ECOS[1]

Un Environnement générique de Co-Spécification pour le prototypage d'applications temps réel.

"De la spécification formelle au partitionnement"

LaMI
Laboratoire Mathématiques/Informatique
Université d'Evry Val d'Essonne
Bd des Coquibus, 91025, France
Tel. 69 47 74 51, Fax 69 47 74 72

1. Introduction

Les avancées technologiques permettent aujourd'hui des développements de composants spécifiques (ASIC, FPGA) pour un coût et dans un temps raisonnables. Pour une application donnée, l'exploration automatique, de l'espace des solutions d'implantation, devient envisageable. Ceci suggère une méthodologie de conception d'un système plus souple, où le logiciel et le matériel peuvent être conçus en interaction. Cette conception de systèmes mixtes est appelée *co-spécification*. Une telle approche doit permettre la réalisation de systèmes dédiés à des applications spécifiques — traitement du signal, télécommunications, ...— où le choix d'implantation matériel/logiciel peut évoluer en fonction des besoins — certaines fonctions réalisées en logiciel au départ pouvant être intégrées au matériel par la suite — ce qui rend possible l'évolution des produits en fonction de la technologie (matérielle/logicielle) et par là même une arrivée plus rapide sur le marché.
Deux problèmes clés sont à aborder dans la co-spécification : d'une part la *spécification* du système d'autre part le *partage matériel/logiciel.*

Le but du projet *ECOS* (Environnement générique de Co-Spécification pour le prototypage d'applications temps réel) est de fournir des outils permettant le développement d'un système (relatif au traitement du signal ou aux télécommunications) des spécifications formelles à son implantation. La Figure 1 illustre les étapes du projet. Les étapes principales sont :

- la spécification formelle
- le partage matériel/logiciel.

2. Présentation de l'approche

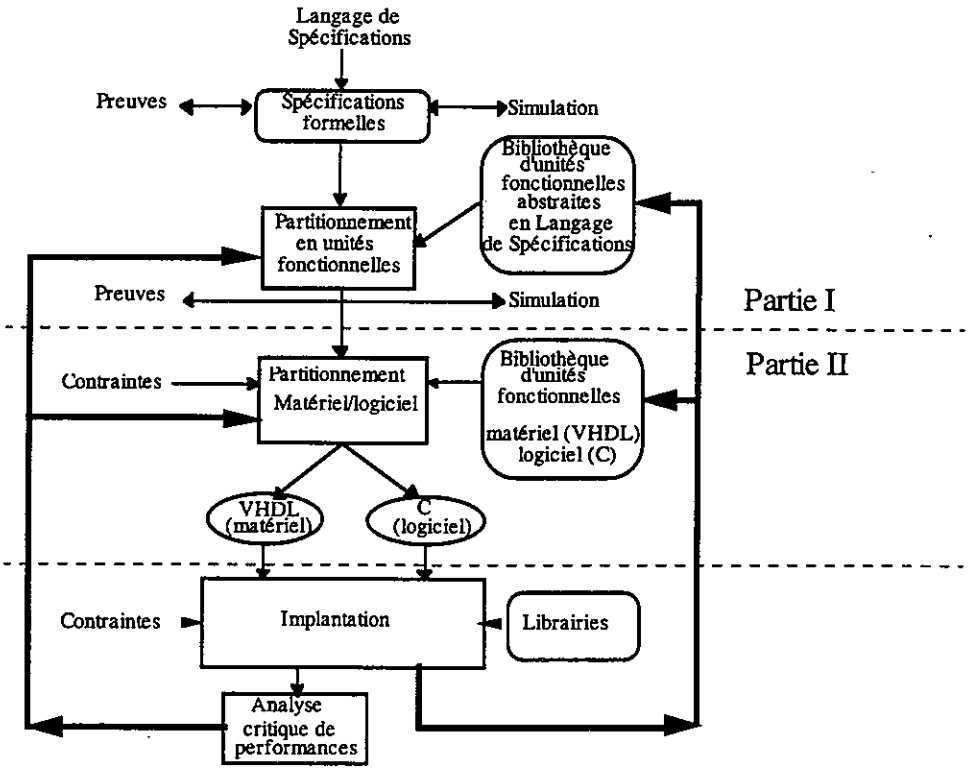
2.1 Spécification formelle

La méthode de spécification formelle que nous proposons est fondée sur la théorie des « ETOILE-spécification ». Il s'agit de spécifications orientées objets pour lesquelles un

[1] M. Aiguier, J. Benzakki, G. Bernot, S. Beroff, D. Dupont, L. Freund, M. Israël, F. Rousseau

effort particulier a été fourni dans le domaine du raffinement des spécifications. Cette approche permet de mener dans un cadre parfaitement établi :

- des preuves avant et après raffinement,
- des vérifications de cohérence et de complétude des spécifications,
- le développement incrémental des spécifications,
- la recherche de raffinements successifs à partir de bibliothèque de modèles abstraits logiciel et/ou matériel.



- **Figure 1 Des spécifications formelles à l'implantation**

Après que les raffinements successifs de spécifications aient été menés à leur terme, on peut déduire un ensemble de fonctionnalités qui forment la base sur laquelle le partionnement travaille.

2.2 Partitionnement

Une fois que plusieurs étapes d'implantations abstraites ont été effectuées, on aboutit à une spécification trés détaillée dans laquelle il est possible d'identifier les fonctionnalités « de base » à réaliser. Ces fonctionnalités de base coïncident le plus souvent avec les objets de plus bas niveau spécifiés dans la spécification la plus raffinée.
L'étape de Partitionnement Logiciel/Matériel consiste alors à choisir entre une réalisation logicielle et une réalisation matérielle pour chacune de ces fonctionnalités et de déterminer la date d'exécution de ces noeuds (ordonnancement). La description de notre méthode est donnée à la section 2.

2.3 Présentation des ETOILE-spécifications

2.3.1 Types d'objets

Ici, nous nous focalisons sur la spécification de ce que nous appelons un « type d'objets ». Intuitivement, lorsque nous spécifions un type d'objets, nous spécifions en fait le comportement d'un objet arbitraire de ce type. Cet objet représentatif sera appelé conventionnellement « self ». Il est le « référentiel » de la spécification et toute propriété φ (appelée également axiome) de la spécification doit être comprise au sens suivant : **Du point de vue de *self*, φ semble toujours vraie**
Une spécification de types d'objets décrit donc un monde « égocentrique »; le centre en est *self* et il est entouré d'une collection d'objets qui peuvent lui rendre des services afin de l'aider à réaliser les méthodes qu'il fournit. Un type d'objets peut alors être vu comme une « étoile ». Le centre c de l'étoile est le type d'objets de *self* que nous désirons spécifier. On l'appelle également le *type d'intérêt*. Les « branches » représentent une vue simplifiée (une abstraction) du comportement des types d'objets qui fournissent des services à l'objet *self*.
De manière classique en logique ou en spécifications algébriques, les spécifications associées à un type d'objets sont définies par une signature et un ensemble de formules (ou encore axiomes) définies sur cette signature.

2.3.1.1 Signature

Définition 1.1 *Un ETOILE-ensemble S est un triplet (c,O,D) où O et D sont des ensembles, $c \notin D$ et $O \cap D = \varnothing$.*

De plus, étant donné un ETOILE-ensemble S = (c,O,D), nous notons $S = O \cup D$ et $S^{\varepsilon} = S \cup \{\varepsilon\}$ où ε dénote le mot vide.
Intuitivement, l'ensemble O contient les noms des types d'objets dont self peut utiliser des services (ou encore méthodes) pour définir son comportement et D l'ensemble des types de données classiques manipulés par self.

Définition 1.2 *Une* ETOILE-signature θ *est un triplet (S,F,M) où :*

- *S est un ETOILE-ensemble.*
- *F est un ensemble de noms de fonctions, chacun muni d'une arité de la forme $(s_1 \times \ldots \times s_n \rightarrow s_{n+1})$ où $s_i \in S$ pour $i \in [1,n+1]$.*
- *pour chaque $o \in O \cup \{c\}$, M^o est un ensemble de noms de méthodes, chacun muni d'une arité de la forme $(s_1 \times \ldots \times s_n \rightarrow s)$ où $s_i \in S$ pour $i \in [1,n]$ et $s \in S^{\varepsilon}$.*

Intuitivement, l'ensemble F contient les opérations classiques sur chaque type de données de la signature. Ces opérations ont un comportement qui n'est pas modifié à travers le temps. A l'inverse, chaque ensemble M^o contient les méthodes dont le comportement dépend de l'état de l'objet exécutant cette méthode. Dans le cas où une méthode a pour profil $(s_1 \times \ldots \times s_n \rightarrow \varepsilon)$, cette dernière doit être comprise comme une procédure (par analogie aux langages de programmation) dont l'effet est uniquement de modifier l'état de l'objet qui l'exécute.

Succinctement, un modèle défini sur une *ETOILE*-signature contient :

- une famille d'ensembles de données indexée par D.
- une famille d'ensembles d'identités indexée par O.
- une famille d'ensembles d'états locaux indexée par $\{c\} \cup O$; De plus, afin d'introduire des aspects dynamiques, ces ensembles sont ordonnés afin de contrôler rigoureusement les effets de bords induis par les méthodes.

Les identités sont vues comme des données supplémentaires et les états sont utilisés comme des "modificateurs de sémantique" pour les méthodes.
Enfin, au contraire d'autres approches [SSC92], [FCSM91], [GD92], nous ne distinguons pas, parmi les méthodes, celles dont l'effet est de faire évoluer les états, et les opérations dont l'effet est d'observer l'état. Ces propriétés sont dynamiques et seront donc caractérisées par des axiomes (c.à.d. dans la spécification).

2.3.1.2 Les termes

Dans cette section, nous supposons donnée une *ETOILE*-signature $\theta = (S,F,M)$ et un ensemble de variables V.
A partir d'une signature, les premiers éléments syntaxiques considérés sont les termes. De façon classique, ces derniers sont inductivement définis à partir des fonctionnalités de la signature θ et l'ensemble des variables V. Nous avons alors des termes de la forme suivante :

- $f(t_1,...,t_n)$ où $(f : s_1 \times ... \times s_n \rightarrow s)$ est une fonction de l'ensemble F et chaque t_i un terme de type s_i.
- $t.m(t_1,...,t_n)$ où $(m : s_1 \times ... \times s_n \rightarrow s)$ est une méthode d'un des ensembles M^o, chaque t_i est un terme de type s_i et t un terme de type o.
- $m(t_1,...,t_n)$ où $(m : s_1 \times ... \times s_n \rightarrow s)$ est une méthode de M^c et chaque t_i est un terme de bonne type s_i (intuitivement, cela signifie que m est effective pour *self*).

De plus, comme nous avons une notion d'états implicites, l'ordre dans lequel les termes sont évalués n'est pas indifférent, contrairement à une approche fonctionnelle classique. Naturellement, nous introduisons le séquencement de termes au moyen de « ; ». Nous avons alors des termes de la forme: (t1 ; ... ; tn) où chaque ti est un terme défini comme ci-dessus. Intuitivement, dans un terme de séquencement (t1 ; ... ; tn), seule la valeur finale calculée par le dernier terme « tn » nous intéresse. Les autres termes de la séquence sont utilisés uniquement pour faire évoluer l'état avant de faire l'observation dénotée par le terme tn.
Enfin, il peut être utile de regarder l'état d'un objet particulier, identifié par un terme t, juste après avoir exécuté une séquence (t1 ; ... ; tn). Nous obtenons alors des termes de la forme suivante : $(t1 ; ... ; tn)\downarrow t$ (resp. $(t1 ; ... ; tn)\downarrow$).
Intuitivement, l'évaluation d'un terme dans un ETOILE-modèle est non-déterministe à cause des différentes évolutions possibles des états. Elle est fondée sur une stratégie "bottom-up".

2.3.1.3 Les formules

A partir des termes, nous pouvons maintenant définir les formules. De façon usuelle, ces dernières sont construites de manière inductive.
- tout d'abord les atomes équationnels :

- $(t_1 ; ... ; t_n) = (u_1 ; ... ; u_m)$ où t_n et u_m sont des termes de même type. Intuitivement, une instance d'un tel atome est satisfaite par un modèle si les évaluations respectives de t_n et u_m donnent la même valeur après que les séquences $(t_1 ; ... ; t_n)$ et $(u_1 ; ... ; u_m)$ aient été exécutées.
- $(t_1 ; ... ; t_n)\downarrow t = (u_1 ; ... ; u_m)\downarrow u$ où t et u sont de termes de même type $o \in O$. Intuitivement, une instance d'un tel atome est satisfaite par un modèle si l'état de l'objet identifié par le terme t après que la séquence $(t_1 ; ... ; t_n)$ ait été exécutée est égale à l'état de l'objet identifié par le terme u après que la séquence $(u_1 ; ... ; u_m)$ ait été exécutée.

- Afin de prendre en compte des aspects explicites de la concurrence (réactivité, interblocage,...), du traitement d'exceptions et autoriser une meilleure création d'objets, nous avons aussi trois prédicats : **succeed, wait** et **alive**. Nous obtenons alors les types d'atomes suivants :

- **succeed**(t_1 ; ... ; t_n) qui est satisfait par un modèle si l'évaluation de la séquence (t_1 ; ... ; t_n) aboutit toujours (c-à-d., ne bloque jamais lors de l'évaluation des termes t_i).
- **wait**(t_1 ; ... ; t_n) qui est satisfait par un modèle si l'évaluation de la séquence (t_1 ; ... ; t_n) n'aboutit jamais.
- **alive**(t_1 ; ... ; t_n) où t_n est un terme de type $o \in O$. Une instance d'un tel atome est satisfaite par un modèle si l'objet identifié par le terme est connu de self dans l'état atteint après l'évaluation de la séquence (t_1 ; ... ; t_n).

Les formules sont définies de façon inductive à partir des atomes ci-dessus, les connecteurs usuels dans $\{\neg,\wedge,\vee\}$ et les quantificateurs habituels $\{\forall,\exists\}$. De plus, comme la notion d'états implicites induit une notion de temps, nous introduisons deux nouveaux opérateurs : after et when.

- **after**[t_1 ; ... ; t_n](φ) où φ est une formule : une telle formule est satisfaite par un modèle si la formule φ est vraie juste après que la séquence (t_1 ; ... ; t_n) ait été exécutée.
- **when**[ψ](φ) où ψ et φ sont des formules : une telle formule est satisfaite par un modèle si à chaque fois que la formule ψ est observée vraie de façon instantanée (c'est le cas où l'on considère l'objet *self* comme isolé du monde), la formule φ est vraie.

Finalement, une *ETOILE*-spécification est un couple SP = (θ ,Ax) où θ est une *ETOILE*-signature et Ax un ensemble de formules définies sur θ.
La sémantique associée à une *ETOILE*-spécification est alors définie par la classe des modèles de la signature qui satisfont les axiomes de la spécification.

2.3.2 Système de types d'objets

La seconde partie de notre formalisme concerne la spécification d'un système de types d'objets. Naturellement, nous définissons un système de types d'objets par une collection d'étoiles. Schématiquement, nous représentons une signature de système par une collection d'*ETOILE*-signatures auquel nous ajoutons des "conditions de recollement" afin d'obtenir des systèmes cohérents. Ces conditions de recollement traduisent le fait que les branches d'une étoile ne peuvent être instanciées que par le centre d'une étoile du système et elles imposent que l'ensemble des méthodes d'un type c utilisées dans un type d'objet c' soit un sous-ensemble de l'ensemble des méthodes que l'on peut trouver dans la signature dont le type d'intérêt est c. En fait, ceci revient à contrôler que chaque branche d'un type d'objets donné définisse bien une abstraction (une vue simplifiée) du comportement global de l'objet utilisé.
Ainsi, une spécification de système est définie par un ensemble de spécifications de types d'objets munies de liens d'abstraction auquel on ajoute un ensemble de formules globales (c.à.d., des formules définies à partir de l'ensemble des fonctionnalités du système). De telles formules ont pour but d'exprimer des propriétés globales que l'on ne peut exprimer au niveau local.
La sémantique associée à une spécification de système de types d'objets est alors définie par une collection de modèles, chacun validant une *ETOILE*-spécification du système. De

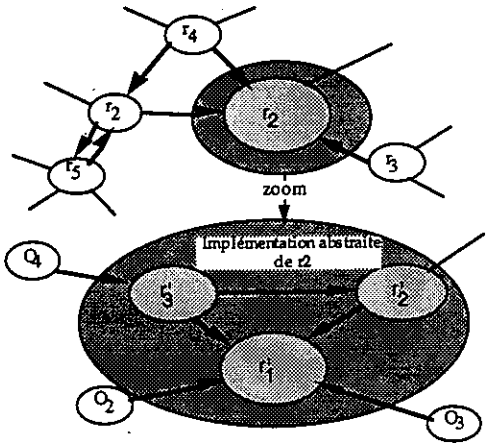
plus, on impose des conditions de compatibilité entre les types d'objets afin de ne considérer que des modèles cohérents du système.

2.3.2.1 Implantation abstraite et raffinement

Un apport décisif pour les spécifications formelles est de permettre le développement de logiciels par étapes successives de raffinements de spécifications jusqu'à l'obtention d'un programme ou d'une spécification exécutable. Chaque étape de raffinement ou encore implantation introduit de nouvelles décisions de conception. Ces dernières peuvent être aussi variées que le choix d'un algorithme particulier ou la représentation de certaines opérations de plus haut niveau par des opérations de plus bas niveau ou encore un choix de représentation de données (par exemple, la représentation de la structure des piles par des tableaux et les entiers naturels).
Le principal avantage d'une telle approche est que les formalismes utilisés à tous les niveaux d'implantation sont les mêmes. Ainsi, ils partagent tous les mêmes techniques de preuves. En conséquence, les preuves de correction sont plus faciles parce qu'elles peuvent être faites dans une logique homogène. Nous pouvons alors prouver que l'implantation de la spécification de plus haut niveau par la spécification de plus bas niveau est correcte et ainsi, montrer que la spécification finale se comporte de la manière spécifiée.

Dans le cadre des ETOILE-spécifications, comme ces dernières peuvent être vues comme des extensions des formalismes algébriques classiques, le concept d'implantation prendra en compte en plus des cas classiques précités ci-dessus, l'implantation d'un type d'objets et son extension au système, c.à.d. l'enrichissement d'une spécifications de système de types d'objets pour obtenir une spécification de système de type d'objets plus concrète.

Bien entendu, le concept d'implantation devient intéressant lorsque nous sommes capable de fournir des critères de correction. Des méthodes générales de preuves peuvent alors en découler.
Dans le cadre des ETOILE-spécifications, la correction consiste à traduire algébriquement le fait que l'implantation « simule » complètement le comportement du type d'objets implanté. Ceci revient à montrer que tous les axiomes de la spécification de haut niveau sont des théorèmes de la spécification plus concrète. En conséquence, nous pouvons utiliser toutes réalisations de l'implantation comme une réalisation possible de la spécification du type d'objets implanté.



Cette notion de correction ne met en jeu que la spécification du type d'objets et son implantation, indépendamment de sa position éventuelle à l'intérieur d'un système Sys donné. Par conséquent, rien n'assure à priori que parmi les réalisations de l'implantation, certaines puissent être utilisées pour définir une réalisation du système Sys.

Naturellement, ceci nous a amené à étudier la réutilisation d'implantations et d'obtenir le résultat important suivant déjà cité ci-dessus :
Si Sys implante correctement un type d'objets o dans un système Sys', et Sys' valide une formule φ, alors le système Sys'[o ←Sys] valide aussi la formule φ.
Un tel résultat nous dit que l'on peut implanter chacun des sous-systèmes de Sys' sans connaître la réalisation choisie (ou qui sera choisie) du système complet Sys'. Ceci nous permet alors d'obtenir un développement de logiciel structuré ou « modulaire », propriété essentielle pour obtenir des logiciels de qualité (c.à.d. extensibles et réutilisables (cf. [MEY88]).

3. Le partitionnement

L'algorithme proposé dans ce document, destiné à être intégré dans notre environnement de développement, cherche à réduire le coût global du produit tout en respectant les performances. Mais c'est l'ordonnancement des fonctionnalités qui est modifié pour rechercher le meilleur compromis Logiciel/Matériel.

3.1 Hypothèses et méthodologie

La description initiale du système est un graphe direct sans cycle (DAG) G = (N, A). Ce graphe est composé d'un ensemble de noeuds N représentant les parties de calculs (tâches ou processus) du système et d'un ensemble d'arcs A décrivant les précédences de contrôle et de données entre ces noeuds. Il est en principe extrait de la spécification la plus raffinée.Cette extraction est actuellement manuelle. Des recherches sont en cours pour l'assister et l'automatiser.
On suppose connu pour chacun des noeuds deux réalisations possibles (l'une en Matériel et l'autre en Logiciel) ainsi que les différentes caractéristiques qui permettent de quantifier la qualité des solutions obtenues. Parmi ces caractéristiques sont considérés les temps d'exécution d'une réalisation matériel et d'une réalisation logicielle sur le processeur choisi comme cible technologique. Ces temps d'exécution doivent être bornés et calculables.
L'implémentation est imposée si une seule réalisation est connue pour un noeud.
L'architecture cible est une architecture monoprocesseur composée d'un processeur (partie logicielle) et de composants externes ou d'un ou plusieurs ASICs (partie matérielle).
La méthodologie de partitionnement est résumée dans la Figure 2.

3.1.1 But de l'algorithme

Chaque réalisation possible implantant la fonctionnalité d'un noeud du graphe fait appel à une ou plusieurs ressources $R_i^{logiciel} = \{l_q, l_{q+1}, ..., l_{q+p}\}$ & $R_i^{matriel} = \{m_r, m_{r+1}, ..., m_{r+s}\}$. Si un coût est attribué à chacune de ces ressources, chaque réalisation possible est ainsi caractérisée par un temps d'exécution et un ensemble de ressources (donc de coûts). En considérant que le coût d'une ressource est nul si celle ci n'est pas utilisée par une autre fonctionnalité à une date donnée, le coût d'un noeud est inférieur ou égal à la somme du coût de chacune des ressources.
Le but de l'algorithme est de trouver un ordonnancement et une implémentation pour tous les noeuds du graphe, pour un coût le plus faible possible. La solution doit respecter une contrainte de temps globale S_{max} c'est à dire que l'exécution de tous les noeuds du graphe soit terminée à la date S_{max}.
Le coût à minimiser est lié aux ressources. Le partage des ressources entre les différents noeuds du graphe réduit le coût global. C'est donc l'un des objectifs visés.

3.2 Principe de la méthode

L'algorithme proposé est une variation de l'algorithme "Force Directed Scheduling" de P. PAULIN et J.P. KNIGHT [PAU89]. Cet algorithme développé pour la synthèse de haut niveau cherche à réduire le nombre de ressources utilisées par modification de l'ordonnancement des tâches.

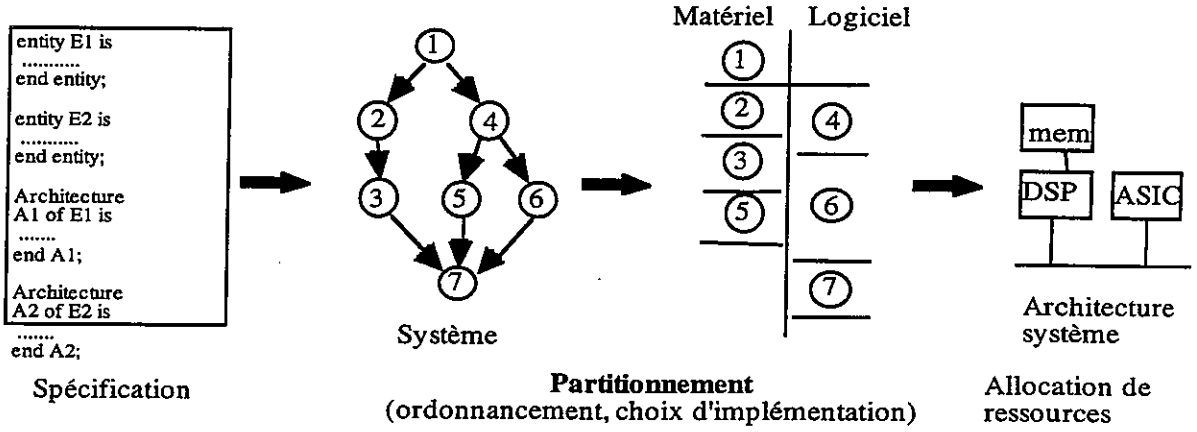


Figure 2: Méthodologie de Partitionnement

Le principe de la méthode est de chercher le coût pour chaque implémentation à chaque date de l'intervalle de temps pour tous les noeuds du graphe. Pour cela on a défini un couple de forces appelé "forces de répulsion" $F_j(i)$. Ces forces représentent le coût d'affectation du noeud i à une date j pour une implémentation (logicielle ou matérielle).

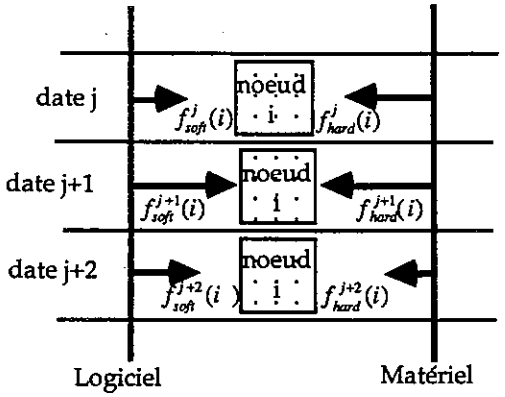
$F_j(i) = \{f^j_{soft}(i), f^j_{hard}(i)\}$

$f^j_{soft}(i)$est appelée "force de répulsion logicielle" et $f^j_{hard}(i)$ est la "force de répulsion matérielle".

Graphiquement, ces forces traduisent les contraintes à chaque dates, et plus la force est grande, plus le coût sera élevé. La Figure 3 donne un exemple de ces forces calculées pour le noeud i aux dates j, j+1, j+2 pour les deux implémentations possibles.

Le calcul de ces forces doit tenir compte des caractéristiques propres au noeud considéré, et de l'influence qu'entraîne son assignation à une date précise sur les autres noeuds. Pour cela, chaque force de répulsion est la somme de 2 forces : Une force propre et une force induite totale.

La force propre reflète le coût de l'assignation de cette tâche à cette date : C'est un coût local equivalent à la somme des coûts des ressources utillisées. *La force induite* traduit les conséquences sur les autres tâches. En effet, l'assignation d'une tâche à une date entraîne une modification de la mobilité des autres tâches et modifie donc les implémentations possibles. La force induite totale est la somme des forces induites sur tous les autres noeuds.



- **Figure 3 Exemple des différentes forces de répulsion**

Une fois calculé toutes les forces de répulsion, le noeud, pour lequel la force de répulsion est la plus faible est sélectionner. Son implentation et sa date d'ordonnancement sont alors déduite directement par cette force. Le processus est alors réitérer.

L'algorithme globale est le suivant :

Répéter jusqu'à ce que tous les noeuds soient traités
Evaluer les intervalles de temps avec ASAP et ALAP
Pour tous les noeuds, à toutes les dates, pour les deux implémentations
    Calculer les forces propres
    Calculer les forces induites
    Calculer les forces de répulsion
Sélectionner un noeud
Chercher sa force de répulsion la plus petite
Affecter ce noeud à cette date et avec cette implémentation
fin

Pour calculer la force induite, l'algorithme est le suivant :

Pour tous les noeuds non assignés
Evaluer les intervalles de temps avec ASAP et ALAP
A toutes les dates, pour les deux implémentations
    Calculer les forces propres
Calculer la moyenne arithmétique des forces propres
fin

3.3 Partitionnement Logiciel/Matériel d'un système de télécommunication

L'exemple traité pour valider notre démarche est un système d'annulation d'écho acoustique basé sur l'algorithme GMDFα (Generalized Multi-delay Frequency-Domain Adaptive Filter). Cet algorithme est un compromis entre vitesse de convergence et complexité. Le principe de ce système est de mettre à jour les coefficients d'un filtre adaptatif, en cherchant à minimiser l'énergie du signal d'erreur (différence entre le signal d'écho estimé et le signal d'écho réel). Pour estimer cet écho, on effectue le produit de convolution des échantillons d'entrée par la réponse impulsionnelle du filtre (représenté par les coefficients du filtre). Le passage dans le domaine fréquentiel ramène le calcul du produit de convolution à un produit complexe. Son implémentation nécessite des opérateurs FFT et FFT^{-1} pour passer du domaine temporel au domaine fréquentiel (et vice versa), et un opérateur effectuant les produits complexes.

Cette application est composée de fonctionnalisés qui peuvent être réalisées en même temps. Ces fonctionnalisés présentent des dépendances dues aux séquencements dans le temps et aux communications de données. L'exploitation du parallélisme intrinsèque à l'application consiste à gérer ces dépendances. La représentation adaptée est une représentation sous forme de graphe.
La granularité qui fixe la taille des noeuds n'est pas figée. Nous avons choisi de ne pas décomposer une fonctionnalité si une implémentation existe dans nos bibliothèques d'opérateurs. Par exemple, nous traiterons la fonctionnalité FFT sans la décomposer, considèrent qu'un opérateur matériel FFT existe, ainsi qu'un code pour le processeur de notre architecture cible.
Pour cette application, c'est dans l'étape de partitionnement que l'implémentation est déterminée pour les différentes fonctionnalisés. De plus une optimisation des ressources est nécessaire pour limiter le coût global, ou pour respecter les contraintes imposées. Cette optimisation dépend de l'ordonnancement des fonctionnalisés.
On peut ainsi appliquer l'algorithme de partitionnement présenté précédemment sur le graphe de ce système et obtenir des partitionnements différents suivant la fonction de coût choisie.

Le prototypage de cette application (architecture cible basée sur un DSP56002) a permis de valider notre approche, de vérifier les timmings, et de déceler les difficultés liées aux volumes de données à traiter dans le processeur (mise en mémoire cache des données et du code, utilisation de DMA).

4. Conclusion

Les travaux présentés dans cet article résultent d'une coopération étroite entre les équipes de génie logiciel et d'architecture du laboratoire LaMi d'Evry.
Notre but est la définition de techniques de conception matérielles et logicielles ciblant des architectures hétérogènes avec des études de cas orientées télécommunication. L'approche que nous suivons est essentiellement fondée sur quatre choix :

- l'usage de méthodes orientées objet,
- l'étude de spécifications formelles adaptées à notre problème, assistées par des outils graphiques,
- un partitionnement s'appuyant sur un découpage en fonctionnalités structurées en graphe,
- l'exploitation systématique de bibliothèques pour effectuer les raffinements de spécification et pour guider le partitionnement logiciel/matériel.

En l'état actuel de notre recherche, nous disposons :

- d'un environnement de spécification fondé sur les ETOILE-spécifications prévu pour pouvoir utiliser des bibliothèques,
- d'un algorithme de partitionnement logiciel/matériel basé sur un découpage en unités fonctionnelles. Le partitionneur est pour l'instant appelé manuellement.

Nous utilisons l'environnement de spécification pour écrire les spécifications de haut niveau et les raffiner vers des spécifications détaillées. A partir de celles-ci, nous pouvons déduire (manuellement) le graphe des fonctionnalités nécessaire au partitionnement. Tout ce processus (aussi bien le raffinement que le partitionnement) est guidé par l'utilisation de bibliothèques de composants déjà disponibles. La connaissance de ces composants et de leur mode de communication peut ainsi être pris en compte pour une meilleure adéquation des choix de raffinement.
Nos efforts actuels portent sur l'extraction des fonctionnalités d'entrée du partitionnement à partir des spécifications détaillées. Nous visons à proposer des méthodes systématiques d'extraction. Ceci suppose d'une part l'explicitation de certaines contraintes au sein des ETOILES-spécifications et d'autre part la prise en compte de l'aspect orienté objet dans VHDL.

Références

[FCSM91] J. Fiadeiro, J.F. Costa, A. Sernadas, and T. Maibaum, « Objects Semantics of Temporal Logic Specification », 8th Workshop on Specification of Abstract Data Types joint with the 3rd COMPASS Workshop, Dourdan Springer-Verlag LNCS 655, pp. 236-253, 1991.
[GD92]J.A. Goguen, and , R. Diacanescu, « Towards an Algebaic Semantic for the Object Paradigm », Recent Trends in Data Type Specification, Selected Papers of the 5th Workshop on Specifications of Abstract Data Types joint with 4th COPASS Workshop, Caldes de Malavella, Spain, pp. 1-29, October 1992.
[MEY88] B. Meyer, : « Object-oriented software construction », Prentice-Hall, 1988.
[PAU89] P.G. Paulin, J.P. Knight, « Force-directed Scheduling for the behavorial Synthesis of ASIC's » IEEE trans. On Computer-aided Design, Vol. 8, n° 6 pp. 661-679, Juin 1989.
[SSC92] A. Sernadas. , C. Sernadas and J.F. Costa, : « Objects Specification Logic », Internal report, INESC, University of Lisbon, 1992.