Chapitre VI- La validation de la composition.

Objectifs du chapitre :

- Expliquer les conséquences de l'utilisation de règles de typage souples dans SEP.
- Présenter le mécanisme de validation des connexions en inférant le type des ports et des connecteurs.
- Montrer les insuffisances du mécanisme de la liaison dynamique pour la résolution des opérateurs binaires en présence de sous-typage. Proposer un mécanisme de remplacement.

Le chapitre II montre comment le modèle générique nous permet de faire une modélisation structurelle de haut niveau en faisant communiquer des composants plus simples. En particulier, il apparaît sur le modèle générique que les ports des composants ne sont pas typés. Le mécanisme de polymorphisme permet la manipulation de données représentées par n'importe quelle classe qui hérite de la classe générique *Value*. Pour sa part, le chapitre III montre comment des services concurrents sont combinés et associés aux modes de fonctionnement de SEP afin de réaliser le comportement des composants élémentaires. Ces services concurrents sont représentés par des méthodes décrites dans le langage Java et sont donc typés. Il est donc possible d'utiliser cette information pour inférer le type des ports relatifs aux services et valider ainsi les connexions entre composants en s'assurant que le type des données potentiellement émises est compatible avec le type attendu des données reçues. L'explication de ce mécanisme mis en place dans SEP est l'objet de ce chapitre.

Dans la section 1-, on rappelle la notion de type et de sous-type, on présente leur implémentation dans SEP et on insiste sur l'intérêt de l'utilisation de règles de typage souples.

La section 2- présente le mécanisme mis en place pour inférer, lors de la construction des composants élémentaires, une information sur le type des ports. Cette information servira ensuite à valider les connexions entre les composants élémentaires lors de la construction de modèles structurels.

Enfin, la section 3- montre les problèmes sous-jacents à l'utilisation de types pour la résolution dynamique d'opérateurs binaires et n-aires. En particulier, on montre les insuffisances du mécanisme de liaison dynamique utilisé par Java. On propose alors un mécanisme de remplacement.

1- Généralités.

type

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

sous-type

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

super-type

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 de tout type τ'' tel que τ' ≤τ''. 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 (cf. section 3-2). 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 sur les données 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 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 type défini par l'utilisateur. Afin de permettre une définition générique des objets qui manipulent les données, toutes les classes définissant des types 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. Ce n'est pas l'objectif recherché.

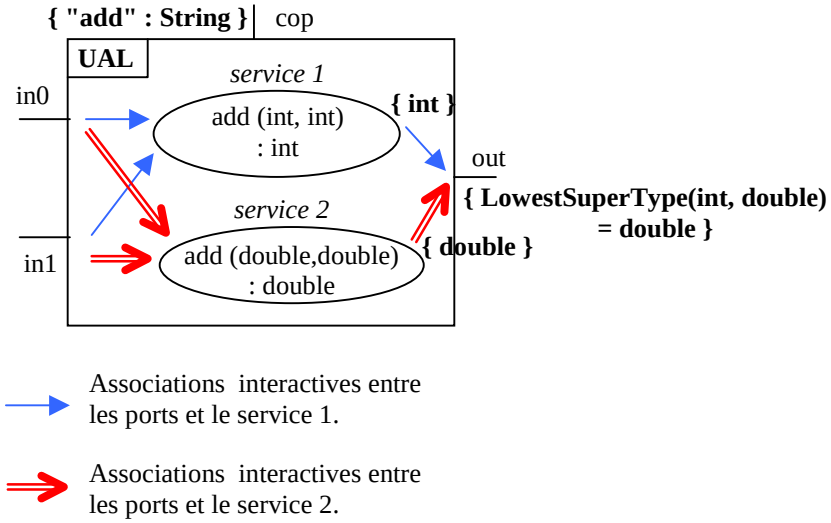

Figure 1 – Inférence du type associé aux ports du composant UAL.

interface

L'interface de communication d'un composant SEP est composée d'un ensemble de ports. Ces ports reçoivent (émettent) des données en provenance (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, une validation de types est effectuée lors de la création de connexions.

2- 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.

Dans une première étape, il s'agit d'inférer le type du composant élémentaire en fonction des types spécifiés par les services concernés. Le type de chaque port est inféré comme étant le plus petit super-type commun aux types des paramètres relatifs des services concernés ('LowestSuperType'). La Figure 1 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*)[1]. Le plus petit super type commun aux types *IntValue* et *DoubleValue* est *DoubleValue* puisque *IntValue* est un sous-type de *DoubleValue*[2]. Le type inféré pour les ports in0, in1 et out de l'UAL sera donc *DoubleValue*. On peut souligner que le plus petit super-type commun existe nécessairement puisque *Value* est le super-type de tous les types.

introspection

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. 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. En fait, il s'agit d'inférer le type du connecteur comme étant le plus petit super-type commun des types

[1] Sur la figure, les types *DoubleValue* et *IntValue* sont respectivement remplacés par la notation condensée *int* et *double*.

[2] 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).

associés aux ports sources. Le type du connecteur doit alors être un sous-type du type de chacun des ports destinations.

Formellement, T_{sep} est l'ensemble des types manipulés par SEP. Soit un connecteur C, $P_{src} \subset T_{sep}$ l'ensemble des types des ports susceptibles d'émettre des données à travers C et $P_{dst} \subset T_{sep}$ l'ensemble des types des ports susceptibles de recevoir des données en provenance de C. Soit $S(P_{src})$ l'ensemble des super-types de P_{src} et τ_C le type du connecteur C. On a :

$$S(P_{src}) = \{\tau' \in T_{sep} \mid \forall \tau \in P_{src}\ \tau \leq \tau'\}$$
$$\tau_C \in S(P_{src}),\ \forall \tau \in S(P_{src}),\ \tau_C \leq \tau.$$

En définitive, il s'agit alors de vérifier que : $\forall \tau \in P_{dst} \quad \tau_C \leq \tau$.
Dans le cas contraire, la connexion est refusée.

3- Invoquer les opérateurs adaptés.

Dans SEP, les composants émettent des données vers les autres composants qui ont alors la charge de les interpréter selon leur propre spécification. Ces données pourront soit déclencher l'exécution d'un service, soit être utilisées pour la réalisation d'un service du composant. En présence de sous-typage et de polymorphisme[3] ce mécanisme devient complexe. En effet, puisque les types sont modélisés par des classes, on aimerait modéliser la relation de sous-typage en utilisant le mécanisme d'héritage.

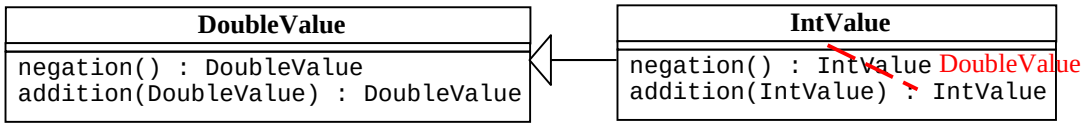

Figure 2 – modélisation naïve d'une relation de sous-typage.

redéfinition

En particulier, les langages orientés objets permettent la redéfinition de méthodes dans une relation d'héritage. Si une classe *DoubleValue* définit la méthode *DoubleValue negation()* pour changer le signe d'un nombre réel, alors une classe *IntValue* héritant de la classe *DoubleValue* peut redéfinir[4] cette méthode (cf. Figure 2). En présence de polymorphisme, le mécanisme de liaison dynamique permettra alors de choisir dynamiquement à l'exécution laquelle des deux méthodes il faudra invoquer en fonction de la classe effective de l'objet manipulé. Sans le mécanisme de liaison dynamique, le choix se ferait à la compilation en fonction du type de la référence manipulée. Une telle modélisation laisserait à la charge de la machine virtuelle Java le choix de l'opérateur adapté. Le premier problème est qu'en Java, en cas de redéfinition, le type de retour doit être strictement identique à celui de la méthode redéfinie. C'est-à-dire que la classe *IntValue*, ne peut redéfinir qu'une méthode *DoubleValue negation ( )*.

Le deuxième problème vient de la méthode addition. La méthode addition de la classe *DoubleValue* est surchargée dans la sous-classe *IntValue*. En effet, on dit qu'une méthode μ2

surcharge

signature

[3] Le polymorphisme est la capacité qu'ont les variables de type τ à référencer des objets dont la classe est de type τ ou toute classe dont le type τ' est un sous-type de τ : $\tau' \leq \tau$.

[4] Il n'y a redéfinition que si la signature de la méthode est conservée. La signature d'une méthode est constituée de son nom et de la liste ordonnée du type de ses paramètres. En Java, en cas de redéfinition, le type de retour de la méthode surchargée doit être conservé.

surcharge μ1 si et seulement si μ1 et μ2 sont définies dans la même relation d'héritage, ont le même nom et des signatures différentes. La signature étant constituée du nom de la méthode et de la liste ordonnée du type des paramètres. La signature n'est pas concernée par le type de retour de la méthode.

Cette implémentation introduit une dissymétrie dans le rôle des paramètres. En effet, le paramètre implicite[5] 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 section 3-1.

Dans SEP, il s'agit de choisir dynamiquement les opérateurs adaptés en fonction de la classe effective[6] 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.

3-1. La liaison dynamique en Java.

compatibilité

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 : $\tau_C \leq \tau$.

méthode applicable

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[7] compatible avec le type du paramètre correspondant.

méthode spécifique

Une méthode μ1 est plus spécifique qu'une méthode μ2 si, à chaque fois que μ1 est applicable pour une invocation donnée, alors μ2 l'est aussi (il peut exister des cas où μ2 est applicable et μ1 ne l'est pas). Plus précisément :

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

[5] Lors de l'invocation d'une méthode d'instance, l'objet sur lequel cette méthode est appliquée, est passé implicitement comme argument et peut être référencé par le mot clé *this*. A l'exécution, tout se passe alors comme si chaque méthode avait déclaré un paramètre de nom *this* et de type identique à la classe définissant la méthode. Ce paramètre est appelé implicite.

[6] 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*.

[7] 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*.

paramètres dont les types sont $\tau 1_2$, …, τn_2 ; Alors $\mu 1$ est dite plus spécifique que $\mu 2$ 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 2, la classe *DoubleValue* définit la méthode *DoubleValue addition(DoubleValue)*, il serait alors souhaitable que la classe *IntValue* surcharge[8] 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 addition(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

[8] 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.

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 :

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

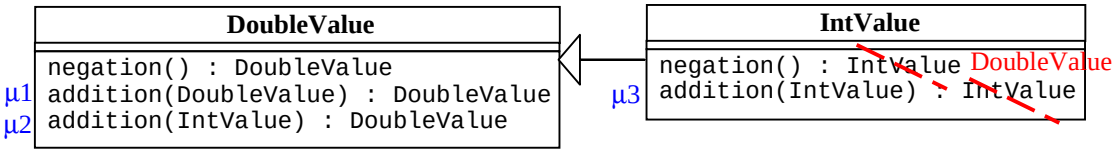

Figure 3 – modélisation complète.

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)[9] 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.

Le schéma de la Figure 3 tient compte de ses modifications, les résultats suivants sont alors obtenus :

<table>
<tr><th colspan="2">Type de this</th><th colspan="2">Type de l'argument</th><th>Méthode invoquée</th></tr>
<tr><td colspan="2">IntValue/DoubleValue</td><td colspan="2">DoubleValue</td><td>μ1</td></tr>
<tr><td colspan="2">DoubleValue</td><td colspan="2" rowspan="2">IntValue</td><td>μ2 puis μ1 (redirection)</td></tr>
<tr><td colspan="2" rowspan="2">IntValue</td><td>μ3</td></tr>
<tr><td rowspan="2">DoubleValue</td><td rowspan="2">IntValue</td><td>DoubleValue</td><td>IntValue</td><td>μ1 puis μ3 (cast)</td></tr>
<tr><td colspan="2">IntValue</td><td>μ3 (liaison dynamique par μ2)</td></tr>
</table>

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 4.

[9] Cast : conversion explicite et forcée du type d'une expression dans un de ses sous-types.

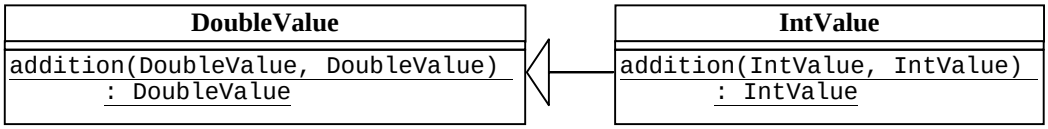

Figure 4 – schéma d'héritage ne faisant intervenir que des méthodes statiques.

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).

Le code Java illustrant les problèmes mentionnés dans cette section et les solutions pouvant être mises en œuvre sont présentés en annexes. 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.

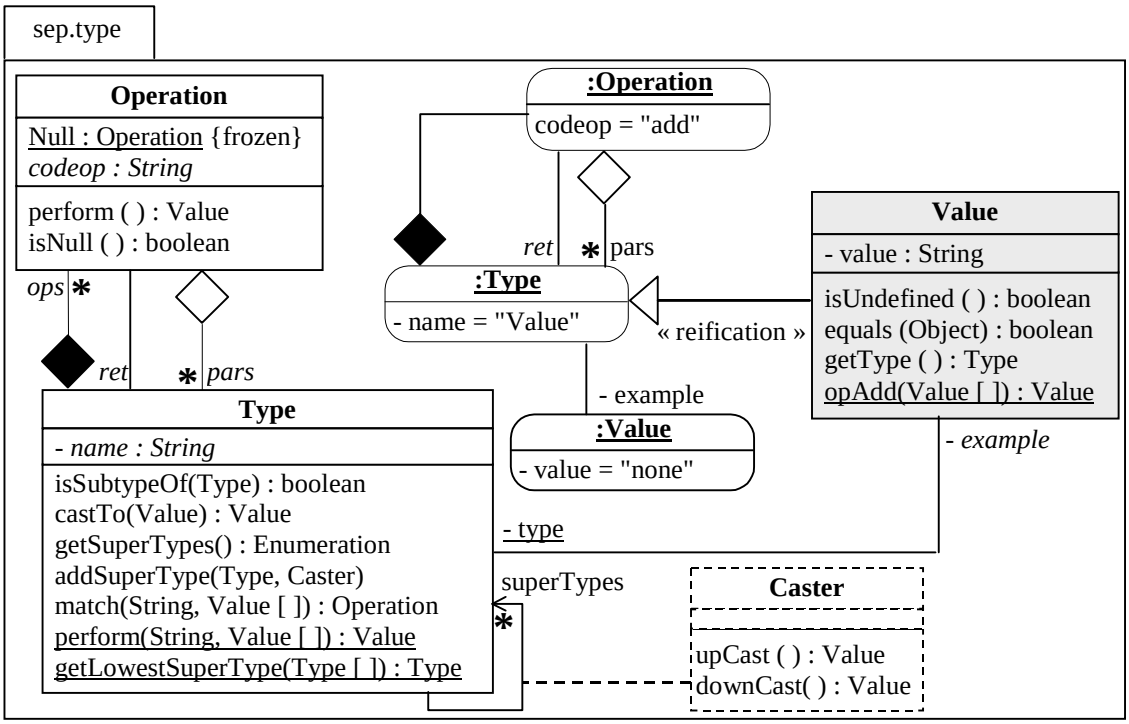

Figure 5 – Méta-modèle des types manipulés par SEP.

3-2. La solution proposée.

Le mécanisme proposé permet de résoudre les problèmes mis en évidence dans la section 3-1. A cette fin, les mécanismes de sélection de méthodes de Java (liaison dynamique et résolution de la surcharge) ne doivent pas être utilisés. SEP doit alors manipuler la notion de type, les relations de sous-typage et les opérateurs. Il ne se contente plus de manipuler des valeurs. En effet, afin de choisir la méthode adaptée, il faut être capable d'associer chaque donnée à son type et non à un type d'une variable qui référence un objet représentant cette donnée. Il faut ensuite être capable de choisir la méthode la plus spécifique en fonction du

type de ces données. Pour cela, on devra manipuler les opérations associées à un type à travers une structure de sous-typage.

C'est pourquoi, un méta-modèle des types SEP a été défini. Ce méta-modèle permet les manipulations requises et permet une extension de l'arbre de sous-typage par les utilisateurs de SEP. Ce méta-modèle est représenté par le diagramme statique UML de la Figure 5.

Le diagramme présente les types comme une structure ayant un nom 'name', une liste d'Operations 'ops', éventuellement des super-types 'superTypes' et l'instance d'une valeur de type 'example'. Le type *Value* est le seul à n'avoir aucun super-type. Tous les autres ont au minimum *Value* ou un de ses sous-types comme super-type.

Les opérations sont définies comme des entités identifiées par un code opération 'codeop', une liste de types de paramètres 'pars' et un type de retour 'ret'.

Un diagramme objet présente un exemple d'instanciation de la classe type pour obtenir une représentation du type *Value*. Un processus de réification nous permet alors de construire la classe *Value* dont hériteront les données de type *Value*.

En pratique, un utilisateur voulant définir un nouveau type devra définir une classe héritant de la classe *Value*. Lors de l'analyse d'un composant utilisant ce type, une base de données correspondant au diagramme de la Figure 5 sera automatiquement constituée sous la forme d'une table d'instances de la classe *Type*.

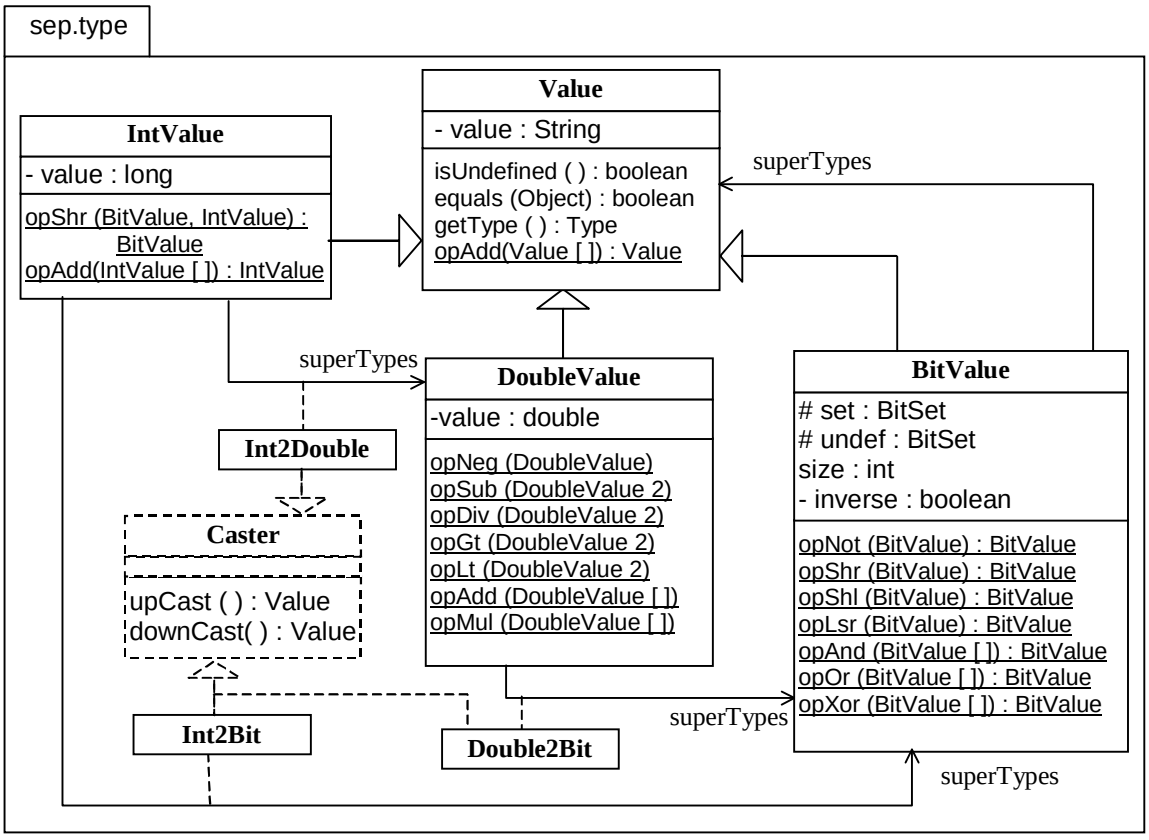

Figure 6 – Arbre de sous-typage (Value, DoubleValue, BitValue).

Par exemple, l'ajout du type *IntValue* représenté par la Figure 6, c'est-à-dire comme un sous-type de *DoubleValue* et définissant en plus les opérations *BitValue Shr(BitValue, IntValue)* et *IntValue add(IntValue)*, se définirait de la manière suivante. Le concepteur doit

d'abord écrire une classe *IntValue* héritant de la classe *Value*. Cette classe définit les attributs nécessaires à la définition du domaine des entiers relatifs, elle définit en plus les méthodes permettant une équivalence avec une représentation textuelle (*String toString()* et *Value getValue(String)*). Les opérations sont représentées par des méthodes statiques. Enfin, la relation de sous-typage est définie par la méthode *Type getType()*. Cette méthode permet l'accès à une référence unique (statique) vers un objet de la classe *Type* de nom "sep.type.*IntValue"*. Les super-types sont alors *BitValue* et *DoubleValue* et les classes associées permettant la conversion vers et à partir d'un *IntValue* sont respectivement *Int2Bit* et *Int2Double*. Un squelette du code à écrire est :

```
package sep.type;
public class IntValue extends Value {
        …
        public static BitValue opShr(BitValue value, IntValue offset) { … }
        public static IntValue copAdd(IntValue values[]) { … }

        private static Type type = null;
        public Type getType() {
            if (type == null) {
                type = new Type("sep.type.IntValue", this);
                type.addSuperType(new BitValue().getType(), new Int2Bit());
                type.addSuperType(new DoubleValue(0).getType(), new Int2Double());
            }
            return type;
        }
}
```

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.

3-3. Mise en œuvre

Notre objectif est que chaque opération soit définie par une seule méthode statique dans la classe représentant le plus petit type concerné dans les paramètres de l'opération et ceci sans modification des classes représentant les super-types. C'est ainsi qu'on définira la méthode statique *BitValue opShr(BitValue, IntValue)* dans la classe *IntValue* plutôt que dans la classe *BitValue*. L'idée est que lors de la définition du type *BitValue*, cette opération n'était pas encore possible. Elle ne devient envisageable qu'à partir du moment où le type *IntValue* est défini. Nous ne voulons pas avoir à modifier a posteriori la classe *BitValue* pour corriger les problèmes liés au polymorphisme.

Lorsqu'une opération de calcul sur des valeurs doit être effectuée[10], il faut d'abord déterminer la classe à parcourir pour trouver l'opération concernée. Etant donné notre objectif, la classe à parcourir est celle représentant le plus petit type 'à l'exécution' des arguments de l'opération à invoquer. Le mécanisme de Java aurait choisi la classe représentant le type du paramètre implicite de l'opération et aurait ainsi introduit une dissymétrie avec les conséquences dont nous avons parlé.

[10] Un utilisateur peut par exemple demander l'exécution d'une opération dont il connaît le code opération et les arguments en appelant la méthode statique *perform* de la classe *Type*.

La deuxième étape consiste à choisir la méthode la mieux adaptée aux classes effectives des arguments d'appel. Pour cela, il faut d'abord déterminer la liste des méthodes applicables puis parmi ces méthodes choisir la plus spécifique.

Si aucune méthode applicable n'est trouvée, il faut alors continuer récursivement la recherche d'une méthode applicable dans les classes qui représentent les plus petits super-types. On remarquera que le plus petit super-type d'un type donné n'est pas forcément unique puisque la relation de sous-typage est une relation d'ordre partiel. Il faudra alors faire l'union des méthodes applicables trouvées.

Si aucune méthode applicable n'est trouvée après la recherche dans la classe *Value*, alors une erreur est levée et l'invocation du service est rejetée. Sinon, il s'agit de choisir parmi les méthodes applicables la méthode la plus spécifique. Nous obtenons ainsi les résultats attendus.

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 références et non les classes effectives ce qui entraîne les problèmes présentés dans la section précédente.

En conclusion, ce chapitre présente le mécanisme de représentation des types utilisé dans SEP. Il montre comment le problème de modélisation des opérateurs n-aires est pris en compte et offre une alternative intéressante à celui proposé par Java. Cette gestion de types et de sous-types nous permet de valider les connexions effectuées entre plusieurs composants élémentaires lors de la construction d'un modèle structurel. SEP peut ainsi rejeter les connexions qui ne permettent pas de garantir une communication raisonnable relativement aux types spécifiés et qui conduiraient inévitablement à des erreurs en simulation.