LABORATOIRE

INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR 6070

ARCHITECTURES EMBARQUÉES ET VALIDATION DANS UN ENVIRONNEMENT ORIENTÉ OBJET

Frédéric Mallet, Fernand Boéri

Projet MOSARTS

Rapport de recherche
I3S/RR–2002-01–FR

février 2002

LABORATOIRE I3S: Les Algorithmes / Euclide B – 2000 route des Lucioles – B.P. 121 –
06903 Sophia-Antipolis Cedex, France – Tél. (33) 492 942 701 – Télécopie : (33) 492 942 898
http://www.i3s.unice.fr/I3S/FR/

RÉSUMÉ : Cet article présente une technique originale, orientée objet de modélisation d'architectures matérielles. L'objectif est de réaliser des modèles de plus haut-niveau pour réduire les temps de modélisation et plus puissants que les modèles réalisés avec les langages classiques de description du matériel tels que VHDL ou Verilog. Nous montrons ensuite comment les mécanismes mis en place, spécifiques des approches orientées objet, permettent la validation de modèles d'architectures à base de processeurs dans un environnement de modélisation orientée objet. Nous présentons une synthèse de notre méthode de modélisation appelée SEP. Cette méthode permet l'évaluation de performances en simulation et permet de valider certains aspects des modèles réalisés par une technique qui s'apparente à une simulation symbolique et qui utilise une technique de liaison dynamique pour la prise en compte d'arbre de décision binaires.

MOTS CLÉS : Architectures de processeurs, modélisation orientée objet, validation, évaluation de performances, simulation.

ABSTRACT:

KEY WORDS :

Architectures embarquées et validation dans un environnement orienté objet.

Frédéric Mallet et Fernand Boéri.

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

Mots clés : Architectures de processeurs, modélisation orientée objet, validation, évaluation de performances, simulation.

Résumé

Cet article présente une technique originale, orientée objet de modélisation d'architectures matérielles. L'objectif est de réaliser des modèles de plus haut-niveau pour réduire les temps de modélisation et plus puissants que les modèles réalisés avec les langages classiques de description du matériel tels que VHDL ou Verilog. Nous montrons ensuite comment les mécanismes mis en place, spécifiques des approches orientées objet, permettent la validation de modèles d'architectures à base de processeurs dans un environnement de modélisation orientée objet.

Nous présentons une synthèse de notre méthode de modélisation appelée SEP.

Cette méthode permet l'évaluation de performances en simulation et permet de valider certains aspects des modèles réalisés par une technique qui s'apparente à une simulation symbolique et qui utilise une technique de liaison dynamique pour la prise en compte d'arbre de décision binaires.

1. Introduction.

1.1 Le problème et les domaines d'application.

Dans le cadre d'une collaboration avec VLSI Technology, filiale de Philips semiconductors, nous nous sommes intéressés à la définition d'un environnement de modélisation et de conception d'architectures de processeurs pour le traitement numérique du signal.

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. L'objectif affirmé était alors de fournir une méthode de modélisation et de prototypage rapide d'architectures. La méthode de modélisation doit permettre la réutilisation des modèles et des outils associés. Le prototypage permet l'élaboration de nouvelles architectures à partir d'anciennes afin d'en optimiser les performances par rapport à des applications critiques caractéristiques.

Les architectures matérielles sont souvent conçues en utilisant des langages normalisés de description du matériel tels que VHDL ou Verilog. Or ces langages proposent des mécanismes pauvres de réutilisation et d'adaptation. De nombreuses approches (**[Pet99]**, **[BeD97]**, **[BaB96]**, **[Swa95]**) consistaient à enrichir ces langages en les dotant des concepts bien connus du génie logiciel pour améliorer la réutilisation, c'est-à-dire originaires du monde orienté objet. Malheureusement, de l'avis de plusieurs auteurs (**[Ben00]**, **[MBD98]**, **[ScN95]**), ces langages conçus pour la synthèse sont trop rigides ; Les tentatives d'extension avec les concepts objets (OO-VHDL) ne sont pas suffisamment souples dans les premières phases de la modélisation.

Nous pensons qu'il est plus efficace de partir de langages à objets et de les enrichir avec les concepts nécessaires à la description d'architectures matérielles à haut niveau. Ce point de vue a été utilisé par plusieurs équipes, en particulier, pour la définition de langages comme Jester **[AnF99]** ou JavaX **[Ben00]**.

De plus, le besoin récurrent d'établir des descriptions structurelles des architectures à concevoir nous a conduit vers la définition d'un langage de description d'architectures (ADL) qui est plus qu'un simple langage de programmation. Nous avons **[Mal00]** considéré pour cela les critères énoncés par Medvidovic et Taylor **[MeT00]** comme caractéristiques des ADL.

Il s'agit donc ici de présenter la synthèse des travaux réalisés pour la définition de notre langage, ADL pour la conception d'architectures de processeurs et d'architectures spécialisées. Ce langage appelé SEP (Simulation et Evaluation de Performances) adresse le problème de la modélisation. Il a été exploité pour l'évaluation de performances en simulation ainsi que pour la vérification des architectures.

1.2 Présentation détaillée de SEP.

SEP est basé sur une méthode incrémentale, orientée composants, de modélisation d'architectures matérielles numériques. Les modèles d'architectures sont construits par un agencement structurel de composants et de connecteurs, suivant des règles de composition décrites dans un modèle générique de composition **[MBD98]**.

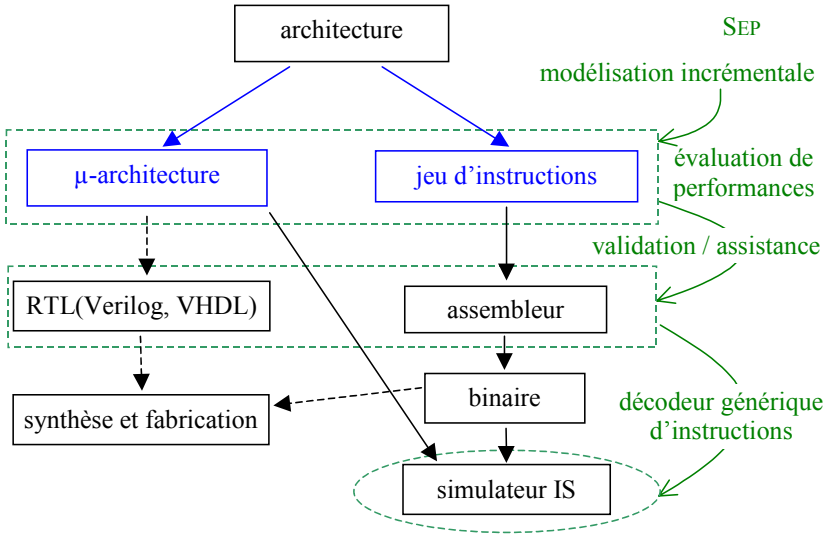


Figure 1 – Flot de conception d'une architecture.

Le comportement des composants élémentaires est décrit avec le formalisme le plus adapté, compatible avec le modèle générique de communication. Pour les architectures étudiées jusqu'à lors, nous avons retenu trois langages.

1. Le langage objet Java qui permet la mise en œuvre aisée des mécanismes orientés objet définis sur les composants SEP.
2. Le langage synchrone Esterel qui permet une spécification précise et rigoureuse des aspects réactifs nécessaires à la description de certains composants, en particulier la concurrence et la préemption. L'encapsulation d'une machine d'exécution Esterel nous permet l'intégration de tels modules dans les descriptions SEP. L'étude détaillée de ce mécanisme est présentée par **[MBD99]**.
3. Enfin, le langage orienté objet SEP-ISDL qui permet la description du jeu d'instructions des architectures à micro-processeurs modélisées. On pourra alors obtenir un simulateur qui prend en compte à la fois la micro-architecture et le jeu d'instructions (cf. Figure 1). Cette intégration est réalisée par la définition d'un composant appelé décodeur générique d'instructions, paramétré avec une description SEP-ISDL. L'héritage et la délégation sont utilisés de manière très efficace pour modéliser les modes d'adressage, beaucoup plus nombreux pour les processeurs de traitement du signal que pour les processeurs standards.

L'originalité dans l'utilisation de Java comme langage de description du matériel vient dans la définition du comportement des composants par composition concurrente ou séquentielle de services élémentaires décrits par des méthodes Java. Concrètement, cela signifie que le comportement des composants est délégué à des classes Java particulières appelées fournisseurs de services, externes au composant. Il est alors possible de :

- Construire et réutiliser des comportements complexes avec les mécanismes orientés-objet standards et en définissant de nouvelles classes Java.
- Assembler ces services élémentaires et définir les contraintes pour construire des composants élémentaires. Les services peuvent être réutilisés pour construire des composants divers dont le comportement diffère par le mode de communication avec l'extérieur ou par les interactions entres les services constituants. Cette composition est réalisée de façon graphique en respectant le méta-modèle des composants SEP.
- Définir la notion d'héritage de comportement entre composants. Les composants peuvent alors hériter soit uniquement des services de leur composant parent, soit à la fois des services et de l'interface de communication (ensemble de ports munis d'une sensibilité).

Cet aspect concernant l'utilisation de la délégation pour améliorer la réutilisation est détaillé dans la section 2.

Le modèle générique de composition et de communication, décrit en UML (Unified Modelling Language), nous permet une modélisation unifiée d'architectures hétérogènes et la construction d'un environnement

graphique d'aide à la conception, non spécifique d'une architecture. Les outils d'analyse définis seront alors réutilisables.

1.3 Evaluation de performances et simulation.

Les modèles réalisés peuvent être analysés. En particulier, nous pouvons évaluer les performances des architectures par rapport à des applications critiques. Pour les architectures embarquées, cette évaluation de performances doit se faire avant synthèse car les coûts de fabrication sont importants. De plus, les techniques de test après synthèse ne permettent pas d'analyser l'influence des différents paramètres sur le comportement. Elles sont souvent intrusives (ex. la norme JTAG) et peuvent modifier le comportement du système étudié **[Has98]**.

D'autre part, les systèmes que nous étudions sont trop complexes pour avoir des modèles analytiques fiables, c'est pourquoi nous avons choisi la simulation comme moyen d'évaluation des performances. Un moteur de simulation a été développé. Il permet une simulation de type événements discrets et intègre la notion de services et de sensibilité.

Les applications sont décrites en assembleur. Le décodeur générique permet très rapidement de prendre en compte les instructions assembleurs, sans définir un langage machine ni les outils nécessaires à sa production.

L'évaluation de performances consiste à évaluer les deux critères énoncés par notre interlocuteur industriel. Premièrement, le nombre de cycles nécessaires pour exécuter une application critique sur l'architecture en construction. Deuxièmement, le taux d'utilisation des composants de l'architecture vis-à-vis d'une application. Il s'agit là d'identifier les éventuels points de congestion ou d'économiser de la surface de silicium et d'améliorer la consommation en éliminant les composants peu utilisés ou en cherchant à mieux les utiliser.

En définitive, cette évaluation permettrait de constituer les bibliothèques d'information nécessaires en amont de nombreuses méthodes de co-conception (codesign) comme Syndex **[Syn97]** ou Codef **[BAP99]**.

Le modèle peut être raffiné ou modifié pour obtenir les performances souhaitées, le simulateur sert alors de référence de conception pour la mise au point d'un code synthétisable VHDL ou Verilog.

La simulation n'est intéressante que si les modèles réalisés sont fiables par rapport au système physique. Afin d'améliorer cette fiabilité nous avons introduit des techniques de vérification qui concernent quatre aspects des modèles réalisés.

1. L'information sur le type des méthodes Java utilisées pour décrire les services est extraite grâce au mécanisme d'introspection de Java pour déduire le type des interfaces des composants. Un mécanisme d'inférence de types permet ensuite de valider la composition des composants, de déterminer les services effectivement utilisables en fonction du contexte et ainsi de détecter et d'interdire la construction de certains modèles ambigus.
2. Un arbre de sous-typage permet de définir les types et opérations utilisables sur ces types de façon indépendante aux modèles d'architectures. Un mécanisme de liaison dynamique permet de choisir les opérations adaptées en fonction du type effectif des paramètres échangés. Nous utilisons ce mécanisme de liaison dynamique pour effectuer une simulation symbolique des services réalisés. Cette simulation symbolique permet la vérification fonctionnelle des services constitués.
3. La composition concurrente des services est validée par construction en vérifiant que les ressources utilisées par les différents services sont correctement partagées.
4. Enfin, l'encapsulation d'une machine d'exécution du langage Esterel dans un composant SEP permet la validation partielle de certains composants orientés contrôle et entraîne une réduction du nombre de jeux de tests nécessaires en simulation.

La suite de cet article traite essentiellement 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. Pour cela, la section 2 rappelle la notion de service et en particulier son utilisation dans le cadre des modules. Ensuite, la section 3 présente les mécanismes de typage et d'inférence pour valider la composition de composants élémentaires. Nous montrons comment la liaison dynamique permet alors la simplification de modèles et l'amélioration de la réutilisation des composants. Enfin, avant de conclure, 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 l'utilisation de graphes de décision binaires. Ce mécanisme est illustré sur l'exemple d'une architecture industrielle traitée avec VLSI Technology.

2. La notion de service dans SEP.

Cette section s'intéresse à la modélisation du comportement des composants élémentaires en Java, elle présente les notion de services et de fournisseurs de services qui permettent d'améliorer la réutilisation. Il s'agit en fait d'utiliser le principe de délégation des langages orientés objet.

2.1 Les fournisseurs de services.

Pour expliquer l'intérêt des fournisseurs de services, nous présentons un extrait de code typique **[MBD98, Sim98]** utilisé pour la description du comportement d'un composant élémentaire. Il apparaît que la spécification du comportement à partir d'un langage de programmation existant mène souvent à considérer en premier lieu des aspects nécessaires mais non fondamentaux. Notre idée est alors de considérer d'abord le comportement primaire du composant. Dans un deuxième temps, on construit le comportement secondaire avec des mécanismes semi-automatiques éventuellement assistés par des procédés graphiques.

Nous avons choisi d'illustrer ce discours sur l'exemple de la construction d'un composant registre. Le premier exemple de code (cf. Exemple 1) montre le code écrit avec les méthodes standards. Le deuxième exemple (cf. Exemple 2) montre le code actuellement utilisé par SEP.

```
package component;

import modele.*;
import implementation.*;
import Debug.Init;

public class Register extends Entity {
  private boolean reset=false;
  Value value = Value.UndefinedValue;

public Register(String name, boolean reset) {
    super(name);
    this.reset = reset;
    InitPort();
} // Construction

protected void InitPort() {
  addInactivePort("in", Port.TOP);
  addInactivePort("out", Port.RIGHT);
  addEdgeUpPort("com", Port.LEFT);
  if (reset)
    addEdgeUpPort("reset", Port.BOTTOM);
  else
    removePort("reset");
} // Initialisation des ports
```

```
public void report(Value v, Port sender) {
 if (sender.getName().equals("reset"))
      value = new IntValue(0);
 else
      value = getPort("in").read();
  getPort("out").write(value, this, pout);
} // Comportement adapté au modèle de
// communication.

public ParameterList getParameterList(){
    ParameterList pl =
        super.getParameterList();
    pl.addParameter(new
BooleanParameter(reset,"Reset?","Yes","No"));
    return pl;
  } // Paramètres du composants.

public void useParameter(ParameterList pl) {
    super.useParameter(pl);
    BooleanParameter par =
        (BooleanParameter)pl.getParameter(1);
    reset = par.booleanValue();
    InitPort();
    repaint();
}} // Paramètres du composants.
```

Exemple 1 – Code standard pour la description d'un registre.

Notons, qu'il ne reste que le comportement primaire du registre : mémoriser une valeur de type générique et la modifier lors du chargement (**load**). Toute l'information éliminée par rapport au premier exemple est l'information dont la syntaxe dépend fortement du modèle de composition et de communication sous-jacent. Chaque environnement devrait permettre la construction du composant à partir du comportement primaire, et introduire sa spécificité (comportement secondaire) grâce à un procédé semi-automatique, voire graphique.

```
import sep.type.*;

public class Register implements sep.model.ServiceProvider {
    protected Value content = LevelValue.Undefined;
    public Value load(Value v) {
           return content=v.clone();
    }
}
```

Exemple 2 – Code utilisé par SEP pour la description d'un registre.

Cette simplification nécessite la capacité de manipuler le modèle des composants qui peuvent être construits ainsi que les procédés de communication et de déclenchement de l'exécution du code, de manipulation des paramètres. Toute cette information a été regroupée dans notre méta-modèle des composants **[Mal00]**. Le procédé de construction du composant à partir du méta-modèle est appelé réification.

Selon notre méta-modèle, un composant est constitué de services, d'attributs et de ports. Les paramètres des services sont transmis par des ports de données et leur activation est commandée par des ports de contrôle. Le procédé de réification permet dans ce cas de sélectionner les attributs, les services et d'y associer des ports de données et de contrôle. Il faut de plus sélectionner la sensibilité des ports de contrôle qui déterminera les conditions sous lesquelles le service sera activé.

Avec un tel mécanisme semi-automatique, il est possible de générer un code équivalent à l'Exemple 1 à partir de l'Exemple 2. C'est ce qui est réalisé dans SEP.

L'utilisation de ce procédé est décrite de façon plus détaillée par la Figure 2 et traite l'exemple d'un service mémoire. Cette figure montre qu'à partir du même fournisseur de services, on est capable de construire plusieurs composants différents.

Dans cet exemple, il apparaît deux façons de réutiliser les services. En effet, les fournisseurs de services sont des classes Java, leur conception relève de la technologie objet et ils bénéficient de toutes les méthodes de conception liées à cette technologie. En outre, les services peuvent aussi être réutilisés après réification par héritage entre composants. Les composants sont constitués de services, de ports et d'attributs. L'héritage entre composants consiste à hériter les services, les attributs et l'interface de communication. Lors d'un héritage, l'interface peut être complétée et les services redéfinis.

Dans cette relation d'héritage, contrairement aux langages comme OO-VHDL, la réutilisation du comportement se fait de manière explicite par la réutilisation des services des composants parents, en respectant la même interaction avec l'interface.

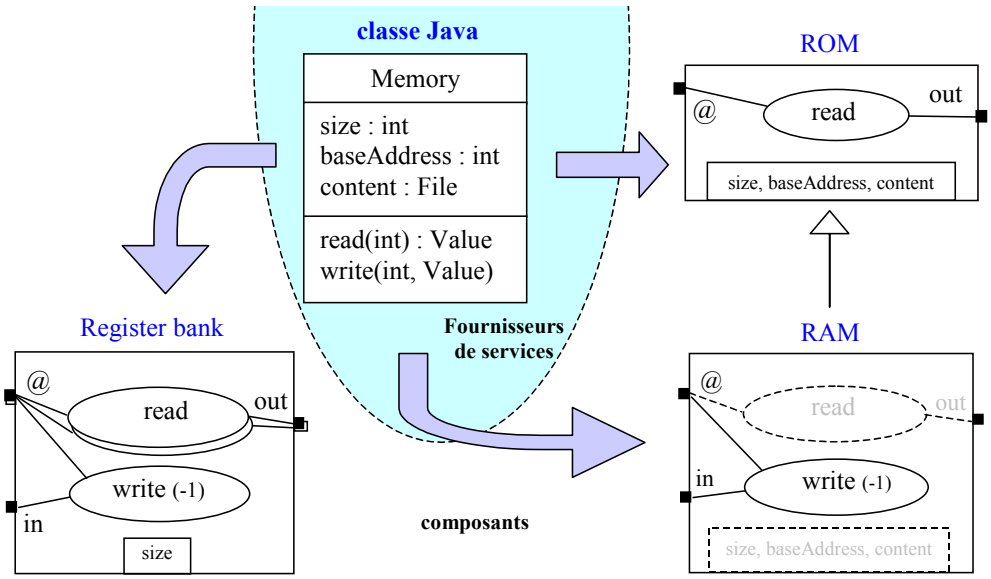


Figure 2 – réification de composants à partir du méta-modèle.

Prenons l'exemple du fournisseur de services *Memory* qui offre les deux services *read* et *write* (cf. Figure 2). Ces services ont un comportement qui dépend des trois attributs *size*, *baseAddress*, *content*. L'attribut *size* définit la taille de la mémoire en nombre d'éléments de type générique *Value*. L'attribut *baseAddress* définit l'adresse du premier élément contenu dans la mémoire et l'attribut *content* définit le contenu de la mémoire sous forme d'un fichier.

A partir de *Memory*, le procédé de réification permet la construction d'une mémoire de type lecture seule (ROM). Le composant ROM a seulement un service de lecture contrôlé par le port *read*. Ce port est construit avec une sensibilité *posedge*, le port réagit et active le service associé lors de l'occurrence d'un front montant. Le service *read* lit l'adresse de lecture sur le port *adr*, puis émet la donnée lue à cette adresse sur le port *out*. Ce composant a trois paramètres de personnalisation. On peut changer sa taille *size*, son adresse de base *BaseAddress* et son contenu peut être lu dans un fichier *File*.

Afin de simplifier le dessin, les ports de commande n'apparaissent pas, il suffit de noter qu'à chaque service correspond un port de commande du même nom.

Une mémoire de type lecture-écriture (RAM) peut aussi être construite si le concepteur utilise le service *write*. Un composant RAM hérite son comportement du composant ROM. Ainsi le composant RAM hérite du service *read*, des ports *in* et *out*, ainsi que des trois attributs du composant ROM. De plus, il définit un service *write* déclenché par l'occurrence d'un front montant sur le port *write*. Pour ce service, l'adresse d'écriture est aussi lue sur le port partagé *adr*. Aucune donnée de sortie n'est émise.

Les deux services *write* et *read* réagissent en concurrence, on utilise un mécanisme de priorité pour résoudre le conflit potentiel lecture-écriture. C'est ainsi, que le service *write* est doté d'une priorité inférieure à celle du service *read*. Quand ils sont invoqués tous les deux simultanément, tout se passe comme si le service *read* avait été exécuté le premier. Après l'invocation des deux services, la donnée présente en mémoire à l'adresse du bus d'adresse est la donnée qui était présente sur le port *in* lors de l'invocation. La donnée présente en sortie sur le port *out* est la donnée qui était en mémoire avant l'invocation des deux services.

Le fournisseur de services *Memory* permet aussi de construire un banc de registres (*RegisterBank*) qui mémorise *size* données de n bits. Il dispose d'une instance du service *write* pour écrire une donnée dans le banc de registre et de deux instances du service *read* pour lire simultanément deux valeurs mémorisées. La priorité est utilisée de la même façon que pour le composant RAM. Le seul attribut utile ici est la taille *size*.

Dans ce modèle la taille des données n'est pas une limitation, en effet, les données sont de type *Value*. Un sous-type de *Value*, *BitValue* permet de gérer les ensembles de bits.

Avec cette même méthode, nous pouvons modéliser une mémoire multi-ports, c'est-à-dire une mémoire avec plusieurs services *write*. Étant donné, qu'il n'est pas possible de définir des priorités relatives entre deux instances d'un même service, les conflits écriture-écriture sont résolus dynamiquement par une fonction de résolution ou à défaut par l'émission d'un message d'avertissement.

Les deux sections suivantes montrent comment ces services améliorent la lisibilité et la réutilisation du comportement des composants.

2.2 Héritage de comportement.

Notre modèle hiérarchique de composition permet, par agencement de composants élémentaires, la définition de blocs appelés modules. Ces modules encapsulent le comportement de leurs composants constituants. Il est parfois intéressant de permettre de façon explicite l'utilisation des services des composants élémentaires directement à travers le module, il s'agit d'un héritage du comportement des composants élémentaires par le module qui les contient.

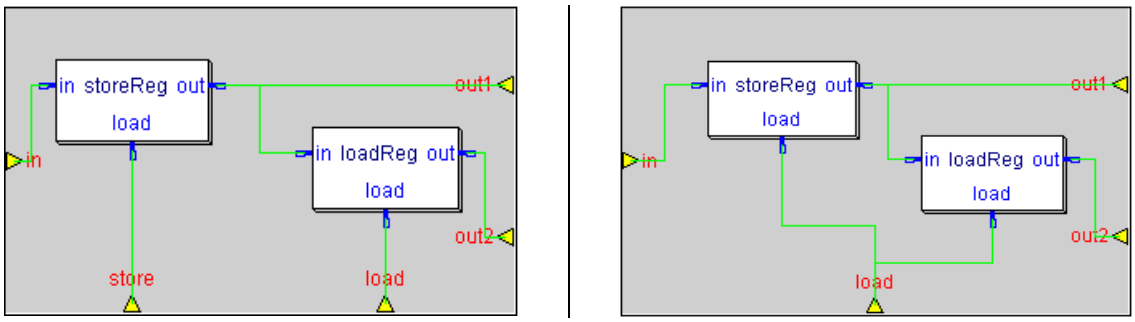


Figure 3 – Contrôle différent et chemins de données identiques.

Illustrons cette notion d'héritage de comportement sur un module composé de deux registres. On peut par exemple réaliser les deux compositions présentées par la Figure 3. Les chemins de données sont dans les deux cas identiques, mais le contrôle est différent. En fait, dans les deux cas, nous essayons par la création de ports, de rendre visible les services élémentaires à partir du module.

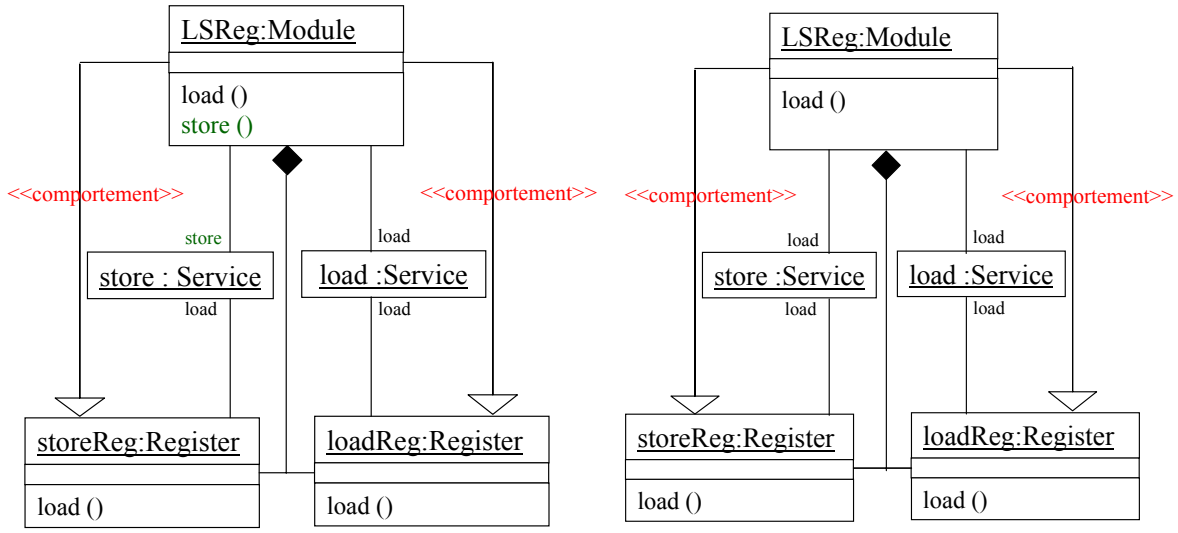


Figure 4 – Héritage de comportement.

Dans le premier cas, le service *load* du registre *loadReg* est vu sous le nom *load* et le service *load* du registre *storeReg* est vu sous le nom *store*. Dans le deuxième cas, les deux services élémentaires sont vus sous le nom *load*, on réutilise une composition parallèle. Dans SEP, nous dirons, que le module hérite du comportement de ses composants constituants. Le nom des services hérités va déterminer une composition parallèle ou un comportement indépendant (cf. Figure 4).

Dans tous les cas, le module présenté dans SEP (cf. Figure 5) aura un seul port de contrôle appelé *com*. Pour activer les services, il suffira d'envoyer sur le port les chaînes de caractères "load", "store" ou "load || store" selon le comportement souhaité. La figure de gauche illustre l'état inactif du port de contrôle dans lequel aucun service n'est demandé, la figure de droite présente le cas où le service *load* est demandé.

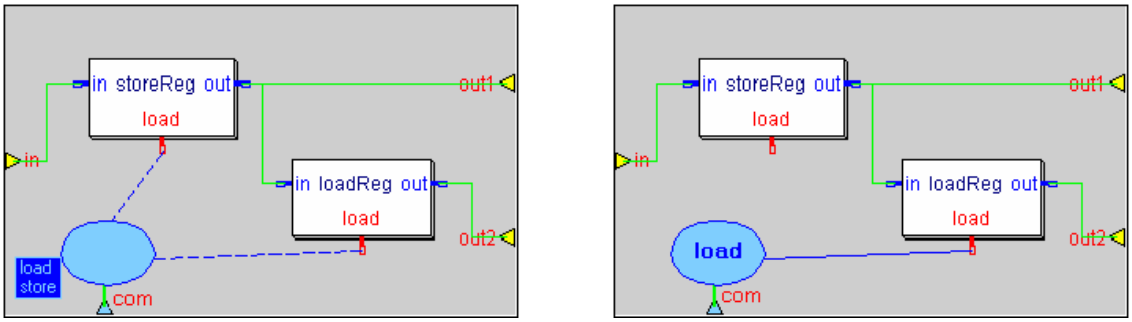


Figure 5 - Port de services.

Cette notion d'héritage de comportement présentée ici sur les services concernent aussi les attributs, un module peut être paramétré en modifiant la valeur d'un attribut de l'un de ses constituants. Pour cela, il faut préciser que le module exporte (hérite) l'attribut d'un de ses constituants.

2.3 Modèles hiérarchiques et services de modules.

Il est également souhaitable de pouvoir composer ces services de façon plus complète. Cette composition séquentielle ou concurrente des services élémentaires permet la définition de services de plus haut niveau – services de modules. Ces services améliorent la lisibilité des modèles et offrent des mécanismes de réutilisation par héritage. En effet, les modules deviennent à leur tour un ensemble de services, d'attributs et de ports.

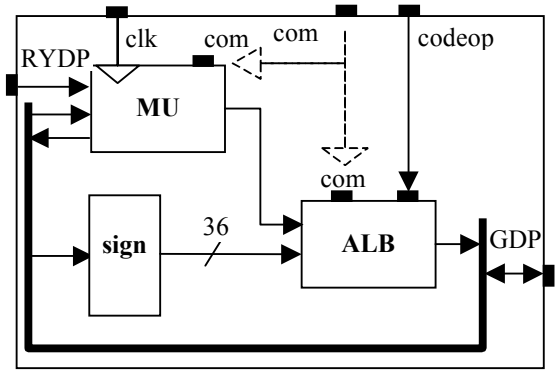


Figure 6 – L'unité de calcul : CU.

Prenons comme exemple la modélisation de l'unité de calcul d'un cœur de processeur de traitement du signal. La Figure 6 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.

Etudions d'une façon un peu plus précise les modules MU et ALB. La Figure 7 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.

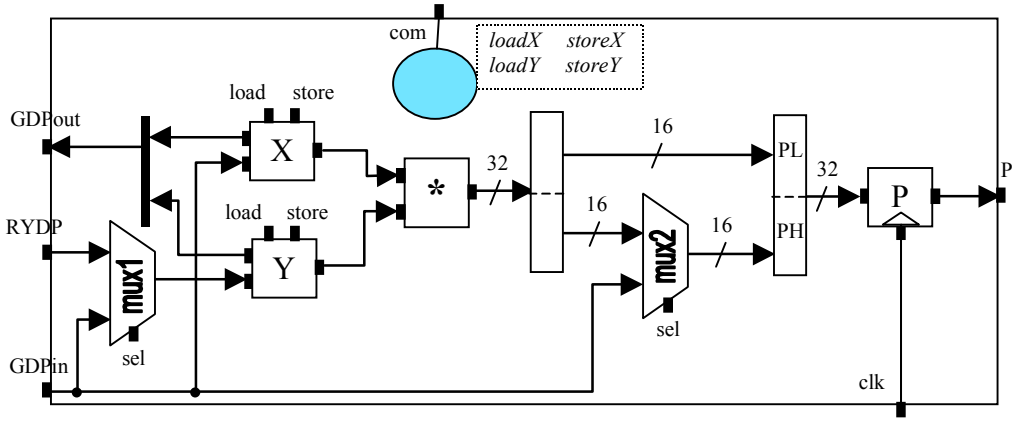


Figure 7 – Le module de multiplication : MU.

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

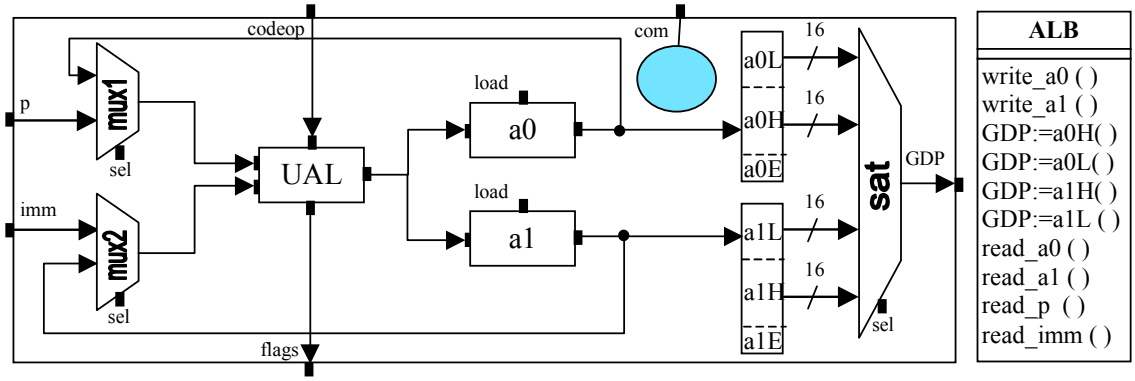


Figure 8 – Le module arithmétique et logique : ALB.

La Figure 8 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 6) 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 construit 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.

2.4 Les services et les vues multiples.

La notion de service étant centrale dans SEP, il apparût indispensable de disposer d'une manipulation graphique de ces services. C'est ainsi que nous avons introduit un mécanisme de vues multiples. La première vue permet de construire l'architecture sous la forme de schémas blocs (cf. Figure 9).

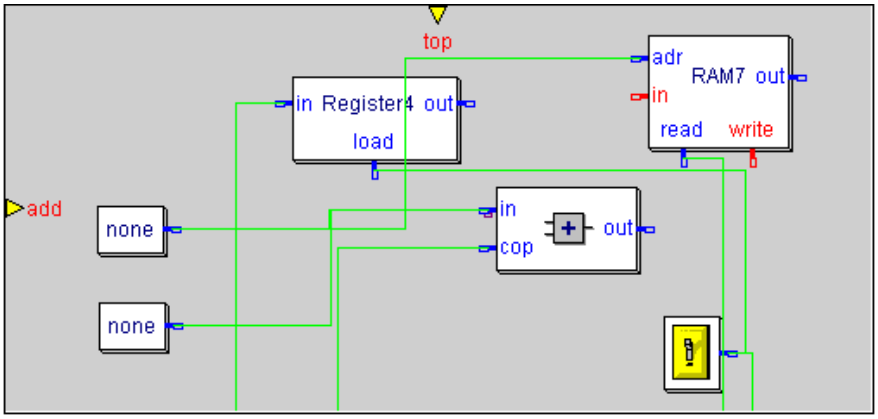


Figure 9 - Exemple de la vue schéma-bloc

La deuxième vue permet une visualisation des services et attributs associés aux composants (cf. Figure 10). Cet exemple met en évidence la manipulation des services dans le cadre d'un schéma d'utilisation (Usage pattern). Ce schéma permet l'observation de l'activation des services de certains composants – identifiés sur le dessin par le mot *component* – par rapport aux activations d'un service de référence – identifié par le mot *horloge*.

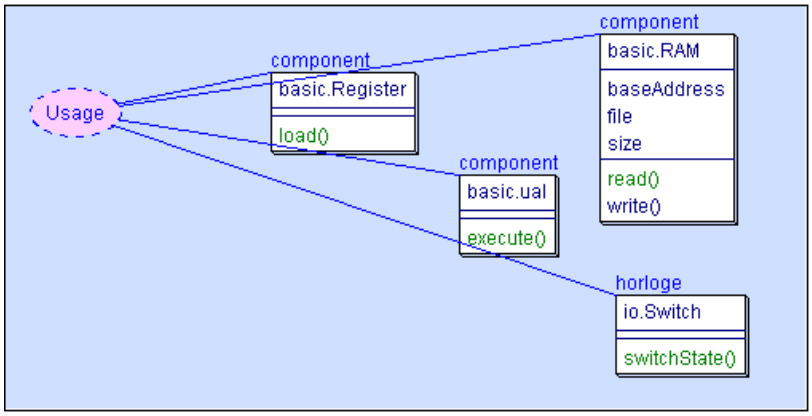


Figure 10 - Exemple de la vue service.

Dans l'exemple présenté ci-dessus, l'utilisateur a sélectionné le service *switchState* du composant *Switch* comme service horloge. Il désire observer l'activation des services *load*, *execute* et *read* des composants *Register*, *UAL* et *RAM*. Cette sélection permet d'obtenir les fenêtres d'évaluation de performances présentées par la Figure 11.

Un autre schéma défini dans SEP est le schéma 'heir' qui est une relation entre un module et un de ses composants. Ce schéma permet de sélectionner les services et les attributs dont le module veut hériter. C'est en utilisant ce schéma que l'utilisateur défini l'héritage de comportement.

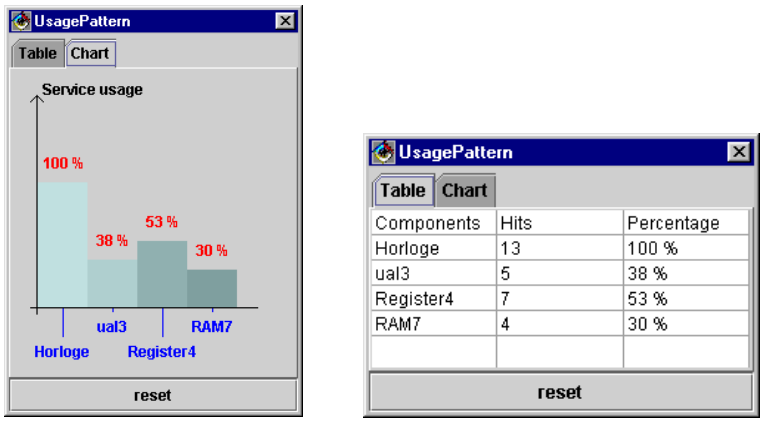


Figure 11 - Evaluation du taux d'utilisation des composants.

Maintenant que la notion de service a été introduite, nous présentons le mécanisme utilisé pour extraire l'information de type sur les composants et valider les modèles construits. La section suivante commence par présenter le mécanisme de validation de la composition. La section 4 présente le mécanisme de validation fonctionnel des services de modules.

3. Composants, services et types.

Le type τ=(Val, Ops) est défini comme une paire constituée d'un ensemble de valeurs Val et d'un ensemble d'opérateurs sur ces valeurs. On se donne une relation d'ordre partiel $\leq$ sur les types, appelée « relation de sous-typage ». Soit τ'=(Val', Ops'), $\tau' \leq \tau$ (τ' est un sous-type de τ) si et seulement si Val'$\subset$Val et Ops $\subset$ Ops'. En clair, un sous-type restreint le domaine des valeurs concernées et peut ainsi agrandir l'ensemble des opérateurs qui y sont relatifs. On dira aussi que τ est un super-type de τ'.

La règle de sous-typage nous permet d'effectuer des conversions d'un sous-type vers un de ses super-types :

$$\frac{\varepsilon : \tau' \quad \tau' \leq \tau}{\varepsilon : \tau} \quad \textit{(règle de sous-typage)}$$

Cette règle énonce que si une expression ε est de type τ' et que le type τ' est un sous-type de τ alors ε est aussi de type τ. La relation de sous-typage étant transitive, ε est aussi de tout type τ'' tel que $\tau' \leq \tau$''. Pour SEP, l'ensemble de tous les types n'est pas fixe. De nouveaux types peuvent être ajoutés conformément à des relations de sous-typage en modifiant l'arbre de sous-typage. Cependant, pour garantir la convergence des algorithmes de sélection d'opérateurs, tous les types définis dans SEP sont des sous-types du type *Value* représenté par la classe *Value*.

Dans SEP, les types des données et des ports sont représentés par des classes. L'ensemble des valeurs correspond à l'ensemble des états définis par les attributs de la classe. Les opérateurs relatifs à un type sont définis par des méthodes de la classe correspondante. Une valeur de type τ est alors représentée par un objet, instance de la classe correspondant au type τ.

Les architectures modélisées manipulent en général des ensembles de bits, cependant il est très désagréable et pénalisant de manipuler en permanence des bits, on utilise alors des abstractions (codages) de plus haut niveau. C'est ainsi qu'on pourra manipuler des données entières (*IntValue*), réelles (*DoubleValue*), représentant des niveaux logiques (*LevelValue*), des chaînes de caractères (*Value*) ou tout autre type défini par l'utilisateur comme par exemple des arbres de décision (cf. section 4). Afin de permettre une définition générique des objets qui manipulent les données, toutes les classes qui définissent un type à manipuler dans SEP doivent hériter de la classe *Value*. Cette classe représente les données sous forme textuelle, il doit donc exister un équivalent textuel non ambigu pour chaque donnée manipulée. Ce choix peut sembler étonnant, mais c'est le choix le moins contraignant. En effet, choisir une représentation sous forme de bits comme représentation minimale obligerait le

concepteur à se concentrer sur le codage binaire de ses données en premier lieu comme on fait en VHDL et ce n'est pas l'objectif recherché. Alors que la représentation textuelle est de toute manière indispensable.

L'interface de communication d'un composant SEP est composée d'un ensemble de ports. Ces ports reçoivent (resp. émettent) des données en provenance (resp. à destination) d'autres ports. SEP a pour objectif d'optimiser la réutilisation des composants, c'est pourquoi les ports d'un composant ne sont pas typés statiquement. En effet, un typage statique des ports, réduirait les possibilités d'ajout de services à un composant et contraindrait le composant à l'utilisation des types existants lors de sa création ou à leurs sous-types. En conséquence, le type des ports est déduit lors de la connexion des composants et une vérification de compatibilité est réalisée. Cette vérification est appelée validation de la composition.

3.1 Le mécanisme de validation des connexions.

Lors d'une définition structurelle, le concepteur doit respecter certaines contraintes inhérentes à la spécification des composants élémentaires et au type des services élémentaires encapsulés. Pour s'assurer que les connexions construites ne risquent pas d'entraîner une violation des contraintes de type imposées par les services, nous procédons en deux étapes.

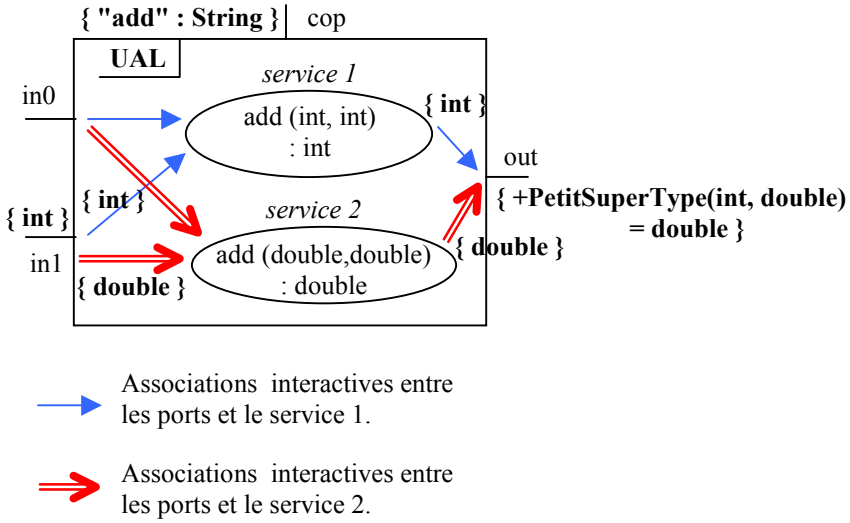


Figure 12 – Typage des ports du composant UAL.

Dans une première étape, il s'agit d'inférer le type du composant élémentaire en fonction des types spécifiés des services qu'il fournit. Le type de chaque port de sortie est inféré comme étant le plus petit super-type commun aux types de retour des services qui émettent sur ce port. En effet, si un composant reçoit une donnée de type int alors qu'il s'attend à recevoir un double, toutes les opérations prévues pourront être exécutées correctement car par définition du sous-type. Toute opération pouvant être effectuée sur une type (double) peut aussi être effectuée sur un de ses sous-type (int).

En revanche, le type de chaque port d'entrée est inféré comme étant le plus grand sous-type commun aux paramètres des services qui utilisent ce port. Dans le cas où le plus grand sous-type commun n'existe pas, une liste des plus grand sous-types est conservée. La Figure 12 montre un exemple d'inférence du type des ports d'une unité arithmétique et logique (UAL). Dans cet exemple, les ports in0, in1 et out sont partagés pour la définition de deux services. Le premier service manipule des nombres entiers (*IntValue*), le second service manipule des nombres réels (*DoubleValue*). Sur la figure, les types *DoubleValue* et *IntValue* sont respectivement remplacés par la notation condensée *int* et *double*. Le plus petit super type commun aux types *IntValue* et *DoubleValue* est *DoubleValue* puisque *IntValue* est un sous-type de *DoubleValue*. En effet, tous les nombres entiers sont aussi des nombres réels. Par contre, il existe des opérations faisant intervenir des entiers et qui n'auraient pas de sens avec des nombres réels (e.g. On peut décaler uniquement d'un nombre entier de bits).

Le type inféré pour les ports in0 et in1 de l'UAL sera donc *IntValue* et le type inféré pour le port out sera *DoubleValue*. On peut souligner que le plus petit super-type commun existe nécessairement puisque *Value* est le super-type de tous les types.

Cette inférence est possible après compilation des services et sans parcours du code Java grâce au mécanisme d'introspection de Java. Ceci rend possible la composition des services sans nécessairement disposer des sources

ce qui est indispensable pour l'utilisation de services contenant de la propriété intellectuelle. L'introspection est un mécanisme puissant de certains langages orientés objets. Ce mécanisme permet à un utilisateur extérieur d'un système à objets (client) de découvrir et de manipuler la structure statique des objets (classe : nom et type des membres, parcours de la hiérarchie d'héritage et de composition, invocation de méthodes).

La deuxième étape s'assure, par une vérification lors de la création de connexions entre composants, que les connexions ne sont pas aberrantes vis-à-vis des types inférés. Ces connexions aberrantes engendreraient des erreurs de conversion dynamique de types lors de la simulation si elles n'étaient interdites.

Les connecteurs ont pour rôle la transmission de données entre plusieurs composants. Dans l'étape précédente, nous avons déduit le type de tous les ports, nous disposons désormais pour chaque connecteur C, du type P_{src} du port source P (émetteur de données) et le type P_{dst} du port destination (récepteur de données).

Il y a alors plusieurs cas possibles :

1. Si P_{dst} n'est pas une liste de types (le plus grand sous-type commun a pu être déterminé) et $P_{src} \leq P_{dst}$ alors la connexion est acceptée inconditionnellement. Sinon voir 3.
2. Si P_{dst} est une liste de types, alors tous les types de P_{dst} qui ne sont pas des super-types de P_{src} sont rejetés et les services correspondant sont désactivés. On peut noter qu'il y aura forcément des types rejetés, sinon P_{src} serait un sous-type commun et aurait pu être considéré comme le plus grand.
3. Si dans les deux cas précédent si la connexion n'est pas acceptée, alors le type P_{src} est remis en question. C'est-à-dire que le service du composant source qui émet le type le plus grand sur le port concerné est désactivé. Le nouveau type P_{src} est réévalué. La procédure de vérification recommence alors avec le nouveau type src. Si tous les services du composant source qui émettent sur P sont désactivés, alors la connexion est refusée.

Dans le cas d'un connecteur où il y a plusieurs émetteurs (bus) et/ou récepteurs, chaque connexion point à point est considérée et éventuellement rejetée indépendamment des autres.

Le rejet d'une connexion consiste en un message d'erreur et rend impossible l'utilisation en simulation des connexions concernées.

En utilisant le type inféré des ports sources (émetteurs de données) et destinations (récepteur de données), on s'assure qu'ils sont compatibles vis-à-vis d'un connecteur.

On pourra par exemple déduire qu'étant données les constructions réalisées, l'unité arithmétique et logique ne pourra pas effectuer d'opérations sur les int et devra se limiter aux services sur les double.

3.2 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 alors de réaliser des opérateurs polymorphes, c'est-à-dire des opérateurs qui sous la même appellation (ex. addition) peuvent représenter des opérations (types) différentes. 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.

Ce mécanisme intéressant des langages objets s'appelle liaison dynamique. La section 3.3 décrit les limitations d'une modélisation naïve en utilisant l'héritage et le mécanisme de liaison dynamique des langages orientés objets populaires tels que Java ou C++. Puis nous présentons dans la section 3.4 un mécanisme plus évolué qui permet d'atteindre nos objectifs. La section 4 montre ensuite comment ce mécanisme permet la validation de modèles dans SEP.

3.3 Surcharge, redéfinition et liaison dynamique en Java.

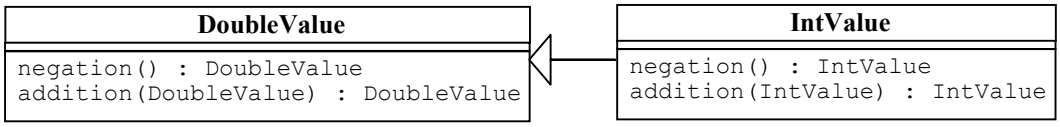


Figure 13 – modélisation naïve d'opérateurs polymorphes en présence d'héritage.

Prenons un exemple simple pour illustrer le problème de la représentation d'opérateurs polymorphes. 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 13. Chacune de ces classes définit les opérations sur les données du type qu'elle représente. Avec cette construction, le mécanisme de liaison dynamique de Java ne permet pas d'obtenir le résultat escompté en présence de polymorphisme, c'est-à-dire en présence de variables qui peuvent référencer des objets de classes (types) différentes mais avec une interface commune (ici une même variable pourrait référencer soit un *DoubleValue*, soit un *IntValue*).

Il y a pour cela deux raisons. La première est que le mécanisme de redéfinition[1] 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. La méthode addition de la classe *DoubleValue* est surchargée dans la sous-classe *IntValue*. On rappelle 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 et des signatures différentes. La signature est constituée du nom de la méthode, de la liste ordonnée du type des paramètres et ne contient pas le type de retour.

Cette implémentation introduit une dissymétrie dans le rôle des paramètres. En effet, le paramètre implicite[2] et le paramètre explicite n'ont pas le même rôle. C'est alors que l'opération 12 (*IntValue*) + 15,5 (*DoubleValue*) ne sera 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 appellera une méthode de classe *DoubleValue* ayant un paramètre de type *IntValue* (ou un super-type). En présence de polymorphisme, le mécanisme de liaison dynamique de Java ne suffit plus et n'a pas le comportement espéré. En fait, ce problème concerne tous les opérateurs n-aires, c'est-à-dire faisant intervenir un ou plusieurs paramètres d'un sous-type de la classe qui définit l'opérateur. Ce problème est détaillé dans la suite de la section.

Une classe C est compatible avec un type τ si et seulement si le type τ_C représenté par la classe C est un sous-type de τ : $\tau_C \leq \tau$.

Une méthode est dite applicable par rapport à une invocation d'opération si et seulement si les deux conditions suivantes sont remplies :

1. Le nombre de paramètres de la méthode déclarée est égal au nombre d'arguments de l'opération invoquée.
2. Le type de chaque référence doit être un sous-type du type du paramètre correspondant. Chaque argument représenté par un type primitif Java doit avoir une classe équivalente[3] compatible avec le type du paramètre correspondant.

Une méthode $\mu 1$ est plus spécifique qu'une méthode $\mu 2$ si, à chaque fois que $\mu 1$ est applicable pour une invocation donnée, alors $\mu 2$ l'est aussi (il peut exister des cas où $\mu 2$ est applicable et $\mu 1$ ne l'est pas). Plus précisément :

[1] 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.

[2] 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.

[3] Pour les types primitifs de Java, une classe équivalente est définie dans SEP. Par exemple, les types *int*, *double*, *boolean*, *String* ont respectivement pour classe équivalente les classes *IntValue*, *DoubleValue*, *LevelValue* et *Value*.

Soit μ1 une méthode de nom m déclarée dans la classe C1 avec n paramètres dont les types sont $\tau 1_1$, …, τn_1 ; Soit μ2 une méthode de nom m déclarée dans la classe C2 avec n paramètres dont les types sont $\tau 1_2$, …, τn_2 ; Alors μ1 est dite plus spécifique que μ2 si et seulement si $\forall i \mid 1 \leq i \leq n, \tau i_1 \leq \tau i_2$.

Le mécanisme d'invocation de méthodes en Java **[Gos&Co00]** est assez compliqué. Tout d'abord, à la compilation la signature de la méthode à invoquer est résolue. Puis à l'exécution, la liaison dynamique permet de choisir la définition adaptée correspondant à la signature résolue.

A la compilation, il faut d'abord choisir la classe S dans laquelle la méthode à invoquer va être recherchée, c'est la classe correspondant au type de l'expression à partir de laquelle la méthode est invoquée (type de l'argument implicite).
Ensuite, il faut choisir la signature adaptée aux arguments d'appel, on commence alors par rechercher dans S l'ensemble des méthodes applicables. Parmi les méthodes applicables, on choisi la méthode la plus spécifique M. Le type de l'expression d'invocation est alors le type de retour de la méthode retenue.

A l'exécution, il faut alors déterminer la classe effective de l'argument implicite si elle est différente du type de l'expression. On parcourt alors la hiérarchie d'héritage à partir de cette classe vers les super-classes à la recherche de la première définition de méthode dont la signature est identique à la méthode retenue M.

Dans l'exemple de la Figure 13, la classe *DoubleValue* définit la méthode *DoubleValue addition(DoubleValue)*, il serait alors souhaitable que la classe *IntValue* surcharge[4] cette méthode en définissant *IntValue addition(IntValue)*. Observons alors le comportement d'un tel système en présence de polymorphisme en fonction des classes des objets concernés et en fonction du type des références concernées :

<table>
<tr><th>Type de l'argument implicite:this</th><th>Classe effective de this</th><th>Type de la référence de l'argument</th><th>Classe effective de l'argument</th><th>Méthode exécutée.</th><th></th></tr>
<tr><td colspan="2">IntValue / DoubleValue</td><td colspan="2">DoubleValue</td><td rowspan="2">DoubleValue addition(DoubleValue)</td><td rowspan="2">(1)</td></tr>
<tr><td colspan="2">DoubleValue</td><td colspan="2" rowspan="2">IntValue</td></tr>
<tr><td colspan="2" rowspan="2">IntValue</td><td>IntValue</td><td>(1)</td></tr>
<tr><td rowspan="2">DoubleValue</td><td rowspan="2">IntValue</td><td>DoubleValue</td><td>IntValue</td><td>DoubleValue addition(DoubleValue)</td><td>(2)</td></tr>
<tr><td colspan="2">IntValue</td><td>DoubleValue addition(DoubleValue)</td><td>(3)</td></tr>
</table>

(1) Tout se passe bien dans ces cas là, l'opérateur adapté est invoqué. Il n'y a pas d'utilisation du polymorphisme puisque le type des références est identique aux classes effectives.

(2) Dans ce cas la méthode addition est surchargée, le mécanisme de liaison dynamique n'est pas utilisé. Le choix de la méthode se fait en fonction du type de la référence de l'argument. La seule méthode applicable avec un argument de type *DoubleValue* est *Double addition(DoubleValue)*. **On remarque que l'opération addition sur les *DoubleValue* sera appliquée même si la classe effective des deux paramètres est *IntValue*. Ce n'est pas le comportement attendu !**

(3) Etant donné que nous sommes en présence de deux objets de classe effective IntValue, nous aurions préféré que ce soit la méthode *IntValue addition (IntValue)*. Or, la recherche d'une méthode applicable se fait à la compilation dans la classe représentant le type de l'argument implicite (ici *DoubleValue*). Avec ce choix, la méthode *IntValue addition(IntValue)* n'est alors pas visible.

Note : La classe *DoubleValue* n'est pas une sous-classe de la classe *IntValue* donc lorsque le type de la référence est *IntValue* la classe effective de l'objet ne peut en aucun cas être *DoubleValue*. De même, lorsque la classe effective est *DoubleValue*, le type de la référence ne peut être *IntValue*.

Afin de résoudre le point (3), il est indispensable de définir dans la classe *DoubleValue* une méthode *DoubleValue addition(IntValue)*. La méthode addition de la classe *IntValue* est alors une redéfinition et doit donc avoir le même type de retour. Afin d'assurer un comportement symétrique, c'est-à-dire que l'addition d'un entier à un double soit traitée de la même façon que l'addition d'un double à un entier, cette méthode devrait être définie de la façon suivante :

[4] Dans ce cas, il y a surcharge car la signature est différente, en effet dans un cas le paramètre explicite est de type *DoubleValue* alors que dans l'autre cas, il est de type *IntValue*. Dans le cas de la surcharge, un type de retour différent est autorisé.

```
public DoubleValue addition(IntValue v) { return v.addition(this); }
```

Le point (2) ne peut être résolu qu'en effectuant un test dynamique de type (typecase) dans la méthode *DoubleValue addition(DoubleValue)* et appeler après une conversion explicite de type (cast) l'opération *IntValue addition(IntValue)* de manière appropriée. Ceci est contre le principe de modularité et rend la maintenance difficile. En effet, il faudrait définir une méthode et donc modifier a posteriori la classe *DoubleValue* à chaque fois qu'un sous-type serait défini.

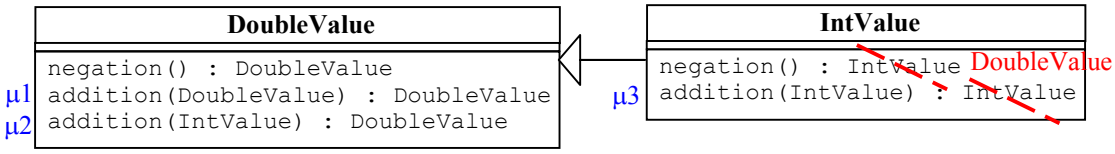


Figure 14 – modélisation complète.

Le diagramme UML de la Figure 14 tient compte de ses modifications, les résultats suivants sont alors obtenus :

Type de this	Classe de this	Type de l'argument	Classe de l'argument	Méthode invoquée
IntValue/DoubleValue		DoubleValue		μ1
DoubleValue		IntValue		μ2 puis μ1 (redirection)
IntValue				μ3
DoubleValue	IntValue	DoubleValue	IntValue	μ1 puis μ3 (cast)
		IntValue		μ3 (liaison dynamique par μ2)

Ces résultats sont satisfaisants quant au comportement obtenu. Mais comme il a déjà été dit, la méthode mise en œuvre ne permet pas la modularité. Une autre façon de réduire la dissymétrie dans le rôle joué par l'argument implicite et par l'argument explicite serait de n'utiliser que des méthodes statiques. On obtiendrait alors la structure de classes présentée par la Figure 15.

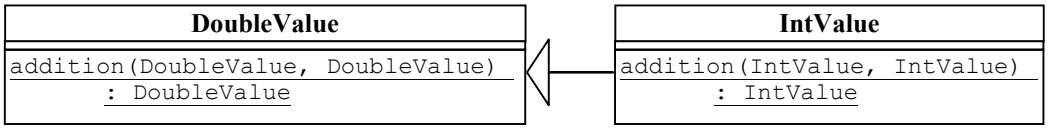


Figure 15 – schéma d'héritage ne faisant intervenir que des méthodes statiques.

Aucun paramètre ne joue un rôle privilégié dans la détermination de la méthode à invoquer, mais le mécanisme de liaison dynamique ne fonctionne plus et le choix se fait alors systématiquement en fonction du type des références des paramètres et non de la classe effective des objets manipulés. On obtient alors le résultat attendu en déterminant dynamiquement la classe effective des arguments manipulés comme pour la résolution du cas (2).

Ces deux solutions qui utilisent les mécanismes offerts par le langage Java ne nous conviennent évidemment pas, car elles ne permettent pas d'ajouter des types à volonté en conservant un comportement cohérent. Nous proposons donc un autre mécanisme d'invocation de méthodes à partir des classes effectives des arguments d'appel. Ce mécanisme implémenté dans les composants SEP, nous permet de choisir le service le mieux adapté aux données qui parviennent aux ports.

3.4 Notre approche.

Avec ce mécanisme, un modèle d'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. 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.

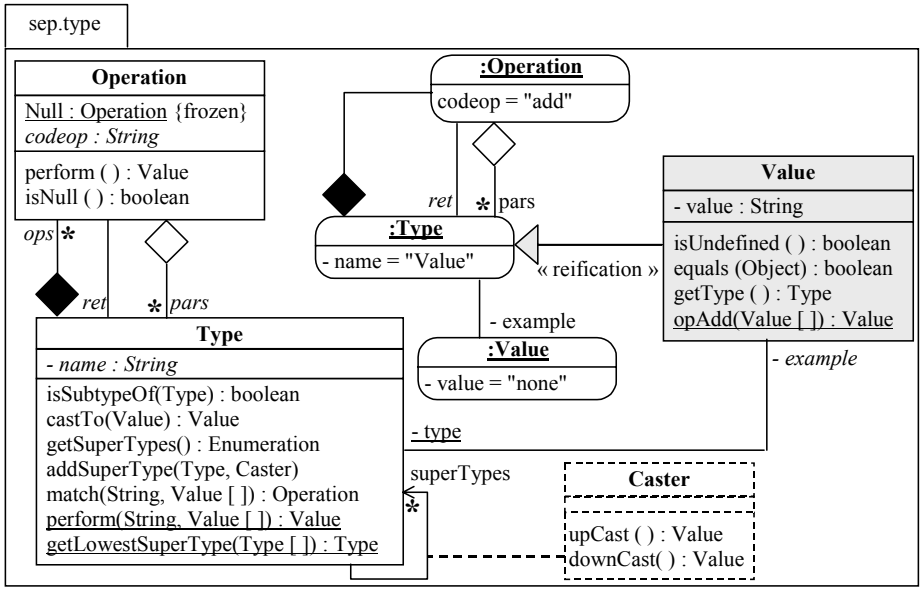


Figure 16 – Méta-modèle des types manipulés par SEP.

Dans SEP, il s'agit de choisir dynamiquement les opérateurs adaptés en fonction de la classe effective[5] des objets transmis et non en fonction du type des arguments qui interviennent. Les classes de plusieurs objets peuvent intervenir dans le choix du service, il s'agit d'une extension du mécanisme de liaison dynamique de Java.

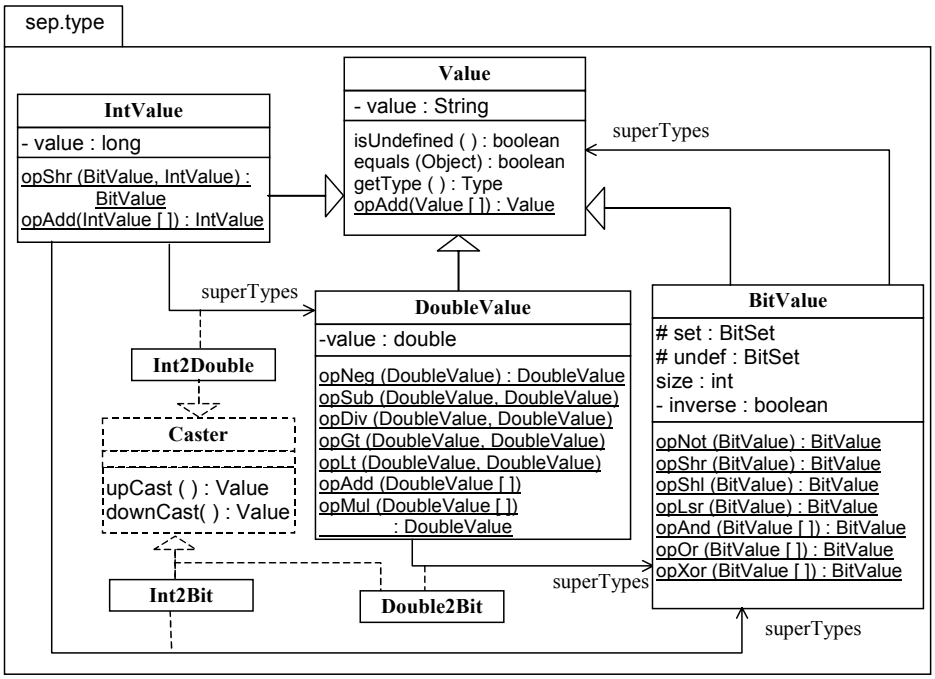


Figure 17 – Arbre de sous-typage (Value, DoubleValue, BitValue).

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 (cf. Figure 16). 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 17). La contrainte que nous nous sommes fixés est de ne pas avoir à modifier la classe représentant un type lors de la

[5] On appelle classe effective, la classe dont est issu l'objet par opposition au type statique de la référence vers cet objet. Par exemple, le polymorphisme autorise un objet de la classe *IntValue* à être référencé par une référence de type statique *Value* puisque IntValue hérite de *Value*, la classe effective est dans ce cas *IntValue*.

définition d'un sous-type. La sous-section précédente montre 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 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 17 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.

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.

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

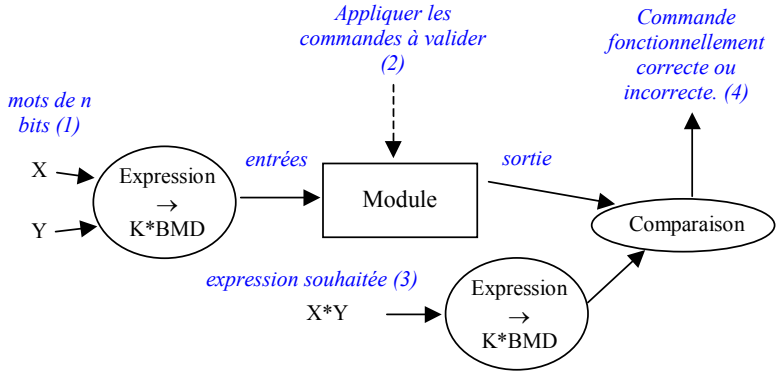


Figure 18 – Principe du mécanisme de validation fonctionnelle dans SEP.

4.1 Le principe.

Notre objectif est de valider fonctionnellement les services de modules que l'on créé. La section 2.3 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.

Le principe est 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 manipulées 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 **[DBR96]**. 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 $B^n \rightarrow Z$, 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 18 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.

On peut noter que Bryant **[Bry91]** avait déjà introduit une technique de validation similaire. L'originalité de notre approche réside dans deux aspects :

1. Bryant utilisait cette technique dans le cadre de l'utilisation classique des arbres de décision, c'est-à-dire pour la validation fonctionnelle de composants arithmétiques de base à partir d'une description sous forme de portes logiques. Il s'agirait par exemple de nos jours de valider une multiplieur 256 bits. Nous proposons une validation de services de plus haut niveau.
2. Cette technique de simulation symbolique est effectuée dans le cadre de SEP, sans changer le modèle de la micro-architecture ou du module étudié. En définitive, il s'agit de profiter du mécanisme de liaison dynamique mis en place afin d'effectuer suivant le type des entrées soit de la simulation standard, soit de la simulation symbolique.

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 $a + \mathbf{f}$, 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 $m \times \mathbf{f}$, 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 $a + m \times \mathbf{f}$.

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 $0+1\times\ldots$ – tous les arcs qui se terminent sur des sommets qui ne sont pas des feuilles, de valuer avec (0,1)

[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 $B^n \rightarrow B$.
[7] K*BMD : Kronecker Multiplicative Moment Diagram.
[8] Edge-Valued Binary Decision Diagram.
[9] Multiplicative Binary Moment Diagram.

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 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 **[Bry92]**. Les K*BMD sont présentés dans **[HoD99]**.

La section 3 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.

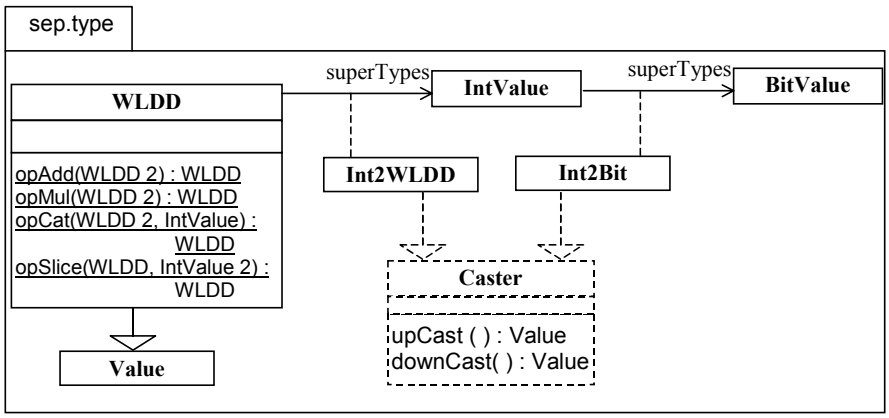
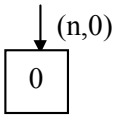


Figure 19 – Définition du type K_BMD.

Le type *K_BMD* est ajouté dans l'arborescence des types de SEP (cf. Figure 19). 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 $B^n \rightarrow Z$. 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 (cf. Figure ci-contre).



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.

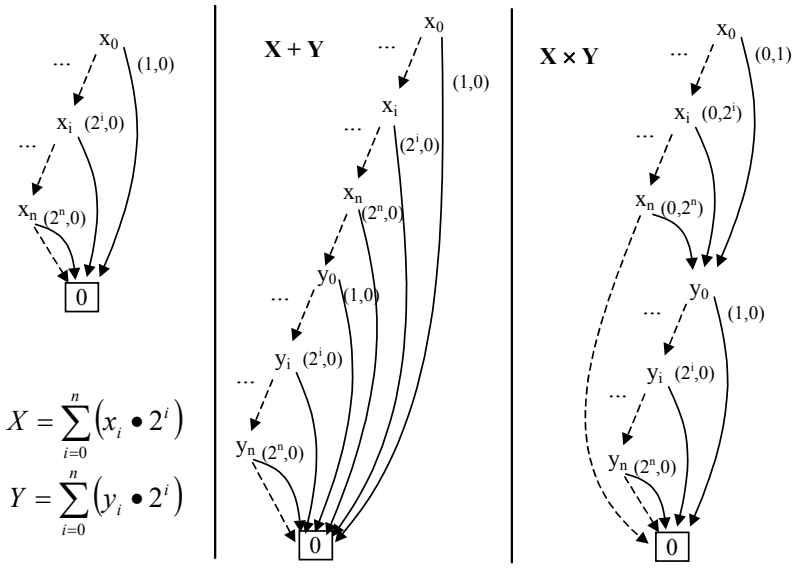


Figure 20 – Addition et multiplication de deux mots de n+1 bits.

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 $B^n \rightarrow Z$ 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 20. Un calcul identique peut être fait pour la soustraction.

Calculons maintenant la décomposition positive de Davio de la fonction $f \times g$ par rapport à la variable x_i. On a $f \times g = (f_i^0 + x_i \times (f_i^1 - f_i^0)) \times (g_i^0 + x_i \times (g_i^1 - g_i^0))$. 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 $(f \times g)_D = (f_D + f_D) \times (g_D + g_D)$. Le K*BMD correspondant est donné par la Figure 20.

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

La concaténation de deux mots de bits a et b est alors $a+2^n$ x 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 $k \geq n+m$ 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 $(f \times g)\%h = (f\%h \times g\%h) \% h$, la taille du K*BMD résultant est souvent linéaire surtout si *f* et *g* sont indépendantes de *h* (cf. **[HoD99]**).

4.4 Un exemple.

Prenons l'exemple de l'unité de calcul présentée au chapitre 2.2. 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 $A0 = \sum_{i=0}^{35} \left(a_i \times 2^i \right)$, les deux entrées RYDP et GDP_{in} prennent les valeurs $RYDP = \sum_{i=0}^{35} \left(rydp_i \times 2^i \right)$ et $GDP_{in} = \sum_{i=0}^{35} \left(gdp_i \times 2^i \right)$. 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 $A0+RYDP \times GDP_{in,}$ qui est une expression de la fonction que l'on souhaite obtenir.

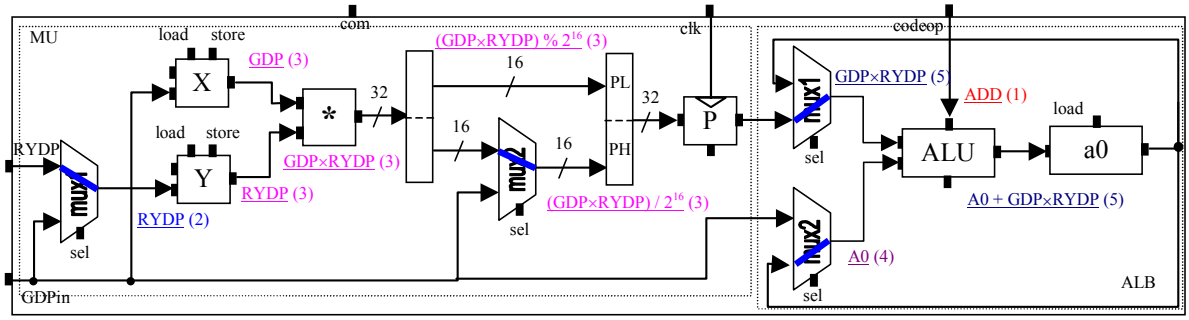


Figure 21 – Scénario de validation.

Dans le cas de l'architecture présentée, la comparaison indique que le service est correct. Pour s'en convaincre, regardons instant par instant le comportement de chaque composant (cf. Figure 21).

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 et perspectives.

Nous avons présenté une technique originale, orientée objet de modélisation d'architectures matérielles. Nous avons montré comment les modèles réalisés permettent une meilleur lisibilité et une meilleure réutilisation qu'avec les langages classiques de description du matériel tels que VHDL ou Verilog. Ces améliorations sont obtenues grâce à :

- La notion de fournisseur de services qui permet réduire à l'essentiel la description du comportement des composants.
- La notion de service de module qui améliore la lisibilité des composants composites et fournit un mécanisme d'héritage de comportement en un composant et le module qui le contient.
- La notion de vue multiple avec laquelle il est possible d'utiliser le langage le plus adapté pour la description de chaque aspect d'un modèle.

Un mécanisme d'évaluation des performances des architectures modélisées est proposé. Ce mécanisme est basé sur une simulation de type événement discret adapté à notre modèle.

Enfin, nous avons présenté une technique basée sur l'utilisation d'arbres de décision pour la validation de modèles d'architectures matérielles. 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 architecture haute performance multi-processeurs.

Parmi les extensions possibles, on peut citer l'intégration de composants VHDL pour réutiliser les modèles existants. D'autre part, le procédé décrit ne dépend pas du mode de communication, il semble très prometteur

d'étudier une adaptation de SEP à des architectures logiciels en intégrant les modes de communication définis dans les 'capsules' ou définis par SDL.

Ces techniques de modélisation doivent être appliquées à des architectures hautes performantes du domaine de la chromodynamique quantique dans le cadre d'une collaboration avec l'université d'Edimbourg et de la comparaison avec l'environnement HASE (Hierarchical computer Architecture design and Simulation Environment).

La dernière étape de ce travail consiste à valider les techniques proposées sur des architectures et des applications issues de projets industriels. Nous avons ainsi modélisé un processeur de traitement du signal de DSP group (OAK+), un processeur RISC de ARM limited (ARM7TDMI) et une architecture bi-cœur composée du OAK+ et du ARM7.

6. Bibliographie.

[AnF99] *« Jester : a Reactive Java extension proposal by Esterel Hosting. »*
M.Antoniotti, A.Ferrari. http://www.parades.rm.cnr.it/projects/jester/jester.html.

[BaB96] « *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

[BAP99] « *A Prototyping Method of Embedded Real Time Systems for Signal Processing Applications. »*
Luc Bianco , Michel Auguin , Alain Pegatoquet. Euromicro 99, 7-10 septembre, Milan, Italie, 1999.

[BeD97] *« Object-Oriented Extensions to CHDL : The LaMI Proposal »*
J. Benzakki, B. Djaffri. IFIP 1997. Chapman & Hall. p. 334-347.

[Ben00] *« Objets pour la modélisation de Systèmes Matériels : intérêts, évolutions et tendance. »*
J.Benzakki, Habilitation à diriger des recherches, Université d'Evry Val d'Essonne, janvier 2000.

[Bry91] *« Formal Hardware Verification by Symbolic Simulation »*
R.E.Bryant. VLSI Logic Synthesis and Design, IOS Press, 1991, p.125-132.

[Bry92] *« Symbolic boolean manipulation with ordered binary-decision diagrams »*
R.E.Bryant. ACM Computing surveys, 24(3) :293-378, 1992.

[DBR96] *« K*BMDs : A new Data Structure for Verification»*
R. Drechsler, B. Becker, S. Ruppertz. IEEE European Design & Test Conference (ED&TC'96), pp. 2-8, Paris, 1996.

[Has98] *« A Hierarchical Computer Architecture Design and Simulation Environment »*
P.S. Coe, F.W. Howell, R.N. Ibbett and L.M. Williams. ACM Transactions on Modeling and Computer Simulation vol. 8, no. 4, octobre 1998.

[HoD99] *« Formal Verification of Word-Level Specifications»*
S. Höreth, R.Drechsler. IEEE Design, Automation and Test in Conference (DATE'99), Munich, 1999.

[Mal00] *« Modélisation et Evaluation de Performances d'architectures matérielles numériques. »*
F. Mallet, thèse de doctorat, Université de Nice-Sophia Antipolis, décembre 2000.

[MBD98] « *Hardware Architecture Modelling using an Object-oriented Method. »*
F. Mallet, F. Boéri, J-F. Duboc. Proceedings of the 24th Euromicro conference, Sept. 1998, vol I, p.147-153.

[MeT00] *« A classification and comparison Framework for Software Architecture Description Languages. »*,
N. Medvidovic, R.N. Taylor, IEEE Transactions on Software Engineering, Vol.26, No. 1, janvier 2000.

[Pet99] *« Proposed Language Requirements for Object-Oriented Extensions to VHDL. »*
G.D. Peterson, Proc. of Forum on Design Languages, FDL'99, France, sept. 99.

[ScN95] *« Inheritance Concept for Signals in Object-Oriented Extensions to VHDL. »*
G. Schumacher, W. Nebel. Proceedings of the EURO-DAC'95 with EURO-CHDL'95.

[Swa95] *« 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, 1st. Qtr. 1995
http://rassp.aticorp.org/newsletter/html/95q1/news_6.html

[Syn97] *« Massively Parallel Computing Systems with Real Time Constraints, The 'Algorithm Architecture Adequation' Methodology. »*
Y. Sorel. Massively ParallelComputing Systems, May 1994.