pas d'exprimer plusieurs contraintes de cadence, ces calculs sont réalisés par une routine déclenchée par interruption timer. Les fonctions `PwmcurDIR`, `PwmcurAVG`, `PwmcurAVD`, `PwmcurARG` et `PwmcurARD` se contentent donc de mettre à jour des cases mémoires lues par les routines d'interruptions des processeurs. De même, les sommets `VitDir`, `VitAVG`, `VitARG`, `VitAVD` et `VitARD` lisent simplement des cases mémoires mises à jour par ces routines d'interruption. Ceci permet d'exécuter les asservissements en PWM de bas niveau à une période fixe (pour le calcul de vitesse des roues) plus petite que la période temps-réel (10ms) définie par le graphe (pour, par exemple, éviter les pics de courant dans les moteurs).

La conduite manuelle sécurisée a servi de base à la réalisation des applications de suivi de véhicule, de localisation et de téléopération. Les graphes logiciels de chacune de ces applications sont donc constitués du graphe d'algorithme de la conduite manuelle auquel a été ajouté des sommets de calculs spécifiques à chacune des applications. Nous donnons figure 13.3 une copie d'écran de chacun de ces graphes pour que le lecteur puisse se faire une idée de la facilité avec laquelle on peut prendre en compte des modifications de l'algorithme et/ou de l'architecture.

Localisation : l'opération `VitLoc` a été modifiée, elle prend maintenant en compte une mesure d'accélération réalisée par le gyromètre `Gyro` et fournit une variation de distance et d'angle à une opération `Odométrie` qui implante les calculs de position du véhicule.

Suivi de véhicule : le graphe d'architecture prend en compte le 80KC196 de la caméra linéaire. Les valeurs des pics correspondant sur l'image de la cible aux balises sont lues par l'opération `Campic`. À partir de ces valeurs l'opération `CalculCam` calcule la position, la vitesse, le cap, etc. du véhicule précédent. L'opération `Suivi` implante le calcul de la consigne en accélération et en cap du véhicule.

Téléopération : le graphe d'architecture prend en compte le PC de l'opérateur et la liaison Ethernet Radio. Les sommets de calcul de la commande `OpeCommand` et de visualisation de l'état du véhicule `CabbyVisu` ont été ajoutés.

13.2 Adéquation

Lorsque les graphes d'algorithme et d'architecture ont été spécifiés, SynDEx peut réaliser l'Adéquation et afficher une prédiction temporelle de l'exécution de l'algorithme sur l'architecture prennant en compte les durées des communications. Les durées des calculs sont les durées données par l'utilisateur lors de la spécification des opérations du graphe d'algorithme. Ces durées peuvent être estimées par l'utilisateur, elle peuvent aussi être mesurées lors d'une première exécution (option du générateur de code) et être réintroduites dans le graphe d'algorithme en vue de réaliser une nouvelle Adéquation prenant en compte des durées de calcul réalistes. La figure 13.4 montre les temps d'exécution prédits par SynDEx pour l'application de conduite manuelle sécurisée, une fois l'Adéquation réalisée. Chaque colonne représente la séquence des opérations assignées à chaque processeur. Le temps se déroulant du haut vers le bas, la hauteur des colonnes représente le temps d'exécution du graphe. Les arcs représentent les communications inter-processeurs. Plus précisemment, l'origine d'un arc est la date à laquelle la donnée est prête à être envoyée, la fin de l'arc est la date à laquelle elle est disponible sur le processeur destinataire.

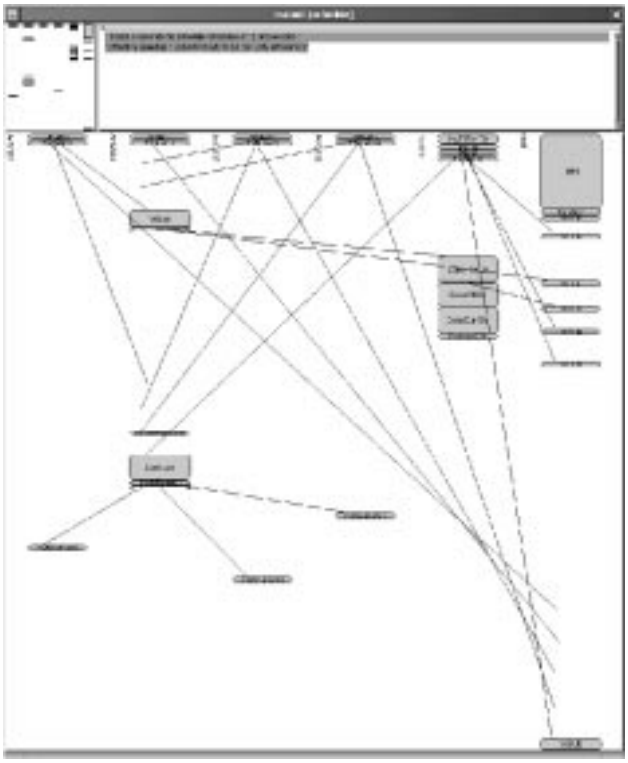


FIG. 13.4 – *Schedule*

Cette prédiction temporelle met en évidence, que les temps de communication sont important comparés aux temps de calcul.

13.3 Génération automatique d'exécutif

Lorsque la prédiction temporelle satisfait l'utilisateur, celui-ci demande à SynDEx de générer l'exécutif de l'application. SynDEx crée alors un fichier pour chaque opérateur du graphe matériel. Chaque fichier décrit l'exécutif implantant la séquence de calcul et de communication que le processeur doit exécuter. Cet exécutif s'appuit sur un ensemble de primitives décrites dans un "noyau générique".

13.3.1 Structure de l'exécutif

L'exécutif gère les communications, garantit le respect des dépendances de données lors de l'exécution et exploite le parallélisme matériel qui existe entre le séquenceur d'instructions du processeur et les interfaces de communication qui lui sont attachées. Afin de limiter les surcoûts de calcul liés à l'exécution de cet exécutif, celui-ci est principalement statique (l'ordonnancement a été choisi lors de l'adéquation). La structure de l'exécutif est très simple, elle est proche du programme que l'utilisateur aurait pu réaliser "à la main" :

```
Début
     - initialisations processeur et allocation mémoire
     - en parallèlle: séquences de communication et de calcul

         • séquence com
                          init   init du média,
                                 étiquetage séquence de com
                          while  séquence de com
                          end    désactivation du média

         • séquence calcul
                          init   init Cstes, et chronos
                                 lancement com
                                 init entrées/sorties
                          while  séquence de calcul
                          end    finalisations entrées/sorties,
                                 collecte chronos
Fin
```

Au lancement de l'application des initialisations (processeur et mémoire) sont réalisées, une séquence de calcul est alors lancée en parallèle d'une séquence de communication. Ces séquences sont toutes les deux divisées en trois parties correspondant à des initialisations précédent une boucle qui exécute indéfiniement la séquence de calcul (resp. de communication) proprement dite. Lorsque l'application est arrêtée une partie finalisation permet par exemple de s'assurer que le matériel est remis dans un état stable et sécurisé.

13.3.2 Mécanismes garantissant le respect des dépendances de données

L'ordonnancement étant séquentiel, les dépendances de données intra-processeurs sont garanties par l'ordre d'exécution de la séquence de calcul. La séquence de communication étant exécutée en parallèle avec la séquence de calcul, les dépendances de données inter-processeurs doivent être garanties par des sémaphores. La libération d'un sémaphore est réalisée lors de la fin d'exécution d'une opération productrice de données, elle autorise l'exécution des fonctions consommatrices des données produites. On garantit ainsi une précédence tampon-plein baptisée *pre*. Il faut aussi garantir, d'une itération sur l'autre, que l'exécution d'une opération consommatrice d'une donnée est terminée avant le début d'exécution de l'opération productrice de l'itération suivante. Cette précédence tampon vide appelée *suc* est réalisée par la prise d'un sémaphore par la fonction consommatrice. SynDEx génère donc une séquence de calcul et une séquence de communication synchronisées à l'aide de ces sémaphores. Un *suc* positionné avant une transmission dans la séquence de communication met celle-ci en attente de la libération du sémaphore par le *pre* correspondant dans la séquence de calcul. Ainsi la donnée ne peut être envoyée que lorsque l'exécution de l'opération de calcul qui la produit est terminée. De la même manière, un *pre* positionné après une réception dans la séquence de communication associée à un *suc* dans la séquence de calcul autorisera l'exécution de l'opération qui consomme la donnée reçue.

13.3.3 Noyau générique

L'exécutif distribué généré par SynDEx repose sur un ensemble de primitives. Il est écrit en macro-code M4 (GNU)[71]. Pour chaque type de processeur et de média de communication utilisés dans le graphe matériel, l'utilisateur doit définir la traduction des macro M4 d'appel à ces primitives en un langage compilable en code exécutable. L'ensemble de ces définitions constitue le "noyau générique". On peut classer les macros qui constituent le noyau en huit catégories :

chargement arborescent des programmes : ces macros permettent de décrire le graphe d'architecture sous une forme arborescente et assure le chargement des programmes sur cette arborescence ;

initialisation hardware : ce sont les macros chargées de réaliser des initialisations spécifiques du matériel au début de l'application (par exemple, initialisation de la vitesse de transmission d'une liaison RS232, etc.) ;

gestion mémoire : ce sont les macros d'allocation, de renommage, d'initialisation de zones mémoires, de copie et de décalage de fenêtres glissantes ;

entrées/sorties : ces macros sont spécifiques à l'application, elles doivent décrire les accès capteurs et actionneurs, les accès clavier et écran, etc. ;

contrôle conditionnel et itératif : ce sont les macros qui permettent de réaliser des boucles et des exécutions conditionnelles (`if`, `else`, `while`, `endif`, etc.) ;

synchronisation : ces macros implantent la gestion des sémaphores (`pre` et `suc`) ainsi que la déclaration et le lancement parallèle des séquences de communication sur la séquence de calcul ;

communication : elles décrivent pour chaque média de communication les opérations de réception et d'envoi de données ;

chronométrage : elle réalisent la mesure des temps d'exécution des opérations et la collecte arborescente de ces mesures pour leur sauvegarde sur disque.

13.3.4 Chaîne de compilation

La chaîne de compilation dépend des processeurs, des langages et outils de compilation choisis par l'utilisateur. Pour notre CyCab, nous avons choisi d'utiliser l'assembleur pour décrire le noyau générique et le langage C pour décrire les procédures de calcul associées aux sommets de calcul du graphe d'algorithme. L'écriture de code assembleur effraie les informaticiens plus souvent habitués à utiliser des langages de plus haut niveau. La réalisation du noyau générique nécessite l'écriture d'une cinquantaine de primitives simples qui réalisent des opérations simples de lecture et d'écriture dans des zones mémoires spécifiques (par exemple lecture d'un capteur, copie d'une case mémoire, etc.). L'assembleur est le langage le mieux adapté pour réaliser ce type d'opérations et maîtriser complètement le fonctionnement de plus bas niveau des applications. L'utilisation de ce type de langage se justifie aussi par l'obtention d'un code compact garantissant de meilleures performances temps réel. De plus, lorsque ces primitives sont écrites et déboguées, il n'est plus nécessaire d'y toucher, l'utilisateur se contente pour toutes les applications qu'il envisage de réaliser avec le CyCab, de fournir les fonctions de calcul dans un langage de plus haut niveau, le langage C. Ces primitives écrites une seule fois et réutilisées par toutes les applications utilisant l'architecture matérielle pour laquelle elles ont été écrites méritent donc d'être particulièrement soignées. La figure 13.5 représente, dans les principes, notre chaîne de compilation.

Le "noyau générique" est écrit une seule fois pour toutes les applications. L'utilisateur décrit le graphe d'architecture et le graphe d'algorithme de l'application, il fournit les procédures C liées à chacune des opérations de calcul du graphe d'algorithme. SynDEx réalise l'Adéquation et génère un fichier m4 décrivant les séquences de calcul et de communication pour chacun des processeurs de l'application. Le macro-processeur GNU M4 transforme les fichiers .m4 en code assembleur (fichiers .as). À ce stade, on dispose d'un programme assembleur par processeur. Ces programmes incluent les appels aux fonctions C utilisateurs. Celles-ci sont compilées et linkées avec ce code assembleur par GCC (Gnu) qui peut alors générer un exécutable par processeur.

Pour nos applications , il existe en réalité deux chaînes de compilation. Toutes les compilations des programmes qui doivent s'exécuter sur les MC68332 sont réalisées sur station de travail. La compilation des programmes du 80CK196 de la caméra et du 486DXII66 est réalisée sur le PC embarqué dans le CyCab. Enfin un loader exécuté sur le PC est chargé d'envoyer sur le bus CAN les programmes à chacun des MC68332 et de lancer leur exécution.

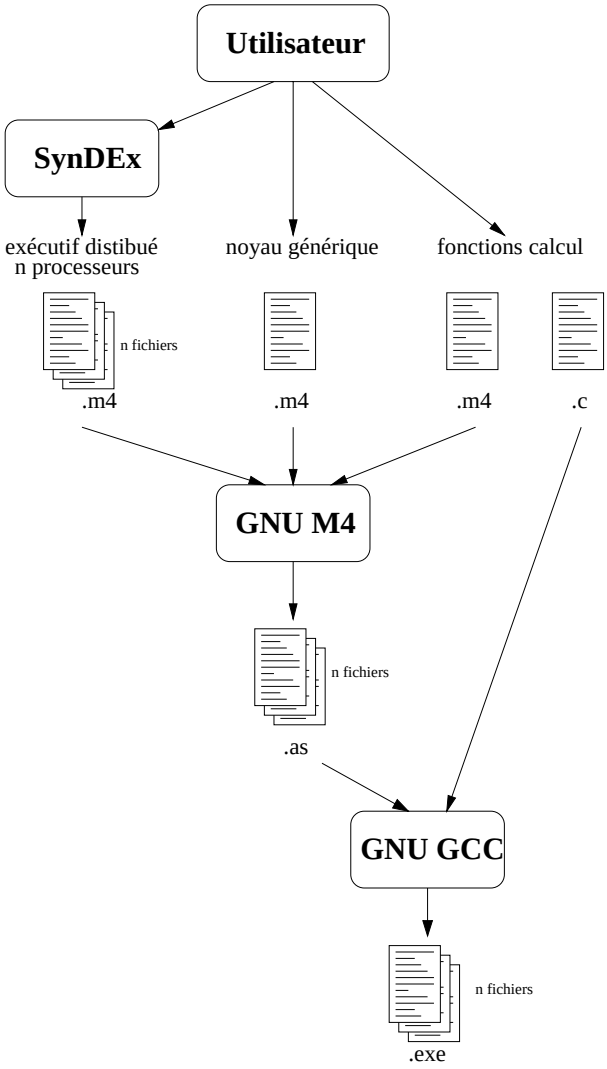


FIG. 13.5 – *Chaîne de compilation*

Cinquième partie

Conclusion

Le cycle de développement en V utilisé pour la réalisation des systèmes temps réel permet selon une hiérarchisation descendante par étape (modélisation, spécification, implantation) d'aboutir à la conception détaillée d'une application à partir d'une description abstraite. A chaque étape de conception correspond une étape de validation (tests unitaires, intégration, validation). Le cycle de développement est réitéré jusqu'à ce que toutes les étapes de conception soient validées. Dans ce cycle, il existe deux grandes ruptures. La première est le passage manuel du modèle à la spécification, la seconde est le passage manuel de la spécification à l'implantation reposant tous deux sur le savoir-faire des ingénieurs. Ces ruptures sont à l'origine d'erreurs nécessitant de nombreuses réitérations du cycle pour les corriger. L'objectif de cette thèse est de contribuer à définir une méthodologie permettant de supprimer ces ruptures afin d'aboutir à moindre coût à la réalisation des applications temps réel embarquées. Les méthodes formelles ont permis de supprimer la rupture entre la spécification et l'implantation grâce à la vérification et à la génération automatique du code conforme à la spécification. Le but de cette thèse est de contribuer à définir des méthodes et outils permettant de supprimer ces ruptures. C'est donc sur une méthode de type formelle, la méthodologie AAA, qu'est fondée ce travail. Cette méthodologie a été conçue pur réaliser une implantation automatique optimisée d'une spécification sur une architecture matérielle distribuée en garantissant que cette implantation est conforme à la spécification.

Cette thèse a permis de montrer, à travers la réalisation d'un prototype de véhicule électrique autonome, le CyCab, que cette méthodologie permet de mener à bien à moindre coût la conception de systèmes temps réel embarquée distribués. Dans cette thèse nous avons aussi cherché à supprimer la rupture qui existe entre la modélisation et la spécification. Cette rupture est d'autant plus grande que ces étapes sont réalisées par des personnes appartenant à des disciplines différentes, l'automatique pour la modélisation, l'informatique pour la spécification. Afin de faciliter les interactions entre les automaticiens et les informaticiens, nous avons établit un lien entre les terminologies couramment employés dans ces deux disciplines. Nous avons aussi montré que les hypothèses émises par l'automaticien lors de la modélisation se traduisent à l'implantation par des contraintes temporelles.

La spécification est l'étape charnière entre la modélisation et l'implantation. Une méthode de développement cohérente permettant de passer de la modélisation à l'implantation sans rupture dans le cycle doit donc fournir un outil de spécification permettant de prendre en compte les besoins de l'automaticien pour concevoir ces modèles et d'autre part elle doit permettre de pouvoir réaliser une implantation efficace. Pour décrire les lois de commandes, l'automaticien utilise généralement une représentation graphique proche des graphes flot de données (schéma-bloc). Pour décrire les discontinuités dans le comportement du système (changement de mode de fonctionnement) il utilise généralement une représentation graphique de type flot de contrôle (automates à états finis, GRAFCET, réseaux de Pétri). La méthodologie AAA et le logiciel SynDEx qui la supporte permet facilement de spécifier les lois de commandes décrites par l'automaticien car elle est basée sur un formalisme de graphes de type flot de données. Dans cette thèse, nous avons étendue la méthodologie afin qu'elle permette aussi de spécifier facilement le changement de mode à l'aide de graphes d'automates. Nous définissons une transformation permettant, à partir d'une spécification composée de ces deux types de graphes, de générer un graphe flot de données équivalent dont l'avantage est de permettre d'exploiter le parallélisme disponible de l'architecture matérielle lors de l'implantation. Nous avons étendu la méthodologie AAA afin qu'elle permette aussi la spécification des contraintes temporelles multiples que l'implantation doit satisfaire. Pour réaliser cette implantation nous proposons une stratégie d'ordonnancement hors-ligne afin de garantir le respect des dépendances de données et afin de minimiser le surcoût de l'exécutif. Afin de tenir compte des contraintes de cadences et de latences sur les entrées/sorties, nous introduisons le minimum possible de préemption.

L'accès aux ressources reste ainsi séquentiel et il n'y a donc pas lieu de mettre en œuvre des techniques visant à résoudre le problème d'inversion de priorité. Le code que l'on peut générer avec cette stratégie d'ordonnancement est très simple.

Une version industrielle du CyCab est en cours de réalisation. La réalisation du prototype sur lequel nous avons travaillé n'a pas bénéficiée des extensions de la méthodologie AAA proposés dans cette thèse. Une version industrielle du CyCab est en cours de réalisation. Il serait intéressant de valider ces extensions sur cette nouvelle version du véhicule.

Lors de la modélisation, l'automaticien ignore les temps de calcul nécessaire à l'exécution des lois de commandes. Ces temps de calcul introduisent des retards dans les lois de commande. Généralement le seul moyen que l'automaticien a pour prendre en compte ces retards est de réaliser une première implantation, de l'exécuter et de l'observer afin d'ajuster les paramètres des lois de commandes ce qui nécessite l'itération complète du cycle de développement. La méthodologie AAA permet de prédire les temps de calcul de l'implantation qu'elle a calculée. Il serait donc intéressant dans ce contexte de permettre de "faire remonter"automatiquement ces retards dans la modélisation afin de permettre à l'automaticien d'affiner son modèle le plus tôt possible dans le cycle en vue de réduire le nombre d'itérations dans le cycle en V.

Lors de cette thèse, nous avons réalisé une interface permettant de spécifier un automate et de le transformer automatiquement en un graphe flot de donnée équivalent. Cette interface repose sur une version de SynDEx ne supportant pas la hiérarchie et dans laquelle l'exécution des opérations est conditionné sur la valeur de signaux booléens. Nous avons montré dans la thèse qu'il serait intéressant d'étendre ce conditionnement à la valeur de signaux de type entier afin de réduire les communications engendrées à l'exécution par le calcul de l'état de l'automate.

L'implantation d'une spécification multi-contrainte sur une architecture multiprocesseur est un problème complexe. Dans cette thèse nous proposons une solution qui, faute de temps, n'a pu être intégrée dans SynDEx. Pour réaliser cette intégration, il faudra formaliser l'heuristique d'ordonnancement proposée et l'implanter dans SynDEx, il faudra aussi modifier l'interface graphique pour que les différentes séquences et la préemption entre ces séquences puissent être affichées dans la prédiction temporelle, enfin le générateur de code devra être réalisé.

BIBLIOGRAPHIE

[1] *OSEK/VDX Operating System Specification v2.0*.

[2] *Algebraic Automata Theory*. – Cambridge University Press, 1982.

[3] *Proceedings of the International Workshop on Open Systems in Automotive Networks*. – University of Karlsruhe, Germany, October 1995.

[4] Real time faq, 1998. `http:www.realtime-info.be/encyc/techno/publi/faq/rtfaq.htm`.

[5] Abdou (Sofiane). – *Spécification, vérification et implémentations de missions appliquées à des véhicules automatiques*. – Thèse de PhD, Institut National Polytechnique de Grenoble, 1997.

[6] Accart Hardin (Thérèse) et Donzeau-Gouge Viguie (Véronique). – *Concepts et outils de programmation, le style fonctionnel, le style impératif avec CAML et Ada*. – informatique intelligence artificielle.

[7] Amagbegnon Tochéou (Pascalin), Le Guernic (Paul), Marchand (Hervé) et Rutten (Eric). – *The SIGNAL data-flow methodology applied to a production cell*. – Research Report n2522, INRIA, March 1995.

[8] Ancourt (C.), Barthou (D.), Guettier (C.), Irigoin (F.), Jeannet (B.), Jourdan (J.) et Mattioli (J.). – Automatic data mapping of signal processing applications. *In : IEEE International Conference on Application Specific Systems, Architectures, and Processors*, pp. 350–362. – Zurich, Suisse, Juillet 1997.

[9] Astrom (Karl J.) et Wittenmark (Bjorn). – *Computer Controlled Systems: Theory and Design*. – Prentice-Hall International, 1984.

[10] Baille (Gérard), Garnier (Philippe), Mathieu (Hervé) et Pissard-Gibollet (Roger). – *Le CyCab de l'INRIA Rhône-Alpes*. – Rapport technique n0229, INRIA, Avril 1999.

[11] Beauvais (J-R.), Houdebine (R.), Le Guernic (P.), Rutten (E.) et Gautier (T.). – *A Translation of Statecharts and Activitycharts into Signal Equations*. – Research Report n3397, INRIA, Avril 1998.

[12] Bensoussan (Stéphane). – Téléopération d'un véhicule urbain. – rapport interne.

[13] Benveniste (Albert) et Berry (Gérard). – The synchronous approach to reactive and real-time systems. *In : Proceedings of the IEEE*.

[14] Benveniste (Albert), Caspi (Paul), Le Guernic (Paul) et Halbwachs (Nicolas). – *Data-Flow synchronous Languages*. – Research Report n2089, INRIA, october 1993.

[15] Berry (Gérard). – *The Esterel V5 Language Primer Version 5.21, release 2.0*. – SIMULOG.

[16] BOSCH. – *CAN Specification V2.0*.

[17] Boussinot (Frédéric) et De Simone (Robert). – The esterel language. *In : Proceedings of the IEEE*, pp. 1293–1304.

[18] Budde (Reinhard). – Esterel applied to the case study production cell. – http://set.gmd.de/EES/.

[19] Buhler (Hansruedi). – *Réglages échantillonnés*. – Presses Polytechniques Romandes, 1982.

[20] Buhler (Hansruedi). – *Conception de systèmes Automatiques*. – Presses Polytechniques Romandes, 1988.

[21] Carroll (J.) et Long (Darrell). – *Theory of Finite Automata*. – Prentice-Hall International, 1989.

[22] Carroll (John) et Long (Darrell). – *Theory of finite automata with an introduction to formal languages*. – Prentice-Hall International, 1989.

[23] The Common Format of Synchronous Languages. – *The Declarative Code DC*, October 1995. Version 1.0.

[24] Daviet (Pascal) et Parent (Michel). – Platooning technique for empty vehicles distribution in the praxitele project. *In : Proceedings of the 4th IEEE Mediterranean Symposium on New Directions in Control and Automation*. – Maleme, Krete, GREECE, June 1996.

[25] Dejenidi (Rachid), Lavarenne (Christophe), Nikoukhah (Ramine), Sorel (Yves) et Steer (Serge). – From hybrid system simulation to real-time implementation. *In : 11th European Simulation Symposium and Exhibition*. – Erlangen-Nuremberg, octobre 1999.

[26] Delacroix (J.). – Nouvelles approches de l'ordonnancement temps réel. *In : Ecole d'été Temps réel'97, Applications, Réseaux et Systèmes*. – ENSMA, Poitiers-Futuroscope, 1997.

[27] Dias (Ailton), Lavarenne (Christophe), Akil (Mohamed) et Sorel (Yves). – Optimized implementation of real-time image processing algorithms on field programmable gate arrays. *In : Fourth International Conference on Signal Processing*. – Beijing, China, octobre 1998.

[28] Dieulesaint (Eric) et Royer (D.). – *Automatique Appliquée : Systèmes Linéaires de Commande à Signaux Analogiques*. – MASSON, 1987.

[29] Dorseuil (A.) et Pillot (P.). – *Le Temps Réel en Milieu Industriel: Concepts, Environnements, Multitâches*. – DUNOD, 1991.

[30] Feautrier (P.). – Toward automatic distribution. *In : Parallel Processing Letters*.

[31] Gaudel (Marie-Claude), Marre (Bruno), Schlienger (Françoise) et Bernot (Gilles). – *Précis de génie logiciel*. – Masson, 1996.

[32] Gautier (Thierry), Le Guernic (Paul) et Maffeis (Olivier). – *For a New Real-Time Methodology*. – Research Report n2364, INRIA, 1994.

[33] Gecseg (F.). – *Products of automata*. – Springer-Verlag, 1986, *EATCS Monographs on Theoretical Computer Science*.

[34] Genest (Géraldine) et Chapellier (Marc). – *Étude et développement de véhicules intelligent*. – Rapport interne, INRIA, 1997.

[35] George (Laurent), Rivierre (Nicolas) et Spuri (Marco). – *Preemptive and Non-Preemptive Real-Time UniProcessor Scheduling*. – Research Report n2966, INRIA, September 1996.

[36] Girault (Alain), Lee (Bilung) et Lee (Edward A.). – Hierarchical finite state machines with multiple concurrency models. *In : Technical Memorandum UCB/ERL M97/57*.

[37] Grandpierre (thierry). – Spécifications en langage orienté objet pour le portage du noyau de syndex. – rapport interne.

[38] Halbwachs (N.). – *Synchronous programming of reactive systems.* – Kluwer Academic Pub., 1993.

[39] Harel (D.) et Pnueli (A.). – On the development of reactive systems. *In : Logics and Models of Concurrent Systems*, éd. par Springer-Verlag, pp. 477–498. – New York, k. r. apt édition, 1985.

[40] Hermant (Jean-François), Leboucher (Laurent) et Rivierre (Nicolas). – *Real-Time Fixed and Dynamic Priority Driven Scheduling Algorithms: Theory and Experience*. – Research Report n3081, INRIA, December 1996.

[41] home page (ASCET-SD). – http://www.etas.de/en/products/embedded_control/ascet-sd/.

[42] home page (Scilab). – http://www-rocq.inria.fr/scilab/.

[43] home page (The MathWorks). – http://www.mathworks.com/.

[44] home page (The MATRIXX). – http://www.isi.com/products/matrixx/.

[45] Kung (S.Y.). – *VLSI Array Processors*. – Englewood Cliffs NJ, Prentice Hall, 1988, *Information and System Sciences Series*. ISBN 0-13-942749-X.

[46] Lamport (Leslie). – Time, clocks, and the ordering of events in a distributed system. *In : Communications of the ACM*.

[47] Landau (Ioan Doré). – *Identification et commande des systèmes*. – Hermes, 1988.

[48] Lavarenne (Christophe) et Sorel (Yves). – Modèle unifié pour la conception conjointe logiciel-matériel. *Traitement du Signal*, vol. 14, n6, 1997.

[49] Le Guernic (Paul) et Gautier (Thierry). – *Data-flow to Von Neumann: the SIGNAL approach*. – Research Report n1229, INRIA, May 1990.

[50] Le Lann (Gérard). – *Critical Issues for the Development of Distributed Real-time Computing Systems*. – Research Report n1274, INRIA, August 1990.

[51] Leguernic (P.), Leborgne (M.), Gautier (T.) et Lemaire (C.). – *Programming real-time applications with SIGNAL*. – Research report, INRIA, June 1991. Research Report.

[52] Lenoir (Jacques). – Les outils de conception système du logiciel enfoui. *Electronique*, no89, Février 1999.

[53] Liu (C. L.) et W. (Layland James). – Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery*, vol. 20, n1, January 1973, pp. 46–61.

[54] Liu (Zhen) et Sanlaville (Eric). – *Preemptive scheduling with variable profile, precedence constraints and due dates*. – Research Report n1622, INRIA, Februar 1992.

[55] Maffeis (Olivier), Morley (Matthew) et Poigné (Axel). – The synchronous approach to designing reactive systems. – http://set.gmd.de/EES/.

[56] Maffeis (Olivier) et Poigné (Axel). – *Synchronous Automata for Reactive, Real-Time or Embedded Systems*. – Arbeitspapiere n967, Sankt Augustin, Germany, GMD SET - EES, January 1996.

[57] Mahé (Thierry). – L'automobile préfère le bus. *Industries et techniques*, vol. 747, Mars 1994, pp. 70–71.

[58] Maraninchi (F.). – The argos language: Graphical representation of automata and description of reactive systems. *In : IEEE Workshop on Visual Languages*. – Kobe, Japan, October 1991.

[59] Maraninchi (F.) et N. (Halbwachs). – Compiling argos into boolean equations. *Lecture Notes in Computer Science*, vol. 1135, September 1996, pp. 72–90.

[60] Marchand (Hervé), Rutten (Eric) et Samaan (Mazen). – *Specifying and verifying a transformer station in Signal and SignalGTi*. – Research Report n2521, INRIA, March 1995.

[61] Maret (Louis). – *Régulation Automatique : Systèmes analogiques*. – Presses Polytechniques Romandes, 1987.

[62] Mathai (Joseph). – *Real-time Systems: Specification, Verification and Analysis*. – Prentice Hall, 1996.

[63] Mead (C.A.) et Conway (L.A.). – *Introduction to VLSI systems*. – Addison-Wesley, 1980.

[64] Ogata (Katsuhiko). – *Discrete-Time Control Systems*. – Prentice-Hall International Editions, 1987.

[65] Parent (M.), Benejam-François (E.) et Hafez (N.). – Praxitèle: a new public transport with self-service electric cars. *In : ISATA Congress*. – Florence, Italy, June 1996.

[66] Parent (M.) et Texier (P.Y.). – Le transport public individuel. *In : Colloque SATCAR*. – Clermont-Fd, 1993.

[67] Perez (J.P.). – *Systèmes Temps Réel: Méthodes de Spécification et de Conception*. – DUNOD, 1990.

[68] Phillips (Charles L.) et Troy (Nagle H.). – *Digital Control System : Analysis and Design*. – Prentice-Hall International Editions, 1984.

[69] Pratt (V.). – Modeling concurrency with partial orders. *International Journal of Parallel Programming*, vol. 15, n1, 1986.

[70] Ramamritham (Krithi), Stankovic (John A.) et Zhao (Wei). – Distributed scheduling of tasks with deadlines and resource requirements. *Advances in Real-Time Systems*, pp. 196–209.

[71] René (Seindal). – Gnu m4, a powerful macro processor. – `http://www.cs.wisc.edu/csl/texihtml/m4-1.4/m4_toc.html`.

[72] Richard (Martin). – *Etude de la conjonction des approches asynchrone et synchrone dans les langages réactifs : Application à Electre*. – Thèse de PhD, Ecole Centrale de Nantes, 1994.

[73] Roux (Olivier). – *Electre : expression et modélisation du comportement de processus temps-réel répartis communicants*. – Thèse de PhD, Ecole Nationale Supérieure de Mécanique, 1985.

[74] Rutten (Eric) et Le Guernic (Paul). – *Sequencing Data Flow Tasks in SIGNAL*. – Research Report n2120, INRIA, November 1993.

[75] Sarkar (V.) et Hennessy (J.). – Compile-time partioning and scheduling of parallel programs. *In : Symp. Compiler Construction*, éd. par ACM Press (New York), pp. 17–26.

[76] Sceptre (Projet). – *Proposition de Standard de Noyau d'Exécutif Temps Réel*. – Rapport du bni, Bureau d'Orientation de la Normalisation Informatique, Mai 1980.

[77] Sha (Lui) et Goodenough (John B.). – Real-time scheduling theory and ada. *Advances in Real-Time Systems*, pp. 119–128.

[78] Sha (Lui), Rajkumar (Ragunathan), Lehoczky (John) et Ramamritham (Krithi). – Mode change protocols for priority-driven preemptive scheduling. *Advances in Real-Time Systems*, pp. 97–118.

[79] Sha (Lui), Rajkumar (Ragunathan) et Lehoczky (John P.). – Priority inheritance protocols: An approach to real-time synchronization. *Advances in Real-Time Systems*, pp. 53–63.

[80] Siarry (Patrick). – *Automatique de base*.

[81] Simon (Daniel), Espiau (Bernard), Castillo (Eduardo) et Kapellos (Konstantinos). – *Computer-aided design of a generic robot controller handling reactivity and real-time control issues*. – Research Report n1801, INRIA, November 1992.

[82] Simulog. – *SyncCharts Tutorial and Reference Manual*.

[83] Sorel (Yves). – Massively parallel computing systems with real time constraints, the "algorithm architecture adequation". *In : Methodology Proc. of Massively Parallel Computing Systems Conference*. – Italy, 1994.

[84] Spuri (Marco). – *Analysis of Deadline Scheduled Real-Time Systems*. – Research Report n 2772, INRIA, January 1996.

[85] Stankovic (John A.) et Ramamritham (Krithi). – What is predictability for real-time systems. *Advances in Real-Time Systems*, pp. 26–33.

[86] Timmerman (Martin). – Rtos market survey preliminary results. *Real Time magazine*, march 1999.

[87] TNI. – *Sildex V5.0 Reference Manual*.

[88] Torngren (Martin). – Fundamentals of implementing real-time control applications in distributed computer systems. *In : Real-Time Systems*, pp. 219–250. – Kluwer Academic.

[89] Turing (A.M.). – On computable numbers, with an application to the entscheindungs problem. *In : Proc. London Math. Soc.*

[90] Vicard (Annie). – *Formalisation et optimisation des systèmes informatiques distribués temps réel embarqués*. – Thèse de PhD, Université Paris-XIII, Institut Galilée, 1999.

[91] webpage at INRIA (SynDEx). – `http://www-rocq.inria.fr/syndex/`.

[92] Zomaya (A.Y.). – *Parallel and distributed computing handbook*. – McGraw-Hill, 1996.