Partie II. Méthodologie formelle pour l'utilisation du parallélisme d'opérations des architectures multicomposants

Chapitre 5. Méthodologie AAA d'Adéquation Algorithme Architecture

Yves Sorel INRIA-Rocquencourt

Table des matières

1 Introduction

1.1 Contexte

Nous considérons ici des applications de contrôle-commande comprenant du traitement du signal et des images, soumises à des contraintes temps réel et d'embarquabilité devant avoir un comportement sûr, comme on en rencontre dans les domaines du transport (avionique, automobile, ferroviaire), des télécommunications etc. . .

La complexité de ces applications, au niveau des algorithmes, de l'architecture matérielle, et des interactions avec l'environnement sous contraintes temps réel, nécessite des méthodes pour minimiser la durée du cycle de développement, depuis la conception jusqu'à la mise au point, autant des prototypes que des "produits de série" dérivés de ces prototypes. Afin d'éviter toute rupture entre les différentes phases du cycle de développement et pour permettre des vérifications formelles et des optimisations, nous proposons une méthodologie formelle d'Adéquation Algorithme Architecture appelée AAA. Elle est fondée sur une approche globale, formalisant l'algorithme et l'architecture à l'aide de graphes et l'implantation, à l'aide de transformation de graphes. L'intérêt principal de ce modèle réside dans sa capacité à exprimer tout le parallélisme, concrètement décrit sous la forme de schémas-blocs, non seulement dans le cas de l'algorithme (graphe flot de données : exemple Simulink) et de l'architecture (interconnexion de composants : exemple VHDL structurel), mais aussi dans le cas de l'implantation de l'algorithme sur l'architecture (distribution et ordonnancement des calculs et des communications). Il permet d'effectuer des optimisations précises prenant en compte des ressources matérielles programmables (processeur) et non programmables (circuits intégrés spécifiques ASIC, FPGA), et de simplifier la génération automatique d'exécutifs pour les processeurs, les plus statiques possible afin de minimiser leur surcoût.

Il est nécessaire de préciser le sens que l'on donnera par la suite aux notions d'application, d'environnement, de système réactif, d'algorithme, d'architecture, d'implantation et d'adéquation.

Dans la méthodologie formelle AAA, une *application* est un système composé de deux sous-systèmes en interaction, un *environnement* physique à contrôler, et un contrôleur discret numérique. Ce dernier est un *système réactif*, c'est-à-dire qui réagit aux variations de l'état $U(t)$ du premier (entrées du contrôleur, discrétisées par l'intermédiaire de capteurs, t entier), en produisant une commande discrète $Y(t)$ pour l'environnement (par l'intermédiaire d'actionneurs, sorties du contrôleur) et un nouvel état interne $X(t)$ du contrôleur, tous deux *fonction* de l'ancien état $X(t-1)$ du contrôleur et de l'état $U(t)$ de l'environnement ($U(t)$, $X(t)$ et $Y(t)$ peuvent être des vecteurs) :

$$(Y(t),X(t)) = f(U(t),X(t-1)) \tag{1}$$

Le système réactif se décompose en une partie matérielle, dont la structure est ci-après dénommée *architecture*, et une partie logicielle, dont la structure est ci-après dénommée *algorithme*.

Une architecture est dite *multicomposant* car sa structure offrant du *parallélisme effectif* comprend en général des capteurs et des actionneurs, des composants programmables (processeurs RISC, CISC, DSP, microcontrôleurs) et des composants non programmables (circuits intégrés spécifiques ASIC éventuellement reconfigurables FPGA), inter-connectés par des moyens de communication. La multiplicité des composants peut être nécessaire pour satisfaire un besoin de puissance de calcul, de modularité ou de localisation des capteurs et des actionneurs par rapport aux composants afin de réduire les distances de transfert des signaux analogiques sensibles aux parasites et émissions électromagnétiques, de réduire la bande passante nécessaire pour transférer des signaux à distance grâce aux pré-traitements locaux, et plus généralement de réduire le câblage grâce à l'uniformisation des signaux numériques permettant leur multiplexage.

Un algorithme est le résultat de la transformation d'une spécification fonctionnelle, plus ou moins formelle, en une spécification logicielle adaptée à son traitement numérique par un calculateur. Plus précisément un algorithme, dans l'esprit de Turing [1], est une *séquence finie d'opérations* (réalisables en un temps fini et avec un support matériel fini). Ici, on étend la notion d'algorithme pour prendre en compte d'une part l'aspect *infiniment répétitif* des applications réactives et d'autre part l'aspect *parallèle* nécessaire à leur implantation distribuée. Le nombre d'interactions effectuées par un système réactif avec son environnement n'étant pas borné a priori, il peut être considéré infini. Cependant, à chaque interaction, le nombre d'opérations (nécessaires au calcul d'une commande en réaction à un changement d'état de l'environnement) doit rester borné, parce que les durées d'exécution sont bornées par les contraintes temps réel. Mais au lieu d'un ordre total (séquence) sur les opérations, on préfère un *ordre partiel*, établi par les dépendances de

données entre opérations, décrivant un *parallélisme potentiel* inhérent à l'algorithme, à ne pas confondre avec le parallélisme effectif du calculateur. Cependant, comme on le verra au chapitre 2.3.2 le parallélisme potentiel en relation avec la granularité de l'algorithme, doit être plus important que le le parallélisme effectif, mais dans un rapport qui ne soit pas tout de même trop grand afin de ne pas rendre prohibitif la recherche d'une implantation optimisée. Le terme *opération* correspond ici à la notion de fonction dans la théorie des ensembles, on le distingue volontairement de la notion d'*opérateur*, qui a pour nous un autre sens lié aux aspects implantation matérielle (cela sera précisé au chapitre 2.2.1). Chaque opération est atomique dans le sens où c'est une séquence finie d'instructions qui ne peut pas être distribuée pour être exécutée sur plusieurs calculateur, nous ne considérerons donc ici que le niveau *parallélisme d'opérations*. L'algorithme sera codé, à différents niveaux, par des langages plus ou moins éloignés du matériel. Comme on peut obtenir, pour un algorithme donné, de nombreux codages différents selon le langage et l'architecture choisis, on préfère utiliser ce terme générique d'algorithme indépendant des langages et des calculateurs plutôt que celui de programme ou logiciel.

1.2 Objectifs

L'implantation consiste à mettre en œuvre l'algorithme sur l'architecture, c'est-à-dire à allouer les ressources matérielles de l'architecture aux opérations de l'algorithme, puis à compiler, charger, et enfin lancer l'exécution du programme correspondant sur le calculateur, avec dans le cas des processeurs le support d'un système d'exploitation (souvent appelé exécutif quand il offre des services temps réel) dont le surcoût ne doit pas être négligé.

Enfin, l'*adéquation* consiste à mettre en correspondance de manière efficace l'algorithme et l'architecture pour réaliser une implantation optimisée. On utilisera par abus de langage dans la suite, cette notion d'implantation optimisée bien qu'on ne puisse pas garantir l'obtention d'une solution optimale pour ce type de problème. On se contentera donc d'une solution approchée obtenue rapidement, plutôt que d'une solution exacte obtenue dans un temps rédhibitoire à l'échelle humaine à cause de la complexité combinatoire exponentielle de la recherche de la meilleure solution. Brièvement, car cela sera développé au chapitre 2.3.3, on va rechercher parmi toutes les implantations que l'on pourrait faire d'un algorithme sur une architecture donnée, une implantation particulière que l'on considérera comme optimisée en fonction d'un objectif que l'on s'est fixé. Plus globalement on va devoir mettre en place un processus itératif consistant à influencer l'architecture par l'algorithme, et vice-versa à influencer l'algorithme par l'architecture.

2 Méthodologie AAA

2.1 Modèle d'algorithme

2.1.1 Graphe flot de contrôle, graphe flot de données

Il existe deux approches principales pour spécifier un algorithme : celle flot de contrôle et celle flot de données. Dans les deux cas on peut modéliser l'algorithme avec un graphe orienté acyclique [2] souvent appelé DAG (Directed Acyclic Graph). C'est la sémantique des arcs et de l'exécution des sommets qui feront les différences.

Dans un graphe flot de contrôle chaque sommet représente une opération qui consomme et produit des données dans des variables pendant son exécution, et chaque arc représente une précédence d'exécution. L'ensemble des arcs définit un ordre total d'exécution sur les opérations. Un arc est un contrôle de séquencement qui correspond soit à un branchement inconditionnel (utilisé pour spécifier une itération finie ou infinie d'une opération), soit à un branchement conditionnel (après évaluation d'une condition). La notion d'itération, inhérente à l'approche flot de contrôle, correspond à la répétition temporelle (par opposition à spatiale comme on le verra plus loin) d'une opération ou d'un sous-graphe d'opérations. Il n'y a pas dans ce cas de relation entre l'ordre sur les opérations à exécuter et l'ordre dans lequel ces opérations lisent ou écrivent les données. Un organigramme est un exemple classique de graphe flot de contrôle habituellement utilisé avant d'écrire un programme dans un langage impératif (par exemple C, ou FORTRAN). Si on veut faire apparaître du parallélisme potentiel dans l'approche flot de contrôle, il faut mettre en parallèle plusieurs graphes flot de contrôle, ce qui conduit à un ordre partiel, et utiliser des variables communes que ces différents graphes se partageront pour établir les transferts de données nécessaires. Cela correspond à l'approche CSP

(Communicating Sequential Processes) de Hoare [3]. Il est important de noter que pour faire apparaître du parallélisme potentiel dans un graphe flot de contrôle qui n'en possède pas, on doit le transformer en un graphe flot de données par analyse des dépendances des données entre opérations à travers l'accès aux variables. Cependant cela peut s'avérer complexe principalement à cause du partage des variables et parfois peu efficace en ce qui concerne le niveau de granularité ainsi imposé. L'importance de la granularité sera vue plus loin au chapitre 2.3.2.

Dans la version basique d'un graphe flot de données [4] chaque sommet représente une opération qui consomme des données avant son exécution et produit des données après son exécution, introduisant ainsi un ordre entre la lecture des données sur toutes les entrées et l'écriture des données résultats sur toutes les sorties. Donc au niveau de la spécification d'un algorithme la notion de variable n'existe pas. Chaque hyper-arc (diffusion d'une donnée) représente une précédence d'exécution induite par une dépendance (transfert) de données entre une opération productrice et une ou plusieurs opérations consommatrices. Deux opérations, si elles n'ont pas à se transférer de donnée, ne sont pas connectées par un arc. L'ensemble des arcs définit donc un ordre patiel d'exécution sur les opérations traduisant naturellement du parallélisme potentiel. Un arc est donc aussi un contrôle de séquencement, la données qu'il véhicule est alors une donnée de contrôle utilisée pour gérer par exemple une itération ou du conditionnement. Il y dans ce cas une relation entre l'ordre sur les opérations à exécuter et l'ordre dans lequel ces opérations lisent ou écrivent les données.

Remarque : on a parlé plus haut d'hyper-graphes [2] orientés à cause des hyper-arcs orientés qui sont nécessaires pour spécifier de la diffusion de données, en effet on peut vouloir produire une donnéees qui sera consommée par plusieurs opérations. Ils correspondent à des n-uplets ordonnés plutôt qu'à des couples ordonnés comme c'est le cas des arcs simples.

En conclusion, dans le cas du flot de contrôle les données sont indépendantes du contrôle, c'est à l'utilisateur qui effectue la spécification de l'algorithme d'assurer l'ordre de consommation et de production des données à travers l'accès aux variables, ce qui peut conduire à des erreurs lorsque par exemple on cherche à réutiliser les variables. En revanche, dans le cas du flot de données, les données sont dépendantes du contrôle et l'ordre d'accès aux données est ainsi directement imposé par l'ordre d'exécution des opérations. Ce type de spécification d'algorithme est moins sujette à des erreurs, ce sera au compilateur des langages qui supportent ce modèle d'assurer l'ordre d'écriture et de lecture dans les variables, qui pourront ainsi être réutilisées sans erreurs. De plus, dans le cas des algorithmes de contrôle-commande, de traitement du signal et des images, il est fondamental de maîtriser l'ordre dans lequel on manipule les données, ce qui n'est pas le cas pour d'autres types d'algorithmes comme ceux par exemple que l'on utilise dans le domaine des bases de données ou d'autres domaines. Par exemple, lorsqu'on calcule un filtre numérique transversal, l'ordre d'arrivée des données dans le filtre et l'ordre de la production des données à la sortie du filtre doivent être cohérents entre eux.

Ainsi pour les deux types de modèles, graphes flot de contrôle mis en parallèle et graphes flot de données, on a étendu la notion initiale d'algorithme liée à un ordre total d'exécution sur les opérations, à un ordre partiel d'exécution. Dans la suite nous utiliserons systématiquement ce modèle étendu quand nous parlerons d'algorithme.

2.1.2 Graphe de dépendances de données factorisé conditionné

Pour toutes les raisons citées ci-dessus le modèle d'algorithme utilisé dans la méthodologie AAA [5] est une extension dans deux directions du modèle graphe flot de données décrit ci-dessus. D'une part nous avons besoin de prendre en compte l'aspect réactif de nos applications, et d'autre part nous avons besoin de pouvoir exprimer le maximum de parallélisme potentiel afin de pouvoir l'exploiter directement (par des transformations simples) quand cela est nécessaire. Par ailleurs nous généralisons la notion de conditionnement car cela est utile pour être compatible avec la sémantique la plus fine (notion d'absence de données) des langages synchrones dont nous parlerons plus loin.

L'exécution d'une application réactive correspond à l'itération infinie de l'équation (1) du contrôleur. Pour faire apparaître du parallélisme potentiel on peut transformer cette répétition temporelle infinie en une répétition spatiale infinie, ce qui permet de modéliser un algorithme par un un graphe de dépendances de données acyclique, résultat de la répétition spatiale infinie d'un sous-graphe (motif de la répétition) qui est un graphe flot de données correspondant à la décomposition de l'équation (1) en opérations.

Ces dernières sont les sommets du graphe, inter-connectés par des dépendances de données ou des dépendances de conditionnement, qui sont toutes deux les arcs du graphe. Chaque arc a pour origine la

sortie de l'opération produisant la donnée ou le conditionnement et pour extrémité(s) (il peut y en avoir plusieurs en cas de diffusion) les entrées des opérations les utilisant. L'exécution de toute opération est conditionnée, à chacune de ses répétitions, par la valeur de la donnée de conditionnement qu'elle reçoit sur son entrée particulière "d'activation". En pratique, ce graphe d'algorithme est spécifié en intention par l'utilisateur, sous forme *factorisée* (c'est-à-dire en ne spécifiant que le graphe flot de données motif de la répétition) non seulement au niveau de la répétition infinie, mais aussi aux niveaux, qui peuvent être imbriqués, des répétitions finies qui peuvent apparaître dans la décomposition de l'équation du contrôleur. Dans le cas de répétitions comportant des dépendances inter-répétitions, la factorisation génère des cycles *apparents*, que l'utilisateur doit marquer, au niveau des dépendances inter-répétitions, avec des sommets spéciaux "retard" (le z^{-1} des automaticiens et des traiteurs de signal). Ces retards stockent explicitement l'état de l'algorithme qu'il est important de bien maîtriser dans les algorithmes de contrôle-commande afin de leut assurer de bonnes propriétés (stabilité, commandabilité, observabilité).

Remarque : dans le cas d'un graphe flot de contrôle l'état devrait être extrait de l'ensemble des variables associées aux données, ce qui n'est généralement pas simple.

Afin de pouvoir exprimer le maximum de parallélisme d'opérations, habituellement appelé parallélisme de tâches (opérations différentes appliquées à des données différentes) ou de parallélisme de données (même opération appliquée à des données différentes), nous proposons de spécifier chaque itération finie de la même manière que l'itération infinie, sous la forme d'une répétition spatiale d'un sous-graphe, qui pourra être inversement transformée en une répétition temporelle si cela est nécessaire lors des optimisations ou de la génération d'exécutifs.

L'algorithme est donc modélisé par un *graphe de dépendances de données factorisé conditionné* [6], où chaque sommet "opération" est :

- soit un calcul, fini et sans effet de bord (les résultats du calcul en sorties de l'opération ne dépendent que des données en entrées de l'opération : pas d'état interne mémorisé entre différentes exécutions de l'opération, ni d'accès à des signaux externes de capteurs ou d'actionneurs),
- soit un délimiteur frontière de factorisation, marquant une dépendance de données entre des sommets appartenant à deux motifs ayant des facteurs de répétition différents, l'un "rapide" multiple de l'autre "lent",

et où chaque hyperarc est une dépendance de donnée entre une (et une seule) sortie d'une opération émettrice en amont, et une (des) entrée(s) d'une (de plusieurs) opération(s) réceptrice(s) en aval. On rappelle qu'il n'y a pas de notion de "variable" au sens des langages impératifs, ces derniers spécifiant un ordre total d'exécution sur les opérations, d'où doivent être extraites pour chaque donnée des dépendances écriture(w)/lecture(r), w/w, r/w et r/r dans les variables associées.

Les sommets frontière de factorisation peuvent être :

- soit un multiplexeur ("Fork"), où chaque répétition du sommet aval rapide consomme une partie différente du tableau produit par le sommet amont lent ; dans le cas de la répétition infinie, le multiplexeur est un capteur, opération sans entrée (source du graphe), produisant en sortie(s) une (des) valeur(s) représentant une acquisition partielle de l'état de l'environnement (une opération capteur peut comprendre si nécessaire des calculs et un état interne pour mettre en forme le signal brut acquis) ;
- soit un démultiplexeur ("Join"), où chaque répétition du sommet amont rapide produit une partie différente du tableau consommé par le sommet aval lent ; dans le cas de la répétition infinie, le démultiplexeur est un actionneur, opération sans sortie (puits du graphe), consommant en entrée(s) une (des) valeur(s) qu'elle utilise pour produire une commande agissant sur une partie de l'environnement (une opération actionneur peut comprendre si nécessaire des calculs et un état interne pour produire un signal brut de commande) ;
- soit un diffuseur ("Diff"), où chaque répétition du sommet aval rapide consomme la même valeur produite par le sommet amont lent ; dans le cas de la répétition infinie, le diffuseur est une constante, opération sans entrée (source du graphe), fournissant toujours la (les) même(s) valeur(s) en sortie(s) de l'opération ;
- soit un retard ("Iterate"), marquant une dépendance de donnée inter-répétition (effet mémoire, ou état, entre répétitions d'un motif), avec une seconde entrée "initiale" pour la première répétition, et (pour les répétitions finies) une seconde sortie "finale" pour la dernière répétition.

La durée d'exécution de toute opération est bornée (pas de boucles ou récursions potentiellement infinies, et pour un capteur ou un actionneur, pas d'attente indéfinie d'un signal externe de synchronisation).

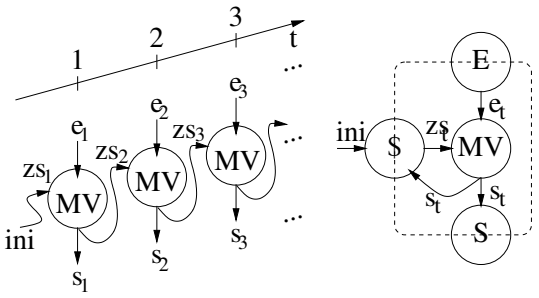

FIG. 1 – *Graphe d'algorithme (répétition infinie du produit matrice vecteur)*

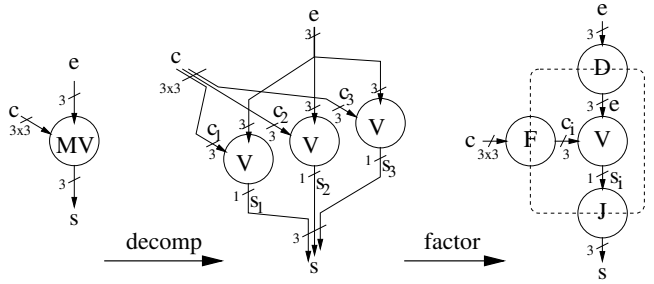

FIG. 2 – *Sous-graphe MV (3 répétitions finies de V)*

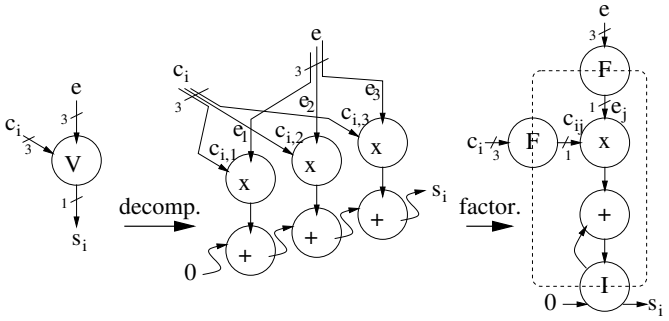

FIG. 3 – *Sous-graphe V (3 répétitions finies de multiplication-accumulation)*

La figure 1 présente le graphe d'algorithme effectuant un *produit matrice vecteur récursif* correspondant à la répétition infinie d'un produit matrice vecteur MV dont le vecteur à 3 éléments e_t est fourni à chaque acquisition par un capteur et la matrice 3x3 s_t est fournie via un retard \$ (dépendance inter-répétition infinie) par le résultat du produit matrice vecteur zs_{t-1} calculé lors de la répétition précédente $t-1$. Le résultat de chaque produit matrice vecteur est aussi transmis (diffusé) à un actionneur. La figure 2 présente le sous-graphe qui calcule un des produits matrice vecteur correspondant à trois répétitions du produits scalaires V. La figure 3 présente le sous-graphe qui effectue un produit scalaire correspondant à trois répétitions des opérations multiplication-accumulation. Ce sous-graphe utilise trois dépendances inter-répétitions finies pour effectuer l'accumulation à partir du résultat de la somme calculée à la répétition précédente.

Le graphe flot de données de l'algorithme peut être spécifié directement comme décrit ci-dessus à l'aide d'une interface graphique ou textuelle, ou bien déduit d'une spécification séquentielle ou CSP (Communicating Sequential Processes de Hoare) flot de contrôle, par analyse de dépendances de données entre les opérations à réaliser. Il peut aussi être produit par les compilateurs des langages synchrones [7] (Esterel, Lustre, Signal, à travers leur format commun "DC" qui est aussi un graphe flot de données). Cette approche présente l'intérêt de permettre d'effectuer des vérifications formelles en termes d'ordre sur les événements

qui entrent et sortent dans l'algorithme en utilisant principalement des techniques de type "model checking" [8] utilisant le plus souvent des BDD (Binary Decision Diagram) [9]. On peut ainsi vérifier des propriétés telles qu'un certain événement ne se produira jamais ou bien toujours après qu'un autre événement se soit produit. Ces vérifications, effectuées très tôt dans le cycle de développement, concrètement bien avant la phase de tests exécutés en temps réel, permettent d'éliminer un grand nombre d'erreurs concernant la logique de l'algorithme qui sont très difficiles à corriger au niveau de son exécution temps réel car elles demandent des moyens complexes de traçabilité.

2.2 Modèle d'Architecture

2.2.1 Graphe multicomposant

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") [10]. 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 graphe orienté, dont chaque sommet est un automate possédant des sorties (machine séquentielle) et chaque arc une connexion orientée entre deux sommets allant d'une sortie d'une machine séquentielle à une entrée d'une autre machine séquentielle [11]. Ce graphe correspond à un réseau d'automates [12]. Il y a cinq types de sommets : l'*opérateur* pour séquencer des opérations de calcul (séquenceur d'instructions), le *communicateur* pour séquencer des *opérations de communication* (canal DMA), le *bus/mux/démux* avec ou sans *arbitre* pour sélectionner, diffuser et éventuellement arbitrer des données, la mémoire pour stocker des données et des programmes. Il y a deux types de sommets mémoire : la mémoire RAM à accès aléatoire pour stocker les données ou programmes locaux à un opérateur, la RAM et la SAM à accès séquentiel pour les données communiquées entre opérateurs ou/et communicateurs. L'arbitre, quand il y en a un dans un bus/mux/démux/arbitre, est aussi un automate qui décide de l'accès aux ressources partagées que sont les mémoires. Les différents sommets ne peuvent pas être connectés entre eux de n'importe quelle manière, il est nécessaire de respecter un ensemble de règles. Par exemple deux opérateurs ne peuvent pas être connectés directement ensemble. Ils peuvent chacun être connecté à une RAM partagée ou à une SAM pour communiquer, en passant ou non par l'intermédiaire de communicateurs pour assurer le découplage entre calcul et communication. L'hétérogénéité ne signifie pas seulement que les sommets peuvent avoir chacun des caractéristiques différentes (par exemple durée d'exécution des opérations et taille mémoire des données communiquées), 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). Un processeur est décrit par un sous-graphe contenant un seul opérateur, une ou plusieurs RAM de données et de programme locaux. Un moyen de communication direct (sans routage) entre deux processeurs, est un sous-graphe contenant au moins une RAM (données communiquées) et des bus/mux/démux/arbitre, ou bien un sous-graphe linéaire composé au minimum des sommets (bus/mux/démux/arbitre, RAM, communicateur, RAM ou SAM, communicateur, RAM, bus/mux/démux/arbitre). Ce modèle permet de faire des spécifications plus ou moins détaillée d'une même architecture en fonction de la précision avec laquelle on souhaite faire l'implantation optimisée. Il faut tout de même noter que plus l'architecture sera détaillée plus le problème d'optimisation sera long à "résoudre".

La figure 4 présente le graphe correspondant à la modélisation détaillée du processeur de traitement du signal TMS320C40 de Texas Instrument. Ici comme comme toutes les connexions sont bi-directionnelles nous avons représentés chaque couple de flèche de sens opposés par un seul segment. Le CPU, son contrôleur mémoire et son unité de calcul sont modélisés par un opérateur. Comme il est capable d'accéder simultanément à deux mémoires internes et/ou externes modélisées par des sommets RAM ($R0$ et $R1$ pour les mémoires internes, R_{loc} et R_{glob} pour les mémoires externes éventuelles), l'opérateur est connecté à deux bus/mux/démux ($b7$ et $b8$) qui sélectionnent les mémoires. Comme ces mémoires sont aussi accessibles par chaque canal du DMA modélisé par un communicateur ($C1$ à $C6$), chaque communicateur est connecté

à ces mémoires internes par l'intermédiaire d'un bus/mux/démux/arbitre ($b9$) qui arbitre l'accès entre tous les communicateurs. Chaque port de communication point-à-point est modélisé par une SAM. Chaque SAM étant à la fois accessible par un canal DMA ou par le CPU, elles sont connectées d'une part à un communicateur et d'autre part à l'opérateur. Le bus/mux/démux $b10$ permet de sélectionner laquelle des SAM est accédée par l'opérateur. Les deux ports de mémoires externes permettent à l'opérateur et aux communicateurs d'accéder aux deux mémoires externes, il y a donc arbitrage que nous modélisons par deux sommets bus/mux/démux/arbitre ($b11$ et $b12$) entre les RAM externes, l'opérateur et les communicateurs. Les bus/mux/démux $b1$ à $b6$ modélise la capacité, pour chaque communicateur, d'accéder soit à une SAM, soit à la mémoire externe.

La figure 5 présente un exemple d'architecture basé sur quatre TMS320C40 communicant deux à deux par des SAM (liens point-à-point) et tous ensemble par une unique mémoire RAM partagée R_{glob}. Dans cet exemple chaque C40 est connecté à une mémoire RAM externe non partagée (R_{loc}).

En encapsulant dans un seul opérateur le graphe de la figure 4 représentant un TMS320C40, on peut réaliser un modèle beaucoup plus simple, mais moins précis, de l'architecture basée sur quatre TMS320C40 communicant deux à deux par des SAM (liens point-à-point) comme sur la figure 5. On obtient alors le modèle simplifié d'architecture présentée sur la figure 6.

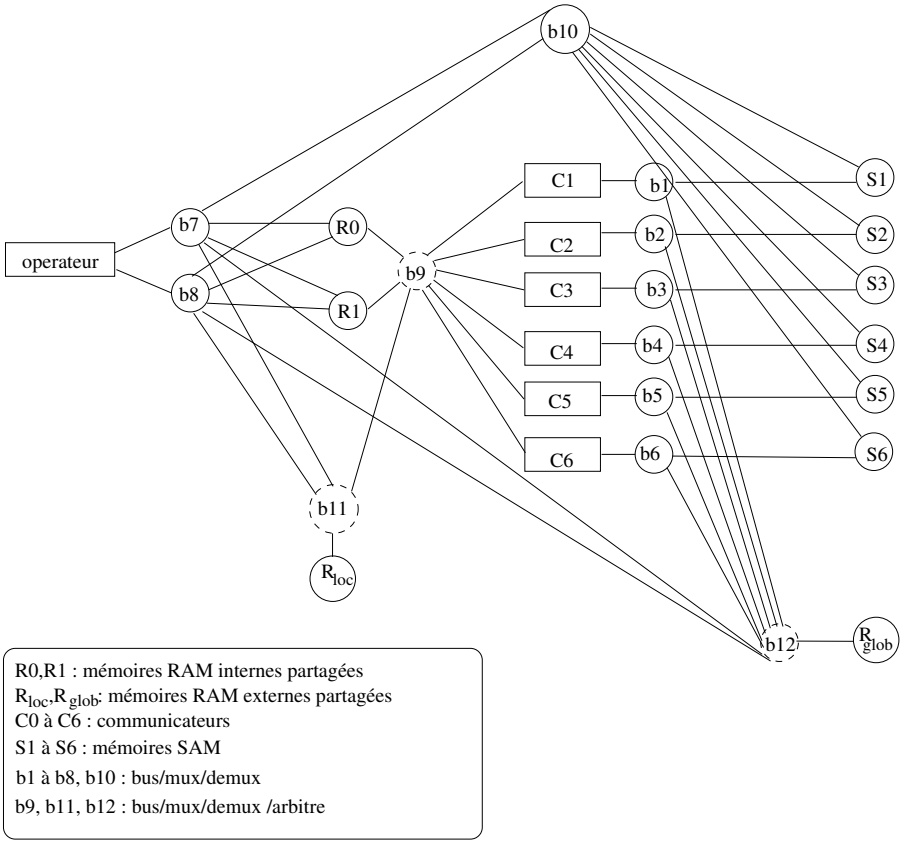

FIG. 4 – *Graphe d'architecture du TMS320C40*

2.2.2 Caractérisation d'architecture

Il s'agit de caractériser les composants et le réseau en fonction des contraintes temps réel et d'embarquabilité. Pour cela on associe à chaque opérateur et à chaque communicateur l'ensemble des opérations que chacun d'eux est capable de réaliser, et pour chaque opération sa durée, son occupation mémoire, la consommation associée à son exécution etc. Par exemple, l'opérateur unité centrale d'un processeur de traitement du signal est capable de réaliser, entre autres, une multiplication et une accumulation (instruction de base du processeur) en un cycle et une FFT (séquence d'instructions de base) en un certain nombre de cycles. De même, pour le DMA associé à un lien de communication d'un processeur de traitement du signal, on associera les opérations de transferts qu'il est capable de réaliser en fonction des types de données utilisés

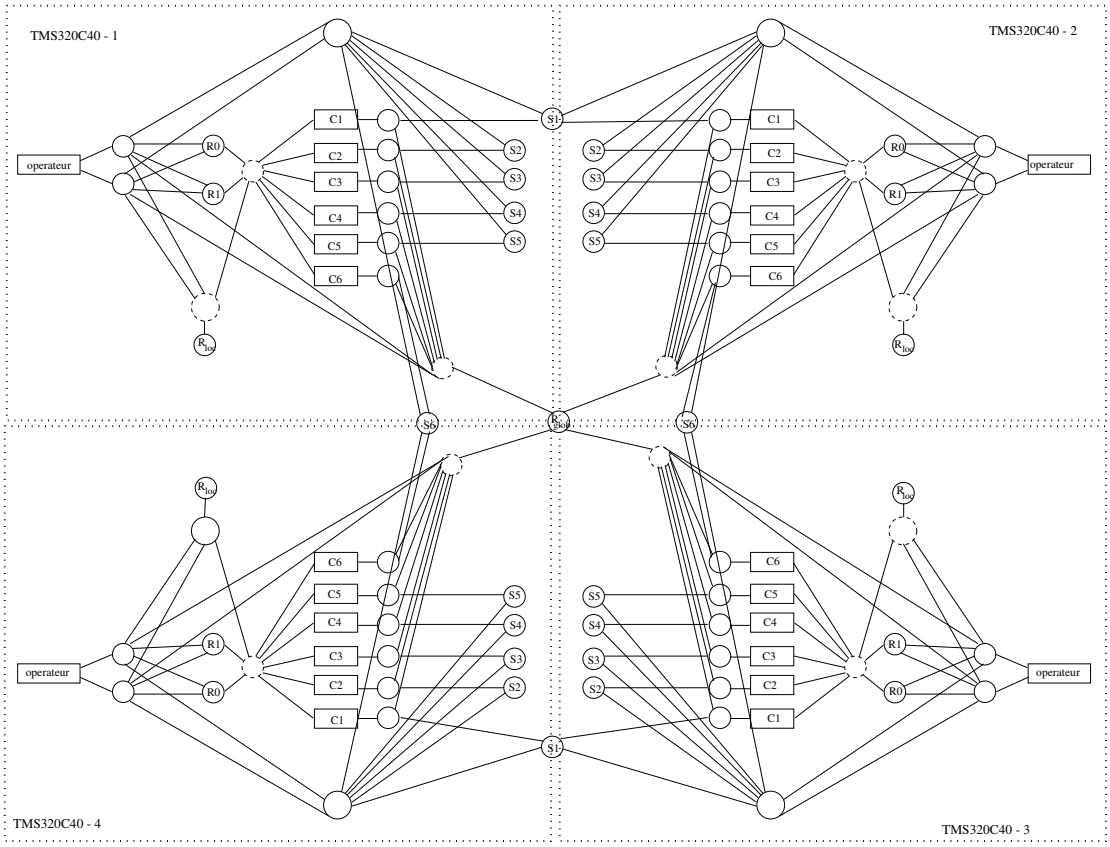

FIG. 5 – *Graphe d'architecture d'un quadri-processeur TMS320C40*

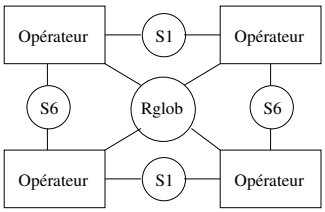

FIG. 6 – *Graphe d'architecture simplifiée d'un quadri-processeur TMS320C40*

(un entier peut prendre moins de temps à être transféré qu'un réel, ou qu'un tableau d'entiers).

L'automate arbitre dans un bus/mux/démux/arbitre joue un rôle crucial, il est caractérisé par une table de priorités et de bandes passantes qui a autant d'éléments que d'arcs qui sont connectés à ce bus/mux/démux/arbitre. Les valeurs de ces éléments sont utilisées pour déterminer lors de l'optimisation lequel des opérateurs et/ou des communicateurs en compétition, accède à la mémoire et avec quelle bande passante.

La caractérisation sera exploitée lors de l'optimisation comme on va le voir plus loin.

2.3 Modèle d'implantation

2.3.1 Distribution et ordonnancement

L'implantation d'un algorithme sur une architecture, consiste à réaliser, en tenant compte des contraintes, une distribution et un ordonnancement des opérations de l'algorithme sur l'architecture caractérisée comme indiqué dans le chapitre précédent. Il faut noter que ce qu'on appelle ici "distribution", est souvent appelé "placement" ou "répartition".

La distribution consiste tout d'abord à effectuer une partition du graphe de l'algorithme initial, en autant (ou moins) d'éléments de partition qu'il y a d'opérateurs dans le graphe de l'architecture. Il faut ensuite af-

fecter chaque élément de partition, c'est-à-dire chaque sous-graphe correspondant du graphe de l'algorithme initial, à un opérateur du graphe de l'architecture. On ne peut affecter qu'un seul type d'opération à un opérateur représentant un circuit intégré spécifique non programmable. Puis il faut affecter chaque transfert de données du graphe de l'algorithme reliant des opérations appartenant à des éléments de partition différents, à une *route*. Chaque route est un chemin dans le graphe de l'architecture qui connecte un couple d'opérateurs. Ce chemin est formé d'une liste de moyens de communication comportant chacun un couple de communicateurs si cette liste possède plus d'un élément. Les communicateurs assurent l'acheminement des données routées dans un processeur sans que le concours de l'opérateur soit nécessaire ("store and forward"). Une route peut se réduire à un seul moyen de communication si on a une communication directe entre deux opérateurs. On obtient l'ensemble des routes d'un graphe d'architecture donné en calculant la fermeture transitive de la relation "être connecté à" définie au chapitre 2.2.1. Il peut bien sûr y avoir plusieurs routes parallèles, de longueurs (nombre d'éléments la constituant) différentes, reliant deux opérateurs.

L'ordonnancement consiste, pour chaque élément de partition, à linéariser (rendre total) l'ordre partiel correspondant au sous-graphe associé. Il se peut que celui-ci soit déjà un ordre total. Cette phase est nécessaire car l'opérateur auquel on affecte un sous-graphe du graphe de l'algorithme initial est un automate, séquentiel par définition, dont le rôle est d'exécuter séquentiellement des macro-opérations (dont chacune est une séquence d'instructions de base).

L'ensemble de toutes les implantations possibles que l'on peut faire d'un algorithme donné sur une architecture donnée peut être mathématiquement formalisé en intention comme la composition de trois relations binaires : le routage, la distribution et l'ordonnancement, chacune d'elles mettant en correspondance deux couples de graphes (algorithme, architecture) [5][13].

Étant donnés un graphe d'algorithme et un graphe d'architecture, on comprend aisément qu'il existe un nombre fini [2], mais qui peut être très grand, de distributions et d'ordonnancements possibles. En effet, on peut effectuer plusieurs partitions du graphe de l'algorithme, en fonction du nombre d'opérateurs, et pour chaque sous-graphe affecté à un opérateur il y a plusieurs linéarisations possibles de ce sous-graphe. La première chose à faire consiste à éliminer toutes les distributions et les ordonnancements qui ne conserveraient pas les propriétés montrées lors de la spécification avec les langages synchrones. Pour cela, il faut préserver la fermeture transitive du graphe de l'algorithme. L'ordre partiel associé au graphe transformé du graphe de l'algorithme après la distribution et l'ordonnancement doit être compatible avec l'ordre partiel du graphe de l'algorithme initial. Nous expliquerons plus loin comment parmi ces distributions et ordonnancements valides on va élire une distribution et un ordonnancement particuliers afin d'obtenir une implantation optimisée. Ce choix se fait en fonction des contraintes temps réel et d'embarquabilité.

Notre modèle d'implantation peut être vu comme une extension du modèle classique RTL ("Register Transfer Level", niveau transfert de registres) [14], que nous qualifions de *Macro-RTL*. Une opération du graphe de l'algorithme correspond à une *macro-instruction* (une séquence d'instructions ou un circuit combinatoire) ; une dépendance de données correspond à un *macro-registre* (des cellules mémoire contigus 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.

2.3.2 Impact de la granularité du graphe de l'algorithme

Le choix de la granularité de décomposition de l'algorithme, aussi bien du point du vue des traitements que du point de vue des données, est du ressort de l'utilisateur. Ce choix n'est pas sans conséquences sur les performances des implantations qu'il permet. Des grains trop gros relativement à la masse totale des calculs, et/ou en nombre insuffisant relativement au nombre de ressources séquentielles de l'architecture, permettent rarement une distribution équilibrée de la charge (de calcul entre processeurs, et/ou de communication entre moyens de communication), conduisant à une efficacité médiocre. En revanche, le surcoût de durée d'exécution dû à l'encapsulation de chaque grain dépendant peu de la taille du grain, des grains très petits, en grand nombre, peuvent entraîner un surcoût total important. De plus, les processeurs modernes tirant leurs performances des caractéristiques non linéaires des caches et des pipelines, choisir une taille de grain suffisante permet de diminuer la marge d'incertitude entre le pire cas et le cas moyen de durée d'exécution. Cela permet aussi une meilleure portabilité entre processeurs hétérogènes.

2.3.3 Contraintes et optimisation

Il s'agit tout d'abord de respecter les contraintes temps réel qui sont de deux types : latence et cadence. La première contrainte concerne la durée d'exécution d'une réaction (c'est-à-dire la durée du chemin critique du graphe de l'algorithme, correspondant à l'équation 1), la seconde concerne la durée qui s'écoule entre deux réactions. On étend souvent cette notion de latence et de cadence, qui concerne tout le graphe de l'algorithme, au niveau de chacune de ses opérations, et ainsi il peut y avoir plusieurs contraintes de latence et de cadence. Pour les aspects embarquabilité, on cherche à minimiser le nombre de ressources, c'est-à-dire le nombre d'opérateurs et de moyens de communication.

2.4 Implantation optimisée : Adéquation

Le problème d'optimisation considéré ici, minimisation de ressources sous contraintes temps réel, fait partie de la classe des problèmes d'allocation de ressources, connus pour être NP-difficiles.

Comme cela avait été souligné lors de l'introduction du chapitre 1, bien que l'on parle d'implantation optimisée, dans le cas général c'est une solution approchée que l'on recherche et non une solution optimale que l'on ne peut obtenir dans un temps raisonnable que pour des cas très simples. Dès que le nombre de sommets des graphes de l'algorithme et de l'architecture est de l'ordre de la dizaine, de telles solutions exactes ne sont pas humainement envisageables. On utilise alors des heuristiques que l'on désire à la fois rapides et donnant des résultats proches de la solution optimale. Ces deux caractéristiques sont bien sûr contradictoires. Dans le cas des applications de contrôle-commande comprenant du traitement du signal et des images, il est intéressant de choisir des heuristiques rapides afin de pouvoir essayer de nombreuses variantes d'implantation en fonction du coût et de la disponibilité des composants et de tester rapidement l'impact de l'ajout de nouvelles fonctionnalités.

C'est pourquoi on privilégie dans un premier temps les heuristiques déterministes "gloutonnes" (c'est-à-dire sans remise en cause des solutions partielles intermédiaires conduisant à une solution finale ; elles sont aussi appelées "sans retour arrière") et plus particulièrement celles "de liste" car ce sont elles qui donnent le plus rapidement leur résultat tout en ayant une bonne précision [15]. Une solution obtenue à l'aide d'une heuristique gloutonne peut être améliorée de manière itérative en mettant en cause certains choix faits localement ou globalement lors de l'élaboration d'une solution partielle, selon des heuristiques dites "de voisinages" [16]. Ceci ralentit bien sûr le processus d'obtention d'une solution. Enfin, la solution finalement trouvée peut servir à son tour comme solution initiale d'une heuristique stochastique, dans laquelle le passage d'une solution à l'autre se fait selon une fonction probabiliste. Ces heuristiques donnent leurs résultats beaucoup moins rapidement (en fait elles peuvent durer très longtemps) mais elles sont plus précises car elles permettent d'éviter les minima locaux ce qui n'est pas le cas des heuristiques déterministes citées précédemment[5].

L'utilisateur peut aussi chercher à améliorer les résultats de l'heuristique en restreignant, au moyen de contraintes de distribution, l'espace des solutions qu'elle a à explorer, en particulier en coupant les "fausses pistes" menant à des minima locaux sans intérêt.

Pour la latence, l'optimisation est basée sur des calculs de chemins critiques sur le graphe de l'algorithme, étiqueté par les durées des opérations et des transferts de données lorsqu'ils sont affectés respectivement aux opérateurs et aux routes. Les étiquettes sont déduites du modèle d'architecture caractérisé. Pour la cadence, l'optimisation est basée sur des recherches de boucles critiques, identifiées par les retards sur le graphe de l'algorithme, pour évaluer les possibilités de "pipe-line" en vue de faire du "retiming" en déplaçant des retards affectés à des mémoires.

On présente dans la suite les principes d'une heuristique de minimisation de la durée du chemin critique correspondant à une latence unique égale à la cadence. Afin de simplifier sa présentation elle ne prend pas en compte le conditionnement, la capacité des mémoires, et traite chaque motif répétitif fini factorisé en bloc, comme s'il s'agissait d'une opération unique encapsulant les répétitions du motif. Ces points sont détaillés respectivement dans [5], [11] et [17]. Des travaux de recherche sont en cours pour d'une part prendre en compte plus finement les motifs répétitifs et la capacité des mémoires, et d'autre part pour prendre en compte dans les optimisations plusieurs contraintes de latence et de cadence et de surcroît relâcher l'hypothèse de non préemtion, ce qui est nécessaire pour améliorer la latence [18]. Cependant comme la préemption a un surcoût non négligeable dû aux changements de contextes, il est nécessaire de minimiser son utilisation, ce qui est un problème difficile.

2.4.1 Exemple d'heuristique d'adéquation

Voici les principes d'un exemple d'heuristique gloutonne dite de type "ordonnancement de liste" simple et performante [19]. La distribution et l'ordonnancement sont faits en même temps, on va tenter de construire un optimum global à partir d'optima locaux de la façon suivante.

L'heuristique itère sur un ensemble O_s d'opérations "ordonnançables" ; une opération non encore ordonnancée devient ordonnançable lorsque tous ses prédécesseurs, retards exclus, sont déjà ordonnancés ; pour un retard, il faut de plus que tous ses successeurs soient déjà ordonnancés. Initialement, O_s comprend donc les capteurs ainsi que les opérations qui n'ont que des retards pour prédécesseurs.

À chaque étape principale de l'heuristique, une opération ordonnançable o et un opérateur p sont choisis (le processus du choix est détaillé ci-dessous) et o est ordonnancée sur p, ce qui a pour conséquence que certains des successeurs de o deviennent à leur tour ordonnançables. Cette étape principale est répétée jusqu'à ce que O_s soit vide, ce qui n'arrive qu'après une exploration complète de la relation d'ordre partiel "est successeur" entre les opérations, c'est-à-dire après que toutes les opérations aient été ordonnancées, dans un ordre compatible avec cette relation d'ordre.

"Ordonnancer" une opération o sur un opérateur p consiste à modifier le graphe de l'algorithme d'une part en ajoutant une "dépendance de contrôle" (relation d'ordre d'exécution) entre la dernière opération ordonnancée sur p, avant o, et o, et d'autre part, pour chaque prédécesseur o' de o, s'il est ordonnancé sur un opérateur $p' \neq p$, en choisissant une route r connectant p et p' (chemin dans le graphe de l'architecture), en insérant entre o' et o une opération de communication pour chacun des moyens de communication composant r, et en ordonnançant chacune de ces communications c sur le moyen de communication m correspondant. "Ordonnancer" une communication c sur un moyen de communication m consiste de même à modifier le graphe de l'algorithme en ajoutant une dépendance de contrôle entre la dernière opération de communication ordonnancée sur m, avant c, et c. On notera que la relation d'ordre d'exécution est totale (séquencement) sur chaque opérateur et sur chaque moyen de communication, et que par construction elle ne peut pas être en contradiction avec la relation d'ordre partiel du graphe de l'algorithme (une opération ne peut pas être ordonnancée qu'après ses prédécesseurs), ce qui garantit une exécution sans interblocage. On notera aussi que l'ordre d'exécution des opérations, entre opérateurs et entre moyens de communication, reste partiel, permettant des exécutions parallèles et des recouvrements calculs-communications.

L'ensemble des modifications du graphe de l'algorithme, conséquentes d'une part au choix d'une opération ordonnançable et d'un opérateur, et d'autre part aux choix des routes pour les dépendances de données inter-opérateur (et à l'ordre dans lequel ces choix sont faits), constituent une transformation "élémentaire" du graphe de l'algorithme. Il faut composer autant de ces transformations élémentaires qu'il y a d'opérations dans le graphe de l'algorithme pour obtenir une implantation. Le but de l'heuristique étant de minimiser la durée du chemin critique du graphe de l'algorithme implanté, la fonction de coût dirigeant les choix de l'heuristique est basée sur les éléments intervenant dans le calcul du chemin critique.

2.4.2 Fonction de coût de l'heuristique

La fonction de coût de l'heuristique est exprimée en termes des dates de début et de fin d'exécution des opérations, elles-mêmes exprimées récursivement en fonction des durées d'exécution des opérations et des communications. Notons $\Delta(o,p)$ la durée d'exécution caractéristique d'une opération (resp. communication) o de *Gal* (le graphe de l'algorithme) par un opérateur p (resp. moyen de communication) de *Gar* (le graphe de l'architecture), et $\Gamma(o)$ (resp. $\Gamma^-(o)$) l'ensemble des successeurs (resp. prédécesseurs, relativement aux dépendances de données) de o. Pour chaque opération ordonnançable, et pour chaque opérateur sur lequel on peut l'ordonnancer, on peut calculer R la durée du chemin critique du graphe de l'algorithme, $S(o)$ et $E(o)$ (resp. $S^-(o)$ et $E^-(o)$) ses dates de début et de fin d'exécution "au plus tôt mesurées depuis le début"

(resp. "au plus tard mesurées depuis la fin"), et $F(o)$ sa marge d'ordonnancement :

$$\begin{array}{rcl} S(o) & = & \max\limits_{x\in\Gamma^-(o)} E(x) \quad (\text{ou } 0 \text{ si } \Gamma^-(o)=\emptyset) \\ E(o) & = & S(o)+\Delta(o,p) \\ E^-(o) & = & \max\limits_{x\in\Gamma(o)} S^-(x) \quad (\text{ou } 0 \text{ si } \Gamma(o)=\emptyset) \\ S^-(o) & = & E^-(o)+\Delta(o) \\ R & = & \max\limits_{o\in Gal} E(o) = \max\limits_{o\in Gal} S^-(o) \\ F(o) & = & R-E(o)-E^-(o) \end{array}$$

On notera la symétrie des formules : ces dates sont mesurées relativement à des directions et des origines de temps opposées : $\min\limits_{o\in Gal} S(o) = 0 = \min\limits_{o\in Gal} E^-(o)$ par définition ; dans la littérature [20], $ASAP = S$ et $ALAP = R - S^-$.

Lorsque l'heuristique considère une opération o, chaque prédécesseur o' de o est déjà ordonnancé (c'est-à-dire que l'opérateur ou le moyen de communication qui exécute o' est déjà choisi), mais aucun successeur de o n'est encore ordonnancé, donc dans le calcul de E^- et de S^-, Δ doit pouvoir être définie indépendamment de tout opérateur. On définit la durée d'exécution estimée d'une opération o non encore ordonnancée comme la moyenne des durées d'exécution de o sur l'ensemble $K(o)$ des opérateurs capables d'exécuter o :

$$K(o) = \{p \in Gar | \Delta(o,p) \text{ est définie}\}$$

$$\Delta(o) = \frac{1}{\text{Card}(K(o))} \sum_{p\in K(o)} \Delta(o,p)$$

Lorsqu'une opération de calcul o est ordonnancée sur un opérateur p, elle l'est après toutes celles déjà ordonnancées sur p, sa durée d'exécution change de $\Delta(o)$ à $\Delta(o,p)$, et pour chaque prédécesseur o' de o ordonnancé sur un autre opérateur que p, des opérations de communication doivent être insérées entre o' et o et doivent être ordonnancées sur des moyens de communication. En conséquence, $S(o)$ et le chemin critique R peuvent prendre des valeurs plus grandes (ou identiques, mais en aucun cas plus petites) $S'(o)$ et R'.

La fonction de coût $\sigma(o,p)$, appelée "pression d'ordonnancement", est la différence entre la "pénalité d'ordonnancement" $P(o) = R' - R$ (allongement du chemin critique dû à cette étape de l'heuristique d'ordonnancement) et la marge d'ordonnancement $F(o)$ (marge avant allongement du chemin critique), soit :

$$\sigma(o,p) = P(o) - F(o) = S'(o) + \Delta(o,p) + E^-(o) - R$$

Par rapport à $F(o)$ qui est classiquement prise comme fonction de coût, $\sigma(o,p)$ est une version améliorée car, $S'(o)$ croissant, lorsque o devient critique, $F(o)$ s'arrête de décroître pour rester nulle, alors que $P(o)$ jusqu'alors nulle commence à croître, donc $\sigma(o,p)$ croît continûment. On notera de plus que si $F(o)$ dépend de R', qui est différent pour chacune des transformations élémentaires possibles à une étape principale de l'heuristique, $\sigma(o,p)$ ne dépend que de R qui reste le même pour chacune de ces transformations, ce qui permet de l'éliminer, ainsi que son calcul coûteux, lors de la comparaison entre les pressions d'ordonnancement des transformations élémentaires d'une même étape principale de l'heuristique.

2.4.3 Choix d'une opération et d'un opérateur

Le "meilleur opérateur" d'une opération ordonnançable o, noté $p_m(o)$, est soit celui où elle est contrainte par l'utilisateur à être exécutée, soit celui lui procurant la plus petite pression d'ordonnancement (c'est-à-dire le plus de marge d'ordonnancement ou le moins d'allongement du chemin critique après ordonnancement ; si plusieurs opérateurs répondent à ce critère, l'un d'eux est choisi au hasard) :

$$\exists p_m(o) \quad | \quad \sigma(o,p_m(o)) = \min_{p\in Gar} \sigma(o,p)$$

Par contre, l'opération ordonnançable la plus urgente à ordonnancer, sur son meilleur opérateur, est celle qui présente la plus grande pression d'ordonnancement, sauf si sa date de début d'exécution est postérieure

à la date de fin d'exécution d'une autre opération ordonnançable, qui peut donc s'exécuter avant. Pour cette raison, l'ensemble des opérations ordonnançables O_s est en pratique restreint à :

$$O'_s = \{o' \in O_s \quad | \quad S_m(o') < \min_{o \in O_s} E_m(o)\}$$

où $S_m(o)$ et $E_m(o)$ sont les valeurs respectives de $S(o)$ et $E(o)$ pour o ordonnancée sur son meilleur opérateur $p_m(o)$. L'opération choisie à chaque étape principale de l'heuristique est donc o_m telle que :

$$\exists o_m \quad | \quad \sigma(o_m, p_m(o_m)) = \max_{o \in O'_s} \sigma(o, p_m(o))$$

(si plusieurs opérations répondent à ce critère, l'une d'elles est choisie au hasard) et l'opérateur choisi est $p_m(o_m)$, le meilleur opérateur de o_m, sur lequel o_m est ordonnancée.

2.4.4 Création et ordonnancement des communications

Lorsqu'une opération o est ordonnancée sur un opérateur p, pour chaque dépendance de donnée inter-opérateur, entre o et un de ses prédécesseurs $o' \in \Gamma^-(o)$ ordonnancé sur un opérateur $p' \neq p$, il faut choisir une route pour transférer les données produites par o' dans la mémoire de p', à travers les moyens de communication et les mémoires des opérateurs intermédiaires de la route, jusque dans la mémoire de p pour qu'elles y soient consommées par o.

Le nombre de routes possibles reliant p' et p pouvant être grand pour des architectures non triviales, le choix exact de la meilleure route (telle que la date de fin de la dernière opération de communication sur la route soit minimale) est trop coûteux en pratique. Pour réduire ce coût, l'heuristique effectue un choix incrémental en s'aidant de tables de routage. Pour chaque opérateur, on calcule et mémorise dans une table de routage la distance (nombre minimal de moyens de communication) qui le sépare de chaque autre opérateur.

L'heuristique de routage incrémental choisit à chaque étape, parmi les moyens de communication connectés à l'opérateur courant (en partant de p'), ceux permettant d'atteindre les opérateurs voisins les plus proches de p, et choisit le moyen de communication m qui terminera au plus tôt l'opération de communication c ; c peut être une opération de communication précédemment ordonnancée sur m ayant déjà transféré les mêmes données (cas d'une diffusion empruntant le même début de route), sinon c est une nouvelle opération de communication qu'il faut créer et insérer dans le graphe de l'algorithme entre l'opération de communication précédente sur la route (ou o' en début de route) et o, et qu'il faut ordonnancer sur m (ajout d'une dépendance de contrôle entre la dernière opération de communication ordonnancée sur m, avant c, et c).

L'heuristique de routage se poursuit ainsi jusqu'à atteindre p, en évaluant à chaque étape les routes parallèles, donc en équilibrant la charge des moyens de communication, et en réutilisant les communications déjà routées, donc en évitant de dupliquer inutilement des communications.

Cette heuristique est sensible à l'ordre dans lequel sont traitées les dépendances de données inter-opérateur entre une opération et ses prédécesseurs. En les traitant par ordre croissant des dates de fin des prédécesseurs, on constate souvent de façon empirique une diminution des périodes d'inactivité des moyens de communication, et donc un meilleur équilibrage de charge des moyens de communication.

2.4.5 Minimisation de l'architecture

Jusqu'à présent on a fait l'hypothèse que le nombre total d'opérateurs, de communicateurs et de moyens de communication étaient donné a priori. On recherche dans ce cas à exploiter au mieux les ressources dont on dispose. Le problème peut être généralisé au cas où le nombre de ressources n'est pas donné a priori. Pour le résoudre, l'utilisateur peut se ramener au cas précédent par approximations successives, en redimensionnant l'architecture à chaque étape, jusqu'à trouver l'architecture minimale qui permette une implantation respectant les contraintes temps réel.

2.5 Génération d'exécutifs distribués temps réel

2.5.1 Générateur d'exécutif

Dès qu'une distribution et un ordonnancement ont été déterminés, il est assez simple de générer automatiquement des exécutifs distribués temps réel, principalement statiques avec une partie dynamique uniquement quand cela est inévitable (calcul des booléens de conditionnement à l'exécution) [21]. Ces exécutifs

sont générés en installant un système de communication inter-processeur sans interblocage (les interblocages ne peuvent provenir que de cycles de dépendance entre opérations, qui sont détectés au niveau du graphe de l'algorithme ; l'heuristique de distribution et d'ordonnancement ne faisant que renforcer l'ordre partiel (sans cycle de dépendance) entre les opérations de l'algorithme, le graphe de l'algorithme implanté est également sans cycle de dépendance, donc l'exécutif généré est sans interblocage).

Il faut encore insister ici sur le fait que dans les applications embarquées de contrôle-commande comprenant du traitement du signal et des images, l'exécutif qui supportera l'exécution distribuée temps réel doit être le moins coûteux possible. C'est pourquoi on évite autant que faire se peut les exécutifs dynamiques, qui s'ils semblent faciles à mettre en œuvre pour l'utilisateur, conduisent à une sur-couche logicielle système, qui prend à l'exécution des décisions d'allocation de ressources (changements de contexte pour allouer le temps CPU à différentes tâches, gestion de files d'attente pour les communications, codage-transfert-décodage d'informations de routage, etc.), dont le surcoût n'est pas négligeable. Avec l'approche préconisée, il suffit lorsqu'on construit l'exécutif, de conserver par un mécanisme simple de sémaphores les propriétés d'ordre, montrées avec les langages Synchrones sur la spécification d'algorithme, qui ont été conservées lors du choix de la distribution et de l'ordonnancement optimisés. Le processus de génération d'exécutif est parfaitement systématique : il correspond à ce que fait habituellement l'utilisateur à la main dans une approche classique après avoir effectué de nombreux tests en temps réel sur l'architecture réelle. Les exécutifs peuvent être totalement générés sur mesure en langage de haut niveau ou en assembleur, en faisant appel à des fonctions utilisateurs écrites en C ou dans une autre langage, compilées séparément si nécessaire. Ils peuvent aussi faire appel à des primitives de noyaux d'exécutifs distribués temps réel dynamiques standards résidents tels Vrtx, Virtuoso, Lynx, Osek etc. Cependant il faut être conscient que cela augmente le surcoût des exécutifs.

Dans le cadre de la méthodologie AAA, l'exécutif n'est donc pas conçu comme un "noyau résident" compilé et chargé indépendamment de l'application, mais plutôt comme l'ossature, la partie intégrante de l'application, générée sur mesure en fonction de l'algorithme et de l'architecture, à partir d'une bibliothèque générique de *macros d'allocation de ressources*, que nous appelons "noyau générique d'exécutif". L'exécutif est compilé et/ou lié avec les macro-opérations spécifiques à l'application (insérées en ligne dans l'exécutif ou compilées séparément), pour produire un code binaire chargeable et exécutable directement sur l'architecture matérielle, ou avec le support d'un système d'exploitation si son usage est imposé. Pour que le générateur d'exécutifs s'adapte facilement à différents types de processeurs cibles (de stations de travail avec un système d'exploitation, de traitement du signal et de microcontrôleurs sans système d'exploitation), il a été scindé en deux parties :

- la première partie traduit la distribution et l'ordonnancement produits par l'heuristique d'optimisation en un macro-code intermédiaire, comprenant pour chaque processeur des appels de macros d'allocation mémoire (allocation statique), une séquence d'appels de macros de calcul et autant de séquences d'appels de macros de communication que de moyens de communication accessibles au processeur. Ce macro-code est générique, c'est-à-dire indépendant du langage cible préféré pour chaque type de processeur;
- la seconde partie traduit le macro-code intermédiaire en code source compilable par la chaîne de compilation spécifique à chaque type de processeur cible. Cette seconde partie est basée sur le macro-processeur m4 (standard sous Unix, version GNU), complété pour chaque type de processeur cible par un fichier de définitions de macros spécifiques à ce type de processeur. Nous appelons "noyau générique d'exécutif" le jeu de macros, extensible et portable entre différents langages cibles (de haut niveau ou assembleur). Pour chaque nouveau type de processeur, si l'on est moins concerné par la performance de l'exécutif que par le temps de portage des macros, on peut par exemple adapter le jeu de macros générant du C, déjà développé pour stations de travail Unix, sinon on peut redéfinir les macros pour qu'elles génèrent de l'assembleur, en s'inspirant des jeux de macros déjà développés pour quelques processeurs populaires.

2.5.2 Noyau générique d'exécutif

Le code intermédiaire généré pour l'exécutif distribué d'une application est constitué :

- d'un fichier source pour chaque processeur, macro-codant l'exécutif dédié à ce processeur, qui sera traduit en source compilable pour ce processeur,

– et d'un fichier source macro-codant la topologie de l'architecture, qui sera traduit en un "makefile" pour automatiser les opérations de compilation.

La structure du code intermédiaire généré pour chaque processeur est directement issue de la distribution et de l'ordonnancement des calculs de l'algorithme et des communications inter-processeur qui en découlent. Dans le cadre des systèmes réactifs, les algorithmes sont par nature itératifs : un ensemble d'opérations de calcul est exécuté répétitivement jusqu'à ce qu'il soit décidé de mettre fin à l'application. La distribution et l'ordonnancement de cette itération sur plusieurs processeurs se traduit sur chaque processeur par *une séquence itérative d'opérations de calcul*. De même, la distribution et l'ordonnancement sur plusieurs moyens de communication inter-processeur, des transferts de données entre opérations exécutées sur des processeurs différents, se traduit sur chaque processeur par *une séquence itérative d'émissions et/ou de réceptions pour chacun des moyens de communication connecté au processeur*. Sur chaque processeur, la séquence de calculs et les séquences de transferts *sont exécutées en parallèle et synchronisées* par l'intermédiaire de sémaphores qui imposent les alternances entre écriture et lecture des tampons mémoire partagés entre opérations exécutées dans des séquences parallèles. L'ordre partiel d'exécution des opérations, spécifié par les arcs de dépendances de données du graphe d'algorithme, est ainsi imposé dans le code généré, soit statiquement par chaque séquence, soit dynamiquement par l'intermédiaire des synchronisations inter-séquences (et transitivement des transferts inter-processeurs).

Afin d'évaluer les performances temps réel des implantations réalisées, on peut générer les exécutifs avec des macro-opérations de chronométrage [22] insérées entre les macro-opérations de calcul. Le chronométrage s'effectue en trois phases : sur chaque processeur on mesure des dates de début et de fin des opérations et des transferts à l'aide de son horloge temps réel, puis en fin d'exécution de l'application on recale les horloges temps réel des différents processeurs, enfin on collecte les mesures mémorisées sur chaque processeur et on les transfère sur le processeur hôte qui possède des moyens de stockage de masse. Ces mesures sont ensuite analysées afin de les comparer à celles calculées lors de la prédiction de comportement temps réel. Ceci permet d'une part d'obtenir la première fois les durées de calcul et de communication qui permettent de caractériser le modèle d'architecture, puis les fois suivantes d'évaluer l'écart entre les modèles d'architecture utilisés et l'architecture réelle.

Lorsqu'on a suffisamment confiance en ces modèles, il est très facile de réaliser des variantes d'implantation en vue d'introduire de nouvelles fonctionnalités dans l'application, en modifiant l'algorithme et le nombre de composants de l'architecture, puis en effectuant une implantation optimisée et en générant l'exécutif. Cela est habituellement beaucoup plus difficile avec une approche classique n'utilisant pas la méthodologie AAA puisqu'il faut réaliser puis tester un nouveau prototype complet pour chaque variante d'implantation.

3 Logiciel de CAO niveau système SynDEx

SynDEx est un logiciel de CAO niveau système, supportant la méthodologie AAA, pour le prototypage rapide et pour optimiser la mise en œuvre d'applications distribuées temps réel embarquées. SynDEx V5 peut être chargée gratuitement à l'URL http://www-rocq.inria.fr/syndex. C'est un logiciel graphique interactif qui offre les services suivants :

- spécification et vérification d'un algorithme d'application saisi sous la forme d'un graphe flot de données conditionné (ou interface avec des langages de spécification et de vérification formelle de plus haut niveau tels les langages Synchrones, et/ou des environnements de simulation de calculs scientifiques tels Scicos) ;
- spécification d'un graphe d'architecture multicomposant (processeurs et/ou composants spécifiques) ;
- heuristique pour la distribution et l'ordonnancement de l'algorithme d'application sur l'architecture, avec optimisation du temps de réponse ;
- visualisation de la prédiction des performances temps réel pour le dimensionnement de l'architecture ;
- génération des exécutifs distribués temps réel, sans interblocage et principalement statiques, avec mesure optionnelle des performances temps réel. Ceux-ci sont construits, avec un surcoût minimal, à partir d'un noyau d'exécutif dépendant du processeur cible. Actuellement des noyaux d'exécutifs ont été développés pour : SHARC, TMS320C40, i80C196, MC68332, MPC555, i80X86 et stations de

travail Unix ou Linux. Des noyaux pour d'autres processeurs sont facilement portés à partir des noyaux existants.

Puisque les exécutifs distribués sont générés automatiquement, leur codage et leur mise au point sont éliminés, réduisant de manière importante le cycle de développement.

3.1 Interface graphique

L'utilisateur interagit avec le logiciel SynDEx par l'intermédiaire d'une interface graphique, avec des menus déroulants contextuels, des raccourcis clavier et souris pour les commandes les plus courantes, une aide en ligne en anglais, des barres de défilement et de zoom, et une zone textuelle pour des messages et une interaction avec l'interpréteur textuel Tcl, donnant à l'utilisateur averti un accès à toutes les commandes Tcl/Tk ainsi qu'à celles spécifiques à SynDEx. L'interface graphique que l'on peut voir sur la figure 7, est aisément portable et ouverte aux extensions grâce à l'utilisation des logiciels libres Tcl/Tk pour l'interface graphique et GNU-C++ pour les extensions Tcl développées pour accélérer les traitements intensifs (vérifications de cohérence des graphes, heuristiques d'adéquation, génération d'exécutif).

L'utilisateur trouvera les habituels menus "File" pour charger-sauvegarder-imprimer des "applications" (application = un graphe d'algorithme + un graphe d'architecture + une implantation), "Edit" pour copier-couper-coller des parties de graphes selectionnées, et "Help" pour explorer l'aide sur les menus et l'utilisation de la souris. La partie algorithme d'un fichier d'application peut avoir été produite par un autre logiciel de spécification-vérification-simulation ; un traducteur de code DC (le code commun des langages Synchrones), ainsi qu'une interface avec Scicos [23] (le simulateur graphique de systèmes dynamiques hybrides, basé sur Scilab) sont en cours de développement.

Trois menus spécifiques, "Algorithm", "Architecture" et "Adequation", permettent respectivement d'éditer les graphes de l'algorithme et de l'architecture (créer-inspecter-modifier-(dé)connecter leurs sommets), de les caractériser (durées des calculs et des communications), de lancer l'heuristique d'adéquation et de visualiser son résultat, et enfin de générer le macrocode de l'exécutif distribué qui, après traduction par le macro-processeur GNU-m4 avec l'aide des noyaux génériques d'exécutifs spécifiques à chaque type d'opérateur, et compilation par les chaînes de compilation spécifiques à chaque type d'opérateur, supportera l'exécution de l'algorithme sur l'architecture tel que distribué et ordonnancé par l'heuristique d'adéquation.

3.2 Spécification et vérification d'un algorithme d'application

Pour créer le graphe de l'algorithme de son application, l'utilisateur peut copier des opérations, des sous-graphes, voire un graphe entier d'une autre application ("exemple" parmi celles livrées avec le logiciel SynDEx, ou "bibliothèque" à copier créée par l'utilisateur), ou bien créer directement de nouveaux types d'opérations, et les interconnecter. La figure 7 présente dans le bas de la fenêtre de gauche un graphe d'algorithme (sommets opération verts) décrivant un "égaliseur adaptatif" souvent utilisé en télécommunications.

La cohérence du graphe est vérifiée à chacune de ses modifications. Un sommet collé est renommé automatiquement si nécessaire afin que tous les sommets portent des noms différents. Une entrée d'une opération ne peut être connecté qu'à une seule sortie, portant le même type de donnée, et à condition que cette connexion ne crée pas un cycle de dépendances, sinon la connexion est refusée.

Une vérification de cohérence supplémentaire est effectuée au début de l'heuristique d'optimisation : toute entrée doit être connectée (par contre une sortie peut ne pas être connectée, son résultat sera simplement inutilisé à l'exécution).

3.3 Spécification d'un graphe d'architecture multicomposant

Comme pour le graphe de l'algorithme, pour créer le graphe de l'architecture de son application, l'utilisateur peut copier des opérateurs et des moyens de communication, des sous-graphes, voire un graphe entier d'une autre application, ou bien créer directement de nouveaux types d'opérateurs et ou de moyens de communication, et les interconnecter. La figure 7 présente dans le haut gauche de la fenêtre de gauche un graphe d'architecture (sommets opérateur et moyen de communication bleus) formé de deux processeurs "root" et "P1" communicant par un bus "B".

Comme pour le graphe de l'algorithme, la cohérence du graphe est vérifiée à chacune de ses modifications, évitant que deux sommets portent le même nom, que des ports de types différents soient connectés au même

moyen de communication, ou qu'un opérateur soit connecté directement à un autre opérateur ou deux fois au même moyen de communication.

Une vérification de cohérence supplémentaire est effectuée au début de l'heuristique d'optimisation : le graphe de l'architecture doit être connexe, afin que le routage soit possible entre toute paire d'opérateurs, et un opérateur doit se nommer "root", afin qu'un arbre de couverture du graphe de l'architecture puisse être construit pour supporter le chargement initial des mémoires programme des opérateurs ainsi que la collecte finale des chronométrages si l'utilisateur a choisi cette option pour la génération d'exécutif.

3.4 Caractérisation et heuristique d'optimisation

L'heuristique d'optimisation effectue des calculs de chemins critiques basés sur les durées des opérations de calcul (resp. des opérations de communications) caractérisées par rapport aux opérateurs (resp. aux moyens de communications) sur lesquels elles peuvent être allouées. La figure 7 présente en bas à gauche la fenêtre de dialogue qui permet d'effectuer ces caractérisations. Elle présente aussi, au niveau du graphe d'algorithme, ce que l'utilisateur peut obtenir comme information lorsqu'il clique sur une opération. Il peut aussi obtenir des informations sur les dépendances de données, ou sur les opérateurs et les moyens de communication. Le logiciel SynDEx utilise une heuristique telle que celle décrite chapitre 2.4, pour des architecture hétérogènes, limitée actuellement à la minimisation du temps de réponse (une latence = cadence).

3.5 Prédiction et visualisation du comportement temps réel

Les calculs de dates effectués lors de l'optimisation de la latence permettent d'obtenir les dates de début et de fin de l'exécution des opérations et des transferts de données implantés sur l'architecture. Ceci permet de tracer un diagramme temporel décrivant une prédiction du comportement temps réel de l'algorithme sur l'architecture.

Ce point est important car il permet de faire l'évaluation des performances d'une application en simulant son comportement temps réel sans l'exécuter réellement (ni par l'architecture réelle, ni par un émulateur, ni par un programme de simulation). Il suffit d'avoir modélisé et caractérisé les composants que l'on veut utiliser pour construire son architecture. Cela permet par exemple d'évaluer, sans posséder réellement la nouvelle version d'un processeur, ce qu'il pourrait apporter en termes de performances.

Le diagramme temporel comprend une colonne pour chaque ressource séquentielle (opérateur ou moyen de communication, le temps se déroule de haut en bas), visualisant la distribution (allocation spatiale) et l'ordonnancement (allocation temporelle) des opérations de l'algorithme sur les opérateurs, et des opérations de communication inter-opérateur sur les moyens de communication. Le diagramme temporel est factorisé : une seule répétition (réaction) de la répétition infinie du graphe flot de données de l'algorithme, est représentée. Les durées d'exécution des opérations sont représentées par des "boîtes" rectangulaires, dont les bords haut et bas sont positionnés à leurs dates de début et de fin d'exécution. Des flèches représentent les dépendances de données intra et inter séquences : leur causalité leur impose de descendre ou au pire d'être horizontales (chaque dépendance entre un retard et son prédécesseur étant inter-répétition, sa flèche peut apparaître montante à cause de la factorisation du diagramme temporel, aussi ces flèches sont tracées en pointillés). L'origine (resp. extrémité) d'une dépendance inter-séquence représente une macro-opération de synchronisation "signal" (resp. "wait"). L'utilisateur peut déplacer horizontalement les colonnes pour rapprocher celles qui l'intéressent, et déplacer verticalement un réticule horizontal pour afficher la date correspondant à la position du réticule. La figure 7 présente dans la fenêtre de droite le diagramme temporel correspondant à l'optimisation de la distribution/ordonnancement de l'algorithme "égaliseur adaptatif" sur l'architecture à deux processeurs. Ici l'heuristique a distribué et ordonnancé les opérations *gensig*, *wind*, $filt$, *visu* sur le processeur *root*, et les opérations *winda*, $filta$, *sub*, *gain*, *adap*, $r2$ sur le processeur $p1$. Elle a aussi distribué et ordonnancé les trois opérations de communication (dépendances de données inter-partition) apparaissant lors de la distribution des opérations, sur le moyen de communication B. Le diagramme temporel prédit la latence optimisée que l'on obtiendra avec cette application, ici 69 unités de temps. Enfin, il suffit de lancer le générateur de macro-code exécutif. Ce dernier après compilation avec les noyaux d'exécutif correspondant à chaque processeur cible et chargement sur ces processeurs, permettra d'exécuter en temps réel l'application avec les performances prédites.

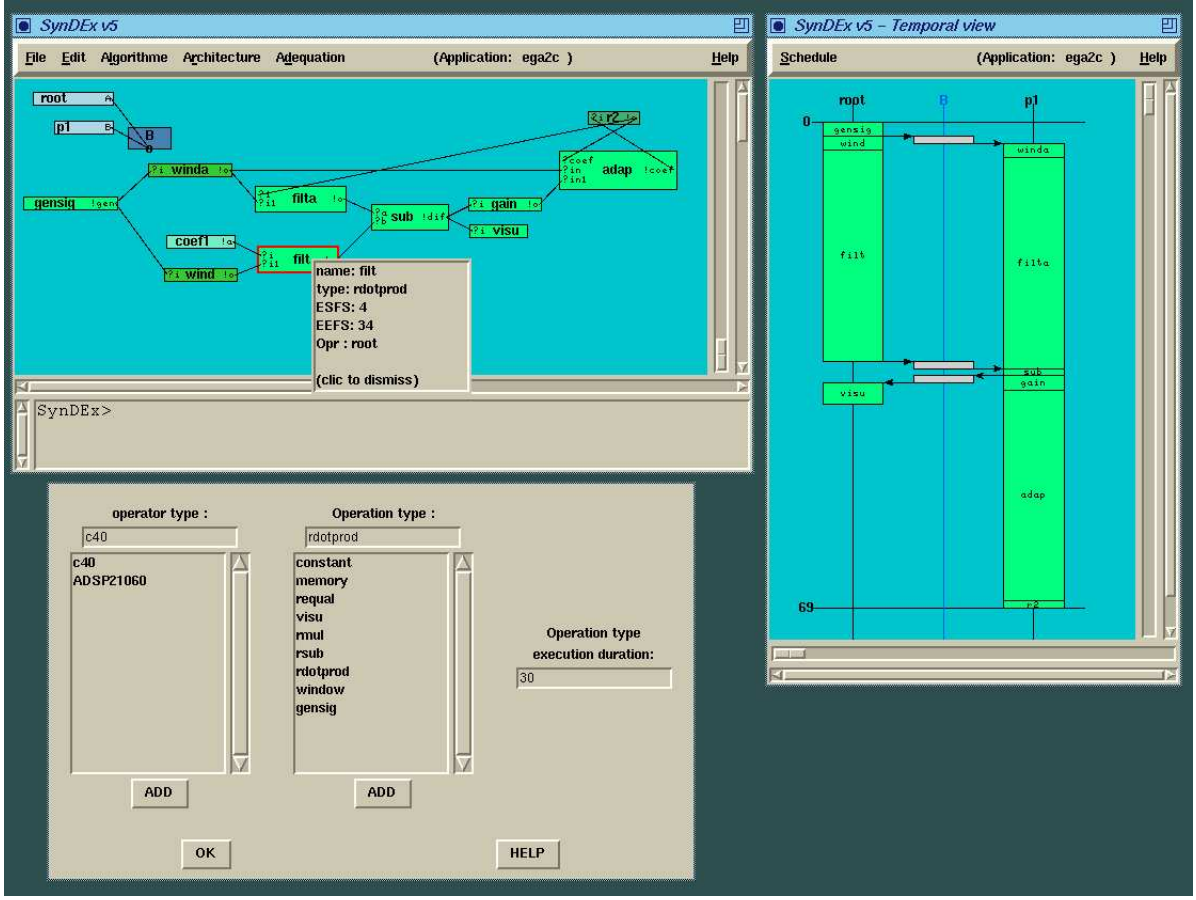

FIG. 7 – *Logiciel SynDEx V5*

4 Conclusion et perspectives

La méthodologie formelle que nous avons présentée permet d'une part de faire des vérifications temporelles en termes d'ordre sur les événements qui entrent et sortent dans l'algorithme permettant d'éliminer un grand nombre d'erreurs concernant la logique de l'algorithme puis d'assurer que ces vérifications restent valides après l'implantation optimisée respectant les contraintes temps réel. Elle permet d'autre part de générer automatiquement des exécutifs taillés sur mesure pour l'application sans interblocage et dont le surcoût est très faible. L'application prototype obtenue ainsi est directement utilisable comme produit de "série". Tout ceci contribue à diminuer fortement le cycle de développement des applications complexes nécessitant d'implanter des algorithmes de contrôle-commande comprenant du traitement du signal et des images sur des architectures multicomposants hétérogènes.

Nous avons considéré ici que les circuits intégrés spécifiques non programmables utilisés dans nos architectures multicomposants, sont des composants conçus avec un autre logiciel que SynDEx. Des travaux de recherche sont en cours pour concevoir aussi ces composants [24] [25] dans le cadre de la méthodologie AAA, en utilisant le formalisme de graphe de dépendances factorisé pour spécifier les algorithmes et pour générer les chemins de données et de contrôle correspondant à chaque composant non programmable. Cela devrait permettre d'avoir un cadre formel unique pour poser le problème de la conception conjointe logiciel-matériel (co-design) et ainsi de pouvoir rechercher des solutions approchées pour son optimisation.

Références

[1] A.M. Turing. On computable numbers, with an application to the entscheindungs problem. In *Proc. London Math. Soc.*, 1936.

[2] M. Gondran and M. Minoux. *Graphes et algorithmes*. Eyrolles, 1969.

[3] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[4] J.B. Dennis. Data flow supercomputers. *IEEE Computer*, 13(11), 1980.

[5] Annie Vicard. *Formalisation et optimisation des systmes informatiques distribus temps rel embarqus*. PhD thesis, Universit de Paris Nord, Spcialit informatique, 5/07/1999.

[6] Christophe Lavarenne and Yves Sorel. Modle unifi pour la conception conjointe logiciel-matriel. *Traitement du Signal*, 14(6), 1997.

[7] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Publishers, Dordrecht Boston, 1993.

[8] E.M. Clarke, E.A. Emerson, and A.P. Sistla. *Automatic verification of finite-state concurrent systems using temporal logic specifications*. ACM TOPLAS, 8(2). 1986.

[9] R.E. Bryant. Graph-based algorithms for boolean function manipulation. In *IEEE Transaction on Computers, C-35(8):677-691*, August 1936.

[10] A.Y. Zomaya. *Parallel and distributed computing handbook*. McGraw-Hill, 1996.

[11] Thierry Grandpierre. *Modlisation d'architectures parallles htrognes pour la gnration automatique d'excutifs distribus temps rel optimiss.* PhD thesis, Universit de Paris Sud, Spcialit lectronique, 30/11/2000.

[12] F. Gecseg. *Products of automata*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1986.

[13] Annie Vicard and Yves Sorel. Formalization and static optimization for parallel implementations. In *DAPSYS'98 Workshop on Distributed and Parallel Systems*, September 1998.

[14] C.A. Mead and L.A. Conway. *Introduction to VLSI systems*. Addison-Wesley, 1980.

[15] Zhen Liu and Christophe Corroyer. Effectiveness of heuristics and simulated annealing for the scheduling of concurrent task. an empirical comparison. In *PARLE'93, 5th international PARLE conference, June 14-17*, pages 452–463, Munich, Germany, November 1993.

[16] Annie Vicard and Yves Sorel. Formalisation et optimisation statique d'implantations parallles. In *Deuximes Journes francophones de recherche oprationnelle*, Sousse, Tunisie, April 1998.

[17] Valrie Lecocq. Optimisation d'applications temps rel embarques par exploitation de la rgularit algorithmique. Rapport de Stage INRIA, September 1997.

[18] Remy Kocik. *Sur l'optimisation des systmes distribus temps rel embarqus : application au prototypage rapide d'un vhicule lectrique semi-autonome*. PhD thesis, Universit de Rouen, Spcialit informatique industrielle, 22/03/2000.

[19] Thierry Grandpierre, Christophe Lavarenne, and Yves Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *CODES'99 7th International Workshop on Hardware/Software Co-Design*, Rome, May 1999.

[20] Y.K. Kwok and I Ahmad. Dynamic critical-path scheduling : An effective technique for allocating task graphs to multiprocessors. In *IEEE Trans. on Parallel and Distributed Systems, 7(5):506–521*, Munich, Germany, May 1996.

[21] Yves Sorel. Massively parallel systems with real time constraints, the "Algorithm Architecture Adequation Methodology". In *Conf. on Massively Parallel Computing Systems*, Ischia, Italy, May 1994.

[22] Franois Ennesser, Christophe Lavarenne, and Yves Sorel. Mthode chronomtrique pour l'optimisation du temps de rponse des excutifs SynDEx. Rapport de Recherche 1769, INRIA, June 1992.

[23] Ramine Nikoukhah and Serge Steer. *SCICOS: A Dynamic System Builder and Simulator*. http://www-rocq.inria.fr/scicos, User's Guide - Version 1.0, juin 1997.

[24] Ailton Dias. *Contribution l'implantation optimise d'algorithmes bas niveau de traitement du signal et des images sur des architectures mono-FPGA l'aide d'une mthodologie d'adquation algorithme-architecture*. PhD thesis, Universit de Paris Sud, Spcialit informatique, 12/07/2000.

[25] Ailton Dias, Christophe Lavarenne, Mohamed Akil, and Yves Sorel. Optimized implementation of real-time image processing algorithms on field programmable gate arrays. In *Fourth International Conference on Signal Processing*, Beijing, China, October 1998.