SympA'7
Paris, 24 - 27 Avril 2001
7ème Symposium sur les Architectures Nouvelles de Machines

Validation d'architectures dans un environnement orienté objet.

Frédéric Mallet et Fernand Boéri.

Laboratoire Informatique, Signaux et Systèmes (I3S) - UMR 6070 CNRS-UNSA.

Résumé

Cet article présente une approche nouvelle pour la validation de modèles d'architectures à base de processeurs dans un environnement de modélisation orientée objet. Notre méthode incrémentale baptisée Sep permet, à partir de modèles de haut niveau, d'évaluer les performances, en simulation, d'architectures nouvelles sur des applications critiques de traitement numérique du signal. Cette évaluation permet la construction interactive d'architectures efficaces, les modèles réalisés très tôt dans le cycle de développement servent, après raffinement, de référence de conception pour la construction de modèles synthétisables. Nous présentons deux des mécanismes mis en place grâce aux paradigmes objets, pour la création de modèles riches et réutilisables: un mécanisme de liaison dynamique et la notion de service de modules. Nous montrons que ces mécanismes permettent la manipulation aisée d'expressions algébriques sous forme de graphes de décision binaire; ce qui permet de valider les services de modules en s'assurant qu'ils réalisent effectivement la fonction pour laquelle ils ont été définis.

1 Le problème.

Dans le cadre d'une collaboration avec VLSI Technology, filiale de Philips semiconductors, nous nous sommes intéressés à la définition d'un environnement de simulation qui permet l'évaluation de performances d'architectures de processeurs en traitement numérique du signal, très tôt dans le cycle de conception, avant la synthèse.

En effet, l'évolution incessante des performances demandées, toujours plus importantes réduit le cycle de vie des architectures matérielles spécialisées. Il est donc indispensable de disposer d'un environnement de conception et d'évaluation de performances qui s'adapte aux nouvelles familles de composants.

En particulier, pour les architectures programmables, il s'agit de disposer très rapidement, d'un outil d'analyse de l'architecture à concevoir et d'un simulateur qui prennent en compte la micro-architecture et le jeu d'instructions .

La méthode proposée (Sep- Simulation et Evaluation de Performances) permet, grâce à son modèle générique d'architectures, de construire de façon incrémentale des modèles dont le niveau d'abstraction est adapté aux aspects que l'on souhaite évaluer [9]. L'environnement graphique permet la construction du jeu d'instructions et de l'architecture, ainsi que la simulation de l'architecture avec une application typique de traitement du signal (GSM, TDES) pour laquelle nous souhaitons évaluer des performances par rapport à une micro-architecture. Une fois les performances souhaitées obtenues à partir du modèle de haut niveau - en termes de nombres de cycles ou de taux d'utilisation de composants -, celui-ci peut être enrichi de façon incrémentale puis servira de référence pour la définition d'un modèle synthétisable en VHDL ou en Verilog.

Des approches concurrentes ([11], [3], [2], [13]), voulant utiliser les avantages des approches

orientées objet, enrichissent des langages de description du matériel avec des mécanismes propres à la conception orientée objet. Ces approches basées sur des langages destinés à la synthèse des systèmes numériques, sont de l'avis de plusieurs auteurs ([4], [9], [12]) difficiles d'utilisation car trop contraignants dans les premières phases de la modélisation.

Notre approche basée sur un langage orienté objet intègre les concepts nécessaires à la description d'architectures matérielles, elle se rapproche des méthodes comme Jester [1], JavaX [4]. Sep se distingue cependant de ces langages de programmation car il fournit la plupart des caractéristiques propres aux langages de description d'architectures (ADL) selon les critères de comparaison donnés dans [10]. Il s'agit d'un ADL pour la conception d'architectures de processeurs ou spécialisées.

A partir de notre modèle objet générique des architectures matérielles visées nous avons construit un environnement de modélisation et de simulation qui ne dépend pas des architectures à concevoir. Nous avons alors souhaité introduire des techniques de validation afin de réduire l'ensemble des tests nécessaires à la qualification des modèles réalisés.

Cet article traite de la validation de modèles d'architectures dans un environnement de simulation et met en évidence les qualités de notre modèle qui ont permis la mise en place d'un tel mécanisme. Dans le chapitre 2, nous montrons comment un mécanisme de liaison dynamique permet la simplification de modèles et l'amélioration de la réutilisation des composants. Ensuite le chapitre 3 rappelle la notion de service de modules qui améliore la lisibilité des modèles. Enfin, le chapitre 4 explique comment l'utilisation de ces mécanismes, propres à la modélisation orientée objet, nous ont permis l'introduction dans un environnement de simulation d'un mécanisme de validation basé sur la comparaison d'expressions algébriques et utilisant des graphes de décision binaires. Ce mécanisme est illustré par l'exemple d'une architecture industrielle traitée avec VLSI Technology.

2 Liaison dynamique pour les opérations arithmétiques et logiques.

Le modèle structurel de Sep permet l'agencement de composants en modules. Chacun de ces composants émet, à travers un connecteur, une donnée vers d'autres composants qui ont la charge de les interpréter selon leur propre spécification. Ces données pourront soit déclencher l'exécution d'un service, soit être utilisées comme paramètres pour la réalisation d'un service. Si elles sont utilisées comme paramètres, le choix de l'opération à invoquer pour réaliser le service peut se faire en fonction du type de ces données. Il s'agit de réaliser des opérateurs polymorphes[1] . Un tel mécanisme est très agréable pour la modélisation des composants qui utilisent des opérations arithmétiques ou logiques comme les UAL, les unités de décalage ou de normalisation, les multiplieurs.

Avec ce mécanisme, notre modèle de l'UAL extrêmement simple permet de réaliser toutes les opérations arithmétiques et logiques définies par le concepteur. C'est ensuite le rôle de l'environnement graphique d'interdire dans certains cas l'utilisation de certaines opérations dont on ne désire pas doter l'architecture modélisée. Ainsi, le modèle de l'UAL réalisé est unique pour toutes les architectures modélisées, il est indépendant du type des données, de la taille des bus et de l'arité des opérations réalisées. Il constitue donc un modèle très appréciable et est réutilisé systématiquement.

Ce chapitre montre brièvement les limitations d'une modélisation naïve en utilisant l'héritage et le mécanisme de liaison dynamique naïf des langages orientés objets populaires tels que Java ou C++. Puis nous présentons un mécanisme plus évolué qui permet d'atteindre nos objectifs. Une étude plus détaillée sur la modélisation d'opérateurs binaires polymorphes est disponible dans [8]. Le chapitre 4 montre comment ce mécanisme permet la validation de modèles dans Sep.

Prenons un exemple simple pour illustrer le problème de la représentation d'opérateurs poly-

[1] Les opérateurs polymorphes sont des opérateurs qui sous la même appellation (ex. addition) peuvent représenter des opérations différentes. Le choix de l'opération à exécuter se fait en fonction du type des opérandes.

morphes. Nous désirons définir les opérations de négation et d'addition sur les entiers relatifs, représentés par la classe *IntValue* et sur les nombres réels, représentés par la classe *DoubleValue*. La modélisation naïve de ces opérateurs conduit à la construction des deux classes présentées par la Figure 1. Chacune de ces classes définit les opérations sur les données du type qu'elle représente. Avec cette construction, le mécanisme de liaison dynamique naïf ne permet pas d'obtenir le résultat escompté en présence de polymorphisme[2] . Il y a pour cela deux raisons. La première est que

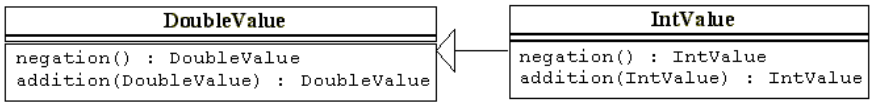

Figure 1: modélisation naïve d'opérateurs polymorphes en présence d'héritage.

le mécanisme de redéfinition[3] de Java n'autorise pas la modification du type de retour. Ainsi, la méthode *negation* de la classe *IntValue* doit nécessairement déclarer un type de retour *DoubleValue*, c'est-à-dire identique à celui de la méthode *negation* de la classe *DoubleValue* qu'elle redéfinit. Le choix de la méthode adaptée se fait grâce au mécanisme de liaison dynamique en fonction du type dynamique des données manipulées et non en fonction du type statique de la référence utilisée. Le deuxième problème vient de la méthode *addition* de la classe *DoubleValue* qui est surchargée[4] dans la sous-classe *IntValue*. En effet, la surcharge est résolue statiquement en Java (et en C++) en fonction du type statique des arguments (paramètres effectifs). Ainsi, il y a une dissymétrie dans le rôle des paramètres. En effet, le paramètre implicite[5] subit la liaison dynamique et le paramètre explicite subit la résolution de la surcharge qui est un mécanisme statique. L'opération 12 (*IntValue*) + 15,5 (*DoubleValue*) ne sera alors pas traitée de la même façon que l'opération 15,5 (*DoubleValue*) + 12 (*IntValue*). Dans le premier cas, on appelle une méthode de la classe *IntValue* ayant un paramètre de type *DoubleValue* (ou un super-type). Dans le deuxième cas, on appelle une méthode de classe *DoubleValue* ayant un paramètre de type *IntValue* (ou un super-type) et éventuellement redéfinie dans une sous-classe de *DoubleValue*.

Dans Sep, on propose un mécanisme général, dynamique et symétrique pour déterminer la méthode à exécuter. Sep doit alors manipuler la notion de type, les relations de sous-typage et les opérateurs. Ceci est rendu possible par la définition d'un méta-modèle des types de Sep. De plus, ce méta-modèle permet la définition de nouveaux types et de nouvelles opérations par extension de l'arbre de sous-typage (cf. Figure 2). La contrainte que nous nous sommes fixés est de ne pas avoir à modifier la classe représentant un type lors de la définition d'un sous-type. Dans [8], nous montrons que ces modifications sont inévitables avec le mécanisme de liaison dynamique de Java.

Lors de la définition d'un nouveau type, le concepteur doit définir une classe qui hérite de *Value* et d'une de ses sous-classes. Il définit les opérations comme des méthodes statiques, ce qui rétablit la symétrie entre les arguments et inhibe le mécanisme de liaison dynamique de Java qui

[2]Une variable polymorphe peut référencer des objets de classes différentes mais avec une interface commune. Ici, une référence de type statique *DoubleValue* peut référencer un objet de type *DoubleValue* ou *IntValue*.
[3]Il y a redéfinition si et seulement si la signature de la méthode est conservée. La signature d'une méthode est constituée de son nom et de la liste ordonnée des types de ses paramètres.
[4]on dit qu'une méthode $\mu 2$ surcharge $\mu 1$ si et seulement si $\mu 1$ et $\mu 2$ sont définies dans la même relation d'héritage, ont le même nom, des signatures différentes et éventuellement des types de retour différents.
[5]Lors de l'invocation d'une méthode d'instance, l'objet sur lequel cette méthode est appliquée, est passé implicitement comme argument et peut être référencé par le mot clé *this*. A l'exécution, tout se passe alors comme si chaque méthode avait déclaré un paramètre de nom *this* et de type identique à la classe définissant la méthode. Ce paramètre est appelé implicite.

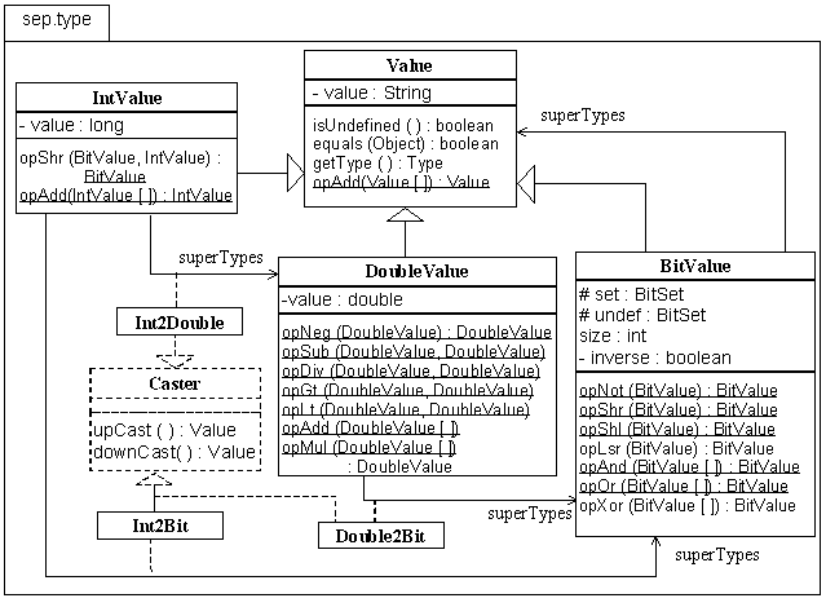

Figure 2: Arbre de sous-typage (Value, DoubleValue, BitValue).

ne fonctionne pas sur les méthodes statiques. Il faut de plus déclarer les super-types du type que l'on définit et les opérateurs conversions explicites. Comme le montre la Figure 2 cette relation de sous-typage n'est pas nécessairement réalisée par de l'héritage.

Lors de la première utilisation de ce type, l'environnement Sep met à jour sa base de données sur les types courants et construits l'arbre de sous-typage.

On peut noter que, bien que la relation de sous-typage soit transitive, et que *DoubleValue* soit définie comme un sous-type de *BitValue*, il est quand même nécessaire de construire une relation de sous-typage entre le type *IntValue* et le type *BitValue*. En effet, la procédure de conversion (upCast) d'un sous-type en son super-type n'est pas transitive pour certains codages. En particulier, il n'est pas équivalent de transformer un entier (*IntValue*) directement en sa représentation binaire (*BitValue*), que de le transformer d'abord en nombre réel (*DoubleValue*), puis en sa représentation binaire.

La méthode utilisée pour la mise en uvre de ce mécanisme est détaillée dans [8]. Le code de l'UAL devient alors:

```
import sep.type.Value;
public class Alu implements sep.model.ServiceProvider {
        public Value execute(Value[] values, String cop) {
                return sep.type.Type.perform(cop, values);
        }
}
```

La méthode *sep.type.Type.perform* parcourt l'arbre de sous-typage à la recherche de la méthode applicable la plus spécifique en fonction du type dynamique des données dans *values*.

C'est ainsi qu'en présence d'une requête d'exécution du service 12+15.5 ou 15.5+12, la méthode *DoubleValue opAdd(DoubleValue [ ])* est sélectionnée, 12 est représenté par un objet de classe *IntValue*, 15.5 est représenté par un objet de la classe *DoubleValue*. Une opération faisant intervenir un *IntValue* et un *DoubleValue* doit être définie dans la classe *IntValue*. On cherche donc dans cette

classe, un méthode applicable. Aucune ne l'est, il faut donc chercher dans la classe représentant le plus-petit super-type de *IntValue*. On trouve alors dans la classe *DoubleValue* la méthode *DoubleValue opAdd(DoubleValue [])* qui est applicable. L'entier (12) est alors converti en nombre réel (12.0) puis l'opération d'addition est invoquée.

Par contre, dans le cas de l'addition de deux entiers, la classe effective (type) des données considérées (*IntValue* dans les 2 cas) nous amène à choisir dans la classe *IntValue* la méthode *IntValue opAdd(IntValue [])*. Java aurait considéré les types statiques des références et non les types dynamiques avec les difficultés signalées précédemment.

3 Modèles hiérarchiques et services de modules.

Notre modèle hiérarchique permet la définition de blocs appelés modules par agencement de composants élémentaires. Des services qui peuvent être des méthodes Java ou des modules Esterel [8] sont composés afin de réaliser le comportement de ces composants élémentaires.

Notre modèle permet, de plus, la définition de services de haut niveau dans les modules ainsi constitués. Ces services de haut niveau sont des compositions séquentielle ou concurrente des services des composants dont le module est constitué. Ces services de haut niveau améliorent la lisibilité des modèles et offrent des mécanismes de réutilisation par héritage de comportement [8].

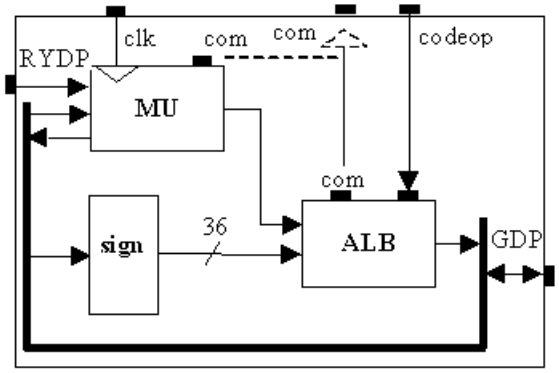

Figure 3: L'unité de calcul: CU.

Prenons comme exemple la modélisation de l'unité de calcul d'un cœur de processeur de traitement du signal. La Figure 3 présente cette unité de calcul constituée du module de multiplication (MU) et du module arithmétique et logique (ALB). Le composant combinatoire *sign* est un composant d'extension de signe. Il transforme les données signées en un équivalent sur 36 bits. Les chemins de données sont représentés par des lignes continues. Les chemins de contrôle sont représentés par des lignes pointillées en utilisant le symbole d'héritage d'UML qui signifie ici que le contrôle est hérité des sous-modules MU et ALB.

Il s'agit maintenant d'étudier d'une façon un peu plus précise les modules MU et ALB. La Figure 4 présente une description détaillée des chemins de données du module de multiplication. Cette description constitue un schéma-bloc de haut niveau dont on suppose qu'il réalise une certaine fonction à base de multiplication. Il s'agit essentiellement d'effectuer la multiplication des valeurs contenues dans les deux registres X et Y. Le résultat sera affecté au registre P synchrone avec l'horloge du processeur.

Le premier objectif des services de modules est de ne pas surcharger le modèle par la description

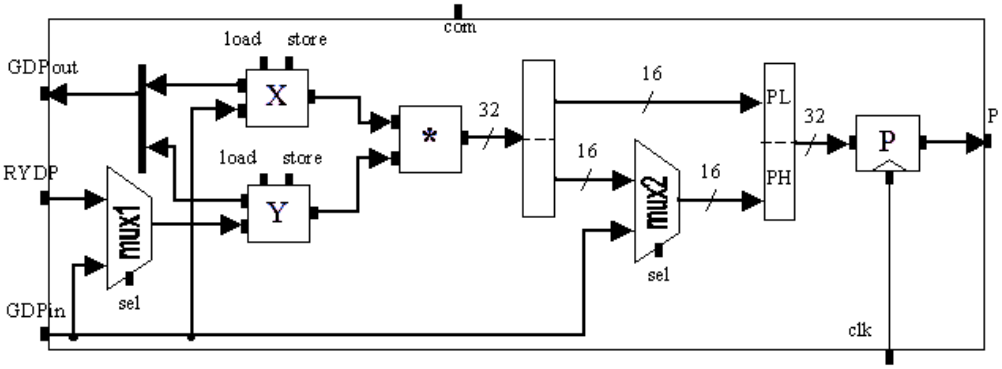

Figure 4: Le module de multiplication: MU.

des chemins de contrôle. Le deuxième objectif est d'enrichir la description, ainsi le module MU présente les services qu'il réalise de façon analogue à un composant élémentaire, alors qu'une description standard présente uniquement l'interface de communication comme une liste de ports typés. De plus, ces deux objectifs améliorent la lisibilité et la réutilisation.

On définit alors quatre services élémentaires. Les services *loadX*, *loadY*, *storeX* sont hérités des services *load* et *store* du registre X, et du service *load* du registre Y. Ils permettent l'utilisation des registres X et Y comme des registres de base, le module de multiplication calcule en permanence le produit des valeurs que ces registres contiennent. Le service *storeY* est un peu plus complexe que les autres. En effet, il faut positionner le multiplexeur *mux1,* afin qu'il sélectionne son entrée reliée au bus GDP par l'intermédiaire du port GDPin, avant d'exécuter le service *store* du registre Y, c'est une composition séquentielle de services.

De plus, on définit le service *p:= X * Y* comme *mux2.sel* <= 0; (storeY || storeX). Ce service positionne les multiplexeurs *mux1* et *mux2* pour charge les registres X et Y.

Enfin, le concepteur désire ajouter le service *ph:= GDP* qui permet d'affecter la partie haute du registre P avec la valeur présente sur le bus GDP - il suffit pour cela de sélectionner correctement le multiplexeur *mux2.*

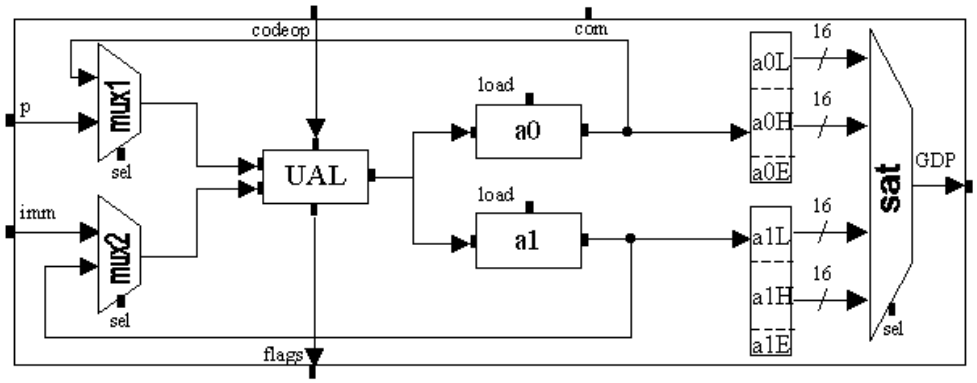

Figure 5: Le module arithmétique et logique: ALB.

La Figure 5 présente les chemins de données du module arithmétique et logique. Ce module contient l'unité arithmétique et logique *UAL* présentée au chapitre précédent et capable d'effectuer suivant son code opération (*codeop*) des calculs arithmétiques ou logiques. Les résultats calculés peuvent être mémorisés (service *load*) dans les deux accumulateurs a0 et a1. Les opérandes sont choisies grâce à deux multiplexeurs *mux1* et *mux2*. Les deux accumulateurs peuvent manipuler des données de taille quelconque, dans le cas qui nous intéresse ils ne manipuleront que des données sur 36 bits. Leurs valeurs peuvent être décomposées en données de 16 bits (a0H, a0L, a1H, a1L) et émises sur le bus GDP grâce à l'unité *sat* qui possède un service séquentiel *sel*.

Ce module propose les services *write_a0* et *write_a1* qui permettent la mémorisation dans les accumulateurs, les services *GDP:= a0H*, *GDP:= a0L*, *GDP:= a1H*, *GDP:= a1L* qui permettent par une sélection de l'unité *sat* d'écrire sur le bus GDP la valeur des registres 16 bits a0H, a0L, a1H, a1L, et les services *read_a0*, *read_a1*, *read_p*, *read_imm* qui permettent de sélectionner les multiplexeurs *mux1* et *mux2* afin de choisir les opérandes de l'*UAL*.

Pour terminer, ces deux modules (ALB et MU) sont composés pour réaliser le module CU (cf. Figure 3) dans lequel on peut désormais définir les services *mulacc_a0* et *mulacc_a1*. Ces services effectuent en parallèle une multiplication sur les données présentes sur les bus GDP et RYDP avec le module de multiplication et une opération entre P et respectivement a0 ou a1, le résultat est accumulé dans l'accumulateur de départ, c'est-à-dire a0 ou a1. L'opération effectuée dépend du code opération présent sur l'ALU au moment de l'opération.

On a ainsi réalisé ce service de multiplication-accumulation assez complexe, il s'agit maintenant de vérifier qu'il réalise la fonction attendue. Pour cela nous allons utiliser le mécanisme de liaison dynamique présenté au chapitre 2.

4 La validation de modèles et les arbres de décision binaire.

4.1 Le principe.

Notre objectif est de valider fonctionnellement les services de modules que l'on créé. Le chapitre 3 montre que la construction de ces services peut être assez complexe. Il s'agit de vérifier que la fonction effectivement réalisée correspond à la fonction souhaitée. Il s'agit pour cela de comparer l'expression algébrique réalisée par un service de module, à l'expression algébrique de la fonction que l'on souhaitait lui faire réaliser.

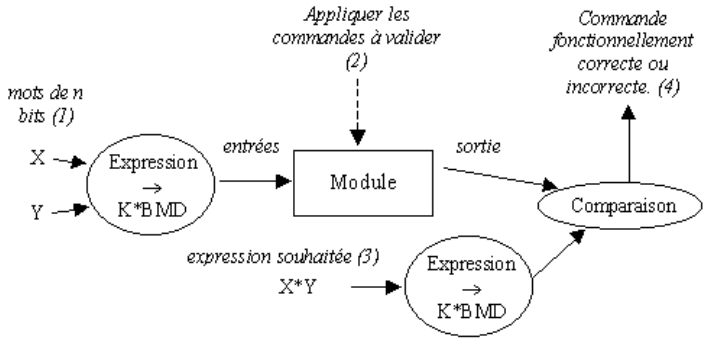

Figure 6: Principe du mécanisme de validation fonctionnelle dans Sep.

La méthode est très simple, mais il est assez délicat de manipuler des expressions algébriques avec

un outil informatisé. En effet, la forme canonique des expressions algébriques est, en général, assez délicate à obtenir pour les expressions complexes, la comparaison de deux expressions algébriques peut alors être très coûteuse. C'est pourquoi, nous souhaitons utiliser une forme de graphe de décisions binaires (BDD[6]) adaptée à la représentation des expressions que nous manipulons couramment. En effet, les BDD ont une forme canonique suffisamment aisée à obtenir pour être manipulés efficacement par un système informatisé. Une fois la forme canonique obtenue, l'opération de comparaison a un coût de calcul linéaire par rapport au nombre de nœuds des graphes.

La forme que nous avons choisi d'utiliser s'appelle K*BMD[7] introduit dans [6]. Nous l'avons retenue car elle est très efficace pour la représentation de mots de bits, c'est-à-dire pour la représentation de fonctions de $\mathcal{B}^n \rightarrow \mathcal{Z}$, des fonctions qui à n bits associent un entier relatif. Alors que les formes élémentaires de BDD, pour représenter un mot de n bits, nécessitent n graphes indépendants, ce qui rend très coûteux la réalisation d'opérations sur les mots aussi simples que l'addition ou la multiplication.

La Figure 6 illustre ce mécanisme de validation. Pour chaque module et chaque commande à valider, il faudra réaliser les quatre étapes suivantes:

- construire les K*BMD à partir de mots de n bits, ces K*BMD constitueront les entrées du module;
- appliquer la ou les commandes à valider, cela peut être un appel de service de module, une combinaison d'appels de services élémentaires: le résultat obtenu sur une sortie du module est alors un K*BMD qui correspond à l'expression algébrique de la fonction effectivement réalisée;
- formuler l'expression de la fonction que l'on souhaite réaliser en appliquant ces commandes et la convertir en K*BMD;
- comparer l'expression souhaitée à celle obtenue pour savoir si les commandes effectuées par rapport à une architecture donnée permettent d'obtenir la fonction souhaitée.

4.2 Les K*BMD et Sep.

Les BDD sont des graphes orientés sans circuit. Initialement basés sur la décomposition de Shannon des fonctions booléennes f, certaines formes dont les K*BMD intègrent désormais les décompositions positive et négative de Davio. En particulier, l'usage mixte de ces trois formes permet d'obtenir une représentation compacte en utilisant à chaque fois la décomposition la mieux adaptée; il s'agit de la famille des Binary Moment Diagram (BMD). Les nœuds sont alors marqués { S, pD, nD } suivant la décomposition utilisée.

Une autre extension des BDD dont profitent les K*BMD est la valuation des arcs. Les graphes EVBDD[8] mono-valués par une valeur a $\in$ Q: la fonction représentée est a + **f**, où **f** est l'expression représentée par le sous-arbre destination de l'arc valué par a. Les graphes *BMD[9] mono-valués par une valeur m $\in$ Q: la fonction représentée est m $\times$ **f**, où **f** est l'expression représentée par le sous-arbre destination de l'arc valuée par m.

Les K*BMD sont valués par une paire (a,m) $\in Q^2$, la fonction représentée a comme expression a + m $\times$ **f**.

Un intérêt majeur des graphes dont les arcs sont bi-valués est qu'il facile de donner un équivalent en K*BMD pour un BDD. En effet, il suffit d'utiliser toujours une décomposition de Shannon, de valuer avec (0,1) - qui signifie 0+1$\times$... - tous les arcs qui se terminent sur des sommets qui ne sont pas des feuilles, de valuer avec (0,1) tous les arcs qui se terminent sur la feuille 0 et de modifier les arcs se terminant sur la feuille 1 pour qu'ils se terminent sur la feuille 0 avec la valeur (1,0) - qui

[6]BDD: Binary Decision Diagram. Les graphes de décisions binaires ont été introduits en 1978 par Akers pour représenter de façon compacte les fonctions booléennes, c'est-à-dire les fonctions de $\mathcal{B}^n \rightarrow \mathcal{B}$.
[7]K*BMD: Kronecker Multiplicative Moment Diagram.
[8]Edge-Valued Binary Decision Diagram.
[9]Multiplicative Binary Moment Diagram.

signifie 1+ 0×0 -. Cette correspondance n'est pas du tout évidente pour les graphes dont les arcs sont mono-valués.

Une description plus complète sur les BDD est fournie par plusieurs auteurs dont [5]. Les K*BMD sont présentés dans [7].

Le chapitre 2 montre le mécanisme de liaison dynamique mis en place afin de choisir les opérateurs arithmétiques adaptés en fonction du type des données manipulées. Ce mécanisme est utilisé ici pour manipuler les K*BMD.

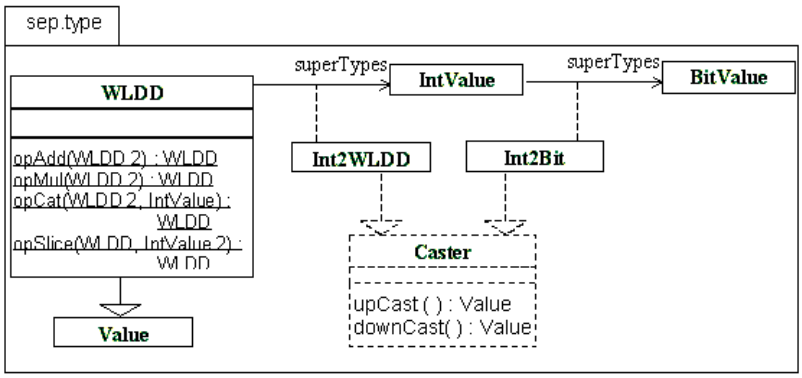

Figure 7: Définition du type K_BMD.

Le type *K_BMD* est ajouté dans l'arborescence des types de Sep (cf. Figure 7). La classe *K_BMD* hérite de la classe *Value*, ainsi les composants Sep pourront émettre et recevoir des K*BMD par l'intermédiaire des bus et des signaux de commande. De plus, le type *K_BMD* est défini comme un sous-type de *IntValue*. Tout ce qui peut être fait avec des entiers peut l'être avec des *K_BMD*, en effet les K*BMD sont des fonctions de $\mathcal{B}^n \rightarrow \mathcal{Z}$. Les opérations de base sur les entiers ou les *BitValue* sont redéfinies afin d'implémenter les algorithmes de manipulation des K*BMD. Un K*BMD qui représente un entier n est en fait un graphe réduit à une feuille.

Dès lors, les opérateurs adaptés à la structure de K*BMD seront utilisés automatiquement sans aucune modification de l'architecture. Lorsque les entrées d'un module sont des entiers, les opérateurs sur les entiers sont utilisés; lorsque les entrées du même module sont des K*BMD, les opérateurs sur les K*BMD sont utilisés. Le type des sorties dépend du type déclaré par les opérateurs.

Les opérations caractéristiques nécessaires pour la modélisation d'architectures matérielles doivent être adaptées à la structure de K*BMD. La section 4.3 présente quelques unes de ces opérations et leur implémentation en K*BMD afin de donner une idée de la complexité d'utilisation de cette structure de données.

4.3 Les opérations élémentaires.

Pour simplifier le propos nous n'utiliserons que des K*BMD avec une décomposition positive de Davio.

L'implémentation de ce mécanisme pour notre exemple nécessite l'implémentation des opérations arithmétiques utilisées dans les modules MU, ALB et CU. En particulier, il faut implémenter les opérations classiques d'une unité arithmétique et logique - nous ne traiterons que l'*addition* et le *et logique* -, l'opération de multiplication, la sélection de n bits consécutifs à partir du m-ième et la concaténation de mots de bits.

Les autres composants - multiplexeurs et registres - n'effectuent aucune modification sur les données, ils se contentent le moment venu de les transmettre ou de ne pas les transmettre.

Les cas les plus simples sont les cas de l'addition et de la soustraction. En effet, si f et g sont des fonctions de $\mathcal{B}^n \to \mathcal{Z}$ alors leurs décompositions positives de Davio sont $f = f_i^0 + x_i \times (f_i^1 - f_i^0)$ et $g = g_i^0 + x_i \times (g_i^1 - g_i^0)$. On a alors $f + g = f_i^0 + x_i \times (f_i^1 - f_i^0) + g_i^0 + x_i \times (g_i^1 - g_i^0)$, c'est-à-dire $(f_i^0 + g_i^0) + x_i \times ((f_i^1 - f_i^0) + (g_i^1 - g_i^0))$.

Si on appelle f_G (respectivement g_G) la fonction représentée par le premier sous arbre de f (respectivement g) et f_D (respectivement g_D) la fonction représentée par le deuxième sous-arbre de f (respectivement g). On a alors $(f + g)_G = f_G + g_G$ et $(f + g)_D = f_D + g_D$.

Pour construire l'arbre de la fonction f+g il suffit donc d'appliquer récursivement l'opérateur addition aux sous-arbres de f et de g. Le K*BMD correspondant à l'addition de deux mots de (n+1) bits est donné par la Figure 8. Un calcul identique peut être fait pour la soustraction.

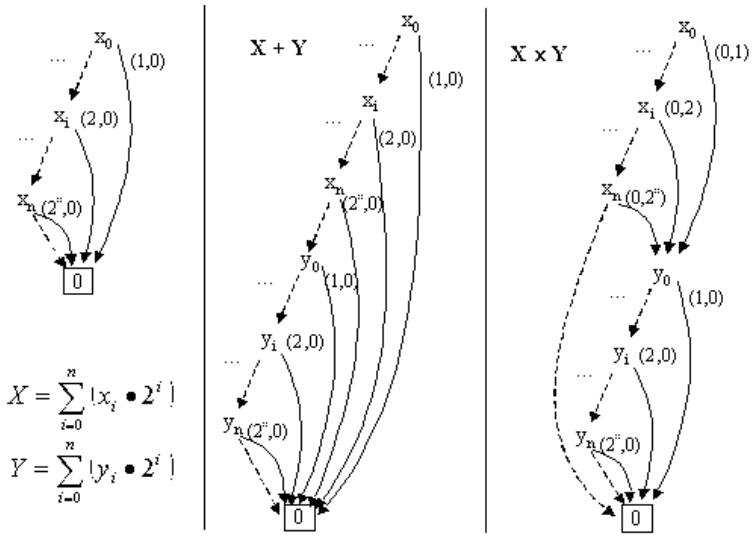

Figure 8: Addition et multiplication de deux mots de n+1 bits.

Calculons maintenant la décomposition positive de Davio de la fonction $f \times g$ par rapport à la variable x_i. On a $f \times g = (f_i^0 + x_i \times (f_i^1 - f_i^0)) \times (g = g_i^0 + x_i \times (g_i^1 - g_i^0))$. Si $x_i = 0$ alors $(f \times g)_i^0 = f_i^0 \times g_i^0$, si $x_i = 1$ alors $(f \times g)_i^1 = f_i^1 \times g_i^1$. De ce fait, $f \times g = f_i^0 \times g_i^0 + x_i \times (f_i^1 \times g_i^1 - f_i^0 \times g_i^0)$.

Ainsi, $(f \times g)_G = f_G \times g_G$ et $(f \times g)_D = (f_D \times g_D) - (f_G \times g_G)$. Le K*BMD correspondant est donné par la Figure 9.

La multiplication d'un K*BMD par une constante C se fait à temps constant, puisqu'il suffit de multiplier les cœfficients m de l'arc entrant de la racine par la constante C. En effet, (a + m × f) × C = (a × C + m × C × f).

La concaténation de deux mots de bits a et b est alors $a + 2^n \times b$ où n est le nombre de bits utilisés pour représenter le mot a. Il s'agit alors d'une multiplication par une constante et d'une addition.

La sélection de n bits à partir du $m^{ième}$ est une opération complexe pour les WLDDs. Cette opération que l'on appellera *slice* est définie pour un mot a de k ≥ n+m bits de la façon suivante: *slice(a, n, m)* = $(a/2^m)\%2^n$.

Les opérations de modulo % et de division sont assez difficiles à réaliser, cependant en utilisant

les propriétés que *(f+g)%h = (f%h + g%h)%h* et *(f × g)%h = (f%h × g%h) % h*, la taille du K*BMD résultant est souvent linéaire surtout si f et g sont indépendantes de h (cf. [7]).

4.4 Un exemple.

Prenons l'exemple de l'unité de calcul présentée au chapitre 3. Il s'agit alors de valider le service **codeop=Add; mulacc_a0; clk** où clk correspond à l'activation du registre p synchrone sur l'horloge du système. Il faut vérifier que cette unité de calcul effectue correctement une opération complète de multiplication-accumulation en deux cycles d'horloge.

Pour cela, le module *unité de calcul* est instancié, la valeur de l'accumulateur a0 est fixée avec un K*BMD qui représente un mot de 36 bits $a0 = \sum_{i=0}^{35} (a_i \times 2^i)$, les deux entrées RYDP et GDP_{in} prennent les valeurs $RYDP = \sum_{i=0}^{35} (rydp_i \times 2^i)$ et $GDP_{in} = \sum_{i=0}^{35} (gdp_i \times 2^i)$. Le service **codeop=Add; mulacc_a0** est appelé sur le module, le registre p est alors activé pour simuler un cycle d'horloge (**clk**). Le K*BMD, obtenu en entrée de l'accumulateur, est comparé au K*BMD, obtenu à partir de l'expression a0 + RYDP×GDP_{in}, qui est une expression de la fonction que l'on souhaite obtenir.

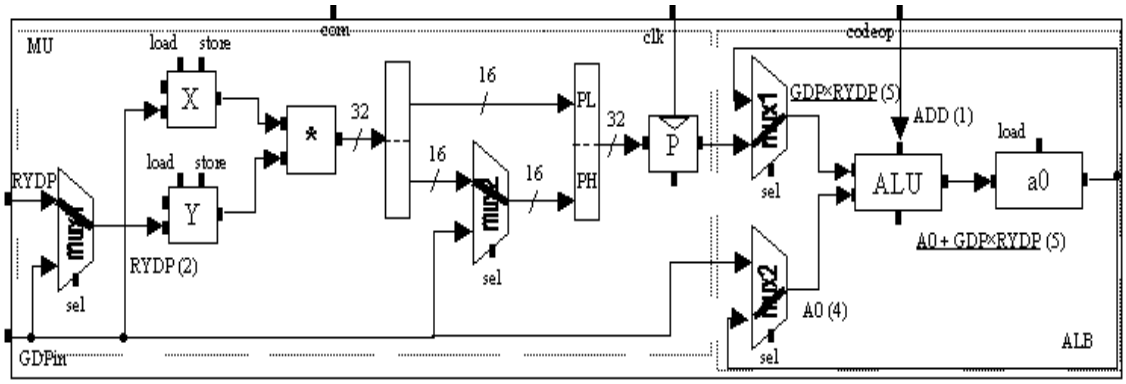

Figure 9: Scénario de validation.

Dans le cas de l'architecture présentée, la comparaison indique que le service est correct. Pour sén convaincre, regardons instant par instant le comportement de chaque composant (cf. Figure 9).

Premier instant: le code opération *Add* est positionné sur l'ALU.

Deuxième instant: MU.mux1.sel<=0 || MU.mux2.sel <= 0 || ALB.mux1.sel<=2 || ALB.mux2.sel<=0. L'expression RYDP est propagée en entrée du registre Y.

Troisième instant: MU.Y.com=store || MU.X.com=store. Les registres X et Y sont chargés, les expressions RYDP et GDP sont multipliées, le résultat est décomposé en sa partie haute *(GDP*RYDP)/16* et sa partie basse *(GDP*RYDP)%16*, puis recomposé par concaténation.

Quatrième instant: *ALB.a0.load.* L'accumulateur a0 est activé, l'expression a0 est propagée en entrée de l'ALU.

Cinquième instant: le registre p est activé, l'expression *GDP×RYDP* est propagée vers l'ALU. Le résultat calculé est alors *A0 + GDP × RYDP.*

Le résultat obtenu manuellement ci-dessus en raisonnant sur des expressions algébriques est produit automatiquement dans une phase de simulation de Sep en utilisant des K*BMD. Grâce au mécanisme de liaison dynamique introduit, l'intégration d'un tel mécanisme dans Sep ne nécessite

aucune modification de Sep, puisqu'il suffit de définir un nouveau type dans l'arbre de sous-typage et d'implémenter un paquetage de gestion des K*BMD.

5 Conclusion.

Nous avons présenté une technique basée sur l'utilisation d'arbres de décision pour la validation de modèles d'architectures matérielles. Il s'agit de construire des modèles de haut niveau, plus puissant, plus lisibles et donc mieux réutilisables que les modèles réalisés avec les langages classiques de description du matériel tels que VHDL ou Verilog. Les mécanismes de haut niveau utilisés, basés sur des techniques propres aux langages orientés objet permettent la prise en compte de types complexes tels que des graphes de décision binaire. Ces graphes permettent dans ce cas précis de valider les fonctions réalisées par les services de haut niveau.

Cette technique générale peut être réutilisée dans n'importe quel environnement de simulation où l'on désire introduire des aspects de validation. Le mécanisme de liaison dynamique peut être utilisé pour la prise en compte d'autres types suivant les besoins. Pour notre part, nous allons poursuivre cet effort de modélisation pour la description d'applications de plus haut niveau et en particulier dans le cadre de la modélisation système d'une application multi-processeurs.

Bibliographie

1. *"Jester: a Reactive Java extension proposal by Esterel Hosting."*
   M.Antoniotti & A.Ferrari, http://www.parades.rm.cnr.it/projects/jester/jester.html, 1999.
2. *"A proposed Design Objectives Document for Object-Oriented VHDL."*
   David L.Barton, Jean Michel Berge. The RASSP Digest - Vol. 3, septembre 1996. http://rassp.aticorp.org/newsletter/html/96sep/news_18.html
3. *"Object-Oriented Extensions to CHDL : The LaMI Proposal"*
   J. Benzakki, B. Djaffri. IFIP 1997. Chapman & Hall. p. 334-347.
4. *"Objets pour la modélisation de Systèmes Matériels : intérêts, évolutions et tendance."*
   J.Benzakki, Habilitation à diriger des recherches, Université d'Evry Val d'Essonne, 2000.
5. *"Symbolic boolean manipulation with ordered binary-decision diagrams"*
   R.E.Bryant. ACM Computing surveys, 24(3):293-378, 1992.
6. *"K*BMDs: A new Data Structure for Verification"*
   R. Drechsler & al. IEEE European Design & Test Conference, pp. 2-8, Paris, 1996.
7. *"Formal Verification of Word-Level Specifications"*
   S. Höreth, R.Drechsler. IEEE Design, Automation and Test(DATE'99), Munich, 1999.
8. *"Modélisation et Evaluation de Performances d'architectures matérielles numériques."*
   F. Mallet, thèse de doctorat, Université de Nice-Sophia Antipolis, décembre 2000.
9. *"Hardware Architecture Modelling using an Object-oriented Method."*
   F. Mallet, F. Boéri, J-F. Duboc. Actes de Euromicro conference, sept. 1998, vol I, p.147-153.
10. *" Classification and comparison Framework for Software Architecture Description Languages."*
    N. Medvidovic, R.N. Taylor, IEEE Trans. on Software Engineering, Vol.26, No. 1, janvier 2000.
11. *"Proposed Language Requirements for Object-Oriented Extensions to VHDL."*
    G.D. Peterson, Proc. of Forum on Design Languages, FDL'99, France, sept. 99.
12. *"Inheritance Concept for Signals in Object-Oriented Extensions to VHDL."*
    G. Schumacher, W. Nebel. Proceedings of the EURO-DAC'95 with EURO-CHDL'95.
13. *"Object-Oriented VHDL Provides New Modeling and Reuse Techniques for RASSP. "*
    Dr. Sowmitri Swamy, Vista RASSP Program Manager. The RASSP Digest - Vol. 2, No. 1, 1^{st}. Qtr. 1995. http://rassp.aticorp.org/newsletter/html/95q1/news_6.html