Chapitre II- Un modèle générique des architectures matérielles numériques.

Objectifs du chapitre :

- Introduire le modèle générique.
- Présenter la sémantique des composants et des connecteurs.
- Introduire la notion de service.
- Définir l'héritage de comportement d'un composant.

Dans le domaine du traitement du signal, le cycle de vie des processeurs et des architectures numériques est de plus en plus court[1]. Il est donc impératif de minimiser toutes les phases qui précèdent la commercialisation sous peine que l'architecture soit immédiatement obsolète. Il faut donc minimiser non seulement le temps de conception des architectures mais aussi le temps de production des outils d'analyse qui doivent permettre aux clients d'évaluer le processeur proposé pour la réalisation de leur application. Par exemple, le client peut vouloir évaluer le comportement d'un cœur de DSP[2] en cours de conception pour l'utiliser dans un ASIC[3] qu'il veut réaliser. Les résultats de cette évaluation influeront sur l'architecture du DSP si le client est important. Dans ce cas, il est indispensable d'adapter à moindre coût l'architecture du DSP pour obtenir des performances satisfaisantes avec l'ASIC.

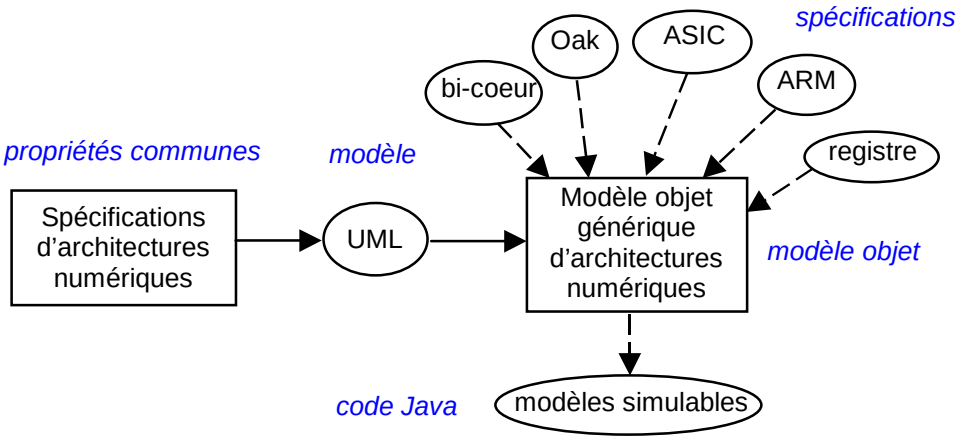

Figure 1- propriétés communes des architectures matérielles regroupées dans un modèle générique.

Actuellement, la réalisation des outils d'analyse et de simulation spécifiques nécessite quasiment la réalisation complète du DSP. Les modifications proposées par les concepteurs de

[1] De l'ordre de deux ans il y a encore quelques années, il est maintenant de l'ordre de 6 mois.
[2] Digital Signal Processor : Processeur de traitement du signal.
[3] Application-Specific Integrated Circuit : Circuits de silicium destinés à une application spécifique et réalisés par combinaison de cœurs de processeurs et de circuits numériques câblés (pré-processeurs).

l'ASIC peuvent alors remettre en cause complètement l'architecture du DSP. C'est pourquoi, nous proposons une méthode de conception interactive et incrémentale qui permet une évaluation de l'architecture, très tôt pendant le cycle de conception, par rapport à une application et dans un contexte particulier. Pour cette évaluation, les détails d'implémentation (codage des instructions, normes de contrôle de bus) ne sont pas primordiaux en première approche. Cette méthode et les outils associés ne dépendent pas de l'architecture modélisée et permettent de réutiliser les modèles précédents archivés en bibliothèque.

Nous avons identifié les propriétés communes aux architectures matérielles que nous voulons modéliser. Ces propriétés sont représentées en UML par un modèle objet générique des architectures. Ainsi, à partir des spécifications d'une architecture qui respectent les propriétés communes identifiées (processeur, ASIC, architecture logique), nous pouvons construire de façon incrémentale un modèle qui permettra l'évaluation de performances en simulation (cf. Figure 1).

Dans ce chapitre, la section 1- présente les propriétés communes identifiées en détaillant le modèle générique. La section 2- explique comment dans le cadre de ce modèle des descriptions de comportement par des modèles logiciels sont possibles en introduisant la notion de service. La section 3- montre ensuite le mécanisme de description structurelle du comportement et l'extension de la notion de services aux modules. Enfin, la section 4- introduit la notion d'héritage d'une description structurelle. Ce chapitre se contente de définir la sémantique des composants et des connecteurs. Le chapitre V présente notre modèle de simulation et la mise en œuvre de techniques qui permettent de respecter cette sémantique.

1- Le modèle générique.

L'entité de base de modélisation dans SEP est le composant. Ce modèle objet générique (cf. Figure 2) décrit en UML la définition récursive et hiérarchique d'un composant. Un rappel sur les notations UML utilisées est présenté en annexe. Ce chapitre détaille l'information contenue dans ce modèle et précise la sémantique des composants et des connecteurs.

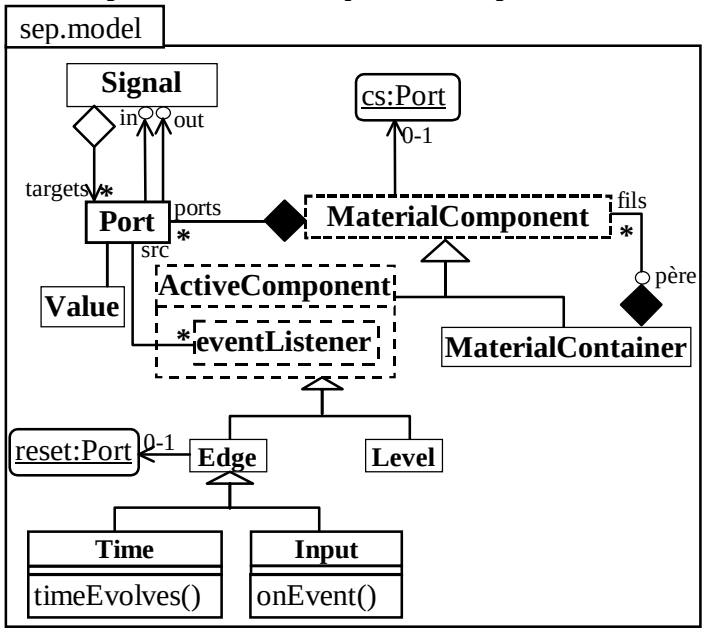

Figure 2 – Le modèle générique des architectures matérielles – diagramme statique UML.

Les composants sont modélisés par la classe *sep.model.MaterialComponent*. On distingue deux catégories de composants :

module
- les composants *sep.model.MaterialContainer* appelés aussi modules et dont le comportement est représenté par une description structurelle, c'est-à-dire par composition d'autres composants ;

composants élémentaires
- les composants élémentaires représentés par la classe *sep.model.ActiveComponent* et dont le comportement est modélisé par un ensemble de services.

Le comportement des composants est complètement autonome, il ne dépend pas de la configuration dans laquelle ils sont placés ce qui assure la modularité et la réutilisation. Les composants communiquent avec l'extérieur à travers leur interface de communication. Cette interface est constituée d'un ensemble de ports munis d'une sensibilité qui définit les conditions sous lesquelles les composants réagissent. Les ports sont reliés entre eux par des signaux, ils communiquent avec le composant dont ils dépendent en utilisant le ou les 'eventListener' qui leur sont associés. Les composants utilisent les ports pour échanger des données (représentées par la classe *sep.model.Value*) à travers ces signaux de communication. Une donnée émise sur un port de sortie p_s d'un composant est diffusée vers le ou les ports connectés par un signal au port p_s. Cette autonomie rend possible la définition de nouvelles configurations qui modifient l'interconnexion des composants. Une connexion directe entre deux composants, c'est-à-dire un composant invoquant explicitement un service d'un autre composant sans connecteur, ne permet ni une modification aisée de la configuration, ni une réutilisation aisée des composants. Comme nous l'avons vu dans le chapitre I, la modélisation avec des composants et des connecteurs permet plus de souplesse que la simple utilisation d'objets.

interface

Tous les composants sont munis d'un port *cs* qui permet d'inhiber le comportement du composant. Le port *cs* des modules est automatiquement relié aux ports *cs* des composants contenus dans le module.

signal

Un signal s est une suite de valeurs pour chacune desquelles une estampille temporelle est associée. Cette estampille, représentée par un nombre réel, correspond à un temps physique t_i, date à laquelle la valeur a été émise sur le signal, $s=(e_i)_{i \in N}$, $ei = (v_i, t_i) \in$ Value x $\Re$. L'ensemble des signaux est noté $\mathcal{S}$. Chaque couple (v_i, t_i) est appelé événement.

événement

signaux synchrones

Deux signaux s_1 et s_2 sont dits synchrones si et seulement si :

$\forall i \in \mathcal{N}$, $[(v_i, t_i) \in s_1 \Rightarrow \exists v \in$ *Value*, $(v, t_i) \in s_2]$ et $[(v_i, t_i) \in s_2 \Rightarrow \exists v' \in$ *Value*, $(v', t_i) \in s_1]$.

C'est-à-dire que pour tout événement de date t_i d'un signal, un événement avec la même date appartient aussi à l'autre signal.

On définit alors une valuation d'un signal s comme une application $\mathcal{F}_s$ qui à tout instant physique $t \in \Re$ fait correspondre la valeur du signal à cet instant. Cette valeur correspond à la dernière valeur émise sur le signal à un instant inférieur ou égal à t, c'est-à-dire :

$\mathcal{F}_s : \Re \rightarrow$ Value,

$\mathcal{F}_s(t) = v$ tel que si $\exists i \in \mathcal{N}$, $(v, t_i) \in s$, $t_i \leq t$ et $[(v_{i+1}, t_{i+1}) \in s \Rightarrow t_{i+1} > t]$;

$\forall t<t_0 \in \Re$, $\mathcal{F}_s(t) = \perp$.

Remarques :

- *Value* est la classe générique de représentation des données discrètes échangées entre composants. L'ensemble des valeurs qui peuvent être représentées n'est pas fixé, il peut être élargi par l'utilisateur en définissant de nouvelles classes qui héritent de *Value*.
- Les événements sont ordonnés par rapport au temps physique qui correspond à l'instant de leur occurrence, c'est-à-dire que $\forall\ j>i,\ t_j > t_i$.
- La valuation d'un signal à un instant donné est unique : Si $(v_i, t_i), (v_j, t_j) \in s$ alors $t_i=t_j \Rightarrow v_i=v_j$.
- La *séquence minimale* $((v_i, t_i))_{i \in \mathbb{N}}$ de représentation d'un signal s est telle que deux événements consécutifs ont des valeurs différentes : $\forall\ i \in \mathbb{N}$ si $(v_i, t_i), (v_{i+1}, t_{i+1}) \in s$ alors $v_i \neq v_{i+1}$.
- Pour deux signaux quelconques, les instants d'occurrence des événements ne correspondent pas nécessairement. C'est-à-dire que le modèle utilisé est un modèle à événements discrets mais les systèmes modélisés ne sont donc pas obligatoirement des systèmes à temps discret[4].

service

Les composants élémentaires encapsulent un ensemble de services[5]. Ces services sont modélisés par des méthodes. Chaque service émet des valeurs vers un seul port de sortie, ces valeurs dépendent d'un certain nombre (éventuellement aucun) d'arguments fournis par d'autres ports. Un service à n entrées est donc défini comme une fonction de $\mathcal{S}^n$ dans $\mathcal{S}$ qui aux n signaux d'entrée e_i associe un signal de sortie o. Plusieurs services d'un même composant peuvent émettre des événements sur un même port de sortie. C'est pourquoi les services sont munis d'une priorité de déclenchement dans un composant. Cette priorité relative entre services d'un même composant permet de résoudre les conflits d'écriture simultanée sur un même port, ainsi que les conflits lecture-écriture. Dans les cas où les priorités sont identiques, on utilise une fonction de résolution.

priorité

Soit la fonction *Priorité* de $\mathcal{S}$ dans $\mathbb{Z}$ qui a un signal associe un entier relatif. On dira que s1 est prioritaire sur s2, si *Priorité*(s1)>*Priorité*(s2).

fusion

Le signal correspondant à un port de sortie est le résultat de la fusion des signaux émis par les services sur ce port. La fusion $Fusion_G(s1,s2)$ de deux signaux $s1 = (e_i)_{i\in\mathbb{N}}$ et $s2 = (e_j)_{j\in\mathbb{N}}$ par rapport à une fonction de résolution G est un signal $o = (e_k)_{k\in\mathbb{N}}$ défini par :

- si $(\mathcal{F}_{s1}(t),t) \in s1$ et $(\mathcal{F}_{s2}(t),t) \notin s2$ alors $\mathcal{F}_o(t) = \mathcal{F}_{s1}(t)$
- si $(\mathcal{F}_{s1}(t),t) \notin s1$ et $(\mathcal{F}_{s2}(t),t) \in s2$ alors $\mathcal{F}_o(t) = F_{s2}(t)$
- si $(\mathcal{F}_{s1}(t),t) \in s1$ et $(\mathcal{F}_{s2}(t),t) \in s2$ et
  - si *Priorité*(s1)=*Priorité*(s2) alors $\mathcal{F}_o(t) = G(\mathcal{F}_{s1}(t), \mathcal{F}_{s2}(t))$
  - si *Priorité*(s1)>*Priorité*(s2) alors $\mathcal{F}_o(t) = \mathcal{F}_{s1}(t)$
  - si *Priorité*(s1)<*Priorité*(s2) alors $\mathcal{F}_o(t) = \mathcal{F}_{s2}(t)$

Il existe dans SEP quatre familles de conditions (modes) de déclenchement d'un service, qui définissent quatre types de services (cf. section 2-). Lorsqu'une condition de déclenchement est remplie la méthode correspondant au service associé est exécutée de façon atomique. Le comportement d'un composant avec n entrées et m sorties est donc défini par

[4] La définition de système à temps discret utilisée ici correspond à la définition utilisée dans Ptolemy **[CHE96]**.
[5] La section 3- montre que les modules aussi peuvent encapsuler des services.

l'ensemble des déclenchements de chaque service qu'il encapsule, c'est-à-dire comme une fonction de $\mathcal{S}^n$ dans $\mathcal{S}^m$ qui aux n signaux d'entrée fait correspondre m signaux de sortie.

2- Le comportement des composants élémentaires.

Tous les services d'un composant sont déclenchés suivant le même mode. Il existe donc quatre classes de composants élémentaires : *sep.model.EdgeComponent*, *sep.model.LevelComponent*, *sep.model.TimeComponent*, *sep.model.InputComponent*[6].

En fait, il y a seulement deux catégories élémentaires de composants. Les composants combinatoires dont les sorties ne dépendent que des entrées : *LevelComponent*. Et les composants séquentiels qui peuvent être modélisés par des machines à états et dont les sorties dépendent de l'état courant du composant et de ses entrées (modèle de Mealy) : *EdgeComponent*. L'état suivant est calculé en fonction de l'état courant et des entrées. L'exécution d'un service séquentiel peut être de nature différente, il existe alors deux sous-familles : *InputComponent* et *TimeComponent*.

2-1. Les composants combinatoires.

La classe *LevelComponent* modélise les composants combinatoires. Elle est donc constituée uniquement de services combinatoires. Chaque service correspond à une fonction H de *Value* ^{n} dans *Value* qui dépend des n entrées s_j et produit le signal s_o tel que $\forall t \in \Re$, $\mathcal{F}_{so}(t) = H(\mathcal{F}_{s1}(t), \mathcal{F}_{s2}(t), \ldots, \mathcal{F}_{sn}(t))$. Une séquence d'événements définie ainsi est potentiellement infinie ; pour la rendre dénombrable, on ajoute la proposition suivante : un événement $(v_k, t_k) \in s_o$ ssi $\exists\ 0<i<n, j \in \mathcal{N}$ tels que $(v_j, t_k) \in s_i$. Afin de construire le signal de sortie, il faut échantillonner la fonction $\mathcal{F}_{so}(t)$ aux instants d'occurrence d'un événement sur au moins un signal d'entrée.

C'est-à-dire qu'on ne représente dans la séquence du signal résultant que les événements dont l'estampille temporelle correspond à au moins un événement j d'au moins une entrée s_i du composant. En pratique, l'exécution d'un service combinatoire est déclenchée par la variation d'au moins une de ses entrées.

Le service *cs* des composants *LevelComponent* est combinatoire. En pratique il simule la mise en haute impédance des entrées du composant combinatoire, c'est-à-dire que la valeur $\perp$ de type *LevelValue* est présente sur les ports d'entrée.

2-2. Les composants séquentiels.

signal de commande

Soit $c = ((v_i, t_i))_{i \in \mathcal{N}}$, un signal tel que $v_i = \perp$ lorsque i est pair, et $v_i \neq \perp$ lorsque i est impair. Un tel signal est appelé signal de commande. L'ensemble des signaux de commande appelé $\mathcal{C}$ est un sous-ensemble de $\mathcal{S}$. Un signal de commande n'est pas nécessairement régulier, c'est-à-dire que $t_{i+1} - t_i$ n'est pas nécessairement constant.

La classe *EdgeComponent* modélise les composants qui correspondent à des machines d'états. Les sorties dépendent de l'état et des entrées. Le nouvel état est alors calculé en fonction des entrées et de l'état présent. Un service séquentiel est déclenché par un signal de commande c. Le service séquentiel sur le vecteur[7] d'entrée s, commandé par un signal c et dont l'état est représenté par le vecteur e, a une sortie s_o telle que :

---

[6] Sur la Figure 2, les classes *EdgeComponent*, *LevelComponent*, *TimeComponent* et *InputComponent* sont représentées pour simplifier par *Edge*, *Level*, *Time*, *Input*.

[7] s est un signal dont les valeurs sont des n-uplets de *Value* ^{n}.

si $\mathcal{F}c(t_0) = \perp$, $\mathcal{F}_{so}(t_0) = \perp$
$\forall\ k>0 \in \mathcal{N}$, si $\mathcal{F}c(t_k) = \perp$, $\mathcal{F}_{so}(t_k) = \mathcal{F}_{so}(t_{k-1})$

$\forall\ k \in \mathcal{N}$, si $\mathcal{F}_c(t_k) \neq \perp$, $\mathcal{F}_{so}(t_k) = H(\mathcal{F}_s(t_k - \varepsilon), \mathcal{F}_e(t_k - \varepsilon))$
où ε est tel que : $\forall\ t \in \Re$, $t_k - \varepsilon \leq t < t_k \Rightarrow \mathcal{F}_s(t_k - \varepsilon)=\mathcal{F}_s(t)$, $\mathcal{F}_e(t_k - \varepsilon)=\mathcal{F}_e(t)$.

Intuitivement cela signifie que la valeur de sortie est maintenue constante tant que la commande vaut $\perp$ et prend une valeur fonction des entrées et de l'état courant lorsque la commande ne vaut plus $\perp$. On remarque que le signal de sortie est synchrone avec le signal de commande, mais pas avec les signaux d'entrée. Les valeurs d'entrée sont échantillonnées à une date strictement antérieure à la date de réaction.

Soit *LevelValue*= {LOW, HIGH, $\perp$} l'ensemble représentant les valeurs de niveau et modélisé par la classe *sep.type.LevelValue*.

*LevelValue** est alors un sous-ensemble de S tel que tous les événements d'un signal de *LevelValue** appartiennent à *LevelValue* x $\Re$.

On définit alors la fonction de commande *posedge* qui, a un signal e de *LevelValue**, associe un signal de commande c :

posedge : *LevelValue** $\rightarrow$ C telle que

$\mathcal{F}_c(t)$ = HIGH si et seulement si $\exists\ \varepsilon > 0 \in \Re$, $\forall\ t^+, t^- \in \Re$,
$t - \varepsilon < t^- < t \Rightarrow \mathcal{F}_e(t^-)$=LOW et $t \leq t^+ < t + \varepsilon \Rightarrow \mathcal{F}_e(t^+)$=HIGH.
$\mathcal{F}_c(t) = \perp$ sinon.

Intuitivement, la fonction *posedge* détermine les fronts montants dans un signal. On peut définir de manière symétrique une fonction *negedge* qui détermine les fronts descendants dans un signal (cf. Figure 3). La classe *sep.model.EdgeComponent* permet de définir des composants dont les services sont séquentiels avec soit une fonction de commande de type *posedge* soit une commande de type *negedge*. Avant chaque exécution d'un service séquentiel, le statut du port *cs* est évalué afin de déterminer si le service doit ou ne doit pas être rendu, *cs* est un service synchrone avec toutes les commandes de tous les composants séquentiels.

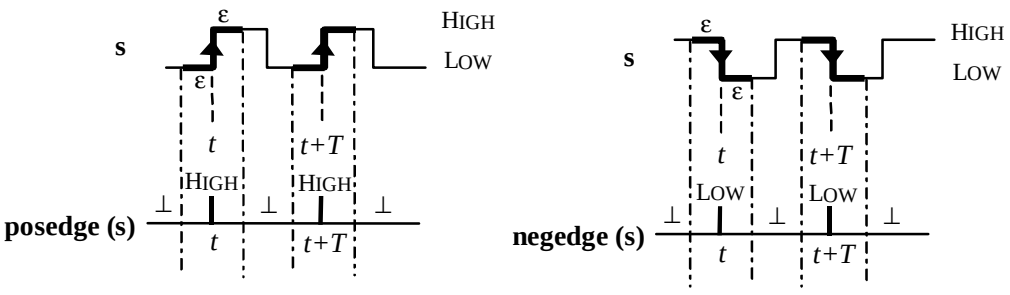

Figure 3 – Fonctions posedge et negedge.

On définit aussi la classe des fonctions $Time_p$ de $\Re$ dans C avec p $\in \Re$ qui, à un réel t, associe un signal de commande o :

$Time_p : \Re \rightarrow C$ telle que $F_o(t)=t$ si et seulement si $\exists\ i \in \mathcal{N}$, $t = i*p$, $F_o(t)=\perp$ sinon.

La classe *sep.model.TimeComponent* permet de modéliser des composants dont les services sont séquentiels avec une fonction de commande de la classe $Time_p$. Ces composants permettent de modéliser les comportements périodiques.

Enfin, la classe *sep.model.InputComponent* permet de modéliser les composants séquentiels dont la fonction de commande dépend des événements émis par l'environnement. Par exemple, la fonction de commande est évaluée à 1 seulement pour les instants où un événement double-clic a été émis sur la représentation graphique du composant correspondant depuis l'occurrence précédente d'un événement. Les événements émis par un composant *InputComponent* sont datés du temps de simulation courant lors du traitement du service concerné. Attention, la machine virtuelle de gestion de l'interface graphique n'étant pas réactive, il se peut que des événements émis par l'environnement soient perdus.

On peut ajouter des modes de déclenchement en définissant de nouvelles fonctions de commande. Par exemple, on peut définir un mode pour déclencher un service séquentiel uniquement si une valeur est présente sur le port pendant une durée (exprimée par rapport à un temps physique) prédéfinie. Dans ce document, nous ne présentons que les modes de déclenchement que nous avons été amenés à utiliser dans nos exemples.

2-3. Les services avec retard.

retard

Lors de la spécification d'un service, le concepteur peut préciser un temps d'exécution du service. Ce retard est exprimé en nanosecondes, il peut être spécifié aussi bien pour les services séquentiels que combinatoires. Un retard ne modifie pas la date d'exécution d'un service, mais seulement la date à laquelle l'événement de sortie sera produit. Par exemple, le comportement d'un service combinatoire avec un retard δ sera $\forall t \in \Re$, $\mathcal{F}_{so}(t + \delta) = H(\mathcal{F}_{s1}(t), \mathcal{F}_{s2}(t), \ldots, \mathcal{F}_{sn}(t))$.

3- Le comportement des modules.

Dans SEP, il est possible de construire un comportement complexe par composition du comportement de composants plus simples. La définition d'une configuration de composants et de connecteurs permet de construire ce comportement. Les connecteurs élémentaires de SEP sont les signaux de contrôle et les bus de données.

Les signaux de contrôle permettent à un et un seul port émetteur de diffuser des données vers plusieurs ports récepteurs. Ce sont des équipotentielles, qui associent aux ports récepteurs le signal du port émetteur.

Les bus de données permettent à plusieurs ports de partager des données. Les bus sont bidirectionnels, tous les ports connectés reçoivent et peuvent émettre des données sur un bus. Le signal associé au bus est la fusion des signaux émetteurs.

Remarques :

- La fusion est instantanée, il n'y a pas de décalage dans le temps (voir la définition).
- Il est donc possible que la valeur émise et la valeur présente sur un même port connecté à un bus à un instant $t \in \Re$ soient différentes.

L'introduction de la notion de module permet de réutiliser des parties de modèles existants et permet une approche incrémentale. Cependant, il est possible de définir pour toute

configuration qui contient des modules un comportement équivalent sans module. Pour cela, il suffit de fusionner les signaux d'entrée et de sortie des ports de modules (cf. Figure 4).

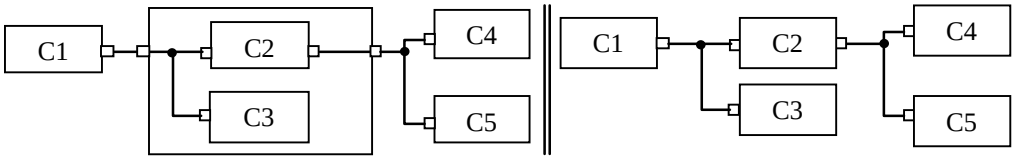

Figure 4 – Comportement équivalent en éliminant les modules d'une configuration.

Les composants élémentaires fournissent des services. Leur composition consiste à créer des services de plus haut niveau. La notion de service est donc aussi présente au niveau des modules, c'est pourquoi un mécanisme qui la fait apparaître explicitement a été mis en œuvre. Ce mécanisme permet la définition de services dans un module, améliore la lisibilité et donc la possibilité de réutilisation.

En effet, le manque de lisibilité vient en partie du grand nombre de ports et de signaux dans une architecture de taille industrielle. La plupart de ces ports permettent l'implémentation de protocoles de communication et de synchronisation, voire permettent la compatibilité avec d'anciens systèmes. De plus, dans le cadre d'une modélisation hiérarchique d'une architecture matérielle, tous les ports de contrôle sont reportés à travers tous les niveaux de modélisation. L'information est tellement peu intéressante au niveau de la compréhension de la fonction réalisée que les schéma-blocs ne font en général pas apparaître cette information et la remplacent par une bulle abstraite nommée 'glue logique'.

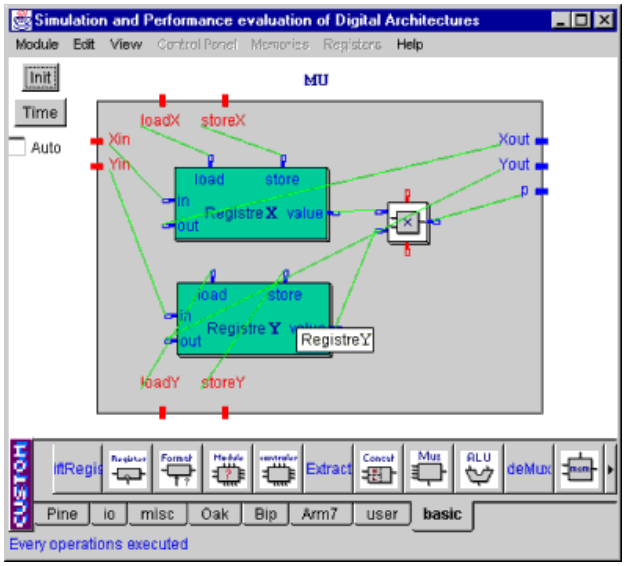

Figure 5 – Modèle de la MU dans SEP (1).

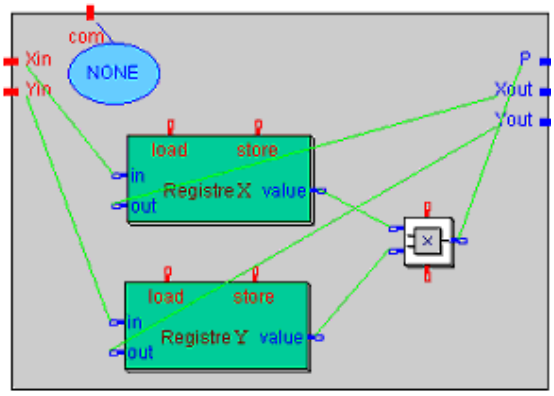

Figure 6 – Modèle de la MU dans SEP (2).

Dans le cadre de l'évaluation de performances, il est possible d'agréger un certain nombre de ces signaux de contrôle et de modéliser un service de communication de plus haut niveau. Evidemment, l'architecture ainsi réalisée n'est pas équivalente en terme de consommation ou de surface de silicium. Le mécanisme proposé consiste à agréger dans un module les signaux de contrôle utilisés pour la réalisation d'un objectif (service) commun.

On peut par exemple imaginer le cas de la modélisation de l'unité de multiplication (MU) du processeur Oak™ de 'DSP group'[8]. Avec un modèle extrêmement simplifié, la MU est constituée de deux registres load/store X et Y et d'un multiplieur 16x16.

[8] Le processeur Oak est présenté au chapitre XI.

Un registre load/store possède une entrée *in*, deux sorties *value* et *out*, et deux services *load* et *store*. Le service *store* charge la valeur d'entrée *in* et la mémorise, la valeur mémorisée est disponible sur la sortie *value*. Le service *load* émet, sur la sortie *out*, la valeur mémorisée.

Les manipulations autorisées sur la MU sont le chargement sur un bus ou la mémorisation à partir d'un bus de chacun des deux registres X et Y ; c'est-à-dire a priori quatre commandes *loadX*, *storeX*, *loadY*, *storeY*. Le multiplieur multiplie les valeurs mémorisées dans les deux registres et émet leur produit sur le port de sortie p. La Figure 5 montre un module correspondant à cette architecture et tel qu'il aurait été modélisé en VHDL. Dans ce module, les chemins de données sont essentiels et peuvent donner lieu à plusieurs configurations qui ne sont pas équivalentes, alors que la connaissance de l'existence de chemins de contrôle est suffisante à la compréhension. C'est ainsi que SEP permet la création du module présenté par la Figure 6. Le port *com* est appelé port de service et agrège les chemins de contrôle du module. Pour l'extérieur la MU est alors un composant qui offre quatre services concurrents *loadX*, *storeX*, *loadY*, *storeY* et n'est plus seulement une liste de ports, la boîte noire devient grise. La validité de l'exécution simultanée de ces quatre services dépend de la configuration à laquelle appartient la MU, ce problème est abordé dans les chapitres VII et VIII. En particulier, le chapitre VII reprend ce composant est en donne un modèle complet.

Par élimination des modules, étant donné qu'un registre load/store est composé de deux registres standards, on obtient la configuration équivalente présentée à la Figure 7.

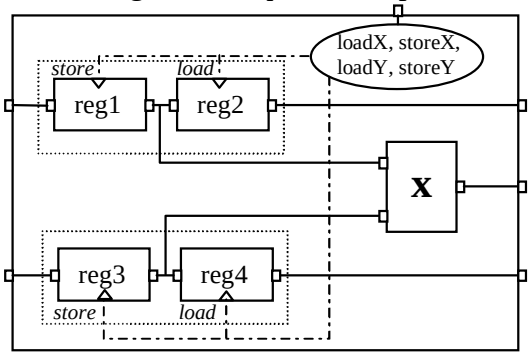

Figure 7 – MU sans module.

L'utilisation d'un port de service revient à modifier individuellement les fonctions de commande des ports de chargement des registres (i.e. du registre *reg1*) afin qu'un événement, dont la valeur contient une chaîne de caractères représentant le service adapté (i.e. "storeX"), soit considéré comme déclencheur du service de chargement du registre.

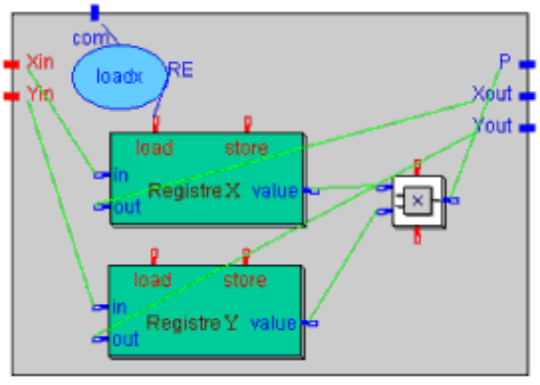

Figure 8 – Traitement du service *'loadX'*.

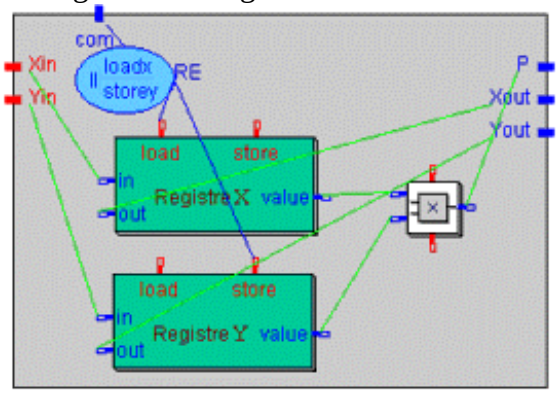

Figure 9 – Traitement du service *'loadX || storeY'*.

En simulation, un composant désirant utiliser un service du MU (i.e. loadX) devra simplement émettre la chaîne de caractères "loadX" sur un signal en direction du port *com* du MU. Le port de service se comporte comme un constructeur dynamique de chemins de contrôle (cf. Figure 8). De même, on peut appeler la composition parallèle des services *loadX* et *storeY* en émettant la chaîne de caractères "loadX || storeY" (cf. Figure 9)[9].

4- Héritage de comportement.

Que la description du comportement des composants soit logicielle ou structurelle, tous les composants sont potentiellement constitués d'un ensemble de services. On définit une relation d'héritage du comportement d'un composant fils c vers son composant père *p* comme d'une part, l'héritage par le fils de l'interface du composant père et d'autre part, l'héritage des services définis par le père. C'est-à-dire que le composant *c* possède dans son interface au minimum les mêmes ports que son père ; bien évidemment, il peut en déclarer de nouveaux ; de plus, il hérite des services définis par *p* et peut les redéfinir.

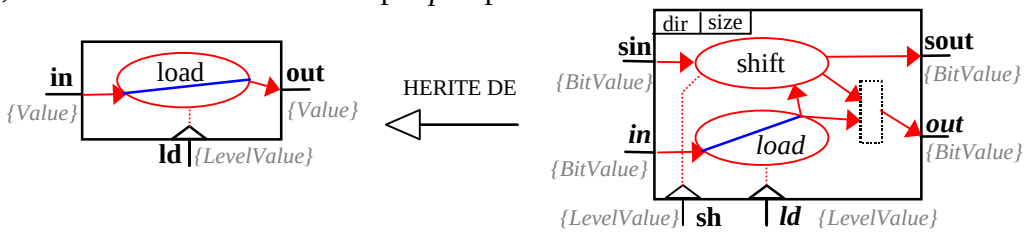

Figure 10 – Registres simple et à décalage.

Prenons l'exemple d'un registre. Dans SEP, les données manipulées par les registres n'ont pas nécessairement une taille constante, ils manipulent des données de type *Value* et peuvent donc contenir aussi bien des bits, que des entiers, des nombres réels ou des instructions. Les registres ont un service séquentiel *load* qui émet sur le port de sortie *out* la donnée présente sur le port d'entrée *in* (cf. Figure 10).

A partir de ce comportement, on peut définir le comportement d'un registre à décalage qui hérite du comportement du registre. Les registres à décalage nécessitent la notion de bit, ils ne peuvent donc manipuler que des données de type *BitValue* (ensemble ordonné de bits). De plus, il peut y avoir deux sens de décalage, on ajoute donc le paramètre booléen *dir*. On ajoute aussi le paramètre entier *size*, qui est nécessaire pour effectuer un décalage. L'entrée série *sin* fournit la donnée qui servira, lors du décalage, à remplacer les bits décalés. La sortie série *sout* contient les bits qui ont été éliminés lors du dernier décalage. Enfin, un nouveau service séquentiel *shift* est défini. Ce service décale la dernière donnée mémorisée et émet le résultat sur la sortie *out*. Le déclenchement simultané des services *shift* et *load* est théoriquement possible, c'est pourquoi le service *load* est défini avec une priorité supérieure à celle du service *shift*.

Attention, ce n'est pas du sous-typage d'interface. En particulier, on ne peut pas remplacer dans une configuration un registre par un registre à décalage. Il ne semble pas que cela ait beaucoup de sens dans le cadre d'une architecture matérielle. Par contre, le comportement est hérité, ce qui réduit la quantité de code à écrire et à valider.

Cette relation d'héritage, nous permet notamment de redéfinir avec une description logicielle les services construits avec une description structurelle. Ainsi, on peut

[9] RE (Rising-edge) est une valeur particulière utilisée pour représenter l'occurrence d'un front montant sur un signal.

Enfin, une deuxième notion d'héritage a été définie, c'est l'héritage des services d'un composant fils par son composant père, c'est-à-dire par le module auquel il appartient. Les services du fils peuvent être directement invoqués sur le père comme s'il les avait définis lui-même. Cette forme d'héritage est illustrée de façon plus complète dans le chapitre VII. Ainsi, le registre *load/store* présenté à la section précédente est composé de deux registres élémentaires *reg1* et *reg2* (cf. Figure 7). Le composant *load/store* peut hériter du service *load* du registre *reg2* et définir son service *store* comme l'appel du service *load* du registre *reg1*.

Remarquons, que l'héritage simultané des services *load* des deux registres est légal et conduit à la définition d'un service *load* dont la signification est le résultat de la composition parallèle des deux services élémentaires, c'est-à-dire à leur exécution synchrone. Ce n'est évidemment pas ce que nous cherchons à réaliser dans ce cas.

L'environnement construit grâce au modèle générique n'est pas spécifique d'un type d'architectures, il permet de modéliser des architectures câblées (co-processeurs), mais aussi des architectures de type DSP, RISC, CISC, VLIW, des architectures micro-programmées voire des ASIC. L'utilisation d'un langage orienté objet pour la modélisation des composants nous a permis de définir une relation d'héritage entre composants. On s'aperçoit alors que cette relation d'héritage peut même avoir du sens pour des descriptions structurelles à partir du moment où la notion de service de module est définie.

La description qui est faite dans ce chapitre concerne les résultats que l'on veut obtenir lors de la description de composants et de modules. Le chapitre V s'intéresse au moteur de simulation et aux techniques mises en œuvre pour implémenter ce comportement.

Enfin, on peut remarquer que l'exécution des services est supposée atomique. Cela permet de modéliser les appels synchrones de services. Aucune vérification n'est faite sur la nature du code Java qui décrit le comportement. En particulier, il est nécessaire que la durée réelle d'exécution d'une méthode soit finie et bornée, et même qu'elle soit suffisamment petite pour que la simulation ne soit pas pénalisée. Dans les cas où l'on veut modéliser les comportements longs ou des communications non bornées le concepteur peut avoir besoin d'utiliser un mécanisme multi-tâches (réseaux à délais de communication non bornés, composants de traitement intensif). Avec le modèle proposé, l'utilisateur peut par exemple simuler une exécution asynchrone d'un service avec le mécanisme de *Thread* de Java, comme par exemple pour la gestion des tâches Esterel dans le chapitre IX. Si on utilise un tel mécanisme, on introduit un indéterminisme dans le comportement des composants qui dépend de l'ordonnanceur de tâches de Java. En particulier, aucun mécanisme de gestion de deadlock n'est mis en place.

Pour ne pas utiliser de mécanisme multi-tâches, on peut parfois décomposer le service, en plusieurs sous services bornés et modéliser les contraintes de communication dans les connecteurs, le comportement modélisé est alors déterministe. Les systèmes à communication non bornée n'entrent pas dans les objectifs de SEP.