Typage souple pour objets contraints

Cécile Capponi* — **Jérôme Gensel****

* *Institut National Polytechnique de Grenoble, ENSIMAG, France.*
** *Université Pierre Mendès-France, Grenoble, France.*
INRIA Rhône-Alpes — Projet SHERPA
655 avenue de l'Europe, 38330 Montbonnot, France
{Cecile.Capponi,Jerome.Gensel}@inrialpes.fr

RÉSUMÉ. *Nous traitons ici du couplage d'un système de représentation de connaissances par objets (*RPO*) avec, d'un côté,* METÉO*, un module de définition et de gestion de types chargé de la vérification et de l'inférence des types de toute entité de représentation et, de l'autre,* MICRO*, un module de programmation par contraintes chargé de la gestion des contraintes numériques et symboliques posées sur les objets. Après avoir rappelé quels sont les apports respectifs de chacun de ces modules pour la* RPO*, nous montrons comment une étroite collaboration entre eux permet de maintenir les types des entités contraintes. Nous illustrons la puissance de cette coopération à travers la notion de filtre, en détaillant la mise en oeuvre du filtrage associé. Après une revue des travaux comparables, nous concluons en posant les bases d'un système unifié intégrant à la fois les fonctionnalités* de METÉO *et celles de* MICRO*.*

ABSTRACT. *Here is addressed the coupling of an object-based knowledge representation system (*OKR*) with, on the one hand,* METEO*, a type system aimed at performing type checking and type inference over knowledge entity types, and on the other hand,* MICRO*, a constraint system aimed at managing numerical and symbolic constraints over knowledge entities. The way both systems contribute to* OKR *is studied. Then a close collaboration between them is exposed, which permits the maintenance of the types of constrained knowledge entities. The powerfulness of this collaboration is pictured through the design of a filtering algorithm. Finally, after a study of related works, the conclusion sets up the basis of an unified system that could fully integrate functionalities of both* METEO *and* MICRO*.*

MOTS-CLÉS : *Représentation par objets, contraintes, types*
KEYWORDS: *Object-based knowledge representation, types, constraints*

1. Introduction

Les systèmes de représentation de connaissances sont reconnus pour leur capacité à représenter à la fois les entités d'un domaine d'application et les mécanismes destinés à les exploiter. Ces systèmes se fondent sur des modèles déclaratifs d'expression de la connaissance. La représentation par objets (RPO) issue des *frames* [FK85] est un exemple de systèmes de représentation, où l'*objet* constitue l'unité de description de la connaissance.

Dans ce contexte, nous proposons la définition d'un système de types pour la vérification des descriptions de connaissances (§ 2), et la définition d'un système de contraintes augmentant les possibilités de la RPO pour l'expression et l'exploitation des connaissances (§ 3). Ces deux systèmes sont faiblement couplés à la RPO. Le système de contraintes intègre un mécanisme de maintenance locale basé sur une propagation de la réduction des domaines des attributs contraints. Nous montrons comment ces deux systèmes collaborent pour la vérification des types des entités contraintes, mais également au niveau des processus généraux de la RPO, pour l'exploitation de la connaissance (§ 4). Le filtrage est un de ces mécanismes (§ 5). Ce travail est comparé à d'autres dans un cadre plus général que la RPO (§ 6). En conclusion, nous proposons plusieurs améliorations de la collaboration de ces deux systèmes, par factorisation des traitements similaires qu'ils mettent en œuvre séparément.

2. Typage d'une représentation par objets

2.1. *Élements de base d'une* RPO

Dans une RPO, les connaissances sont organisées en classes et instances. Une instance est la représentation d'un individu unitaire du domaine d'application, par exemple *Luxembourg* si l'on s'intéresse à la géographie politique et économique. Une classe est la représentation d'un groupe d'instances, par exemple *État-Européen* a pour but de regrouper tous les États européens. Le Luxembourg fait partie de l'ensemble des États européens. Ceci se traduira, dans la RPO, par l'appartenance de l'instance *Luxembourg* à la classe *État-Européen*. Toute l'extension d'une classe n'est pas explicitement représentée par l'énumération de ses instances, ne serait-ce que parce que cette extension peut être infinie ; en contre-partie, la description d'une classe — son intension — est une tentative de caractérisation de l'extension. Les classes constituent un ordre partiel appelé la spécialisation, sur lequel est défini un mécanisme classique d'héritage dynamique. La spécialisation est établie sur le principe suivant : une classe C_1 est plus spécialisée qu'une classe C_2, noté $C_1 \leq C_2$, si l'extension de C_1 est inclue dans celle de C_2.

Une description de classe est un ensemble d'attributs à chacun desquels est associé un domaine de valeurs ; la description de la classe C est notée $d(C) = \{a_j : d_j\}_{j=1..n}$. Le domaine d_j d'un attribut a_j dans $d(C)$ est un ensemble de valeurs construit à partir de descripteurs. Ce domaine est en fait une expression déclarative (*i.e.* interprétable sans exécution) d'une contrainte unaire que doivent satisfaire les valeurs de cet attribut. Les descripteurs qui construisent le domaine d_j d'un attribut a_j de $d(C)$ sont distribués dans les descriptions des super-classes de C et dans $d(C)$. Par exemple, l'attribut *km-*

côtes de la classe *État-Intérieur*, a pour description, après héritage de la description de ce même attribut donné dans sa super-classe *État*, *km-côtes* **un entier intervalle** [0..500000] **valeur** 0 ; sa valeur équivaut donc à 0. Notons que les domaines attachés aux descripteurs (par exemple les bornes d'un intervalle) sont des constantes. Plusieurs descripteurs existent dans la classe de RPO que nous considérons, qui indiquent la nature du type principal (**un**, **liste**, **ensemble**), et qui introduisent de restriction de domaines (**intervalle**, **domaine**, **sauf**, **parmi**, **interdit**, **valeur**, **cardinalité**). Leur sémantique est disponible dans [Cap97]. Plusieurs descripteurs peuvent être attachés à un même attribut. Un attribut, par ses descripteurs, peut faire référence à un type de données ou à une classe : ses valeurs seront des instances de cette classe. Toutefois, la récursivité dans les descriptions de classes n'est pas autorisée ici.

La description d'une instance i consiste en la valuation des attributs de sa (ses) classe(s) d'appartenance ; elle est notée $d(i) = \{a_k = v_k\}_{k=1..m}$.

La relation d'appartenance d'une instance à une classe, notée $i \in C$, et la relation de spécialisation entre classes, notée $C \leq C'$, donnent toutes deux lieu à des conditions nécessaires — mais non suffisantes — sur les descriptions des entités en relation. Ces conditions se traduisent par des opérateurs ensemblistes entre valeurs et domaines d'attributs. Formellement, si $d(i) = \{a_k = v_k\}_{k=1..m}$, $d(C) = \{a_j : d_j\}_{j=1..n}$ et $d(C') = \{a_j : d'_j\}_{j=1..n'}$; alors l'appartenance s'exprime par $i \in C \Rightarrow n \leq m$ et $\forall j \in 1..n, v_j$ est inconnue ou $v_j \in d_j$; par exemple, pour qu'une instance d'*État* puisse appartenir à sa sous-classe *État-Intérieur*, il faut que la valeur de *km-côtes* dans l'instance soit égale à 0 ou inconnue. La spécialisation s'exprime comme $C \leq C' \Rightarrow n' \leq n$ et $\forall j \in 1..n', d_j \subseteq d'_j$. La condition de la spécialisation sur les descriptions des classes assure que dans tous les mondes possibles, si une instance appartient à une classe C, alors elle appartient à toutes les super-classes de C.

Les relations d'appartenance et de spécialisation sont garantes de l'organisation hiérarchique d'une base de connaissances à objets. On les retrouve unifiées dans les logiques de description sous l'appellation de subsomption [ND92]. Sur ces deux relations s'appuie la sémantique de plusieurs mécanismes d'inférence, comme la classification [Att91] ou la catégorisation.

Si nous considérons que le domaine spécifié pour un attribut a dans une classe C est le type de cet attribut, noté $t(a, C)$, alors la description d'une classe, en tant qu'ensemble d'associations attribut/domaine, n'est rien d'autre qu'un type record [1]. Les conditions de la spécialisation posées sur les descriptions de classes correspondent alors au sous-typage entre types records [CM91], qui se projette sur le sous-typage entre les types des attributs. Similairement, la description d'une instance est une valeur record, et $v(a, i)$ désigne la valeur de l'attribut a pour l'instance i. Les conditions posées par l'appartenance sur les descriptions de classes et d'instances correspondent alors à l'appartenance d'une valeur record à un type record ; celle-ci se projette sur l'appartenance

1. Nous faisons l'hypothèse qu'il n'y a pas de méthodes de classes ; en conséquence, nous ne nous intéressons pas à l'interprétation fonctionnelle d'un record.

d'une valeur à un type d'attribut. L'appartenance de l'instance i à la classe C est ainsi valide si, pour tous les attributs a de C, $v(a,i) \in t(a,C)$.

2.2. MÉTÉO *: un système de types pour la* RPO

Nous donnons ici l'architecture principale du système de types MÉTÉO qui a été conçu pour prendre en main le typage d'une RPO telle que présentée précédemment. MÉTÉO est constitué de deux niveaux connectés de types [Cap95]. Les δ-types (premier niveau) sont des expressions syntaxiques normalisées des-

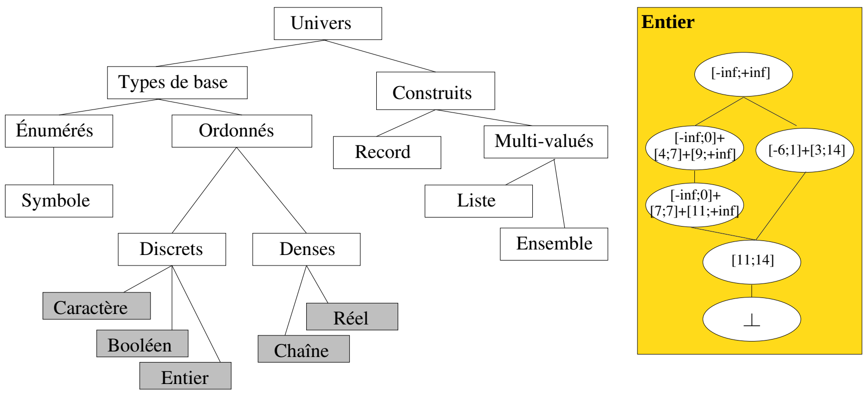

Figure 1. À gauche, *arbre $\mathcal{A}$ de C-types de* MÉTÉO, *indépendant de la* RPO. *Les C-types sur fond grisé sont ceux qui pourraient être dynamiquement ajoutés par un utilisateur, et qui n'existent pas dans la configuration minimale de* MÉTÉO. **À droite**, *δ-types issus du C-type* `Entier`, *résultats du typage de la* RPO, *et ordonnés par le δ-sous-typage.*

tinées à accueillir les types des entités de représentation : types d'attributs, de classes et d'instances. Ils expriment des sous-ensembles de valeurs d'un type de données particulier. Par exemple, t(*km-côtes*,*État*) est un δ-type issu du type de données `Entier`, exprimé par $\langle$`Int`$; [0..500000]\rangle$. Un C-type (second niveau) correspond à un type de données étendu. D'une part, il encapsule l'implémentation du type de données disponible dans un langage externe, comme `Entier` ou le constructeur `Liste`. D'autre part, un C-type contient la syntaxe des δ-types construits sur le type de données considéré, et l'implémentation des opérations ensemblistes usuelles manipulant ces δ-types, telles que l'appartenance $\in_\tau$, l'inclusion $\leq_\tau$ ou l'intersection $\sqcap_\tau$. Les C-types sont organisés en un arbre d'héritage $\mathcal{A}$, ordonné par l'inclusion ensembliste (figure 1).

Par exemple, le C-type `Entier`, en qualité de type `Discret`, définit ses δ-types comme des listes d'intervalles à bornes fermées, ceci étant possible du fait de l'existence des opérateurs `succ`, `pred` et de l'ordre total sur les entiers, exploités génériquement par `Discret` à travers des fonctions de second ordre.

Le typage d'un attribut de classe revient à interpréter les descripteurs de l'attribut vers la syntaxe de représentation des δ-types du type principal (C-type) de l'attribut. En reprenant l'exemple de l'attribut *km-côtes*, sa description est typée par le δ-type $\langle \texttt{Int}; [0..0] \rangle$. Le typage d'une classe consiste à typer ses attributs par des δ-types, puis à créer le δ-type record constitué des associations attribut/δ-type. Le δ-type d'une instance est un δ-type record qui résulte de l'intersection des δ-types records des classes auxquelles l'instance appartient, sauf lorsqu'une valeur d'attribut est connue pour l'instance, par exemple 0 : elle remplace sous forme d'un δ-type "singleton" le δ-type de cet attribut dans le δ-type de l'instance, soit dans l'exemple le δ-type $\langle \texttt{Int}; [0..0] \rangle$.

Les δ-types d'un même C-type sont partiellement ordonnés par le δ-sous-typage qui traduit l'inclusion ensembliste. Par composition de l'arbre d'héritage $\mathcal{A}$ et du δ-sous-typage distribué dans les C-types (car il dépend de la syntaxe des δ-types), METÉO définit le γ-sous-typage qui permet la comparaison ensembliste de δ-types issus de C-types différents, comme `Entier` et `Pair` [Cap97].

METÉO est extensible, indépendamment de la RPO à laquelle il est couplé. Cela signifie que le concepteur d'une base de connaissances peut ajouter un nouveau C-type à METÉO, par exemple `Pair`. Pour cela, il choisit sa position dans $\mathcal{A}$, par exemple sous-type de `Entier`, et fournit des opérateurs de base en les programmant ou en créant des liens dynamiques avec une implémentation de ce nouveau type disponible dans une bibliothèque. Ces opérateurs sont, quelle que soit la position du nouveau C-type, l'appartenance d'une valeur quelconque au C-type et l'égalité entre deux valeurs du C-type. Ensuite, d'autres opérations peuvent être requises selon la position dans $\mathcal{A}$, comme l'ordre total entre deux valeurs exigé par `Ordonnés`, le successeur et/ou le prédécesseur exigées pour tout sous-type de `Discret`, etc. Une fois précisées ces opérations, le nouveau C-type est complètement intégré à METÉO. En particulier, et c'est là que se situe l'originalité principale de METÉO, la gestion ensembliste des δ-types du nouveau C-type est automatiquement générée [Cap97].

2.3. *Apports du couplage* RPO/METÉO

Dans le couplage, les δ-types de METÉO résultent du typage des entités de la RPO. Ce typage constitue l'interface entre la RPO et METÉO, et consiste essentiellement en l'interprétation des descripteurs d'attributs de la RPO vers la syntaxe de représentation des δ-types. La normalisation, réalisée à travers un système de réécriture des δ-types, est indépendante de la RPO. La gestion ensembliste des δ-types est elle aussi interne à METÉO. On distingue alors deux avantages du couplage METÉO/RPO parmi ceux analysés dans [Cap95]. Premièrement, METÉO est utile à des fins de vérification et d'inférence de types. Les relations de spécialisation et d'appartenance sont testées directement sur les δ-types, évitant la redondance des calculs grâce à la normalisation, via respectivement le γ-sous-typage évoqué précédemment et l'appartenance d'une valeur à un δ-type. Les résultats de vérifications ou d'inférence obtenus sur les

δ-types sont valides au regard des descriptions des connaissances [Cap95]. Ceci permet une délégation complète et correcte dans METÉO de tout processus de la RPO qui opère sur les descriptions des connaissances. Cette propriété est conservée même en présence de nouveaux types de données externes [Cap97]. Deuxièmement, modifier le δ-type qui correspond à une entité de représentation ne modifie pas la description de cette entité telle qu'écrite par le concepteur. En ce sens, les δ-types sont dits *souples*, d'autant plus que toute modification est facilitée par la forme normale des δ-types.

Cette seconde propriété du fonctionnement de METÉO vis-à-vis de la RPO est une propriété clé qui a permis un couplage évolué entre METÉO et un système de contraintes MICRO dont une des activités est la réduction des domaines des attributs contraints. Le couplage consiste à profiter de la souplesse des δ-types pour accueillir les résultats — même temporaires — de la propagation de contraintes mettant en jeu des domaines d'attributs. Le δ-type d'un attribut est alors la compilation de son domaine calculé par héritage puis réduit par la propagation de contraintes.

3. Objets contraints

3.1. *Objets et contraintes : une association polymorphe*

Associer objets et contraintes relève d'un souci de disposer d'un moyen déclaratif d'énoncer des contraintes symboliques ou numériques sur des objets. Cet objectif ne va pas sans 1) le maintien de la cohérence des objets ainsi contraints et peut être prolongé jusqu'à 2) la résolution des réseaux de contraintes définis de façon à valider, compléter ou produire les seuls ensembles d'objets viables sous cette chappe de contraintes.

Historiquement, ce sont vers les objets de programmation que se sont portés les premiers efforts à travers des langages, comme THINGLAB [Bor81], destinés à la gestion d'interfaces dans lesquels évoluent des objets contraints. Ce genre de langages de programmation orientée-objet avec contraintes sont plus particulièrement axés sur le maintien et non pas sur l'énumération de solutions.

Une autre approche consiste à fournir des bibliothèques, telles que SOLVER [ILO95], ou des langages à part entière, tel que LAURE [Cas94], de programmation par contraintes (PPC) dans lesquels les contraintes peuvent être immergées dans un monde de classes et d'objets afin de bénéficier des avantages de la programmation orientée-objet, notamment de l'héritage.

Dans le domaine de la RPO, l'approche la plus répandue consiste à représenter les divers composants d'un problème de satisfaction de contraintes (CSP) par des objets (PROSE [Ber92], CSPOO [Kök94]) à l'aide du même langage de représentation, et de faire cohabiter ainsi les objets contraints et les objets décrivant ces contraintes.

Nous avons préféré d'abord mener une étude des besoins et des conséquences de l'introduction de contraintes au sein d'une RPO [Gen95] avant de juger de la forme qu'elle devait prendre. Nous résumons ici les principales réflexions qui ont précédé la réalisation d'un module de PPC dédié aux objets d'une RPO.

3.2. *Pourquoi et comment contraindre une* RPO

L'introduction de contraintes dans une RPO doit permettre d'exprimer, de maintenir, voire de résoudre des relations entre plusieurs objets. Ainsi, là où les traditionnels prédicats ne fonctionnent que comme de simples tests, l'emploi de techniques CSP garantit d'une part la cohérence locale des contraintes exprimées et, d'autre part, l'inférence de valeurs dès que possible.

Les CSP sont définis par trois composants : un ensemble de variables, l'ensemble des domaines de ces variables et un ensemble de contraintes impliquant les variables. Il nous faut donc déterminer la nature de ces trois composants dans le contexte d'une RPO afin de définir ce que nous appelons les CSP RPO.

En fait, les seules entités de représentation contraintes sont les attributs, car même si l'on peut imaginer des contraintes opérant sur des entités de représentation de niveau supérieur (instances, classes), il est toujours possible, en considérant le méta-niveau d'un modèle réflexif offrant des attributs concernant la description purement informatique de l'objet (**`identificateur, self, nombre d'instances`** ...), de ne contraindre ultimement que des attributs. Les attributs sont donc les variables désignées des CSP RPO.

Les domaines des attributs d'une RPO étant *a priori* quelconques et possiblement infinis, il en va de même pour les domaines des CSP RPO. À ce titre, l'approche de la PPC sur intervalles semble bien adaptée car elle propose un sur-ensemble des capacités techniques de la PPC numérique classique, capable de traiter des domaines finis et infinis.

Quant aux contraintes, leur éventail doit prend en compte la multiplicité des domaines d'attributs et on peut s'attendre à devoir gérer des CSP RPO qui soient numériques à intervalles, booléens, ensemblistes, mais aussi symboliques. En réalité, il est possible de composer à volonté et de compléter cet ensemble de contraintes envisageables par des contraintes disjonctives (ou non déterministes), des contraintes conditionnelles, etc.

En considérant les États européens, on peut créer la classe *État-Européen-ok-fin-pub* de ceux satisfaisant pour leurs finances publiques le critère de convergence de la monnaie unique du traité de Maastricht par la contrainte : (*déficit-public / P.I.B*) < 0.03. Cette contrainte n'est pas fonctionnelle (ne permet pas d'inférence, sauf cas particulier) alors que l'est une autre contrainte : *densité = nb-babitants / superficie*, où *déficit-public*, *P.I.B*, *densité*, *nb-babitants*, *superficie* sont des attributs.

Une fois les CSP RPO définis, la question se pose de la portée effective de ces contraintes. Les deux niveaux de représentation de l'approche classe/instance suggèrent que soient distinguées les contraintes de classe qui portent sur toutes les instances d'une classe et les contraintes d'instances qui portent ponctuellement sur une ou plusieurs instances.

Si nous considérons que les variables des CSP RPO sont les attributs des objets, il est souvent commode de rassembler dans l'expression simple d'une contrainte sur des instances, l'ensemble des contraintes sur attributs auquel elle est équivalente (par exemple, une contrainte d'égalité entre deux instances de la classe `Point2D` se traduit en deux contraintes d'égalité sur les attributs représentant les abscisses et les ordonnées de ces points). La possibilité de définir des contraintes sur des instances à partir de contraintes sur des attributs doit donc être offerte pour des besoins de déclarativité et de factorisation.

Les CSP RPO peuvent donc comporter des contraintes de classe, des contraintes d'instance mais également des contraintes entre instances. Les contraintes entre classes impliquent des ensembles d'instances de classes distinctes. Elles sont soit des contraintes de classe sur un attribut typé par une classe, soit des contraintes d'une association ou relation reliant ces classes.

Concernant la représentation des contraintes en tant qu'objets de la RPO, si on peut supposer que cela permet *a priori* de manipuler de manière uniforme objets et contraintes à l'aide des primitives du modèle, la sémantique associée aux contraintes, selon leur portée, conduit à de sérieuses restrictions (on ne peut, par exemple, modifier ou supprimer une contrainte de classe sur une instance particulière) et finalement à représenter et maintenir les CSP hors de la RPO à travers un couplage avec un module de programmation par contraintes. Celui que nous avons conçu pour cette tâche se nomme MICRO.

3.3. MICRO *: module de maintenance de contraintes pour la* RPO

MICRO est plus particulièrement destiné à la maintenance de CSP numériques à intervalles et de CSP ensemblistes. Il propose un ensemble de contraintes de bases (des opérateurs de comparaison et des opérateurs arithmétiques, mais aussi des contraintes disjonctives, des conditionnelles et des méta-contraintes) qui peuvent être assemblés pour former des contraintes complexes. La définition d'un CSP se fait à partir de la description des variables contraintes (identificateur, type et domaine) et des contraintes qui les impliquent.

Dans la syntaxe de MICRO, les contraintes données ci-dessus s'expriment : (`mic-lt` (`mic-div` *déficit-public P.I.B*) 0.03) et (`mic-eq` *densité* (`mic-div` *nb-babitants superficie*)).

Fondé sur les principes de l'arithmétique des intervalles [Moo66] et de la théorie des ensembles, MICRO garantit la consistance locale des CSP grâce à une propagation de contraintes déclenchée dès qu'une perturbation survient. De plus, MICRO est capable de gérer des CSP dynamiques dans lesquels est possible à tout moment l'ajout ou le retrait d'une contrainte, ou la modification du domaine d'une variable. La résolution des CSP est également assurée par MICRO bien que, dans le domaine de la RPO, cette tâche soit placée au second plan pour des raisons évidentes de coût et parce qu'elle n'est justifiée que dans un but prospectif de l'évolution d'une base de connaissances. La définition, la modification et la résolution des CSP sont commandées par une interface fonctionnelle qui fait de MICRO un module de PPC indépendant de toute RPO.

Le couplage de MICRO avec une RPO se fera en associant à chaque attribut contraint de la RPO une variable contrainte de MICRO, avec ou sans duplication.

3.4. *Conséquences et apports du couplage* RPO/MICRO

Nous rappelons ici les principaux apports de la gestion de CSP RPO à travers le couplage RPO/MICRO, étudiés dans [Gen95].

L'intégration de contraintes dans une RPO doit préserver la sémantique propre à chaque moyen de représentation ou d'inférence : instanciation, classification, héritage.

Les aspects dynamiques des CSP RPO conduisent à énoncer un comportement des objets et des chemins contraints vis-à-vis de l'ajout ou du retrait de contrainte ; par exemple, que faire d'une instance qui ne vérifie pas la nouvelle contrainte de sa classe ? D'autre part, la cohérence d'une base de connaissances contraintes est maintenant dévolue au niveau de consistance atteint sur les CSP RPO et, de ce fait, aux capacités du module de PPC associé. Ainsi, redondances et incohérences introduites par les contraintes ne sont pas toujours détectables sans la résolution dont le coût peut être par ailleurs prohibitif.

Malgré ces limitations intrinsèques, nous avons montré [Gen95] que les contraintes peuvent être pour la RPO d'un apport conséquent en augmentant la déclarativité et les capacités d'inférence du modèle. Parmi les usages collaboratifs qui peuvent être faits des contraintes dans une RPO, citons : le partage de propriétés au sein d'objets composites, le flot de données dans un modèle de tâches, l'exploration d'une base de connaissances par raisonnement hypothétique, l'expression et la gestion de la sémantique de relations entre objets, ou encore l'expression et la gestion de filtres.

Introduire des contraintes dans une RPO, c'est permettre l'expression et la maintenance de relations entre objets mais c'est aussi bénéficier de ces contraintes pour accroître l'expressivité et contrôler la sémantique d'autres fonctionnalités du modèle.

4. Objets typés contraints

Assurer un certain niveau de consistance locale pour les domaines contraints consiste à éliminer des domaines des attributs contraints d'une RPO les valeurs qui ne pourront pas apparaître dans une solution d'un CSP RPO. Dans la mesure où l'ensemble de contraintes portant sur un attribut peut varier, le domaine — donc le type — de l'attribut contraint est susceptible de varier aussi. Ces variations sont, soit des réductions (lors de l'ajout d'une contrainte), soit des relaxations (lors d'un retrait). Nous montrons ici l'intérêt de prendre en compte l'impact des contraintes au niveau des types des entités de représentation.

En l'absence de contraintes, la validation d'une valeur proposée pour un attribut consiste, pour MeTéo, à s'assurer qu'elle appartient bien au type initial de l'attribut représenté par un δ-type. En présence de contraintes sur cet attribut, non seulement la valeur proposée doit appartenir au type initial de l'attribut mais, pour chaque contrainte, on doit également s'assurer que l'instanciation formée de valeur de l'attribut et des valeurs des attributs attenants ne viole pas la contrainte.

Aussi, si MeTéo ne conserve que les types initiaux, il peut entériner une valeur inconsistante et ce n'est que lors de la phase de maintenance de la contrainte, réalisée par Micro, que cette valeur sera écartée. En conservant le type courant d'un attribut contraint — qui est une restriction du type initial dont le domaine associé est consistant — MeTéo est en mesure de refuser immédiatement cette valeur. C'est pourquoi tout domaine évalué par la phase

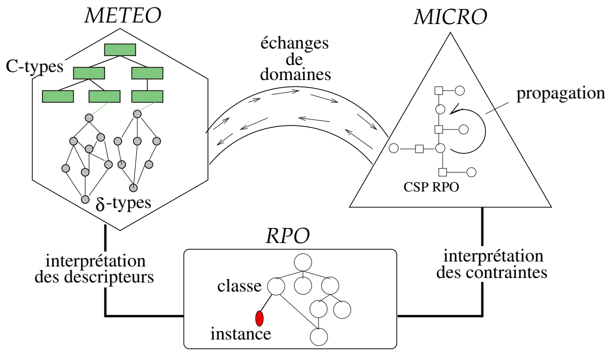

Figure 2. *Échange de domaines entre* MeTéo *et* Micro*.*

de maintenance déclenchée à l'ajout ou au retrait d'une instance sera conservé par MeTéo sous la forme d'un δ-type remplaçant le δ-type initial. Ainsi, le type courant d'une entité contrainte est un domaine consistant au regard de l'ensemble des contraintes portant sur cette entité et des capacités du moteur de propagation.

Afin que les types soient consistants avec les contraintes présentes, une collaboration étroite doit s'engager entre MeTéo et Micro. Sans cette collaboration, MeTéo conserverait des informations non cohérentes avec l'état des connaissances ; son rôle en présence de contraintes serait alors inconsistant. Micro sera chargé du calcul de la restriction des δ-types, MeTéo de leur conservation (figure 2). Le domaine restreint d'un type d'attribut a obtenu par Micro est passé à MeTéo qui l'exprime alors sous la forme d'un nouveau δ-type, factorisant en une seule expression les descripteurs de a et l'éventuelle réduction du domaine de a opérée par la phase de propagation du maintien de la consistance locale des contraintes portant sur a.

La coopération entre Micro et MeTéo telle que réalisée dans ce travail illustre le fait que la partie statique des contraintes, même si elle est implicite, peut être extraite, et rendue explicite sous forme de δ-type, par la mise en évidence et l'exploitation des propriétés algébriques des types de données qui sont à la base des règles de maintenance locale. La section suivante illustre cette propriété, par la mise en œuvre d'un algorithme de pré-filtrage, qui correspond à l'extraction de l'information disponible statiquement.

5. Application: mise en œuvre du filtrage

5.1. *Définitions*

Un filtre est posé sur une entité, appelée *base du filtre*, qui est une classe ou un ensemble de classes. Un filtre est une description qui restreint celle de sa base. Cette restriction est exprimée à l'aide des descripteurs classiques d'attributs et, dans une RPO contrainte, à l'aide des contraintes.

Un filtre est le support du mécanisme de filtrage [Dek94, Fau91]. Le filtrage consiste à parcourir l'extension de la base du filtre pour retrouver la ou les instances qui vérifient les conditions du filtre.

Un filtre peut être ou non attaché à un attribut ; dès lors, nous considérons trois utilisations possibles d'un filtre dans une RPO. Tout d'abord, en qualité de moyen d'affinement du type d'un attribut, il représente une classe virtuelle dont le type, intersecté avec celui de la base du filtre, constitue le type de l'attribut [Cap95]. Ensuite, en qualité de moyen d'obtention d'une valeur d'attribut, il apparaît dans un descripteur de l'attribut (`$sib-filtre` en SHIRKA [RFU90]). Enfin, en qualité de requête portant sur l'extension de la base du filtre, le filtre est rémanent ou volatile ; il pourra être classé comme une classe pour la résolution de la requête [Dek94].

Un filtre qui consiste à identifier tous les grands États maritimes (pour lesquels les kilomètres de côtes, supérieurs à 3000 km, sont aussi supérieurs aux kilomètres de frontière terrestre) s'exprime par : `Filtre base` *État* `conditions` *km-côtes* `intervalle [3000;+`∞`[ contraintes` *km-côtes* $\geq$ *km-frontières*.

5.2. *Mise en œuvre*

Le mécanisme du filtrage est intuitif : chaque objet de la base du filtre doit être passé au tamis des conditions constituant la description du filtre. L'extension de la base définit ainsi l'espace de recherche du filtrage. METÉO type les attributs, classes et instances, mais aussi les filtres en tenant compte des résultats des propagations de contraintes effectuées par MICRO. Nous avons conçu un algorithme de pré-filtrage dont le but est de réduire l'espace de recherche, en exemptant de la vérification des conditions du filtre, les instances appartenant à la base dont on peut dire avec certitude qu'elles ne satisfont pas le filtre.

Une fois le type du filtre $t(F)$ calculé par intersection du type $t(B)$ de la base et du δ-type record des conditions (on a donc $t(F) \leq_\tau t(B)$), une zone de pré-filtrage est définie : elle contient tous les sous-types de $t(B)$. Les sous-types de $t(B)$ sont jugés compatibles. Les autres types dans la hiérarchie ne sont pas pertinents pour le filtre.

La zone de pré-filtrage est alors parcourue, du haut vers le bas, depuis $t(B)$. À chaque type rencontré t, une intersection avec $t(F)$ est calculée — par l'opérateur d'intersection ensembliste définie pour les δ-types records —. Si elle s'avère vide, alors t est incompatible. Par conséquent, tous les sous-types de t sont également incompatibles. On propage donc jusqu'aux sous-types feuilles une incompatibilité décelée à un niveau donné. Si l'intersection entre t et $t(F)$ n'est pas vide, alors t est compatible et la recherche continue vers le bas.

Une fois $\{t_i\}_{i=1..n}$ l'ensemble des types compatibles obtenu, on ne garde que ceux qui correspondent à des types d'instances de la base du filtre. Le résultat du pré-filtrage est ainsi un sous-ensemble $\mathcal{I}$ des instances de la base, dont l'intersection du type avec celui du filtre n'est pas nulle (figure 3).

À partir de là, le filtrage n'est nécessaire que si des contraintes sont présentes dans les conditions du filtre. En effet, chaque élément de $\mathcal{I}$ vérifie les conditions statiques du filtre, modulo les valeurs inconnues d'attributs dont le domaine est assurément compatible avec ceux imposés par le filtre. Si le filtre comporte des contraintes, une vérification dynamique du contenu de chaque élément de $\mathcal{I}$ est réalisée par MICRO.

Le pré-filtrage s'avère payant dans la mesure où le rapport entre le nombre d'instances de $\mathcal{I}$ et le nombre d'instances de la base est petit. Déterminer ce nombre n'est pas possible *a priori*, mais il est sans doute estimable selon les affinements réalisés par le filtre sur les attributs par rapport à la définition de ces attributs pour la base, mais également, selon la nature des contraintes exprimées dans le filtre.

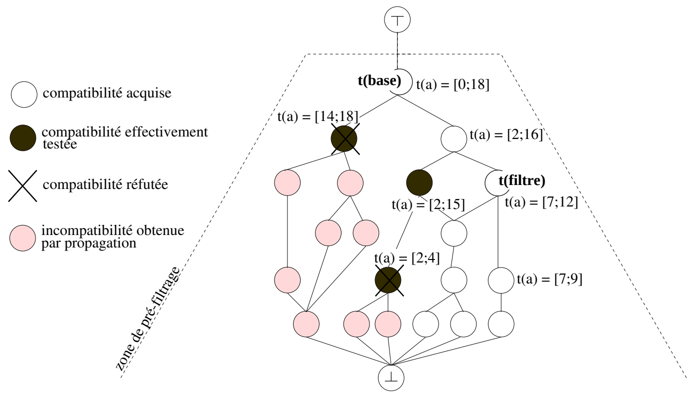

Figure 3. *Réduction dans* METÉO *(graphe des δ-types records) de l'espace de recherche du filtrage. Pour une question de lisibilité, un seul attribut (a) est considéré.*

6. Travaux comparables

Il n'existe pas, à notre connaissance, de travaux concernant le couplage d'un système de gestion de types avec une RPO. Les objectifs d'une RPO sont, par bien des aspects, très différents de ceux des langages de programmation par objets (LPO) : migration d'instances dans les RPO vs. instances dépendante de leur classe de création dans les LPO, pas d'envoi de messages en RPO pour respecter les principes liés à la déclarativité, forte dynamicité des objets et de leurs descriptions en RPO, etc. Les mises en œuvre de ces deux formes de modèles à objets sont donc différentes, en particulier les techniques de compilation et/ou d'interprétation, et *a fortiori* les caractéristiques des systèmes de types.

Les principes de METÉO dérivent des exigences de la RPO, et l'on ne retrouve donc pas nécessairement certaines problématiques propres aux LPO, telles que le sous-typage de méthodes ou la vérification dynamique lors de la sélection. METÉO n'est pas un système de types pour la programmation par objets, il correspond plutôt à une couche supplémentaire, pour une gestion extensible des domaines dynamiques, au dessus du système de types du langage cible de l'interpréteur de la RPO à laquelle il est couplé. L'originalité de METÉO, si l'on cherche toutefois à le comparer avec d'autres systèmes de types rencontrés dans les LPO, réside dans sa capacité à représenter et à manipuler des sous-ensembles arbitraires et dynamiquement modifiables de types de données. Le seul système de types, en programmation, comparable en ce sens avec METÉO, est celui de ADA qui permet la représentation de sous-ensembles. Pourtant, les possibilités offertes par ADA sont très limitées. ADA est de plus instable au regard du sous-typage puisqu'aucune inférence de sous-typage n'y est possible. C'est grâce à cette gestion de sous-ensembles de types de données quelconques que METÉO s'adapte parfaitement à la RPO puisqu'elle nécessite la gestion des domaines d'attributs. METÉO est dans la lignée des reflexions amorcées par A. Borgida [Bor92]. L'extensibilité de METÉO est elle aussi originale, même si des travaux similaires ont débuté en ce sens autour du système CLASSIC [BIM96]. Cette propriété d'extensibilité rejoint d'ailleurs les objectifs de l'extensibilité de la collection de descripteurs dans les logiques de descriptions [BB92].

Concernant les couplages objets et contraintes, PROSE comme SOLVER et LAURE constituent des langages de programmation par contraintes avec objets et non pas des RPO avec contraintes auxquelles est destiné MICRO. Toutefois, une comparaison peut toujours s'opérer au niveau des CSP traités par les différents solveurs intégrés de ces outils. À ce titre, les techniques CSP employées dans MICRO sont proches des deux derniers langages cités. CSPOO s'appuie sur une RPO mais pour modéliser des solveurs. SOCLE [Har86] est dans l'esprit le plus proche d'un couplage RPO/MICRO mais se montre nettement plus rudimentaire (pas de possibilité d'associer des domaines aux attributs, propagation de valeurs uniquement).

Enfin, l'existence d'un système prônant la collaboration d'un système de types et d'un système de contraintes n'a jamais été proposée jusqu'ici, dans le cadre de la RPO et dans le cadre plus large des logiques de description. Des travaux moins génériques existent cependant, comme [BH91] ou [Gai93]. Ils ne permettent pas l'indépendance de fonctionnement entre les deux systèmes collaborant, puisque les capacités propres à un système de types n'y ont pas été réellement identifiées et sont parties intégrantes du système englobant et/ou du système de contraintes. [BH91] concerne l'intégration de prédicats sur des domaines concrets (ex. entier, réels, donc comparables aux C-types) dans une logique de description. En ce qui concerne la possibilité de choix entre divers types pour un attribut (disjonction), nous envisageons l'extension de l'arbre $\mathcal{A}$ afin d'intégrer les *variants* [CM91] en tant que C-type, ce qui permettra en aval de traiter la négation disponible dans [BH91]. Quoi qu'il en soit, le couplage MICRO/METÉO repose sur un moteur de contraintes offrant maintien de consistance et résolution là où les prédicats n'offrent que des tests.

7. Conclusion et perspectives

Cet article est une proposition pour étendre toute RPO vers la gestion de types et de contraintes. Elle s'appuie sur la présentation des apports de deux modules. D'un côté, il s'agit d'adjoindre à la RPO, METÉO, un module de gestion de types qui se charge de la vérification des types des entités représentées dans une base de connaissances, même en présence de nouveaux types de données dont METÉO facilite l'intégration dans la RPO. De l'autre, nous proposons de coupler la RPO avec un module de PPC, MICRO, qui permet à la fois la définition et la maintenance de contraintes numériques ou symboliques entre les objets, mais également d'accroître les capacités de la RPO en se chargeant de la description et de la cohérence de mécanismes spécifiques.

De plus, une communication entre ces deux modules permet à METÉO de maintenir les types des entités contraintes à partir des domaines dont la consistance locale est assurée par MICRO.

La gestion des filtres illustre bien l'apport de cette double extension. Ainsi, la description de filtres peut maintenant être complétée par des contraintes gérés par MICRO alors que le filtrage est amélioré par METÉO en étant d'abord opéré sur les types des objets.

À présent ce travail doit être poursuivi vers la conception d'un système unifié intégrant à la fois les fonctionnalités de METÉO et celles de MICRO. L'idée sous-jacente est de permettre à ce système de gérer des CSP dont les types seraient définis par les soins de l'utilisateur.

En examinant les fonctions respectives de METÉO et de MICRO, plusieurs remarques s'imposent. Tout d'abord, il y a duplication de l'information : les domaines contraints gérés par MICRO sont également stockés par METÉO. Ensuite, la normalisation que MICRO intègre sur les domaines des variables numériques ou multivaluées, est identique à celle opérée par METÉO pour uniformiser l'expression des types. Il y a donc possibilité de rassembler ce code commun. De même, si l'on observe les règles de maintenance associées aux opérateurs arithmétiques binaires $(+, -, *, / \ldots)$, on s'aperçoit qu'une seule contrainte, réprésentant un opérateur générique, suffirait pour factoriser l'effet de toutes les autres moyennant un paramétrage. En effet, en indiquant le type des arguments, l'opérateur associé et son inverse, il est possible d'associer à cette contrainte trois règles de maintenance (une pour chaque argument de l'opérateur binaire considéré) valables pour tous les opérateurs. Il y a donc possibilité de polymorphisme et donc de généricité dans la définition et la gestion des contraintes. Enfin, la représentation d'un type contient les prédicats d'appartenance, ou de comparaison ainsi que des opérateurs associés à ce type. La définition et la sémantique de ces opérateurs peuvent être fixées. Il y a donc possibilité de fabriquer de nouvelles contraintes de types nouveaux.

Le système intégrant les principes de MICRO et de METÉO nécessite l'enrichissement de la hiérarchie actuelle de C-types et l'introduction parallèle d'un méta-niveau de contraintes tirant parti des propriétés algébriques des opérateurs de C-types prédéfinis, ces derniers encapsulant alors les contraintes génériques. Le résultat attendu est la généralisation de la propriété d'extensibilité de METÉO à MICRO : l'ajout d'un nouveau type de données dans METÉO s'accompagnera de la génération automatique des contraintes possibles impliquant les valeurs du nouveau type, et du traitement associé à ces contraintes.

Remerciements

Nous remercions les relecteurs anonymes de cet article, ainsi que Michelle Hut, pour leurs commentaires détaillés et leurs suggestions avisées.

Références

[Att91] G. Attardi. An analysis of taxonomic reasoning. *Inheritance Hierarchies in Knowledge Representation and Programming Language*, éd. par D. Nardi M. Lenzerini et M. Simi, chapitre 3, pages 29–49. J. Wiley, 1991.

[BB92] A. Borgida et R.J. Brachman. ProtoDL : a customizable knowledge base management system. *1st CIKM*, pages 482–490, Baltimore (MA, US), 1992.

[Ber92] P. Berlandier. *Étude de mécanismes d'interprétation de contraintes et de leur intégration dans un système à base de connaissances.* Thèse de doctorat, Université de Nice Sophia Antipolis, (France), 1992.

[BH91] F. Baader et P. Hanschke. A scheme for integrating concrete domains into concept languages. *12th International Joint Conference on Artificial Intelligence*, Sydney (AU), Août 1991.

[BIM96] A. Borgida, C.L. Isbell, et D.L. McGuinness. Reasoning with black boxes: handling test concepts in CLASSIC. *International Description Logics Workshop*, Cambridge (MA, US), Nov 1996.

[Bor81] A. Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, Oct 1981.

[Bor92] A. Borgida. From type systems to knowledge representation: natural semantic specifications for description logics. *International Journal on Intelligent and Co-operative Information System*, 1(1), 1992.

[Cap95] C. Capponi. Identification et exploitation des types dans un modèle de connaissances à objets. Thèse de doctorat, Université Joseph Fourier, Grenoble (France), Oct 1995.

[Cap97] C. Capponi. Design and implementation of a type system for a knowledge representation system. RR 3096, INRIA Rhône-Alpes, France, Jan 1997.

[Cas94] Y. Caseau. Constraint Satisfaction with an Object-Oriented Knowledge Representation Language. *Journal of Applied Intelligence*, 4:157–184, 1994.

[CM91] L. Cardelli et J.C. Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1(1):3–48, Mars 1991.

[Dek94] L. Dekker. *FROME: représentation multiple et classification d'objets avec points de vue.* Thèse de doctorat, LIFL, Université des Sciences et Technologies de Lille, Lille (France), Juin 1994.

[Fau91] C. Faucher. *Élaboration d'un langage extensible fondé sur les schémas, le langage Objlog+.* Thèse de doctorat, Université d'Aix-Marseille III, Marseille (France), Juil 1991.

[FK85] R. Fikes et T. Kehler. The role of frame-based representation in reasoning. *Communications of the ACM*, 28(9):904–920, Sep 1985.

[Gai93] B.G. Gaines. A class library implementation of a principled open architecture knowledge representation server with plug-in data types. *13th International Joint Conference on Artificial Intelligence*, éd. par Ruzena Bajcsy, volume 1, pages 504–509, Chambéry (France), Sep 1993.

[Gen95] J. Gensel. *Contraintes et représentation de connaissances par objets. Application au modèle* TROPES. Thèse de doctorat, Université Joseph Fourier, Grenoble (France), Oct 1995.

[Har86] D. R. Harris. A Hybrid Structured Object and Constraint Representation System. *Proc. of AAAI'86*, pages 986–990, Philadelphia (PA, US), 1986.

[ILO95] ILOG, Gentilly, France. SOLVER *Version 3.0 : manuel de référence*, 1995.

[Kök94] T. Kökény. *Satisfaction de Contraintes dans un Environnement Orienté Objets.* Thèse de doctorat, Université de Montpellier II, Montpellier (France), Déc 1994.

[Moo66] R. Moore. *Interval Arithmetic.* Prentice-Hall, Englewoods Cliff (NJ, US), 1966.

[ND92] A. Napoli et R. Ducournau. Subsumption in object-based representations. *ERCIM Workshop on Theoretical and Experimental Aspects of Knowledge Representation*, pages 1–9, Pisa (Italy), 1992.

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