Expression et satisfaction de contraintes dans TROPES, un modèle de représentation de connaissances par objets

Jérôme Gensel

Projet SHERPA—IRIMAG-LIFIA
Bâtiment Hitella
46, avenue Félix Viallet
38031 Grenoble Cedex
Tel : (33) 76 57 45 90
Fax : (33) 76 57 46 95
E-mail Jerome.Gensel@imag.fr

Résumé

TROPES est un modèle de représentation de connaissances par objets multi-points de vue dédié à la gestion et à l'évolution de bases de connaissances. Il exploite divers mécanismes de raisonnement (méthodes, classification, héritage, …) afin de compléter les connaissances acquises. L'objectif de l'intégration de contraintes présentée ici est de doter TROPES d'un nouvel outil permettant la modélisation et la résolution d'une plus large étendue de problèmes. Nous décrivons les principales caractéristiques du modèle avant de détailler les différents types de contraintes qui peuvent être déclarées et posées sur des objets TROPES. Nous avons choisi de représenter ces contraintes par des classes TROPES afin que leur manipulation (définition, activation, modification, suppression) soit entièrement contrôlée par des mécanismes du modèle. La satisfaction des contraintes est déléguée à PECOS, une librairie de programmation par contraintes, par le biais d'une interface dont les actions sont décrites. Enfin, nous donnons les perspectives de recherche offertes par les choix effectués dans cette intégration.

Mots clés : contraintes, objets, instanciation, programmation par contraintes, satisfaction de contraintes, relations…

Catégorie de l'article : technique

1. Introduction

Les systèmes de représentation de connaissances basés sur des objets mettent à profit la puissance déclarative de ce formalisme pour décrire, organiser et stocker de grandes quantités de connaissances. Ils offrent également divers mécanismes de raisonnement (héritage, méthodes, classification, …) qui, appliqués sur les faits présents dans une base de connaissances, permettent de déduire de nouveaux faits implicites ou hypothétiques.

Une contrainte décrit une relation à maintenir entre des variables possédant un domaine de valeurs associé. Les langages de programmation par contraintes permettent d'énoncer de manière déclarative de telles relations alors que la satisfaction est prise en charge par le résolveur de contraintes du langage. En général, les types admis pour les variables contraintes varient selon le langage (entiers, booléens, réels, objets,…)

Le but de cet article est de décrire l'intégration de contraintes à TROPES, un modèle de représentation de connaissances par objets. Fusionner contraintes et représentation par objets doit mener non seulement à contraindre des objets mais aussi à représenter les contraintes par des objets.

A ce titre, le système présenté ici est plus proche du système PROSE [1], où les concepts de la programmation par contraintes (contraintes, variables, relations) sont représentés par des objets, que de THINGLAB [2] ou KALEIDOSCOPE [5], langages de programmation par contraintes orientée-objet, où les contraintes sont spécifiées par des règles placées à l'intérieur des objets.

La satisfaction des contraintes définies dans TROPES est réalisée par PECOS ([9], [12]), une bibliothèque LE-LISP [8] de programmation par contraintes, grâce à une interface de couplage des deux systèmes.

Dans cet article sont d'abord exposés les principes de TROPES. Puis, les quatre types de contraintes offerts par le modèle sont décrits. Les différentes tâches de l'interface avec PECOS sont alors énoncées. Enfin, contributions et perspectives de recherche issues de ces travaux sont présentées.

2. Le modèle TROPES

TROPES [10] est un modèle de représentation de connaissances par objets basé sur une approche classe/instance et multi-points de vue. Dans TROPES, une **base** de connaissances est constituée de **concepts** indépendants. Chaque concept modélise une famille générique d'objets et regroupe un ensemble d'**instances** qui n'appartiennent qu'à ce concept. Un concept est partitionné en différents **points de vue** sous lesquels chaque instance peut être considérée. Les points de vue d'un concept permettent d'attacher à une instance des considérations différentes. La consultation d'une instance dans un point de vue particulier permet d'obtenir les informations particulières à ce point de vue.

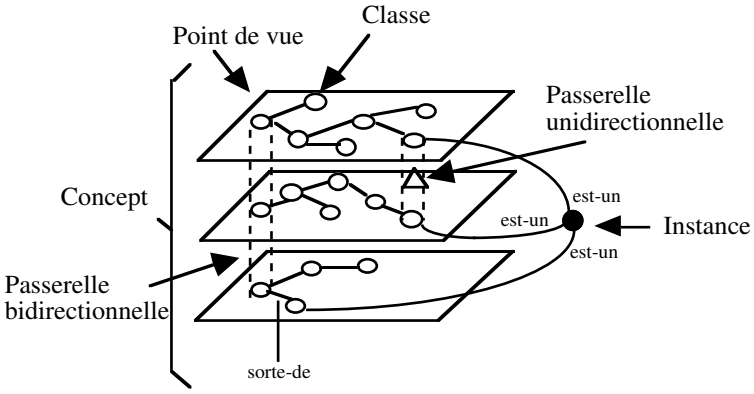


Figure 1 : Les principales entités de TROPES

Dans chaque point de vue, les **classes** sont organisées en une hiérarchie de spécialisation et reliées par le lien '**sorte-de**'. Une classe fournit en intention une description

d'instances particulières du concept; l'ensemble de ses instances constitue son extension. Une classe n'ayant qu'une seule sur-classe, dans un point de vue donné, l'héritage est simple. Les extensions des sous-classes directes d'une classe sont supposées être disjointes deux à deux et leur union être incluse dans l'extension de la classe. La classe-racine de chaque point de vue doit décrire toutes les instances du concept. Une instance est attachée *directement* à une seule classe dans chaque point de vue par le lien d'instanciation '**est-un**', mais appartient à chaque classe située sur le chemin allant de la classe-racine de la hiérarchie à cette classe. Une instance est rattachée à une classe dans chaque point de vue (cf Figure 1).

Un ensemble d'**attributs** est associé à chaque concept. Un **attribut** exprime soit une propriété, soit un lien '**partie-de**', soit un lien relationnel. Un attribut a une définition donnant ses traits généraux et constants (type, moyen d'inférence…) *pour le concept*, et une définition *pour chaque classe* où son type est affiné à l'aide de **facettes** optionnelles précisant pour l'attribut :

- son nouveau type (facette **dans**),
- le domaine des valeurs possibles (facette **domaine**),
- le domaine des valeurs impossibles (facette **sauf**),
- le cardinal de la valeur lorsque celle-ci est définie par un constructeur (liste, ensemble) (facette **card**).

La spécialisation repose sur l'ajout, dans une sous-classe, de nouveaux attributs du concept ou de nouvelles contraintes —à l'aide des facettes— sur des attributs de la sur-classe.

Exemple : Soit A un attribut défini dans une classe C, de type T_A et de domaine de valeurs admissibles Dom_A. A est affiné à nouveau dans une sous-classe C' de C. Son nouveau type est T'_A et son nouveau domaine est Dom'_A. La spécialisation requiert que :

$$T'_A \leq T_A \text{ (où } \leq \text{ est l'opérateur de la relation de sous-typage)}$$

$$Dom'_A \subseteq Dom_A$$

Lorsque la description d'un attribut demeure inchangée, elle est héritée de sa sur-classe. Un point de vue ne permet d'accéder qu'à une partie seulement des attributs du concept et constitue ainsi un masque de visibilité.

L'**instanciation** permet de créer une instance et de la rattacher, après vérification, à la classe spécifiée. L'appartenance d'une instance à une classe n'est acquise que lorsque les valeurs fournies satisfont les contraintes portant sur les attributs de la classe et exprimées à l'aide des facettes ci-dessus.

Une **passerelle** peut être établie entre deux classes (**source** et **destination**) de deux points de vue différents. Elle représente alors une relation ensembliste d'égalité (passerelle **bidirectionnelle**) ou d'inclusion (passerelle **unidirectionnelle**) entre leur extension. Ces passerelles sont utilisées lors de la classification.

La **classification** d'une instance complète ou incomplète consiste à déterminer, dans un point de vue, la classe la plus spécifique à laquelle l'instance peut être rattachée. L'instance en cours de classification "descend" dans la hiérarchie, depuis sa classe d'instanciation vers des classes plus précises, dans un but de complétion. Ce mécanisme repose sur une confrontation des valeurs présentes dans l'instance aux descriptions des attributs correspondants. Lorsque la classification ne peut être poursuivie dans un point de vue, il est possible, s'il existe une passerelle pour une des classes entérinées jusque là, de continuer le processus dans un autre point de vue et d'obtenir ainsi des informations complémentaires.

Dans TROPES, les **objets composites** possèdent des attributs (composants) qui modélisent un lien '**partie-de**' entre leur concept et ceux de leurs parties. Les valeurs des attributs composants sont des instances, appelées **composants**, d'autres concepts. Les composants dépendent du composite de manière existentielle (ils ne sont créées que pour lui) et exclusive (ils n'appartiennent qu'à lui). La classification d'un objet composite peut, le cas

échéant et conformément à la description des attributs composants, engendrer la création de ses instances composantes manquantes [11].

Une **hiérarchie de types** basée sur la relation de sous-typage est gérée parallèlement à la construction de toute base TROPES [3]. Contenant les types des attributs et des classes définis jusqu'alors, elle est exploitée pour la validation d'affinements d'attributs et de déclaration de nouvelles classes ou passerelles.

Enfin, un système de maintien du raisonnement [7] gère la **justification** et la **descendance** des attributs afin de garantir la consistance d'une base lors des modifications subies.

3. Expression de contraintes dans TROPES

3.1. Expression des contraintes à l'aide de prédicats

Les facettes de typage d'un attribut se révèlent insuffisantes lorsqu'il s'agit de décrire une propriété que devra satisfaire toute valeur. Cette propriété peut, par contre, s'exprimer aisément par une combinaison de **prédicats unaires** à l'aide des opérateurs logiques ($\wedge$, $\vee$,$\neg$), dans la facette **cond**.

Un prédicat unaire est représenté dans TROPES par une classe d'un concept 'PREDICATS', décrivant l'argument du prédicat, la fonction LE-LISP équivalente, ou une expression logique du prédicat en fonction de prédicats déjà définis.

De plus, dans une classe prédicat figure non seulement le lien 'sorte-de', mais également un double lien d'**implication** (**implique_si_vrai** et **implique_si_faux**) donnant les classes prédicat en relation d'implication logique avec le prédicat.

Exemple : Soient 'P', 'Q' et 'R' trois classes représentant des prédicats. Si l'on sait que $\forall x$ Q(x) => P(x) et que $\forall x \neg$ Q(x) => R(x) alors dans 'Q' le lien 'implique_si_vrai' contient 'P' et le lien 'implique_si_faux' contient 'R'.

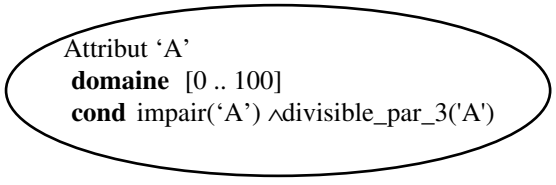
Cet apport de connaissance permettra notamment de lever l'indécision lors de la comparaison de deux attributs, dont les domaines sont identiques mais dont les facettes **cond** ont des valeurs distinctes, facilitant la construction de la hiérarchie des types et la vérification lors de raffinement d'attributs.

Supposons que la valeur d'un attribut 'A' dans une classe C doive vérifier un prédicat P. A est ensuite redéfini dans une sous-classe C' où il doit vérifier le prédicat Q. La spécialisation impose :

si Q => P **alors** l'affinement de A est correct

sinon (sans information sur P et Q) P, puis Q doivent être appliqués à toute valeur candidate (P est hérité), pour assurer l'affinement de A.

Une valeur candidate pour un attribut devra d'abord être conforme au type décrit par les facettes de typage. Puis, en présence de la facette **cond**, l'expression logique sera évaluée pour décider de son acceptation.



Exemple : L'évaluation de l'expression logique de la facette **cond** se fait par instanciation des deux classes de prédicats spécifiées, puis par évaluation de l'expression dans laquelle les prédicats sont substitués par leur résultat.

La facette **cond** en fonctionne en génération : si par son application, le domaine se réduit à une seule valeur, elle devient valeur de l'attribut pour toute instance de la classe.

3.2. Expression de contraintes à l'aide d'objets

La déclarativité du formalisme objet peut être mise à profit en décrivant une contrainte portant sur des entités TROPES par une classe TROPES d'un concept prédéfini 'CONTRAINTES'. Cette classe donne :

- le type de chaque variable contrainte

- une référence à la fonction LE-LISP destinée à poser effectivement la contrainte.

La description externe d'une contrainte sous la forme d'une classe TROPES permet de faire correspondre la pose effective de la contrainte à l'instanciation de cette classe. La fonction LE-LISP chargée de déclencher la contrainte n'a qu'un seul paramètre : l'instance de contrainte. Elle fait appel à des fonctions prédéfinies de construction de contraintes qui agiront sur les valeurs des attributs de l'instances (i.e. les variables contraintes). Ce principe, à la base de l'attachement procédural dans SHIRKA [13], est ici repris pour définir des contraintes entre instances, sur des classes et entre classes.

3.2.1. Contraintes entre instances

Il est possible dans TROPES de contraindre des attributs d'instances distinctes (ou de la même instance) en instanciant simplement la classe de 'CONTRAINTES' (cf Figure 2) correspondant à la relation à maintenir entre ces attributs.

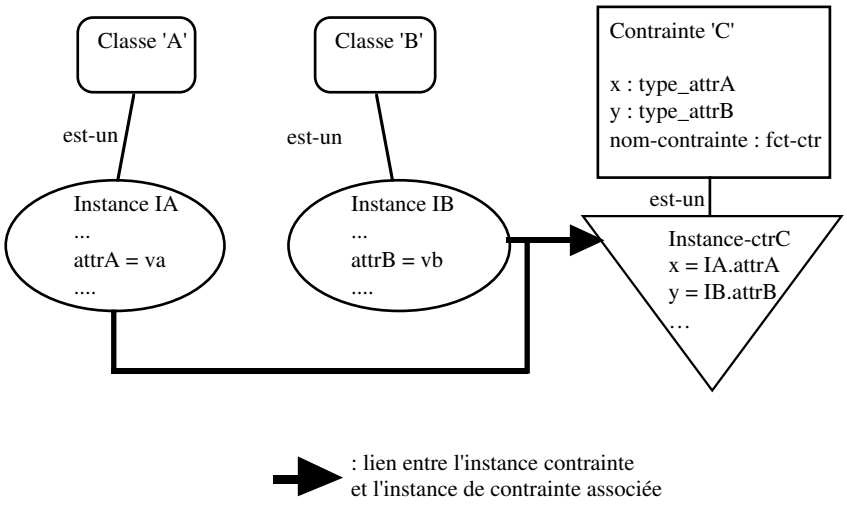


Figure 2 : Contraintes entre instances. Une instance ctrC est créée pour lier les attributs attrA et attrB de deux instances IA et IB. Au niveau de l'instance contrainte, les liens avec instances de contraintes sont enregistrés.

La représentation par une instance de l'existence d'une contrainte entre des instances de la base, permet d'utiliser les opérations classiques de modification et de suppression d'instance :

• modifier la valeur d'un attribut dans une instance de contrainte revient à faire porter la contrainte sur un autre attribut de la base, de type compatible, et donc à libérer l'ancien attribut contraint. Lorsqu'un attribut dont la valeur fut inférée par la contrainte est libéré, il n'est alors plus supporté et sa valeur n'est donc plus valide, pas plus que toute valeur de sa descendance. Une propagation paresseuse est alors effectuée par le Système de Maintien du Raisonnement pour signaler l'inconsistance.

• supprimer une instance de contrainte revient à libérer les instances contraintes de ce lien et à propager l'invalidité des valeurs depuis tout attribut inféré.

Lorsqu'un attribut contraint est modifié par l'utilisateur :

- soit l'attribut n'a pas encore de valeur : si la valeur fournie se trouve dans le domaine réduit par la contrainte, elle est acceptée, sinon l'incompatibilité est signalée

- soit l'attribut a une valeur (inférée ou non par la contrainte) : celle-ci n'est remplacée par la nouvelle valeur que si la contrainte peut être également satisfaite avec cette valeur, sinon l'inconsistance due à la présence de la contrainte est signalée.

3.2.2. Contraintes de classe

Une contrainte de classe exprime une relation à maintenir entre les attributs d'une même classe. Toute instance de cette classe doit satisfaire cette contrainte. Pour chaque classe TROPES, l'ensemble de ses contraintes est donné par l'attribut '**contraintes**'

```
Classe 'Cercle'
attributs = (centre_x, centre_y,
             rayon, surface, …)
contraintes = { Pi-R2(surface, rayon),
                …}
```

Dans la classe 'Cercle' , la propriété :

surface = PI * rayon* rayon,

est établie en indiquant la classe de 'CONTRAINTES' à instancier et les attributs contraints.

Pour chaque instance dont l'appartenance à une classe contrainte est établie, chaque contrainte est chargée :

• de vérifier, lorsque les valeurs des attributs contraints sont fournies, qu'elles satisfont bien la relation, (vérification de la consistance locale),

• de réduire, lorsque des valeurs d'attributs contraints sont manquantes, le domaine de ces attributs, en accord avec la relation (consistance d'arc),

• d'évaluer, lorsque c'est possible, la dernière valeur manquante d'attribut contraint et de la propager (propagation de valeur).

Il faut noter que lors de la classification, le rôle de la contrainte doit se limiter à la première tâche : une simple vérification. En effet, la contrainte de classe ne peut agir en génération puisque précisément l'appartenance à la classe n'est pas encore prouvée.

```
Contrainte 'Pi-R2'
 x dans réel
 y dans réel
 nom-contrainte = 'tr-ct-pi-r2'
```

```
(de tr-ct-pi-r2 (inst-de-contrainte)
        (tr-idem 'x
                (tr-mult 3.159
                        (tr-power 'y 2)))
)
```

Exemple : la classe 'Pi-R2' décrit le type de chaque attribut impliqué ainsi qu'une fonction LE-LISP 'tr-ct-pi-r2' donnant les fonctions TROPES qui permettront de poser *effectivement* la contrainte.

Lorsqu'une instance est rattachée à la classe contrainte, une instance de la contrainte correspondante est créée.

tr-idem, tr-mult and tr-power sont des fonctions TROPES de l'interface TROPES/PECOS qui gèrent la pose des contraintes PECOS équivalentes (cf §4).

La justification d'un attribut dont la valeur a été inférée par une contrainte fait référence à l'instance de contrainte.

Vis-à-vis des passerelles, les règles suivantes doivent être appliquées :

- Une contrainte portant sur une des classes d'une passerelle bidirectionnelle porte également sur l'autre classe.

- Une contrainte portant sur la classe destination d'une passerelle unidirectionnelle est également fixée sur la classe source, car ses instances sont toutes instances de la classe destination.

- Une contrainte portant sur la classe source d'une passerelle unidirectionnelle n'est pas reportée sur la classe destination.

Le long du lien 'sorte-de', les contraintes d'une classe sont héritées par ses sous-classes.

La modification et la suppression d'une instance de contrainte de classe ne sont pas possibles *directement*. La contrainte étant la "propriété" de la classe, une instance ne peut s'en défaire en particulier. A ce titre, les instances de contraintes créées pour des instances de classe contrainte sont "cachées". C'est l'élimination de la référence dans l'attribut 'contraintes' de la classe qui libérera la classe (toutes ses instances) de la contrainte.

Enfin, les contraintes étant elles-mêmes des classes, il est possible de définir des contraintes sur des contraintes, mais cette perspective n'a pas encore été approfondie…

3.2.3. Contraintes entre classes ou relations

Dans TROPES, des attributs d'une classe sont destinés à exprimer la possible implication de l'instance dans une relation particulière avec des instances d'autres classes. Ces attributs **relation** représentent une potentialité de la classe : dans les instances sur lesquelles la relation ne porte pas, ils n'ont pas de valeur.

Une **relation** entre des classes induit un ensemble de contraintes portant sur les attributs des instances reliées. Elle est modélisée par une classe du concept 'RELATION' (cf Figure 3) qui décrit les classes impliquées dans la relation (qui contiennent un attribut dédié à cette relation) et l'ensemble des contraintes imposées par la relation (donné par l'attribut '**requiert**').

La définition d'un attribut relation pour une classe est chargée (via l'attribut **attr-rel**) d'indiquer l'attribut correspondant dans la classe 'RELATION'. Dans chacune des instances en relation, l'attribut relation représente le lien inverse de la relation au regard des autres instances.

A la première affectation de valeur d'un des attributs relation d'une instance, le lien inverse est complété et une instance de la relation associée est créée. Cette instanciation déclenche à son tour les instanciations des classes de contraintes répertoriées par l'attribut 'requiert'. Ainsi, les contraintes sont posées sur les attributs contrôlés par la relation dans les instances nouvellement unies.

Exemple : Soit la relation 'Père/Enfant' établie entre deux classes 'Homme-Adulte' et 'Homme-Enfant' à laquelle sont associées les contraintes suivantes :

- le nom d'un enfant doit être celui de son père

- l'âge d'un enfant doit être inférieur à celui de son père d'au moins quinze ans.

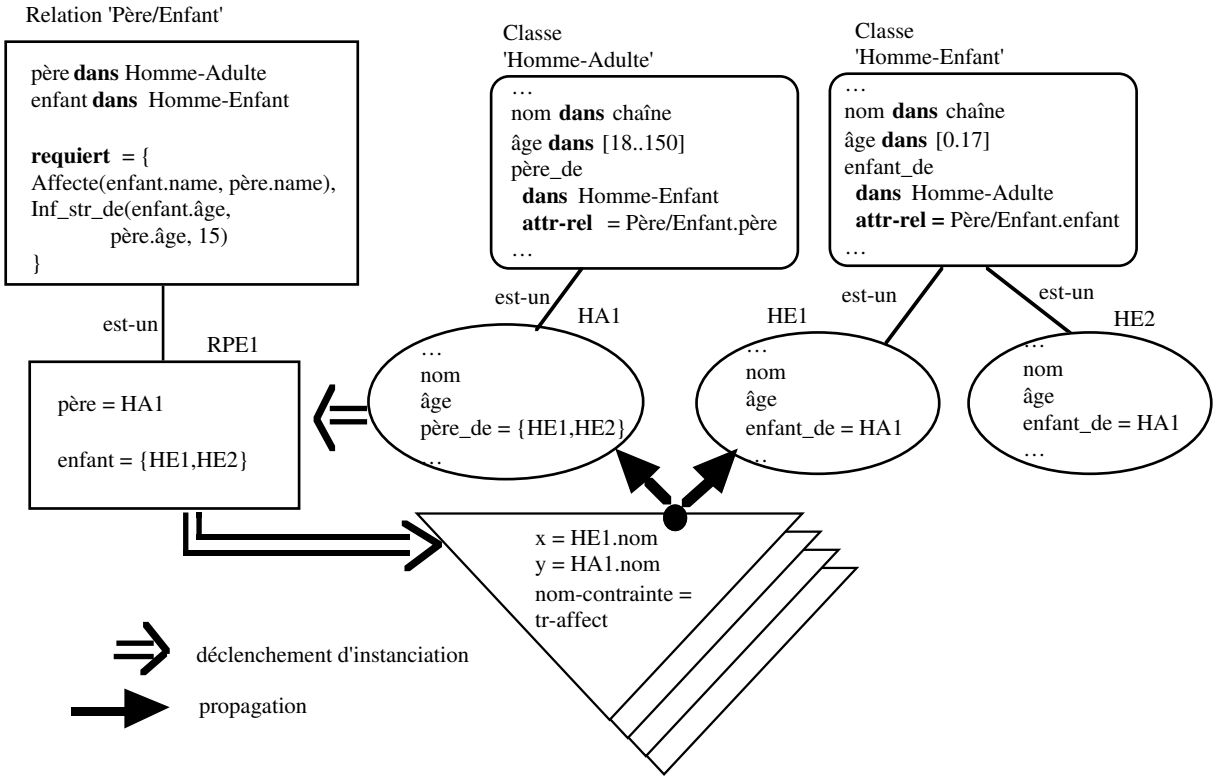


Figure 3 : Relation. Une classe 'Père/Enfant' décrit la relation donnée en exemple. L'instance RPE1 symbolise la relation 'Père/Enfant' entre HA1 et HE1, HE2. Les contraintes instanciées permettront de déduire les domaines ou valeurs de ces attributs (ici nom et âge).

De même, l'utisateur peut choisir d'instancier une classe de relation, les valeurs de l'instance créée seront alors reportées dans les attributs relation des instances reliées.

La modification d'une instance de relation peut étendre, modifier ou réduire l'ensemble des instances en relation. Lorsqu'une instance est libérée, il faut annuler la valeur de l'attribut relation et procéder à une propagation de l'invalidité des valeurs depuis les attributs inférés par les contraintes de la relation.

La suppression d'une instance de relation libère l'ensemble des instances qu'elle liait.

4. Satisfaction de contraintes dans TROPES

4.1. Utilisation de la librairie PECOS

La satisfaction des contraintes décrites et fixées dans TROPES est assurée par PECOS, une librairie LE-LISP, qui permet une programmation par contraintes sur des variables :

- ayant un domaine fini sans restrictions de types (entières, chaînes de caractères, listes, …)
- entières dont le domaine est énuméré ou un intervalle,
- réelles (à virgule flottante) sur un intervalle,
- ensemblistes dont le domaine est un ensemble d'ensembles.

Des opérations de base permettent de manipuler les variables contraintes (déclaration, affectation, élimination de valeur du domaine, …). Des contraintes prédéfinies (d'égalité, arithmétiques, de cardinalité, …) peuvent alors être combinées et appliquées aux variables définies. De plus, l'utilisateur a la possibilité de définir ses propres contraintes grâce à un mécanisme de démons basé sur des *schémas* de contraintes qui, activés, déclencheront les règles de propagation. Une contrainte définie et posée peut être inhibée (les variables contraintes sont libérées mais ne retrouvent pas leur domaine initial) et réactivée.

Dans PECOS, la satisfaction des contraintes est assurée par une propagation locale qui consiste à :

- propager les valeurs connues afin d'inférer les valeurs manquantes
- éliminer les valeurs impossibles des domaines

PECOS offre également une programmation en mode non déterministe qui, grâce à un mécanisme de retour-arrière, permet de tester différentes alternatives (par exemple, par énumération des domaines, on peut chercher à obtenir une ou toutes les solutions d'un problème). Enfin, bien que PECOS gère également des contraintes sur des objets ou des classes LE-LISP, TROPES n'utilise pas cette option mais se base sur un principe similaire : réaction et pose de la contrainte à chaque rattachement d'une instance à la classe.

4.2. L'interface TROPES/PECOS

Tropes utilise la puissance de résolution de Pecos en proposant une interface chargée de lier tout objet contraint de TROPES à une variable contrainte équivalente gérée par PECOS. Le rôle de cette interface consiste à :

• offrir des fonctions de programmation par contraintes écrites en LE-LISP qui déclencheront les fonctions PECOS chargées de la pose de contraintes.

• gérer la création, la mise à jour et la suppression des variables contraintes PECOS sous-jacentes, en accord avec la sémantique donné par TROPES à chaque type de contrainte. Il faut notamment tenir compte du fait que TROPES permet la modification de valeurs d'attributs alors que PECOS interdit l'extension des domaines et la ré-instanciation.

• gérer l'union d'intervalles que peut caractériser un domaine d'attribut de TROPES, alors que PECOS n'associe qu'un intervalle unique à une variable contrainte. Ceci est réalisé par l'introduction de **variables contraintes composites** qui correspondent à une liste de

variables contraintes PECOS. Ainsi, une contrainte portant sur une variable contrainte composite est appliquée à chacune de ses variables contraintes. Si l'une d'entre elles présente un domaine vide, c'est que l'intervalle correspondant n'offre pas de valeurs satisfaisant la contrainte : l'union peut donc être réduite.

• Analyser et réagir aux erreurs signalées par PECOS. Les fonctions du mode non déterministe s'avèrent alors utiles pour contrôler les échecs et restaurer un état valide sur les variables contraintes PECOS.

5. Contributions de cette intégration

L'intégration de contraintes permet à présent dans TROPES :

- d'associer une propriété à la valeur d'un attribut et donc d'étendre la description d'un attribut (contrainte sur un attribut)

- d'établir à tout moment un lien entre des instances en reliant des attributs par règles qui, devant être vérifiées, pourront fonctionner en inférence (contrainte entre instances, contraintes entre classes)

- d'ajouter un niveau de description supplémentaire aux classes en permettant l'expression de propriétés entre les valeurs d'attributs de chaque instance d'une classe (contrainte de classe)

- d'impliquer, autrement que par le lien 'partie_de', des instances de classes d'un même concept ou de concepts différents dans une relation dont les contraintes permettront d'inférer les valeurs des attributs dépendants de cette relation (contrainte entre classe).

Les principes de cette intégration sont semblables à ceux adoptés dans PROSE ou PECOS.

Cependant, PROSE est avant tout un modèle abstrait de représentation et de manipulation de contraintes destiné à un modèle de représentation de connaissances par objets quelconque. Cette boîte à outils offre donc une représentation des contraintes basées sur trois classes *Variable*, *Constraint* et *Relation*, et se charge également de la satisfaction des contraintes ainsi que du maintien de la cohérence. La pose d'une contrainte se fait par instanciation de la classe *Constraint*, l'instance résultante étant destinée à lier des instances de *Variable* (objets représentant les attributs contraints) à une instance de *Relation* (qui décrit le prédicat à respecter et les règles de déduction de valeurs). Dans TROPES, la classe *Variable* n'a pas d'équivalent car la contrainte porte directement sur les attributs, alors que la classe CONTRAINTE rassemble les informations des classes *Constraint* et *Relation*. Le maintien de la cohérence est assuré dans TROPES par le système de maintien du raisonnement et par l'interface TROPES/PECOS. D'autre part, PROSE ne permet pas de poser des contraintes sur des classes.

PECOS offre des fonctions de programmation orientée-objet et permet de contraindre un objet donné ou tous les objets d'une classe. Mais la pose de contrainte doit être anticipée dès la définition de la classe de cet objet en définissant les attributs contraints comme tels. Dans TROPES, les contraintes entre instances n'impose pas ce prérequis.

Enfin, la description d'une contrainte ou d'une relation par un objet TROPES permet à l'utilisateur de déclarer des contraintes sans changer de formalisme et de les poser, modifier ou supprimer à sa guise comme il le fait pour tout autre objet de TROPES.

6. Travaux futurs

Les fonctions de l'interface TROPES/PECOS sont actuellement en cours de développement.

Le travail de conception réalisé doit maintenant être étendu à l'expression et à la gestion :

- d'opérateurs usuels sur les relations : union, intersection, composition…
- de propriétés usuelles des relations : (anti) réflexivité, (anti) symétrie, transitivité

• de contraintes (de contrôle) sur les relations :

Soient R une relation entre deux classes C et C', I et I' instances de C et C'. On peut vouloir imposer que ∀ I, ∀ I', I R I' , ou imposer une contrainte de cardinalité sur le nombre d'instances I' de C' telles que I R I', etc…

• de contraintes entre un objet composite et ses composants ([4], [6]) signifiant que :

- le composite hérite certaines propriétés de ses composants (lien d'agrégation)

- les composants héritent certaines propriétés du composite (lien de décomposition)

Ces partages semblent pouvoir s'exprimer et être effectués aisément, grâce aux objets relations TROPES.

• de contraintes entre des instances d'une même classe (qui semble être un cas particulier de relation) :

Soient I et I', deux instances d'une classe C. On peut exiger que :

∀ I, ∀ I' I.attr Λ I'.attr , où Λ est un opérateur de comparaison.

Ce type de contrainte introduit un ordre entre des instances qui pourra être exploité dans des problèmes de contraintes temporelles.

• de hiérarchies de relations. Il faudra alors définir la sémantique de la spécialisation adoptée.

En effet, en reprenant l'exemple du §3.2.3 :

- La relation 'Père/Fils' peut être vue comme une sous-relation de 'Père/Enfant',

- la relation 'Père/Enfant' entre la classe 'Homme-Adulte' et 'Homme-Enfant' peut être vue comme une sous-relation de la relation 'Père/Enfant' entre 'Homme' et 'Homme', puisque 'Homme' est sur-classe de 'Homme-Adulte' et de 'Homme-Enfant'.

Par spécialisation, l'ensemble des contraintes attaché à une relation devrait alors être hérité ou renforcé dans ses sous-relations.

Enfin, l'interface TROPES/PECOS devra plus encore mettre à profit le mode non déterministe. Notamment, l'énumération des domaines peut mener à un raisonnement hypothétique, permettant d'observer l' évolution d'une base de connaissances, avec remise en cause des valeurs hypothétiques choisies qui mènent à une inconsistance.

7. Conclusions

TROPES est un modèle de représentation de connaissances par objet dans lequel l'intégration de contraintes repose sur deux tâches distinctes : l'expression et la satisfaction.

La représentation des contraintes est assurée par des classes de concepts prédéfinis de TROPES. Dès lors, la déclaration, la pose, la modification et la suppression des contraintes sont contrôlées par les outils de construction et de manipulation d'objets offerts par le modèle.

La satisfaction des contraintes, quant à elle, est déléguée une bibliothèque de programmation par contraintes, les échanges (traduction des contraintes TROPES, interprétation des résultats PECOS) étant contrôlés par une interface entre les deux systèmes.

Exprimées en tant qu'objets du modèle, les contraintes peuvent alors être manipulées comme tout autre objet, pour être posées, modifiées ou supprimées.

Nos recherches doivent maintenant s'orienter à la fois vers les relations entre classes (gestion d'opérateurs et de propriétés, établissement de liens de spécialisation, objets composites…) et vers l'intégration d'un raisonnement prospectif à base d'hypothèses exploitant les primitives de la programmation en mode non déterministe de PECOS.

Bibliographie

[1] P. Berlandier, B. Neveu. PROSE : Un système de contraintes facilement intégrable dans un langage objet. *Actes des 1ères journées Représentation par objets*, La Grande Motte, France, Juin 1992.

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

[3] C. Capponi, M. Chaillot. Construction incrémentale d'une base de classes correcte du point de vue des types. *Actes des 2èmes Journées "Validation des sytèmes à base de connaissances" JAVA'93 (à paraître)*, Saint Raphaël, France, Mars 1993.

[4] P. Dugerdil. Les mécanismes d'héritage d'OBJLOG : vertical et sélectif multiple avec points de vue. *Actes du 6ème congrès RFIA*, Antibes, France, Novembre 1987.

[5] B. Freeman-Benson. KALEIDOSCOPE: Mixing Objects, Constraints, and imperative Programming. *Proceedings of the 1990 Conference on Object-Oriented Programming*, pages 77-88, Ottawa, Canada, Octobre 1990. ACM.

[6] M. Fornarino. Mise en œuvre de l'héritage au moyen de relations. *Rapport de Recherche INRIA N°1417*, Sophia Antipolis, France, Avril 1991

[7] J. Gensel. Gestion des dépendances et des hypothèses dans un modèle de connaissances à objets. *Rapport de DEA d'Informatique*, INPG, Grenoble, Juin 1990.

[8] ILOG. LE-LISP *Version 16.01*, *Reference Manual*, Gentilly, France, Avril 1992.

[9] ILOG. PECOS *Version 1.1*, *Manuel de référence,* Gentilly, France, Mars 1992

[10] O. Mariño, F. Rechenmann, P. Uvietta. Multiple Perspectives and Classification Mechanism in Object-Oriented Representation. *Proceedings of the ECAI'90*, Stockholm, Suède, Août 1990.

[11] O. Mariño. Classification d'objets composites dans un système de représentation de connaissances multi-points de vue. *Actes du 8ème congrès RFIA*, Lyon, France, Novembre 1991.

[12] J.F. Puget. Programmation par contraintes orientée-objet. *Actes de la conférence scientifique Avignon'92*. Avignon, France, Juin 1992.

[13] F. Rechenmann, P. Fontanille, P. Uvietta. SHIRKA : système de gestion de bases de connaissances centrées-objet, *Manuel d'utilisation*, Grenoble, 1990.