Chapitre III- Un méta-modèle des composants SEP.

Objectifs du chapitre :

- Présenter le méta-modèle des composants SEP.
- Montrer comment il permet de générer des composants à partir d'une spécification en Java.

La description du comportement d'un composant (*ActiveComponent*) commence par la définition des services fournis par le composant. Ces services sont décrits par les méthodes d'une classe appelée *ServiceProvider* associée à ce composant. Ensuite, chaque service est encapsulé dans un composant SEP afin de réaliser un comportement. En effet, le service constitue seulement une partie du comportement, il est exécuté en concurrence avec d'autres services du même composant ou d'autres composants. Le chapitre précédent introduit la sémantique des composants et des connecteurs, il montre comment les composants communiquent entre eux et comment l'exécution des différents services est déclenchée. En particulier, il faut modéliser pour chaque service ses contraintes d'exécution, c'est-à-dire sa priorité et son mode de déclenchement. Dans SEP, un mécanisme semi-automatique permet cette composition de services pour constituer un composant. Un tel mécanisme nécessite la possibilité de manipuler dynamiquement, à partir de Java, les notions présentées dans le chapitre précédent. C'est pourquoi, nous avons construit un modèle du fonctionnement fondamental des composants SEP, c'est-à-dire un méta-composant. Ce méta-composant nous permet de construire dynamiquement de nouveaux composants, c'est-à-dire de leur ajouter et de modifier les attributs des services (priorité, mode de déclenchement) et de choisir les ports de l'interface qui participeront à la réalisation du service ; il faut choisir non seulement les ports auxquels parviendront les signaux de commande, mais aussi les ports dont les signaux fournissent les paramètres des services. Ce mécanisme est assisté par une interface graphique adaptée.

méta-composant

Une fois le méta-composant construit, l'environnement génère et archive la classe Java correspondant au composant. Cette classe pourra alors être instanciée et le composant ainsi créé sera composé pour réaliser un module.

Ce chapitre explique ce mécanisme et l'illustre par un exemple simple.

1- Le modèle des méta-composants.

Le mécanisme proposé lit l'information contenue dans un *ServiceProvider*, puis guide le concepteur dans la construction du comportement ; on construit ainsi un méta-composant (*sep.model.meta.MaterialComponent*) dont la structure statique est illustrée en UML par la Figure 1. A partir de ce méta-composant, un programme produit automatiquement une classe Java qui représente le composant SEP avec les propriétés définies par le méta-composant. Cela repose sur le mécanisme d'introspection[1] fourni par Java **[Gos&co00]**. Toutes les classes qui

[1] L'introspection est le mécanisme inverse de la réification, il permet de déterminer à partir d'un objet l'information générique de la classe qui l'a engendrée (signature des méthodes et des attributs, hiérarchie d'héritage).

permettent d'implémenter ce mécanisme sont dans le paquetage *sep.model.meta*. En particulier, c'est la classe *ServiceIntrospector* qui permet d'introspecter un *ServiceProvider* à la recherche des méthodes qui seront considérées pour la gestion des paramètres et des méthodes pour la gestion des services.

Pour un composant, un paramètre est une valeur typée qui peut être lue ou écrite afin de personnaliser le comportement d'un composant. Le chapitre II prend l'exemple du registre à décalage qui a un premier paramètre entier *size* qui représente sa taille en octets et un second paramètre booléen *dir* qui représente le sens de décalage (décalage à gauche ou à droite). Pour chaque paramètre, le concepteur doit écrire une méthode de lecture et une méthode d'écriture. Afin qu'on puisse facilement faire la différence entre une méthode de gestion des paramètres (lecture ou écriture) et une méthode de gestion d'un service, le nom d'une méthode de gestion de paramètre doit commencer par *"get"* (respectivement *"set"*) pour les méthodes de lecture (respectivement écriture) et se finir par le nom du paramètre. Par exemple, pour le registre à décalage le concepteur doit écrire les quatre méthodes (*setSize*, *getSize*, *setDir*, *getDir*). Les méthodes de lecture retournent lors de leur invocation la valeur unique correspondant au paramètre concerné ; on passe en argument des méthodes d'écriture une valeur qui remplacera la valeur courante des paramètres.

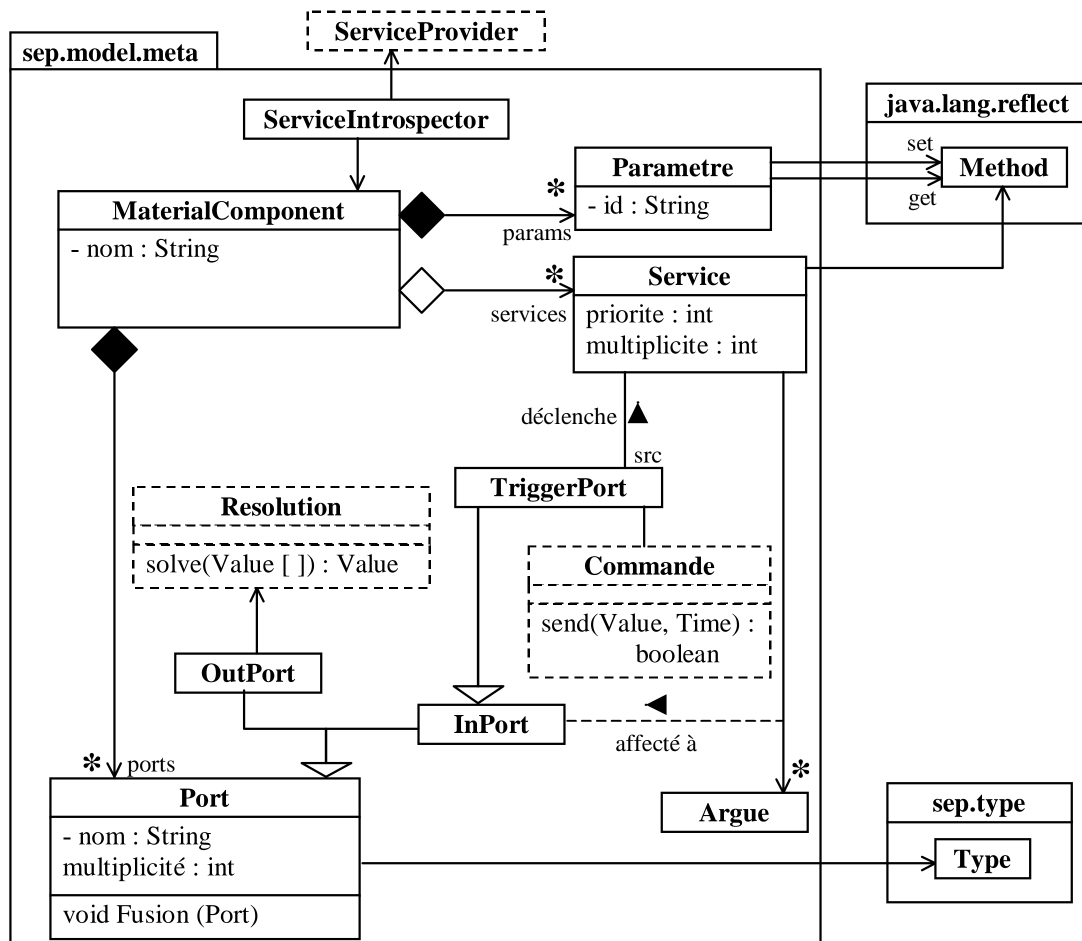

Figure 1 – modèle statique UML d'un méta-composant SEP.

Le travail du *ServiceIntrospector* consiste alors à parcourir un *ServiceProvider* à la recherche des méthodes qui commencent par *"set"* ou *"get"*, il vérifie qu'elles fonctionnent par paires – chaque méthode de lecture doit avoir une méthode d'écriture associée –, il

s'assure que les signatures sont compatibles entre elles – c'est-à-dire que les méthodes de lecture retournent une valeur dont le type est un sous-type[2] du paramètre unique de la méthode d'écriture –, enfin il gère une liste des paramètres potentiels que le concepteur pourra ajouter au composant en cours de définition.

2- La modélisation des services.

Pour les méthodes de service il faut décrire deux aspects. D'une part, il faut décrire le contrôle, c'est-à-dire les modes de déclenchement et les ports qui interviennent. D'autre part, il faut décrire les chemins de données, c'est-à-dire les ports qui fourniront les données nécessaires à l'exécution du service.

La première étape liée à la partie de contrôle d'un service est de choisir la multiplicité du service. C'est-à-dire le nombre d'appels simultanés de ce service. La multiplicité d'un service peut être fixée pour un composant ou être laissée en paramètre, mais elle doit être connue avant la simulation. Cette contrainte, permet de faire des validations sur les services de modules et sur le jeu d'instructions (cf. chapitre VII). Très restrictive pour les architectures logicielles, cette contrainte ne l'est pas pour les architectures matérielles, en effet il ne s'agit pas de modifier l'architecture dynamiquement pendant la simulation d'une application.

La deuxième étape consiste à choisir le mode de déclenchement du service. Pour cela, il faut choisir un ou plusieurs ports de déclenchement *TriggerPort* et leur affecter une fonction de commande en implémentant la méthode *boolean send (Value, Time)* de la classe abstraite *Commande*. Cette méthode est évaluée à vrai lorsque l'événement passé en argument (Value, Time) doit déclencher l'exécution du service. Cette méthode représente la fonction de commande définie au chapitre II, son résultat dépend éventuellement des événements précédents. Les fonctions de commande (*posedge*, *negedge*, *level*, $Time_p$, *input*) sont prédéfinies. Il est aussi possible de déclencher un service à chaque fois qu'un autre service, appelé service de contrôle, est exécuté. Il faut alors noter qu'un service de contrôle est toujours exécuté avant ses services contrôlés, l'ordre relatif d'exécution des services contrôlés doit être donné.

Enfin, pour les composants séquentiels, il peut être nécessaire de choisir la priorité relative du service par rapport aux autres afin de pouvoir résoudre les conflits éventuels.

Il faut aussi décrire les chemins de données. Les services sont associés à des méthodes, toutes les méthodes publiques[3] d'un *ServiceProvider* qui ne sont pas des méthodes de gestion de paramètres sont potentiellement des méthodes de service pour un composant. Une méthode de service retourne au plus une valeur à chaque invocation. Dans ce cas, il faut choisir un port de sortie du composant *OutPort* qui portera un signal constitué des valeurs retournées par la méthode lors de ses invocations successives. Si plusieurs services fournissent leur résultat sur le même port, le conflit écriture-écriture est résolu soit grâce à la priorité relative des services, soit grâce à une fonction de résolution. Pour définir une fonction de résolution, l'utilisateur doit implémenter la méthode *Value solve (Value [ ])* de la classe abstraite *Resolution*. Cette méthode doit calculer une valeur de retour en fonction de plusieurs valeurs en conflit passées en argument.

Un service peut utiliser des arguments (*Argue*). Ces arguments sont fournis par des ports d'entrée *InPort*. Un port d'entrée peut être partagé par plusieurs services, voire par plusieurs instances d'un même service. Dans le cas des services combinatoires, les ports d'entrée sont

---

[2] Voir le chapitre VI pour les notions de type et de sous-type.

[3] Les méthodes de service doivent nécessairement être publiques car aucune supposition n'est faite sur le lien qui peut exister entre un composant et son *ServiceProvider* associé. Ils n'appartiennent pas nécessairement au même paquetage et il n'existe pas nécessairement une relation d'héritage qui les lie.

des ports de déclenchement. Dans le cas des services séquentiels, on utilise la priorité relative des services pour résoudre les conflits éventuels lecture-écriture.

3- Exemple : Le ServiceProvider Memory.

Afin d'illustrer ces propos, considérons un exemple basé sur le *ServiceProvider Memory* décrit en Java de la façon suivante :

```
import sep.type.*;

public class Memory implements sep.model.ServiceProvider {

// ----- Début de la description des services -----
        public Value read(int address) {
                return address<base?LevelValue.Undefined:b.read(address-base);
        }

        public void write(int address, Value v) {
                if (address>=base)
                        b.load(v, address-base);
        }
// ----- Fin de la description des services -----

// ----- Gestion des paramètres -----
        public void setSize(int size) { b.setSize(size); }
        public int getSize() { return b.getSize(); }
        public int getBaseAddress() { return base; }
        public void setBaseAddress(int newBase) { base = base; }
        public void setFile(java.io.File f) { /* lit le contenu de la mémoire depuis un fichier */ }
        public java.io.File getFile() {  /* Ecrite le contenu courant de la mémoire dans un fichier */ }

        private int base = 0;
        private sep.model.Bank b = new sep.model.Bank();  // Ensemble indicé de sep.type.Value
}
```

A partir de cette classe, il est possible d'instancier un objet à partir duquel le *ServiceIntrospector* peut aisément fournir deux listes. Une liste des méthodes potentielles de service (*read*, *write*), une liste de paramètres dont dépend l'exécution des services (*size*, *baseAddress*, *file*). Cette classe permet aux utilisateurs de SEP de construire plusieurs composants différents suivant les configurations qu'ils choisissent (ROM, RAM, banc de registre, mémoire multi-ports) à partir d'une interface graphique (cf. Figure 2).

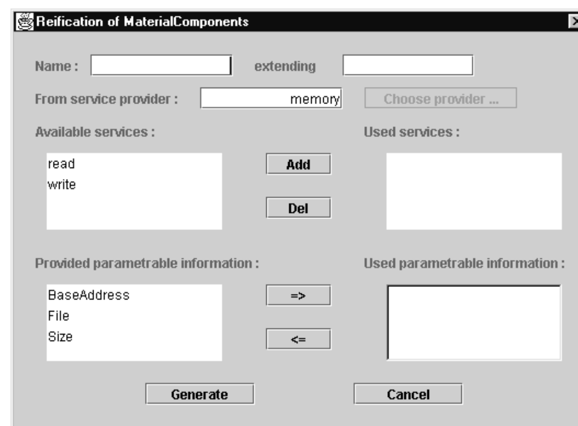

Figure 2 – Modélisation du comportement.

ROM

Par exemple, ce *ServiceProvider* permet de construire une mémoire de type ROM[4]. Un composant ROM a seulement un service de lecture. Ce service est un signal de commande *posedge* contrôlé par le port *read*. Il a pour rôle de lire la donnée mémorisée à l'adresse lue sur le port *adr*, puis il émet cette donnée sur le port *out* (cf. Figure 3). Enfin, ce composant a trois paramètres de personnalisation. On peut changer sa taille *size*, son adresse de base[5] *BaseAddress* et son contenu peut être lu dans un fichier *File*.

RAM

Une mémoire de type RAM[6] peut aussi être construite si le concepteur utilise le service *write*. Un composant RAM hérite son comportement du composant ROM, dans SEP il s'agit de remplir le champs *extends*. Ainsi le composant RAM possède le service *read* ainsi que les trois paramètres de personnalisation du composant ROM. De plus, il possède un service *write* déclenché par l'occurrence d'un front montant sur le port *write*. L'adresse d'écriture est aussi lue sur le port *adr* mais aucune donnée n'est émise. La donnée à écrire est lue à partir du port *in*. Les deux services *write* et *read* réagissent en concurrence, on utilise le mécanisme de priorité pour résoudre le conflit potentiel lecture-écriture. C'est ainsi, que le service *read* est doté d'une priorité supérieure à celle du service *write*. 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.

Figure 3 – Les chemins de contrôle et de données.

RegisterBank

Notre exemple suivant est la modélisation d'un banc de registres (*RegisterBank*). Un banc de registre permet de mémoriser *size* données de n bits. Il dispose d'une instance du service *write* qui permet d'écrire une donnée dans le banc de registre et de deux instances du service *read* qui permettent de lire simultanément deux valeurs mémorisées. La priorité est utilisée de la même façon que pour le composant RAM. Le seul paramètre utile est la taille *size*.

[4] Read Only Memory : mémoires où seules les lectures sont possibles (service *read*).

[5] Conceptuellement, les architectures peuvent avoir plusieurs blocs de mémoires, on sépare ainsi les différentes zones de données ou d'instructions. En pratique, il n'y a qu'un seul bloc de mémoire et les zones sont différenciées à partir de leur adresse de base et de leur taille (i.e. données X : 16 Ko, adresse de base = 0x0000 ; données Y : 128 Ko, adresse de base = 0x2000 ; instructions : 256 Ko, adresse de base = 0x4000 ; banc de registres : 16*32 bits, adresse de base = 0x8000).

[6] Random Access Memory : mémoire qui autorise aussi bien les lectures (service *read*) que les écritures (service *write*).

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

Avec la 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, dans ce cas, les conflits écriture-écriture sont résolus dynamiquement en simulation par l'émission d'un message d'avertissement. Il est aussi possible de résoudre de tels conflits par l'utilisation d'une fonction de résolution.

En conclusion, les méta-composants construits avec cette méthode permettent la génération d'une classe Java. Cette classe, une fois compilée, représente un composant SEP qui pourra être archivé et réutilisé dans un module. Cette méthode permet de réutiliser, par héritage, les services d'autres composants. De plus, les services d'un même ServiceProvider peuvent être réutilisés pour définir plusieurs composants.

Le type des paramètres des méthodes de service est utilisé pour inférer le type des données émises ou reçues sur un port. Ce type inféré permettra de valider les connexions (cf. chapitre VI).

Cette méthode a été implémentée dans l'environnement SEP. Le code généré pour les différents exemples présentés dans ce chapitre est disponible en annexes.