# Modélisation d'architecture pour le reciblage et la paramétrisation d'une chaîne de compilateur DSP

**Deruyter Thomas,**

Freescale
BuroClub
29 bvd des Alpes
38246 – Meylan
Tel : 04 76 61 88 40
Fax : 04 76 61 44 44
Thomas.Deruyter@freescale.com

**Fernandez Jean-Claude,**

Verimag Imag
Centre equation
2 av de Vignate
38160 - Gières
Tel : 04 56 52 03 79
Fax : 04 56 52 03 44
Jean-Claude.Fernandez@imag.fr

**Potet Marie-Laure**

LSR Imag
BP72
681 rue de la passerelle
38402 - St Martin d'heres
Tel : 04 76 82 72 69
Fax : 04 76 82 72 87
Marie-Laure.Potet@imag.fr

**Résumé :** Les compilateurs reciblables, par opposition aux compilateurs paramétrés, visent à réutiliser un même cœur de technologie pour différentes architectures. De ce fait, les compilateurs reciblables nécessitent une description complète et détaillée de l'architecture cible aussi bien au niveau du jeu d'instructions que des ressources. Les architectures du traitement du signal (DSP) sont principalement utilisées dans les systèmes embarqués pour des applications temps réel. De ce fait, ces architectures sont complexes et présentent des particularités architecturelles telles que des ressources dédiées et des instructions spécifiques. De plus, pour réduire la consommation d'énergie, les processeurs DSP présentent souvent des espaces mémoire séparés avec des accès spécifiques. De ce fait, les architectures DSP sont dites irrégulières et la compilation pour ces architectures est très complexe, en particulier les phases de sélection d'instruction et d'allocation de registres qui doivent prendre en compte les spécificités de l'architecture. Ainsi, les compilateurs DSP reciblables s'appuient sur des langages de descriptions d'architectures qui permettent de décrire le jeu d'instructions ainsi que les contraintes sur les opérandes, telles que les registres autorisés, les modes d'adressages ainsi que les relations entre opérandes. De plus, le processus de compilation doit être contrôlé pour assurer que les irrégularités de l'architecture sont prisent en compte.

**Mots clés** : Modèles d'architectures, Compilation reciblable, vérifications formelles, satisfaction de contraintes.

# *A model-based approach to parameterize a retargetable DSP compiler chain.*

***Abstract:*** *Compiler retargetability is the capability to adapt a compiler technology for different processors. By opposition to parameterized compilers which only permit to adjust some compiling parameters as number of registers or cost of instructions, retargetable compilers require a complete description of the target architectures including the instructions set, registers and memory structure. Because digital signal processors (DSP) are dedicated to embedded signal applications, they present a variety of specificities as instructions well-adapted to classical signal treatments. Moreover, in order to master energy consumption, DSP architectures generally offer specialized units with special-purpose registers. Thus DSP architectures are qualified as irregular architectures and the compiling process is made more difficult. In particular the instruction selection phase and the allocation registers phase strongly depend on the target architectures. As a consequence a retargetable DSP compiler chain must be based on an architecture description language which allows describing instructions set and constraints about instruction operands as authorized registers, addressing modes and relations between operands. Moreover the compiling process must be controlled in order to verify that irregularities are taken into account in a right way.*

***Keywords:*** *architecture models, retargetable tools, formal verification, constraints satisfaisability.*

## 1 – Introduction

Les DSP (Digital Signal Processors) sont les processeurs du traitement du signal utilisés dans les secteurs de l'embarqué tels que l'aéronautique, les contrôleurs de voitures (GPS, ABS, ...) et la téléphonie mobile. De part les contraintes induites par l'embarqué leur architecture ainsi que leur programmation diffèrent de celles des processeurs classiques. La plupart des architectures DSP sont équipées de plusieurs unités de calcul pouvant s'exécuter en parallèle. Par exemple, le StarCore 140 [1] (architecture Motorola) est équipé de quatre unités arithmétiques et logiques (UAL) et de deux unités de calcul d'adresses (UGA). Cette architecture permet donc d'exécuter jusqu'à six instructions en un même cycle. Par opposition aux architectures RISC, les architectures DSP sont dites non régulières. En effet, leur jeu d'instructions peut présenter des dissymétries. Ainsi, sur le StarCore 140, l'instruction de transfert d'octets dans le mode d'adressage post-incrémenté n'est possible que pour transférer la valeur d'un registre vers la mémoire et non l'inverse. De ce fait, le transfert d'une valeur depuis la mémoire vers un registre peut nécessiter l'utilisation de deux instructions. De plus, les DSP présentent des ressources spécialisées. Ainsi, on ne trouve pas de bancs de registres généraux mais plusieurs bancs de registres spécialisés et attachés aux unités de calcul.

Les applications ciblant ces architectures sont souvent des applications temps réel. De ce fait, la compilation pour les DSP se doit de produire du code extrêmement performant, aussi bien en terme de vitesse que de taille de code. Du plus, ces architectures étant principalement utilisées dans le domaine de l'embarqué, la gestion de la mémoire est un problème majeur. Dans la plupart des DSP, le modèle mémoire n'est pas homogène, ce qui permet une meilleure gestion de l'énergie. En effet, la mémoire est souvent scindée en plusieurs espaces avec des coûts d'accès en terme de temps et de consommation d'énergie très différents. Il est donc important que le compilateur effectue un choix optimal pour le placement des opérandes en mémoire, de sorte à minimiser les temps d'accès et la consommation. De part la présence d'instructions spécifiques et de ressources dédiées, l'un des principaux enjeux de la compilation pour les architectures DSP réside en la qualité de la sélection de code. Pour être efficace, il a été montré que cette sélection doit être couplée à l'allocation des ressources [2], [3]. En effet, les ressources étant dédiées aux unités fonctionnelles, l'allocation d'une ressource pour une variable doit tenir compte des futures utilisations de cette même variable, afin de minimiser le nombre de transferts. Par exemple, sur le StarCcore 140, une addition entre deux entiers peut aussi bien être réalisée par une UGA que par une UAL avec un coût identique. Cependant selon que le résultat de l'addition est utilisé pour un calcul arithmétique ou pour un accès mémoire, le code résultat sera très différent. Par exemple, considérons le code C suivant :

```
for ( i = 0 ; i < 32 ; i++ ) {  tab[i] = i+3; }
```

Une optimisation classique consiste à calculer l'adresse de ***tab*** puis à incrémenter directement cette adresse dans la boucle plutôt que de recalculer à chaque fois l'adresse à affecter. Une traduction optimisée exploitant les spécificités de l'architecture choisira automatiquement d'effectuer les calculs liés à l'adresse des éléments de ***tab*** sur une UGA, et donc placera cette adresse dans un registre associé à cette unité fonctionnelle.

**Traduction optimisée :**

```
   move    #_tab,r0        \\ r0 contient l'adresse de tab
   move    #<3,d0          \\ on stocke 3 dans d0
Loop_start
   move    d0,(r0)+        \\ on stocke la valeur de d0 à l'adresse r0 puis r0 est incrémenté.
   inc     d0               \\ on  incrémente d0
Loop_end
```

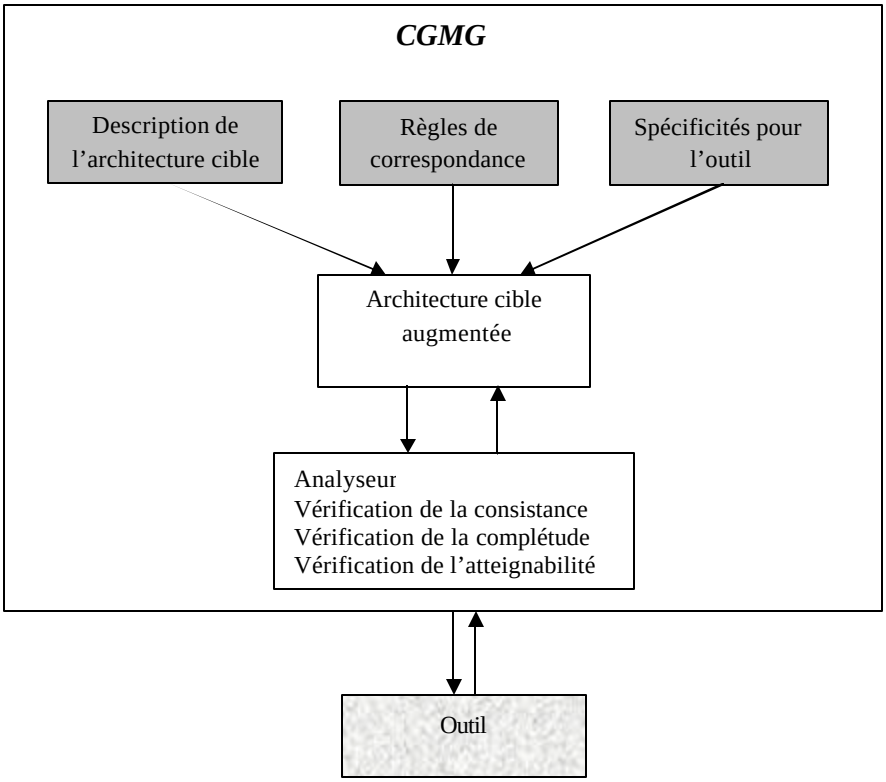
Ici, les registres ***d*** sont des registres de données et les registres ***r*** des registres d'adresses. Si le choix des registres n'est pas fait de manière adaptée, l'adresse pourrait être placée dans un registre de données, ce qui aurait conduit à utiliser une instruction de transfert supplémentaire. Contrairement à l'architecture classique d'un compilateur dans laquelle la phase de sélection d'instructions est indépendante de la phase d'allocation des registres, les architectures irrégulières nécessitent un fort couplage de ces deux phases afin de pouvoir sélectionner au mieux des instructions tout en évitant des séquences de transfert inutiles [3] [4].

Un autre problème majeur dans la compilation pour les DSP est la réutilisation du coeur de la technologie. En effet, pour atteindre le niveau de performance exigé par les clients, les équipes de développeurs de compilateurs ont mis au point des algorithmes d'optimisation extrêmement performants, mais aussi extrêmement complexes. De ce fait, l'un des enjeux est de permettre une réutilisation de ces algorithmes lors du développement des outils associés à une nouvelle architecture. Cette nécessité est renforcée par l'évolution très rapide des architectures DSP qui impose une réactivité importante de la part des développeurs de compilateurs afin de répondre aux exigences du marché.

Ainsi, il apparaît clairement qu'une solution pour la réutilisation et le reciblage des technologies de compilation dans le domaine des DSP est la paramétrisation des outils par des modèles d'architecture. L'approche que nous présentons est basée sur ce principe.

## 2 – L'APPROCHE

L'architecture retenue est présentée ci-après :




La description de l'architecture cible correspond aux informations du manuel de référence. En dehors des aspects structuraux (mémoire, forme des registres, jeu d'instructions) cette description devra contenir des informations plus spécifiques, comme les contraintes sur les opérandes d'entrées et de sorties des instructions. L'architecture est décrite à l'aide d'un langage défini par une grammaire qui a été conçu pour suivre au plus près les manuels de référence des architectures et faciliter la tâche de l'utilisateur.

Les règles de correspondance établissent le lien entre les instructions du langage intermédiaire de la chaîne d'outils et les instructions de l'architecture cible. Elles définissent en quelque sorte la sémantique des instructions du langage machine et sont utilisées par les outils pour la phase de sélection de code.

Les spécificités pour un outil correspondent aux informations liées à l'architecture cible et nécessaires pour cet outil. Par exemple, c'est dans cette partie du modèle qu'on décrira les registres virtuels qui représentent une extension du nombre de registres présents sur la machine.

Le langage de description de l'architecture cible offre des constructions proches des langages tels que nML [5] ou MARION [6] et autres [7] [8], ainsi que des travaux tels que flexCC2 [9]. L'originalité de l'approche Freescale réside en l'explicitation des contraintes sur les opérandes, point délicat pour la production de code correct et efficace, comme vu précédemment. Ce choix est motivé par le fait que la prise en compte des contraintes de placement est une source importante d'erreurs ou de sous optimalité du code produit, en particulier lors du reciblage du cœur de la technologie en vue d'une nouvelle architecture. L'objectif est d'associer à ce modèle des outils permettant d'effectuer certaines vérifications liées aux contraintes de placement. Ces vérifications vont porter sur le modèle et sur le code produit.

La principale difficulté lors de l'écriture du jeu d'instructions consiste en l'extraction des contraintes. En effet, les contraintes sont souvent disséminées dans plusieurs sections du manuel de référence. Ainsi, certaines contraintes se retrouvent dans la réécriture binaire de l'instruction, par exemple lorsque deux opérandes sont codées par un seul groupe de bits, d'autres sont liées à la nature de l'instruction, et donc décrites dans une section à part, et finalement certaines sont explicitement données lors de la description de l'instruction. Il est donc nécessaire de disposer d'un outil permettant de détecter les instructions pour lesquelles l'ensemble des contraintes est trop restreint voir insatisfaisable. Cet outil permettra à la fois de vérifier le modèle (a-t-on introduit des erreurs) et de quantifier la difficulté d'utilisation d'une instruction (beaucoup de restrictions peuvent influencer le coût d'une instruction). Le second outil de vérification aura pour objectif de garantir la correction du code produit, eu regard aux contraintes décrites dans le modèle.

Dans ce papier, nous ne présenterons que le modèle de la machine physique (description de l'architecture cible) ainsi que les vérifications des contraintes exprimées dans ce modèle. La section suivante présente le sous-ensemble du langage de description ARCHL permettant la description du jeu d'instructions ainsi que les contraintes associées. Ensuite nous verrons comment les contraintes sont vérifiées, puis nous présenterons quelques résultats expérimentaux.

**3 – Le langage**

Comme dans la plupart des langages de description d'architecture, une description en ARCHL se décompose principalement en deux modules. Le premier correspond à la description des ressources (mémoires, registres, etc.) ainsi qu'à la description du jeu d'instructions. Le second module correspond à description des règles de réécriture du langage de haut niveau vers le langage de bas niveau. Dans la suite du document nous ne décrirons en détail que le sous-ensemble des éléments qui mettent en jeu les contraintes.

**3.1 – Les registres**

Les principales ressources sont les registres, les mémoires et en particulier les modes d'accès à ces mémoires. Un registre est défini par son nom, sa taille en nombre de bits et son type d'accès (lu, écrit, protégé, constant). Un registre constant est un registre qui peut être directement utilisé par le programmeur mais dont la valeur n'est jamais modifiée. Un registre protégé est un registre dont l'accès explicite n'est pas autorisé. Dans les processeurs, les registres peuvent aussi être décrits sous forme de bancs de registres. Un banc de registres est une liste de registres contigus et identiques, où chaque registre peut être accédé par son indice. Pour cette raison, un banc de registres est déclaré comme un registre simple avec un indice inférieur et un indice supérieur. Un registre peut souvent être décomposé en plusieurs segments. Pour exemple nous pouvons vouloir identifier la partie de poids fort et la partie de poids faible d'un registre. C'est pourquoi il est nécessaire de permettre la description des « tranches » de registre. La définition d'un registre prend la forme :

```
d [0..15] {
    size 40;
    access write;
    decomp .e:[39..32],.h:[31..16],.l:[15..0];
}
```

Ici, on déclare un banc de 16 registres, de nom d et d'indices 0 à 15, accessibles en lecture et écriture et qui se décomposent en 3 tranches, une tranche d'extension (.e) de 8 bits, une tranche des bits de poids forts (.h) et une des poids faibles (.l) de 16 bits.

### 3.2 – Les listes de registres

Les opérandes d'instructions prennent généralement leur valeur dans des listes de registres autorisés, ou des accès mémoire. De ce fait, le langage ARCHL permet de déclarer des listes de registres de la façon suivante :

```
D16 = d[0..15] ; // les registres d0 a d15
Re = r0,r2,r4,r6,r8,r10,r12,r14 ; // les registres d'indice pair
Ro = r1,r3,r5,r7,r9,r11,r13,r1 5; // les registres d'indice impair
```

### 3.3 - Les contraintes

L'une des particularités de notre langage est l'explicitation des contraintes associées aux opérandes d'une instruction ou d'un accès mémoire. La grammaire permettant la description de ces contraintes est la suivante :

```
<Contrainte> :=
    Index(<idf>) <Operateur> Index(<idf>) + <Num>
    ¦ Reg_name(<idf>) <Operateur> Reg_name(<idf>)
    ¦ Decomp_name(<idf>) <Operateur> Decomp_name(<idf>)
    ¦ <idf> <Operateur> <idf>

<Operateur> := == ¦ !=
```

Les contraintes portent sur des opérandes qui peuvent être de deux natures différentes, soit des registres, soit des accès mémoire. On vérifie que les opérandes apparaissant dans une contrainte sont de même nature. Ainsi, on peut déclarer que deux opérandes doivent utiliser un registre de même indice ou que l'écart entre les indices de deux opérandes est fixe. De même il est fréquent de voir des contraintes liant les entrées et les sorties d'une instruction, comme par exemple pour spécifier qu'une instruction réutilise un opérande d'entrée en sortie.

### 3.4 – Les accès mémoire

Les accès mémoire ont différentes structures (directs, indirects, indexés, …). Nous avons choisi de les décrire par des schémas en nous inspirant de ce qui est fait dans le langage nML. De façon globale, on peut considérer un accès mémoire comme un opérande complexe, composé de sous-opérandes qui sont soit des valeurs immédiates soit des registres appartenant à des listes particulières de registres. Par exemple :

```
Indexed_by_reg {    inputs op1:Re , op2: Ro;
                    constraints Index(op2) == Index (op1) + 1;
                    dump_pattern "(@ + @)", op1, op2;
                    … }
```

Cet exemple décrit une forme d'accès indirect indexé pour lequel l'adresse accédée correspond à la somme du contenu des deux opérandes op1 et op2. Dans cet exemple, il y a une contrainte entre l'indice de l'opérande op1 et l'indice de l'opérande op2. Les couples possibles pour cette instruction seront donc (r0,r1), (r2,r3) …

### 3.5 – Les instructions

Pour chaque instruction on déclare l'ensemble des opérandes d'entrée et de sortie, les contraintes, le coût, la réécriture en binaire ainsi qu'un certain nombre d'éléments que nous ne détaillerons pas ici. Les opérandes d'une instruction peuvent être de natures différentes. Ainsi, un opérande d'instruction peut être soit une valeur immédiate, soit un registre ou encore un accès mémoire. Une instruction prend donc la forme suivante :

```
ADD {
    case IMM_U5 {
            inputs op1:#u5, op2:D16;
            outputs op3:D16;
            constraints op2 == op3;
            …       }
… }
```

Ici on déclare l'instruction d'addition entre une valeur immédiate non signée, donnée sur 5 bits, et un registre de la liste D16, le résultat étant dans un registre de la liste D16. La contrainte associée à cette instruction impose que le registre utilisé en opérande d'entrée (op2) soit réutilisé en sortie (op3). Lorsqu'un opérande est de nature accès mémoire les contraintes peuvent aussi porter sur les sous-opérandes de l'accès mémoire. L'accès aux sous-opérandes se fait à l'aide d'une notation pointée. Par exemple si on a *op1 : Indexed_by_reg*, la contrainte *op1.op1 == op1.op2* impose que les deux registres constituant un accès mémoire *Indexed_by_reg* soient identiques.

**4 – Vérifications et outils**

En raison de la complexité des architectures et de leur description dans les manuels de référence, il est nécessaire de disposer d'outils permettant de valider les modèles. Le premier outil, VERIF-MODEL, se propose de valider les contraintes associées aux instructions, en calculant les solutions attachées aux instructions. Cet outil est basé sur un solveur de contraintes, CGMG-CSP, permettant de traiter des contraintes sur des domaines finis. Le second outil, VERIF-CODE, est chargé de vérifier la conformité des codes assembleur produit, eu égard aux contraintes. Cet outil est lui aussi basé sur le solveur CGMG-CSP.

**4.1. CGMG-CSP**

Le langage de contraintes décrit en section 3.3. permet d'énoncer des égalités sur des opérandes représentant des registres ou des accès mémoire. L'égalité est définie de la manière suivante :

- L'égalité sur les registres est une égalité structurelle qui impose l'identité du nom du registre, l'égalité de l'indice si ces registres sont indicés et l'identité des noms de décomposition si ces registres sont décomposables.
- L'égalité entre accès mémoire impose l'identité du nom du schéma d'accès mémoire et l'égalité des sous-opérandes de l'accès mémoire. Un opérateur d'égalité structurelle est donc construit automatiquement pour chaque schéma d'accès mémoire.

Le solveur de contraintes travaille à partir d'un ensemble de contraintes ($C_1$, … $C_k$) portant sur un ensemble de variables ($v_1 : D_1$, … $v_n : D_n$) où $D_i$ désigne le domaine fini des valeurs possibles pour $v_i$. Le solveur CGMP-CSP travaille par énumération des domaines $D_i$ et implémente différents algorithmes (chronological backtracking, forward checking) [10]. Il calcule l'ensemble des solutions possibles pour un jeu de contraintes et des ensembles de valeurs. Dans la suite on note *P=(C, V)* une entrée du solveur CGMG-CSP où *C* est l'ensemble de contraintes et *V* l'ensemble des variables avec leur domaine de valeurs.

**4.2 Instanciation d'un jeu de contraintes**

Un renommage *r* pour un problème *P*=(C, V) est un ensemble de couple $(v_i, w_i)$ où $v_i$ est une variable de *V* et $w_i$ une nouvelle variable ayant le même domaine de valeurs que $v_i$. Un renommage doit être une fonction injective : une variable $v_i$ est renommée au plus une fois et aucune égalité implicite n'est ajoutée par le renommage. Soit *P=(C, V)* un système de contraintes, une instanciation *P'=(C', V')* de *P*, pour un renommage *r*, se construit de la manière suivante :

- C'est l'ensemble des contraintes obtenu par recopie de C en remplaçant chaque occurrence de $v_i$ par $w_i$.
- *V'* est obtenu à partir de *V* en substituant les variables $v_i$ par $w_i$.

Les conditions données précédemment sur les renommages garantissent que V' défini un domaine de valeurs unique pour chaque variable $w_i$. Dans la suite on notera $[v_1 := w_1, \ldots, v_n := w_n]$ (C, V) le système de contraintes obtenu par renommage.

**4.3. Vérification du modèle**

Soit OPER $op_1 : D_1$ , …, $op_n : D_n$ la définition d'une instruction à laquelle est associée l'ensemble de contraintes $C_{oper}$. Le jeu de contraintes associé à cette instruction est obtenu en complétant $C_{oper}$ par les contraintes inhérentes aux opérandes de nature accès mémoire. Soit $op_i : D_i$, avec $D_i$ un nom d'accès mémoire dont l'ensemble des contraintes associées est $(C_i, V_i)$. Le système de contraintes $(C_i', V_i')$ associé à l'opérande $op_i$ est :

$$[v_1 := op_i.v_1, \ldots, v_n := op_i.v_n] (C_i, V_i)$$

avec $v_1$, …, $v_n$ les variables de $V_i$ (i.e. les sous-opérandes de l'accès mémoire). L'ensemble des contraintes associé à l'instruction est donc :

$$((C_{oper} \cup \{C_i' | op_i \in AM\}), (\{op_1 : D_1, \ldots, op_n : D_n\} \cup \{V_i' | op_i \in AM\}))$$

où AM est l'ensemble des opérandes de l'instruction de type accès mémoire.

L'outil VERIF-MODEL construit l'ensemble des contraintes associées à chaque instruction du modèle, utilise le solveur de contraintes CGMG-CSP pour calculer les solutions et présente les résultats sous la forme d'un tableau.
Voici par exemple le tableau pour l'instruction ADD_IMM_U5 :

| Instruction | Op1 | | Op2 | Op3 |
|---|---|---|---|---|
| sol_1 | #u5 | d15 | | d15 |
| sol_2 | #u5 | d14 | | d14 |
| sol_3 | #u5 | d13 | | d13 |
| … | … | … | | … |
| sol_16 | #u5 | d0 | | d0 |

Les informations à afficher peuvent aussi être paramétrées, par exemple pour ne visualiser que les instructions ayant moins d'un certain nombre de solutions.

### 4.4. Vérification de code

Traditionnellement la correction du code produit par un compilateur peut prendre différentes formes. La première solution consiste à établir la correction du compilateur en lui-même. Cette approche, bien que déjà mise en oeuvre pour différents langages [11, 12, 13], n'est pas vraiment applicable dans notre cas. En effet la mise en œuvre d'une telle approche n'est actuellement pas réaliste lorsqu'on s'intéresse à des compilateurs optimisants pour des architectures sophistiquées. De plus établir la correction d'un compilateur reciblable est une tâche plus complexe puisqu'il faut établir la correction d'algorithmes génériques, paramétrés par des règles, comme ceci a été fait dans [14].

Pour ces différentes raisons A. Pnueli a proposé une approche qui consiste à vérifier individuellement chaque code produit [15]. Une dernière approche est le compilateur qui produit la preuve de correction du code [16]. Ici, en raison du contexte et de la simplicité de la vérification statique à effectuer, nous avons choisi de développer un outil permettant de vérifier chaque programme assembleur eu égard aux contraintes de placement.

Un programme assembleur généré par la chaîne de compilation est une suite d'instances d'instruction machine. Vis-à-vis des contraintes, la vérification est compositionnelle. Il suffit de vérifier que pour chaque instance d'instruction les valeurs des opérandes appartiennent aux domaines de valeurs des opérandes et vérifient les contraintes associées à l'instruction.

Soit OPER $v_1$, …, $v_n$ une instanciation d'une instruction définie pour les opérandes ($op_i : D_i$). Pour chaque opérande $op_i$ on vérifie la condition $v_i : D_i$. Si une de ces conditions est fausse la vérification échoue. Sinon on construit le problème $P = (C, <op_1 : \{v_1\}, \ldots, op_n : \{v_n\}>)$ où C est l'ensemble des contraintes associées à l'opérande, comme défini en section 4.3. Si l'ensemble des solutions est vide alors l'une des contraintes n'est pas respectée sinon l'instruction est correcte vis-à-vis des contraintes de placement.

## 5 – Résultats

Pour Freescale, le principal objectif de cette approche est la facilité à recibler la chaîne de compilation pour une nouvelle architecture, ainsi qu'une amélioration de la qualité du code produit. Actuellement, les expérimentations ont principalement porté sur le langage de description d'architectures et la validation des modèles.

Dans la technologie existante de Freescale, le principal problème, lors de l'implémentation d'une nouvelle architecture était l'écriture de la description de l'architecture cible. En effet, chaque composant du compilateur possédait sa propre description et certains éléments étaient directement codés dans les outils. L'approche reciblable permet de réduire considérablement la taille de la description car il n'y a qu'un seul modèle à écrire au lieu d'un par outil. De plus, ceci réduit le nombre d'erreurs potentiellement introduites pendant la phase de modélisation. Le tableau suivant résume la taille de la description de l'architecture StarCore 140 (Motorola) et d'une nouvelle plateforme DSP en terme de ligne de code :

| **Architecture** | **Description avec l'ancienne technologie (nombre de lignes)** | **Description avec ARCHL (nombre de lignes)** |
|---|---|---|
| StarCore 140 | 53.000 + Code interne | 9.000 |
| Nouvelle Architecture | 64.000 + Code interne | 8.000 |

La description de l'architecture StarCore 140 pour l'ancienne technologie comptait environ 53.000 lignes en plus du code inclus dans les outils. Il faut noter que ce chiffre correspond à la somme des descriptions réparties dans les différents composants du compilateur. La description du même modèle par le langage ARCHL ne prend approximativement que 9.000 lignes.

Un autre avantage est l'économie de temps passé à la description d'une nouvelle architecture. Dans la méthode d'origine, les descriptions étaient fortement dépendantes des composants et incombaient donc à des ingénieurs spécialistes du composant. Avec ARCHL, un seul ingénieur, non forcément expert des composants du compilateur, est nécessaire pour la réalisation de la description. Ainsi, on a pu comparer les temps de développement :

| **Architecture** | **Description avec l'ancienne technologie (nombre de jours)** | **Description avec ARCHL (nombre de jours)** |
|---|---|---|
| StarCore 140 | 20 + 10 + 8 + Code interne | 15 |
| Nouvelle Architecture | 15 + 8 + 7 + Code interne | 13 |

Le langage ARCHL est base sur la description du manuel de référence, avec la possibilité d'étendre la syntaxe très facilement aux besoins des outils.

Finalement le fait d'avoir formalisé les contraintes a permis de valider le modèle à l'aide de l'outil VERIF-MODEL. Dans tous les cas la validation nécessite de comparer les solutions obtenues vis-à-vis du manuel de référence. Par contre l'utilisation d'un formalisme bien défini, le typage des opérandes et des contraintes ont permis de détecter quelques erreurs. Les tests ne sont pas complètement significatifs en terme d'erreurs détectées car il a été écrit par des personnes connaissant bien à la fois le langage ARCHL et l'architecture StarCore 140.

**6 – Conclusion**

Dans ce papier, nous avons présenté une méthode permettant de recibler des compilateurs pour DSP. Nous l'avons expérimenté sur le compilateur CodeWarrior de Freescale, pour la famille des processeurs StarCore. Notre approche a la particularité de bien prendre en compte les aspects irréguliers des instructions DSP par l'introduction des contraintes dans le langage. L'outil CGMG-CSP calcule alors, étant donné un ensemble de contraintes, l'ensemble de solutions, ce qui permet ainsi de garantir la correction du modèle. Cet outil est aussi utilisé pour vérifier la correction du code produit vis-à-vis des contraintes du modèle. Les avantages de notre approche sont, au niveau de la chaîne de compilation, la diminution de la taille de la description de l'architecture, la diminution du temps de développement et de mise au point, et au niveau du code produit, une assurance de qualité.

La prochaine étape consiste à développer un outil permettant de vérifier les règles de correspondance. En effet elles décrivent des schémas de traduction pour lesquels il faut aussi pouvoir calculer l'ensemble des contraintes associées. Un enjeu important est aussi de s'intéresser, par un jeu de règles de correspondance, à la complétude et l'atteignabilité. La complétude doit permettre de garantir que toute instruction correcte du langage source peut être traduite et, à l'inverse, l'atteignabilité garantit que les règles de correspondance permettent d'atteindre chaque instruction du langage cible. L'évaluation de la complétude est relative à la correction du processus de traduction et l'évaluation de l'atteignabilité est relative à l'efficacité de la traduction.

**7 – Bibliographie**

[1] StarCore 140 DSP Core Reference Manual, Reference ; Manual, Rev. 3, November 2001.

[2] R. Leupers and P. Marwedel. Retargetable code generator, based on structural processor descriptions. Design Automation of Embedded Systems, 3, no.1, 1998.

[3] R. Leupers and P. Marwedel : Retargetable Compiler Technology for Embedded Systems - Tools and Applications ; Kluwer Academic Publishers, 2001.

[4] R. Leupers : Retargetable Code Generation for Digital Signal Processors ; Kluwer Academic Publishers, 1997.

[5] M. Freericks : The nML machine description formalism Technical report tr. SM -IMP/DIST/08, 1993.

[6] D. B radlee, R. Henry, and S. Eggers : The marion system for retargetable instruction scheduling ; Conference on Programming language Design and Implementation. ACM SIGPLAN, 1991.

[7] P. Grun and A. N. et al. Expression: A language for architecture exploration through compiler/simulator retargetability. Design, Automation and Test in Europe Conference (DATE 1999), March 1999.

[8] G. Hadjiyiannis, S. Hanono, and S. Devadas. Isdl: An instruction set description language for retargtability. In Proceedings of Design Automation Conference (DAC), Anaheim, CA, May 1997. Design Automation Conference.

[9] V. Bertin, J.-M. Daveau, P. Guillaume, T. Lepely, and all. Flexcc2: An optimizing retargetable c compiler for dsp processors ; EM SOFT'02. Lecture Notes in Computer Science, 1820, 2002.

[10] C. Bessiere, P. Meseguer, E. C. Freuder, and J. Larrosa: On Forward Checking for Non-binary Constraint Satisfaction ;Principles and Practice of Constraint Programming, 1999.

[11] W. Polack : Compiler Specification and Verification ; Springer-Verlag, 1981, LNCS 124

[12] J. D. Guttman and T. D. Ramsdell and M. Wand : VLISP: A Verified Implementation of Scheme ; Lisp and Symbolic Computation volume 8, 1-2, 1995

[13] M. Strecker : Formal Verification of a Java Compiler in Isabelle ; 8th Conference on Automated Deduction, CADE-18, LNCS 2392 A. Voronkov, Springer-Verlag, 2002

[14] F. Badeau, D. Bert, S. Boulmé, C. Métayer, M-L. Potet, N. Stouls et L. Voisin : Adaptabilité et validation de la traduction de B vers C - Points de vue du projet BOM ; Technique et Science Informatiques, RSTI, série TSI, numéro 7/2004, H ermès-Lavoisier.

[15] A . Pnueli, M. Siegel et O. Shtrichman : Translation Validation for Synchronous Languagues ; ICALP 1998, LNCS 1443, Springer-Verlag, 19998

[16] George C. Necula, Peter Lee : Proof-Carrying Code ; Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)