Guide Utilisateur AROM v2.0

Guide Utilisateur AROM v2.0

par Christophe Bruley, Philippe Genoud, et Véronique Dupierris

Publié le $Date: 2003/03/05 09:06:48 $
Ce document est encore en cours de rédaction. Il est identifié par le numéro de release CVS du document principal : $Id: ug.xml,v 1.33 2003/03/05 09:06:48 dupierri Exp $

Table des matières

Liste des tableaux

Liste des illustrations

Liste des exemples

Liste des équations

I. Introduction au système AROM

Ce document décrit la plate-forme AROM dans sa version 2.0. Ce document est à la fois destiné aux utilisateurs d'AROM qui y trouveront une description complète du modèle de représentation de connaissances proposé par la plate-forme, mais aussi aux développeurs qui souhaitent intégrer AROM à leurs propres applications puisqu'il décrit l'architecture de la plate-forme et l'API du système.

Chapitre 1

Qu'est-ce qu'AROM ?

1.1. Introduction

Pour modéliser des connaissances et/ou des données dans un domaine d'application, de nombreuses méthodes de conception ([Che76] et [RBP+91], par exemple) proposent d'utiliser deux concepts différents : d'une part, des classes d'entités ou d'objets pour décrire les regroupements d'individus similaires, et, d'autre part, des relations ou associations pour regrouper les liens similaires entre ces différents individus. Pour exprimer ces modèles, la grande majorité des systèmes de représentation de connaissances par objets (SRCO) ne disposent, quant à eux, que d'un seul concept : la classe (ou une variante).

L'absence de représentation explicite des associations est, selon nous, très dommageable à la déclarativité et à l'expressivité des SRCO. Elle oblige le concepteur à implanter les associations plutôt qu'à les représenter effectivement, ce qui nuit à l'intelligibilité des bases de connaissances. De plus, elle interdit la mise en oeuvre de mécanismes et d'opérations spécifiques aux associations, comme la maintenance de la cohérence, ou l'adaptation aux associations d'opérations habituellement définies sur les classes, telles que la spécialisation. Dans AROM, les liens entre objets sont explicitement représentés par des entités nommées associations. Celles-ci occupent dans AROM une place d'importance égale à celle des classes.

1.2. Historique

La plate-forme AROM est distribuée depuis Janvier 2000, dans sa version 1.0. La version 2.0 est essentiellement un travail de ré-écriture de cette première version, destiné à rendre le système AROM plus modulaire et plus extensible. L'extensibilité doit en particulier permettre de construire sur la base du système AROM d'autres modèles de représentation de connaissances, comme c'est par exemple le cas avec le système de représentation de connaissances méthodologiques AROMTasks.

Chapitre 2

Représenter des connaissances avec AROM

2.1. Classes et objets

Une classe décrit un ensemble d'objets ayant des propriétés communes. Les classes sont descriptives : elles fournissent un ensemble de conditions nécessaires mais non suffisantes d'appartenance des objets à la classe. Chaque classe est caractérisée par un ensemble de propriétés appelées variables (cet ensemble de propriétés défini *l'intention* de la classe). Les classes ne disposent pas de méthodes au sens des langages de programmation orientée objet.

Une variable correspond à une propriété dont le type n'est pas issu d'une classe de la base de connaissances. En effet, il n'existe pas de variable référençant un ou plusieurs objets ; il est nécessaire de passer par une association.

Chaque variable est caractérisée par un ensemble de facettes qui peut être subdivisé en trois catégories :

- Les facettes de restriction du domaine des valeurs acceptées par la variable : ces facettes vont permettre de contraindre l'ensemble des valeurs admises par la variable en précisant par exemple le type des valeurs acceptées, ou les propriétés que doivent respecter ces valeurs pour être acceptées.
- Les facettes d'inférence qui permettent d'inférer la valeur d'une variable lorsque celle-ci n'a pas été fixée dans l'objet. Parmis les moyens d'inférences utilisés, on peut citer la valeur par défaut fixée dans la classe pour toutes ses instances, l'équation du langage algébrique ou encore la méthode Java attachée à la variable chargée de calculer la valeur de celle-ci.
- Les facettes de documentation qui permettent d'associer diverses informations à la variable.

Les classes sont structurées de manière hiérarchique par la relation de spécialisation qui, en AROM, est simple. La spécialisation d'une classe est réalisée par au moins une des deux opérations suivantes : ajout ou modification d'une valeur de facette ou ajout d'une nouvelle variable. Une classe hérite les facettes de variables qu'elle ne redéfinit pas. Par ailleurs, les sous-classes d'une classe ne sont pas supposées mutuellement exclusives (leurs descriptions peuvent englober des instances communes),

ni exhaustives (l'union des ensembles d'instances des sous-classes n'est pas nécessairement égal à l'ensemble des instances de leur super-classe).

Un objet AROM représente une entité distinguable du domaine modélisé. Chaque objet est attaché à une classe mais appartient également à tous les ancêtres de cette classe. Le terme "attaché", utilisé dans le système TROEPS [Euz93], met en exergue la contingence et l'aspect dynamique du fait d'appartenir à une classe. Il reflète le fait que, contrairement à la programmation orientée objet, les SRCO comme AROM autorisent l'utilisateur ou un de ses programmes à déplacer un objet d'une classe vers une autre parce que des informations additionnelles ont été obtenues sur cet objet.

2.2. Associations et tuples

En AROM, les associations jouent un rôle aussi important dans la représentation de connaissances que les classes. Elles sont semblables aux associations du modèle d'UML : une association représente un ensemble de liens similaires entre n (n >= 2) classes, distinctes ou non. Un lien est un n-uplet d'objets appartenant aux extensions (ensembles d'instances) des classes reliées par l'association. Une association définit ainsi un sous-ensemble du produit cartésien des classes qu'elle relie. En AROM, chaque association possède un nom.

Une association est décrite par ses rôles et ses variables. Un rôle r d'une association correspond à une connexion entre l'association et une des classes connectées, appelée classe correspondante du rôle et notée C(r). Chaque association n-aire possède donc n rôles et la valeur de chaque rôle ri (1<= i<= n) est une instance de la classe correspondante C(ri). Chaque rôle possède un nom et une multiplicité. La multiplicité d'un rôle r a le même sens qu'en UML. Il s'agit d'un intervalle d'entiers tel que, si l'on fixe la valeur des n-1 rôles différents de r, le nombre d'instances de C(r) pouvant apparaître dans le rôle r doit appartenir à cet intervalle. La multiplicité est décrite par la facette multiplicity: pour laquelle est donnée la valeur minimum (min:) et la valeur maximum (max:). Pour cette dernière, on utilise, comme en UML, le symbole * pour dénoter une valeur infinie. Outre la facette multiplicity:, la facette documentation: permet d'associer une documentation à un rôle.

Une variable d'association représente une propriété associée à un lien. D'un point de vue mathématique, une variable v d'une association dont les rôles sont r1,r2,...,rn, est définie comme une fonction :

```
 v: C(r1)XC(r2)X...XC(rn)->T
(o1,o2,...,on) -> v(o1,o2,...,on)
```

où T est un type d'AROM et oi un objet de la classe C(ri). Les variables d'association disposent des mêmes facettes que les variables de classe.

Un tuple d'une association n-aire possédant m variables vi (1<= i <= m) est le (n+m)-uplet formé des n objets du lien et des valeurs des m variables de l'association :

Une association étant un ensemble de liens, deux tuples ne peuvent pas contenir le même lien.

Les associations sont organisées en hiérarchies grâce à une relation de spécialisation. La spécialisation d'associations, comme celles des classes, est simple en AROM. Elle permet de greffer sur une

hiérarchie d'associations un héritage de rôles, de variables et de facettes. Une association A2 spécialise une association A1 par au moins une des opérations suivantes :

- Spécialisation de la classe correspondante d'un rôle de A1 ;
- Modification ou ajout d'une facette à une variable ou à un rôle de A1;
- Ajout d'une variable à A1.

La spécialisation d'association correspond à l'inclusion ensembliste des liens. Autrement dit, les liens d'une association appartiennent à l'ensemble des liens de sa super-association, si elle existe. En conséquence, la spécialisation d'association par ajout de rôle n'est pas autorisée en AROM : elle reviendrait à considérer l'inclusion de n+1-uplets dans un ensemble de n-uplets. L'arité d'une association est donc préservée dans toute sa hiérarchie.

2.3. BNF du langage AROM

```
kb ::=
  KNOWLEDGE_BASE IDF
  documentations
  kb_components;

documentations ::=
  // vide | documentations documentation ;

documentation ::=
  DOCUMENTATION (STRING_LITERAL | LONG_LITERAL | URL_LITERAL);

kb_components ::=
  // vide | kb_components kb_component ;

kb_component ::=
  class | association | instance | tuple ;

class ::=
  CLASS IDF
  super_class_descriptor
  documentations
  class_variables_and_composition ;

super_class_descriptor ::=
  // vide | SUPER_CLASS IDF ;

class_variables_and_composition ::=
  // vide
  | VARIABLES
    variables ;

variables ::=
  // vide | variables  variable ;

variable ::=
```

```
  VARIABLE IDF
  variable_descriptors ;

variable_descriptors ::=
  // vide | variable_descriptors variable_descriptor   ;

variable_descriptor ::=
  TYPE type
  | DOMAIN domain
  | CARDINALITY variable_cardinality
  | DEFAULT value
  | documentations
  | UNIT STRING_LITERAL
  | ATTACHMENT (BOOLEAN_LITERAL | attachment)
  | DEFINITION equations ;

attachment ::=
  IDF classnames COLON IDF ;

classnames ::=
  // vide | classnames classname ;

classname ::=
   DOT IDF;

variable_cardinality ::=
  min_bound max_bound ;

min_bound ::=
  // vide | MIN INTEGER_LITERAL ;

max_bound ::=
  // vide | MAX INTEGER_LITERAL | MAX MULT ;

type ::=
  base_type | SET_OF base_type | LIST_OF base_type  ;

base_type ::=
  INTEGER | FLOAT | BOOLEAN | STRING | IDF ;

domain ::=
  domain_set | domain_interval ;

domain_interval ::=
  LBRACK (value | MULT) DOTDOT (value | MULT) RBRACK ;

domain_set ::=
  LCURL values RCURL ;

values ::=
  value | values COMMA value ;

value ::=
  literal
```


```
  | LCURL RCURL
  | LCURL literals RCURL
  | LBRACK RBRACK
  | LBRACK literals RBRACK  ;

literals ::=
  literal
  | literals COMMA literal;

literal ::=
  INTEGER_LITERAL
  | MINUS INTEGER_LITERAL
  | FLOAT_LITERAL
  | MINUS FLOAT_LITERAL
  | STRING_LITERAL
  | BOOLEAN_LITERAL
  | IDF ;

association ::=
  ASSOCIATION IDF
  super_association_descriptor
  documentations
  association_roles
  association_variables ;

super_association_descriptor ::=
  // vide | SUPER_ASSOCIATION IDF ;

association_variables ::=
  // vide
  | VARIABLES
    variables  ;

association_roles ::=
  // vide
  | ROLES
    roles  ;

roles ::=
  // vide
  | roles
    ROLE IDF
    role_descriptors ;

role_descriptors ::=
  // vide
  | role_descriptors role_descriptor  ;

role_descriptor ::=
  TYPE IDF
  | DOMAIN domain_role
  | MULTIPLICITY role_multiplicity
  | documentations ;
```

```
role_multiplicity ::=
  min_bound max_bound ;

domain_role ::=
  LCURL instance_values RCURL ;

instance_values ::=
  instance_value | instance_values COMMA instance_value ;

instance_value ::= IDF ;

instance ::=
  INSTANCE IDF
  ISA IDF
  instance_descriptors;

instance_descriptors ::=
  // vide | instance_descriptors instance_descriptor ;

instance_descriptor ::=
  IDF EQUAL value | documentations;

tuple ::=
  TUPLE
  ISA IDF
  tuple_descriptors ;

tuple_descriptors ::=
  // vide | tuple_descriptors tuple_descriptor ;

tuple_descriptor ::=
  IDF EQUAL value | documentations ;

equations ::=
  // vide | equations equation ;

equation  ::=
  portee COLON expression_equation | expression_equation ;

portee ::=
  portee_variable | portee COMMA portee_variable ;

portee_variable ::=
  IDF IN expr_cond ;

expression_equation ::=
  expr_gauche EQUAL expr_droite ;

expr_gauche ::=
  IDF DOT IDF | IDF ;

expr_droite ::=
 expr_cond
 | IF expr_cond
```


```
    THEN
    expr_droite
    ELSE
    expr_droite ;

expr_quantifiee ::=
  ALL IDF IN expr_cond COLON expr_cond
  | EXISTS IDF IN expr_cond COLON expr_cond
  | ALL IDF IN expr_cond COMMA expr_quantifiee
  | EXISTS IDF IN expr_cond COMMA expr_quantifiee ;

expr_cond ::=
  expr_quantifiee
  | expr_cond AND expr_cond
  | expr_cond OR expr_cond
  | NOT expr_cond
  | expr_cond EQUAL expr_cond
  | expr_cond INEQUAL expr_cond
  | expr_cond GT expr_cond
  | expr_cond GTE expr_cond
  | expr_cond LT expr_cond
  | expr_cond LTE expr_cond
  | expr_cond MEMBER expr_cond
  | expr_cond PLUS expr_cond
  | expr_cond MINUS expr_cond
  | expr_cond MULT expr_cond
  | expr_cond DIVISION expr_cond
  | expr_cond DIV expr_cond
  | expr_cond MOD expr_cond
  | expr_cond POWER expr_cond
  | LOG10 LPAREN expr_cond RPAREN
  | LN LPAREN expr_cond RPAREN
  | EXP LPAREN expr_cond RPAREN
  | SQRT LPAREN expr_cond RPAREN
  | SIN LPAREN expr_cond RPAREN
  | COS LPAREN expr_cond RPAREN
  | TAN LPAREN expr_cond RPAREN
  | ABS LPAREN expr_cond RPAREN
  | ROUND LPAREN expr_cond RPAREN
  | FLOOR LPAREN expr_cond RPAREN
  | CEIL LPAREN expr_cond RPAREN
  | MIN2 LPAREN expr_cond COMMA expr_cond RPAREN
  | MAX2 LPAREN expr_cond COMMA expr_cond RPAREN
  | SIZE LPAREN expr_cond RPAREN
  | LEFT LPAREN expr_cond COMMA expr_cond RPAREN
  | RIGHT LPAREN expr_cond COMMA expr_cond RPAREN
  | INDEX_OF LPAREN expr_cond COMMA expr_cond RPAREN
  | LENGTH LPAREN expr_cond RPAREN
  | SUBSTRING LPAREN expr_cond
    COMMA expr_cond COMMA expr_cond RPAREN
  | NTH LPAREN expr_cond COMMA expr_cond RPAREN
  | RANDOM LPAREN expr_cond COMMA expr_cond RPAREN
```

```
  | CONCATENATE LPAREN portee COLON expr_droite
    COMMA expr_cond RPAREN
  | MIN_ELEMENT LPAREN portee COLON expr_droite RPAREN
  | MAX_ELEMENT LPAREN portee COLON expr_droite RPAREN
  | MIN_VALUE LPAREN portee COLON expr_droite RPAREN
  | MAX_VALUE LPAREN portee COLON expr_droite RPAREN
  | SUM LPAREN portee COLON expr_droite RPAREN
  | PRODUCT LPAREN portee COLON expr_droite RPAREN
  | AVERAGE LPAREN portee COLON expr_droite RPAREN
  | STD_DEV LPAREN portee COLON expr_droite RPAREN
  | VARIANCE LPAREN portee COLON expr_droite RPAREN
  | MINUS expr_cond %prec UMINUS
  | LPAREN expr_cond RPAREN
  | expr_cond DOT IDF
  | expr_cond AT IDF BANG IDF
  | expr_cond BANG IDF
  | BOOLEAN_LITERAL
  | IDF
  | INTEGER_LITERAL
  | FLOAT_LITERAL
  | STRING_LITERAL
  | SET LPAREN portee COLON expr_droite RPAREN
  | KNOWN_SET LPAREN portee COLON expr_droite RPAREN
  | SELECT LPAREN portee COLON expr_droite RPAREN
  | UNION LPAREN portee COLON expr_droite RPAREN
  | INTER LPAREN portee COLON expr_droite RPAREN
  | LBRACK RBRACK
  | LBRACK liste_expr_cond RBRACK
  | LCURL RCURL
  | LCURL liste_intervalle RCURL
  | IS_DEFINED LPAREN expr_droite RPAREN
  | IS_VOID LPAREN expr_droite RPAREN
  | IS_KNOWN LPAREN expr_droite RPAREN
  | __ERROR LPAREN expr_droite RPAREN  ;

liste_intervalle ::=
  intervalle | liste_intervalle COMMA intervalle ;

intervalle ::=
  expr_cond DOTDOT expr_cond | expr_cond  ;

liste_expr_cond ::=
  expr_cond | liste_expr_cond COMMA expr_cond  ;
```

Terminaux du langage AROM :

```
LineTerminator = \r|\n|\r\n
InputCharacter = [^\r\n]
WhiteSpace = {LineTerminator} | [ \t\f]
Comment = {TraditionalComment} | {EndOfLineComment}
TraditionalComment = "/*" [^*] {CommentContent} \*+ "/"
EndOfLineComment = "//" {InputCharacter}* {LineTerminator}
CommentContent = ( [^*] | \*+[^*/] )*
```


```
Identifier = [:jletter:][:jletterdigit:]*
DecIntegerLiteral = 0 | [1-9][0-9]*
DoubleLiteral = {FLit1}|{FLit3}|{FLit4}
FLit1 = [0-9]+ \. [0-9]+ {Exponent}?
FLit3 = [0-9]+ {Exponent}
FLit4 = [0-9]+ {Exponent}?
Exponent = [eE] [+\-]? [0-9]+
StringCharacter = [^\r\n\"\\]

KNOWLEDGE_BASE ::= "knowledge-base:" ;
DOCUMENTATION ::= "documentation:" ;
CLASS ::= "class:" ;
SUPER_CLASS ::= "super-class:"  ;
ASSOCIATION ::= "association:"  ;
SUPER_ASSOCIATION ::= "super-association:" ;
VARIABLES ::= "variables:" ;
VARIABLE ::= "variable:"  ;
ROLES ::= "roles:" ;
ROLE ::= "role:"  ;
INSTANCE ::= "instance:"  ;
TUPLE ::= "tuple:" ;
ISA ::= "is-a:" ;
TYPE ::= "type:"  ;
DOMAIN ::= "domain:" ;
DEFAULT ::= "default:" ;
DEFINITION ::= "definition:" ;
UNIT ::= "unit:" ;
ATTACHMENT ::= "attachment:"  ;
SET_OF ::= "set-of" ;
LIST_OF ::= "list-of"  ;
MIN ::= "min:" ;
MAX ::= "max:" ;
MULTIPLICITY ::= "multiplicity:"  ;
CARDINALITY ::= "cardinality:" ;
INTEGER ::= "integer" ;
FLOAT ::= "float" ;
BOOLEAN ::= "boolean" ;
STRING ::= "string" ;
BOOLEAN_LITERAL ::= "true" | "false";

LBRACK ::= "[" ;
RBRACK ::= "]" ;
LCURL ::= "{" ;
RCURL ::= "}" ;
COMMA ::= "," ;
DOTDOT ::= ".." ;
EQUAL ::= "=" ;
COLON ::= ":" ;
LTE ::= "<=" ;
GTE ::= ">=" ;
INEQUAL ::= "<>" ;
LT ::= "<" ;
```

```
GT ::= ">" ;
PLUS ::= "+" ;
MINUS ::= "-" ;
MULT ::= "*" ;
DIVISION ::= "/" ;
POWER ::= "^" ;
LPAREN ::= "(" ;
RPAREN ::= ")" ;
DOT ::= "." ;
BANG ::= "!" ;
AT ::= "@" ;
IN ::= "in" ;
MEMBER ::= "member" ;
THEN ::= "then" ;
ELSE ::= "else" ;
IF ::= "if" ;
AND ::= "and" ;
OR ::= "or" ;
NOT ::= "not" ;
LOG10 ::= "log10" ;
LN ::= "ln" ;
EXP ::= "exp" ;
SIN ::= "sin" ;
COS ::= "cos" ;
TAN ::= "tan" ;
SQRT ::= "sqrt" ;
MIN_ELEMENT ::= "minElement" ;
MAX_ELEMENT ::= "maxElement" ;
MIN_VALUE ::= "minValue" ;
MAX_VALUE ::= "maxValue" ;
MIN2 ::= "min" ;
MAX2 ::= "max" ;
SUM ::= "sum";
PRODUCT ::= "product" ;
UNION ::= "union";
INTER ::= "inter";
SIZE ::= "size" ;
SELECT ::= "select";
KNOWN_SET ::= "knownSet";
SET ::= "set";
RANDOM ::= "random";
DIV ::= "div";
MOD ::= "mod";
ABS ::= "abs";
NTH ::= "nth";
LEFT ::= "left";
RIGHT ::= "right";
LENGTH ::= "length";
INDEX_OF ::= "indexOf";
SUBSTRING ::= "substring";
CONCATENATE ::= "concatenate";
ROUND ::= "round";
```


```
CEIL ::= "ceil";
FLOOR ::= "floor";
AVERAGE ::= "average";
STD_DEV ::= "stdDev";
VARIANCE ::= "variance";
ALL ::= "all";
EXISTS ::= "exists";
IS_VOID ::= "isVoid";
IS_KNOWN ::= "isKnown";
IS_DEFINED ::= "isDefined";
```

II. Description de la plate-forme AROM version 2.0

Cette partie décrit comment l'organisation des différents composants qui constituent la plate-forme AROM

Chapitre 3

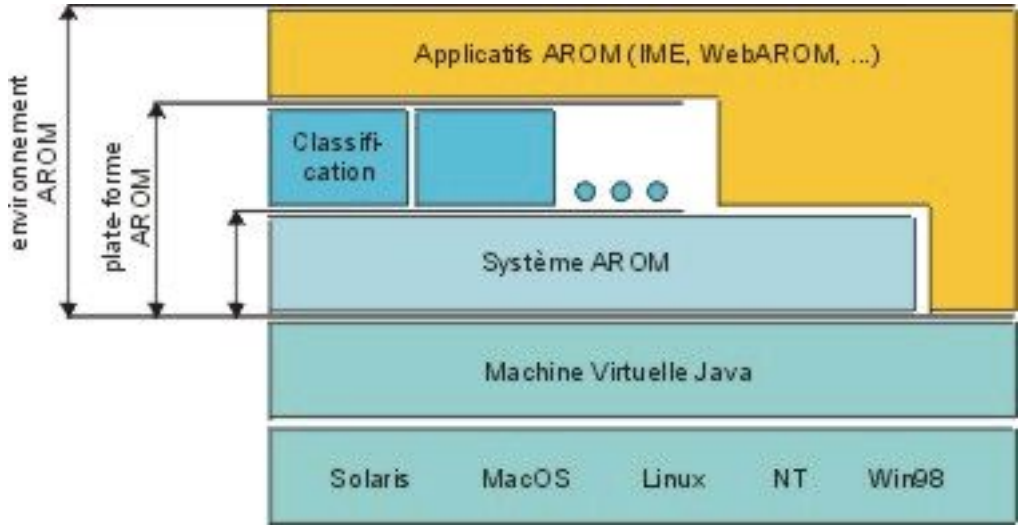
Architecture de la plate-forme AROM

3.1. Terminologie

Tout au long de ce document, nous utiliserons pour parler d'AROM les termes de modèle, de plate-forme ou parfois tout simplement le mot AROM. Définissons brièvement l'usage de ces termes :

- *Le modèle AROM* désigne le formalisme de représentation des connaissances proposé par AROM. C'est le modèle objet de représentation des connaissances. Ce modèle est décrit dans le chapitre suivant.
- *Le système AROM* est l'implémentation du modèle de représentation de connaissances : autrement dit la représentation sous forme d'objets informatiques des entités du modèle AROM.
- *La plate-forme AROM* est la partie applicative d'AROM qui englobe le système AROM et les outils d'exploitation de ces connaissances qui sont délivrés avec AROM (comme par exemple l'algorithme de classification des instances).
- *L'environnement AROM* est le terme le plus général qui désigne à la fois le système de représentation de connaissances et l'ensemble des applications qui gravitent autour de ce système : interface graphique de modélisation, interface de consultation web, etc ...

Figure 3-1. Désignation des différents éléments de l'environnement



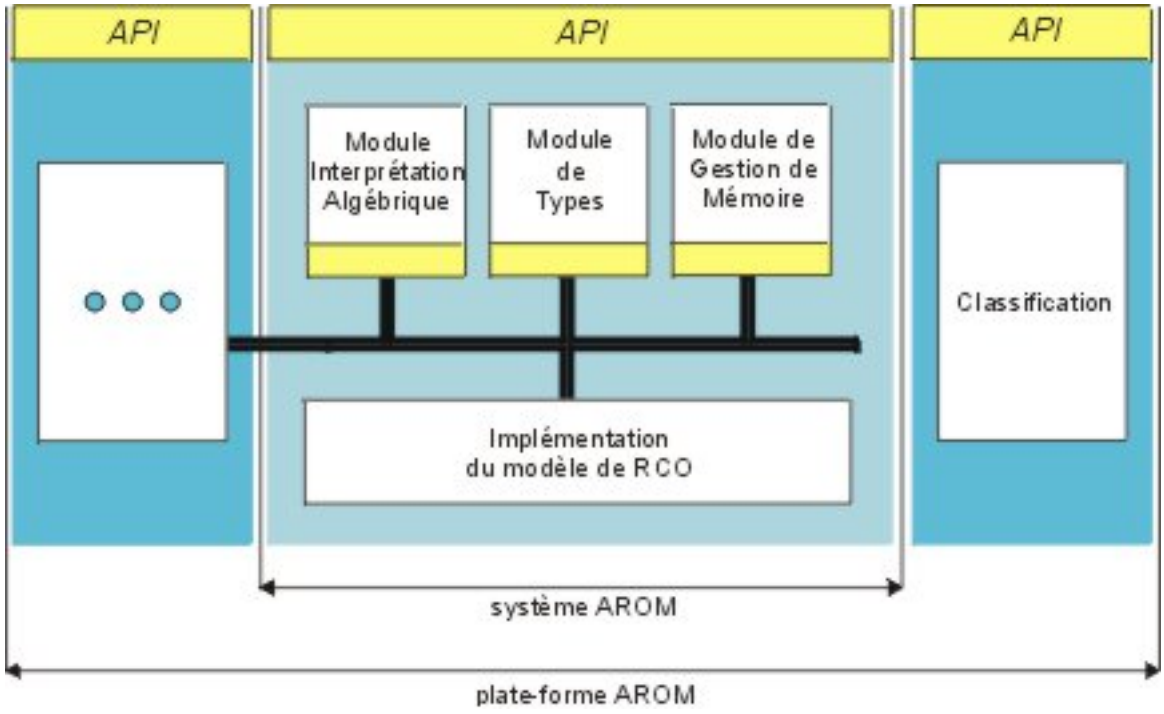
3.2. Modularité de la plate-forme

L'architecture de la plate-forme AROM est décrite ici de manière succincte. Pour une description complète de l'architecture logicielle, reportez vous au document d'implémentation de la plate-forme.

La plate-forme AROM est organisée en modules (ou composants). Chaque module a en charge la réalisation d'une fonctionnalité précise du système. La plate-forme actuelle est constituée des modules suivants :

- *Module de gestion de la mémoire* : assure le chargement et le déchargement des instances AROM depuis le disque vers la mémoire, afin d'optimiser l'occupation de la mémoire de la machine virtuelle Java lorsque les bases de connaissances comportent un grand nombre d'instances.
- *Module de types* : défini l'ensemble des types reconnus dans une base de connaissances AROM et les opérations possibles sur ces types.
- *Modèle de représentation de connaissances* : (pas encore sûr que cela fasse l'objet d'un module à part entière)
- *Module d'interprétation Algébrique* : assure l'interprétation d'équations algébriques dans AROM.

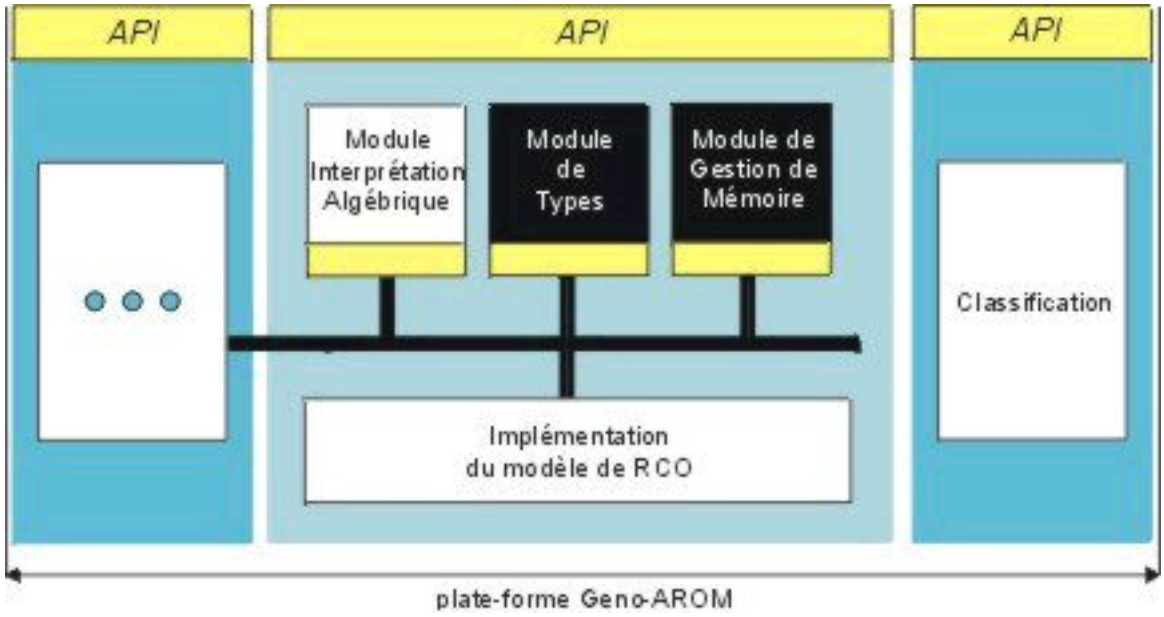
Figure 3-2. Organisation modulaire de la plate-forme AROM



Tous ces modules étant indispensables au bon fonctionnement de la plate-forme, il est clair qu'ils dépendent plus ou moins directement les uns des autres. Cependant, les communications entre modules passent par des API définies pour chacun des modules de la plate-forme. De cette manière, il est possible de modifier la plate-forme AROM en changeant l'implémentation de l'un des modules par une autre implémentation. La plate-forme AROM devient donc une plate-forme configurable, dont il est alors possible de décliner différentes versions ou configurations en fonction des implémentations de modules utilisées.

Ainsi, la plate-forme Geno-AROM est une configuration de la plate-forme AROM dédiée aux problèmes de génomique exploratoire. Geno-AROM est bâtie sur la plate-forme AROM, mais elle étend les possibilités de cette plate-forme en changeant les implémentations du module de type et du module de gestion de mémoire.

Figure 3-3. Organisation de la plate-forme Geno-AROM



Le module de gestion de mémoire de Geno-AROM est spécialisé dans le traitement de grands volumes de données. La persistance des instances du modèle AROM est assurée par une base de données relationnelle ou objet.

Le module de types de Geno-AROM est une extension du module de types d'AROM qui définit de nouveaux types adaptés à la représentation de connaissances génomiques (exemple : un type Sequence).

3.3. Configuration du système AROM

Le système AROM est construit sur une API et sur une implémentation de cette API. L'API d'AROM est composée d'un ensemble d'interface via lesquelles il est possible de manipuler les bases de connaissances AROM. Toutefois, ces seules interfaces ne suffisent pas à rendre le système AROM opérant : pour modéliser les bases de connaissances AROM, il faut que le système instancie des classes concrètes implémentant les interfaces définies dans l'API d'AROM.

Pour éviter que le code utilisant l'API d'AROM fasse référence à une implémentation spécifique, la référence à l'implémentation utilisée est contenue dans un fichier de configuration. De ce fait, en modifiant ce fichier de configuration, il devient possible d'exécuter une même application (ou un même code) basée sur l'API d'AROM avec des implémentations différentes du système.

Chapitre 4

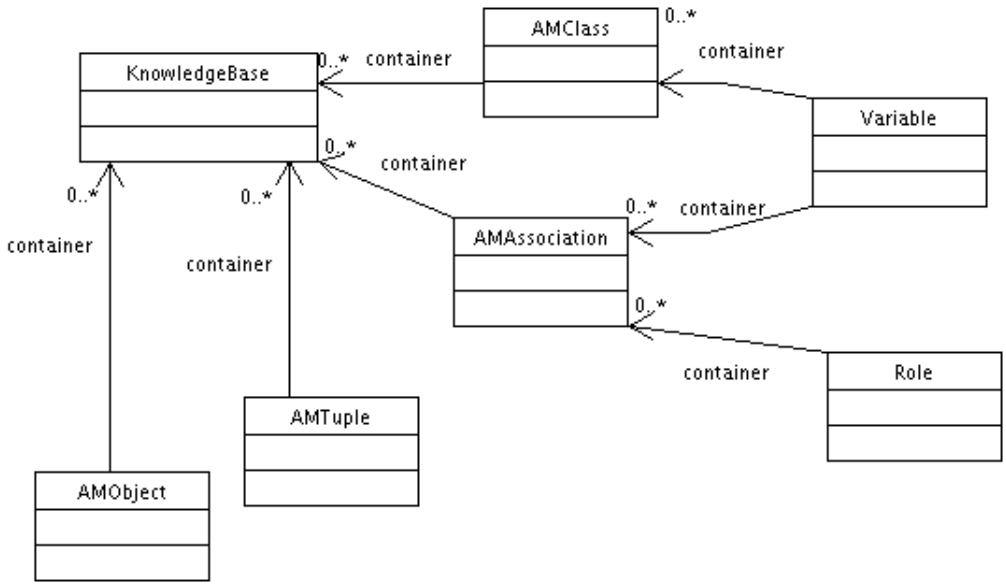
Représentation des entités du système

4.1. Entités AROM

Le modèle AROM propose à l'utilisateur ce que l'on nomme des entités qui sont les éléments constitutifs des bases de connaissances AROM. C'est en créant et en supprimant des entités que l'utilisateur définit une base de connaissances. Les entités peuvent être complexes, autrement dit être elles-mêmes définies par d'autres entités. C'est par exemple le cas d'une classe qui est décrite par un ensemble d'entités plus élémentaires : les variables. Dans ce cas, la durée de vie d'une entité dans le modèle est celle de l'entité qui la contient. Lorsqu'une entité complexe est supprimée, les entités qui la composent sont également supprimées.

La relation de composition qui s'établit entre les entités du système est organisée de la manière suivante :

Figure 4-1. Relations de composition entre les entités AROM



4.1.1. Suppression des entités

Lorsque l'on parle de supprimer une entité AROM il ne s'agit pas en réalité d'une suppression effective de l'objet informatique représentant l'entité AROM. L'opération de suppression est à entendre comme une suppression logique : lorsqu'une entité a été *supprimée*, il n'existe aucun moyen de la réintégrer dans le système AROM. Cependant l'objet informatique représentant cette entité peut continuer à exister dans la machine virtuelle Java. Cet objet peut donc répondre aux appels de méthode, mais son comportement n'est plus garanti : ainsi tenter de changer le domaine de valeurs d'un slot qui a été préalablement supprimé de la classe ou de l'association à laquelle il appartenait peut conduire le système à lever une exception.

Lorsque l'on supprime une entité AROM, toutes les entités qu'elle contenait sont également supprimées. De plus, lorsque l'entité supprimée est un *AMObject*, instance de classe AROM, les *AMTuples* qui référencaient cet objet sont également supprimés. En effet, un *tuple* modélise un lien entre plusieurs *objets*, si l'un d'eux vient à être supprimé le tuple n'a plus de raison d'exister.

4.2. Propriétés des entités AROM

Les entités du modèle AROM possèdent des propriétés. La liste des propriétés décrivant une entité découle de la nature même de l'entité. Toutefois, les entités AROM possèdent toutes les deux propriétés suivantes :

- *Un identificateur*. Chaque entité possède un identificateur unique dans le système. Un identificateur est attribué à une entité de manière automatique par le système AROM lui-même.
- *Une documentation*. Les entités du système constituent les unités élémentaires au travers desquelles un utilisateur du système AROM va représenter des connaissances. Pour rendre cette connaissance intelligible, il est indispensable de pouvoir annoter tous les éléments de connaissance représentés dans le système. C'est pourquoi chaque entité AROM est susceptible d'être documentée.

4.3. Espaces de nommages

Une entité du modèle AROM qui est composée d'autres entités, se comporte comme un *espace de nommage* lorsque les entités qui la composent sont des entités *nommées*. L'espace de nommage définit l'ensemble des noms qui peuvent être utilisés par les entités appartenant à cet espace et les conventions de nommage qui s'appliquent dans cet espace.

Chaque entité appartenant à un espace de nommage est identifiée par un nom unique dans cet espace. Il est donc possible de rechercher une entité via son nom à partir de son espace de nommage. La définition des espaces de nommage est récursive : une entité se comportant comme un espace de nommage peut contenir des entités qui sont elles-même espaces de nommage pour d'autres entités.

Dans la plupart des cas, la relation entre l'espace de nommage et les entités nommées appartenant à cet espace est une relation equivalente à la relation de composition qui lie les entités complexes aux entités qui la composent.

4.3.1. Spécialisation d'un espace de nommage

Losque l'entité qui se comporte comme un espace de nommage est un entité spécialisable, l'espace de nommage qu'elle définit n'est plus limité aux seules entités qui composent l'entité spécialisable.

La définition que nous avons donnée de l'espace de nommage d'une entité est donc complété par la définition suivante dans le cas des entité spécialisables :

- *l'espace de nommage étendu* d'une entité spécialisable est l'ensemble des entités nommées qui la composent, auxquelles viennent s'ajouter les entités héritées par la relation de spécialisation.

4.4. Spécialisation d'entités

Certaines entités AROM peuvent être spécialisées. Une entité qui en spécialise une autre est appelée *spécialisation*. A l'inverse une entité qui possède des spécialisations est appellée *généralisation* de ces entités. Lorsqu'une entité est la spécialisation d'une autre entité, elle hérite des propriétés et de l'espace de nommage de celle-ci.

Selon l'entité considéré et les propriétés qu'elle définit, les règles de spécialisation peuvent varier. Par conséquent, ces règles seront donnés dans les chapitres traitant spécifiquement des propriétés (slots, facettes, ...).

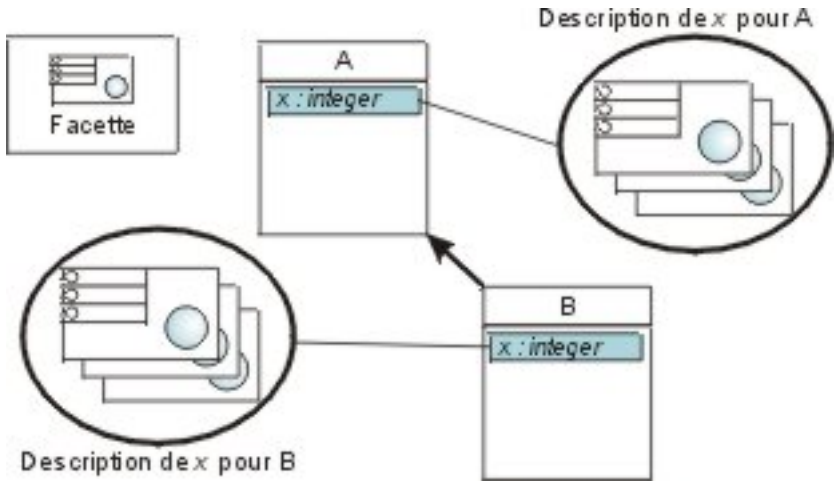
4.5. Classes et Associations AROM

4.6. Slots

Un slot est l'entité AROM utilisée pour décrire l'intention d'une classe ou d'une association. Un slot possède les caractéristiques suivantes :

- Un slot est une entité *nommée*. L'espace de nommage du slot est la structure, classe ou association, dans laquelle le slot est déclaré, étendu par les espaces de nommage des super-structures de cette structure.
- Un slot peut être *valué*. Une valeur peut être attachée à un slot si elle respecte les contraintes sur les valeurs admises par le slot.
- Un slot possède une *description*. La description d'un slot est l'ensemble des facettes qui caractérisent le slot (voir Figure 4-2).

Figure 4-2. Slots et description de slots



Dans cet exemple, on voit bien que pour une hiérarchie de structures donnée, il n'existe qu'UN seul slot et AUTANT de descripteurs de slot que de structures.

Ces trois caractéristiques sont communes à tous les slots. Cependant, les différents *types* (ou les différentes familles) de slot se distinguent par :

- Les règles de spécialisation qui s'appliquent aux slots. Les slots sont en effet utilisés pour décrire l'intention des structures. Les structures sont des entités spécialisable et la spécialisation peut en particulier être réalisée au travers de la spécialisation des slots qui décrivent la structure. Cependant, tous les slots n'acceptent pas la spécialisation de leur description (voir Section 4.6.2).

- la nature de leur description. La description d'un slot est un ensemble de facettes attachées à ce slot. Les slots peuvent être décrits par des ensembles de facettes différents, mais ils ont en commun d'être tous décrits au minimum par une facette de type.
- les règles de valuation du slot (ce que l'on nomme le *contexte de valuation* du slot). On a dit de manière un peu trop simplifiée qu'un slot admettait une valeur. En réalité, la définition du slot est attachée à une structure, mais c'est le contexte de valuation qui définit les entités pour lesquelles le slot en question sera "visible" et par conséquent les entités qui pourront attribuer une valeur au slot. Un slot admet donc un contexte de valuation hors duquel il est impossible de lui affecter une valeur : l'exemple le plus illustratif est celui des slots d'instance et des slots de structure (i.e slot de classe ou slot d'association) qui ont pour contexte de valuation respectivement les instances ou les structures.(voir Section 4.6.1 et Section 4.6.2).

4.6.1. Slots d'instance

Un slot d'instance est défini dans une structure AROM et il est valué pour chaque instance de cette structure. Un slot d'instance participe à la spécialisation de la structure dans laquelle il est défini en autorisant la spécialisation de sa description.

Considérons la figure Figure 4-2. Le slot *x* déclaré dans une structure *A* est spécialisé dans une sous-structure de *A*: *B*.Il s'agit bien dans *A* et *B* de la même entité (du même slot) *x*. A ce titre, le slot *x* est représenté dans le système AROM par une unique entité. Toutefois, un slot d'instance autorisant la spécialisation de sa description, le slot *x* possède plusieurs descriptions : une à chaque niveau de la hiérarchie des structures dans laquelle il apparaît.

4.6.2. Slots statiques

Actuellement non implémenté

Un slot statique est défini dans une structure AROM et il est valué pour la structure elle-même et pour ses sous structures. A l'inverse des slots d'instance, un slot statique ne participe pas à la spécialisation de la définition d'une structure. La description d'un slot statique est donc unique et elle ne peut être spécialisée.

4.7. Facettes

4.7.1. Definition

Les facettes permettent de représenter des propriétés dans AROM. Elles peuvent apparaître à différents niveaux dans AROM :

- La facette de documentation peut être définie par toute entité AROM.
- La facette de type permet de décrire les slots.

- La facette d'inférence permet de spécifer des valeurs ou méthodes d'inférence pour les variables.
- La facette de contrôle d'instanciation permet de controler l'instanciation des structures AROM.

Par conséquent, les facettes peuvent être liées à des entités spécialisables, comme les types des slots, ou non spécialisables comme la documentation des instances. L'état d'une facette liée à une entités spécialisable dépend donc non seulement des modificateurs qui lui ont été appliqués, mais également de l'état des facettes dont elle est la spécialisation (la figure du chapitre *Spécialisation d'une facette de type* illustre cela dans le cas d'une facette de type).

La modification de *l'état d'une facette* se fait en appliquant ou en annulant un *modificateur de facette*. La facette, si elle reconnaît ce modificateur, modifie son état interne pour refléter la modification demandée.

4.7.2. Facettes de Slots

Un slot est décrit par un ensemble de *facettes* (type, inférence, documentation, etc ...). Ainsi, une variable AROM est décrite par trois facettes : une facette de documentation, une facette de type et une facette d'inférence, alors que le role n'est définit que par les facettes de documentation et de type. Les slots sont décrits par des descripteurs organisés selon la même hiérarchie que les structure pour lesquelles ils décrivent le slot. Par conséquent, une facette étant liée à la description d'un slot, il existe alors plusieurs objets facettes différents pour un même slot : un pour chaque description du slot (donc un pour chaque niveau de la hiérarchie des structures où le slot apparaît).

La modification de la description d'un slot se fait donc par l'application (l'annulation) de modificateurs de facette. La facette ne fait que modifier son état interne, par contre, il est possible d'obtenir la liste des modificateurs appliqués auprès des descripteurs.

Pour résumer la situation on peut dire que c'est le slot qui décide s'il est possible ou non de spécialiser sa description en spécialisant ses facettes, mais chaque facette dicte les règles qui régissent cette spécialisation. Ces règles de spécialisation des facettes dépendent naturellement de la nature des facettes.

4.7.3. Spécialisation d'une facette de documentation

La documentation d'une entité AROM peut-être divisée en deux parties. Une première partie au format *texte* et une seconde au format *URL*. Seul l'un des deux formats, ou les deux simultanément, peuvent être spécifiés pour une seule et même facette. Dans tous les cas, les modificateurs et les règles de spécialisation sont identiques.

Pour les entités spécialisables, La facette de documentation peut être différente d'une niveau à un autre. Ceci permet d'attribuer une documentation de plus en plus précise tout au long de la hiérarchie de spécialisation. La règle de spécialisation de la facette de documentation impose qu'une facette de documentation masque la ou les documentations qu'elle spécialise. [lorsqu'un modificateur lui est appliqué.]

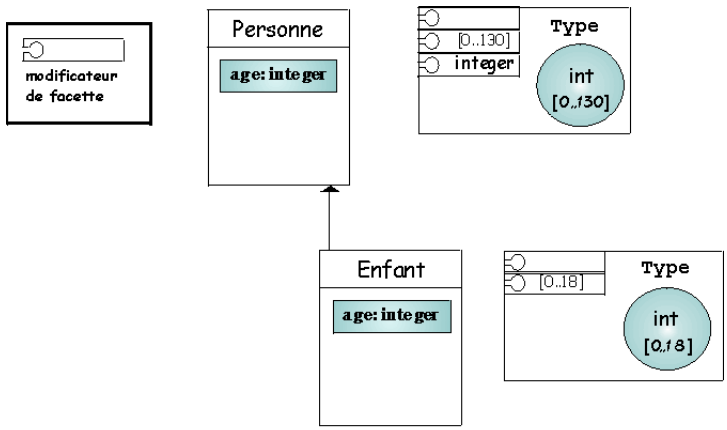
Il serait pourtant intéressant de permettre à l'utilisateur du système AROM de choisir pour chaque facette (ou pour chaque entité (ex : pour le slot x, mais pas pour le slot y) ou encore pour chaque type d'entité spécialisable (ex : pour les slots, pour les classes mais pas pour les associations) s'il

souhaite qu'une facette de documentation masque les documentations dont elle hérite ou s'il souhaite au contraire qu'une facette de documentation se comporte comme un complément de la documentation dont elle hérite. Dans le second cas, la documentation d'une entité serait alors la concaténation des documentations héritées par cette entité.

4.7.4. Spécialisation d'une facette de type

L'état d'une facette de type dépend de la facette héritée et des modificateurs de facettes appliqués localement. Ainsi dans la figure ci dessous, l'état de la facette de type du slot *x* pour la structure *B* dépends non seulement de l'état de cette même facette pour la structure *A* mais aussi du modificateur appliqué à *x* pour *B*.

Figure 4-3. Spécialisation d'une facette de type



L'application d'un modificateur de facette de type est acceptée même s'il n'a pas de réelle conséquence sur l'état de la facette. En effet, si l'on considère l'exemple ci-dessus et que l'on applique le modificateur [-1..100] à *x* pour *B*, la facette de Type résultante pour *B* sera la même que celle pour *A*: [0..10]. Les cas qui entraînent un refus de l'application d'un modificateur de facette de type sont décrits ici.

- La facette de Type résultante est l'ensemble vide. Il faut également que cette règle soit vérifiée après répercution de l'application du modificateur aux facettes des sous-structures.
- La valeur du slot considéré pour une instance existante devient invalide.

4.7.5. Spécialisation d'une facette de contrôle d'instanciation

Cette facette est liée aux Structures AROM. En effet, elle définit des règles premettant de contrôler les instanciations des classes et associations. On trouve dans cette facette, entre autres, la multiplicité. La multiplicité, bien qu'associé aux rôles, permet de contrôler l'instanciation des associations. En effet, une multiplicité de 1 pour un role r_1, signifie qu'une seule valeur est possible pour r_1 lorsque tous les autres

rôles sont valués. Par conséquent, la création d'un tuple peut être interdite aux vues de la multiplicité associée à un des rôles de son association. Plus d'information est donné dans le chapitre *Représenter des connaissances avec AROM*. Par la suite, d'autres aspects, tel que le singleton, pourraient venir enrichir cette facette.

L'état de cette facette, plus précisement de la multiplicité, dépend de la facette héritée et des modificateurs de facettes appliqués localement. L'application de modificateurs de multiplicité est acceptée même si elle n'a pas de réelle conséquence sur l'état de la facette. Ainsi, si une multiplicité de [1..5] est définit à un niveau n, l'application d'un modificateur de multiplicité [1..10] à un niveau n+1 n'entraînera aucune modification de la multiplicité, même au niveau n+1. Par contre, l'application d'un modificateur de multiplicité [8..10] provoque un erreur puisque la multiplicité résultante est invalide.

4.7.6. Spécialisation d'une facette d'inférence

L'état de la facette d'inférence dépend de la facette héritée et des modificateurs appliqués localement. Une facette d'inférence définit actuellement 2 modes d'inférences : la valeur par défaut et l'attachement.

Chacun des modes de la facette d'inférence masque le même mode d'inférence dont elle a hérité lorsqu'un modificateur de facette lui est appliqué. Ainsi, si l'on considère un descripteur qui hérite d'une facette d'inférence définissant les deux modes et si un modificateur pour l'attachement lui est appliqué, la facette résultante définira toujours les deux modes d'inférences: la valeur par défaut héritée et l'attachement procédural redéfinit.

4.8. Les instances AROM

Nous avons vu que les structures AROM définissent des intentions. C'est à dire que chaque structure spécifie les propriétés qui la caractérisent au travers de slots. De plus les structures sont organisées hiérarchiquement et une structure hérite de l'intention définie par sa structure mère.

L'*intention* permet donc d'identifier les propriétés communes aux individus d'un même groupe, la structure. En parallèle, on définit l'*extension* d'une structure qui correspond à l'ensemble des individus qui appartiennent effectivement à la structure. Ces différents individus, les *instances*, doivent satisfaire les propriétés de l'intention afin d'appartenir à l'extension d'une structure mais l'inverse n'est pas nécessairement vrai. En effet, même si une entité satisfait les propriétés de l'intention d'une structure, elle ne fait pas pour autant partie de son extension, il faudra qu'elle ait été spécifiquement ajouté à cette extension.

Dans AROM, les structures se divisent en Classes et en Associations. Les *instances* représentent les individus des structures en général, les *objets* sont les instances des classes et les *tuples* représentent les instances des associations.

4.8.1. Les Objets AROM

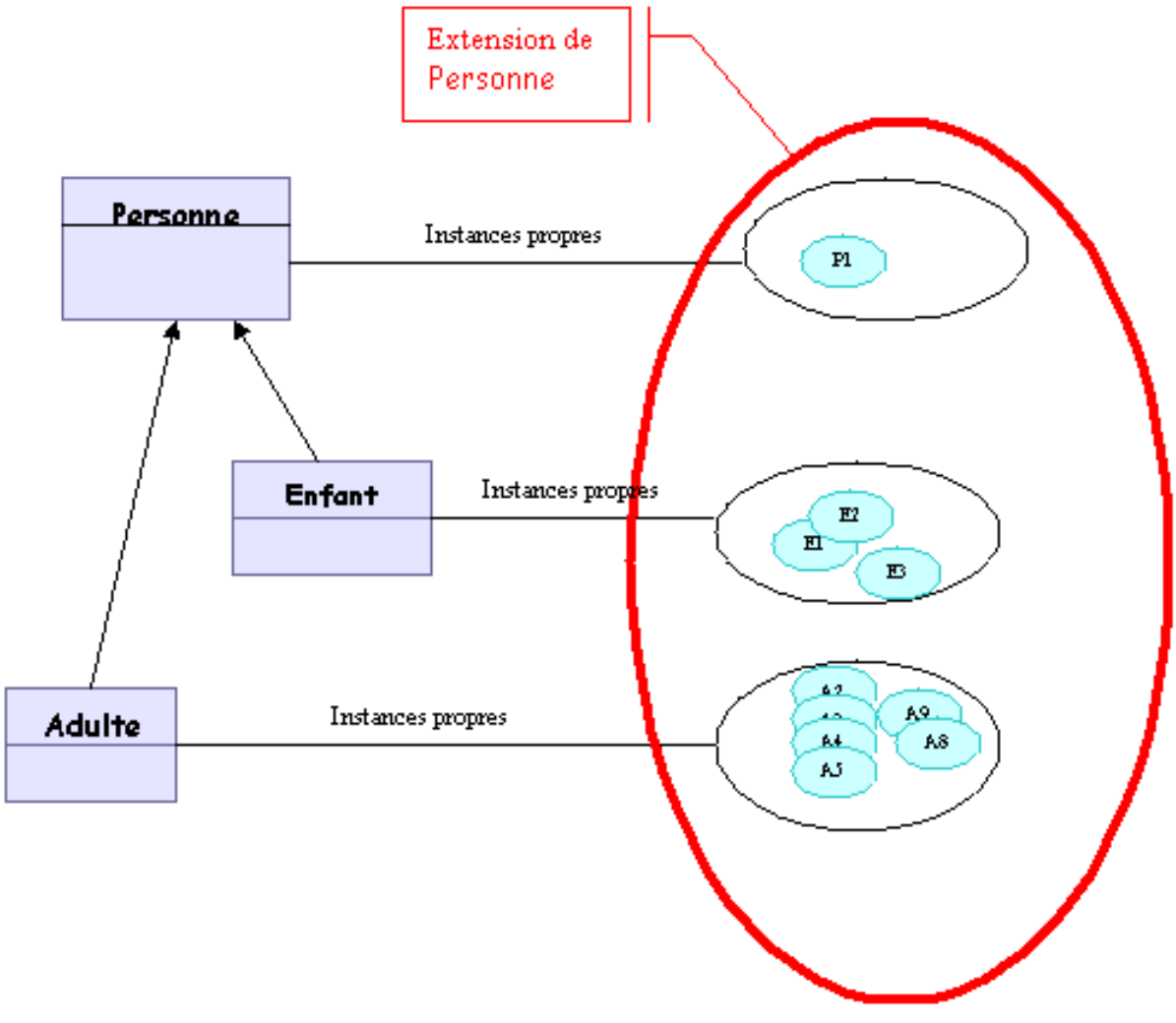
Un objet AROM est rattaché à une classe spécifique dont l'intention n'est définie qu'au travers de variables. Les objets AROM acceptent des valeurs pour ces variables. Ainsi, un objet qui appartient à une structure définit un ensemble de couples <variable/valeur>. Toutes les variables de la structure ne

sont pas obligatoirement valuées mais elles doivent être reconnues. C'est à dire que l'objet doit pouvoir accepter à tout moment une valeur pour cette variable.

Comme nous l'avons vu dans les sections précédentes, les structures AROM sont des entités spécialisables. Aux vues des règles de spécialisation définies dans le modèle AROM, on peut dire que les objets d'une classe sont également objets de ses super-classes. Ainsi, dans l'exemple ci-dessous, les objets des classes `Adulte` et `Enfant` sont également des objets de la classe `Personne`.

On definit donc deux niveaux d'appartenance des instances aux structures, voir figure *Extension d'une structure*. En effet, l'extension de la classe `Personne`, dans l'exemple ci-dessous, est constitué de l'ensemble des instances qui lui sont propres ainsi que de l'ensemble des instances propres à chacune de ses sous-classes, `Adulte` et `Enfant`. Par instances propres à la structure, on entends instances qui sont directement rattachées à cette strutcure.

Figure 4-4. Extension d'une structure



4.8.1.1. Migration

La migration permet de faire évoluer un objet AROM dans la hiérarchie des classes. Ainsi, on recherche la classe la plus spécialisée qui accepte l'objet dans son ensemble *d'instances propres*. Tout objet AROM devant appartenir à une structure, le changement de classe d'appartenance d'un objet sera effectué en une seule opération. Comme nous l'avons vu ci-dessus, pour qu'un objet soit accepté par une classe il faut qu'il satisfasse les propriétés de la nouvelle classe et notamment, qu'il accepte (ou puisse accepter) une valeur pour chacune des variables de cette classe.

Plus précisement, pour chacune des variables déjà connues de l'objet, il faut que la nouvelle classe

définisse une variable:

- ayant le même identifiant, c'est à dire le même nom.
- ayant un CType compatible avec celui définit dans la structure de départ. Ceci est vrai si il est possible de déterminer un CType ancêtre commun à ces deux CTypes.

Néanmoins, la nouvelle classe peut définir de nouvelles variables qui ne seront pas valuées par l'objet.

4.8.2. Les Tuples AROM

On retrouve les même caractéristiques pour les tuples que celles décritent ci-dessus pour les objets. Par contre, l'intention des associations AROM est définie au travers de variables et de roles. Par conséquent, les tuples doivent non seulement définir des valeurs pour les variables, comme c'est le cas pour les objets AROM, mais ils doivent également référencer les objets pour les différents roles. En effet, les roles permettent d'identifier les classes qui sont mises en jeu dans une association donnée, les valeurs des roles sont donc des objets de ces classes. De plus, dans un tuple tous les roles doivent nécessairement être définis. Si l'on reprend l'exempele donné ci-dessus et que l'on considère l'association `Parent` reliant les classes `Adulte` et `Enfant`, les tuples de cette association devront définir un objet de la classe `Adulte` et un objet de la classe `Enfant`.

4.9. Mécanisme de notification

La notification permet à l'utilisateur de l'API d'AROM d'être averti lorsqu'un changement survient au niveau d'une base de connaissances. Il y a deux notions à prendre en compte. Tout d'abord, il faut identifier quels sont les changements susceptible d'être notifiés. Ensuite, il est important de définir à quel niveau l'évènement de notification est généré et quel forme cet évènement prends.

4.9.1. Propriétés notifiables

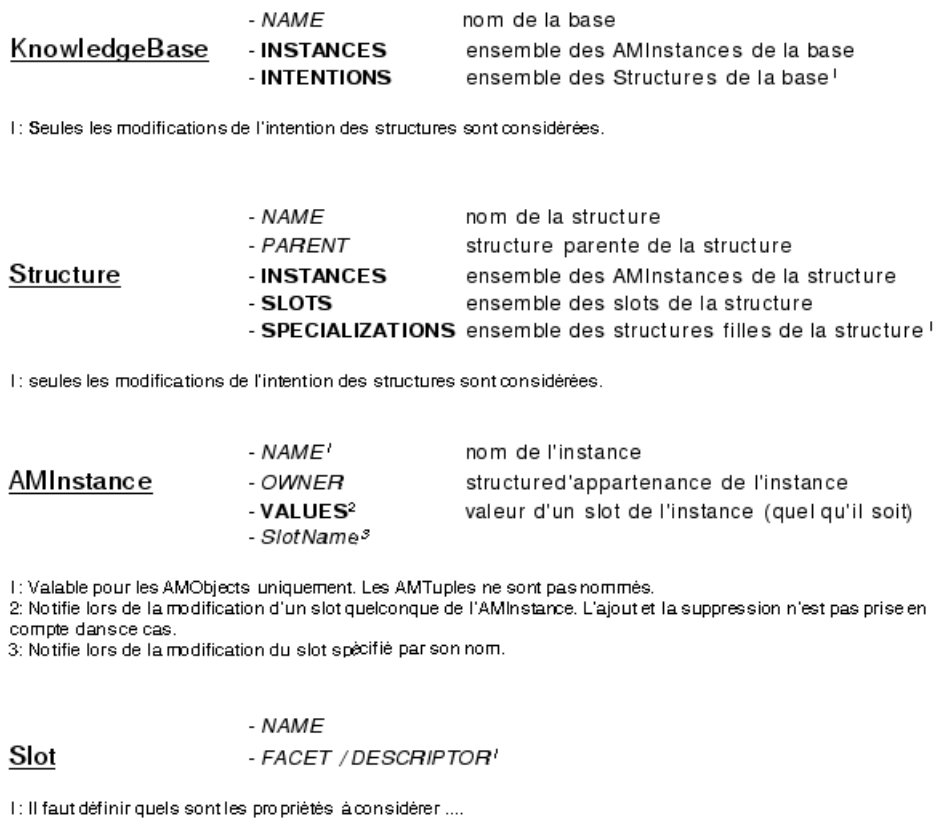
Les notifications peuvent être générées dans AROM lors :

1. du changement du *Nom* d'une entité AROM, quelle qu'elle soit.
2. du changement de la valeur d'un des *Slots* (variable ou role) d'une *Instance* (object ou tuple). Il est possible de prévenir un utilisateur lors du changement d'un slot précis ou pour tout changement de valeurs d'une instance. Par contre, lorsque la définition d'un slot est supprimée/ajoutée au niveau de la structure, il n'y a pas de notification faite au niveau des instances.
3. du changement de la structure d'appartenance d'une *Instance*.
4. du changement de la valeur d'une *Facette* d'un *descripteur* de slot. Toutes les facettes ne génèrent pas forcement une notification, ce n'est peut être pas utile pour la facette de documentation par exemple. *Ne sera pas implémenté dans un premier temps.*

5. de l'ajout/suppression/modification d'un élément de l'ensemble des *Instances* d'une *Structure*. Une instance est dite modifiée lorsqu'elle notifie elle-même un changement, voir point 1, 2 et 3. C'est donc l'extension (l'ensemble des instances) de la structure qui est pris en compte ici. Par ensemble des instances on considère les instances propres à la structure et les instances de ses sous structures.
6. de l'ajout/suppression/modification d'un élément de l'ensemble des *Slots* d'une *Structure*. Dans ce cas, c'est une modification de l'intention de la structure qui est prise en compte. Un *Slot* est dit modifié lorsqu'il est lui-même susceptible de notifier un changement, voir point 1 et 4.
7. de l'ajout/suppression/modification d'un élément de l'ensemble des *Spécialisations* d'une *Structure*. Seule la modification des intentions des sous-structures sera notifiée. En effet, la modification de l'extension d'une sous-structure est déjà prise en compte par la structure parente (voir le point 5).
8. de l'ajout/suppression/modification d'un élément de l'ensemble des *Instances* d'une *KnowledgeBase*. La même politique que celle définit pour les structures (point 5) est utilisée.
9. de l'ajout/suppression/modification d'un élément de l'ensemble des *Intentions* d'une *KnowledgeBase*. Dans ce cas, seules les modifications des intentions des structures sont notifiées. En effet, les modifications sur les extentions sont notifiées via l'ensemble des *Instances* de la base, point précédent.

Comme on peut le voir dans la description faite ci-dessus, il existe deux types de propriétés *écoutables* dans AROM. Les propriétés simples, d'une part, qui correspondent aux attributs directement valués des entités AROM, tels que le nom ou la valeur d'un slot d'une *Instance*; Ces propriétés sont en *italique* ci-dessous. Les propriétés composites, d'autre part, qui représentent des ensembles d'entités ou de propriétés, pouvant être elles-mêmes composites; Ces propriétés sont représentées en **gras** dans les schémas qui suivent. C'est par exemple l'ensemble des `AMInstance` d'une structure ou l'ensemble des `Intention` d'une base de connaissances. Les schémas qui suivent détaillent les propriétés écoutables pour les différentes entités AROM.

Figure 4-5. Propriétés dans AROM



En ce qui concerne les slots, trois possibilités sont offertes:

- Soit on considère que ce sont les `Slot`s qui sont écoutables. Dans ce cas, une notification est lancée lorsque le nom de ce slot change ou lorsqu'un de ses *descripteur*s change donc lorsqu'une *facette* décrivant un descripteur est modifiée.
- Soit on considère que ce sont les *descripteurs* qui sont écoutables et par conséquent une notification est lancée uniquement lorsqu'une modification est réalisée sur une facette décrivant le *descripteur*. Un changement du `Slot` associé au *descripteur* n'entraîne pas de notification.
- Il est également envisageable d'autoriser les deux notifications citées ci-dessus. Par contre, en ce qui concerne la propriété composite SLOTS de `Structure`, seules les modifications sur le `Slot-Descriptor` de la structure en question sont intéressantes, en plus d'un changement de nom du slot. En effet, une modification du `SlotDescriptor` d'une sous-structure est notifiée via la propriété SPECIALIZATIONS mais elle n'a pas de conséquence sur l'Intention de la structure considérée.

4.9.2. Evènements

Pour être notifié des modifications d'une propriété, il est nécessaire de s'enregistrer comme *listener* auprès de l'objet définissant cette propriété. Pour être *listener*, il faut implémenter des interfaces spécifiques qui permettent de recevoir les évènements associés.

Les évènements liés aux propriétés simples permettent de signaler que la valeur de la propriété considérée a changé. Les évènements liés aux propriétés composites permettent, quant à eux, de signaler que l'ensemble des données identifié par la propriété a changé. C'est à dire qu'un élément de l'ensemble a été ajouté ou supprimé ou encore qu'il a été modifié. Les éléments d'une propriété composite, *Composition*, pouvant être eux-mêmes des propriétés composites, la modification d'un élément de l'ensemble peut être dûe à la modification d'une propriété simple, *p*, d'une entité, *s1*, appartenant à une propriété composite de niveau x, *Composition_x*.

```
E: Composition {e1, e2, ...,en }
                  |
                   -> Composition 2 {f1, f2, .., fm}
                                     |
                                      -> ...
                                         ...
                                      |
                                       -> Composition x {s1, s2,...,sy}
                                                          |
                                                           ->Element dont une
                                                             propriété simple (p)
                                                             a été modifié.
```

Dans le cas d'une propriété simple, la description de la modification (l'entité modifié, le nom de la propriété modifiée ainsi que l'ancienne et la nouvelle valeur) est transmise à l'utilisateur via un *Evenement* simple tel qu'on les trouve pour les propriétés des JavaBean.

Pour ce qui est des propriétés composites, lors de l'ajout ou de la suppression d'un élément, l'information transmise doit préciser l'entité modifié, c'est à dire celle contenant la propriété composite, le nom de la propriété modifiée, le type de l'opération effectuée (ajout, suppression), et l'élément ajouté ou supprimé.

Par contre, lors de la modification de l'un des éléments d'une propriété composite, il faut transmettre un évènement définissant l'entité modifié, donc celle contenant la propriété composite, le nom de la propriété, l'élément modifié et la modification effectué. La modification elle-même est transmise via un nouvel *Evenement*. Dans le cas où l'élément modifié est une propriété simple, le même évènement que celui décrit dans le paragraphe précédent est transmis. Par contre, dans le cas où l'élément modifié est lui-même une propriété composite, il faut transmettre de nouveau un évènement définissant l'élément dans la propriété composite qui a été modifié et la modification effective.

Même si les informations à transmettre ne sont pas identiques dans le cas d'une modification et dans le cas d'un ajout ou d'une suppression, c'est un même type d'*Evenement* qui est émis.

Chapitre 5

Module de types

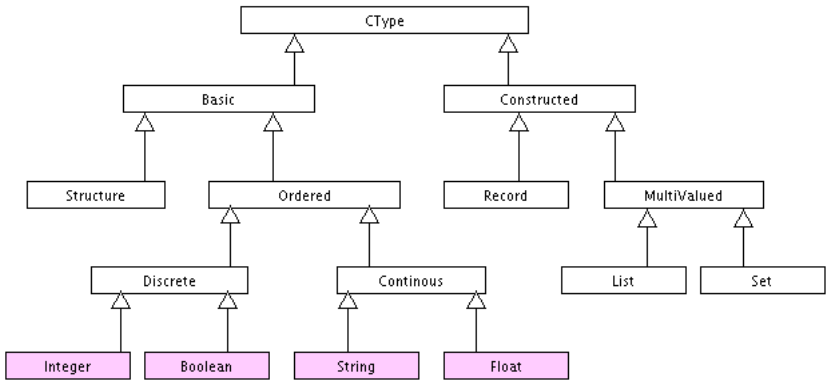
5.1. C-Types et δ-Types

Le module de types définit l'ensemble des types de données qui pourront être utilisés dans une base de connaissances AROM, pour décrire l'intention d'une structure par exemple. Des opérations sur ces types de données sont également proposées par le module de types. De la même manière que dans le système de types METEO [Cap95], le module de types d'AROM considère deux niveaux de représentation des types :

- Un *C-type* (Classe de types) représente l'ensemble, fini ou non, des valeurs partageant une même structure, tel que l'ensemble des réels ou l'ensemble des chaînes de caractères. Chaque C-type définit des opérations qui lui sont applicables.
- Les *δ-types*, quant à eux, permettent de représenter des sous-ensembles des C-types. Chaque δ-type est associé à un C-type qui représente le type principal de celui-ci. Les informations relatives aux données se trouvent donc dans le C-type. Par contre, les δ-types contiennent des informations concernant la restriction du domaine défini par le C-type. Pour un même C-type il peut donc exister plusieurs, voire une infinité, de δ-types. Pour les opérations concernant les δ-types, celles-ci dépendent du type principal des valeurs manipulées, c'est pourquoi ces opérations sont définies au niveau du C-type et non au niveau du δ-type. Par exemple, les méthodes d'intersection ou d'union entre deux δ-types sont définies au niveau des C-types.

Des classes de C-types sont prédéfinies et organisées hiérarchiquement afin de regrouper les C-types selon leurs caractéristiques.

Figure 5-1. Hiérarchie des classes de `CTypes`



A un premier niveau les C-types sont divisés en C-types *basics* et *construits*. On retrouve ensuite :

- Les *C-types de structures* qui permettent de représenter les Structures AROM.
- Les *C-types ordonnés* qui représentent les C-types acceptant des valeurs ordonnées. On peut distinguer les C-types ordonnés discrets et les C-types ordonnés continus.
- Les *C-types multivalués*, eux-même divisés en *C-types List* et *C-types Set*, qui représentent des collections de valeurs de même type.
- Les *C-types records* qui représentent des ensembles de couples *étiquettes/types*.

Par défaut, seuls les C-types simples, en couleur dans le diagramme ci-dessus, suivants sont définis dans le module de types et peuvent être utilisés afin de typer les variables AROM. Ils n'acceptent qu'une seule instance de C-type, c'est pourquoi ils peuvent êtr prédéfinis et qu'ils sont accessibles via leur nom :

- `string` qui représente l'ensemble des chaînes de caractères.
- `boolean` qui représente l'ensemble des booléens.
- `float` qui représente l'ensemble des réels.
- `integer` qui représente l'ensemble des entiers.

Des C-types construits peuvent également être utilisés dans AROM. Les C-types construits principaux sont les ListCT et les SetCT qui représentent respectivement des listes et des ensembles de δ-types. En effet, les C-types construits le sont sur des δ-types et non sur des C-types afin d'affiner les C-types offerts et de simplifier les tests d'appartenance. En effet, pour représenter une Date, le C-type *record <jour/integer [0..30]; mois/integer [0..11]; année/integer>* est plus précis que <jour/integer; mois/integer; année/integer>.

A une variable est donc associé un C-type qui permet de spécifier le comportement et la structure de celle-ci. Mais afin de décrire l'ensemble des valeurs autorisées pour la variable, c'est le δ-type qui est

utilisé. Si aucune restriction n'a été appliquée à la variable, l'ensemble dénoté par le δ-type sera le même que celui dénoté par le C-type. Par contre, lorsqu'une restriction des valeurs est réalisée auprès de la variable, le C-type reste inchangé, c'est le δ-type qui est modifié.

La restriction d'un ensemble de valeurs est réalisée par l'intermédaire de deux descripteurs: *set* ou *interval*. Le nouvel ensemble de valeurs associé à une variable correspondra au domaine spécifié par le decripteur d'intervalle ou d'ensemble. Un seul de ces descripteurs est pris en compte. L'application d'un descripteur entraînera la suppression du descripteur précédent.

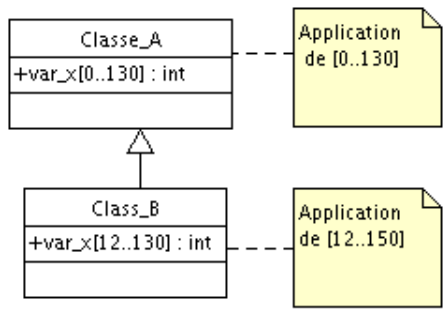
Dans l'exemple qui suit, le salaire d'un ingénieur niveau 2 est compris entre 175037,44 Fr et 288378,74 Fr. Si l'on applique à cette variable le descripteur *set : {153691,25; 175037,44; 186321,45; 21035678 }* le nouveau domaine sera l'ensemble {153691,25; 175037,44; 186321,45; 21035678} et non l'union de cet ensemble et de l'intervalle [175037,44..288378,74].

Exemple 5-1. Resctriction de domaine

```
Class: Ingenieur_niveau_2
variables:
 variable: salaire
   type: float
   domain: [175037,44..288378,74]
```

Par contre, dans le cadre d'une hiérarchie de Structures AROM, une variable peut se voir restreinte par l'application de plusieurs descripteurs. En effet, dans l'exemple du schéma *Application de descripteurs*, l'ensemble des valeurs de la variable `var_x` pour la structure `Class_B` est le résultat de l'application des descripteurs *interval* [0..130] et *interval* [12..150] soit l'intervalle [12..130]. Maintenant, si au niveau de la structure `Class_B` le descripteur *set* {-2, 12,65,119,456} était appliqué, le descripteur *interval* [12..150] serait annulé et le résultat serait l'ensemble {12,65,119}.

Figure 5-2. Application de descripteurs



5.2. Domaines associés aux δ-Types

A chaque C-type correspond donc un ensemble de δ-types qui représentent des sous-ensembles de valeurs. Un δ-type est donc associé à un type de base, un `CType`, et à une restriction du domaine définit sur le C-type. De façon générale, un domaine peut-être représenté par un ensemble de valeurs et/ou d'intervalles de valeurs autorisées et un ensemble de valeurs et/ou d'intervalles de valeurs interdites. Un δ-type est donc en fait associé à deux domaines, d_a et d_i représentant respectivement le domaine des valeurs autorisées et celui des valeurs interdites.

- d_a accepte soit un ensemble de valeurs, soit un ensemble d'intervalles de valeurs soit la valeur `ALL` représentant l'ensemble des valeurs du C-types.

- d_i accepte soit un ensemble de valeurs, soit un ensemble d'intervalles de valeurs soit la valeur `EMPTY` pour signifier qu'aucune valeur n'est interdite.

Les valeurs `ALL` pour d_i, `EMPTY` pour d_a n'ont pas de sens puisqu'ils décrivent alors un ensemble vide.

Dans le cas où d_a n'est pas `ALL`, d_i ne peut définir qu'un sous ensemble de d_a. En effet, il n'y a aucun intérêt à interdire une valeur qui n'est, de toute façon, pas autorisée...

Selon le δ-type, et plus précisement le C-type duquel est issu le δ-type, la définition du domaine associé est différente. Si le C-type est un :

- C-type *ordonné*, la définition des δ-types associés est de la forme : <ordered; [d_a]>. Tout ensemble de valeurs autorisés ou interdites peut être traduit en un ensemble d'intervalles dont l'union représente le domaine de valeurs autorisées. Seuls les ensembles d'intervalles sont donc intéressants pour les C-types ordonnés. De plus, lorsqu'un intervalle de valeurs *interdites* est ajouté au domaine, il est facile de modifier l'ensemble des intervalles *autorisés* pour prendre en compte cette interdiction.

  Par conséquent, dans le cas des C-types ordonnés, d_i est toujours `EMPTY`, c'est pourquoi il n'est pas représenté dans la définition des δ-types ordonnés.

- C-type *construit*, la définition des δ-types associés est de la forme : <constructed; [T_c; d_a,d_i]>. T_c est l'ensemble des δ-types des différentes étiquettes. Les ensembles d_a et d_i ne peuvent être définis que par des ensembles non ordonnés de valeurs, puisque les types *construits* ne sont pas ordonnés. Si l'ensemble d_a est `ALL`, l'ensemble d_i peut-être `EMPTY` ou un ensemble de valeurs. Par contre, si d_a est un ensemble de valeurs, d_i est `EMPTY`. En effet, seules des valeurs déjà autorisées peuvent-être interdites donc si une valeur vient à être interdite, il suffit de la supprimer des valeurs autorisées et d_i reste `EMPTY`.

  Dans le cas où d_i est un ensemble de valeurs et d_a est `ALL`, et qu'une valeur est positionnée comme autorisée, alors d_i est réduit à `EMPTY`. *C'est d_a qui est prioritaire sur d_i.*

  Pour les CTypes *multivalués*, la cardinalité est également à prendre en compte dans le domaine. La définition des δ-types *multivalués* est alors <multivalue; [T_c; d_a,d_i, [a,b]]>.

5.3. Relation de sous-typage dans le module de types

5.3.1. Sous-typage des C-types

Une hiérarchie structurelle, illustrée dans la figure *Hiérarchie des classes de CTypes* du début de chapitre, est définie pour les différentes Classes de C-types. Cette hiérarchie permet de structurer les C-types selon leurs caractéristiques. Ainsi, lorsque l'on souhaite ajouter un nouveau C-type, on l'intègrera dans cette hiérarchie pour qu'il hérite des caractéristiques pré-définies (ordonné, construit,...).

Pour typer un slot dans AROM on n'utilisera pas directement une de ces classes. En effet, une variable n'est pas de type *discret* ou *multivalué* mais de type *entier* ou *list d'entier de 0 à 120*. On utilise non pas les classes de C-type mais des instances de ces classes.

Ce chapitre traite de la relation de sous-typage qui existe entre les instances des C-types. Cette relation est orientée vers les données représentées par les C-types, elle n'est pas maintenue mais elle est vérifiée au cas par cas sur demande. La comparaison des instances se fait pour chaque classe de C-type, c'est pourquoi une définition est donnée pour chacune de ces classes de C-types.

Néanmoins, cette relation est toujours basée sur un même principe qui est l'inclusion ensembliste.

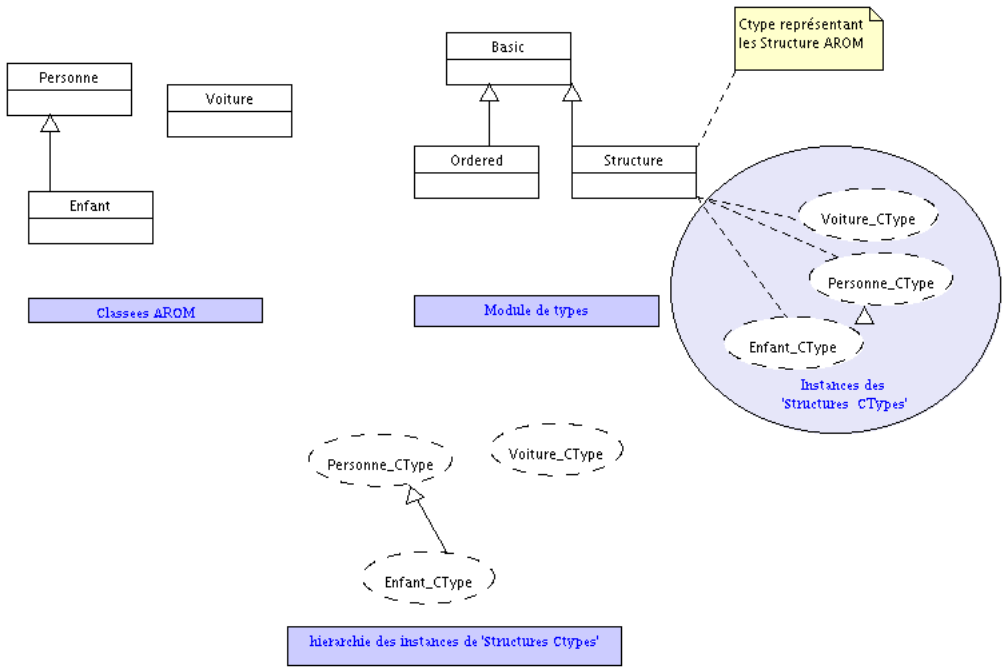
- Les `CTypes` ordonnés n'acceptent que des singletons comme instances, par conséquent le problème de relation de sous-typage des instances de C-type ne se pose pas. *Dans le cas où un nouveau CType est intégré dans les C-types ordonnés, cet aspect sera à prendre en compte.*

  Deux `CTypes` ordonnés ne peuvent donc pas avoir de super C-type commun.

- Les `StructureCT` permettent de typer les structures AROM. Il est donc logique que l'on retrouve la même relation d'ordre que celle des structures. Dans l'exemple ci-dessous, le type *Pesonne_CType* est le Super C-type de *Enfant_CType*. Par contre le type *Voiture_CType* est dissocié des deux autres.

  A partir de deux `StructureCT` il est donc facile de déterminer si il existe un super C-type commun, puisque la hiérarchie des `StructureCT` est accessible via celle des structures AROM.

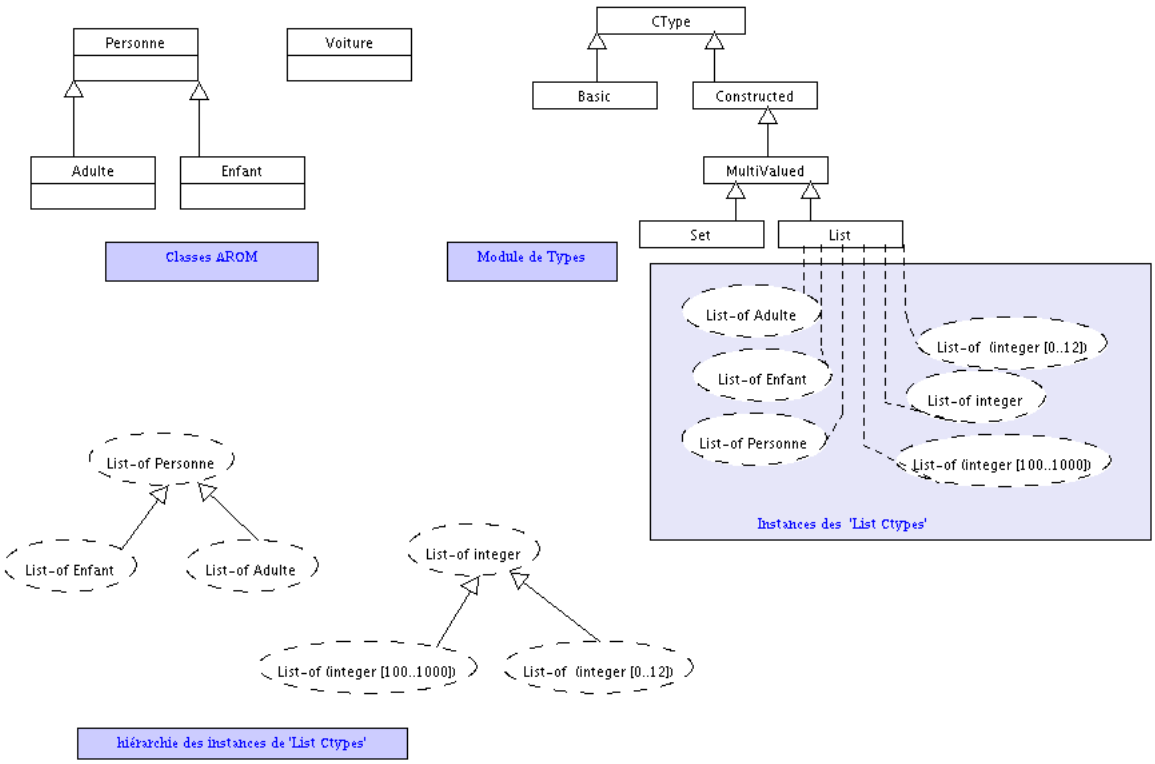
Figure 5-3.



- Les `MultiValCT`, qui se déclinent en `ListCT` et en `SetCT`, sont des collections d'objets appartenant à un δ-type donné. La relation de sous-typage de ces C-types est donc la même que celle existante entre les δ-types des éléments. Dans les cas les plus simple, lorsque le δ-type décrit un C-type, les mêmes règles que celles décrites dans cette section sont appliquées. Par exemple, si l'on considère l'exemple qui suit, le type *List-of Personne* est super C-type de *List-of Enfant* et *List-of Adulte*. Pour le cas où les δ-types des éléments de la collection définissent un domaine, la relation de sous-typage des `MultiValCT` est celle des δ-types, c'est à dire que c'est une relation d'ensembles. Ainsi, le type *List-of Integer* est super C-type de *List-of (Integer [0..12])* et de *List-of (Integer [100..1000])*. Par contre les deux sous C-types n'ont pas de relation de sous-typage. Voir le chapitre qui suit sur la relation de sous-typage des δ-types pour plus d'informations.

  Le Super C-type commun à deux `MultiValCT`, *mv1* et *mv2*, est un `MultiValCT` basé sur le super δ-type commun aux δ-types composant *mv1* et *mv2*.

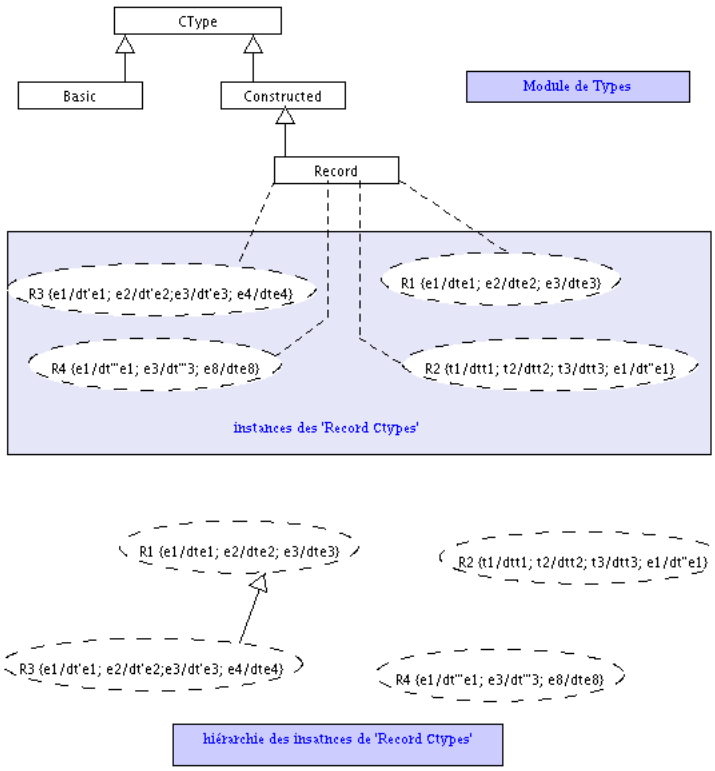
Figure 5-4. Relation de sous-typage des `MultiValCT`



- Enfin, il reste les `RecordCT` qui définissent un ensemble de couples "étiquettes/δ-types". Les `StructureCT` et les `RecordCT` ne diffèrent que par le fait que les `StructureCT` représentent explicitement des structures AROM alors que les `RecordCT` représentent toute donnée coposée de couple *étiquette-δ-type*. Les `RecordCT` sont d'ailleurs utilisés pour représenter les intentions des `StructureCT`. En effet, une structure est définie par un ensemble de slots ayant un identifiant (nom) et un δ-type. Par conséquent, les mêmes règles de spécialisations que celles des Structures AROM sont utilisées. C'est à dire que pour qu'un `RecordCT` soit sous C-type d'un autre `RecordCT` il faut qu'il définisse les mêmes étiquettes et que celles-ci soient associées au même δ-type ou à un sous δ-type. D'autre étiquettes peuvent être définies au niveau du sous C-type.

  Dans l'exemple ci-après, le C-type *R1* est super C-type de *R3* si les δ-types *dt'e1*, *dt'e2* et *dt'e3* sont égaux ou sous δ-types de, respectivement, *dte1*, *dte2* et *dte3*. Par contre, le C-type *R2* est entièrement dissocié des autres C-types malgrès la définition de l'étiquette e1. Le C-type *R4* ne peut pas non plus aspirer à être un sous C-type de *R1* puisqu'il ne définit pas l'étiquette e2.

Figure 5-5. Relation de sous-typage des `RecordCT`



De la même façon que pour les `MultiValCT`, il est possible de déterminer un Super C-type commun à deux `RecordCT`, *RecCT1* et *RecCT2*. Ce `RecordCT` devra ne définir que les étiquettes communes à *RecCT1* et *RecCT2*. Les δ-types associés à ces étiquettes devront être des super δ-types de ceux associés aux étiquettes de *RecCT1* et *RecCT2*. Un exemple est donné ici:

```
RecCT1 :
e1 /  integer [0 .. 20]
e2 /  string {"julien", "barnabé", "ivan"}
e3 /  float

RecCT2 :
e1 /  integer [100 .. 2000]
e2 /  string {"nathalie", "helene", "pierre"}
e4 /  boolean

le RecordCT parent commun minimal à RecCT1 et RecCT2 est :
e1 /  integer [0 .. 20] U [100 .. 2000]
e2 /  string {"julien", "barnabé", "ivan", "nathalie", "helene", "pierre"}
```

5.3.2. Relations de sous-typage des δ-types

De la même façon que pour les C-types, on s'intéresse à la relation de sous-typage des instances de δ-types. La hiérarchie structurelle est identique à celle des C-types et elle est utile pour les utilisateurs désirant ajouter de nouveaux types dans AROM.

Les δ-types sont composés d'un C-type de base et d'un domaine qui est un sous-ensemble du domaine de valeurs décrit par le C-type. Par conséquent, la hiérarchie définit pour les C-types est appliquées ici mais on l'affine en fonction des domaines spécifiés.

Dans tous les cas, quelque soit la catégorie à laquelle on s'intéresse, le δ-type *Max*, qui représente le même domaine que le C-type auquel il est associé, est au sommet de la *hiérarchie*, il est super δ-type de tous les autres δ-types appartenant à la même classe. Inversement, le δ-type *Min*, qui représente l'ensemble vide, est en bas de la *hiérarchie*. Une description plus précise pour les autres cas de figure est donné pour chaque catégorie de δ-type.

D'une manière générale, la règle de spécialisation appliquée est: Pour qu'un δ-type soit sous δ-type d'un autre, il faut que toutes ses valeurs soient acceptées par le second.

- Les `DeltaType` ordonnés hérite des C-types d'un unique niveau qui est le singleton définit par ces C-types et qui représente le δ-type *Max*, c'est à dire le super δ-type de tous les autres. Ce niveau définit donc l'ensemble des entiers, des réels, des chaînes de caractères ou des booleens selon le `DeltaType` considéré.

  Les domaines associés aux δ-types ordonnés sont représentés par des listes d'intervalles de valeurs autorisées. Si l'on reprend la règle de spécialisation énoncée ci-dessus, pour qu'un δ-type *d2* soit sous type d'un δ-type *d1*, il faut que le domaine de *d2* soit inclus dans le domaine de *d1*. Par exemple, Integer {[- inf .. 12] U [58 .. 8996]} est super δ-type de Integer {[- inf .. -1002] U [0 .. 10] U [789 .. 1000]}

  Cette définition permet également de dire que le super δ-type *minimum* commun à deux δ-types distincts est l'union de ces deux types. De la même façon le sous δ-type *maximum* commun à deux δ-types distincts est l'intersection de ces deux types. Deux δ-types sont dits distincts si ils ne peuvent pas être hiérarchisés, aucun n'est sous type de l'autre. De plus, par minimum (maximum) on entends le δ-type ayant le moins (plus) de valeurs autorisées.

- Les valeurs des `StructureCT` sont les instances des Structures AROM représentées. Les domaines des `StructureDT` peuvent donc toujours être représentés par un ensemble d'instances autorisées. En effet, si l'on précise un ensemble d'instances interdites, l'ensemble résultant est l'ensemble de toutes les instances auquel on aura retiré les instances précisées par l'utilisateur. Un `StructureDT` δt_{s2} définissant un domaine `S2` est sous δ-type d'un `StructureDT` δt_{s1} définissant un domaine `S1` si il est possible de déterminer une relation de sous-typage entre les `StructureCT` qui leur sont associés et si la règle ci-dessous est vrai :

  i.

  $S1 = X\ (X=\{x_1,...,x_n\})$ $\qquad$ $S2 = Y\ (Y=\{y_1,...,y_m\})$

  $\delta t_{s2} < \delta t_{s1}$ ssi

  $\forall\ y \in Y,\ y \in X$

La relation qui peut exister entre les δ-types δt_{s2} et δt_{s1} n'est pas obligatoirement la même que celle qui existe entre leur C-type, C_{s1} et C_{s2}.

Dans le cas où une relation entre deux δ-types ne peut être déterminé, mais qu'il existe une relation entre les `StructureCT` qui leur sont associé, il est possible de définir le super δ-type *minimum* commun. Celui-ci est issu d'un des deux `StructureCT` associé aux δ-type, celui qui est super type de l'autre, et il a pour domaine l'union des domaines des deux δ-types δt_{s1} et δt_{s2}.

- Les `MultiValDT` restreignent les `MultiValCT` en définissant, d'une part, des domaines sous forme d'ensembles de valeurs autorisées ou interdites et, d'autres part, une cardinalité.
  - Dans le cas où un domaine de valeurs est définit, la cardinalité n'entre en ligne de compte que lors de l'ajout de valeurs à ce domaine. Un `MultiValDT` δt_{mv2} définissant un domaine `S2` est sous type d'un `MultiValDT` δt_{mv1} définissant un domaine `S1` si une des règles ci-dessous est vrai :

    i.

    `S1 = X (X={`$x_1,...,x_n$`})` `S2 = Y (Y={`$y_1,...,y_m$`})`
    $\delta t_{mv2} < \delta t_{mv1}$ ssi
    $\forall y \in Y, y \in X$

    ii.

    `S1 = !X (X={`$x_1,...,x_n$`})` `S2 = Y (Y={`$y_1,...,y_m$`})`
    $\delta t_{mv2} < \delta t_{mv1}$ ssi
    a. $\delta t_{mv1}MAX \geq \delta t_{mv2}MAX$:
    $\forall y \in Y, y \notin X$
    b. $\delta t_{mv1}MAX < \delta t_{mv2}MAX$:
    $\forall y \in Y, (y \notin X \;\&\&\; y \in \delta t_{mv1}MAX)$

    iii.

    `S1 = !X (X={`$x_1,...,x_n$`})` `S2: !Y (Y={`$y_1,...,y_m$`})`
    $\delta t_{mv2} < \delta t_{mv1}$ ssi
    a. $\delta t_{mv1}MAX \geq \delta t_{mv2}MAX$:
    $\forall x \in X, (x \in Y \parallel x \notin \delta t_{mv2}MAX)$
    b. $\delta t_{mv1}MAX < \delta t_{mv2}MAX$:
    $Z = \delta t_{mv2}MAX$ diff $\delta t_{mv1}MAX$
    $\forall x \in X, x \in Y \;\&\&\; \forall z \in Z, z \in Y$

    iv.

    `S1 = X (X={`$x_1,...,x_n$`})` `S2 = !Y (Y={`$y_1,...,y_m$`})`
    $\delta t_{mv2} < \delta t_{mv1}$ ssi
    a. $\delta t_{mv1}MAX \geq \delta t_{mv2}MAX$:
    $Z = \delta t_{mv2}MAX$ diff Y
    $\forall z \in Z, z \in X$
    b. $\delta t_{mv1}MAX < \delta t_{mv2}MAX$:
    $Z = \delta t_{mv1}MAX$ diff Y
    $U = \delta t_{mv2}MAX$ diff $\delta t_{mv1}MAX$

$\forall$ z $\in$ Z, z $\in$ X && $\forall$ u $\in$ Y, u $\in$ Y

Les points iii et iv ci-dessus ne sont pas toujours vérifiable. En effet, le $\delta t_{mv}MAX$ peut etre infini, si la cardinalité maximum est infini et/ou le nombre d'éléments composant les collections est infini.

Pour déterminer un super δ-type commun (le minimum ?) à deux `MultiValDT` qui ne sont reliés par le sous-typage, les règles suivantes s'appliquent:

i.

S1 = X (X={x_1,...,x_n}) S2 = Y (Y={y_1,...,y_m})
δtMV_{parent} > δt_{mv1} et δtMV_{parent} > δt_{mv2} ssi
S_{parent} = S1 $\cup$ S2
&& C-type associé à δtMV_{parent} =
1. C-type associé à δt_{mv1} si $\forall$ y $\in$ Y, y $\in$ $\delta t_{mv1}MAX$
2. C-type associé à δt_{mv2} si $\forall$ x $\in$ X, x $\in$ $\delta t_{mv2}MAX$
3. Super C-type commun de δt_{mv1} et δt_{mv2}, sinon.

ii.

S1 = X (X={x_1,...,x_n}) S2 = !Y (Y={y_1,...,y_m})
δtMV_{parent} > δt_{mv1} et δtMV_{parent} > δt_{mv2} ssi
Z = Y diff X
S_{parent} = !Z,
&& C-type associé à δtMV_{parent} =
1. C-type associé à δt_{mv2} si $\forall$ x $\in$ X, x $\in$ $\delta t_{mv2}MAX$
2. Super C-type commun de δt_{mv1} et δt_{mv2}, sinon.

iii.

S1 = !X (X={x_1,...,x_n}) S2 = !Y (Y={y_1,...,y_m})
δtMV_{parent} > δt_{mv1} et δt_{mv2} ssi
Z = (X diff $\delta t_{mv2}MAX$) $\cup$ (Y diff $\delta t_{mv1}MAX$) $\cup$ (Y $\cap$ X)
S_{parent} = !Z,
&& C-type associé à δtMV_{parent} = Super C-type commun de δt_{mv1} et δt_{mv2}.

Dans le point ii ci-dessus, on remarque que ce n'est pas le super δ-type commun qui est définit. En effet, par exemple, dans le cas où S1 $\notin \delta t_{mv2}$ Z = Y et le C-type de δtMV_{parent} et le super C-type de ceux associés à δt_{mv1} et δt_{mv2}. Par conséquent, tous les éléments de δt_{mv1} seront inclus dans δtMV_{parent} au lieu des seules éléments de X.

- Par contre, si aucun domaine n'est défini, toutes les valeurs du C-type sont autorisées à partir du moment où elles respectent la cardinalité. Pour qu'un `MultiValDT` δt_{mv2} ayant une cardinalité `C2` soit sous type d'un `MultiValDT` δt_{mv1} ayant une cardinalité `C1` il faut que l'intervalle d'entiers défini par `C2` soit inclus dans l'intervalle d'entiers défini par `C1`. En effet, si la cardinalité est [1..8] pour `C1` et [2..5] pour `C2`, les valeurs de δt_{mv2} seront correctes pour δt_{mv1}.

Le super δ-type commun à deux `MultiValDT` non reliés par sous-typage, est un `MultiValDT` définissant une cardinalité ayant pour minimum le plus petit minimum des deux δ-types et pour maximum le plus grand maximum des deux δ-types. Si l'on considère un δ-type avec une cardinalité `C1` de [1..8] et un δ-type avec une cardinalité `C2` de [10..100], le super δ-type commun définira une cardinalité `C3` de [1..100].

- Le dernier cas à considérer est le cas où il faut classer un `MultiValDT` δt_{mv1} définissant un domaine `S1` et un `MultiValDT` δt_{mv2} définissant une cardinalité `C2`. Les règles suivantes s'appliquent alors :

  i.

  `S1 = X (X={x1,...,xn})`
  $\delta t_{mv1} < \delta t_{mv2}$ `ssi`
  $\forall$ `x` $\in$ `X, length x` $\in$ `C2`

  ii.

  `S1 = !X (X={x1,...,xn})`
  `Impossible de classer les deux types puisque l'ensemble de toutes les valeurs du type est infini.`

- Les `RecordDT` restreignent les `RecordCT` en définissent pour domaines des ensembles de valeurs autorisées ou interdites. On retrouve donc la même définition que celles des `MultiValDT`. Par conséquent, les règles citées pour les `MultiValDT` s'appliquent. Tout comme pour les `MultiValDT`, les points iii et iv de ces règles sont particulièrement difficile à satisfaire car il suffit qu'un des δ-types associés aux étiquettes soit infini pour que l'ensemble des valeurs du C-type de base soit infini. De plus, même lorsque cet ensemble n'est pas infini, le nombre de valeurs possible peut être *astronomique*.

  De la même façon, pour ce qui est de déterminer un super δ-type commun à deux `RecordDT`, les règles définies pour `MultiValDT` s'appliquent.

Actuellement, il n'est pas possible de définir un δ-type à partir d'un `RecordCT` ou d'un `MultiValCT` en redéfinissant les δ-types des éléments des collections ou étiquettes. Cette possiblité pourrait nécessité des corrections sur la définitions des relations de sous-typage.

5.4. Nommage des C-types

Par défaut, le système de types définit des C-types dits simples tels que les entiers, les réels, les booléens ou les chaînes de caractères. De plus, il permet à l'utilisateur de faire référence à une Structure AROM afin de désigner le C-type représentant cette Structure dans le système de types. D'autre part, l'utilisateur peut créer des nouveaux C-types construits, tels que des listes ou des ensembles, en

spécifiant un nom référençant ce nouveau C-type. Chaque C-type est donc nommé et il est référencé auprès du système de types. En effet, le système de types permet d'obtenir tout C-type d'après son nom.

Pour les C-types simples, aucun problème ne se pose puisqu'ils sont définis par des singletons. Les noms des instances de ces C-type sont: *integer*, *float*, *boolean* ou *string*.

Par contre, les C-types construits admettent une multitude, voire une infinité, d'instances (ListCT de string, ListCT de integer[10..30], ...). Par défaut aucun C-type construit n'est défini, leur création se fait via le système de types, auquel on spécifie le nom du C-type. Ainsi, une fois un C-type créé, il est toujours possible d'y accéder via le système de types, à partir de son nom. Dans cette configuration, on voit qu'il est tout à fait possible de créer plusieurs instances de C-types, égaux dans leur définition mais ayant un nom différent. Aucune détection n'est faite dans ce sens par le système de types. Néanmoins, la méthode d'égalité vérifiera si les définitions des C-types sont égaux sans considérer le nom.

Il existe une autre règle de nommage-création. En effet, les C-types peuvent avoir n'importe quel nom à partir du moment où il est conforme aux règles de nommage d'AROM. Mais si celui-ci est composé de *list-of* ou *set-of* plus un nom d'un autre C-type, si le système de type n'a pas déjà référencé ce C-type, il le créera. Ainsi, si on demande au système de type le C-type *list-of integer* et que celui-ci n'est pas référencé, un CType `ListCT` basé sur le δ-type issu de *integer* est créé et référencé.

Lors de la création d'un C-type à partir d'une structure AROM, c'est le *fully qualified name* de cette structure qui sera utilisé pour référencer le C-type. En effet, le système de types étant le même pour toutes les bases AROM, l'utilisation du nom simple pourrait provoquer une situation de conflit si deux structures AROM appartenant à des bases différentes avaient le même nom. De la même façon que précédement, si l'on souhaite obtenir un C-type multivalué basé sur un Structure C-type, il faut utiliser le *fully qualified name* après l'identifiant *list-of* ou *set-of* .

Afin de simplifier l'accès aux C-type, il est possible d'y accéder non pas depuis le module de types mais depuis une base de connaissances. Ainsi, les noms simples des Structures AROM peuvent être utilisés soit pour accéder aux structures C-types soit pour accéder aux C-types multivalués basés sur les structres C-types.

Enfin, cette simplification du nom utilisé peut également être utilisé au niveau du fichier txta ou dans une interface utilisateur. Pour un C-type donné, il est possible de connaître, en plus de son nom unique, son nom usuel. Celui-ci est accessible depuis le système de types.

5.5. C-types et Valeurs

Les différents C-types du module de types acceptent chacun des valeurs bien spécifiques. En effet, pour qu'un objet soit considéré comme valide pour un C-type, voire un δ-type, donné il doit répondre à certaines règles. Les couples (C-types,Valeurs) sont donnés dans le tabeau qui suit:

Tableau 5-1. Valeurs admises pour les C-types de base

C-Type	Type Java des Valeurs
integer	java.lang.Long
boolean	java.lang.Boolean

C-Type	Type Java des Valeurs
string	java.lang.String
float	java.lang.Double
ListCT	java.util.List
SetCT	java.util.Set
RecordCT	arom.kr.util.Record

Avertissement

Pour les valeurs des C-types `ListCT` et `SetCT`, il est à noter que deux comportements sont possibles. Si l'on prend l'exemple d'une variable typée par `ListCT`

- Lors de l'attribution d'une valeur à cette variable, l'utilisateur passe au système n'importe quelle valeur de type `List`. Une copie de cette liste est sauvegardée par le système. Ceci signifie que TOUTES modifications ultérieures ne seront PAS VISIBLES au niveau de la variable.
- Par contre, lorsque l'utilisateur demande la valeur de cette variable au système celui-ci renvoie la liste elle-même. Par conséquent, TOUTES les modifications ultérieures réalisées sur cette liste seront VISIBLES au niveau de la variable.

Ces comportement sont également vrais pour les valeurs des types `SetCT`.

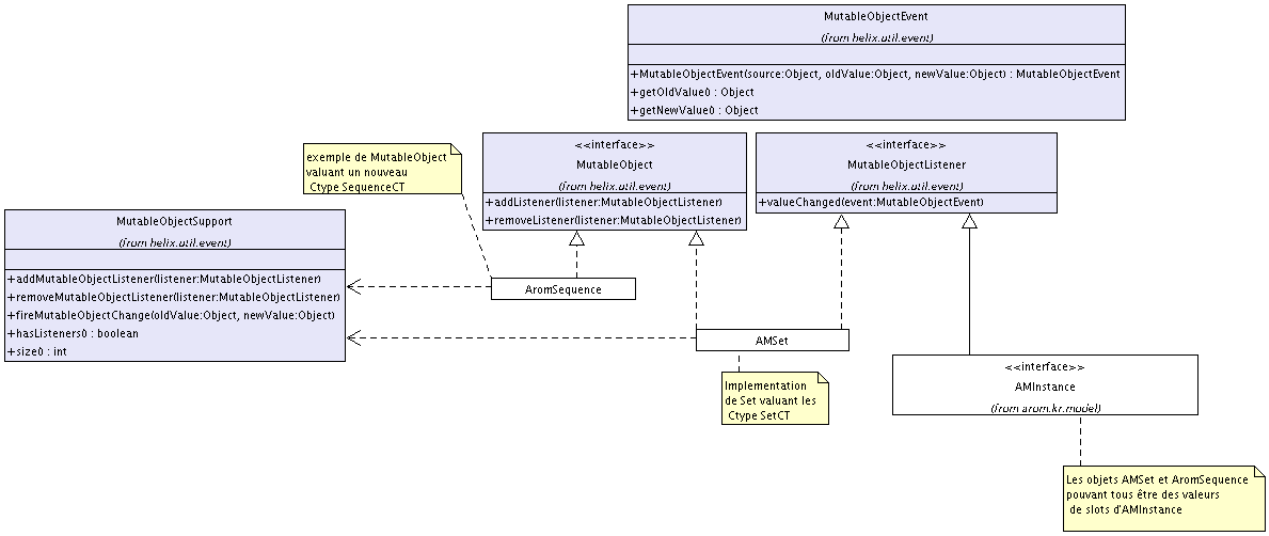
5.5.1. Objets MUABLE

La descrition du comportement des objets valuant des `ListCT` et `SetCT` lève un problème qui devra également être pris en compte par les nouveaux C-types. En effet, les instances permettent à des *écouteurs* d'être avertis lorsque la valeur d'un de ses slots est modifiée. Or, si il est possible de modifier la valeur d'un slot sans passer par les méthodes de l'API (setValue()), l'instance n'est pas avertie qu'une modification a eu lieu et elle ne peut donc pas notifier ses propres *écouteurs*.

C'est pour remédier à ce problème que le concept de *MutableObject* a été définie. L'interface `MutableObject` doit être implémentée par tout Objet (Valeur de C-types) dont l'état interne peut changer. D'autre part les `AMInstance` tout comme les valeurs des C-types construits implémentent l'interface `MutableObjectListener`. Ces *listeners* s'enregistrent auprès des `MutableObject` qui les composent. Ainsi, lorsqu'un tel Objet verra son état changé, il notifiera ses *listeners*. Cette solution permet donc aux `AMInstances` d'etre avertis lorsque l'état interne d'un de ses slots est modifié. Mais elle permet également aux valeurs de C-types construits, tel que les `List` ou les `Set` d'etre notifiées lorsqu'un élémént de leur valeur, ici un élément de la collection, est modifié. Ces valeurs construites peuvent alors eux-meme notifier leurs éventuelles *listeners*.

Il est à noter que, par conséquent, ce sont des implémentations particulières de `List` et de `Set` qui sont utilisées dans l'implémentation d'AROM afin qu'elles soient également des `MutableObject` et des `MutableObjectListener`.

Figure 5-6. Les MutuableObjects et *associés*.



5.6. API relative aux types

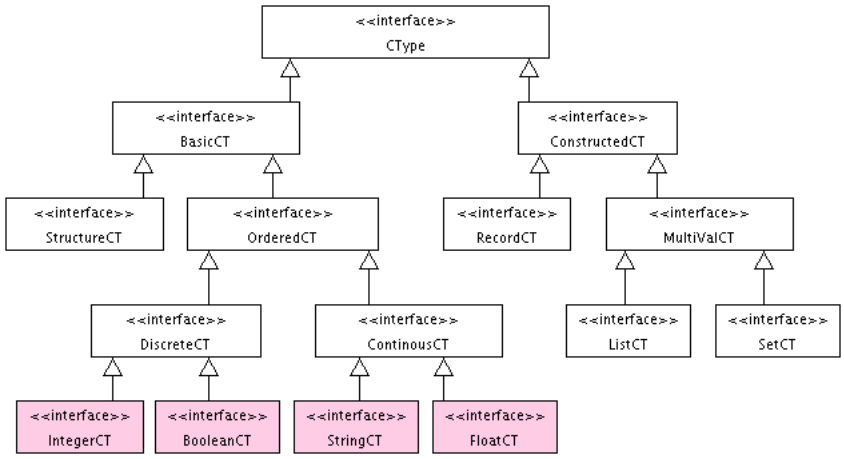
Le point d'entrée du module de types est `TypeSystem`. Une instance de la classe d'implémentation de l'interface `TypeSystem` est obtenue par l'intermédiaire de la classe `AromSetUp` :

```
TypeSystem ts_ = AromSetUp.getAromSystem().getTypeSystem();
```

Depuis la classe `TypeSystem` ainsi obtenue, il est alors possible d'accéder aux différentes données et opérations proposées par le module de types.

Les interfaces des C-types proposées dans AROM sont représentées dans le diagramme qui suit. Ce sont des instances de ces C-types qui représentent effectivement un C-type donné.

Figure 5-7.



5.6.1. Typage des variables

L'une des possibilités offertes est d'accéder aux différents C-types définis par le module de types. Ce sont ces C-types qui servent à typer un slot d'une base de connaissances AROM, comme illustré dans l'exemple qui suit. En effet, lors de la création d'une variable il est nécessaire de spécifier des `FacetModifiers` décrivant cette nouvelle variable. Le seul *modifieur* qui soit obligatoire est le `CType-Modifier` permettant de savoir quel `CType` type la variable en question. La classe `TypeSystem` offre des méthodes permettant de récupérer les `CTypes` simples. Mais elle offre également la possibilité de créer de nouveaux C-types construits, des `ListCT` ou des `SetCT`, qui seront référencés dans le module de types par le nom spécifié en paramètre lors de cette création.

Pour rechercher un C-type à partir de son nom, il existe deux possibilités. Soit on réclame le type à l'objet `TypeSystem`, soit on le réclame à la `KnowledgeBase`. La différence entre ces deux accès apparaît uniquement lorsque le nom spécifiée est celui d'une Structure AROM. Dans le premier cas il est nécessaire d'utiliser le *fully-qualified name* de la structure pour retrouver le C-type la représentant. Par contre, si l'on s'adresse à la `KnowledgeBase` le nom simple suffit puisque le *fully-qualified name* est facilement accessible. De plus, lorsque l'on s'adresse à la `KnowledgeBase` et que l'on ne précise que le nom simple, si aucun C-type n'avait été préalablement enregistré, celui-ci sera créé et retourné à l'utilisateur.

Exemple 5-2. Typer une variable.

```
       import arom.AromSetUp;
import arom.kr.factory.AromSystem;
import arom.kr.factory.ModifierFactory;
import arom.kr.model.AMClass;
import arom.kr.model.KnowledgeBase;
import arom.kr.model.Variable;
```


```
import arom.kr.model.InvalidNameException;
import arom.kr.model.EntityCreationException;
import arom.kr.model.Structure;
import arom.kr.model.type.TypeSystem;
import arom.kr.model.type.CType;
import arom.kr.model.modifiers.CTypeModifier;
import arom.kr.model.modifiers.DomainModifier;
import arom.kr.model.modifiers.FacetModifier;
import java.util.Vector;
import java.util.Iterator;


/**
 * Cet exemple montre comment acceder aux differents C-Types definis
 * dans le module de types et comment les utiliser afin de typer une variable.
 *
 * @author Veronique DUPIERRIS
 */

public class CreateTypedVars {

  /**
   * Cette methode cree une nouvelle base de connaissances nommee "BaseTest" et
   * cree une classe dans cette base : la classe racine 'Personne'.
   *
   * @return l'object KnowledgeBase representant la nouvelle base de
   * connaissances.
   */
  public static KnowledgeBase createKb(){
    String kbName = "BaseTest";
    AromSystem aromFactory = AromSetUp.getAromSystem();
    KnowledgeBase kb = null;
    try{
      kb = aromFactory.createKB(kbName, new Object[0]);
      AMClass personne = kb.createClass("Personne", null);
    } catch(EntityCreationException ece){
      System.out.println("Une erreur est survenue lors de la creation de la base : ");
      System.out.println(ece.getMessage());
      // Arrete le system AROM proprement
      AromSetUp.getAromSystem().cleanup();
      System.exit(1);
    }
    return kb;
  }

  /**
   * Cette methode recupere une instance de la classe d'implementation de
   * TypeSystem, qui est le point d'entre du module de types. Les instances
   * des C-Types sont ensuite obtenues et sont utilisees pour typer de nouvelles
   * variables de la structure specifiee.
   *
   * @param structure La Structure AROM a laquelle sera ajoutees quatre variables
```

```
 * de type String, Integer, Float et Boolean.
 */
public static void createAllTypedVariables(Structure structure){
  //Recupere le module de type.
  TypeSystem typeSystem = AromSetUp.getAromSystem().getTypeSystem();

  // Recupere l'objet permettant de creer les modifieurs de facettes
  ModifierFactory modifierFactory = AromSetUp.getAromSystem().getModifierFactory();

  //Recupere les differents CTypes definis et les utilise pour typer une variable :
  try {
    // Chaine de caracteres
    CType stringCType = typeSystem.getStringCType();
    // Creer le CTypeModifier permettant de typer une variable
    CTypeModifier ctypeModifier = modifierFactory.createCTypeModifier(stringCType);
    // Creer une nouvelle variable (nom) a la structure de type String
    FacetModifier[] facettes = new FacetModifier[1]; // un seul modifiers: CType
    facettes[0] = ctypeModifier;
    structure.createVariable("nom", facettes);

    // Entier
    CType integerCType = typeSystem.getIntegerCType();
    // Creer le CTypeModifier permettant de typer une variable
    ctypeModifier = modifierFactory.createCTypeModifier(integerCType);
    // Creer une nouvelle variable (age) a structure de type Integer
    facettes[0] = ctypeModifier; // sauvegarde le nouveau CType
    structure.createVariable("age", facettes);

    // Reel
    CType floatCType = typeSystem.getFloatCType();
    // Creer le CTypeModifier permettant de typer une variable
    ctypeModifier = modifierFactory.createCTypeModifier(floatCType);
    // Creer une nouvelle variable (salaire) a structure de type Float
    facettes[0] = ctypeModifier; // sauvegarde le nouveau CType
    structure.createVariable("salaire", facettes);

    // Booleen
    CType booleanCT = typeSystem.getBooleanCType();
    // Creer le CTypeModifier permettant de typer une variable
    ctypeModifier = modifierFactory.createCTypeModifier(booleanCT);
    // Creer une nouvelle variable (marie(e)) a structure de type Boolean
    facettes[0] = ctypeModifier; // sauvegarde le nouveau CType
    structure.createVariable("mariee", facettes);
  } catch(InvalidNameException ine) {
    System.out.println("Une erreur est survenue lors de la creation d'une variable: ");
    System.out.println(ine.getMessage());
    // Arrete le system AROM proprement
    AromSetUp.getAromSystem().cleanup();
    System.exit(1);
  }
}
```

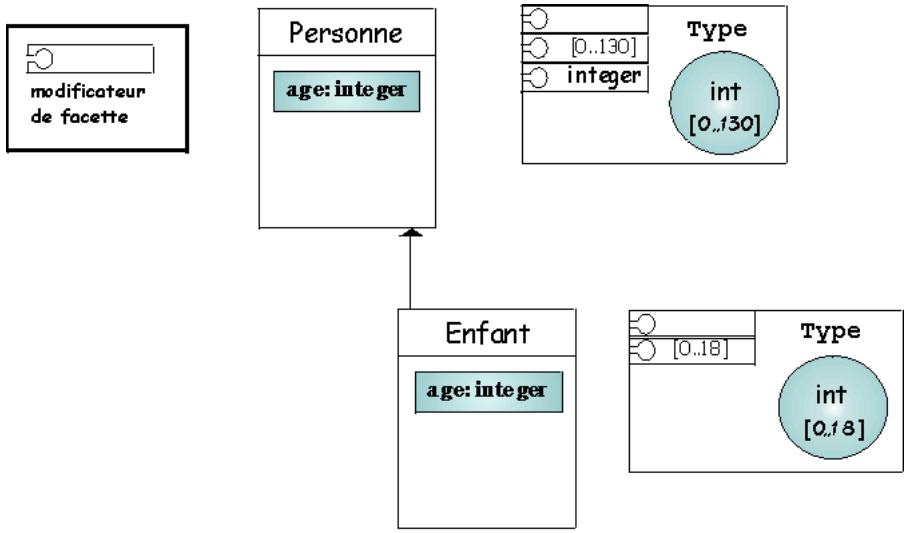
```
  public static void main(String[] args){

    KnowledgeBase kb = createKb();
    AMClass personne = kb.lookupClass("Personne");
    createAllTypedVariables(personne);
    System.out.println("Nouvelle classes : "+ personne.getFQName());
    System.out.println("Slots : ");
    Iterator personneVariables = personne.variables();
    while(personneVariables.hasNext()){
      Variable nextVariable = (Variable) personneVariables.next();
      CType variableType = nextVariable.getDescriptionFor(personne).getType().getCType();
      System.out.println(nextVariable.getFQName()+" ("+variableType+")");
    }

    // Arrete le system AROM proprement
    AromSetUp.getAromSystem().cleanup();
  }

}// CreateTypedVars
```

5.6.2. Restriction du domaine

A une variable est associé un ensemble de descriptions, voir la section *Slots*, qui contiennent la définition des facettes (documentation, inférence, δ-type) pour chacune des structures dans laquelle elle apparaît. Le δ-type d'une variable pour une structure donnée est donc défini dans cette description.

Si l'on considère l'exemple de la section *Spécialisation d'une facette de type*, redonné ci-après, le slot `age` est défini par la classe `Personne`. Un descripteur du slot `age` pour la classe `Personne`, $desc_{ageP}$, est donc créé et, plus précisemment, un δ-type, δt_{xA}, est défini. Ce δ-type a été créé en appliquant successivement le modificateur de CType *Ctype = integer* et le modificateur de domain *interval=[0..130]* au descripteur $desc_{ageP}$. La classe `Enfant` est une sous classe de `Personne`, par conséquent elle hérite du slot `age`. Un descripteur de ce slot pour la classe `Enfant` est donc également créé, $desc_{ageE}$ et il hérite du δ-type δt_{ageP}. Un modificateur de domaine *interval=[0..18]* est ensuite appliqué à $desc_{ageE}$ définissant un nouveau δ-type, δt_{ageE}, dont le domaine est l'intervalle [0..18]

Figure 5-8. Restriction de domaine d'une variable



Toujours en considérant cet exemple, l'application du modificateur de domaine *interval=[20..130]* à $desc_{ageP}$ génère une erreur puisque l'ensemble qui serait dénoté par δt_{ageP} serait disjoint de celui dénoté par δt_{ageE}. Dit autrement les contraintes exprimées par δt_{ageP} (age >= 20 et age <=130) sont incompatibles avec les contraintes exprimées par δt_{ageP} (age >= 0 et age <=18). Une solution, pour l'utilisateur, consisterait à annuler le modificateur de domaine appliqué à $desc_{xB}$, δt_{ageE} serait de nouveau identique à δt_{ageP}, puis à modifier δt_{ageP} en appliquant le modificateur de domaine désiré à $desc_{ageP}$.

Le code correspondant à cet exemple est donné ci-après.

Exemple 5-3. Modifier le domaine d'une variable.

```
/**
 * Cet exemple  montre comment modifier le domaine, et donc le delta-type, associe a
 * une variable. Pour cela des modificateurs de domaine sont appliques et d'autre sont
 * annules.
 *
 * @author Veronique DUPIERRIS
 */
import arom.kr.model.KnowledgeBase;
import arom.kr.factory.AromSystem;
import arom.AromSetUp;
import arom.kr.model.AMClass;
import arom.kr.model.type.CType;
import arom.kr.model.modifiers.CTypeModifier;
import arom.kr.factory.ModifierFactory;
import arom.kr.model.modifiers.FacetModifier;
import arom.kr.model.EntityCreationException;
```


```
import arom.kr.model.InvalidNameException;
import arom.kr.model.modifiers.InvalidModifierException;
import arom.kr.model.VariableDescriptor;
import arom.kr.model.modifiers.DomainModifier;

public class ChangeVarsDomain {

    /**
     * Cette methode cree une nouvelle base de connaissances nommee "BaseTest" et
     * cree les classes dans cette base : la classe racine 'Personne' et la classe
     * 'enfant' la specialisant. Une variable age ayant pout CType integer est
     * egalement creee pour la classe Personne.
     *
     * @return l'object KnowledgeBase representant la nouvelle base de
     * connaissances.
     */
    public static KnowledgeBase createKb(){
 String kbName = "BaseTest";
 AromSystem aromFactory = AromSetUp.getAromSystem();
 KnowledgeBase kb = null;
 try{
     kb = aromFactory.createKB(kbName, new Object[0]);
     AMClass personne = kb.createClass("Personne", null);

     CType integerCType = aromFactory.getTypeSystem().getIntegerCType();
        // Creer le CTypeModifier permettant de typer une variable
     ModifierFactory modifierFact =
       AromSetUp.getAromSystem().getModifierFactory();
     CTypeModifier ctype = modifierFact.createCTypeModifier(integerCType);
        // Creer une nouvelle variable (age) de type Integer
     FacetModifier[] facettes = new FacetModifier[1];
     facettes[0] = ctype; // sauvegarde le nouveau CType
     personne.createVariable("age", facettes);

     AMClass enfant = kb.createClass("Enfant", personne);

 } catch(EntityCreationException ece){
     System.out.println("Une erreur est survenue lors de la creation de la base : ");
     System.out.println(ece.getMessage());
     System.exit(1);
 } catch(InvalidNameException ine) {
     System.out.println("Une erreur est survenue lors de la creation de la base : ");
     System.out.println(ine.getMessage());
     System.exit(1);
 }
 return kb;
    }

    /**
     * Cette methode applique un nouveau domaine au descripteur de variable specifie,
     * variable qui doit etre de type integer.
     *
```

```
     * @param descrVar Le descripteur de variable de type integer auquel il faut
     * ajouter un domaine dont les bornes sont specifiees.
     * @param min la borne min de l'intervalle
     * @param max la borne max de l'intervalle
     */
    public static void applyDomain(VariableDescriptor descrVar, Long min, Long max)
                                            throws InvalidModifierException {
// Crée le modificateur de domaine
        ModifierFactory modifierFact =  AromSetUp.getAromSystem().getModifierFactory();
        Object[] values= {min,max};
        DomainModifier intervalModifier = modifierFact.createDomainModifier(
                                          ModifierFactory.INTERVAL_MODIFIER, values);

// Applique le modificateur de domaine au descripteur
descrVar.applyTypeModifier(intervalModifier);
    }

    /**
     * Cette methode annule le modificateur de domaine au descripteur de variable
     * specifie, variable qui doit etre de type integer. Le modificateur de domaine
     * doit avoir ete prealablement applique
     *
     * @param descrVar Le descripteur de variable de type integer auquel il faut
     * ajouter un domaine dont les bornes sont specifiees.
     * @param min la borne min de l'intervalle
     * @param max la borne max de l'intervalle
     */
    public static void cancelDomain(VariableDescriptor descrVar, Long min, Long max)
      throws InvalidModifierException {
      // Crée le modificateur de domaine
      ModifierFactory modifierFact = AromSetUp.getAromSystem().getModifierFactory();
      Object[] values= {min,max};
      DomainModifier intervalModifier = modifierFact.createDomainModifier(
ModifierFactory.INTERVAL_MODIFIER, values);

      // Applique le modificateur au descripteur
      descrVar.cancelTypeModifier(intervalModifier);
    }

 public static void main(String[] args){

    KnowledgeBase kb = createKb();
    //Recupere les entites AROM
    AMClass personne = kb.lookupClass("Personne");
    AMClass enfant = kb.lookupClass("Enfant");
    VariableDescriptor agePersonne =
      personne.lookupVariable("age").getVariableDescriptionFor(personne);
    VariableDescriptor ageEnfant =
      personne.lookupVariable("age").getVariableDescriptionFor(enfant);

    //Application de modificateurs de domaine à age pour Personne et Enfant
    try {
```


```
      applyDomain(agePersonne, new Long(0), new Long(130) );
      applyDomain(ageEnfant, new Long(0), new Long(18) );
      System.out.println(" *** Suite a l'application des domaines, le descripteur de age as-
socie a la classe personne a pour delta-type : "+agePersonne.getType() );
      System.out.println(" *** Suite a l'application des domaines, le descripteur de age as-
socie a la classe enfant a pour delta-type : "+ageEnfant.getType() );
      System.out.println("");

    } catch(InvalidModifierException ime) {
      System.out.println("Une erreur est survenue lors de l'application du modifica-
teur de domaine : ");
      System.out.println(ime.getMessage());
      System.exit(1);
    }

    try {
      applyDomain(agePersonne, new Long(20), new Long(130) );
    } catch(InvalidModifierException ime) {
      System.out.println(" *** Impossible d'appliquer le domain [20..130] au descrip-
teur de age pour la classe Personne.");
      System.out.println("");
    }

    try {
      cancelDomain(ageEnfant, new Long(0), new Long(18) );
      applyDomain(agePersonne, new Long(20), new Long(130) );
      System.out.println("Suite a la modification des domaines, le descripteur de age as-
socie a la classe personne a pour delta-type : "+agePersonne.getType() );
      System.out.println("Suite a la modification des domaines, le descripteur de age as-
socie a la classe enfant a pour delta-type : "+ageEnfant.getType() );
    } catch(InvalidModifierException ime) {
      ime.printStackTrace();
      System.out.println("Une erreur est survenue lors de l'application du modifica-
teur de domaine : ");
      System.out.println(ime.getMessage());
      System.exit(1);
    }

    AromSetUp.getAromSystem().cleanup();
  }

}// ChangeVarsDomain
```

Chapitre 6

Module de gestion de la mémoire

6.1. Objectifs du module de gestion de mémoire

Le module de gestion de mémoire (MGM) est le sous-système du système AROM qui permet de ne pas avoir en mémoire (en mémoire signifiant dans la mémoire de la machine virtuelle Java) l'intégralité des instances AROM d'une base de connaissances (par instance AROM on entend un objet Java représentant une instance de classe ou d'association, autrement dit les objets et les tuples).

6.2. Principes de fonctionnement

Le système de gestion de mémoire est invisible pour l'utilisateur du système AROM. La décision du retrait ou non d'une instance de la mémoire de la JVM doit est prise par le MGM. Cependant, il existe un moyen de contrôler en partie le fonctionnement du module de gestion de mémoire. Le MGM ne peut retirer de la mémoire de la machine virtuelle Java une instance qui est utilisée, autrement dit une instance pour laquelle il existe un pointeur la référençant. De ce fait, conserver un pointeur sur un ensemble d'instances, est un moyen de s'assurer que ces instances ne seront jamais retirées de la mémoire.

Chapitre 7

Les READER - WRITER pour AROM

7.1. Format AROM

Le modèle de représentation de connaissances AROM s'appuie sur une description textuelle dont la BNF est donnée dans le document des *Specifications*. Toute base de connaissances AROM peut donc être décrite grâce à cette BNF et peut être sauvegardée dans ce format, appelé format "txta" (texte AROM).

L'API d'AROM fournit des méthodes, comme nous le verrons par la suite, permettant de lire une base AROM au format *txta* et de créer les objets Java correspondant. De la même façon, il est possible de sauvegarder au format *txta* les objets Java représentant une base AROM. Mais, il est tout à fait envisageable qu'une base de connaissances AROM soit sauvegardée, et donc lue, dans un autre format que le format *txta*. En effet, il suffit que ce nouveau formalisme permette de créer des objets Java AROM qui respectent le méta-modèle AROM. Les fichiers *records* en sont un exemple mais d'autres formalismes, tel que XML ou un dérivé, peuvent également être utilisés.

7.2. Accès aux différents formats AROM

Si plusieurs formats sont autorisés pour représenter une base de connaissances AROM, il est nécessaire de permettre à l'utilisateur d'accéder à ces différents formats. De plus, le nombre et la nature des formats autorisés sont dépendant de l'implémentation. C'est pourquoi l'utilisateur doit se référer à la documentation de l'implémentation qu'il utilise pour connaître les formats autorisés avant de demander au système AROM un Objet permettant de lire (ou d'écrire) suivant ce format.

Pour cela, une méthode définie au niveau du système AROM permet de récupérer un `KBReader` (ou `KBWriter`) pour un format donné, spécifié au travers d'un identificateur. Cette technique permet d'étendre, sans modification de l'API d'AROM, les formats supportés. De plus, quel que soit le format dans lequel les bases AROM sont sauvegardées, l'utilisateur utilise toujours une même interface pour lire ces bases. Le système AROM autorise la lecture et l'écriture des bases AROM au format txta quelle que soit l'implémentation utilisée. Ces opérations se font par l'intermédiaire des `KBReader` et `KBWriter`

dits *Standard*. Par défaut, il est également possible d'accéder à un `KBReader` (et `KBWriter`) qui lit et écrit les instances AROM au format txta. La différence entre ce dernier Reader/Writer et le standard est que seules les instances (objets et tuples) sont traitées et, par conséquent, une technique plus rapide est utilisée. Ce Reader/Writer est recommandé lorsqu'un grand nombre d'instances a été défini. L'identificateur de ce format est "*instances*".

Avertissement

Section spécifique à l'Implémentation de Référence d'AROM.

Un autre format est supporté par cette implémentation. C'est le format *records* qui, comme pour le format *instances*, ne sauvegarde et ne lit que les instances AROM. Mais, à l'inverse d'*instances*, le format *records* sauvegarde les informations en binaire, donc dans un format illisible pour l'homme, et il utilise directement l'implémentation pour la création des objets AROM. Cette dernière spécificité permet de tirer parti de l'implémentation afin de réduire les temps de lecture et d'écriture.

III. API du système AROM version 2.0

Dans cette seconde partie, nous détaillons l'API proprement dite du système AROM dont les principales entités ont été présentées précédemment. Plutôt qu'une longue énumérations des classes et de leurs méthodes (disponible dans le javadoc de l'API), nous nous efforçerons de présenter de manière didactique *comment* l'API d'AROM doit être utilisée.

Les différents exemples donnés dans cette partie sont issus des exemples distribués avec AROM et sauvegardés sous le repertoire *samples*.

Chapitre 8

Lecture/Ecriture d'une base de connaissances AROM

8.1. Construction des entités

Comme nous l'avons vu précédement, voir la section *Entités AROM*, une relation de composition est établie entre les différentes entités AROM. Ainsi, les classes, les associations et les instances sont défines au niveau de la base de connaissances et, de la même façon, les variables et les rôles sont définis au niveau des structures, classes ou associations. Chacune de ces entités est donc créée depuis l'entité qui la définie, exception faite des instances qui sont créées depuis les structures auxquelles elles sont rattachées et non depuis la base de connaissances.

Le même schéma est appliqué pour les bases de connaissances. En effet, celles-ci sont créées depuis une tierce entité, un *Factory*. Un *factory* permet donc de créer de nouvelles base de connaissances AROM mais il est également le point d'entrée pour les opérations de lecture et d'écriture sur les bases AROM. Des exemples simples sont donnés ci-après afin d'illustrer le rôle de ce *factory*.

8.2. Création de bases de connaissances

La création d'une nouvelle BC est une opération simple. En effet, il suffit d'obtenir un objet *factory* et de lui demander la création d'une BC en lui spécifiant le nom de cette nouvelle base.

L'objet jouant le role de *factory* dans AROM est l'objet `AromSystem` que l'on obtient depuis l'objet `AromSetUp`, comme indiqué dans l'exemple qui suit. Pour créer un objet `KnowledgeBase` il faut appeler une méthode qui prend deux paramètres en entrée, le nom de la base à créer et un tableau d'objets. Ce dernier n'est pas utile pour la création de base de connaissances AROM, c'est pourquoi un tableau vide est passé dans l'exemple ci-dessous.

Exemple 8-1. Création de base de connaissances

```
/**
 * Cet exemple montre comment creer une nouvelle base de connaissances vide
 * de tout contenu. Le seul parametre necessaire est le nom de la base que
 * l'on souhaite creer.
 *
 * @author Veronique DUPIERRIS
 */
import arom.kr.model.KnowledgeBase;
import arom.AromSetUp;
import arom.kr.factory.AromSystem;
import arom.kr.model.EntityCreationException;

public class CreateKb {

  /**
   * Cette methode cree une nouvelle base de connaissances nommee "BaseTest".
   *
   * @return l'object KnowledgeBase representant la nouvelle base de
   * connaissances.
   */
  public static KnowledgeBase createNewKB(){
    //nom de la nouvelle base de connaissances.
    String kbName = "BaseTest";

    // recupere une instance de AromSystem qui est en charge de la
    // creation des objets KnowledgeBase.
    AromSystem kbFactory = AromSetUp.getAromSystem();

    // cree une nouvelle base de connaissancers AROM nommee BaseTest.
    // aucun autre attribut n'est necessaire lors de la creation de cette base.
    KnowledgeBase newKb = null;

    try{
      newKb = kbFactory.createKB(kbName, new Object[0]);
    } catch(EntityCreationException ece){
      System.out.println("Une erreur est survenue lors de la creation de la base : ");
      System.out.println(ece.getMessage());
      System.exit(1);
    }
    return newKb;
  }

  public static void main(String[] args){
    KnowledgeBase newKb = CreateKb.createNewKB();
    System.out.println("Nom de la nouvelle KB : "+newKb.getName());
    AromSetUp.getAromSystem().cleanup();
  }
}// CreateKB
```

8.3. Lecture de bases de connaissances

La lecture d'une base de connaissances est l'opération qui permet d'instancier une base de connaissances AROM à partir de sa description textuelle.

La lecture d'une base AROM est réalisée via un objet de la classe `KBReader`. Cette classe permet, d'une part, de créer une nouvelle base AROM et de lire son contenu depuis un flux d'entrée donné. Mais elle permet également de lire des informations qui viendront compléter une base AROM existante. Ces informations peuvent appartenir au domaine du modèle, c'est à dire de nouvelles classes ou associations, ou au domaine des instances, tuples ou objets. Un court descriptif de chacune des méthodes proposées dans `KBReader` est donné ci-après :

- `createKB ( InputStream, TestResult);`
  : Cette méthode crée une nouvelle KnowledgeBase et lit sa description à partir du flux d'entrée.
- `readContent ( AMAssociation, InputStream, TestResult);`
  : Cette méthode permet de lire des informations qui vont venir compléter l'objet AMAssociation spécifié. Cet objet doit obligatoirement existé.
- `readContent ( AMClass, InputStream, TestResult);`
  : Cette méthode permet de lire des informations qui vont venir compléter l'objet AMClass spécifié. Cet objet doit obligatoirement existé.
- `readContent ( KnowledgeBase, InputStream, TestResult);`
  : Cette méthode peut être utilisée dans plusieurs cas. 1_ Pour créer une nouvelle base de connaissances, dans le cas où la KnowledgeBase passée en paramètre est nulle. Le résultat est alors identique à l'appel de la méthode `createKB`. 2_ Pour ajouter de nouvelles informations à une base de connaissances existante. Dans ce cas, la KnowledgeBase passée en paramètre est la base existante à laquelle les nouvelles informations, descriptions de structures ou d'instances, vont être ajoutées.
- `readFacetModifier( Class, InputStream, TestResult);`
  : Cette méthode permet de créer un nouvel objet FacetModifier, dont la classe d'appartenance précise est spécifiée, à partir des informations qui vont être lues.

Mais avant de lire la description d'une base, ou d'un de ses composants, il est nécessaire de spécifier quel est le format utilisé. Pour cela, l'objet `AromSystem` offre la possibilité d'obtenir un objet `KBReader` pour un format donné. Par défaut, les seul formats connu sont le format *standard et le format instances. Pour obtenir un `KBReader` pour ces formats il suffit:*

- d'appeler la méthode `getStandardReader`

qui retourne un reader au format *standard*

- ou d'appeler la méthode `getReader`, en passant *"standard"* ou *"instances"* en paramètre.

C'est également cette seconde méthode qui sera utilisée pour obtenir des `KBReader` pour des formats autres. L'exemple qui suit est un exemple de creation/lecture d'une base de connaissances au format standard.

Exemple 8-2. Lecture d'une base de connaissances

```
/**
 * Cet exemple montre comment creer une base de connaissances a partir
 * de la description de cette base au format AROM. La description de la
 * base est lue dans un fichier nomme enseignement.txta fourni avec cet
 * exemple.
 *
 * @author Veronique DUPIERRIS
 */
import arom.kr.model.KnowledgeBase;
import arom.kr.factory.AromSystem;
import arom.AromSetUp;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import arom.kr.parser.KBReader;
import helix.util.TestResult;

public class ReadKb {

  /**
   * Instancie et retourne la base de connaissances dont le contenu est
   * decrit dans le fichier enseignement.txta.
   *
   * @return un objet KnowledgeBase representant la base AROM 'enseignement'.
   */
  public static KnowledgeBase getSampleKB(){

    KnowledgeBase enseignant = null;

    // recupere une instance de la classe AromSystem qui permet de creer
    // ou d'ouvrir des bases de connaissances AROM.
    AromSystem factory = AromSetUp.getAromSystem();

    try {
      // ouvre une connexion sur le fichier contenant la description de la base
      // 'enseignement'.
      FileInputStream fileEnseignant =
 new FileInputStream("enseignement.txta");
      KBReader reader = factory.getStandardReader();

      // Affiche les erreurs non bloquantes sur la sortie standard
      reader.displayWarning(true);
```


```
      //Utilise un objet TestResult pour tester si l'execution est correcte
      TestResult result = new TestResult();

      // lit le contenu de la base depuis la connexion et instancie cette
      // base en creant les objects Java correspondant aux entites de la base
      // de connaissances.
      enseignant = reader.createKB(fileEnseignant, result);
      if (!result.wasSuccessful())
 System.out.println(" Une erreur s'est produite "+result);

    } catch(FileNotFoundException fnfe){
      System.out.println("Le fichier enseignement.txta Introuvable");
      fnfe.printStackTrace();
    } catch(IOException ioe) {
      System.out.println("Erreur lors de la lecture de la base");
      ioe.printStackTrace();
    }
    return enseignant;
  }

  public static void main(String[] args){
    KnowledgeBase enseignement = ReadKb.getSampleKB();
    System.out.println("Nom de la KB : "+enseignement.getName());
    //Arrete la base correctement
    AromSetUp.getAromSystem().cleanup();
  }

}// ReadKb
```

8.4. Ecriture de bases de connaissances

L'écriture d'une base de connaissances consiste à traduire les différents objets Java d'une base de connaissances AROM selon un formalisme donné. Cette descripton peut être enregistrée et pourra subir l'opération inverse, voir la section *Lecture de bases de connaissances*, afin d'instancier de nouveau un objet `KnowledgeBase`.

L'écriture se fait par l'intermédiaire d'objets `KBWriter` et, tout comme pour la lecture, il existe des objets acceptant des formats AROM différents. L'obtention du `KBWriter` pour un format précis se fait en appelant la méthode `getWriter` d'`AromSystem` avec l'identificateur du format en paramètre.

Mais quelque soit le format choisi, l'objet `KBWriter` retourné permet d'écrire sur un flux de donné:

- la base de connaissances complète, c'est à dire l'ensemble de ses structures ainsi que l'ensemble de ses instances
- une structure donnée, c'est à dire l'ensemble des slots qui sont définis au niveau de cette structure.

- une instance donnée, c'est à dire l'ensemble des valeurs définies pour cette instance.

Exemple 8-3. Écriture de base de connaissance

```
import arom.AromSetUp;
import arom.kr.factory.AromSystem;
import arom.kr.model.KnowledgeBase;
import arom.kr.model.AMClass;
import arom.kr.parser.KBWriter;
import arom.kr.model.EntityCreationException;
import java.io.File;
import java.io.IOException;
import java.io.FileOutputStream;

/**
 *
 * Cette exemple montre comment ecrire (sauvegarder) la description d'une base de
 * connaissances au format AROM.
 *
 * @author Veronique DUPIERRIS
 */

public class WriteKb {

  /**
   * Cette methode cree une nouvelle base de connaissances nommee "BaseTest" et
   * cree un ensemble de classes dans cette base.
   *
   * @return l'object KnowledgeBase representant la nouvelle base de
   * connaissances.
   */
  public static KnowledgeBase createKb(){
    String kbName = "BaseTest";
    AromSystem kbFactory = AromSetUp.getAromSystem();
    KnowledgeBase kb = null;
    try{
      kb = kbFactory.createKB(kbName, new Object[0]);
      AMClass personne = kb.createClass("Personne", null);
      AMClass femme = kb.createClass("Femme", personne);
      AMClass homme = kb.createClass("Homme", personne);
    } catch(EntityCreationException ece){
      System.out.println("Une erreur est survenue lors de la creation de la base : ");
      System.out.println(ece.getMessage());
      //Arrete la base correctement
      AromSetUp.getAromSystem().cleanup();
      System.exit(1);
    }
    return kb;
  }
```

```
  /**
   * Cette methode enregistre la base de connaissances passee en parametre
   * dans le fichier newBase.txta.
   *
   * @param kb La base de connaissance a ecrire.
   */
  public static void writeKB(KnowledgeBase kb){

    // recupere une instance de la classe AromSystem qui permet
    // d'obtenir un Writer permettant l'ecriture d'une base AROM.
    AromSystem aromSystem = AromSetUp.getAromSystem();
    KBWriter kbWriter = aromSystem.getStandardWriter();

    File kbFile = new File("newBase.txta");
    try{
      // ouvre une connexion sur ce fichier
      FileOutputStream writer = new FileOutputStream(kbFile);
      //ecrit la base de connaissances au format AROM.
      kbWriter.write(kb, writer);
    }catch(IOException ioe){
      ioe.printStackTrace();
      //Arrete la base correctement
      AromSetUp.getAromSystem().cleanup();
      System.exit(1);
    }
  }

  public static void main(String[] args){
    KnowledgeBase kb = createKb();
    writeKB(kb);
    System.out.println("Nom de la KB : "+kb.getName());
    AMClass classFemme = kb.lookupClass("Femme");
    System.out.println("Nom de la Classe : "+classFemme.getName());
    //Arrete la base correctement
    AromSetUp.getAromSystem().cleanup();
  }


}// WriteKb
```

Chapitre 9

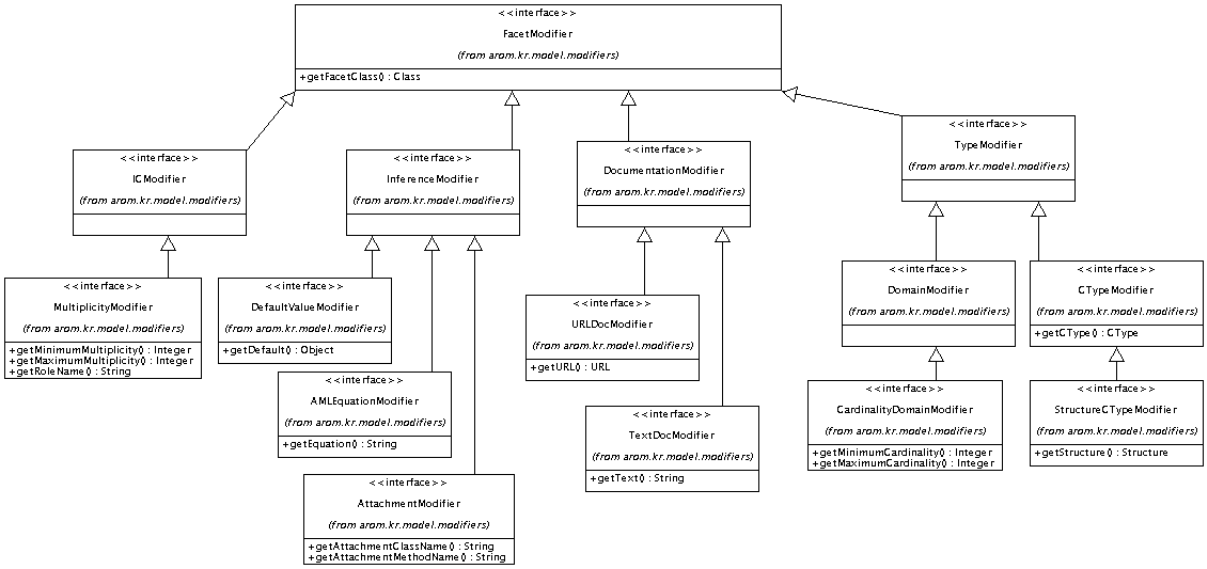
Modificateurs de facettes

9.1. Organisation

Les modificateurs de facettes, représentés sous forme d'interfaces, permettent de représenter un état ou plus précisément de transmettre une information aux entités définissant la facette, information qui est ensuite utilisée afin de modifiée l'état interne de la facette considérée. Le contenu d'un modificateur est fixé à sa création, les données qu'il contient ne peuvent pas être modifiées à posteriori (ce sont des objets *immuables*). Lorsque l'on souhaite modifier l'état d'une facette, on crée donc le modificateur correspondant et on l'applique au possesseur de la facette. Le modificateur ne pourra plus être utilisé à moins qu'une même modification doive être appliquée à plusieurs entités. Il est également possible d'annuler un modificateur. Seul un modificateur précédemment appliqué pourra être annuler. En effet si on applique un modificateur de domaine [1..10], il ne sera pas autorisé de demander à annuler le modificateur de domaine [1..3] pour obtenir au final le domaine [4..10]. Si l'utilisateur souhiate définir le domaine [4..10], il devra tout d'abord annuler le modificateur de domaine [1..10], puis appliquer le modificateur de domaine [4..10]. Le domaine hérité étant également à prendre en compte.

Les interfaces des modificateurs de facettes sont organisées hiérarchiquement, comme illustré dans la figure *Modificateurs de facettes*. Il existe une interface pour chacune des facettes définies dans AROM. Ces interfaces pouvant être spécialisées. Les différentes implémentations d'AROM peuvent donc, et même doivent, définir des objets implémentant ces interfaces. Ces objets seront accessibles à l'utilisateur via un `ModifierFactory` qui permet d'obtenir des objets typés par `DomainModifier` ou `DocumentationModifier`. Par conséquent, les implémentations peuvent créer des nouveaux types de modificateurs de facettes dans la mesure où ceux-ci héritent d'une des quatre interfaces `DomainModifier`, `InferenceModifier`, `ICModifier` ou `DocumentationModifier`. En effet, une implémentation pourra définir un modificateur *Expression Régulière*, par exemple, applicable aux types String. Ce nouveau modificateur devra implémenter l'interface `DomainModifier` afin d'être accessible à l'utilisateur, voir la section suivante. Des interfaces plus spécifiques sont également données, telle que `URLDocModifier`, pour les modificateurs *courants* qui devraient être implémentés dans toutes les implémentations.

Figure 9-1. Modificateurs de facettes



9.2. Création

La création des objets implémentant les différentes interfaces dérivées de `FacetModifier` est réalisée par l'intermédiaire de `ModifierFactory`, qui est lui-même accessible grâce à `AromSystem`. La ligne de code permettant d'obtenir un `ModifierFactory` est :

```
ModifierFactory factory = AromSetUp.getAromSystem().getModifierFactory();
```

Depuis cet objet, il est donc possible de créer des `DocumentationModifier`, des `InferenceModifier`, des `ICModifier` ou des `TypeModifier` spécifiques. Le type du modificateur à créer est spécifié par le paramètre `identificateur` passé à la méthode de création. En effet, chaque implémentation d'AROM doit pouvoir avoir ses propres modificateurs qui seront accessibles via l'identificateur qui leur sera associé. Les identificateurs attendus et les paramètres associés sont entièrement dépendant de l'implémentation, ils devront donc être documentés dans celles-ci. Néanmoins, il est prudent d'admettre que les modificateurs donnés en exemple ci-après seront définis dans toutes les implémentations.

Avertissement

l'Implémentation de Référence d'AROM défini tous ces modificateurs mais n'en ajoute aucun.

- Pour la facette de Documentation (ModifierFactory.createDocModifier(String identifier, Object[] args))

  - `URLDocModifier`: Définit une URL à associer à la documentation. Les paramètres nécessaire à sa création sont *identifier* = `ModifierFactory.URL_DOC_MODIFIER` et *args[0]* = une `URL`.
  - `TextDocModifier`: Définir un texte à associer à la documentation. Les paramètres nécessaire à sa création sont *identifier* = `ModifierFactory.TEXT_DOC_MODIFIER` et *args[0]* = une `String`.

- Pour la facette de Type (ModifierFactory.createCTypeModifier(CType ctype), ModifierFactory.createStructureCTypeModifier(Structure structure) ou ModifierFactory.createDomainModifier(String identifier, Object[] args))
  - `CTypeModifier`: Définit le type principal associé à la facette de type. Le paramètre nécessaire à sa création est *ctype* = le `CType` désiré .
  - `StructureCTypeModifier`: Définit le type principal associé à la facette de type pour un `Role`. Le paramètre nécessaire à sa création est *structure* = la `Structure` à partir de laquelle le CType sera créé .
  - `DomainModifier`.*Interval*: Définit un interval représentant le domain à fixer pour la facette. Les paramètres nécessaire à sa création sont *identifier* = `ModifierFactory.INTERVAL_MODIFIER` et *args[0]* = la borne inférieur, *args[1]* = la borne superieur, *args[2]* = un boolean specifiant si la borne inférieur est fermée ou non et *args[3]* = un boolean specifiant si la borne suprérieur est fermée ou non.
  - `DomainModifier`.*Set*: Définit un ensemble de valeurs représentant le domaine à fixer pour la facette. Les paramètres nécessaire à sa création sont *identifier* = `ModifierFactory.SET_MODIFIER` et *args[0]* = l'ensemble des valeurs du domaine.
  - `DomainModifier`.*Cardinality*: Définit la cardinalité pour les valeurs multivaluées. La cardinalité est un interval définissant le nombre minimum et maximum d'éléments que les valeurs multivaluées peuvent contenir. Ce modifieur n'a donc de signification que pour les `MultiValCT`. Les paramètres nécessaire à sa création sont *identifier* = `ModifierFactory.CARD_MODIFIER`, *args[0]* = le nombre minimun d'élements autorisé et *args[1]* = le nombre maximum d'élements autorisé ou null si + infinie.

- Pour la facette d'inférence (ModifierFactory.createInferenceModifier(String identifier, Object[] args))
  - `DefaultValueModifier`: Définit une valeur par défaut. Les paramètres nécessaire à sa création sont *identifier* = `ModifierFactory.DEFAULT_VALUE_MODIFIER` et *args[0]* = la valeur par défaut.
  - `AttachmentModifier`: Définit l'attachement procedural associé à une variable. Les paramètres nécessaire à sa création sont *identifier* = `ModifierFactory.ATTACHMENT_MODIFIER` et *args[0]* = une `String` définissant le nom complet de la classe Java à instancier (nom de package inclus) et *args[1]* = une `String` définissant le nom de la méthode à appeler.
  - `AMLEquationModifier`: Définit l'equation AML associé à une variable. Les paramètres nécessaire à sa création sont *identifier* = `ModifierFactory.AML_EQUATION_MODIFIER` et *args[0]* = une `String` représentant l'équation AML.

- Pour la facette de contrôle d'instanciation (ModifierFactory.createICModifier(String identifier, Object[] args))

- `MultiplicityModifier`: Définit la multiplicite pour un role d'une strutcure. Les paramètres nécessaire à sa création sont *identifier* = `ModifierFactory.MULTIPLICITY_MODIFIER` et *args[0]* = le nom du role auquel s'applique la multiplicite, *args[1]* = la valeur minimum de la multiplicite et *args[2]* = la valeur maximum de la multiplicite.

Chapitre 10

Traitements spécifiques aux structures

10.1. Création de structures.

Les structures, classes et associations, sont des entités nommées, voir chapitre *Représentation des entités du système*, qui appartiennent à l'espace de nommage défini par la base de connaissances. Par conséquent, la création d'une structure ne peut être réalisée qu'à partir de l'objet représentant la base de connaissances, l'objet `KnowledgeBase`. L'exemple qui suit illustre la création de classes.

Exemple 10-1. Création d'une classe

```
/**
 * Cette exemple montre comment creer une classe AROM.
 *
 * @author Veronique DUPIERRIS
 */
import arom.kr.model.KnowledgeBase;
import arom.kr.factory.AromSystem;
import arom.AromSetUp;
import arom.kr.model.EntityCreationException;
import arom.kr.model.AMClass;

public class CreateClass {

  /**
   * Cree une nouvelle base de connaissances nommee 'Test'.
   *
   * @return la nouvelle KnowledgeBase
   */
  public static KnowledgeBase newKB(){
    //nom de la nouvelle base de connaissances.
    String kbName = "Test";
```

```
  // recupere une instance la classe AromSystem qui est en charge de la
  // creation des objets KnowledgeBase.
  AromSystem kbFactory = AromSetUp.getAromSystem();

  // creer une nouvelle base de connaissancers AROM nommee Test.
  // aucun autre attribut n'est necessaire lors de la creation de cette base.
  KnowledgeBase newKb = null;

  try {
    newKb = kbFactory.createKB(kbName, new Object[0]);
  } catch (EntityCreationException ece){
    System.out.println("Une erreur est survenue lors de la creation de la base :");
    System.out.println(ece.getMessage());
    System.exit(1);
  }
  return newKb;
}

/**
 * Cette methode cree une nouvelle classe racine 'Cours' dans
 * la base test ainsi qu'une sous classe de la classe
 * cours, la classe 'Mathematiques'. La base modifiee est ensuite
 * retournee a l'appelant.
 *
 * @param kb La base de connaissances dans laquelle les classes
 * seront inserees.
 */
public static void createClasses(KnowledgeBase kb){

  //nom de la nouvelle classe a creer
  String className = "Cours";
  // 'Cours' est une classe racine, sans superClasse
  AMClass parent = null;
  try{
    AMClass classeCours = kb.createClass(className, parent);
    //cree une sous classe de la classe 'Cours'.
    //nom de la nouvelle classe
    className = "Mathematiques";
    //Mathematiques est une sous classe de cours
    parent = classeCours;
    AMClass classMath = kb.createClass(className, parent);
  }catch (EntityCreationException ece){
    System.out.println("Un erreur est survenue lors de la creation de classes :");
    System.out.println(ece.getMessage());
  }
}

public static void main(String[] args){
  KnowledgeBase kBase = newKB();
  createClasses(kBase);
  System.out.println("Nouvelle classes : "+kBase.lookupClass("Mathematiques").getFQName());
```

```
    AromSetUp.getAromSystem().cleanup();
  }
}// CreateClass
```

La création d'une association est identique à celle d'une classe. Seuls le nom et la structure parente sont spécifiés. Les différents slots de la nouvelle structure, rôles et/ou variables, sont définis par la suite, à partir de cette nouvelle structure qui représente l'espace de nommage des slots.

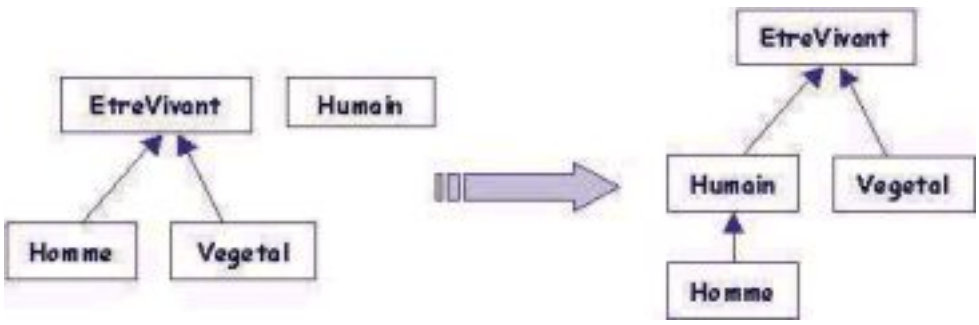
10.2. Spécialisation des structures

Les structures sont des entités qui sont organisées en hiérarchie et plus particulièrement en hiérarchie simple. Cela signifie que toute structure peut admettre une, et une seule, structure comme super structure, voir la section *Spécialisation d'entités*.

L'API d'AROM permet d'accéder à cette hiérarchie soit en lecture, pour connaître les structures qui en spécialisent une autre, soit en écriture afin de modifier la hiérarchie courante.

Lors de la création d'une structure, voir la section précédente *Création de structures.*, il est nécessaire de spécifier sa généralisation, ou super structure. Une valeur null signifiant que la structure en création est une structure racine. Mais il est également possible de modifier une hiérarchie existante en modifiant la généralisation d'une structure, comme illustré dans l'exemple qui suit.

Figure 10-1. Modification d'une hiérarchie de structures



L'objectif ici est d'insérer une structure dans une hiérarchie existante. Ce problème peut se décomposer en deux sous-problèmes. Tout d'abord, la structure `Humain` devient une *spécialisation* de la struture `EtreVivant`. Dans un second temps, il faudra changer la *généralisation* de la structure `Homme` de `EtreVivant` en `Humain`.

Exemple 10-2. Modification d'une hiérarchie de structures.

```
/**
 * Creer un ensemble de classes AROM dans la base de connaissances
 * passée en paramètre.
 *
 * @param kb La base de connaissances dans laquelle les classes
 * seront inserees.
 */
public static void createClasses(KnowledgeBase kb){

  AMClass etreVivant = null;
  AMClass humain =  null;
  AMClass homme = null;
  AMClass vegetal = null;

  try {
    // cree les structures de departs (Humain et EtreVivant,
    // specialise en Vegetal et Homme).
    etreVivant = kb.createClass("EtreVivant", null);
    humain = kb.createClass("Humain", null);
    homme = kb.createClass("Homme", etreVivant);
    vegetal = kb.createClass("Vegetal", etreVivant);
  } catch (EntityCreationException ece) {
    System.out.println("Une erreur est survenue lors de la creation des classes :");
    System.out.println(ece.getMessage());
    System.exit(1);
  }
}

/**
   * Modifie la hierarchie de structures d'une base. La modification
   * consiste a inserer la classe 'Humain' entre les classes  'EtreVivant'
   * et 'Homme'.
   *
   * @param kb la base de connaissances qui sera modifiee.
   */
public static void modifyHierarchy(KnowledgeBase kb) {

AMClass etreVivant = kb.lookupClass("EtreVivant");
AMClass humain =  kb.lookupClass("Humain");
AMClass homme = kb.lookupClass("Homme");

// la classe 'EtreVivant' devient la super-classe de
// 'Humain'.
AMClass[] superStructures = {etreVivant};
try {
    humain.setGeneralization(superStructures);
} catch (UnsupportedOperationException uoe){
    System.out.println("L'operation de modification de la super Structure est invalide: ");
```


```
    System.out.println(uoe.getMessage());
    System.exit(1);
} catch (InvalidSpecializationException ise){
  System.out.println("Le changement de super structure est invalide: ");
  System.out.println(ise.getMessage());
  System.exit(1);
}

// la classe 'Homme' devient sous classe de la classe 'Humain'.
try{
  humain.addSpecialization(homme);
} catch (InvalidSpecializationException ise){
  System.out.println("L'ajout d'une sous structure a echouer: ");
  System.out.println(ise.getMessage());
  System.exit(1);
}
   }

 public static void main(String[] args){
   KnowledgeBase kb = newKb();
   createClasses(kb);
   modifyHierarchy(kb);
 }
```

Cet exemple montre qu'il est possible de modifier une hiérarchie en modifiant soit la partie supérieure, c'est à dire en changeant la super structure d'une structure, soit, au contraire, en modifiant la partie inférieure, c'est à dire l'ensemble des sous structures d'une structure donnée.

De la même façon, la lecture d'une hiérarchie à partir d'une structure donnée peut consister soit à lire la super structure soit à lire l'ensemble de ses sous structures, comme c'est le cas dans l'exemple qui suit. En effet, cet exemple permet de lire toute la hiérarchie de classes d'une base de connaissance AROM à partir des classes racines de cette base.

Exemple 10-3. Lecture d'une hiérarchie de classes.

```
   /**
    * Lit et affiche sur la sortie standard la hierarchie des classes de la
    * base de connaissances passee en parametres.
    *
    * @param kb La base de connaissance a afficher
    */
   public static void printSpecialization(KnowledgeBase kb){

// recupere un iterateur sur toutes les classes racines de
// la base.
Iterator rootClasses = kb.rootClasses();
```

```
while(rootClasses.hasNext()){
    AMClass classe = (AMClass)rootClasses.next();
    // affiche la hierarchie propre a une classe.
    printClassLevel(classe, 1);
}
   }

   /**
    * Cette methode affiche sur la sortie standard la classe passee
    * en parametre puis affiche récursivement toute ses sous classes.
    *
    * @param classe la classe qui doit etre affichee
    * @param level le niveau de la classe dans la hierarchie globale de
    * la base de connaissances.
    */
   private static void printClassLevel(AMClass classe, int level){
//Affiche la classe specifie, selon son niveau dans la hierarchie
for(int i=0; i<level; i++)
    System.out.print("  ");
System.out.println("** Classe "+classe.getName());

// Affiche les niveaux suivants
int newLevel = ++level;
Iterator specializedClasses = classe.specializations();
while(specializedClasses.hasNext()){
  AMClass nextSpecialization = (AMClass)specializedClasses.next();
  printClassLevel(nextSpecialization, newLevel);
}
   }

 public static void main(String[] args){
   KnowledgeBase kb = newKb();
   createClasses(kb);
   modifyHierarchy(kb);
   printSpecialization(newKb);
 }
```

10.3. L’accès aux slots

Toute structure, classe ou association, peut définir un ensemble de slots. Actuellement, seules les variables sont des slots valides pour les classes, par contre, les associations acceptent comme slots des variables ou des rôles. Les méthodes relatives aux manipulations, création et modification, des slots seront discutées dans la section, *Traitements spécifiques aux slots*, qui suit.

Pour chaque structure, il est possible :

- de rechercher un slot spécifique grâce aux méthodes `lookupSlot()`, `lookupVariable()` et `lookupRole()`.
- de connaître l'ensemble des slots qui sont définis par l'intermédiaire des méthodes `slots()`, `variables()` ou `roles()` qui retournent des *iterateurs*.
- de connaître le nombre de slots définis avec les méthodes `slotCount()`, `variableCount()` ou `roleCount()`.

Les différentes méthodes relatives aux rôles n'étant évidemment valides que pour les associations.

10.4. L'accès aux instances

Les instances sont des entités AROM rattachées à des structures. En effet, les instances représentent les individus modélisés par les différentes structures d'une base de connaissances AROM. Les instances regroupent les objets, qui sont les instances des classes, et les tuples, qui sont les instances des associations. La section *Traitements spécifiques aux instances* discute des méthodes de création et de modification propres aux instances.

En ce qui concerne les structures, pour chacune d'elle il est possible :

- de connaître l'ensemble des instances qui lui sont rattachées grâce à la méthode `instances()`. Celle-ci retourne un *iterateur* sur l'ensemble des instances de la structure. Il est à noter que les instances créées à partir de sous-structures sont également prises en comptes. En effet, Si la structure B spécialise la structure A et que B définit une instance b, cette instance sera également instance de A.
- de connaître le nombre d'instances composant cet ensemble, grâce à la méthode `size()`.
- de savoir si une instance est rattachée ou non à cette structure par l'intermédiaire de la méthode `contains()`.

Deux autres méthodes, `addInstance()` et `removeInstance()`, permettent d'attacher une instance existante à la structure ou, au contraire, de détacher une instance de cette structure. Dans ce dernier cas, l'instance est en fait supprimée de la base puisqu'il n'est pas autorisé d'avoir une instance qui n'est rattachée à aucune structure dans AROM.

Chapitre 11

Traitements spécifiques aux slots

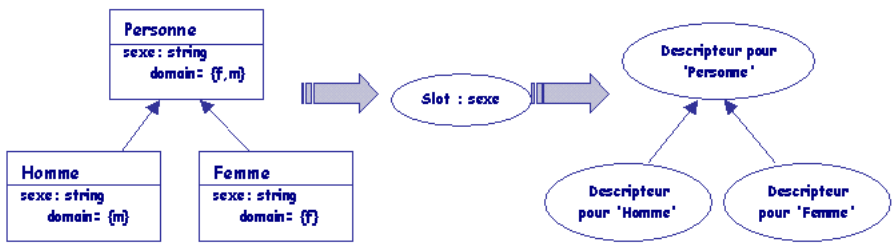
11.1. Slots et Facettes

Seule une présentation sommaire des slots et facettes est faite ici. Pour plus d'information se reporter à la section *Slots*.

Les slots sont définis au niveau des structures afin de caractériser celles-ci. Chaque slot possède un nom, il peut être valué et il possède une description pour chacune des structures de la hiérarchie considérée.

La description d'un slot est défini par l'ensemble des facettes qui sont attachées au slot. Ces facettes décrivent des propriétés du slot, telles que la documentation, le type ou l'inférence. Ainsi, dans l'exemple de la figure *Structures, slots et descriptions de slots*, à la variable *sexe* correspond un seul slot, mais une description différente de cette variable est définie pour chacune des classes `Personne`, `Homme` et `Femme`.

Figure 11-1. Structures, slots et descriptions de slots



Les descripteurs répertorient les différents modificateurs de facette qui ont été appliqués aux différentes facettes d'un slot. Lorsque l'on souhaite connaître l'état d'un slot pour une structure donnée on obtient en fait l'état de l'ensemble des facettes du slot, pour la structure en question.

Toutes les manipulations sur les slots sont donc en réalité des manipulations sur les facettes du slot, et plus précisemment sur les facettes du descripteur du slot. De plus, la modification d'une facette n'est pas directement réalisable, il est nécessaire d'appliquer ou d'annuler des modificateurs de facettes sur un descripteur de slot pour modifier celui-ci.

11.2. Création de slots

L'espace de nommage des slots est défini par une structure donc la création de slots est réalisée depuis ces structures. L'exemple qui suit illustre la création de la variable *sexe* définie pour la classe `Personne`. Quelque soit le slot créé, il est nécessaire de spécifier son type, par l'intermédaire d'une facette `CType-Modifier`, comme paramètre d'entrée de la méthode de création. D'autre facettes, telles que la facette de domaine, de documentation ou de valeur par défaut, peuvent également être spécifiées lors de la création d'un slot. Dans l'exemple qui suit, seule une facette de domaine est précisée, en plus du type, pour la nouvelle variable *sexe*.

Exemple 11-1. Création d'un slot

```
/**
 * Cet exemple montre comment creer une nouvelle variable dans une classe
 * AROM avec l'API.
 *
 * @author Veronique DUPIERRIS
 */
import arom.kr.model.KnowledgeBase;
import arom.kr.factory.AromSystem;
import arom.AromSetUp;
import arom.kr.model.AMClass;
import arom.kr.model.EntityCreationException;
import arom.kr.factory.ModifierFactory;
import arom.kr.model.type.TypeSystem;
import arom.kr.model.type.CType;
import arom.kr.model.modifiers.CTypeModifier;
import arom.kr.model.modifiers.DomainModifier;
import arom.kr.model.modifiers.FacetModifier;
import arom.kr.model.Variable;
import arom.kr.model.InvalidNameException;
import java.util.Collection;
import java.util.ArrayList;

public class CreateSlot {

  /**
   * Cette methode cree une nouvelle base de connaissances nommee "BaseTest" et
   * cree un ensemble de classes dans cette base.
   *
   * @return l'object KnowledgeBase representant la nouvelle base de
```


```
 * connaissances.
 */
public static KnowledgeBase createKb(){
  String kbName = "BaseTest";
  AromSystem kbFactory = AromSetUp.getAromSystem();
  KnowledgeBase kb = null;
  try{
    kb = kbFactory.createKB(kbName, new Object[0]);
    AMClass personne = kb.createClass("Personne", null);
    AMClass femme = kb.createClass("Femme", personne);
    AMClass homme = kb.createClass("Homme", personne);
  } catch(EntityCreationException ece){
    System.out.println("Une erreur est survenue lors de la creation de la base : ");
    System.out.println(ece.getMessage());
    System.exit(1);
  }
  return kb;
}

/**
 * Cette methode recupere une base de connaissances definissant
 * une classe racine 'Personne' et deux autre classes, 'Homme'
 *  et 'Femme' qui la specialisent.
 * La variable 'sexe' est ensuite associee a la classe Personne.
 * La base de connaissances ainsi configuree est alors retournee
 * a l'appelant.
 *
 * @param personne la classe AROM 'Personne' dans laquelle le slot
 * va etre ajoute.
 */
public static void createSlot(AMClass personne){

  // Cree le slot 'sexe' avec son domaine dans la classe personne
  // Cree les differentes facettes de la variable :

  //Recupere le module de type afin d'obtenir le type String associe.
  TypeSystem kbTypeSystem = AromSetUp.getAromSystem().getTypeSystem();
  CType sexeType = kbTypeSystem.getStringCType();

  //Cree la facette de type pour le slot "sexe".
  ModifierFactory modifierFact =
    AromSetUp.getAromSystem().getModifierFactory();
  CTypeModifier cType = modifierFact.createCTypeModifier(sexeType);


  // Cree le domaine associe au slot "sexe". Ce domaine est l'ensemble de valeurs:
  Object[] values = { new ArrayList() };
  ((Collection)values[0]).add(new String("f"));
  ((Collection)values[0]).add(new String("m"));
  // Cree la facette correspondant au domaine
  DomainModifier domain = modifierFact.createDomainModifier(
    ModifierFactory.SET_MODIFIER,
```

```
        values);

      // Construit le tableau des facettes necessaire a la creation du slot
      FacetModifier[] facettes = new FacetModifier[2];
      facettes[0] = cType;
      facettes[1] = domain;

      //Cree la variable
      try {
        Variable sexe = personne.createVariable("sexe", facettes);
      } catch(InvalidNameException ine){
        System.out.println("Le nom de la nouvelle variable est invalide: ");
        System.out.println(ine.getMessage());
      } catch(EntityCreationException ece){
        System.out.println("Une erreur est survenue lors de la creation de la variable: ");
        System.out.println(ece.getMessage());
      }
    }

    public static void main(String[] args){

      KnowledgeBase kb = createKb();
      AMClass personne = kb.lookupClass("Personne");
      createSlot(personne);
      System.out.println("Nouvelle classes : "+ personne.getFQName());
      System.out.println("Slots : "+personne.lookupSlot("sexe").getFQName());
      AromSetUp.getAromSystem().cleanup();

    }

  }// CreateSlot
```

11.3. Modification d’un slot

La modification d’un slot équivaut à modifier une ou plusieurs facettes du descripteur de slot et nécessite la création d’un modificateur de facette. Une facette est un objet qui représente l’état d’une des propriétés d’un slot, or la modification de la représentation d’une propriété ne se traduit pas par la modification de la propriété elle-même. C’est pourquoi lorsque l’on souhaite apporter une modification à l’une des facettes d’un slot, comme lorsque l’on souhaite définir une nouvelle facette, il faut appliquer un objet spécifique au slot, objet appelé *modificateur de facette* et qui décrit quels sont les changements désirés.

A chaque descripteur de slot est potentiellement associée une facette de type, une facette de documentation et une facette d’inférence. Plusieurs méthodes de lecture de l’état d’une facette pour un descripteur de slot donné sont proposés par l’API d’AROM :

- `lookupTypeModifier()`, `lookupInferenceModifier()` et `lookupDocumentationModifier()` qui permettent de lire les modificateurs de facette de type, d'inference ou de documentation associés à un slot donné.
- `getType()`, `getInference()` et `getDocumentation()` qui permettent, quant à eux, d'obtenir un objet précis, spécialisant l'objet *Facet*, associé au slot considéré.

Mais quelque soit la méthode choisie, et par conséquent l'objet obtenu en retour, il n'est pas autorisé de modifier directement cet objet. L'exemple qui suit illustre les deux cas de modification que sont la définition d'une nouvelle facette et la modification d'une facette existante.

Exemple 11-2. Modification de slots

```
import arom.AromSetUp;
import arom.kr.factory.AromSystem;
import arom.kr.factory.ModifierFactory;
import arom.kr.model.AMClass;
import arom.kr.model.KnowledgeBase;
import arom.kr.model.type.TypeSystem;
import arom.kr.model.type.CType;
import arom.kr.model.Variable;
import arom.kr.model.InvalidNameException;
import arom.kr.model.VariableDescriptor;
import arom.kr.model.EntityCreationException;
import arom.kr.model.NoSuchEntityException;
import arom.kr.model.modifiers.InvalidModifierException;
import arom.kr.model.modifiers.CTypeModifier;
import arom.kr.model.modifiers.FacetModifier;
import arom.kr.model.modifiers.InferenceModifier;
import java.util.Iterator;
import java.util.Vector;
import arom.kr.model.modifiers.DomainModifier;
import arom.kr.model.modifiers.DefaultValueModifier;
import arom.kr.model.modifiers.TypeModifier;

/**
 * ModifySlot.java
 *
 *
 * @author Veronique DUPIERRIS
 */

public class ModifySlot {

  /**
   * Cette methode cree une nouvelle base de connaissances nommee "BaseTest" et
   * cree un ensemble de classes dans cette base. Cet ensemble comprend : la
   * classe racine 'Personne' specialisee par les classes 'Homme'et 'Femme'.
   * La variable 'sexe' est associee a la classe 'Personne'.
   *
```

```
 * @return l'object KnowledgeBase representant la nouvelle base de
 * connaissances.
 */
public static KnowledgeBase createKb(){
  String kbName = "BaseTest";
  AromSystem kbFactory = AromSetUp.getAromSystem();

  // Recupere l'objet permettant de creer les modifieurs de facettes
  ModifierFactory modifierFactory = AromSetUp.getAromSystem().getModifierFactory();

  KnowledgeBase kb = null;
  try{
    kb = kbFactory.createKB(kbName, new Object[0]);
    AMClass personne = kb.createClass("Personne", null);
    AMClass femme = kb.createClass("Femme", personne);
    AMClass homme = kb.createClass("Homme", personne);
    //Cree la variable sexe
    FacetModifier[] modifiers = new FacetModifier[1];
    modifiers[0] = modifierFactory.createCTypeModifier(kbFactory.getTypeSystem().getStringCTy
    personne.createVariable("sexe", modifiers);
  } catch(EntityCreationException ece){
    System.out.println("Une erreur est survenue lors de la creation de la base : ");
    System.out.println(ece.getMessage());
    //Arrete le System Arom proprement
    AromSetUp.getAromSystem().cleanup();
    System.exit(1);
  } catch(InvalidNameException ine) {
    System.out.println("Une erreur est survenue lors de la creation de la base: ");
    System.out.println(ine.getMessage());
    //Arrete le System Arom proprement
    AromSetUp.getAromSystem().cleanup();
    System.exit(1);
  }
  return kb;
}

/**
 * Cette modifie le domaine de la variable 'sexe' pour la classe 'Femme'
 * et une valeur par default est definie pour la classe 'Homme'.
 *
 * @param kb la base de connaissance contenant les classes 'Personne',
 * 'Homme' et 'Femme'
 */
public static void modifySlots(KnowledgeBase kb){

  AMClass homme = kb.lookupClass("Homme");
  AMClass femme = kb.lookupClass("Femme");
  Variable sexe = femme.lookupVariable("sexe");

  // Recupere l'objet permettant de creer les modifieurs de facettes
  ModifierFactory modifierFactory = AromSetUp.getAromSystem().getModifierFactory();
```


```
// Modifie le domaine de la variable 'sexe':
Vector newValues =  new Vector();
newValues.add("f");

// Cree un nouveau modificateur de domaine (definit un ensemble) pour la facettes
// de type.
Object[] args = { newValues };
DomainModifier nvDomain = modifierFactory.createDomainModifier(ModifierFactory.SET_MODIFIER

//applique ce modificateur a la variable sexe pour la classe femme
// ... donc au descripteur de la variable associe a la classe Femme.
try {
  VariableDescriptor femmeSexe = sexe.getVariableDescriptionFor(femme);
  femmeSexe.applyTypeModifier(nvDomain);
} catch(NoSuchEntityException nsee){
  System.out.println("La variable n'est pas defini pour la classe 'femme': ");
  System.out.println(nsee.getMessage());
  //Arrete le System Arom proprement
  AromSetUp.getAromSystem().cleanup();
  System.exit(1);
} catch(InvalidModifierException ime) {
  System.out.println("Une erreur est survenue lors de l'application du modifier: ");
  System.out.println(ime.getMessage());
  //Arrete le System Arom proprement
  AromSetUp.getAromSystem().cleanup();
  System.exit(1);
}

// definit une valeur par defaut pour la variable 'sexe' de la classe 'Homme'
// cree un nouveau modificateur de facettes
args = new Object[1];
args[0] = "h";
InferenceModifier defaultValueModifier = modifierFactory.createInferenceModifier(ModifierFa

// applique ce modificateur au descripteur de la variable sexe
// pour la classe Homme.
try {
  VariableDescriptor hommeSexe = sexe.getVariableDescriptionFor(homme);
  hommeSexe.applyInferenceModifier(defaultValueModifier);
} catch(NoSuchEntityException nsee){
  System.out.println("La variable n'est pas definie pour la classe 'homme': ");
  System.out.println(nsee.getMessage());
  //Arrete le System Arom proprement
  AromSetUp.getAromSystem().cleanup();
  System.exit(1);
} catch(InvalidModifierException ime) {
  System.out.println("Une erreur est survenue lors de l'application du modifier: ");
  System.out.println(ime.getMessage());
  //Arrete le System Arom proprement
  AromSetUp.getAromSystem().cleanup();
  System.exit(1);
}
```

```
  }

  public static void main(String[] args){

    KnowledgeBase kb = createKb();
    modifySlots(kb);
    AMClass homme = kb.lookupClass("Homme");
    Variable sexe = homme.lookupVariable("sexe");

    if(homme == null) {
      System.out.println("la classe Homme est introuvable !");
      //Arrete le System Arom proprement
      AromSetUp.getAromSystem().cleanup();
      System.exit(1);
    }

    VariableDescriptor hommeSexe = sexe.getVariableDescriptionFor(homme);
    InferenceModifier[] hommeDefault = hommeSexe.lookupInferenceModifier(DefaultValueModifier.c
    if(hommeDefault==null || hommeDefault.length == 0){
      System.out.println("La valeur par defaut de la variable 'sexe' est inexistante pour la cl
    } else {
      String defVal = (String) ((DefaultValueModifier) hommeDefault[0]).getDefault();
      System.out.println("valeur par default = "+defVal);
    }

    AMClass femme = kb.lookupClass("Femme");
    if(femme == null) {
      System.out.println("la classe Femme est introuvable !");
      //Arrete le System Arom proprement
      AromSetUp.getAromSystem().cleanup();
      System.exit(1);
    }

    VariableDescriptor femmeSexe = sexe.getVariableDescriptionFor(femme);
    TypeModifier[] femmeDomain = femmeSexe.lookupTypeModifier(DomainModifier.class);
    if(femmeDomain == null || femmeDomain.length == 0 ){
      System.out.println("Le domaine de la variable 'sexe' est inexistant pour la classe Femme"
      //Arrete le System Arom proprement
      AromSetUp.getAromSystem().cleanup();
      System.exit(1);
    }

    System.out.println("valeur(s) du domaine :"+femmeDomain[0]);

    //Arrete le System Arom proprement
    AromSetUp.getAromSystem().cleanup();
  }

}// ModifySlot
```

Dans cet exemple les deux méthodes utilisés, `applyTypeModifier()` et `applyInferenceModifier()`, permettent d'appliquer un modificateur de facette de type et d'inférence à un descripteur de slot donné. Il existe, de la même façon, une méthode permettant d'appliquer un modificateur de facette de documentation à un descripteur de slot.

Des méthodes similaires sont également définies afin de permettre d'annuler un modificateur qui a été précédemment appliqué, ce qui peut se traduire dans certain cas à supprimer une propriété donnée. Ces méthodes sont `cancelTypeModifier()`, `cancelInferenceModifier()` et `cancelDocumentationModifier()` pour les facettes de type, d'inference et de documentation respectivement.

Chapitre 12

Traitements spécifiques aux instances

12.1. Création d’instances

Les instances sont les seules entités AROM qui ne sont pas créées depuis l’entité représentant leur espace de nommage, c’est à dire la base de connaissances. En effet, les instances sont créées à partir des structures auxquelles elles sont rattachées, voir section *Les instances AROM*. L’exemple qui suit illustre cette création d’instances, objets et tuples.

Exemple 12-1. Création d’instances, objet et tuple

```
/**
 * CreateInstance.java
 *
 * Cet exemple montre comment creer des objets AROM (des instances de
 * classes AROM) et des tuples (des instances d’associations AROM).
 *
 * @author Veronique DUPIERRIS
 * $Id: CreateInstance.java,v 1.7 2001/12/06 11:55:05 fauchera Exp $
 */
import arom.kr.model.AMClass;
import arom.kr.model.AMAssociation;
import arom.kr.model.AMObject;
import arom.kr.model.EntityCreationException;
import arom.kr.model.Link;
import arom.kr.model.AMTuple;
import arom.kr.model.KnowledgeBase;
import java.util.Iterator;
import arom.kr.factory.AromSystem;
import arom.AromSetUp;
import arom.kr.model.modifiers.FacetModifier;
```

```
import arom.kr.model.InvalidNameException;

public class CreateInstance {

  /**
   * Cette methode cree une nouvelle base de connaissances nommee "BaseTest" et
   * cree un ensemble de classes dans cette base. Cet ensemble comprend : la
   * classe racine 'Personne' specialisee par les classes 'Homme'et 'Femme'.
   * Une association 'Couple' relie les objets de ces deux classes.
   *
   * @return l'object KnowledgeBase representant la nouvelle base de
   * connaissances.
   */
  public static KnowledgeBase createKb(){
    String kbName = "BaseTest";
    AromSystem kbFactory = AromSetUp.getAromSystem();
    KnowledgeBase kb = null;
    try{
      kb = kbFactory.createKB(kbName, new Object[0]);
      AMClass personne = kb.createClass("Personne", null);
      AMClass femme = kb.createClass("Femme", personne);
      AMClass homme = kb.createClass("Homme", personne);
      AMAssociation couple = kb.createAssociation("couple", null);
      // --- cree les roles pour l'association
      // Aucun modifier n'est precise, la classe typant le role etant explicite.
      FacetModifier[] modifiers = new FacetModifier[0];
      couple.createRole("mari", homme, modifiers);
      couple.createRole("femme", femme,  modifiers);
    } catch(EntityCreationException ece){
      System.out.println("Une erreur est survenue lors de la creation de la base : ");
      System.out.println(ece.getMessage());
      System.exit(1);
    } catch(InvalidNameException ine) {
      System.out.println("Une erreur est survenue lors de la creation de la base: ");
      System.out.println(ine.getMessage());
      System.exit(1);
    }
    return kb;
  }

    /**
     * Cree une instance pour chacune des classes 'Homme' et 'Femme'.
     *
     * @param la base de connaissances contenant les classes 'Homme', 'Femme'
     * et l'association 'Couple'.
     */
  public static void createInstances(KnowledgeBase kb){

 AMClass homme = kb.lookupClass("Homme");
 AMClass femme = kb.lookupClass("Femme");
 AMAssociation couple = kb.lookupAssociation("couple");
```


```
//creer un objet pour chacune des classes
AMObject leon = null;
AMObject ginette = null;
try{
  leon = homme.createObject("Leon");
  ginette  =femme.createObject("Ginette");
} catch( EntityCreationException ece) {
    System.out.println("Une erreur s'est produite lors de la creation des objets: ");
    System.out.println(ece.getMessage());
    System.exit(1);
}
// creer un lien utilise pour la creation d'un tuple
// de l'association couple et identifiant pour chaque
// role de cette association les objets.
String[] roleNames = new String[2];
roleNames[0] = "mari";
roleNames[1] = "femme";
AMObject[] roleValues = new AMObject[2];
roleValues[0] = leon;
roleValues[1] = ginette;
Link lienDuCouple = new Link(roleNames,roleValues);
// creer un tuple
try{
  AMTuple nouveauCouple = couple.createTuple(lienDuCouple);
} catch( EntityCreationException ece) {
  System.out.println("Une erreur s'est produite lors de la creation du tuple: ");
  System.out.println(ece.getMessage());
  System.exit(1);
}
   }


   public static void main(String[] args){
KnowledgeBase kb = createKb();
createInstances(kb);
Iterator allClasses = kb.rootClasses();
while(allClasses.hasNext()){
  AMClass nextClass = (AMClass) allClasses.next();
  System.out.print("La classes "+ nextClass.getName());
  System.out.println(" a pour instance(s) :"  );
  Iterator allInsts = nextClass.instances();
  while(allInsts.hasNext())
    System.out.println("- "+ ((AMObject) allInsts.next()).getName());
}
Iterator allAssocs = kb.rootAssociations();
while(allAssocs.hasNext()){
  AMAssociation nextAssoc = (AMAssociation) allAssocs.next();
  System.out.print("L'assoc "+ nextAssoc.getName());
  System.out.println(" a pour instance(s) :" );
  Iterator allInsts = nextAssoc.instances();
  while(allInsts.hasNext())
    System.out.println("- "+ (AMTuple) allInsts.next());
```

```
 }
 AromSetUp.getAromSystem().cleanup();
    }
}// CreateSlot
```

Chapitre 13

Mécanismes d'inférence

13.1. Attachement Procédural

L'attachement procédural est un mécanisme d'inférence des valeurs des variables des instances d'une base AROM. Ce mécanisme permet d'associer à une variable et à une classe AROM une méthode Java réalisant le calcul de la valeur de cette variable pour les différentes instances de la classe.

L'attachement procédural est réalisé en désignant explicitement la classe Java et la méthode de cette classe qui devra être appelée par le système AROM pour inférer la valeur de la variable. La classe et la méthode sont toutes deux désignées par leur nom : le nom complet (fully qualified name) de la classe et le nom de la méthode.

13.1.1. Propriétés de la méthode attachée

La méthode appelées par le système AROM doit respecter les propriétés suivantes :

- la méthode retourne une valeur sous la forme d'un objet (type `java.lang.Object`).
- la méthode peut prendre posséder un argument de type `arom.kr.model.AMInstance`. Si c'est effectivement le cas, ce paramètre est utilisé par le système AROM pour passer à la méthode l'instance AROM pour laquelle le calcul est déclenché.
- Enfin, la méthode peut être une méthode d'instance ou une méthode statique. Dans le cas d'une méthode statique, il est inutile de créer une instance de la classe à laquelle la méthode Java appartient. Dans le cas d'une méthode d'instance, il faut que le système AROM puisse récupérer une référence à une instance de la classe Java à laquelle cette méthode appartient. Pour ce faire, le système tente d'appeler la méthode statique `getInstance` de cette classe, et cette méthode doit retourner au système une instance qui sera utilisée pour effectuer l'appel. De cette manière, c'est le concepteur de la classe attachée qui décide de la manière de créer les instances utilisées par l'attachement procédural.

13.1.2. Attachement procédural version 1.0 vs version 2.0

La manière dont on spécifie l'attachement procédural, telle qu'elle vient d'être décrite, est très différente de celle de la version 1.0. Dans la version 1.0, l'association (classe, variable)-méthode Java était basé sur une convention de nommage de la classe java réalisant le calcul de la valeur. De cette manière, le concepteur de la base de connaissances devait uniquement indiquer (par un booléen) si une variable peut-être calculé ou non via un attachement procédural. Le nom de la classe Java associée au calcul de la variable était alors automatiquement déterminé par le système à partir du nom de la variable et du nom de la classe dans laquelle l'attachement était déclaré.

Ainsi, pour calculer par attachement procédural la valeur de la variable `salaire` de la classe `Professeur`, le système AROM recherchait automatiquement une classe `PAProfesseur` (PA pour Procedural Attachment, suivi du nom de la classe AROM : `Professeur`) et appelait la méthode `getSalaire()` de cette classe.

Ce mode de fonctionnement a été abandonné au profit d'une déclaration plus explicite de la classe et de la méthode qui doit être appelée. Toutefois, le parser standard reconnaît l'ancienne forme de la déclaration de l'attachement et reproduit l'ancien fonctionnement en reconstruisant le nom de la classe Java et le nom de la méthode à partir du nom de la classe AROM et du nom de la variable.

13.1.3. Exemple

L'exemple ci-dessous illustre le fonctionnement de l'attachement procédural en utilisant des méthodes statiques.

Exemple 13-1. Attachement procédural de la variable salaire de la classe Professeur

```
/**
 * Attachement procedural permettant le calcul de la variable salaire
 * de la classe AROM Professeur.
 *
 * @author Christophe Bruley
 */

public class PAProfesseur {

 private static PAProfesseur instance_;

  public static PAProfesseur getInstance() {
    if (instance_ == null)
      instance_ = new PAProfesseur();
    return instance_;
  }

  private PAProfesseur (){

  }
  public Object getSalaire() {
```


```
      return new Long(210000);
    }

  }// PAProfesseur
```

IV. Annexe

Chapitre 14

D’AROM version 1 à AROM version 2

14.1. Modèle de représentation de connaissances

Inchangé.

14.2. Représentation des objets AROM

Un objet AROM est représenté par une instance d’objet Java. Ce choix n’est pas anodin du point de vue de la part de la mémoire dédiée aux instances AROM.

14.3. API

14.3.1. API vs implémentation

varier les implémentations

faciliter l’évolution vers une architecture client-serveur.

Eviter les écueils de la version 1.0 dans laquel l’API n’est autre que la partie publique de l’implémentation.

14.3.2. Définition d’autres modèles

L’API doit faciliter la définition de modèles de représentation de connaissances autres qu’AROM. Par exemple le modèle de tâches.

14.3.3. Conception de l'API

Les interfaces de l'API correspondent à ce que nous avons identifié comme des "rôles" ou des "fonctionnalités" des entités d'une base de connaissances.

14.4. Développement

Jeux de tests systématiques.

Gestion des versions.

Documentation (articles, présentations, javadoc, guide implementation/utilisateur, FAQ, bug report).

14.5. Implémentation

14.5.1. Modularité

Assurer l'indépendance entre les différentes fonctionnalités (modules) du noyau. C'est par exemple le cas pour :

- module de types
- module de gestion de mémoire
- module d'interprétation algébrique

14.5.2. Extensibilité

Faciliter l'implémentation d'autres modèles de représentation de connaissances par héritage ou composition des classes iméplementant le noyau AROM.

Références bibliographiques

[Che76] « The Entity-relationship Model: Towards a Unified View of Data », P. Chen, 1984, 9-36, *ACM TODS*, 1, (1).

[RBP+91] *Object-Oriented Modeling and Design*, J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, et W. Lorensen, 1991, Prentice Hall.

[RJB99] *The Unified Modeling Language Reference Manual*, J. Rumbaugh, I. Jacobson, et G. Booch, 1999, Addison-Wesley.

[BW77] *An Overview of KRL, Knowledge Representation Language*, D.G. Bobrow et T. Winograd , 1977, *Cognitive Science*, 1, (1), 3-46.

[RG77] *The FRL Manual*, R.B. Roberts et I.P. Goldstein, 1977, AI Lab, MIT, Cambridge, (MA) USA.

[WFA84] *SRL 2 User's Manual*, J.M. Wright, M.S. Fox, et D.L. Adam, 1984, Robotics Institute, Carnegie Mellon University, Pittsburgh, (PA) USA.

[RFU90] *SHIRKA : Système de gestion de bases de connaissances centrées-objets*, F. Rechenmann, P. Fontanille, et P. Uvietta, 1990, INRIA et Laboratoire Artémis IMAG, Grenoble, France.

[Duc88] *YAFOOL version 3.22 manuel de référence*, R. Ducournau, 1988, SEMA.MATRA, Montrouge, France.

[Euz93] *On a purely taxonomic and descriptive meaning for classes*, J. Euzenat, 1993, *13th IJCAI Workshop on Object-Based Representation Systems*, Chambery, France, 81-92.

[Euz95] *Troeps Reference Manual*, J. Euzenat, 1995, Projet Sherpa, INRIA Rhône-Alpes, Montbonnot, France.

[Dek94] *FROME : représentation multiple et classification d'objets avec points de vue*, L. Dekker, 1994, Thèse de doctorat, Université des Sciences et Techniques de Lille Flandres Artois, France.

[Pag00] « Représentation de connaissances au moyen de classes et d'associations : le système AROM », M. Page, J. Gensel, C.Capponi, C. Bruley, P. Genoud, et D. Ziébelin, 91-106, *Actes Langages et Modèles à Objets*, LMO'00, 25-28 janvier 2000.

[Cap95] *Identification et exploitation des types dans un modèle de représentation de connaissances à objets*, Capponi Cécile, Octobre 1995, Thèse de doctorat, Université Joseph Fourier, France.