Le projet CAIRN : vers la conception d'architectures à partir de SIGNAL et ALPHA [1].

Thierry Gautier, Paul Le Guernic, Patrice Quinton,
Sanjay Rajopadhye, Tanguy Risset, Irina Smarandache

IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France

1. Les recherches menées dans ce projet sont soutenues par le CNET et France-Télécom (contrat de recherche 95 1B 036)

Résumé

L'objectif du projet CAIRN *est de définir des outils de conception conjointe logiciel–matériel à partir de spécifications de haut niveau, pour des applications temps-réel comportant des calculs intensifs.* CAIRN *repose sur le langage* SIGNAL *d'une part, et le langage* ALPHA *de l'autre, et tire parti des environnements de développement qui ont été conçus autour de ces langages.* SIGNAL *permet la spécification d'applications temps-réel, en prenant en compte les problèmes d'expression du temps et de la synchronisation.* ALPHA *est destiné à la dérivation d'architectures parallèles à partir de spécifications équationnelles. Dans cet article, nous présentons les langages* SIGNAL *et* ALPHA, *et nous décrivons les principes de* CAIRN. *Nous montrons ensuite comment co-simuler des programmes exprimés en utilisant* SIGNAL *et* ALPHA *et comment interfacer* SIGNAL *et* ALPHA *en définissant dans chaque langage des signaux ayant des horloges affines.*

1 Introduction

Les progrès de la technologie permettent désormais de réaliser des systèmes de très grande complexité dans un tout petit volume de matériel, en utilisant les techniques d'intégration. Dans de nombreux domaines — télécommunications, imagerie médicale, avionique, automobile, etc. — il est nécessaire de réaliser certains algorithmes, voire des systèmes complets, directement en silicium. Tous ces systèmes constituent ce qu'on appelle des *systèmes enfouis*, c'est-à-dire, des dispositifs informatiques auxquels l'utilisateur n'a pas directement accès dans des conditions normales. À titre d'exemples, un système de compression/décompression d'images, le système informatique d'un véhicule automobile ou le contrôleur d'une photocopieuse sont des systèmes enfouis.

La conception de tels systèmes requiert de mettre en œuvre des techniques de plus en plus complexes. En effet, ces systèmes doivent être corrects et fiables, leur modification après installation étant particulièrement coûteuse. Ceci impose de les valider, soit par simulation, soit par vérification formelle. Le temps de conception doit être aussi court que possible, pour des raisons de rapidité d'accès au marché (*time to market*). Enfin, ces systèmes doivent être optimaux en surface et en vitesse, et souvent satisfaire un grand nombre d'autres contraintes physiques qui dépendent du domaine d'application concerné. Pour cette raison, un système enfoui fait appel à des technologies très variées, qui vont de la programmation de processeurs programmables généraux, à la réalisation de circuits intégrés spécifiques, en passant par l'utilisation de processeurs programmables spécialisés (*ASIP : Application Specific Integrated Processor*), de micro-contrôleurs, ou de circuits configurables électriquement (*FPGA : Field Programmable Gate Arrays*).

Concevoir un système enfoui est donc une tâche longue et difficile, qui comprend de nombreuses étapes : spécification fonctionnelle, simulation et validation fonctionnelle, partitionnement technologique, programmation des processeurs, conception de circuits, assemblage, etc. Ces étapes font appel à des modèles différents, chacun reflétant le niveau de description correspondant aux traitements à effectuer : description comportementale, structurelle, physique, programme de haut niveau, assembleur, micro-code, ainsi que des combinaisons de ces descriptions. Au cours du processus de conception, une application est donc sujette à de nombreuses transformations qui fournissent des vues différentes des mêmes objets.

La nécessité de devoir produire simultanément des descriptions de type logiciel et des composants matériels a donné lieu récemment à des recherches sur ce qu'on appelle la *conception mixte matériel/logiciel* (*hardware/software codesign*). Il s'agit d'explorer les différentes alternatives de conception d'un système mixte matériel et logiciel en vue d'optimiser des critères de coût et/ou de performances. La modélisation des systèmes comprenant du matériel et du logiciel à différents niveaux d'abstraction est appelée *co-spécification*. La validation des conceptions soit par simulation, soit par vérification formelle du matériel et du logiciel à différents niveaux d'abstraction est appelée *co-simulation* ou *vérification formelle*.

Enfin, le partitionnement de l'application en deux parties, matérielle et logicielle, l'exploration et l'évaluation des différentes possibilités, la génération de circuits correspondant au matériel, la génération du code correspondant au logiciel, ainsi que la production de l'interface entre les deux sont désignés par le terme de *co-synthèse*.

Il n'existe pas de méthode universellement acceptée pour spécifier et mettre en œuvre un système complet. Cependant, plusieurs tentatives de structuration en vue d'obtention d'une méthodologie ont été proposées. Dans tous les cas, la conception repose sur un langage de spécification et son modèle sous-jacent. En général, on utilise un langage qui permet d'intégrer dans un seul cadre des descriptions matérielles et logicielles : il s'agit soit d'un langage de description de matériel enrichi de possibilités pour décrire du logiciel (VHDL avec possibilités d'appel à C), soit d'un langage de programmation incluant des concepts pour décrire du matériel (C_x[1] et HardwareC[2][3] qui sont des extensions de C).

Le modèle sous-jacent est généralement un graphe qui représente les fonctionnalités du système. Ce modèle doit permettre l'analyse ou la vérification de propriétés logiques du système et la possibilité d'évaluer ses performances par co-simulation. Il importe donc que ce langage soit exécutable.

Les approches actuellement utilisées sont limitées par un certain nombre de difficultés. Sur le modèle fonctionnel, le partitionnement en parties matérielles et logicielles s'effectue souvent manuellement; des techniques de partitionnement automatique sont certainement souhaitables pour assister le concepteur dans sa tâche. Toujours dans le cadre du processus de co-synthèse, la génération de l'interface entre matériel et logiciel est un problème assez complexe. D'autre part, le langage VHDL constitue le standard pour la description des sous-systèmes implémentés en matériel et pour la construction d'une plate-forme de co-simulation du système complet, mais il n'est pas possible de synthétiser de manière automatique des circuits à partir d'une description VHDL arbitraire.

L'état de l'art consiste à utiliser un cadre unique pour la spécification, la modélisation et la simulation du système. L'inconvénient essentiel des approches disponibles dans des environnements de conception commercialisés est l'absence d'une sémantique formelle des langages de spécification proposés jusqu'à présent, ce qui rend très difficile leur utilisation pour la vérification formelle.

Des réponses nouvelles, mais partielles, sont données par l'utilisation de modèles de type flot de données, tels que Cossap[4], Ptolemy[5], SPW[6], ou encore Phideo[7], qui permettent l'assemblage de fonctions et leur simulation, voire dans certains cas, la génération de code.

Le projet CAIRN aborde une partie des problèmes que nous venons de mentionner, essentiellement, le choix de "bons" formalismes pour la spécification et la synthèse. CAIRN réunit pour cela deux formalismes différents : SIGNAL et ALPHA. SIGNAL[8] a été inventé il y a 15 ans environ, pour spécifier des systèmes temps réel, en insistant en particulier sur la représentation du temps et sa manipulation formelle. Expérimenté sur des applications très diverses et de taille réaliste, SIGNAL a donné lieu à des développements commerciaux, et des environnements pour ce langage existent. ALPHA est un langage fonctionnel basé sur l'expression d'algorithmes par des récurrences multidimensionnelles. Il a pour but la synthèse d'architectures parallèles et régulières pour des algorithmes qui nécessitent une puissance de calcul importante. Un environnement de synthèse, appelé MMALPHA, a été développé à partir de ALPHA, afin de concevoir des circuits intégrés spécialisés [9].

L'alliance de SIGNAL et ALPHA permet d'atteindre plusieurs objectifs. Ces deux langages étant *déclaratifs*, on dispose ainsi de langages de spécifications d'un niveau très élevé d'abstraction. L'intérêt de cette propriété est que l'on peut décrire l'application en se concentrant sur les algorithmes, en faisant abstraction de l'architecture qui va les exécuter. Cette démarche est d'autant plus saine que l'on ne sait pas a priori quelle sera cette architecture.

D'autre part, la validation fonctionnelle de l'application est simplifiée par l'absence de détails architecturaux. Enfin, la nécessité de prendre en compte les changements de technologie de mise en œuvre imposés par les progrès très rapides des composants est un argument en faveur de spécifications purement algorithmiques : pendant le cycle de vie d'une application, il est fréquent que celle-ci doive être ré-implémentée sur des plate-formes matérielles différentes. Les algorithmes, quant à eux, changent moins souvent.

CAIRN poursuit un autre but, en associant SIGNAL et ALPHA. Ces deux langages possèdent chacun une sémantique formelle précise. Il est donc possible d'une part, de vérifier certaines propriétés des systèmes que l'on décrit (SIGNAL), et d'autre part, d'utiliser les propriétés axiomatiques de ALPHA pour effectuer des transformations dont la validité est de facto assurée.

L'association de SIGNAL et ALPHA est une tentative pour utiliser deux modèles différents, mais complémentaires. SIGNAL permet la description d'algorithmes dans lesquels la synchronisation temporelle des calculs est dominante. Le modèle flot de données sur lequel SIGNAL repose est muni d'opérateurs permettant la manipulation fine du temps, et le compilateur de SIGNAL est directement fondé sur cette particularité. SIGNAL est donc bien adapté à la représentation des applications où le contrôle est prédominant. ALPHA décrit des calculs multidimensionnels, et permet de modifier ces calculs pour faire apparaître du parallélisme. Il est donc approprié à la description d'algorithmes dans lesquels prédominent les calculs.

Le projet CAIRN a pour objectif de construire à partir de SIGNAL et de ALPHA des outils permettant la spécification, la simulation et la réalisation matérielle et logicielle d'une application. L'application à étudier est spécifiée en utilisant un mélange de SIGNAL — architecture globale, structuration en blocs de traitement, synchronisation de ces blocs, etc. — et ALPHA — description des algorithmes de calcul. Ces descriptions sont intimement liées, et peuvent être co-simulées. On peut aussi vérifier certaines de leurs propriétés. Chacune peut être transformée, afin de préparer la mise en œuvre de l'application sur une architecture donnée. On peut ensuite traduire chaque langage dans un langage approprié pour les parties logicielles, ou dans une description matérielle pour les éléments destinés à des circuits.

Chacune de ces étapes pose, bien entendu, un certain nombre de problèmes qui doivent être résolus. Le premier est de comprendre comment SIGNAL et ALPHA peuvent être interfacés pour pouvoir être associés sans perdre leurs propriétés, et ce de façon suffisamment intime pour que l'on puisse envisager des co-simulations, et une co-synthèse. C'est essentiellement à cet aspect que cet article est consacré.

Le reste de cet article est organisé de la façon suivante. Dans la partie 2, nous présentons l'utilisation du langage synchrone SIGNAL pour la spécification et la simulation fonctionnelle d'un système. Les concepts de base et les principaux outils existant dans l'environnement de programmation de SIGNAL sont décrits. La section 3 présente le langage ALPHA et explique son utilisation pour la synthèse d'architectures régulières. Le cœur de ce document est consacré à l'intégration de SIGNAL et ALPHA dans une chaîne de conception. Après une courte présentation de notre approche de la conception conjointe (section 4), nous décrivons plus précisément la co-simulation (section 5) puis nous abordons le problème des horloges multiples, dont la maîtrise en SIGNAL et en ALPHA est indispensable pour l'intégration des deux langages. Nous concluons par un bilan sur ce qui a été réalisé et les perspectives ouvertes par ce travail.

2 Le langage Signal

Signal[8] est un langage à flots de données appartenant, avec des langages comme Esterel et Lustre [10], à la famille des langages synchrones [11]. Il est utilisé pour la spécification, la programmation, la vérification et la mise en œuvre de systèmes réactifs et temps réel.

L'approche synchrone est basée sur l'hypothèse que les calculs ou les réactions du système spécifié s'effectuent en temps nul. Cela permet d'exprimer et de vérifier des propriétés logiques d'une application avant de prendre en compte les propriétés physiques d'une implémentation particulière. Une implémentation spécifique du processus est obtenue en associant un temps d'exécution à chaque action élémentaire, qui doit être a priori de durée bornée.

Un programme Signal est un système d'équations (pouvant être structuré en sous-systèmes), dont les variables sont des suites. Le compilateur de Signal se charge de la résolution partielle de ces systèmes d'équations de manière à vérifier la correction fonctionnelle du contrôle du programme et à générer, lorsque c'est possible, un code exécutable.

Autour du langage Signal et de son compilateur ont été développés un certain nombre d'outils qui constituent plus largement l'environnement de programmation de Signal. L'utilisateur dispose ainsi d'une interface graphique [12] pour la spécification de processus sous la forme de blocs interconnectés par des signaux. Il dispose aussi de générateurs de code C et Fortran pour des simulateurs fonctionnels [13], de générateurs de code intermédiaire pour l'accès aux outils de vérification [14], et d'outils pour la distribution de programmes Signal sur des architectures multi-processeurs [15]. Enfin, il existe un prototype de générateur de code VHDL [16] qui permet l'accès aux outils de synthèse existant.

La structure sous-jacente commune aux principaux composants de l'environnement Signal est constituée par le *graphe de flots de données synchronisés* (GFDS) [17] représentant un programme. Ce graphe est construit au cours de la compilation, et il est le support pour une série de vérifications : le programme présente-t-il des définitions contradictoires ou des dépendances circulaires de données ou de contrôle ? est-il déterministe ? impose-t-il des contraintes temporelles sur ses entrées ? etc. Certaines de ces vérifications conditionnent la génération de code, séquentiel ou distribué, qui est faite à partir de ce graphe.

Avant d'introduire brièvement les principaux opérateurs du langage Signal on se propose de donner quelques définitions. Dans la terminologie du langage, un *signal* est une suite infinie de valeurs indexée implicitement par un indice temporel. Le domaine de l'indice définit les instants logiques de présence du signal. Un signal peut être présent ou absent à un instant logique donné. L'*horloge* du signal est constituée par l'ensemble des instants logiques où le signal est présent (et a une valeur). Deux signaux ont la même horloge s'ils sont toujours présents aux mêmes instants logiques. Il existe une relation d'ordre partiel entre les horloges d'un programme.

Le noyau de base de Signal est constitué par un nombre restreint d'équations élémentaires (appelées aussi *processus*). Chaque processus a une sémantique formelle bien définie et introduit une équation sur les horloges et des dépendances de données entre les signaux impliqués. Des manipulations mathématiques sur les équations obtenues permettent de vérifier de manière formelle certaines propriétés du programme. Les processus élémentaires du langage et les principales extensions sont présentés succinctement dans le tableau 1. Pour une description plus détaillée de Signal, sa sémantique et ses diverses applications, nous

Opérateurs	Syntaxe SIGNAL	Description
extensions aux suites des opérateurs élémentaires	X := A op B	op est un opérateur arithmétique, relationnel ou booléen
retard	ZA := A $ N	ZA a pour valeur la N^e valeur passée de A
sous-échantillonnage	R := A when B	R vaut A quand B est présent et vrai
mélange avec priorité	R := X default Y	si X est présent R vaut X ; sinon, si Y est présent, R vaut Y ; sinon, R est absent
composition de processus	(| P | Q |)	processus dont les variables sont celles de P et de Q
Extensions	**Syntaxe**	**Description**
	when B	l'horloge des instants vrais de B
	event X	la présence des instants de X (horloge de X)
	synchro {A, B}	l'horloge de A est égale à celle de B
	R := A cell B	si A est présent R vaut A ; sinon, si B est présent et vrai, R a pour valeur la dernière valeur de A ; sinon, R est absent
tableau de processus	array i to N of P with CONNEX end	Reproduction N fois du processus P ; CONNEX définit les relations entre les processus ainsi obtenus

TAB. 1 – *Opérateurs de base de* SIGNAL *et extensions*

renvoyons le lecteur à [8] et [14].

L'opérateur `array` permet de définir de manière itérative un ou plusieurs signaux, non nécessairement synchrones, de types tableaux, par la réplication d'un même processus SIGNAL. Ainsi cette réplication peut former un réseau où les entrées d'un nœud sont les sorties des nœuds voisins immédiats.

Pour illustrer l'utilisation de SIGNAL, considérons le programme réalisant un sur-échantillonnage, décrit par la figure 1 : il s'agit d'un processus qui reçoit un entier variable `FB` et qui doit émettre à la sortie `FB` valeurs successives, *avant la prochaine occurrence de* `FB`.

`ZN` est la valeur retardée de `N` ; d'après l'équation 4, la valeur de `N` est égale à celle de `FB` avec priorité, ou à celle de `(ZN - 1)` par `défaut` si `FB` est absent. L'équation 6 exprime le fait que l'entrée `FB` est acceptée quand le signal booléen `(ZN <= 1)` porte la valeur vrai. On peut noter que l'horloge de la sortie `N` est plus “fréquente” que celle de l'entrée `FB`.

Le comportement de ce programme est illustré par le diagramme de la figure 2 pour une séquence particulière du signal `FB`.

La possibilité de spécifier des programmes comportant du sur-échantillonnage est une caractéristique particulière et puissante du langage SIGNAL, que l'on va exploiter dans la suite de cet article, dans le cadre de l'interfaçage SIGNAL-ALPHA.

```
process MUX =                                % déclaration de processus     %
        { ? integer FB;                      % ? : déclaration des entrées  %
          ! integer N                        % ! : déclaration des sorties  %
        }
        (| N := FB default (ZN - 1)          % corps du processus           %
         | ZN := N$1
         | synchro {FB, when (ZN <= 1)}
         |)
     where integer ZN init 1
end                                          % fin du corps du processus    %
```

FIG. 1 – *Programme de sur-échantillonnage en* SIGNAL

	t_0	t_1	t_2	t_3	t_4	t_5	$\cdots$
FB :	3			2		5	$\cdots$
ZN :	1	3	2	1	2	1	$\cdots$
N :	3	2	1	2	1	5	$\cdots$

FIG. 2 – *Exemple de signaux décrits par le programme* SIGNAL *de la figure 1*

3 Le langage ALPHA

ALPHA est un langage fonctionnel fortement typé. Un programme ALPHA est un *système* avec des paramètres qui peuvent prendre une valeur quelconque dans un *domaine* prédéfini. Les domaines sont fondamentaux en ALPHA, et doivent être des polyèdres, c'est-à-dire des ensembles d'indices entiers qui satisfont des inégalités linéaires.

La figure 3 décrit un programme permettant le calcul du produit d'une matrice `N`×`N` par un vecteur `N`. La première ligne indique que ce système a un paramètre, `N`, qui peut être un entier positif quelconque. Un système possède des variables d'entrée, de sortie, ou locales dont les domaines et les types sont déclarés statiquement. Le système `prodMatVect` rend un vecteur `c` défini sur un domaine à une dimension, et utilise une seule variable locale `C` définie sur un domaine à deux dimensions contenant N^2 points.

Chaque variable ALPHA peut être vue comme un tableau multidimensionnel ayant la forme d'un polyèdre (pas seulement un rectangle), ou de façon équivalente, comme une fonction d'un ensemble de points de polyèdres vers des valeurs.

Le corps d'un programme est un système d'équations délimité par les mots-clés `let` et `tel`. Chaque équation suit la syntaxe *Var* = *Expression*, où l'expression a une des formes suivantes :

Atomique : V, une variable (d'entrée, de sortie, ou locale) déclarée ou une constante.

Opération point à point : $Exp_1 \oplus Exp_2$ ou $\oplus$ est un opérateur binaire, et Exp_1 et Exp_2 sont deux expressions. Par exemple, si `A` et `B` sont des variables définies sur {`i | 0 <= i`}, alors `A+B` définit l'addition point à point des variables `A` et `B`, i.e., `A[i] + B[i]` pour

```
1 system prodMatVect: {N | N>1}                                  -- Paramètre
2              (a : {i,j|1 <= i,j <= N} of integer;            -- Entrées du système
3               b : {i|1 <= i <= N} of integer)
4      returns (c : {i|1 <= i <= N} of integer);               -- Sortie
5  var
6     C :  {i,j|1 <= i <= N; 0<= j <=N} of integer;            -- variable locale
7  let                                                         -- Corps du système
8       C = case                                        -- Equation définissant C
9            {|j=0} : 0.(i,j-> );
10           {|j>=1} : C.(i,j->i,j-1)+a.(i,j->i,j)*b.(i,j->j);
11          esac;
12      c = C.(i,j->i,N);
13  tel;                                                       -- Fin du système
```

FIG. 3 – *Programme Alpha effectuant un produit matrice-vecteur*

tout i. Les opérateurs point à point sont des extensions des opérateurs arithmétiques et logiques classiques. ALPHA permet aussi des opérateurs unaires et ternaires.

Restriction : *Dom* : *Exp*, où *Dom* est un domaine décrit avec la syntaxe esquissée ci-dessus. Par exemple, `{i | 3 <= i <= 10}: A`, définit une expression égale à A mais restreinte au sous-domaine $3 \leq i \leq 10$ de `A`.

Dépendance : *Exp.Dep* est la composition de *Exp* par une dépendance affine dont la syntaxe est `(idx, idx, ... ->i-expr, i-expr, ...)`. Chaque `idx` est un nom d'indice et `i-expr` est une expression *affine* des paramètres du système et des `idx`. Par exemple, `A.(i ->i-1)` définit une expression *Exp* telle que $Exp[i] = A[i-1]$ pour tout i. La ligne 9 de la figure 3 illustre le fait qu'on peut ne rien trouver à droite de la flèche si la variable correspondante est scalaire.

Case : case $Exp_1, \ldots Exp_i$ **esac**, par exemple, **case** `{i | i<10 }: A; {i | i >= 10}: B;` **esac** définit une expression qui vaut `A[i]` lorsque $i < 10$ et `B[i]` lorsque $i \geq 10$.

Noter que cette syntaxe peut être utilisée pour écrire des programmes arbitrairement complexes, et pas seulement les équations récurrentes affines conventionnelles telles qu'elles sont étudiées dans la littérature[18]. ALPHA, en tant que langage fonctionnel, respecte le *principe de substitution :* pour toute variable définie par une équation $V = Exp$, on peut textuellement remplacer une occurrence de V dans une expression quelconque par *Exp*.

ALPHA possède un certain nombre de propriétés importantes qui permettent de l'utiliser dans un environnement de transformation de programmes. Chaque programme ALPHA peut être simplifié dans une *forme normale* unique, appelée forme *cas-restriction-dépendance*. La sémantique du langage est telle que l'analyse statique permet de détecter un grand nombre d'incohérences avant l'exécution. Une transformation très utilisée est le *changement de base :* on peut montrer que les variables locales peuvent être réindexées sans modifier le sens d'un programme donné. Cette transformation est une des clés de la synthèse d'architectures régulières ou de la génération de code pour des machines parallèles car elle permet de faire apparaître le parallélisme [9].

Ces propriétés font d'ALPHA un langage adapté pour l'implémentation matérielle de traitements réguliers. Par contre, du fait même de cette spécificité, l'expression de traitements irréguliers (traitements nécessitant un contrôle complexe) sera vite lourde et l'implémentation en résultant risquerait d'être inefficace. Ceci justifie l'intégration d'ALPHA dans un environnement de co-synthèse avec SIGNAL.

4 Conception conjointe avec SIGNAL et ALPHA

On peut envisager une application de traitement du signal suffisamment complexe comme composée de parties de calculs numériques importantes reliées par du contrôle.

Pour que le système visé respecte des contraintes de performance (temps d'exécution) et de coût (surface de silicium) imposées par son utilisation, il peut être nécessaire de transformer les spécifications initiales, en particulier, en séparant ce qui doit être réalisé en logiciel de ce qui deviendra uen architecture matérielle (partitionnement logiciel/matériel). SIGNAL et ALPHA permettent les transformations de programmes nécessaires.

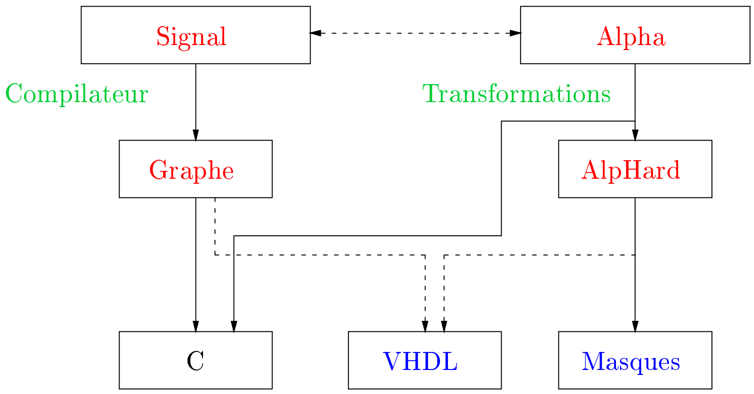

FIG. 4 – *Interaction entre* SIGNAL *et* ALPHA *dans* CAIRN

La démarche méthodologique proposée est illustrée par la figure 4. L'application est décrite en utilisant les deux langages: SIGNAL pour le cadre général de l'application, c'est-à-dire le contrôle, et ALPHA pour les composantes intensives en calcul. À ce niveau de description, les spécifications sont exécutées en faisant abstraction de l'architecture visée. La co-simulation est obtenue en compilant le code SIGNAL d'une part, et en générant à partir de ALPHA un programme C évaluant les variables à la demande [19].

Dans une première approche, la spécification séparée en utilisant SIGNAL et ALPHA correspond à un partitionnement initial en matériel et logiciel : on va considérer que toutes les parties décrites en ALPHA sont destinées au matériel, tandis que celles spécifiées en SIGNAL vont être implémentées en logiciel. L'interfaçage au niveau C décrit ci-dessus permet de réaliser une première co-simulation pour vérifier les algorithmes.

Comme on l'a vu dans les paragraphes précédents, un programme ALPHA peut être mis en œuvre sur des architectures parallèles de différentes structures : en utilisant les outils disponibles dans l'environnement MMAlpha, on peut, par une suite de raffinements de programmes qui garantissent l'équivalence entre les diverses spécifications intermédiaires, passer d'une version fonctionnelle d'un programme Alpha, à une architecture parallèle. Cette architecture est spécifiée dans un sous-ensemble de ALPHA appelé ALPHARD qui peut être traduit en VHDL [20].

Les raffinements de programmes effectués en ALPHA entraînent une altération de l'ensemble des spécifications fonctionnelles (celles en SIGNAL incluses), puisque le comportement temporel du programme transformé est modifié. Deux problèmes se posent alors :

– en premier lieu, il faut pouvoir co-simuler les nouvelles spécifications ainsi obtenues. Pour ce faire, il est nécessaire de définir un protocole d'échange entre SIGNAL et ALPHA. Ce point est examiné dans la partie 5.

– par ailleurs, il faut "resynchroniser" le programme SIGNAL en prenant en compte les modifications effectuées sur le programme ALPHA. Ceci exige de définir en SIGNAL de nouvelles transformations sur les variables (les transformations affines, abordées dans la partie 6), qui préservent les propriétés sémantiques du langage et qui permettent de traiter les resynchronisations.

Dans les paragraphes qui suivent, on propose une courte présentation des relations affines (et horloges régulières induites) en SIGNAL et un exemple simple de leur utilisation pour l'interfaçage entre SIGNAL et ALPHA. Le cas traité illustre la manière dont on peut décrire et simuler conjointement des spécifications SIGNAL et des descriptions de circuits en ALPHA, en utilisant le sur-échantillonnage de signaux existant en SIGNAL. Les relations affines permettent aussi de spécifier des sous-échantillonnages réguliers sur une horloge de base, donc de décrire des circuits synchrones qui ont des horloges différentes. En ce qui concerne le langage ALPHA, il serait donc intéressant de définir et de traiter des variables qui ont des horloges différentes et de pouvoir synthétiser des circuits adaptés en utilisant l'environnement de transformation de programmes existant. Ce problème est brièvement abordé et expliqué dans la suite.

Les extensions affines proposées pour les deux langages permettent d'interfacer les spécifications de systèmes à plusieurs niveaux d'abstraction et de changer le partitionnement initialement proposé (le passage de ALPHA à SIGNAL et inversement peut être réalisé en principe sur les sous-ensembles réguliers des deux langages). Il peut aussi être nécessaire d'implémenter en matériel des spécifications SIGNAL qui ne peuvent pas être décrites en ALPHA ou de réaliser en logiciel certains sous-systèmes initialement prévus pour une implémentation matérielle et décrits en ALPHA. SIGNAL et ALPHA disposent ou disposeront à terme, tous les deux, de générateurs de code C et VHDL, ce qui permet de modifier en conséquence le partitionnement imposé par les spécifications initiales. Les problèmes du partitionnement automatique et de la génération d'une interface entre les parties matérielles et logicielles restent à resoudre.

5 Co-simulation SIGNAL-ALPHA

SIGNAL et ALPHA disposent chacun d'un générateur de code C. La production de code est réalisée à partir de la structure sous-jacente au programme (GFDS pour SIGNAL, arbre syntaxique pour ALPHA). Un simulateur du système complet peut être construit avec les générateurs de code C des deux langages afin de faire une première vérification fonctionnelle.

ALPHA dispose d'outils de synthèse d'architectures régulières à partir d'une spécification formelle. Les transformations effectuées sont correctes par construction et sont semi-automatiques. L'architecture finale est décrite dans un sous-ensemble du langage, ALPHARD.

Enfin, toujours dans le contexte de la co-simulation, SIGNAL disposera à moyen terme d'outils de vérification de propriétés de coût et de performances [21] qui permettront de tester diverses implémentations, en utilisant des informations temporelles fournies par les sous-systèmes ALPHA.

On obtient ainsi des simulateurs pour chaque type de système considéré. Le problème est alors le suivant : étant donné deux programmes (ici ALPHA et SIGNAL), possédant chacun une procédure de simulation spécifique, comment construire un environnement de simulation global qui imbrique les procédures de simulation des systèmes composants. Cela revient finalement à imbriquer les spécifications et les formalismes induits par chacun des langages.

5.1 Génération de code `C` à partir de Signal

Afin de raisonner sur le contrôle d'un programme, le compilateur de SIGNAL associe à chaque signal le statut suivant : *absent*, *vrai* ou *faux* pour les signaux booléens, *absent* ou *présent* pour les autres types de signaux. En substituant dans chaque programme SIGNAL toute valeur non booléenne par l'étiquette *présent*, on génère un programme contenant seulement des horloges et des booléens, en l'occurrence un automate. De cette manière, le *calcul d'horloges* d'un programme SIGNAL code l'état de son contrôle. A chaque état, on peut associer ses calculs : le compilateur produit un ordonnancement des calculs en fonction des états du système. Le programme final est représenté sous la forme d'un arbre de modules. Ainsi, la compilation d'un programme SIGNAL engendre une représentation de la spécification initiale sous la forme d'un Graphe de Flots de Données Synchronisés (GFDS), qui regroupe en un seul formalisme la hiérarchie d'horloges et les sous-graphes de flots de données associés à leur horloges d'activation. Cette représentation sépare clairement le contrôle et les calculs, et elle constitue la structure de base pour la génération de code séquentiel : l'activation de chaque sous-graphe de flots de données est contrôlée par la présence ou l'absence de son horloge.

Le programme C généré à partir du GFDS est constitué du module décrivant un "instant" (comme expliqué ci-dessus), d'un module gérant les enchaînement d'instants et d'un module gérant les entrées-sorties.

5.2 Simulation d'ALPHA en `C`

Dans l'environnement MMALPHA, un traducteur produit, à partir de l'arbre syntaxique d'un programme ALPHA, un programme C destiné à simuler le programme ALPHA. Ce programme C a une exécution dirigée par la demande : pour chaque variable a du programme ALPHA (supposons que son domaine soit de dimension deux), le traducteur génère une fonction `compute_`a`(i,j)` qui calcule la valeur de a au point `(i,j)` de son domaine. Ce calcul se fait récursivement en appelant les fonctions qui calculent les valeurs des variables servant à la définition de a(i,j) (information contenue dans l'équation définissant a dans le programme ALPHA). Ce code n'est donc pas ordonnancé a priori, c'est la demande à l'exécution qui ordonnance les calculs. Un générateur de code C "ordonnancé" (c'est le flot de données qui ordonnance les calculs) est en cours de développement. Il constituera le support d'une co-simulation réelle.

Ce traducteur peut produire indifféremment un code destiné à simuler seul le système ALPHA ou un code interfaçable avec un programme SIGNAL. Dans ce dernier cas, le traducteur produit une fonction (du nom du programme ALPHA) et les données transitent par les paramètres de cette fonction (et non plus par les entrées et sorties standards).

5.3 Co-simulation au niveau fonctionnel

A ce niveau de co-simulation, il s'agit tout simplement d'isoler des calculs instantanés en SIGNAL (représentés sous la forme de *fonctions instantanées*), de décrire ces calculs en ALPHA, de générer les simulateurs C des parties décrites en SIGNAL et en ALPHA, et de faire communiquer les deux simulateurs en passant les paramètres du programme SIGNAL vers la fonction C générée à partir de ALPHA.

Ce processus ne pose pas de problème particulier. Il permet d'obtenir un simulateur fonctionnel d'une interface entre SIGNAL et ALPHA. La co-simulation au niveau architecturale présente beaucoup plus de difficultés.

5.4 Co-simulation au niveau architectural

La transformation du programme ALPHA en une architecture s'accompagne de la production d'informations (illustrées ci-après). Le programme SIGNAL ne doit plus faire un appel de fonction mais une succession d'appels modifiant différentes données à chaque top d'une *nouvelle horloge* introduite par ALPHA.

La co-simulation à ce niveau nécessite un simulateur de code ordonnancé pour ALPHA, d'autre part elle aborde les problèmes de simulation de la mémoire nécessaire pour garder des résultats obtenus entre des instants succésifs d'appel de la fonction C ordonnancée.

A titre d'exemple, considérons ici un programme ALPHA réalisant le produit d'une matrice par un vecteur. Pour réaliser un circuit à partir du programme de la figure 3, on va transformer ce programme et obtenir un nouveau programme ALPHA qui pourra être interprété en terme d'architecture (programme ALPHARD, le réseau systolique résultant est représenté sur la figure 5). Comme on le voit sur la figure 5, à chaque top d'horloge, certaines données sont lues par le réseau et certains résultats sont produits. Ici par exemple, sont entourés en pointillé les données entrant au deuxième top d'horloge, le coefficient c_1 est calculé au quatrième top d'horloge. On obtient donc, à partir de ce réseau, des informations temporelles : date d'arrivée des coefficients dans le réseau, date de sortie des coefficients résultats, et période du réseau (latence minimale entre deux utilisations successives du circuit), et des informations spatiales : architecture du réseau, taille du réseau.

Ici, les informations tirées du processus de synthèse sont pour un produit matrice–vecteur (matrice de dimension N×N, vecteur de dimension N) : le réseau est linéaire à N processeurs, les entrées sont envoyées entre `t=0` et `t=2N-2`, les sorties sont récupérées entre `t=N-1` et `t=2N-2`

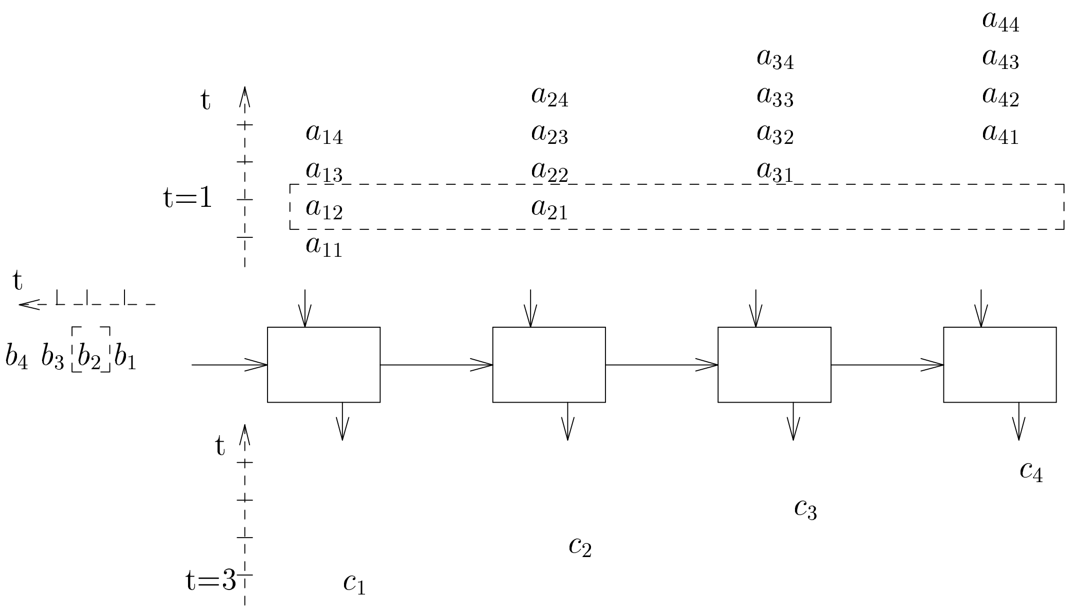

FIG. 5 – *Architecture du réseau synthétisé pour le produit matrice–vecteur (taille 4×4). Les données sont représentées dans l'ordre où elles entrent dans le réseau (ou en sortent pour c). Les données entrant au top $t = 2$ sont entourées en pointillé.*

et la période du réseau est `2N-1`. Le programme SIGNAL tire profit de ces informations pour construire :

– Un sur-échantillonnage des variables SIGNAL `a`, `b` et `c` (images des variables du programme ALPHA de la figure 3). On introduit donc trois variables `mema`, `memb`, `memc` qui vont valoir `a`, `b` et `c` sur la nouvelle horloge `T`: `T=Ta/(2N-1)`) ;

– Un programme d'affectation des variables SIGNAL `taba`, `tabb` et `tabc`, tableaux tampons dont l'horloge est `T`. Ces trois tableaux vont représenter, à chaque `t` de `T` les coefficients de `a`, `b` et `c` transitant entre le réseau et le programme SIGNAL appelant.

La co-simulation de ce niveau pose deux problèmes non encore complètement résolus.

– On doit gérer du côté SIGNAL des envois successifs n'ayant (a priori) aucun rapport entre eux, alors que du côté du programme ALPHA, les données peuvent être conservée plusieurs cycles dans le réseau. Cela nécessite la gestion d'une mémoire puisqu'à un cycle donné, on peut avoir besoin de données entrées aux cycles précédents.

– SIGNAL doit pouvoir manipuler les transformations affines sur les horloges afin de gérer les nouvelles horloges introduites. Pour avoir un ensemble cohérent, ALPHA doit aussi pouvoir manipuler des programmes multi-horloge. Ces points sont développés dans la section 6.

6 Horloges affines en SIGNAL et ALPHA

Cette partie traite de la gestion en SIGNAL et en ALPHA de signaux possédant des horloges ayant entre elles une relation affine. Comme nous l'avons vu dans la section 5, la manipulation de tels signaux est indispensable pour l'interface entre SIGNAL et ALPHA.

6.1 Relations affines en SIGNAL

Le compilateur du langage SIGNAL constitue un outil adéquat pour la vérification des contraintes entre les instants de présence (horloges) des signaux dans un programme SIGNAL. Les relations prises en compte par le calcul d'horloges sont le sous-échantillonnage par une condition booléenne, la borne supérieure (union d'ensembles d'instants), la borne inférieure (intersection) et la différence. D'autres relations peuvent être exprimées ; dans l'exemple d'interfaçage entre SIGNAL et ALPHA, apparaissent ainsi des relations *affines* entre horloges. Une extension utile du calcul d'horloges consisterait donc à enrichir les mécanismes de preuve existant dans le compilateur pour prendre en compte ce type de relation dans la résolution du système de contraintes construit à la compilation.

Deux signaux arbitraires `x` et `y` sont en relation affine si l'horloge $\mathtt{H_y}$ de `y` a été obtenue par une transformation affine sur les indices temporels de l'horloge $\mathtt{H_x}$ de `x` (on considère une horloge comme un ensemble d'indices entiers) ou vice-versa. Une transformation affine de paramètres entiers positifs (n, Φ, d) appliquée à l'horloge $\mathtt{H_x}$ de `x` va induire une horloge $\mathtt{H_y}$ en insérant $\mathtt{n} - 1$ instants entre deux instants de $\mathtt{H_x}$, puis en prenant chaque $\mathtt{d}^{\mathtt{e}}$ instant à partir de l'instant Φ. Une telle transformation est décrite en SIGNAL par la relation $\mathtt{affine\{\ n,\Phi,d\ \}(\ H_x,H_y\ )}$. La figure 6 présente un exemple de relation affine. On peut noter que la relation affine présentée ici ne s'applique qu'aux horloges des signaux `x` et `y` (de manière analogue à l'opérateur `synchro`) : les valeurs des signaux sont spécifiées séparément.

Sur l'exemple du produit matrice–vecteur présenté dans la section 5, l'expansion dans le temps du produit initialement instantané introduit des relations affines entre les horloges des entrées–sorties `a`, `b`, `c` et l'horloge sur-échantillonnée `T` : on a ainsi $\mathtt{affine\{\ 1{,}0{,}2*n-1\ \}(\ H_t,H_a\ )}$ et $\mathtt{affine\{\ 1{,}2*n-2{,}2*n-1\ \}(\ H_t,H_c\ )}$, qui induisent aussi $\mathtt{affine\{\ 2*n-1{,}2*n-2{,}2*n-1\ \}(\ H_a,H_c\ )}$.

De manière similaire, tout calcul instantané sur des données régulières de SIGNAL peut être expansé dans le temps par des transformations affines sur des variables initialement synchrones. La spécification d'une application par composition de systèmes SIGNAL comme celui décrit précédemment va induire des contraintes de synchronisation sur des signaux en relation affine d'horloges et la nécessité de définir, lorsque c'est possible, l'horloge la plus rapide du système global.

Sur l'exemple de la figure 6 une horloge *plus fréquente* que celles de `x` et `y` a été exhibée ; les instants de cette horloge sont indiqués par les traits en pointillé (bien qu'ils apparaissent équidistants sur la figure, cette équidistance n'a pas de signification dans la sémantique de SIGNAL : seule la position relative des instants est importante). Cette horloge décrit par exemple le cadencement d'un système SIGNAL dans lequel `x` et `y` sont respectivement consommé et produit. La composition de processus en SIGNAL permet la juxtaposition de sous-systèmes de ce type et, dans certains cas, la construction d'une horloge du système

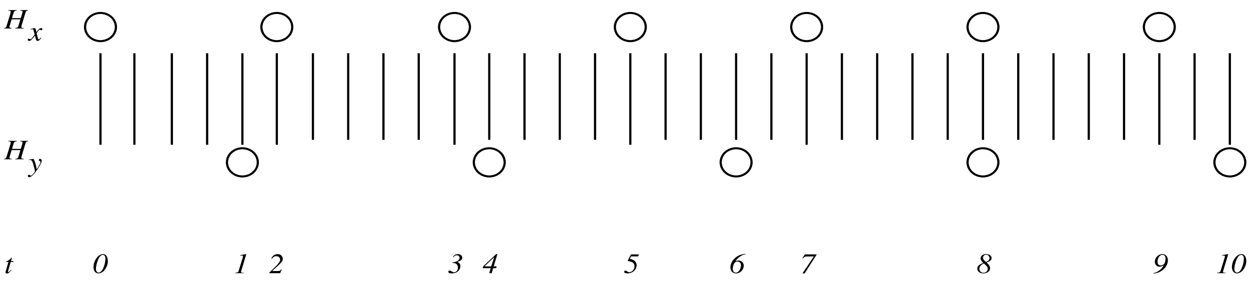

FIG. 6 – *Exemple :* affine{ 5,4,7 }($\mathtt{H_x}$,$\mathtt{H_y}$)

englobant supérieure aux horloges de cadencement de chaque sous-système. La manière dont les instants des sous-horloges de cadencement se retrouvent sur l'horloge globale du système doit respecter les contraintes de synchronisation exprimées par l'utilisateur.

Considérons par exemple un programme SIGNAL dans lequel seules les relations suivantes ont été exprimées :

affine{ 5, 4, 7 }($\mathtt{H_x}$,$\mathtt{H_{y_1}}$)
affine{ 5, 3, 7 }($\mathtt{H_x}$,$\mathtt{H_{y_2}}$)

En l'absence de contraintes supplémentaires, on peut construire l'horloge la plus rapide, T, du système de plusieurs façons ; deux d'entre elles sont illustrées sur la figure 7.

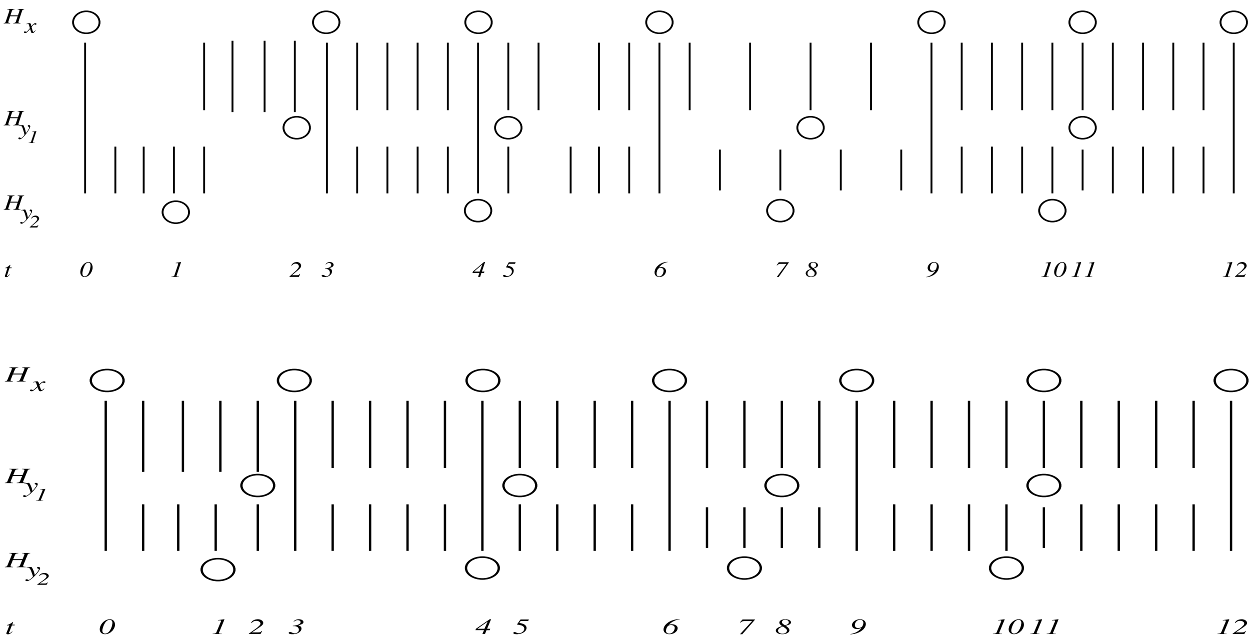

FIG. 7 – affine{ 5,4,7 }($\mathtt{H_x}$,$\mathtt{H_{y_1}}$) *et* affine{ 5,3,7 }($\mathtt{H_x}$,$\mathtt{H_{y_2}}$)

Une équation de synchronisation entre $\mathtt{y_1}$ et $\mathtt{y_2}$ (synchro{$\mathtt{y_1}$,$\mathtt{y_2}$}) peut être ajoutée au système initial. Le compilateur doit être en mesure de vérifier que la nouvelle contrainte peut

```
system Tiertop (x : {i | i >= 0} of integer )
           returns (y : {i | i >= 0} of integer );
var
    A,B : {i | i >= 0} of integer;
let
    B = A . (i -> 3i);
    A = case
                {i | i = 0} : 0;
                {3i | i > 0} : B.(i -> i/3 - 1) + A.(i -> i-1);
                {3i+1 | i > 0} : A.(i -> i-1);
                {3i+2 | i > 0} : A.(i -> i-1);
        esac;
    y = B;
tel
```

FIG. 8 – *Un programme* ALPHA *qui utilise des domaines périodiques*

ou non être satisfaite (dans l'exemple - cf. figure 7 - les instants de présence des signaux $\mathtt{y}_1$ et $\mathtt{y}_2$ ne peuvent pas coincider étant donné leur position relative par rapport aux instants de présence de `x`). Au contraire, si $\mathtt{y}_1$ et $\mathtt{y}_2$ sont définis tels que `affine{ 5,4,7 }(` $\mathtt{H_x}, \mathtt{H_{y_1}}$ `)` et `affine{ 10,8,14 }(` $\mathtt{H_x}, \mathtt{H_{y_2}}$ `)` (on a multiplié par 2 les paramètres $(5, 4, 7)$ de la première relation affine), on va pouvoir resynchroniser $\mathtt{y}_1$ et $\mathtt{y}_2$ et parler ainsi d'équivalence entre les relations affines `affine{ 5,4,7 }(` $\mathtt{H_x}, \mathtt{H_y}$ `)` et `affine{ 10,8,14 }(` $\mathtt{H_x}, \mathtt{H_y}$ `)`.

Afin de résoudre des contraintes de ce type, une étude sur les relations (n, Φ, d)-affines dans le cadre de la sémantique de traces de SIGNAL est en cours. Il s'agit principalement de déterminer toutes les relations affines équivalentes à une relation donnée, de composer deux relations, de déterminer l'inverse d'une relation, d'envisager et caractériser des opérations de réunion et intersection sur les horloges obtenues par transformation affine.

6.2 Multi-horloge en Alpha

Le problème des horloge multiples en ALPHA est légèrement différent. Les algorithmes traités étant intrinsèquement réguliers, il est raisonnable de considérer les horloges comme régulières (chaque cycle est équidistant en temps du top précédent). Modéliser un ensemble de tops consécutifs revient alors à décrire l'ensemble des entiers compris entre deux bornes. Comme les domaines manipulés par ALPHA sont des polyèdres convexes, cette modélisation ne pose pas de problème.

Par contre, lorsqu'on veut exprimer qu'une horloge est trois fois plus rapide qu'une autre, il faut décrire des polyèdres avec des "trous" (en l'occurrence, deux trous tous les trois points). Ces trous étant répartis de manière répétitive, on va pouvoir modéliser de tels ensemble avec les *$\mathcal{Z}$-polyèdres* [22, 23], qui sont exactement des domaines avec des trous réguliers.

Nous décrivons ici un exemple simple qui illustre l'utilisation de $\mathcal{Z}$-polyèdres. Considérons la figure 8 qui décrit un programme ALPHA faisant intervenir deux variables : `A` et `B` (tableaux unidimensionnels). `B[i]` dépend de `A[3i]` et, pour `i` multiple de 3, `A[i]` dépend de `B[i/3-1]`. Les trois derniers domaines de la figure 8 sont des $\mathcal{Z}$-polyèdres. Si nous interprétons l'indice `i` comme le temps, l'exécution du programme peut être représentée par la figure 9. Ce graphe n'étant pas uniforme, il va produire une architecture peu efficace (une valeur de `B` produite au top `t` devra être mémorisée jusqu'au top `3t` pour être consommée).

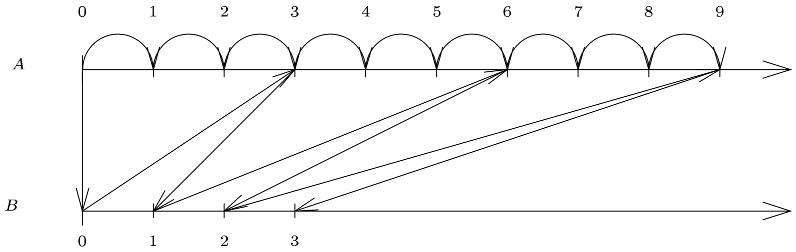

FIG. 9 – *Graphe de dépendance du systéme* `Tiertop`. *Les flèches représentent les dépendances de données, les nombres représentent les indices des variables* `A` *et* `B`.

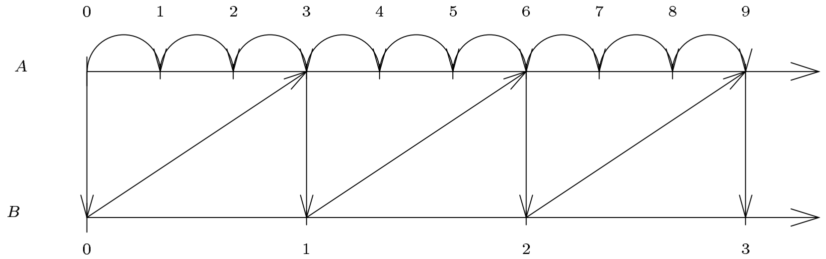

FIG. 10 – *Graphe de dépendance uniformisé par dilatation du domaine de* `B`.

Or dans le contexte du traitement du signal, on peut résoudre ce problème en considérant que les variables `A` et `B` ont des horloges différentes. Ici, par exemple, on peut *dilater* l'horloge de `B` par un facteur trois et on l'on obtient un graphe uniforme (figure 10) : le calcul des valeurs de `A[3i]` et `B[i]` sont alignés temporellement. Le domaine de `B` devient alors un $\mathcal{Z}$-polyèdre, `B` n'existe qu'un cycle sur trois.

Pour bien comprendre un telle transformation, considérons l'exemple de la figure 6 illustrant les transformations affines de SIGNAL. Pour exprimer une telle relation entre les horloges de deux variables ALPHA, considérons le domaine : `D={i | 0<=i }` Nous pouvons obtenir les domaines de H_x and H_y (sur la figure 6) en appliquant respectivement sur ce domaine les transformation affines : `(i ->5i)`, et `(i ->7i+4)`. On obtient alors les domaines : pour `x` : $\{$`5i | i>=0 `$\}$ et pour `y` $\{$`7i+4| i>=0 `$\}$.

La transformation présentée ici est très proche de celle présentée précédemment pour SIGNAL, il existe néanmoins une légère différence. En ALPHA nous supposons implicitement que toutes les variables sont synchronisées par une horloge globale (horloge *la plus rapide*). Cette hypothèse nous permet d'assurer que les programmes multi-horloges exprimés en ALPHA et en SIGNAL seront compatibles.

7 Conclusion

Nous avons présenté les principes d'une méthodologie de conception conjointe basée sur les environnements SIGNAL et ALPHA. La conjonction de ces deux langages permet, d'une part d'exprimer le comportement d'un système par une spécification de haut niveau (indépendante de la technologie), d'autre part de profiter des possibilités de transformations de programmes présentes dans les deux environnements pour obtenir de manière semi-automatique une description structurelle précise du système à implémenter. Nous avons détaillé la cosimulation fonctionnelle Signal-Alpha ainsi que les extensions des deux langages nécessaires pour rendre compatibles le traitement de signaux ayant des horloges différentes. Ces extensions théoriques sont essentielles pour obtenir un environnement stable pour la conception conjointe de haut niveau. Un certain nombre de problèmes restent à résoudre, en particulier le partitionnement automatique de l'application de départ et la réalisation pratique de l'interface entre le logiciel et le matériel.

Références

[1] Rolf Ernst, Jörg Henkel, and Thomas Benner. – Hardware-software cosynthesis for microcontrollers. – In *IEEE Design & Test of Computers*, pages 64–75, December 1993.

[2] Rajesh K. Gupta, Claudionor N. Coelho, and Giovanni De Micheli. – Program implementation schemes for hardware-software systems. – In *Computer*, pages 48–55, January 1994.

[3] Rajesh K. Gupta and Giovanni de Micheli. – Hardware-software cosynthesis for digital systems. – In *IEEE Design & Test of Computers*, pages pages 29–41, September 1993.

[4] J. Kunkel. – COSSAP: A stream driven simulator. – In *IEEE International Workshop on Microelectronics and prototyping*, Interlaken, Switzerland, 1991.

[5] C. Passerone, M. Chiodo, W. Gosti, L. Lavagno, and A. Sangiovanni-Vincentelli. – Evaluation of trade-offs in the design of embedded systems via co-simulation. – Technical Report UCB/ERL M96/12, UC Berkeley, 1996. – submitted to Int. Conf. on Computer Aided Design 1996.

[6] *ComDisco Systems Inc. SPW – The DSP Framework. Data Sheet*, 1993.

[7] M.F.J. Verhaeg, P.E.R. Lippens, E.H.L Aarts, J.H.M. Korst, J.L. Van Meerbergen, and A. Van Der Werf. – Modelling periodicity by phideo streams. – In *Int. Work. on High Level Synthesis*, pages 256–266, November 1992.

[8] Paul Le Guernic and Michel Le Borgne and Thierry Gautier and Claude Le Maire. – Programming real time applications with Signal. – Technical Report 1446, INRIA-Rennes, Juin 1991.

[9] D. Wilde and O. Sie. – Regular array synthesis using ALPHA. – Technical Report Internal Publication 829, IRISA, Rennes, France, May 1994.

[10] *Special section : R/T Programming, Proceedings of the IEEE*, volume 79, September 1991.

[11] Albert Benveniste and Gérard Berry. – The Synchronous Approach to Reactive and Real-Time Systems. – In *Proceedings of the IEEE*, volume 79, septembre 1991.

[12] Patricia Bournai and Paul Le Guernic. – Un environnement graphique pour le langage Signal. – Technical Report 2040, INRIA-Rennes, Septembre 1993.

[13] Olivier Maffeïs. – *Ordonnancements de graphes de flots synchrones; application à la mise en œuvre de Signal.* – PhD thesis, Université de Rennes I, January 1993.

[14] Tochéou Amagbegnon and Paul Le Guernic and Hervé Marchand and Eric Rutten. – The Signal dataflow methodology applied to a production cell. – In *Lecture Notes in Computer Science LNCS*, number 891. Springer Verlag, January 1995.

[15] C. Lavarenne and O. Seghrouchni and Y. Sorel and M. Sorine. – The Syndex software environment for real-time distributed systems design and implementation. – In *ECC91 European Control Conference, Grenoble*, pages 1684–1689, Grenoble, France, July 1991.

[16] Mohammed Belhadj. – *Conception d'architectures en utilisant* SIGNAL *et* VHDL. – PhD thesis, Université de Rennes I, 1994.

[17] Loïc Besnard. – *Compilation de Signal : horloges, dépendances, environnement.* – PhD thesis, Université de Rennes I, Septembre 1992.

[18] R.M. Karp, R.E. Miller, and S. Winograd. – The organization of computations for uniform recurrence equations. – *Journal of the ACM*, 14(3):563–590, July 1967.

[19] P. Quinton, S. Rajopadhye, and D. Wilde. – Deriving imperative code from functional programs. – In *7th Conference on Functional Programming and Computer Architecture*, pages 36–44, La Jolla, Ca, USA, June 1995. SIGPLAN/SIGARCH/WG2.8.

[20] P. Le Moenner, L. Perraudeau, P. Quinton, S. Rajopadhye, and T. Risset. – Generating Regular Arithmetic Circuits with ALPHARD. – In *2d International Conference on Massively Parallel Computing Systems*, Ischia, Italy, May 1996.

[21] Apostolos A. Kountouris and Paul Le Guernic. – Profiling of signal programs and its application in the timing evaluation of design implementations. – In *Hardware-Software Cosynthesis for Reconfigurable Systems – Colloquium organised by Professional Group C2 (Hardware and systems engineering)*. Hewlett Packard Laboratories, Bristol, February, Thursday 22 1996.

[22] P. Quinton, S. V. Rajopadhye, and T. Risset. – Extension of the ALPHA language to recurrences on sparse periodic domains. – In *Int. Conf. on Application Specific Array Processors (To appear)*, 1996.

[23] P. Quinton, S. V. Rajopadhye, and T. Risset. – On manipulating z-polyhedra. – Technical Report Internal Publication 1016, IRISA, Rennes, France, 1996.