THÈSE

présentée par

Jérôme Gensel

pour obtenir le titre de

DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER

(Arrétés ministériels du 5 juillet 1984 et du 30 mars 1992)

Spécialité

INFORMATIQUE

Contraintes et représentation de connaissances par objets
Application au modèle Tropes

Soutenue le 26 octobre 1995 devant le jury composé de :

M.	Laurent	TRILLING	Président
Mme.	Marie-Catherine	VILAREM	Rapporteur
M.	Pierre	COINTE	Rapporteur
M.	Bernard	CARRÉ	Examinateur
M.	Jérôme	EUZENAT	Examinateur
M.	François	RECHENMANN	Directeur de thèse

Thèse préparée au sein du laboratoire LIFIA/IMAG

Je tiens a remercier,

Monsieur Laurent Trilling, Professeur à l'Université Joseph Fourier, de participer au jury de cette thèse. Cet honneur lui revient de droit : il fut le premier à me parler de contraintes dans ses cours de Prolog.

Madame Marie-Catherine Vilarem, Professeur au Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier, et Monsieur Pierre Cointe, Professeur à l'École des Mines de Nantes, d'avoir accepté de juger ce travail dans des délais aussi serrés et de m'avoir aidé à améliorer ce mémoire.

Monsieur Bernard Carré, Maître de Conférences à Lille, d'avoir bien voulu s'intéresser à mon travail.

Monsieur Jérôme Euzenat, Chargé de Recherche à l'INRIA Rhône-Alpes, d'avoir suivi mon travail avec intérêt. Ses critiques avisées lors du développement de cette étude ont été très enrichissantes.

Monsieur François Rechenmann, Directeur de Recherche à l'INRIA Rhône-Alpes, de m'avoir accueilli dans son équipe. Je lui suis très reconnaissant de la confiance qu'il m'a témoignée toutes ces années.

Les lecteurs de ma thèse – Cécile, Danielle, Alejandro, Jérôme, Patrice et Pierre – pour leurs précieux conseils.

Cécile Capponi pour sa gentillesse, sa disponibilité et sa collaboration de tous les instants. Merci infiniment de m'avoir supporté (au sens propre et figuré !).

Alejandro Quintero pour son soutien scientifique et moral sans relâche, même depuis Bogota, durant les derniers mois de cette thèse. Merci d'avoir sacrifié jusqu'à tes lendemains de fête !

Pierre Girard et Rubby Casallas-Gutierrez pour m'avoir aidé à parcourir ce long chemin chaotique qui aujourd'hui arrive, pour nous trois, à destination. Merci d'avoir tant partagé.

Les autres membres (nouveaux et anciens) du projet Sherpa pour leur bonne humeur et pour leur aide dans la présentation de ce travail :

Danielle ("la vie est belle ?"), Florence ("le café est prêt"), Françoise ("ça y est, c'est réglé"), Jutta ("je cherche Jérôme"), Katia ("un autre morceau de gâteau ? ") , Nathalie ("pourquoi tu dis ça ?"), Nina ("Syrien, ca va passer"), Olga ("j'ai vu le couvreur. . . "), Sueli ("porque é que você não fala brasileiro ?"), Ysabelle ("Jérôme, avec un "G", non ?").

Bruno (dépanneur Unix en son genre), Gilles (éleveur de pommes), Jean-Marc (confiseur), Jean-Yves (ah! quel bonheur d'avoir un ami bricoleur), Olivier (expérimentateur LaTeX), Patrice (éditeur), Petko (3615 code MATHS), Pierre (gardien du parc).

Mes parents et ma famille pour leur présence, leur aide et leurs encouragements.

Enfin et surtout, Marie-Laure, pour tout.

Table des matières

Chapitre 1

Introduction

1.1 Le contexte

Les Systèmes de Représentation de connaissances Par Objets (ou SRPO) permettent de décrire, d'organiser et de stocker des connaissances à l'aide d'entités de description appelées *objets*. Le plus souvent ces systèmes distinguent deux types d'objets : les *classes* et les *instances*. Les classes sont les objets générateurs qui permettent la partition du domaine modélisé en diverses catégories. Une classe recueille les caractéristiques communes aux éléments d'une catégorie. Ces caractéristiques sont appelées *attributs* et sont spécifiées par un ensemble de *facettes* ou *descripteurs*. Les instances, quant à elles, représentent des éléments de ces catégories. Elles ont la même structure que les classes auxquelles elles appartiennent mais contiennent les valeurs des attributs.

Comme dans les langages de programmation orientée-objet, les classes sont organisées hiérarchiquement par une relation de spécialisation qui induit une inclusion entre les ensembles d'instances (ou *extensions*) de deux classes ainsi reliées. Sur la hiérarchie de classes, un mécanisme d'*héritage* est mis en place qui favorise la factorisation de l'information aux niveaux les plus hauts.

De plus, ces systèmes intègrent des mécanismes d'inférence dont le but est de produire de la connaissance à partir de la connaissance disponible. Les deux principaux mécanismes sont l'activation de méthodes d'obtention de valeurs d'attributs et la classification d'instances.

Ces fonctionnalités font de ces systèmes des outils déclaratifs de représentation et d'exploitation de connaissances structurées qui sont suffisamment généraux pour être utilisés dans des domaines aussi divers que la médecine, la biologie moléculaire, le calcul scientifique, etc.

TROPES est le SRPO qui sert de support à cette étude. Il intègre les principales caractéristiques des SRPO et présente deux particularités à travers les notions de *concept* et de *point de vue*. Un concept est une entité de représentation génératrice d'instances qui comporte un certain nombre de points de vue. À chaque point de vue correspond une hiérarchie arborescente de classes (qui sont des sous-catégories du concept). L'unique classe de rattachement d'une instance de concept dans un point de vue désigne la catégorie dans laquelle est placée l'instance observée sous ce point de vue. La notion de *passerelle* permet d'indiquer une relation d'inclusion ou d'égalité entre les extensions de classes de points de vue distincts. Dans TROPES, la classification constitue le principal mécanisme d'inférence et opère sur les différents points de vue en exploitant les passerelles.

1.2 La problématique

Les notions centrales de concept et de point de vue sur lesquelles reposent toute modélisation d'un domaine de connaissances dans TROPES, font de lui un système original qui intègre également les outils de représentation (classes, attributs, facettes) et les mécanismes d'exploitation de la connaissance (méthodes, classification) des autres SRPO.

Cependant, dans TROPES comme dans les autres SRPO, si la panoplie des descripteurs d'attributs est suffisamment complète pour en décrire le domaine de valeurs, elle ne l'est pas lorsqu'il s'agit d'exprimer et de maintenir des contraintes (numériques ou symboliques) impliquant plusieurs attributs du même objet ou d'objets distincts. Tenter de maintenir la cohérence d'une contrainte posée sur les attributs d'un objet à partir des outils disponibles – prédicats et méthodes – va à l'encontre de la déclarativité du système de représentation et ne rend pas compte non plus de la *portée* de cette contrainte auprès des domaines des attributs contraints.

Ce constat est le point de départ de cette étude dont l'objectif premier est de permettre un raisonnement par contraintes, c'est-à-dire l'expression et la résolution de contraintes définies sur les objets de TROPES. Au delà des considérations techniques, nous devons également mesurer les répercussions, sur les entités de représentation et sur les mécanismes d'exploitation, de la présence de contraintes au sein d'un SRPO comme TROPES. Enfin, nous devons juger si, hormis ses utilisateurs, le SRPO lui-même a quelque avantage à retirer de la fonctionnalité qui vient de lui être ajoutée. Une telle investigation autour de l'association contraintes/représentation de connaissances par objets n'a pas été menée jusqu'à ce jour, et mérite donc d'être effectuée, notamment dans le cadre de TROPES.

1.3 Les éléments de solution

Pour répondre à la question " comment intégrer des contraintes dans un système de représentation de connaissances? ", il faut s'intéresser aux outils et aux techniques de la programmation par contraintes.

La programmation par contraintes connaît un remarquable essor depuis le milieu des années quatre vingts bien que ses bases aient été lancées depuis les années soixante.

D'un côté, les langages de programmation par contraintes dépassent de plus en plus souvent le stade prototypique des laboratoires de recherche pour devenir des produits commercialisés qui témoignent de leur puissance et de leur accessibilité dans des domaines autrefois réservés à la recherche opérationnelle, comme les problèmes d'ordonnancement, d'optimisation, d'allocation de ressources, etc.

De l'autre, la théorie s'évertue depuis vingt ans à proposer des algorithmes de maintenance et de résolution pour des Problèmes de Satisfaction de Contraintes (ou CSP) à traiter, qu'ils soient numériques, booléens ou ensemblistes, figés ou évolutifs – selon la nature des contraintes et des domaines des variables du problème –, qui sont par essence NP-complets. Ces algorithmes exploitent souvent des résultats de la théorie des graphes ou encore de l'analyse numérique.

Parmi la diversité des cas de CSP abordés par cette théorie, notre choix doit se porter vers ceux qui s'identifient le mieux à notre problématique.

En identifiant les attributs des objets comme les variables des CSP définis dans TROPES (ou CSP TROPES), nous sommes plus particulièrement intéressés par des CSP à domaines finis ou infinis. Aussi, l'étude des CSP numériques à intervalles, bien que généralement limitée aux CSP à intervalles uniques, est un premier élément de réponse apporté par la théorie des CSP pour l'intégration à réaliser.

Par ailleurs, la dynamicité étant une propriété essentielle des bases de connaissances des SRPO, nous souhaitons que les CSP en bénéficient également et qu'il soit donc possible d'ajouter ou de retirer à tout moment une contrainte. Aussi, l'étude des CSP dynamiques, bien que limitée aux CSP à domaines finis, est un second élément de réponse apporté par la théorie des CSP pour l'intégration à réaliser.

Nous nous proposons d'adapter les résultats concernant les CSP à intervalles et les CSP dynamiques au contexte des SRPO en nous intéressant notamment à la gestion de CSP dynamiques dont les domaines sont représentés par des unions d'intervalles finis ou infinis.

1.4 La démarche suivie

Partant du constat du gain d'expressivité que constitue l'intégration de contraintes dans TROPES, nous cherchons tout d'abord à identifier les composants d'un CSP TROPES, c'est-à-dire les variables, leurs domaines et les contraintes.

- Les attributs de TROPES sont désignés comme les seules variables des CSP TROPES.
- Dès lors, les domaines des CSP TROPES sont connus et correspondent aux domaines des attributs contraints.
- Les contraintes doivent permettent l'expression et la maintenance de relations entre les attributs. Les types des contraintes attendues sont donc dictés par les types des attributs à contraindre. Aussi, outre des contraintes d'égalité et de différence applicable à tous les types, et les classiques contraintes numériques et booléennes à définir sur des attributs monovalués, des contraintes sur des ensembles ou des listes semblent également indispensables pour contraindre les attributs multivalués de TROPES. De même, des contraintes non déterministes (dont la pose est sujette à un choix) permettent d'augmenter la flexibilité des CSP.

Les contraintes étant définies, il faut alors définir les niveaux de représentation auxquels elles peuvent être déclarées. Dans TROPES, trois niveaux sont suggérés : le concept, la classe et l'instance. L'extension d'un concept ou d'une classe définit alors la portée d'une contrainte, de sorte que, cette contrainte soit posée sur toute instance de l'extension.

Afin de désigner les attributs contraints, la notion d'*accès* est introduite. Un accès est un argument de contrainte qui désigne un unique attribut, ou bien un ensemble ou une liste d'attributs impliqués par la contrainte. Les instances de TROPES pouvant être liées entre elles par des attributs-liens (dont la valeur est une instance, un ensemble ou une liste d'instances), il est possible de désigner indirectement un attribut à contraindre depuis une instance. L'accès passe alors par différents attributs-liens depuis l'instance source pour atteindre le ou les attributs destinations à contraindre. La notion d'*accès* dans TROPES constitue une extension de la notion de *chemin* présente dans d'autres langages alliant contraintes et objets.

Si les contraintes deviennent des éléments de représentation de connaissances à part entière dans TROPES, il convient d'être en mesure de les manipuler de manière dynamique. Aussi, il doit être possible d'ajouter, de supprimer une contrainte, comme toute autre entité de représentation de connaissances dans le modèle, ou de modifier le domaine d'un attribut contraint. Nous étudions les répercussions de l'ajout et du retrait d'une contrainte.

La maintenance d'un certain niveau de *consistance locale* dans les CSP TROPES influe directement sur le domaine des attributs contraints. Il s'agit de garantir que, pour toute valeur d'un domaine d'attribut contraint, il existe une valeur dans le domaine de chaque attribut avec lequel il est impliqué dans une contrainte, telle que la contrainte en question soit satisfaite. Le filtrage opéré à cette fin est alors susceptible de ne conserver qu'une seule valeur dans le domaine d'un attribut contraint permettant de ce fait l'inférence de la valeur en tant que seule possible et cohérente. Les contraintes constituent donc un moyen d'inférence supplémentaire pour TROPES.

Plus généralement, les types des entités contraintes doivent refléter la consistance atteinte et donc se baser sur les domaines établis par la propagation de contraintes. Autrement dit, il doit s'opérer une communication entre le module de gestion des CSP TROPES et METÉO [Capponi95], le module de gestion des types de TROPES.

Les contraintes augmentent donc l'expressivité, la cohérence et les capacités d'inférence de TROPES.

À ce stade de notre étude, les fonctionnalités du module de programmation par contraintes destiné à gérer les CSP TROPES sont énoncées : il doit être capable de maintenir et de résoudre des CSP dynamiques à domaines finis ou infinis impliquant des nombres, mais aussi des booléens, des ensembles ou des listes.

Nous avons tout d'abord couplé TROPES à PECOS, un module de programmation par contraintes commercial. Cette solution n'ayant pas donné réellement satisfaction, nous avons choisi de concevoir un module de programmation par contraintes, appelé MICRO (Module pour l'Intégration de Contraintes et de Relations dans les Objets), dont les principales caractéristiques sont :

- le traitement de CSP numériques à unions d'intervalles ;
- le traitement de CSP booléens ou semi-booléens ;
- le traitement de CSP sur des ensembles ;
- le traitement de CSP sur des listes ;
- un algorithme de propagation de contraintes ;
- un algorithme de résolution de contraintes pour les CSP à domaines finis et les CSP numériques à domaines infinis ;
- la gestion de CSP dynamiques ;
- un ensemble prédéfini de méta-contraintes (contraintes sur des ensembles ou des listes de variables, contraintes conditionnelles et contraintes non déterministes).

Ce module fonctionne en mode autonome. Une interface entre TROPES et MICRO permet de réaliser un couplage *semi-faible* dans lequel les attributs contraints de TROPES se substituent *directement* aux variables des CSP gérés par MICRO, évitant ainsi les redondances inévitables lors d'un couplage faible.

Grâce à cette interface, le modèle TROPES peut accueillir des contraintes sur les attributs de toutes les instances d'un concept, de toutes les instances d'une classe, ou d'instances particulières. Les contraintes doivent alors cohabiter avec les divers mécanismes d'exploitation des connaissances du modèle. Nous étudions donc le comportement de ces mécanismes en présence de contraintes et décidons d'un mode d'utilisation des contraintes lors de l'instanciation et de la classification, face aux méthodes et dans les passerelles. Cette étude, menée dans le cadre de TROPES, vaut également pour tous les SRPO intégrant des principes de représentation et des mécanismes d'exploitation similaires.

L'intégration réalisée permet à la fois d'exprimer des relations entre les attributs des instances TROPES dont l'expression n'était pas possible avant, et de garantir un certain niveau de cohérence qui n'était pas maintenable avant.

Le modèle TROPES permettant à présent la définition et la résolution de contraintes sur les objets des bases de connaissances, nous poursuivons notre étude en nous intéressant à l'apport des contraintes pour le système lui-même.

Nous montrons que les contraintes peuvent être mises à profit pour contrôler le partage de valeurs entre des objets, notamment des objets composites, ou encore le passage de paramètres entre les objets d'un modèle de tâches. De même, la résolution de contraintes s'avère un précieux atout dans l'exploration d'une base de connaissances pour la production d'instances hypothétiques.

Enfin, nous proposons une façon d'introduire dans TROPES les notions de *relation* et de *filtre* en montrant que l'expression et la construction de relations ou de filtres peuvent exploiter et être facilitées par les contraintes.

1.5 Les apports de ce travail

Les contributions de ce travail se situent dans le domaine de la représentation de connaissances par objets et ont été appliquées, dans ce contexte, au modèle TROPES qui a servi de base d'étude et de développement.

- Nous augmentons l'expressivité et les capacités d'inférence du modèle en permettant la déclaration et la maintenance de contraintes sur les attributs des instances et ce, à divers niveaux de représentation.
- Nous exposons les conséquences de la présence de contraintes dans un système de représentation de connaissances, tant en ce qui concerne les mécanismes d'exploitation que la gestion

de la cohérence des bases de connaissances.

- Nous mettons l'accent sur les liens étroits qu'entretiennent types et contraintes au sein d'un SRPO dynamique et fortement typé tel que Tropes.
- Nous montrons que les contraintes peuvent être utilisées par le modèle lui-même dans l'expression et la maintenance de partages d'informations entre objets, ou dans l'exploration automatique des bases de connaissances.
- Nous étendons la notion de *chemin* – qui permet de désigner un attribut à contraindre – en proposant une gestion adaptée aux attributs multivalués.
- Nous illustrons l'utilité des contraintes pour le SRPO lui-même à travers l'expression et la maintenance de *relations* (dans l'acception la plus générale du terme) entre objets.
- Nous proposons dans Tropes divers usages de la notion de filtre dont l'expression est étendue par les contraintes et dont le mécanisme sous-jacent – le filtrage – exploite la présence de Metéo, le module de types de Tropes.

Initialement, les contraintes manquaient à Tropes pour impliquer des attributs dans des relations. Aujourd'hui, l'utilisateur peut les manipuler dans une base Tropes, mais surtout, les contraintes peuvent être les briques de base de l'expression et du contrôle de nouveaux mécanismes ou extensions du modèle envisagés par ses concepteurs.

En ce qui concerne la programmation par contraintes, les contributions du module Micro conçu pour cette intégration se situent dans les propositions :

- d'une gestion d'unions d'intervalles dans les CSP numériques qui augmentent le niveau de consistance locale maintenu ;
- d'un ensemble de contraintes prédéfinies pour les variables de type ensemble ou liste ;
- d'un ensemble de méta-contraintes qui trouvent leur origine et leur utilité dans le couplage avec un système de représentation de connaissances par objets ;
- d'une gestion de CSP dynamiques à domaines éventuellement infinis ;
- d'un algorithme de résolution mixte, c'est-à-dire capable de traiter des CSP à domaines finis et/ou infinis.

Le module Micro est en cours d'intégration avec la nouvelle version de Tropes. Étant autonome, il peut donc être exploité pour d'autres couplages.

1.6 Le plan du mémoire

Ce mémoire est constitué de trois parties :

- La première partie introduit le contexte et la problématique ; elle comporte quatre chapitres. Le premier chapitre décrit les principes de la représentation de connaissances par objets qui constitue le contexte général de cette étude. Le second chapitre présente le modèle Tropes et introduit la problématique : le besoin d'expression et de maintenance de contraintes dans Tropes. Le troisième chapitre s'intéresse à différents courants de la théorie des problèmes de satisfaction de contraintes qui présentent un intérêt dans la perspective d'intégrer des contraintes dans un système tel que Tropes. Le quatrième et dernier chapitre de cette partie propose un aperçu des systèmes associant contraintes et objets.
- La seconde partie présente l'intégration de contraintes réalisée dans Tropes ; elle est composée de cinq chapitres. Le premier chapitre définit les composants d'un CSP Tropes. Le second chapitre s'intéresse aux liens entre les contraintes et les types gérés par Metéo ainsi qu'à la gestion de CSP dynamiques dans Tropes. Ces deux chapitres établissent un cahier des charges pour la conception d'un module de programmation par contraintes capable de gérer les CSP Tropes. Ce module, appelé Micro, est présenté dans le troisième chapitre. Le quatrième chapitre décrit l'interface qui permet de coupler Tropes à Micro. Enfin, le cinquième et dernier chapitre étudie les conséquences de la présence de contraintes dans Tropes.

- La troisième et dernière partie montre l'utilité des contraintes pour le modèle et fait deux propositions d'extension de notions de représentation – les relations et les filtres – dans TROPES ; elle est constituée de trois chapitres. Le premier chapitre montre que la présence de contraintes peut être exploitée par le modèle TROPES lui-même (ses concepteurs) pour mettre en place un partage de propriétés entre objets composites, un passage de paramètres entre les objets d'un modèle de tâches ou pour la production d'hypothèses dans la perspective d'une exploration des configurations possibles d'une base de connaissances. Le second chapitre est une proposition pour étendre l'expression, la maintenance et la construction de relations entre les objets de TROPES grâce aux contraintes. Le troisième et dernier chapitre propose d'étendre la notion de filtres dans TROPES encore une fois grâce aux contraintes.

Nous terminons ce mémoire en dressant un bilan du travail réalisé et en donnant quelques perspectives de recherche pour la poursuite de l'intégration de contraintes présentée ici.

En annexes de ce mémoire sont fournies une description des règles de maintenance de MICRO (annexe A), une description de l'interface fonctionnelle de MICRO et du couplage TROPES/MC (annexe B), une description des procédures et fonctions utilisées par l'algorithme de résolution proposé (annexe C) et une description formelle de TROPES (annexe D).

Première partie

Objets, contraintes ? Le contexte

Chapitre 2

Représentation de connaissances par objets

Le cadre général de cette étude est un courant de recherche de l'Intelligence Artificielle (IA) appelé Représentation de connaissances Par Objets (RPO). Les Systèmes de Représentation de connaissances Par Objets (SRPO)[1] sont destinés à décrire, organiser et stocker d'importants volumes de connaissances en s'appuyant sur les principes de représentation du paradigme *objet* que l'on retrouve également en génie logiciel ou dans les bases de données : notions de classe, d'instance, de hiérarchie, de spécialisation de classes, d'héritage...Imposant une structuration hiérarchique de la connaissance, ces systèmes proposent divers mécanismes d'exploitation (héritage, méthodes, instanciation, classification...) capables de rendre explicites des faits implicites ou de produire de nouveaux faits à partir de faits établis.

La présente étude concerne plus particulièrement un SRPO appelé TROPES. Mais afin de juger des caractéristiques qui lui sont propres, il convient tout d'abord de présenter les principes de représentation et les mécanismes d'exploitation communs à la plupart des SRPO (*cf.* section 2.1).

2.1 Les systèmes de représentation de connaissances par objets

Les systèmes de représentation de connaissances ou modèles de connaissances cherchent à clairement séparer les connaissances des mécanismes de raisonnement qui vont permettre leur exploitation. En ce sens, ce sont avant tout des langages de modélisation et non pas des langages de programmation. Aussi, il faut distinguer d'emblée, l'interprétation du terme "objet" qui est faite dans le contexte des langages de programmation orientée-objet de celle entendue dans le domaine des représentations de connaissances par objets dans lequel se situe cette étude.

Si des points communs existent entre les interprétations de l'entité "objet" propres à ces deux paradigmes comme, notamment, l'organisation des objets dans une hiérarchie de spécialisation où des objets plus génériques dominent des objets plus spécifiques, les différences fondamentales résident dans leurs finalités respectives : les langages de programmation orientée-objet sont destinés à écrire des programmes, les systèmes de représentation de connaissances à objets (SRPO) sont destinés à supporter des mécanismes d'inférences.

L'une des règles d'or des SRPO est d'être les plus déclaratifs possible, c'est-à-dire d'offrir les moyens d'exprimer des connaissances, sans préjuger de la manière dont elles seront exploitées par les mécanismes de raisonnement mis à la disposition des utilisateurs.

Les réseaux sémantiques [Quillian68] et plus particulièrement l'étude conceptuelle menée par Marvin Minsky sur les "frames" [Minsky75][2] sont le point de départ de nombreux systèmes ou

[1] On les appellera encore Modèle de Connaissances à Objets (MCO).

[2] On trouvera une présentation de ces notions dans [Masini et al.89].

langages dédiés à la représentation de connaissances (KRL [Bobrow et al.77], FRL [Roberts et al.77], SRL [Wright et al.84], SHIRKA [Rechenmann et al.90]...). Des idées énoncées par Minsky, ces systèmes ont d'abord retenu celle du frame (rebaptisé *schéma* ou *objet*) en tant qu'unité structurée de description. Le frame y est employé pour décrire des concepts génériques ou des individus de ces concepts, dont les caractéristiques sont décrites par des slots (rebaptisés attributs). Cependant, ces systèmes peuvent être distingués par :

1. le fait qu'ils considèrent, soit un seul lien *est-un*, soit deux (*sorte-de*, *est-un*) pour organiser et lier les frames,
2. l'*héritage* (partage de propriétés entre frames reliés par les liens de type *sorte-de*) multiple ou simple qu'ils prennent en charge,
3. les droits qu'ils associent à la valeur par défaut (selon qu'elle est admise lors d'un manque d'information comme une connaissance sûre ou hypothétique),
4. la réalisation de l'appariement (par *filtrage* et/ou *par classification*),
5. la distinction qu'ils font entre un frame générique (*classe*) et un frame individuel (*instance*[3]) et qui est liée à l'existence d'un seul ou de deux liens (*cf.* 1).

Cette dernière distinction est à l'origine de deux approches dans les SRPO : l'approche prototypique et l'approche classe/instance. Dans l'approche prototypique, il n'existe qu'un seul type de frame. Un concept est décrit par la donnée d'un frame que l'on peut considérer comme la représentation moyenne (ou prototypique) d'un membre du concept. Tout frame de ce concept (qu'il en représente un individu ou un sous-concept) est engendré à partir de ce frame ou d'un des sous-frames créés jusque là. Tout frame a donc la capacité d'être recopié afin de produire des copies modifiées appelées sous-frames. Par rapport à son sur-frame prototype, un frame hérite les informations qu'il ne redéfinit pas. Les caractéristiques nouvelles qu'il contient viennent enrichir la connaissance sur ce frame (concept ou individu). Les informations qui viennent contredire ou masquer les informations portées par son sur-frame sont acceptées. Au moment de connaître toutes les informations disponibles sur ce frame, on hérite donc des connaissances contenues dans le sur-frame qui ne sont pas modifiées dans le frame.

Les inconvénients de cette approche, vis-à-vis de la cohérence de la structuration de la connaissance, ont été signalés par Ronald Brachman dans [Brachman85]. Il indique que les propriétés exprimées par les slots d'un frame ne peuvent pas être considérées comme des conditions nécessaires d'appartenance au concept représenté, puisque ces propriétés peuvent être remises en cause par un sous-frame. Dès lors, les hiérarchies qui peuvent être établies ne sont pas fiables : les liens frame/sous-frame ne peuvent refléter la réelle appartenance de sous-frames à la même famille d'individus représentée par le sur-frame, ni même l'inclusion structurelle d'un frame-concept dans ses sous-frames concepts – *cf.* le fameux exemple de l'éléphant qui grimpe aux arbres. Si l'approche prototypique se révèle utile dans la représentation des exceptions et la construction de hiérarchies non complètement définies, elle ne favorise pas la mise en place de mécanismes d'inférence comme le filtrage ou la classification.

En règle générale, l'approche classe/instance est préférée à l'approche prototypique dans les langages de représentation de connaissances par objets. Elle consiste à distinguer deux types de frames : les *classes* et les *instances*. Les classes sont les frames qui décrivent des catégories ou concepts ou familles d'individus du monde. Les instances sont les frames qui représentent les individus appartenant aux concepts décrits par les classes. Par opposition à l'approche prototypique où chaque frame peut servir de modèle pour une copie, les classes ont ici un rôle de frames génériques et générateurs alors que les instances sont des frames spécifiques non générateurs.

Dans la suite de cette section, nous ne nous intéressons qu'aux SRPO qui ont choisi une approche classe/instance dans la représentation des connaissances. Il existe deux types de systèmes de représentation de connaissances par objets basés sur une approche classe/instance : les *langages*

[3] Les termes de classe, d'instance, d'héritage, de filtrage, de classification évoqués ici font référence à des mécanismes des SRPO décrits par la suite.

de frames et les *langages hybrides*. Comme les premiers (parmi lesquels KRL, FRL, SRL, Shirka, Rome [Carré et al.88, Carré89] ou Frome [Dekker94]...), les seconds (parmi lesquels Yafool [Ducournau88], Objlog [Dugerdil87], Mering [Ferber84], Airelle [Adam-Nicolle et al.88]...) ont repris et adapté l'idée de représentation et structuration de la connaissance sous forme de frames mais mêlent également des fonctionnements propres à la programmation fonctionnelle, à la programmation orientée-objet, à la programmation logique, aux systèmes déductifs à base de règles... d'où leur nom. Les langages hybrides sont nés d'un souci de fournir à l'utilisateur plusieurs formalismes de représentation de connaissances à l'intérieur d'un même système. En contrepartie, leur emploi peut s'avérer difficile à maîtriser en raison de la diversité des outils qu'ils proposent [Masini et al.89]. Dans la suite, nous ne nous intéresserons qu'à la composante RPO de ces langages.

2.1.1 Principes de représentation

Nous décrivons ici les principes de représentation des SRPO proposant une approche classe/instance de la modélisation de la connaissance.

Si les SRPO distinguent deux types de frames – classe et instance –, toute unité de représentation repose ici sur une structure à trois niveaux frame-attribut-facette [Masini et al.89] (*cf.* figure 2.1). Au niveau du frame, on trouve son nom et un lien (*sorte-de* pour une classe, *est-un* pour une instance). Au niveau des attributs, on trouve les propriétés qui définissent la structure de l'objet. Au niveau des facettes, on trouve la description (typage, valeurs, moyen d'inférence) des propriétés. Le frame est une classe lorsqu'il s'agit de décrire un concept du monde réel, une instance lorsqu'il s'agit de décrire un individu particulier.

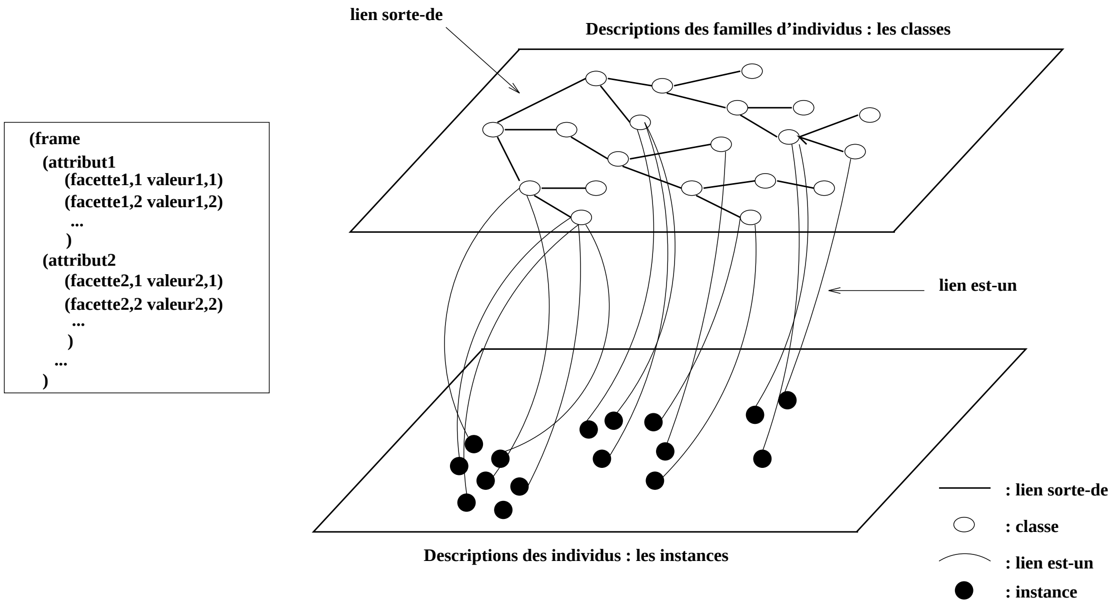


Fig. 2.1 - : À gauche, la structure à trois niveaux d'un frame. À droite, les deux niveaux de description de l'approche classe/instance.

2.1.1.1 La classe

Comme dans les langages de programmation orientée-objet, la classe joue le rôle de moule générateur d'instances en donnant la structure (liste d'attributs) que doit posséder chacune des instances qui lui sont rattachées par le lien d'appartenance *est-un* (*cf.* figure 2.1).

La description d'une classe – aussi appelée son *intension* – est formée de l'ensemble des descriptions des attributs de cette classe. L'intension fournit un ensemble de conditions d'appartenance que chacune des instances de la classe satisfait. L'ensemble des instances de la classe est appelé son *extension.* Les classes sont liées entre elles par le lien *sorte-de* qui décrit la *relation de spécialisation*

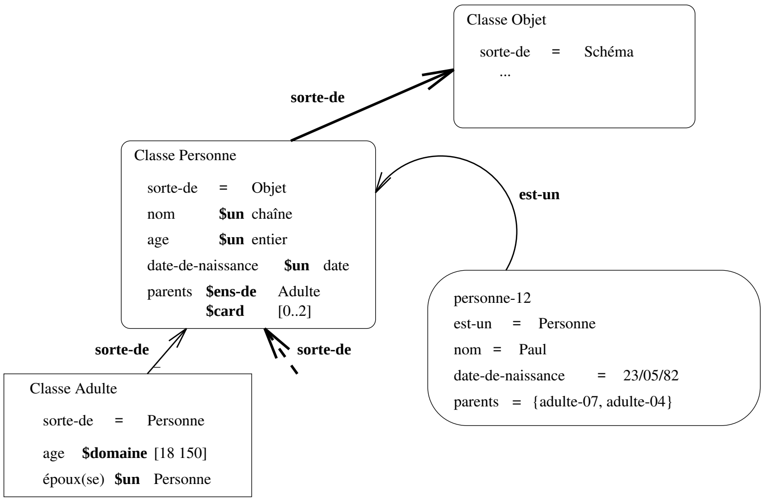


FIG. 2.2 - : La classe Personne et l'instance personne-12. La classe est liée à sa sur-classe directe par le lien *sorte-de*; les attributs des individus qu'elle représente sont décrits par des facettes ($un, $ens-de, $card). La classe peut être spécialisée (Personne en Adulte) par définition de nouveaux attributs (époux(se)) ou restrictions sur les descriptions d'attributs (âge). La classe Objet est la racine de la hiérarchie. L'instance est liée à la classe par le lien *est-un* et contient des valeurs qui satisfont la description de la classe. L'instance peut être liée à d'autres instances (adulte-07, adulte-04) par des attributs modélisant une relation.

entre classes. Si la classe C' est une sorte de classe C alors C' est dite plus spécialisée que C (ou encore sous-classe de C). À l'inverse, C est dite plus générale que C' (ou encore sur-classe de C'). Si C' est une sous-classe de C, alors l'extension de C' est incluse dans l'extension de C. Ceci est dû au fait que l'intension de C' affine celle de C par les descriptions de nouveaux attributs non présents dans C ou/et la complète par des descriptions plus restrictives pour des attributs déjà présents dans C. Dans ce dernier cas, le sous-typage est imposé pour une description $d_{A_{C'}}$ d'un attribut A dans la classe sous-classe C' vis-à-vis de la description d_{A_C} du même attribut A dans sur-classe C. Ainsi, l'ensemble des valeurs décrit par le type de A dans $d_{A_{C'}}$ est inclus dans l'ensemble des valeurs décrit par le type de A dans d_{A_C}. La spécialisation de classes repose donc à la fois sur la définition de nouveaux attributs ou/et sur l'affinement des descriptions des attributs existants. Les classes sont organisées par la relation de spécialisation en une hiérarchie dont la racine est une classe, nommée en général OBJET ou THING, qui définit le lien *est-un* dont ses sous-classes héritent pour leurs instances.

2.1.1.2 L'instance

Une *instance* est rattachée à sa classe par le lien d'appartenance *est-un* (*cf.* l'instance *personne-12* sur la figure 2.2). La structure d'une instance est le record (ou enregistrement) formé à partir de la liste des descriptions des attributs de sa classe d'appartenance. L'instance contient les valeurs des attributs de la classe qui dénotent un individu particulier. Au regard de la description de sa classe d'appartenance, une instance peut être complète (tous les attributs de sa classe d'appartenance ont une valeur) ou incomplète (des attributs de la classe n'ont pas de valeur dans l'instance). Dans tous les cas, les valeurs contenues dans l'instance satisfont les conditions énoncées par les descriptions d'attributs.

À la différence des langages de programmation orientée-objet, le lien d'appartenance n'est pas toujours fixe et certains systèmes (comme SHIRKA ou FROME) autorisent l'instance à migrer vers une autre classe lorsque son contenu le permet. Cette migration peut se faire lors de la création de l'instance ou lors de la modification de son contenu. L'instance migre alors vers la classe la plus spécialisée dont elle satisfait la description. Cette migration est effectuée par un mécanisme appelé *classification* et décrit en 2.1.2.

2.1.1.3 L'attribut

Un *attribut* peut représenter :

- une propriété de l'objet (classe ou instance) qui permet de le décrire indépendamment de tout autre objet. La valeur de cet attribut est d'un type simple (nombre, chaîne de caractères...) ou d'un type défini (date...). L'attribut *âge* sur la figure 2.2 est un exemple de propriété ;
- une relation dans laquelle l'objet peut être impliqué. Cette relation, en général binaire, lie une instance de la classe à une ou plusieurs[4] autre(s) instance(s). L'attribut *parents* sur la figure 2.2 est un exemple de relation ;
- un composant qui traduit le fait que l'objet est un objet composite. L'objet est donc impliqué dans une relation *partie-de* ; il est le tout et ses parties sont des objets d'autres classes.

Cette dernière interprétation de l'attribut n'est présente que dans les systèmes qui gèrent une sémantique particulière pour la relation de composition (comme LOOPS [BOBROW ET AL.83], YAFOOL, OBJLOG...)

Un attribut est dit *monovalué* s'il n'admet qu'une seule valeur. Il est dit *multivalué* si sa valeur est un ensemble ou une liste de valeurs.

2.1.1.4 Les facettes

Les *facettes* permettent la description des attributs dans les classes. Parmi l'éventail proposé par les divers SRPO, on distingue :

- les facettes de typage qui définissent les valeurs possibles pour l'attribut, parmi lesquelles :
  - les facettes de type ($un, $ens-de, $liste-de...) qui donnent le type de l'attribut et indiquent s'il est mono ou multivalué. Selon les fonctionnalités du système, le type d'un attribut peut être prédéfini et simple (entier, chaîne...) ou défini (comme date...) ou bien encore un type construit (Personne...) qui réfère à une classe de la base de connaissances. Les facettes de typage des attributs multivalués donnent le type des éléments des ensembles ou listes ou files. À l'exception de FRL qui ne type pas les attributs, ces facettes de typage sont présentes dans la plupart des SRPO. Par exemple, sur la figure 2.2, l'attribut *nom* dans la classe *Personne* est de type chaîne et monovalué comme l'indique la facette $un ;
  - la facette de restriction de type ($domaine) qui énumère les valeurs que peut prendre l'attribut (OBJLOG, SHIRKA...). Par exemple, sur la figure 2.2, l'attribut *âge* dans la classe *Adulte* est de type entier et monovalué mais son domaine est réduit à l'intervalle [18, 150] comme l'indique la facette $domaine ;
  - la facette de cardinalité ($card) qui donne, pour les attributs multivalués, la cardinalité (resp. la longueur) possible des ensembles (resp. des listes) de valeurs. Cette cardinalité peut être représentée par un entier – elle est unique – ou un ensemble d'entiers – plusieurs cardinalités sont possibles – (OBJLOG, SHIRKA...). Par exemple, sur la figure 2.2, l'attribut *parents* dans la classe *Personne* est de type construit *Adulte* et multivalué mais sa cardinalité varie entre 0 et 2 comme l'indique la facette $card ;
  - la facette d'exception $sauf, qui donne les valeurs qui ne sont pas valides pour l'attribut (SHIRKA). Cette facette offre une facilité d'écriture dans la définition de l'extension du type. Elle est notamment utile pour les attributs multivalués et permet d'exclure des ensembles ou des listes de valeurs du domaine, notamment lorsque ce domaine est infini ou non énumérable ;

[4] Il s'agit alors d'une association (1,N) décomposable en N associations (1,1)

   - la facette de vérification $à-vérifier, qui associe un prédicat à l'attribut que toute valeur proposée devra satisfaire (SHIRKA, FRL, KRL, OBJLOG, YAFOOL...). Elle exprime une propriété des valeurs du type qui ne peut être exprimée au moyen des autres facettes (par exemple, la parité d'une valeur numérique);
- les facettes d'inférence qui permettent de calculer la valeur de l'attribut dans une instance (nous les décrivons à la section 2.1.2);
- les facettes réflexes qui permettent d'associer un pré ou un post-traitement à une tentative d'obtention de la valeur d'un attribut, mais aussi lors l'ajout, de la modification ou de la suppression d'une valeur.

Les facettes de réaction à un événement ou réflexes pourraient être absorbées par les fonctionnalités générales du système (c'est-à-dire par les opérations de suppression, de modification, d'ajout) ou même dans les méthodes de calcul. Cependant, elles permettent d'associer une réaction particulière à un attribut spécifique.

Il faut souligner qu'en toute logique[5], les facettes d'inférence, comme les réflexes, ne devraient être utilisées que lorsque l'appartenance de l'instance est acquise [Rechenmann92].

Les facettes de typage et de restriction de type doivent être consultées et vérifiées lors de la proposition d'une valeur pour l'attribut (donnée par l'utilisateur ou par une méthode): elles et elles seules décident donc de la validité de l'appartenance de l'instance à la classe.

Des langages comme OBJLOG+ [Faucher91] ou PTILOO [Ferber88] considèrent les attributs comme des classes et offrent la possibilité d'étendre le nombre et la spécificité des facettes. Cependant, disposer d'un nombre important de facettes peut entraîner des redondances dans l'expression des types (un même type pourra être décrit par plusieurs expressions). Dans ce cas, lorsqu'il s'agit de comparer des types entre eux, pour valider la définition d'une classe, par exemple, une étape de normalisation peut être nécessaire pour obtenir une expression standardisée du type à partir de laquelle un test uniforme de sous-typage peut alors être appliqué.

D'autres langages comme KEE [Fikes et al.85] et AIRELLE optent pour une expression plus compacte du type. L'expression du type n'est plus dispersée dans de multiples facettes mais dans la combinaison de constructeurs ou combinateurs. Si l'expression du type a le mérite de la compacité, elle n'a pas pour autant celui de l'unicité, la diversité des combinateurs et des constructeurs peut, ici encore, entraîner des redondances.

Enfin, au regard des facettes de définition du type d'un attribut multivalué, le descripteur de type *set specification* de KRL, offre un puissant moyen d'élaborer un ensemble, une séquence ou une liste d'objets (à partir des descripteurs SetOf, ListOf, Sequence). Un objet peut être décrit dans les termes de son appartenance à un ensemble (descripteur In), et un ensemble peut être décrit en termes des objets qu'il contient ou ne contient pas (descripteurs Item, AllItems, NotItems).

En règle générale, la puissance de description d'un SRPO est donc exprimée par l'arsenal de facettes dont il dispose. Nous verrons cependant (*cf.* section 3.5) que si l'ensemble de facettes associé à un attribut permet d'en décrire plus ou moins le type et le domaine de valeurs, les facettes sont mal adaptées à l'expression de relations entre attributs.

2.1.1.5 La spécialisation et l'héritage

Dans une base de connaissances, les différents concepts du monde réel modélisés sont représentés par des hiérarchies de classes établies sur la relation de spécialisation. Cette relation constitue un ordre partiel (elle est réflexive, antisymétrique et transitive). Le lien *sorte-de* détermine la ou les classes immédiatement plus générales que la classe courante: ses sur-classes directes. D'un point de vue ensembliste, la sous-classe contient un ensemble d'instances inclus (en général, strictement) dans l'ensemble d'instances décrit par sa sur-classe directe.

La *spécialisation* est basée sur la définition de nouveaux attributs ou/et sur l'affinement des

[5] Ce principe n'est pas toujours respecté par les systèmes lors de la classification (*cf.* 2.1.2).

attributs présents dans les sur-classes. Cet affinement correspond à un sous-typage : le domaine de l'attribut doit être un sous-ensemble du domaine donné par sa dernière définition, c'est-à-dire la première rencontrée en remontant le graphe de spécialisation.

Parce que la relation de spécialisation assure que toute propriété dans une classe est présente (affinée ou non) dans ses sous-classes, tout individu décrit dans une classe est également individu des sur-classes atteignables depuis cette classe en remontant le lien *sorte-de* jusqu'à la racine de la hiérarchie.

L'*héritage* est un mécanisme qui exploite la relation de spécialisation entre les classes. Il permet la factorisation des connaissances en se chargeant de parcourir la hiérarchie de classes à la recherche d'une information.

Il est notamment utilisé lors de la vérification de la spécialisation. Lorsqu'une définition est proposée pour un attribut, le mécanisme d'héritage remonte la hiérarchie afin de trouver la dernière définition donnée pour l'attribut. Si une telle définition n'existe pas, c'est qu'il s'agit de la première définition de l'attribut. S'il existe une telle définition, le système s'assure que la nouvelle définition correspond à un affinement (*cf.* figure 2.3).

L'héritage est également sollicité lors de la vérification de la valeur d'un attribut pour une instance. S'il n'existe pas de définition de l'attribut pour la classe, alors la hiérarchie de classes est parcourue vers le haut afin de trouver la dernière définition donnée pour l'attribut. La valeur sera confrontée à cette définition.

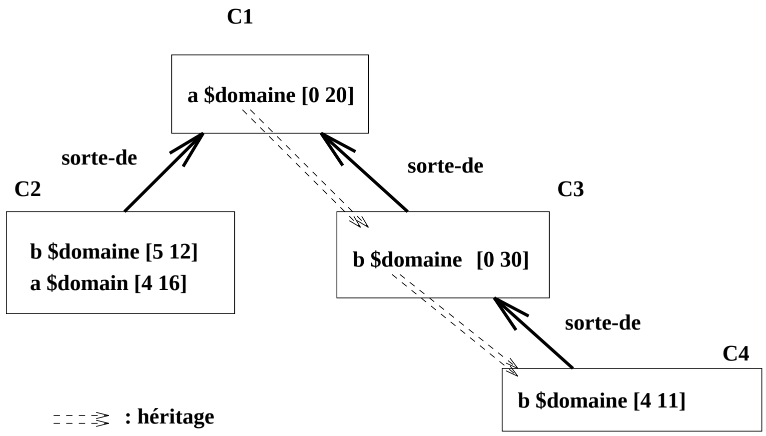


FIG. 2.3 - : Les classes C3 et C4 héritent de la définition de l'attribut a de C1. C4 n'hérite pas b de C3, puisqu'elle le redéfinit. Le sous-typage garantit la spécialisation.

Les descriptions des attributs qui ne sont pas modifiées ou nouvelles ne sont pas, en général, héritées statiquement des sur-classes au moment de la définition de la classe. Elles seront héritées dynamiquement en cas de besoin : lors de la création d'une instance, pour connaître la structure de la classe – donc de l'instance –, ou lors de la validation de la valeur d'un attribut, pour connaître la définition à laquelle il faut se référer.

Lorsque l'héritage des propriétés relève d'un simple parcours ascendant et linéaire dans une hiérarchie de classes dans laquelle toute classe (racine exceptée) a, au plus, une sur-classe directe, on parle d'*héritage simple*. Lorsque le lien *sorte-de* autorise plusieurs sur-classes directes pour une classe, on parle d'*héritage multiple*. Rester alors fidèle au principe d'héritage des propriétés s'avère problématique lorsqu'une caractéristique (non redéfinie par la classe) est présente dans au moins deux des sur-classes (parce qu'elle y est, soit définie, soit héritée).

Les règles les plus générales pour lever les conflits sont les suivantes :

- Lorsqu'une caractéristique n'est redéfinie que dans une seule classe directe et héritée dans les autres, c'est la description de cette sur-classe qui est héritée.
- Lorsqu'une caractéristique est redéfinie dans plusieurs sur-classes directes, le conflit peut-être

résolu par diverses techniques basées sur un parcours en profondeur d'abord ([Ducournau et al.89]) ou en largeur d'abord (MERING) du graphe des sur-classes. Mais il demeure des cas de hiérarchies (*cf.* [Masini et al.89] p.57) pour lesquels l'ordre établi entre les sur-classes par ces parcours n'est pas satisfaisant.

Une autre approche consiste à éviter les conflits de noms en préfixant les propriétés du chemin allant de leur classe de définition à la sur-classe. Ainsi, toutes les caractéristiques sont héritées mais distinguées par leur classe d'origine (ROME, SHOOD [Rieu et al.92]). Si l'héritage multiple permet le partage d'informations de sources diverses et évite la définition de caractéristiques identiques, l'héritage simple lui est souvent préféré, qui oblige le concepteur de la hiérarchie à définir avec précision la sémantique de chaque caractéristique. L'héritage simple est donc très restrictif et insuffisant dans l'expression de caractéristiques communes à plusieurs classes. La notion de *point de vue* (*cf.* section 2.1.1.8) est un moyen d'étendre l'héritage simple.

2.1.1.6 Les objets composites

La relation de spécialisation entre classes rend compte d'une structuration des connaissances. La relation d'appartenance d'une instance à une classe rend compte de la structure de l'instance. Ces deux relations sont à la base de l'approche classe/instance et sont des relations dont la sémantique est intégrée dans les fonctionnalités d'un système de représentation de connaissances se réclamant de cette approche. À ces deux relations prédéfinies, certains systèmes en ajoutent éventuellement une troisième : la relation de *composition*.

Un objet est dit *complexe* lorsqu'au moins un de ses attributs a pour valeur un (ou plusieurs) autre(s) objet(s) de la base de connaissances. Le type de cet attribut est le type d'une classe de la base de connaissances. Un objet complexe est dit *composite* lorsqu'il est l'agrégation d'autres objets – appelés *composants* – qui forment ses parties (*cf.* figure 2.4).

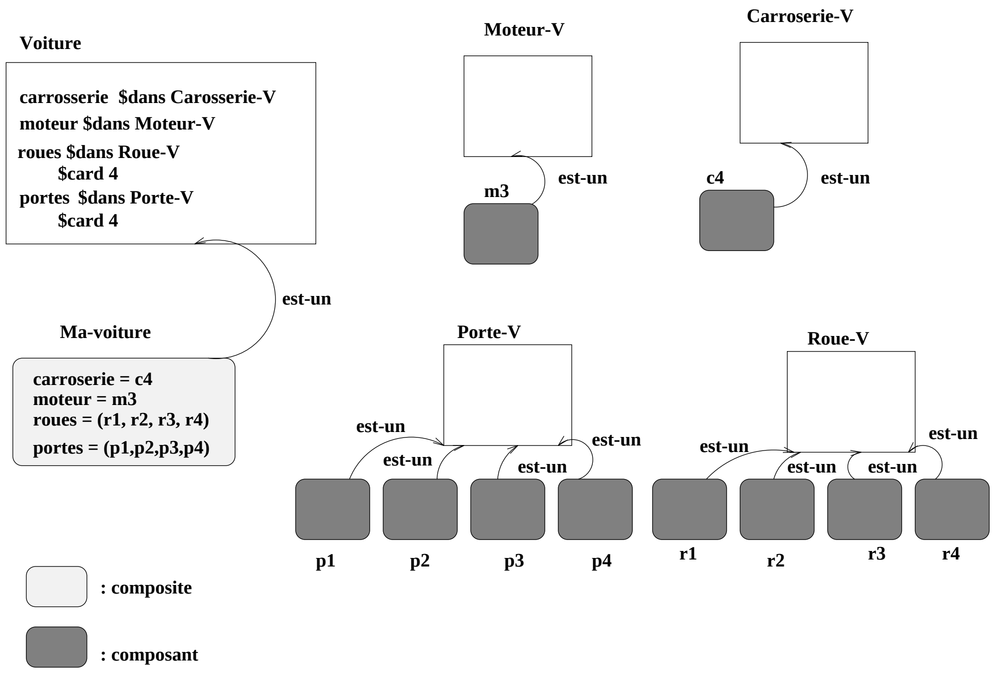


FIG. 2.4 - : Une voiture est un objet composite formé des composants carrosserie, moteur, roues et portes.

Les composants d'un objet composite sont liés à celui-ci par la relation de composition, aussi appelée *partie-de*. Six types distincts de relation de composition *partie-de* ont été proposés par Winston [Winston et al.87] en jugeant de la fonctionnalité (le composant a-t-il une fonction au sein du composant qui définit sa situation ?), de l'hétérogénéité (le composant est-il du même type que le composite?) et de la séparabilité (le composant peut-il être considéré sans son composite?) des composants par rapport à leur composite (*cf.* table 2.1).

		Propriétés des éléments		
Relation	Exemple	Fonctionnalité	Homogénéité	Séparabilité
composant/tout	roue/voiture	+	-	+
membre/collection	arbre/forêt	-	-	+
portion/masse	grain/sel	-	+	+
matière/objet	fer/clef	-	-	-
trait/activité	payer/faire des courses	+	-	-
lieu/région	oasis/désert	-	+	-

TAB. 2.1 - : Les six types de relation de composition (d'après [Winston et al.87]).

Le plus souvent dans les systèmes de représentation qui prennent en charge la composition, celle-ci est interprétée comme la première relation : *composant/tout*.

Quelque soit la catégorie dans laquelle on classe une relation *partie-de* particulière, elle demeure une relation :

- non réflexive (une instance ne peut-être partie d'elle-même),
- anti-symétrique (si O_2 est partie de O_1 alors O_1 ne peut être partie de O_2), et
- transitive (si O_3 est partie de O_2 et si O_2 est partie d'un objet O_1, alors O_3 est partie de O_1 : O_1 est composite, O_3 est composant, O_2 est composite de O_3 et composant de O_1).

Pourtant, la transitivité n'est valable que lorsque l'on reste à l'intérieur d'un type spécifique de relation *partie-de*. Ainsi, si l'on combine deux types de composition, l'inférence attendue de la transitivité n'est pas valide.

Par exemple, la main de Pierre est une partie (composant) de Pierre, Pierre est une partie (un membre) de l'équipe de basket-ball, mais la main de Pierre n'est ni un composant, ni un membre de l'équipe de basket-ball !

Compartimentée à l'intérieur d'un même type de relation de composition, la transitivité permet de considérer des objets composites du point de vue de la récursivité : un objet composite peut avoir des composants qui sont eux-mêmes composites dont les composants sont eux-mêmes composants...jusqu'à parvenir à un composant qui n'est plus composite.

Les concepteurs du système ORION [Kim et al.89] ont été les premiers à recenser les diverses propriétés qui peuvent être attachées au lien de composition :

- Un lien de composition entre un objet composite O et son composant O' peut être *exclusif* : O' est composant de l'objet composite O seulement (personne/main).
- Un lien de composition entre un objet composite O et son composant O' peut être *partagé* : O' peut être composant d'autres composites que O (pays/rivière).
- Un lien de composition entre un objet composite O et son composant O' peut être *dépendant* : l'existence de O' est liée à celle de O. Si O disparaît, O' aussi (personne/os).
- Un lien de composition entre un objet composite O et son composant O' peut être *indépendant* : l'existence de O' n'est pas liée à celle de O. Si O disparaît, O' lui survit (équipe/joueur).

La combinaison de ces propriétés permet de considérer quatre types de liens de composition (dépendant-exclusif, dépendant-partagé, indépendant-exclusif, indépendant-partagé). À chacun de ces types est associé un comportement particulier. Le système SHOOD [Escamilla et al.90] a repris ces idées et envisage ces différentes sémantiques du lien de composition.

Parmi les systèmes de représentation par objets qui proposent un traitement de la relation de composition, on distingue les systèmes qui traitent spécifiquement la relation de composition en lui associant une sémantique fixe et prédéfinie (comme LOOPS, YAFOOL, ORION [Kim et al.89], OBJLOG...) et les systèmes (comme SRL, SHOOD, OTHELO [Fornarino et al.90b], DBMS [Rumbaugh87]) qui traitent indifféremment toute sorte de relation (en général, binaire) en proposant des outils de construction de relation, en particulier pour la relation de composition.

Nous nous intéresserons également au chapitre 11 au partage de propriétés entre un objet composite et ses composants.

2.1.1.7 Les relations

Outre les relations de spécialisation, d'appartenance et de composition, certains SRPO se sont intéressés à l'expression et à la maintenance de relations (au sens général du terme) entre les objets.

La représentation d'une relation (en général, binaire) entre les instances d'une même classe ou entre les instances de deux classes distinctes peut être abordée de deux manières.

La première consiste à représenter chaque relation dans laquelle est impliquée la classe par un attribut. Cet attribut est appelé *lien* de la relation. Dans chaque instance, sa valeur indique la ou les instances avec la ou lesquelles l'instance est en relation. La classe qui est le co-domaine de la relation est désignée par le type de l'attribut-lien. La représentation de la relation inverse peut alors se faire aussi par la définition d'un attribut-lien dans la classe co-domaine. De sorte que, lorsqu'un attribut-lien dans une instance I reçoit une instance J pour valeur, si son lien inverse est défini dans J, il doit prendre pour valeur I afin d'assurer la cohérence de la relation.

Dans YAFOOL où deux objets sont liés l'un à l'autre par deux liens inverses, un attribut-lien possède une facette *$inverse* qui renseigne sur le nom de l'attribut-lien inverse dans la classe-co-domaine (*cf.* figure 2.5). La propagation de l'information est assurée par des réflexes prédéfinis qui répercutent tout ajout ou retrait de valeur d'un attribut-lien auprès de son attribut-lien-inverse. Le même principe de description d'un lien inverse par une facette est repris dans OBJLOG+. Lorsque un lien est utilisé par plusieurs relations, la facette $inverse indique plusieurs liens inverses. Cependant,

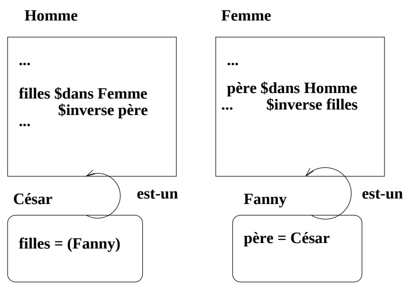


FIG. 2.5 - : La relation entre un père et ses filles est représentée par les deux attributs ou liens inverses père et filles (représentation à la YAFOOL).

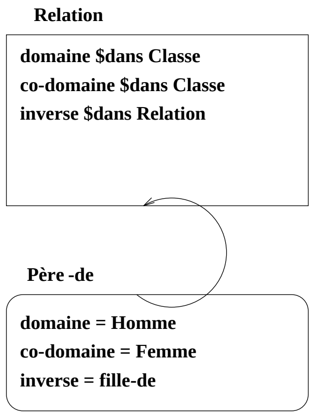


FIG. 2.6 - : Le lien père-de est représenté par un objet (instance) d'une classe relation (représentation à la SRL).

en représentant une relation à l'aide d'un attribut-lien (et de son inverse), on procède à une duplication de l'information : les classes domaine et co-domaine contiennent chacune une description de la relation. Ces deux descriptions symétriques peuvent paraître redondantes. Aussi, de nombreux systèmes (SRL, SHOOD, VIEWS [Davis87], OTHELO, DBMS) ont choisi une seconde approche en représentant toute relation par un objet (*cf.* figure 2.6). Cet objet porte les informations propres à la relation, la description des liens inverses qui la composent, et les méthodes visant à maintenir la cohérence de la relation.

SRL [Fox et al.86, Wright et al.84] reprend les idées de la plupart des autres systèmes de représentation mais son originalité réside dans la possibilité offerte à l'utilisateur de définir ses propres relations, depuis la description de l'héritage (en décidant quels attributs et quelles valeurs seront hérités), jusqu'à l'élaboration de la structure des attributs entre les schémas et l'appariement arbitraire des attributs et des valeurs entre schémas. Dans SRL, tout attribut est une relation. Une relation est décrite par une classe prédéfinie. La sémantique de l'héritage associé à la relation peut être établie à travers cinq spécifications (attributs du schéma relation) :

- l'inclusion indique les attributs et les valeurs à hériter directement,
- l'exclusion indique les attributs et les valeurs qui ne sont pas hérités,
- l'élaboration indique pour les attributs quelle est leur structure,
- l'application d'une fonction à un attribut ou une valeur,
- l'introduction indique les attributs et valeurs à introduire dans un concept lorsque la relation est créée.

L'inverse d'une relation est définie (par l'utilisateur ou automatiquement par le système) afin d'assurer la réciprocité des informations.

James Rumbaugh [Rumbaugh87] a également plaidé en faveur de la représentation de relations par des objets en avançant que la sémantique d'une relation, lorsque la relation est simplement décrite par des attributs, n'est pas capturée dans une structure sémantique mais dispersée dans les classes que la relation relie. Il propose de faire d'une relation l'instance d'une classe générale prédéfinie *Relation*. Les relations sont exprimées entre des objets et non entre des attributs d'objets. La classe *Relation* comporte des attributs destinés à donner les caractéristiques (le nom, les champs et des contraintes de cardinalité sur chacun des champs) de la relation. Les méthodes associées à cette classe permettent d'ajouter ou de supprimer un élément à une relation, d'indexer un champ, de tester l'appartenance d'un n-uplet à une relation et de visionner tous les éléments d'une relation. Le système résultant de ces principes, DSM (Data Structure Manager), est un modèle objet-relation qui combine le modèle orienté-objet au modèle entité-relation des bases de données. Un des inconvénients de faire d'une relation, non pas une classe mais une instance de classe, est qu'il n'existe qu'une relation désignable par le même nom. Ainsi, doit-on choisir si la relation *maître* fait référence à la relation *maître/élève* ou à la relation *maître/chien* [Dekker94].

Dans SHOOD, on distingue les relations verticales et les relation horizontales. Les premières sont la spécialisation et la disjonction (par ce lien, il est imposé que les deux classes reliées aient des ensembles d'instances disjoints), les secondes (parmi lesquelles la composition) sont toutes les autres relations qui lient des instances de classes. Il est possible d'instaurer et de combiner cinq types de dépendances – proches des dépendances d'ORION – entre les objets. Lorsqu'un attribut d'une instance, appelée *receveur*, est défini en dépendance exclusive avec ses *donneurs* (les instances qui sont liées à elle par l'attribut), alors ces instances ne peuvent être valeurs que d'attributs à dépendance nulle. Si la dépendance est existentielle, les donneurs sont supprimés lorsque le receveur est supprimé, sauf si ces donneurs sont receveurs en dépendance existentielle d'autre donneurs. Si la dépendance est partagée alors tout manipulateur de l'instance peut modifier le comportement des attributs diffusés. Si la dépendance est nulle – ce qui est le cas par défaut –, le receveur ne peut pas manipuler les donneurs. Enfin, la dépendance spécifique vise à modifier la dépendance d'un attribut dans une instance et peut-être vue comme une exception.

La diffusion de valeurs permet d'*hériter* des valeurs dans les deux sens entre receveur et donneurs. L'accès aux informations à diffuser se fait par un chemin qui peut comporter des indirections successives. Un attribut diffusé ou importé dans une instance y est vu comme une constante et ne peut être modifié sauf si une dépendance le permet.

Un autre aspect intéressant de SHOOD est la gestion de propriétés (comme la réflexivité, l'anti-réflexivité, la symétrie ...) et d'opérations (l'union, l'intersection, la réciproque, la composition) sur les relations. Si bien que liens sémantiques – la description de l'attribut (descripteurs + dépendance) – et propriétés mathématiques forment un tout indissociable qui définit entièrement la relation.

Dans VIEWS [Davis87], les relations lient entre elles les parties (assimilables aux composants) d'une vue (assimilable au composite). VIEWS ne possède pas les relations prédéfinies *sorte-de* et *est-un*, mais leur sémantique pourra être obtenue. Les relations d'une vue sont à rapprocher des attributs d'un objet. Les relations peuvent avoir diverses sémantiques (*partie-de*, *sous-classe*, *à-gauche-de*...) et se réduisent à des relations binaires indépendantes (même interprétation que pour les attributs multivalués). Les relations d'héritage ont un caractère actif : tout changement sur la partie source d'une relation est répercuté sur la partie cible d'après les règles d'inférence qui donnent la sémantique de l'héritage à réaliser. La définition d'une relation indique les parties qui sont reliées, quelles parties sont transférées de la source à la cible, et si la relation est transitive. Des contraintes dans VIEWS limitent les valeurs possibles des parties et des relations d'une vue et renforce la sémantique de la vue (contraindre la partie *Clyde* (instance) d'une vue *Cirque* à être partie (instance) d'une vue *Éléphants*, ou la cardinalité d'une partie *pattes* d'une vue *chien* à valoir 4). Il est également possible de combiner des contraintes avec des opérateurs logiques (ou, et, not). Une contrainte est composée d'une liste de composants, d'un prédicat qui doit être satisfait et d'un mode échec appelé lorsque le prédicat n'est pas satisfait. Le prédicat et le mode échec sont spécifiés par des procédures LISP. Les contraintes sont avant tout des prédicats à vérifier et non des règles d'inférence (qui n'apparaissent que dans le mode échec). Le partage de propriétés entre composite et composants se ferait en VIEWS dans la description de la sémantique d'une relation de composition spécifique à ce composite.

L'implantation des relations dans OTHELO [Fornarino et al.90a] est née de l'idée d'utiliser l'approche objet pour décrire un langage de relations dites *de dépendance* qui sont des relations orientées entre un objet appelé *maître* et un objet appelé *esclave*. La gestion de la cohérence de ce type de relation implique que toute modification intervenant sur un objet maître doit être répercutée, selon la sémantique de la relation, sur l'objet esclave. Les relations sont représentées par des classes ; une instance de ces classes est une relation individuelle (un élément). La classe *lien* de OTHELO est la sur-classe de toutes les classes de relations. Sa méta-classe *méta-lien* définit les méthodes de création et de destruction des relations individuelles. La classe *lien* transmet à ses sous-classes les attributs *objet-influant* et *objet-dépendant* et contient les méthodes *cohérence* et *activer*. Pour chaque classe lien, une liste de points d'activation est spécifiée. Un point d'activation est constitué de sélecteurs qui correspondent aux actions (méthodes) qui nécessitent le rétablissement de la cohérence du lien et d'une procédure de mise à jour. La méthode *cohérence* définit la sémantique de la relation et s'exécute à la création d'un lien individuel (instance). La méthode *activer* est appelée lors de la modification d'un objet qui joue le rôle d'objet influant dans un ou plusieurs liens individuels. Les points d'activation du lien sont alors passés en revue, jusqu'à trouver ou non celui dont un sélecteur *correspond* à la modification, puis l'action associée à ce point d'activation est exécutée. [Fornarino91] fournit plusieurs exemples de classes de liens :

- les liens conditionnels : lors de modifications de l'objet influant, la cohérence de ces liens n'est rétablie que sous certaines conditions ;
- les liens sur propriétés qui ne concernent que certaines propriétés des objets mis en relation ;
- les liens entre liens : l'établissement de relation entre liens permet de déduire de nouvelles relation ou de détecter des incohérences (inverse-de, incompatible) ;
- les liens à influants et/ou dépendants multiples permettent d'exprimer la composition, ou la relation de cause à effet ;
- les liens à automates dont les activations font changer d'état un automate dont certains états appellent la modification des objets dépendants ;
- les liens d'héritage (lien de spécialisation, d'instanciation, de partie, de généralisation) sont des liens propriétés particuliers qui assurent l'héritage de propriétés (vertical ou sélectif).

Il existe donc des formes variées de l'expression et de la maintenance des relations. En général, les SRPO leur accorde un traitement particulier "codé en dur". Nous illustrerons au chapitre 12 comment la présence de contraintes au sein d'un SRPO peut également contribuer à l'expression et à la maintenance de relations.

2.1.1.8 La notion de point de vue

En représentation des connaissances, la notion de *point de vue* ou *perspective* réfère à la capacité d'un système de modéliser une même entité selon des points de vue différents, reflétant les diverses perceptions que divers concepteurs ou utilisateurs ont de cette entité. Ainsi une personne employée et sportive peut être considérée d'un point de vue professionnel et d'un point de vue sportif. Un utilisateur seulement intéressé par les informations du point de vue sportif recueillies sur cette personne, doit pouvoir n'accéder qu'à elles seules. Rares[6] sont les systèmes de représentation de connaissances qui offrent une telle fonctionnalité qui sous-tend une partition de la vision d'une instance, la description des points de vue, la détermination des attributs propres à un point de vue, ou encore l'interrogation d'une instance depuis un certain point de vue.

2.1.2 Mécanismes d'exploitation

Une fois représentés à l'aide des structures descriptives, les faits d'une base de connaissances (classe, instance, valeurs) peuvent alors être soumis à divers mécanismes d'exploitation qui étendent et/ou rendent explicite la connaissance. Ces mécanismes d'exploitation illustrent la sémantique du modèle et se plient aux diverses exigences des principes de représentation choisis par le SRPO. Ils sont également le reflet des capacités d'inférence du modèle. Les plus fréquemment rencontrés sont l'instanciation (sans elle, rien ne serait possible) et l'attachement procédural. Le filtrage et la classification, de plus en plus fréquents, n'apparaissaient pas dans les premiers SRPO.

2.1.2.1 L'instanciation

L'instanciation est l'action qui consiste à créer une instance. Dans l'approche classe/instance classique, une instance n'existe qu'en tant qu'individu d'une classe à laquelle elle est reliée par le lien *est-un*. Aussi, à l'instar des langages de programmation orientée-objet, la création d'une instance s'effectue en indiquant sa classe d'appartenance. Toutefois, à la différence des langages de programmation orientée-objet, l'instanciation n'est pas une méthode propre à la classe, mais un mécanisme général du système. Le lien *est-un* modélise la relation d'appartenance dans la vision ensembliste d'une classe.

L'instanciation nécessite donc de fournir, d'un côté la classe d'appartenance de l'instance, de l'autre les informations destinées à déterminer cet individu et à le distinguer des autres instances de la classe. Afin que l'instanciation soit validée, c'est-à-dire que l'instance figure bien dans la base comme individu de cette classe, il faut que les informations fournies satisfassent les conditions d'appartenance.

Outre un éventuel nom pour l'instance, les informations attendues lors de l'instanciation sont les valeurs des attributs de l'instance. Pour chaque valeur d'attribut fournie, une vérification de type doit être effectuée qui teste l'appartenance de cette valeur à l'ensemble de valeurs possibles défini par la description de l'attribut dans la classe. Si la valeur est acceptée, le mécanisme de validation de l'appartenance de l'instance à la classe se poursuit en considérant la prochaine valeur à tester. Lorsque toutes les valeurs fournies ont passé le test de validation avec succès, l'instance est intégrée dans la base de connaissances, le lien d'appartenance *est-un* de l'instance désigne la classe proposée (*cf.* figure 2.7).

Des valeurs peuvent ne pas être fournies pour certains attributs de l'instance, auquel cas l'instance est dite *incomplète*. Lorsqu'aucune valeur n'est fournie, le système peut ou non valider la création de l'instance vide (c'est possible dans SHIRKA). Lorsqu'une des valeurs ne satisfait pas la description de l'attribut correspondant, la création est en général refusée, bien qu'il soit possible de tenter alors de rattacher l'instance à une autre classe dans la hiérarchie. Au regard de l'instanciation, la classe décrit les conditions nécessaires imposées à l'instance pour lui appartenir. Si l'instance ne vérifie pas ces conditions, elle n'appartient pas à la classe et le lien d'appartenance proposé ne peut être établi : l'instanciation échoue. Par contre, si l'instance vérifie les conditions données par la description de la classe, elle y est effectivement rattachée par la demande d'instanciation.

[6] Une présentation détaillée de la notion de perspective dans KRL, LOOPS, ROME, VIEWS est faite par Olga Mariño dans [Mariño93].

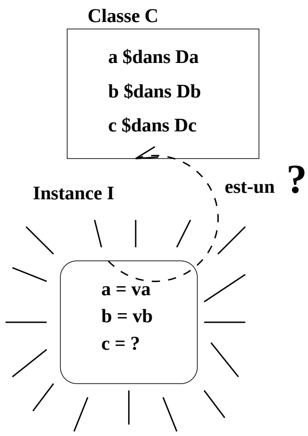


FIG. 2.7 - : Instanciation : l'instance I est créée pour être rattachée à la classe C. Pour valider ce lien d'appartenance, il faut s'assurer que va $\in$ Da, vb $\in$ Db, vc $\in$ Dc. Si une des valeurs d'attributs est absente, elle n'influe pas sur la validation mais l'instance est incomplète.

2.1.2.2 Les inférences de valeur d'attribut

Une fois le lien d'appartenance validé, si l'instance est incomplète, le système peut tenter de déterminer les valeurs manquantes des attributs dans la mesure où ces attributs possèdent des facettes indiquant un moyen d'obtenir cette valeur. La classe est alors sollicitée comme modèle de construction d'instance. Afin de compléter les valeurs d'attributs manquantes, les mécanismes d'inférence de valeur d'attributs dont dispose le système sont alors déclenchés. Ainsi, si l'attribut possède une facette :

- $valeur, alors la valeur donnée par cette facette est une constante de la classe[7] et devient également la valeur de l'attribut dans toute instance rattachée à la classe.
- $sib-filtre, alors la facette décrit un ensemble de conditions portant sur une ou plusieurs classes. Parmi les instances de ces classes, le filtre ne retient que celles qui satisfont ces conditions. L'ensemble des instances obtenues devient valeur de l'attribut. SHIRKA, FROME, OBJLOG sont parmi les rares systèmes proposant ce mécanisme à travers l'expression d'une facette.
- $si-besoin, alors cette facette indique une méthode qui peut être exécutée pour tenter de déterminer la valeur de l'attribut dans l'instance. Cette facette permet ce qu'on appelle un *attachement procédural*. L'activation de cette méthode n'est possible que si les valeurs des paramètres (des constantes mais aussi d'autres valeurs d'attributs) sont présentes. Si tel est le cas et si la méthode retourne un résultat, il doit appartenir à l'ensemble des valeurs possibles, afin d'être définitivement entériné comme valeur de l'attribut dans l'instance.
- $défaut, alors la valeur donnée par cette facette est une valeur non pas constante (si elle ne souffrait pas d'exception, la valeur serait donnée par la facette $valeur), mais à affecter à l'attribut lorsque sa valeur ne peut être obtenue ni de l'utilisateur, ni de quelque autre moyen d'inférence en l'état des connaissances.

Pour les systèmes comportant toutes ces facettes (comme SHIRKA), la facette $valeur a priorité sur toutes les autres facettes ($si-besoin, $sib-filtre, $défaut). Entre ces dernières, la priorité est donnée à la facette apparaissant dans la dernière définition de l'attribut dans la hiérarchie.

2.1.2.3 Le filtrage

Dans le contexte des représentations de connaissances par objets, le filtrage est un mécanisme de recherche d'un ensemble d'objets qui satisfont des conditions énoncées dans un modèle ou filtre.

[7] En ce sens, on ne peut parler effectivement d'inférence.

Il est inspiré des requêtes par fragments des réseaux sémantiques [Quillian68] et de l'appariement proposé pour les frames par Minsky. Le processus de base du filtrage, appelé *appariement* ou *pattern-matching*, consiste en une comparaison entre un objet de la base de connaissances et la description d'un objet proposée par le filtre et qui sera retenu. Trois possibilités s'offrent à l'issue de cet appariement :

- l'objet satisfait toutes les conditions du modèle. Il fait partie du résultat du filtrage ;
- l'objet ne satisfait pas les conditions du modèle. Il ne fait pas partie du résultat du filtrage ;
- l'objet satisfait partiellement les conditions du modèle : l'information qu'il contient satisfait toutes les conditions du modèle mais l'objet manque d'informations et la satisfaction de certaines conditions du filtre ne peut être vérifiée. L'objet fera partie du résultat du filtrage seulement si un filtrage partiel est autorisé.

Lorsque le filtrage effectué peut n'être que partiel, on peut assimiler le filtrage à un prédicat à un argument (l'objet à apparier) de la logique à trois valeurs : vrai, faux, inconnu [Napoli92] pour laquelle l'introduction de la valeur *inconnu* repose sur le manque d'informations pour statuer sur l'objet filtré. La prise en compte d'un filtrage partiel nécessite d'établir un raisonnement défaisable : toute déduction faite en considérant l'objet incomplet comme résultat du filtrage devra être annulée s'il advient que l'objet, une fois complété, ne satisfait plus le filtre.

Dans les systèmes de représentation de connaissances, le filtrage peut être introduit au moyen de requêtes (KRL, FRL, YAFOOL) dont la réponse est la liste des objets satisfaisant le filtre mais dont l'effet peut aussi être la création et le positionnement dans la hiérarchie de la classe d'objets correspondant au filtre (FROME, OBJLOG+). Ces requêtes sont des fonctionnalités, des commandes du langage. Mais le filtrage peut également être introduit à l'aide d'une facette : le résultat du filtrage (un ensemble d'objets) devient alors la valeur de l'attribut multivalué possédant cette facette (MERING, FROME, SHIRKA) [8].

```
{ homme
      sorte-de        =             personne ;
      lui-meme     $var-nom      lui ;
      a-pour-pere     $un             homme
      petits-fils     $liste-de     homme
                      $com            "les petits-fils d'un homme sont les"
                                      "hommes qui ont pour pere les "
                                      "hommes qui ont pour pere l'homme "
                                      "dont il est question "
                      $sib-filtre
                          { homme
                              lui-meme              $var->       petits-fils ;
                               a-pour-pere              $var-nom    pere
                                                        $sib-filtre
                                                           { homme  lui-meme        $var-> pere ;
                                                                    a-pour-pere   $var<- lui }}
```

FIG. 2.8 - : Expression d'une filtre en SHIRKA. Le filtre imbriqué qui suit la facette *$sib-filtre* du schéma *homme* permet de déterminer pour un homme donné la liste de ses petits-fils en tant que valeur de l'attribut *petits-fils*

Dans KRL [Bobrow et al.77], le filtrage s'effectue sous la forme d'une requête *match* qui consiste à comparer deux descriptions (une donnée et un modèle) ou à bien à extraire d'un ensemble de données, celles qui satisfont les conditions du modèle. Lorsque les descriptions à apparier ont une structure interne d'objets complexes, l'appariement est divisé en sous-tâches qui consiste à apparier les divers morceaux de la structure. Dans les cas où l'appariement ne relève pas simplement d'une comparaison, même récursive, mais bien d'un raisonnement et d'une déduction, le filtrage peut être aidé par des procédures spécialisées introduites par l'utilisateur. Afin de ne pas ralentir le système et lui permettre d'effectuer d'autres tâches, le filtrage peut s'exécuter en tâche de fond et ne réapparaître que pour délivrer son résultat ou pour poser des questions à l'utilisateur lorsqu'il manque d'informations. Quatre types de résultats sont considérés pour un filtrage : la réussite totale

[8] En fait, il existe une utilisation particulière du filtre en SHIRKA qui permet de chercher la valeur d'un attribut monovalué.

(la délivrance du filtrage), l'échec total (la donnée contient des informations incompatibles avec le modèle), l'échec partiel par faute de ressources (le temps ou les ressources alloués au système pour le filtrage ne lui ont pas permis de décider) ou l'échec partiel par indécision totale malgré les stratégies connues. Dans KRL, la durée d'un filtrage peut être fixée. L'idée du *best match* ou meilleure correspondance est lancée ici en accompagnant la requête de critères qualitatifs ou de fiabilité dans les informations utilisées pour établir la correspondance. Enfin, l'idée de *forced match* ou correspondance forcée tend à étendre le filtrage afin qu'il puisse donner ce qu'il faudrait croire ou établir pour que l'appariement réussisse à travers une identification de la différence entre la donnée et le modèle. Ce filtrage ouvre des perspectives intéressantes d'établissement d'un raisonnement par analogie.

Le mécanisme de filtrage de FRL [Roberts et al.77] est plus rudimentaire que celui de KRL. Une requête est effectuée par la fonction *FQUERY* à partir d'une description (le modèle) référant la classe où se trouve les frames à filtrer (par le lien *a-kind-of*) et les diverses conditions sur les attributs exprimées à l'aide des facettes. Une variable (le frame recherché) est référencé par le symbole *?* et l'expression d'une requête parcourant plusieurs niveaux d'imbrication de frames complexes devient vite très compliquée.

Dans YAFOOL, un filtre correspond à une combinaison à l'aide des connecteurs logiques *ou* et *non* de listes de couples (attribut, expression). Une expression pouvant être un filtre, plusieurs niveaux d'imbrications sont possibles. L'originalité du filtrage réside ici en l'information restituée qui classe les objets en :

- *certains* qui vérifient le filtre : pour chaque attribut désigné dans le filtre, le domaine (ou la valeur) dans l'objet est inclus dans le (ou appartient au) domaine requis par le filtre ;
- *impossibles* qui ne vérifient pas le filtre : il existe au moins un attribut désigné dans le filtre pour lequel le domaine (ou la valeur) dans l'objet est disjoint du (ou n'appartient pas au) domaine requis par le filtre ;
- *possibles* pour lesquels chaque attribut désigné dans le filtre, le domaine (ou la valeur) dans l'objet a une intersection non vide avec le domaine requis par le filtre.

Les objets désignés dans chacune de ces catégories sont aussi bien des classes que des instances. La partition offerte par le résultat étend le domaine du filtrage et permet à l'utilisateur de se faire une idée d'un objet générique ou d'une classe correspondant au filtre.

Dans MERING, les requêtes peuvent être établies en utilisant des prédicats de contenance ou d'égalité. Les prédicats de contenance indiquent quelle(s) valeur(s) d'un certain type ou quelle(s) valeur(s) parmi plusieurs types possibles contient un attribut. Les prédicats d'égalité permettent de réaliser un appariement partiel (en spécifiant sur quels attributs portent la comparaison) ou complet (tous les attributs) entre deux objets, qui repose sur l'inclusion des domaines ou l'égalité des valeurs des attributs à comparer. D'autres requêtes plus complexes peuvent être formulées à l'aide de structures de contrôle classiques IF, WHILE, ..., d'opérateurs logiques AND, OR...et de quantificateurs logiques FORALL, EXIST...Les requêtes satisfaites permettent d'accéder à des objets entiers ou à des valeurs d'attributs d'objets. La méthode *match* existe également, elle est une généralisation du prédicat d'égalité (plusieurs objets sont sujets à l'appariement au modèle défini par une liste de descriptions d'attributs). La puissance de description du langage de requêtes et les prédicats de comparaison offrent de grandes possibilités de filtrage. Enfin, il est possible de formuler une requête depuis la description d'un attribut à l'aide d'une facette $si-req dont le résultat devient valeur de l'attribut.

OBJLOG+ et FROME sont deux systèmes de représentation de connaissances par objets qui donnent aux filtres une existence au sein de la base en qualité de classe. Cette approche semble assez naturelle si l'on considère que le filtrage consiste à extraire d'un ensemble d'instances, généralement d'une classe, celles qui satisfont des conditions exprimées par des descriptions d'attributs. Le filtre ne représente alors rien d'autre que la description d'une sous-classe de la classe filtrée, sous-classe dont les instances sont retenues par le filtre. Le filtre est une classe abstraite que ces deux systèmes proposent de faire devenir concrète, notamment pour pouvoir ré-utiliser des filtres existants en s'épargnant les étapes d'appariement.

Dans FROME [Dekker94], on peut décider de la persistance des filtres en tant que classes en leur attribuant un caractère *rémanent* ou *volatile*. Les filtres *conjonctifs* sélectionnent les frames appartenant à plusieurs classes. Un filtre conjonctif rémanent est placé directement (sans autre recherche de placement plus spécifique) dans la hiérarchie comme sous-classe directe des classes spécifiées dans le filtre. Les instances ayant satisfait le filtre deviennent alors instances de la nouvelle classe-filtre. Une fois que l'ensemble des instances retenues est établi, il est possible d'appliquer sur chacune d'elles une action (la faire évoluer vers une autre classe par exemple, lorsque le filtre est volatile). Les filtres *disjonctifs* représentent une disjonction de sous-filtres conjonctifs, c'est-à-dire l'union des ensembles d'instances satisfaisant les sous-filtres conjonctifs. La classe qui représente un filtre disjonctif rémanent est placée dans la hiérarchie comme sur-classe directe (la plus petite super-classe commune) des classes représentant ses sous-filtres. Lenneke Dekker précise que si l'extension de cette classe correspond à l'ensemble des instances satisfaisant le filtre, l'intension ne décrit pas les caractéristiques communes de ces instances. Enfin, un filtrage pour inférer la valeur d'un attribut est possible à travers la facette $if-needed-match, mais sans imbrication de filtre et sans variable pour impliquer un autre attribut dans le filtre, comme cela est possible dans la notion de filtre de SHIRKA.

Dans OBJLOG+ [Faucher91], le filtrage s'opère à l'aide de filtres ou de masques qui sont tous deux des classes. Un filtre est un ensemble de conditions imposées à des attributs de diverses classes appelés les *référents* du filtre. Comme en FROME, le filtre constitue une sous-classe virtuelle des référents du filtre qui va être placée, selon le même principe, dans la hiérarchie de classes. Le filtrage peut être total ou partiel selon que l'on cherche à déterminer toutes ou partie des solutions d'un filtre. Le filtrage repose sur deux phases : construction de la sous-classe des référents du filtre ; tentative de liaison par le lien structurel (pour un lien *est-un*, on vérifie les conditions sur les attributs des instances des référents). Si aucune solution n'est trouvée, la sous-classe est détruite ; sinon les instances solutions lui sont désormais liées. Un masque est une classe qui permet de sélectionner dans une liste de classes de la base, les instances qui vérifient la description du masque où figurent certains aspects, comme l'égalité, qui sont des conditions supplémentaires exprimées à l'aide d'un prédicat. Les masques se distinguent du filtre par le fait qu'ils ne sont pas introduits dans la hiérarchie de classes.

Dans SHIRKA [Rechenmann et al.90], le filtrage est opéré lors de la recherche d'une valeur (manquante) d'un attribut qui possède dans sa description la facette $sib-filtre. La valeur de la facette $sib-filtre est la description d'un schéma de classe accompagnée de restrictions supplémentaires sur certains attributs. L'attribut qui possède le filtre fait en général référence à d'autres attributs de son schéma de classe par l'intermédiaire de variables qui permettent de lever les ambiguïtés. La propagation des valeurs se fait à l'aide de deux facettes $var-> et $var<- qui indiquent le sens. Les filtres sont imbriqués (*cf.* figure 2.8) lorsqu'un filtre intervient dans la description d'un autre filtre. Lorsque plusieurs filtres apparaissent derrière la facette $sib-filtre, ils sont essayés séquentiellement. Tant que la cardinalité maximale de l'attribut n'est pas atteinte, les instances retenues par les filtres sont collectées. Enfin, lorsque tous les attributs décrits par un filtre ont une valeur, le balayage des instances d'une classe est inutile, la valeur cherchée est disponible immédiatement.

Nous proposerons au chapitre 13 de faire usage des filtres dans la description du type d'un attribut et nous montrerons également comment les contraintes peuvent faciliter l'expression des filtres et intervenir dans le filtrage.

2.1.2.4 La classification

Dans les langages de frames, la classification est un mécanisme peu répandu, au contraire des langages terminologiques [Haton et al.91, Napoli92] dont elle est l'opération de base. Alors que le filtrage se propose de déterminer l'ensemble des objets qui obéissent aux conditions d'un modèle, la classification cherche à déterminer, parmi un ensemble de modèles, celui qui correspond le plus précisément à un objet donné. Dans le domaine des représentations par objets, les classes offrent un ensemble de modèles organisés hiérarchiquement selon la relation de spécialisation. Selon le type – instance ou classe – de l'objet à comparer avec les différents modèles qui sont proposés, on distingue :

- la classification d'une instance (SHIRKA, FROME, SHOOD) consiste à déterminer par un parcours descendant dans la hiérarchie, la liste des classes *sûres*, *impossibles* et *possibles* au regard du contenu de l'instance. Le contenu de l'instance est comparé à la description de la classe en cours d'inspection. Une classe est étiquetée :
  - sûre, lorsque les valeurs des attributs de l'instance satisfont les conditions de toutes les descriptions d'attributs de cette classe (l'instance est complète et cohérente vis-à-vis de la classe);
  - impossible, lorsqu'au moins une des valeurs de l'instance ne satisfait pas les conditions d'appartenance à cette classe (l'instance est inconsistante vis-à-vis de la classe) ;
  - possible, lorsque toutes ses valeurs satisfont les conditions des descriptions des attributs correspondants, mais que certains attributs de cette classe n'ont pas encore de valeur définie (l'instance n'est pas en contradiction vis-à-vis de la classe : elle est cohérente mais incomplète).

  La classification est accélérée par une propagation d'étiquettes : les sous-classes de classes impossibles étant directement étiquetées impossibles. Il est donc inutile de continuer l'appariement plus bas dans la hiérarchie. Ayant obtenu la liste des classes sûres pour une instance, la sous-liste réduite aux classes sûres situées le plus bas possible sur des branches distinctes de la hiérarchie, détermine les classes les plus spécialisées (les plus petites familles ou catégories) auxquelles l'instance peut être rattachée de façon sûre (*cf.* figure 2.9).

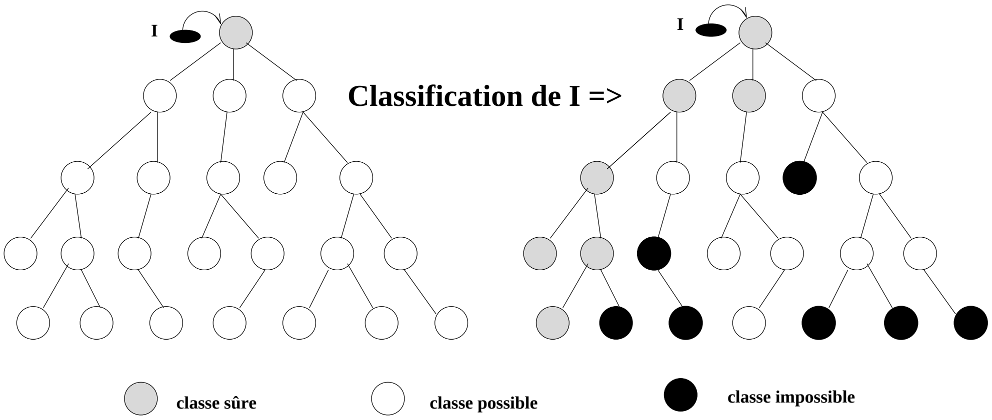


FIG. 2.9 - : Classification d'instance : la hiérarchie de classes est parcourue vers le bas afin de déterminer quelles classes sont sûres, possibles ou impossibles pour I. L'instance pourra être attachée aux classes étiquetées sûres et aux possibles aussi.

- la classification d'une classe qui consiste à déterminer la position d'une classe dans la hiérarchie établie (SHIRKA, FROME). Dans SHIRKA [Capponi93], elle est épaulée par un module de gestion des types d'attributs et la spécialisation est rapportée à la relation de sous-typage entre descriptions de classes. La classification de classes s'apparente à la classification de concepts dans les langages terminologiques [Haton et al.91, Napoli92].

Dans les deux cas de classification, la donnée initiale est un ensemble organisé de modèles existants : la hiérarchie. Il existe une troisième forme de classification appelée catégorisation qui consiste à construire des classes à partir d'un ensemble d'individus qui sera partitionné ; les différentes partitions (regroupements d'individus) sont établies selon des critères de ressemblance. Ce processus est utilisé en apprentissage, en catégorisation conceptuelle, ainsi qu'en acquisition de connaissances ([Napoli et al.91, Michalski et al.84]). Dans cette approche, qui ne sera pas détaillée davantage ici, rien n'existe au départ qu'un ensemble d'instances, les classes (modèles générateurs) sont à construire et à organiser.

Une étude générale des systèmes classificatoires a été proposée par Jérôme Euzenat [Euzenat93a, Euzenat94]. Un système classificatoire est un modèle abstrait de la structure sur lesquels peut être appliquée l'opération de classification. Il est montré que le comportement et l'algorithme associé à la classification sont dépendants des propriétés sémantiques et graphiques de la taxonomie (ensemble de classes ou catégories ordonnées par une relation d'ordre).

2.1.3 Cohérence et vérification de type dans les SRPO

Dans une base de connaissances, la cohérence d'un fait – représenté dans sa forme la plus élémentaire par la valeur d'un attribut – doit être établie à la fois :

- relativement à son type : une valeur proposée pour un attribut doit satisfaire le type de cet attribut. On parle dans ce cas de *vérification de type* ;
- relativement aux autres faits auxquels il est lié : si les valeurs des paramètres d'une méthode de calcul de la valeur d'un attribut changent, le contexte d'exécution de la méthode a été modifié, la valeur n'est donc *a priori* plus cohérente avec ce nouveau contexte, il faut relancer le calcul de cette valeur. On parle dans ce cas de *maintien de cohérence.*

La vérification de type intervient à la fois à la création d'une classe et à la création d'une instance.

- Lors de la création d'une classe, chaque description d'attribut doit faire l'objet d'une vérification de conformité à la relation de spécialisation de classes. S'il s'agit de la première occurrence de l'attribut dans la hiérarchie, elle est acceptée sous réserve d'être syntaxiquement correcte. Si, au contraire, l'attribut fait l'objet d'une redéfinition dans la classe, alors il faut garantir que le type décrit est un sous-type du type décrit par la dernière description de l'attribut (située dans une des sur-classes).

  Dans SHIRKA, l'intégration d'un module de gestion de types [Capponi93] assure la spécialisation de classes au moment de la définition d'une classe. En raison de la non-garantie de l'unicité de l'expression d'un type $T_{a_{C'}}$ d'un attribut a dans une classe C', une étape de normalisation doit précéder l'étape de sous-typage qui va le comparer au type T_{a_C} donné par le dernier affinement de a dans une sur-classe C (non nécessairement directe) de C'. Pour chaque attribut de la base, ce module peut, à la demande, construire un graphe de son type, dont les nœuds représentent les divers affinements ou restrictions effectués sur ce type et renseignent sur les classes où cet attribut a le type du nœud. Les arêtes représentent le lien de sous-typage entre deux types (nœuds). Ce graphe des types est utilisé lors de la classification de classes, mais aussi pour aider l'utilisateur à définir de nouvelles classes, en lui proposant les descriptions d'attributs à affiner ou à réutiliser.
- Lors de la création d'une instance, le système doit vérifier que toute valeur proposée est bien du type attendu en se référant à la description de l'attribut valué. Sont consultées dans l'ordre les facettes de typage, de restriction de type, de cardinalité, de vérification d'un prédicat.

Si la cohérence le long des liens de dépendance entre attributs (liens introduits par l'emploi de méthodes) peut être gérée par l'utilisateur à travers les réflexes $si-ajout, $si-enleve (YAFOOL), elle peut aussi être assurée automatiquement à l'aide d'un système de maintien du raisonnement (SMR) basé sur les principes du TMS de Jon Doyle [Doyle79] ou de l'ATMS de Johan DeKleer [Kleer86] (KEE, KRS [Marcke87]). Pour un système de maintien de raisonnement d'approche TMS, chaque attribut *inféré* à partir d'autres valeurs d'attributs garde dans une structure appelée *justification*, l'ensemble des noms des attributs *inférants* qui ont permis son calcul. En contrepartie, les attributs qui ont permis l'inférence contiennent dans une structure appelée *descendance* l'ensemble des noms des attributs qu'ils ont permis d'inférer. Lorsque la valeur d'un inférant est modifiée, une propagation paresseuse (il ne s'agit pas de recalculer mais de signaler que la valeur inférée n'est plus valide) est lancée le long des descendances. Lorsque la valeur d'un inféré est modifiée, il faut signaler aux inférants qu'ils ne le sont plus.

Dans SHIRKA, un tel système de maintien de la vérité [Euzenat87] permet d'instaurer un *caching* de valeurs : une valeur d'attribut obtenue par un mécanisme d'inférence (méthode, filtre) est stockée dans l'instance. Le système de maintien de la vérité évite donc le recalcul systématique des valeurs d'attributs. Tant qu'un attribut inféré n'a pas été atteint par la propagation d'une modification, l'obtention de sa valeur est donc immédiate. Si la propagation d'une modification a atteint l'attribut, sa valeur ne sera recalculée que lors de la prochaine demande.

À la solution coûteuse mais efficace d'intégrer un SMR pour des bases de connaissances souvent consultées et rarement modifiées, des langages de frames préfèrent simplement procéder au recalcul systématique, ce qui, à l'inverse, est un facteur de ralentissement lors de fréquentes interrogations d'une base de connaissances.

2.2 Conclusion

Ce chapitre a présenté les principes généraux de la représentation de connaissances par objets. Les systèmes de représentation de connaissances par objets prônent une structuration hiérarchique de la connaissance, obtenue par spécialisation, sur laquelle peut s'appliquer un mécanisme d'héritage permettant la factorisation des connaissances. En général, la connaissance dans ces systèmes est exprimée à deux niveaux :

- au niveau descriptif (intensionnel), par des objets – appelés classes – qui décrivent les propriétés (attributs) de l'ensemble d'individus qui peuvent leur être attachés. Ce sont ces descriptions qui permettent une organisation hiérarchique des objets par spécialisation.
- au niveau factuel (extensionnel), par des objets – appelés instances – qui sont les individus recensés des concepts du monde réel décrits au niveau descriptif.

Cette séparation de la connaissance est désignée par le terme d'approche *classe/instance*.

Les systèmes de représentation de connaissances à objets offrent des facilités de description et d'accès aux attributs. Les inférences sont possibles par l'association de méthodes, de filtres ou de réflexes aux attributs dans les classes. Certains, comme SHIRKA, proposent une classification des instances ou bien des classes. Les requêtes auprès d'une base de connaissances peuvent également se faire par filtrage à partir d'un modèle d'objet (FROME). La validité des informations proposées et contenues dans une base de connaissances nécessite la mise en place d'une vérification des types et d'un maintien du raisonnement. Ces précautions, si elles peuvent s'avérer coûteuses, sont le prix à payer pour garantir la cohérence des faits.

Cette étude s'inscrit plus particulièrement dans le cadre d'un système de représentation de connaissances appelé TROPES que nous décrivons dans le chapitre suivant.

Chapitre 3

Le modèle Tropes

Ce chapitre a pour but de présenter les principes du système de représentation de connaissances par objets TROPES sur lequel notre étude se base. La plupart des notions abordées dans le chapitre précédent (entités de représentation et mécanismes d'exploitation) sont intégrées dans TROPES. Cependant, par rapport aux autres SRPO décrits dans le chapitre 2, TROPES propose trois notions nouvelles : le *concept*, *le point de vue* et la *passerelle* qui imposent une nouvelle structuration de la connaissance, et, donc, une nouvelle façon de la modéliser, et permettent de faire de la classification d'instances le mécanisme d'inférence principal.

Les principes de TROPES ont été énoncés par Olga Mariño dans [Mariño et al.90, Mariño93]. Certains d'entre eux ont été étendus, approfondis ou modifiés pour les besoins de l'implémentation actuelle [Sherpa95]. Le noyau de base de ce système est décrit à travers ses deux aspects : le modèle de représentation d'un côté (*cf.* section 3.1), les mécanismes d'inférence disponibles de l'autre (*cf.* section 3.2). Nous décrivons ensuite comment s'effectue dans TROPES la vérification de types (*cf.* section 3.3), opération élémentaire du maintien de la cohérence des bases de connaissances. Pour finir avec la description de TROPES, nous rapportons trois études qui ont été proposées comme extensions du modèle TROPES (*cf.* section 3.4) et qui, à terme, viendront compléter l'implémentation actuelle : un modèle de tâches [Gensel et al.92], un système de gestion du raisonnement hypothétique [Girard95], et un module de gestion de types [Capponi95].

Le modèle TROPES ainsi décrit – noyau + extensions – est la base de départ de ce travail. À partir de là, nous montrons certaines insuffisances (*cf.* section 3.5) du modèle (ces insuffisances sont aussi celles de tout SRPO) relatives à la représentation et à la maintenance de relations mathématiques ou symboliques – typiquement, des contraintes. Combler ces insuffisances est l'objectif des travaux présentés dans ce rapport.

L'implémentation actuelle de TROPES a été réalisée avec le langage de programmation ILOG Talk [Ilog94]. Une interface de programmation d'applications ou API [Sherpa95] est disponible. Elle contient des fonctions qui permettent en Talk d'utiliser TROPES : c'est-à-dire créer, modifier, interroger ou détruire des bases de connaissances.

3.1 Les entités de description

TROPES est un système de représentation de connaissances par objets, basé sur une approche classe/instance dont les principes de représentation de la connaissance rejoignent ceux des autres SRPO décrits au chapitre précédent. Il intègre cependant deux entités descriptives supplémentaires : le *concept* et le *point de vue*. Les notions de *classe*, d'*instance*, d'*attribut* et de *facette* sont également présentes mais ont un comportement et une portée de description régis par les deux entités maîtresses que sont le concept et le point de vue.

3.1.1 La base de connaissances

Une *base de connaissances* est une structure qui englobe toute la connaissance relative au domaine d'application ou à la partie du monde réel modélisée. Par exemple, la base de connaissances *GEO-ECO-POLITIQUE* destinée à contenir les descriptions à la fois de divers États, de diverses institutions politiques, de ressources naturelles...

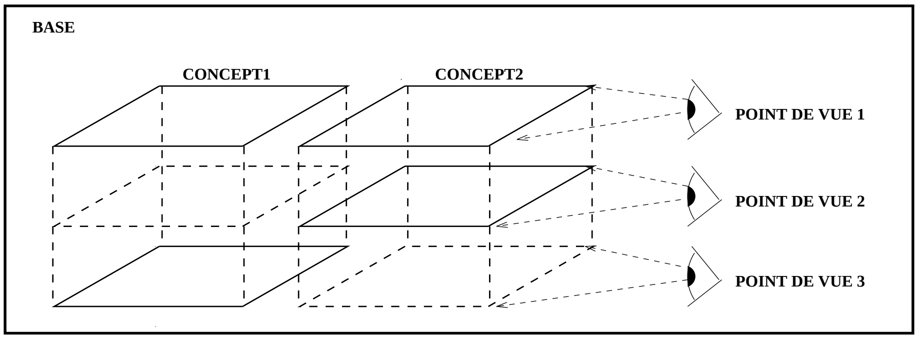


FIG. 3.1 - : Une base de TROPES est formée de concepts disjoints mais possiblement inter-dépendants et de points de vue. Ici, le concept 1 est visible depuis les points de vue 1 et 3, le concept 2 depuis les points de vue 1 et 2.

3.1.2 Le concept

Une base de connaissances est constituée de familles d'entités distinctes (États, institutions politiques, personnes, ressources naturelles) qui partitionnent l'univers modélisé dans la base. En TROPES, chaque famille est représentée par une structure appelée *concept* (*cf.* tableau 3.1). Les concepts sont considérés comme disjoints deux à deux au sens où un individu (un État, une institution, une personne, une ressource particulière) ne peut appartenir qu'à un seul concept.

Dans les RPO classiques, une base de connaissances est structurée en une seule hiérarchie dont la racine est la classe *Objet*. Les familles d'individus (les concepts de TROPES) sont des classes de cette hiérarchie. Mais le multi-héritage permet de les mélanger en créant des sous-classes qui héritent (à la levée de conflits près) les propriétés de leurs sur-classes. Avec cette façon de procéder, il n'est pas garanti que la classe résultante ait une signification (qu'est-ce qu'une instance d'un État-céréale?). Dans TROPES, la disjonction des concepts empêche ce genre de modélisation. Pour autant, la disjonction de concepts ne signifie pas que les concepts ne sont pas en relation (un État produit des céréales). La représentation de ces liens entre concepts est assurée par les attributs. Les concepts décrivent des familles d'individus. Les individus d'une même famille sont représentés

concept ÉTAT =
{ *clef: nom-État,*
  points de vue: (géographique, démographique, économique),
  attributs: (nom-État, nb-habitants...),
  passerelles: ∅ *;;; pas encore de passerelles définies*
}

TAB. 3.1 - : Description d'un concept : le concept ÉTAT avec les points de vue *géographique, démographique, économique*

par les *instances* du concept. Toute caractéristique commune à chaque individu est représentée par un *attribut*. L'espace des noms d'attributs est le concept : une caractéristique est donc référencée par le même nom dans tout le concept ; des caractéristiques de même nom apparaissant dans deux concepts distincts sont distinguées (par leur concept).

Le concept est chargé des prérogatives *ontologiques* des instances (c'est-à-dire ce qui concerne leur structure, leur intégrité et leur identité) [Euzenat93b]. En effet, la structure de chaque instance du concept est établie sur la liste des attributs du concept. Ces attributs donnent des caractéristiques générales qui régissent l'appartenance de l'instance au concept et veillent à l'intégrité de l'instance tout au long de sa vie. Enfin, l'identité de l'instance est assurée par une *clef*. Cette clef est une liste de valeurs d'attributs du concept (choisis et formant la clef) dont la donnée est nécessaire pour procéder à la création de l'instance. En tant qu'identité de l'instance, l'unicité de chaque clef est exigée ainsi que son invariance (la clef d'une instance ne peut être modifiée). Afin de veiller à la cohérence de la base de connaissances, les cycles entre clefs sont interdits : un attribut d'une clef d'un concept K ne peut faire référence à un concept dont la clef renvoie sur K.

3.1.3 L'attribut de concept

Un *attribut* (*cf.* tableau 3.2) est tout d'abord défini pour le concept en tant que caractéristique commune à toutes les instances de ce concept. La description de l'attribut pour le concept donne pour cet attribut :

- des informations constantes et immuables dans le concept comme sa *nature* (indication sur ce que représente l'attribut : une *propriété*, un *composant* ou une *relation*[1]), son *constructeur* (l'attribut est soit monovalué, soit multivalué, auquel cas sa valeur est un ensemble ou une liste), les *tâches* ou *méthodes* capables de calculer sa valeur ;
- des informations générales, comme le domaine maximal de valeurs que peut prendre cet attribut dans le concept, qui seront précisées dans les points de vue, au niveau des classes. En général, il est fait référence à un type de base (réel, entier, chaîne...) ou à un concept (l'ensemble de valeurs est l'extension – les instances – de ce concept) ou à un type défini construit à l'aide du module de gestion de types (date...).

attribut-concept nom-État/ÉTAT =
{ *points-de-vue : (géographique, démographique, économique)*
  dans : chaîne,
  nature : propriété,
  constructeur : un,
}

TAB. 3.2 - : Description d'un attribut de concept : l'attribut *nom-État* du concept ÉTAT est une chaîne de caractères.

3.1.4 Le point de vue

La base de connaissances est partitionnée en différents concepts mais la possibilité de décrire ou de considérer un même concept selon des perspectives d'observation et d'intérêt différentes est donnée par la structure de *point de vue*. La base est donc également partitionnée en points de vue. Il est possible d'observer plusieurs concepts depuis le même point de vue, alors qu'un concept peut être observé depuis plusieurs points de vue.

La notion de *point de vue* (*cf.* tableau 3.3) a été introduite dans TROPES pour permettre à des spécialistes de domaines différents de ne s'intéresser qu'aux propriétés d'un élément de connaissance modélisé qui sont particulières à leur domaine d'expertise. D'autres spécialistes dans d'autres domaines ont sans doute un point de vue différent et s'intéressent à un autre aspect de cette connaissance. Au niveau du concept, on retrouve les différents points de vue sous lesquels il est observé. Évidemment, les points de vues peuvent s'intéresser aux mêmes propriétés ou aspects de l'objet modélisé. De même, la caractérisation d'un ensemble d'individus du concept peut s'avérer identique ou incluse dans la caractérisation proposée par un autre point de vue (*cf.* section 3.1.8).

Bien que définie en dehors des concepts, la notion de point de vue est essentielle à l'intérieur du concept. Dans chaque point de vue d'un concept, n'apparaît qu'un sous-ensemble de l'ensemble des attributs du concept dits *visibles* ou *pertinents* sous ce point de vue. Parmi les attributs visibles

[1] La nature *relation*, bien que prévue dès sa conception, a été écartée dans l'implémentation courante du noyau où elle n'est pas distinguée de la nature *propriété*.

d'un point de vue figurent obligatoirement ceux qui composent la clef. Le point de vue joue le rôle de masque sur l'ensemble des attributs du concept. Un point de vue définit une décomposition de l'ensemble des instances du concept – elles sont toutes visibles sous ce point de vue – en hiérarchie de classes. La hiérarchie de classes d'un point de vue est établie sur la spécialisation correspondant à l'inclusion ensembliste des extensions. Cette hiérarchie a une structure d'arbre, il n'y a donc pas de multi-héritage dans TROPES, l'héritage est simple. De plus, chaque instance du concept n'est attachée qu'à une seule classe de chaque point de vue du concept. Aussi, s'il n'y a pas multi-instanciation dans un point de vue, on peut considérer qu'elle existe au niveau du concept. C'est d'ailleurs ce principe qui permet de modéliser ce que d'autres SRPO représentent par le multi-héritage sans avoir les inconvénients des conflits de noms : les caractéristiques des deux sur-classes dans un cas de multi-héritage sont ici réparties dans deux points de vue.

La racine du point de vue est une classe qui dénote tous les individus du concept afin que chacun d'eux soit observable et localisable dans ce point de vue. Elle n'a pas de sur-classe. La description de chacun de ses attributs ne peut être contrainte davantage que la description de ces attributs pour le concept car la classe racine est censée décrire l'ensemble de toutes les instances du concept.

À l'intérieur d'un point de vue, on retrouve les différents éléments des SRPO classiques : classe, instance (du concept, mais dont la vision est réduite à ce point de vue), attribut, facette.

point de vue économique/ÉTAT =
{ *racine : État-root*
}

TAB. 3.3 - : Description d'un point de vue : le point de vue *économique* est un point de vue du concept ÉTAT

3.1.5 La classe

Une *classe* (*cf.* tableau 3.4) d'un concept est avant tout classe d'un point de vue. Une classe décrit une sous-famille d'individus du concept à partir de tout ou partie des caractéristiques visibles sous son point de vue. La définition d'une classe contient la description des attributs qui sont affinés par rapport à la dernière description faite dans une sur-classe ou par rapport à la description donnée par la définition de l'attribut pour le concept, lorsqu'il s'agit de la première occurrence de l'attribut dans la hiérarchie de classes. La classe hérite les descriptions d'attributs qu'elle ne restreint pas.

La classe *racine* d'un point de vue est créée automatiquement à partir des définitions pour le concept des attributs-clefs. L'extension de la classe est l'ensemble des instances appartenant à cette classe. Elle donne l'ensemble *effectif* de ses instances (i.e. celles qui lui appartiennent) L'intension d'une classe est la description de ses attributs (hérités, affinés et définis). Elle décrit l'ensemble *potentiel* des instances de la classe.

La spécialisation de classes repose sur le sous-typage des classes. Le type d'une classe est le record des types de tous ces attributs : ceux qu'elle redéfinit et dont le type est affiné, ceux qu'elle hérite. Ce sous-typage garantit que toute instance appartenant à une classe C appartient à chacune de ses sur-classes situées sur un chemin allant de la classe C à la classe racine du point de vue. Deux hypothèses majeures sont faites sur la hiérarchie de classes :

classe État-root/économique =
{ *sorte-de : ∅ ;;; pas de super-classe*
  attributs : (nom, nb-habitants, pnb)
  spec : État-riche, État-pauvre ;;; les sous-classes
  instances : ∅, ;;; pas encore d'instance
}

TAB. 3.4 - : Description d'une classe : la classe racine du point de vue *économique* contient les attributs *nom, nb-habitants, pnb* et elle est sur-classe des classes *État-riche* et *État-pauvre*

- *hypothèse d'exclusivité* : deux classes sœurs (qui ont même sur-classe directe) sont exclusives :

leur extension ne peut contenir la même instance. Si l'on requiert que les extensions sont disjointes, cette contrainte n'est pas reportée sur l'intension des classes sœurs qui ne sont contraintes que vis-à-vis de l'intension de la sur-classe. Les intensions de deux classes sœurs peuvent donc dénoter des instances communes. Aussi, l'exclusivité est une propriété subjective que le système rappelle lors de la tentative de rattachement d'une instance à deux classes sœurs.

- *hypothèse de non-exhaustivité*: l'union des extensions des sous-classes directes d'une classe est incluse (non strictement) dans l'extension de cette classe. Autrement dit, une instance peut appartenir à une classe et n'appartenir à aucune de ses sous-classes. Il y a non exhaustivité des descriptions des sous-classes directes vis-à-vis de la description de la classe.

La classe est chargée des prérogatives *taxinomiques* de l'instance [Euzenat93b]. La classe est une projection de la structure du concept selon un point de vue qui ne retient que les attributs du concept à considérer et propose des contraintes sur les descriptions de ces attributs. Une classe offre la description d'une partition plus fine (racine exceptée) des instances du concept. Cette description est utilisée, lors de la création de l'instance si l'appartenance est déjà suggérée, mais surtout, lors de la détermination de l'appartenance à une classe plus spécifique, c'est-à-dire lors de la classification. De même que son établissement découle d'un acte délibéré de l'utilisateur approuvé par le système, à tout moment le lien de rattachement de l'instance à la classe, de nature taxinomique, peut être rompu. En revanche, le lien de rattachement de l'instance au concept, de nature ontologique, ne le peut pas.

3.1.6 L'attribut de classe

Un attribut d'une classe d'un point de vue d'un concept (*cf.* tableau 3.5) est décrit par des facettes (ou descripteurs).

attribut-classe nb-habitants/Vieillisant-peuplé =
{ *facettes: ($intervalle [50.000.000 +inf[)*
}

TAB. 3.5 - : Description d'un attribut de classe : l'attributs *nb-habitants* de la classe *Vieillissant-peuplé* (du point de vue *démographique*) est un nombre supérieur ou égal à 50 millions.

3.1.7 Les facettes de l'attribut de classe

Des facettes, optionnelles, permettent d'affiner la description d'un attribut. Contrairement à la plupart des SRPO, le noyau de TROPES ne comporte ni facettes procédurales, ni réflexes. Les descripteurs d'attributs de TROPES sont donc des facettes de typage.

- la facette $dans indique le type de l'attribut dans la classe. Ce type doit être à la fois sous-type du type général donné par la définition de l'attribut pour le concept, et sous-type du dernier type défini pour cet attribut dans la hiérarchie. Lorsqu'il s'agit d'affiner un type construit à partir d'un autre concept C de la base, cette facette peut indiquer, soit une classe de C ou une conjonction de classes de C de points de vue différents dont la valeur de l'attribut sera instance.

  Il est également possible d'indiquer comme type une sous-classe virtuelle d'une classe particulière à l'aide d'un *filtre* (*cf.* section 3.2.3).
- La facette $domaine énumère la liste de valeurs possibles pour l'attribut parmi l'ensemble des valeurs du type. Elle peut décrire des unions d'ensembles de valeurs.
- La facette $intervalle énumère, sous la forme d'unions d'intervalles, l'ensemble des valeurs possibles pour l'attribut. Elle offre une facilité d'écriture du domaine de valeurs possibles lorsque le type est ordonné.
- La facette $sauf énumère la liste des valeurs impossibles pour l'attribut.

- La facette $card indique le nombre d'éléments de la valeur (liste ou ensemble) d'un attribut multivalué. La valeur de cette facette peut être un entier, un ensemble ou un intervalle d'entiers.
- La facette $cond donne un ensemble de prédicats unaires qui sont à vérifier par la valeur de l'attribut.

Il faut noter que ces facettes, si elles facilitent la description du type, ne garantissent pas l'unicité de l'écriture du domaine de valeurs possibles. Aussi, TROPES procède à la normalisation des types exprimés par ces facettes.

3.1.8 La passerelle

À l'intérieur d'un concept, pour signifier qu'il existe une relation ensembliste entre les extensions des classes de points de vue différents, la notion de *passerelle* (*cf.* tableau 3.6) a été introduite. Les passerelles sont utilisées pour propager, entre différents points de vue, des informations sur l'appartenance d'une instance à une classe lors de la classification (*cf.* section 3.2.4). Une passerelle est constituée d'une ou plusieurs classes sources et d'une seule classe destination. On distingue :

- La passerelle unidirectionnelle qui relie une classe source S dans un point de vue PV_S à une classe destination D dans un autre point de vue PV_D. Elle exprime l'inclusion ensembliste de S dans D (si I est instance de S dans PV_Salors est I est instance de D dans PV_D).
- La passerelle unidirectionnelle à sources multiples qui relie plusieurs classes S_i (une par point de vue, et, au maximum, p - 1 classes sources, p étant le nombre de points de vue du concept) à une classe destination D. Elle exprime l'inclusion ensembliste des instances appartenant à chacune des classes S_i du point de vue PV_i ($\forall$ i $\in$ 1..n où n < p - 1) dans D (si I est instance de S_1 dans PV_{S_1}, et de S_2 dans PV_{S_2} $\ldots$, et de S_n dans PV_{S_n} alors I est instance de D dans PV_D).
- la passerelle bi-directionnelle qui relie deux classes C_1 et C_2. Elle exprime l'égalité ensembliste entre ces deux classes : toute instance de C_2 est instance de C_1 et réciproquement.

Par définition, il existe une passerelle bi-directionnelle entre chaque racine des points de vue, puisque celles-ci décrivent toutes les instances du concept.

| *passerelle* ÉTAT/*Europe/géographique* -> ÉTAT/*Vieillisant/démographique*
| { *type : unidirectionnelle*
| }

TAB. 3.6 - : Description d'une passerelle : la passerelle unidirectionnelle signifie qu'être un pays d'Europe du point de vue géographique, c'est être un pays vieillissant du point de vue démographique.

3.1.9 L'instance

Une instance est définie dans un concept de manière unique par la donnée de sa clef. On ne peut donc pas créer d'instance vide, sans clef ou avec un clef partiellement remplie. La structure d'une instance est définie à partir de la liste de tous les attributs du concept. En effet, en tant qu'individu du concept, l'instance se doit de comporter toutes les caractéristiques de tous les points de vue confondus du concept. La valeur d'une instance est la liste de tous les couples (attribut-valeur) du concept. L'instance peut être complète ou incomplète.

Une instance d'un concept est rattachée à une classe et une seule dans chaque point de vue du concept (d'après l'hypothèse d'exclusivité). Par le fait de la spécialisation des classes, dans chaque point de vue, elle appartient à chaque classe située sur un chemin menant de sa classe de rattachement à la racine de la hiérarchie. Par défaut, une instance est rattachée à la classe racine du point de vue : c'est le cas lorsque seuls les attributs nécessaires à son existence (i.e. les attributs clefs) ont une valeur. La structure globale d'une instance (celle formée à partir de la liste des attributs du concept) devient partielle dans un point de vue d'où ne sont visibles que les attributs

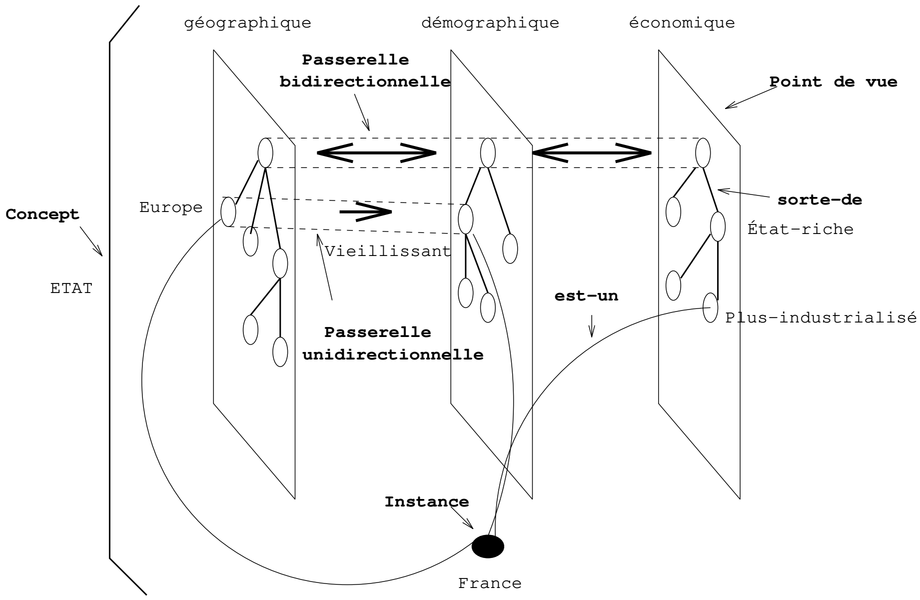


FIG. 3.2 - : Le concept État est visible depuis trois points de vue (géographique, démographique et économique). Les racines des points de vues donnent une description minimale de tous les États. Une passerelle bi-directionnelle est établie entre chacune d'elles.

pertinents.

Le lien qui lie une instance d'un concept à une classe dans un point de vue n'est pas fixe. TROPES permet la migration des instances par le biais d'un mécanisme de classification qui, en fonction du contenu d'une instance, cherche sa classe la plus spécifique dans chaque point de vue. Ce mécanisme peut être appelé après la création d'une instance pour tenter de faire descendre davantage l'instance dans chacune des sous-hiérarchies de classes dont ses classes d'appartenance sont les racines. Il peut aussi être appelé lors de la modification de la valeur d'un attribut dans une instance. Si la valeur vérifie toujours les conditions imposées par la classe, l'instance est susceptible de migrer vers une sous-classe. Sinon, si la valeur non valide est maintenue par l'utilisateur, l'instance est à reclasser depuis la racine de la hiérarchie (ce qui revient à la remonter vers la sur-classe la plus spécialisée à laquelle elle puisse être rattachée).

instance État-01 =
{ nom-État : "France"
nb-habitants = 55.000.000,
}

TAB. 3.7 - : Description d'une instance : la France est un état qui possède 55 millions d'habitants.

3.1.10 Objets composites

Dans TROPES, un objet composite (*cf.* tableau 3.8) est un objet qui a, parmi ses attributs, des attributs de nature *composant*. Ceci signifie que l'objet est impliqué dans une relation *partie-de* ou de composition avec d'autres objets d'autres concepts. L'objet (le composite) est le tout, les objets désignés par les attributs composants sont ses parties (ses composants). Un composant peut lui-même être composite, la composition est une relation transitive. Dans la décomposition récursive d'un objet composite, TROPES assure qu'il ne soit pas partie de lui-même.

Des diverses interprétations de la relation de composition présentées en section 2.1.1.6, TROPES établit une *dépendance exclusive* – le composant ne peut être composant que d'un seul composite – et *existentielle* – un composant créé pour un composite doit disparaître avec lui – du composant vis-à-vis du composite. De plus, un composant doit être visible depuis tous les points de vue du composite sous lesquels l'attribut-composant apparaît (il peut posséder d'autres points de vue qui lui sont propres). Un attribut composant multi-valué peut être assujetti à un affinement particulier

```
classe République =
{ RÉGIME/politique,
    ∅,
    nom : $dans chaîne,
    constitution : $dans constitution,
    partis : $dans partis
    présidents : $dans personne,
}
```

TAB. 3.8 - : Description d'une classe d'objet composite : la classe *République* du concept RÉGIME considéré du point de vue *politique* est un objet composite formé d'un nom, d'une constitution, de partis politiques et de présidents.

appelé *éclatement* qui consiste à décrire plus précisément le type de chaque instance (dans l'exemple de la classe *République*, on pourra éclater l'attribut *partis* en *partis-de-gauche*, *partis-de-droite*, *sans-étiquette*).

Un composite peut diffuser des valeurs vers ses composants qui héritent alors des propriétés et des valeurs du composite.

3.1.11 Relations

Dans le noyau de TROPES, les relations sont à représenter par des attributs de nature *relation*. Un attribut *relation* a pour type un type construit à partir d'un concept, d'une classe ou d'un ensemble de classes de la base de connaissances, et pour valeur un objet ou un ensemble ou une liste d'objets avec lequel l'objet est en relation. Ces attributs n'ont pas de comportement particulier et ont le même traitement que les attributs-propriétés.

3.2 Les mécanismes d'exploitation

3.2.1 L'instanciation

L'*instanciation* correspond à la création de l'instance. L'instance est définie par son concept et par sa clef, puis par la donnée de classes de rattachement (une seule par point de vue) et par une liste de couples (attribut, valeur) qui forment la valeur de l'instance. Aussi, dans TROPES, c'est le concept qui est l'entité génératrice, et non pas la classe, comme dans les autres SRPO.

Par défaut, lorsque dans un point de vue, il n'est pas spécifié de classe d'appartenance, l'instance est rattachée à la classe-racine de ce point de vue. Pour chaque classe d'appartenance proposée, on confronte les valeurs des attributs de cette classe dans l'instance avec les descriptions fournies ou héritées par la classe. Si l'une des valeurs proposées n'appartient pas au domaine des valeurs possibles pour l'attribut dans cette classe, l'instance n'est pas créée[2]. À la création de l'instance, l'appartenance à la classe n'est pas remise en cause par l'absence d'une valeur d'un attribut de la classe. Si toutes les valeurs contenues par l'instance et relatives aux attributs de la classe sont des valeurs admissibles, l'appartenance de l'instance à la classe est accordée.

L'instance est créée lorsque l'appartenance à toutes les classes proposées est vérifiée. Il est possible que des valeurs aient été fournies pour des attributs qui n'apparaissent pas dans les classes d'appartenance. Une telle valeur devra alors satisfaire la description de l'attribut correspondant pour le concept, l'instance étant avant tout instance du concept.

[2]On pourrait cependant, sous réserve que la clef soit correcte, tenter de déterminer la classe d'appartenance de l'instance à partir de la racine.

3.2.2 Le détachement procédural

Dans la version initiale du noyau, la définition de l'attribut pour le concept était prévue pour indiquer une tâche ou, plus simplement, une procédure destinée à calculer la valeur de l'attribut. La localisation de cette information fait que le terme d'*attachement procédural* employé dans les SRPO classiques doit être substitué par celui de *détachement procédural* dans TROPES, puisque l'information procédurale concernant l'attribut n'est plus attachée à la classe, mais reportée au niveau du concept. Ce choix peut être justifié de deux façons :

- par l'existence de l'instance qui n'est pas subordonnée à son attachement à une classe dans un point de vue (au pire, la classe racine fait l'affaire) mais à la donnée de sa clef qui lui donne vie dans le concept ;
- au titre de la cohérence de l'appariement utilisé lors de la classification. En effet, lorsqu'une instance incomplète est confrontée à la description d'une classe pour juger de son appartenance, activer une procédure désignée par cette classe afin de compléter l'instance pour ensuite décider en fonction du résultat de l'appartenance à cette classe, c'est déjà anticiper sur l'appartenance à la classe [Rechenmann92, Rechenmann93]. Le contenu d'une instance ne peut être complété que lorsque le lien de rattachement est établi. Aussi, si des moyens sont connus pour calculer la valeur d'un attribut, ils doivent être fournis au niveau du concept et non plus dans les classes qui sont réduites au rôle d'entité d'identification (et non de construction) d'instance durant la classification.

De plus, dans TROPES, l'attribut est une caractéristique du concept, qui peut apparaître dans plusieurs points de vue. Aussi, il est raisonnable de décrire des stratégies de calcul de cet attribut, qui prennent en compte divers contextes d'exécution. En fonction de l'information disponible dans l'instance et dans la base, tel contexte d'exécution pourra être choisi. Le calcul de l'attribut n'est donc plus dépendant de ses classes d'appartenance.

Bien que non encore implémenté dans la version actuelle de TROPES, l'étude du détachement procédural a débouché sur la conception d'un modèle de tâches décrit à la section 3.4.2.

3.2.3 Le filtrage

Non présente dans les spécifications initiales du modèle, la notion de filtre a été introduite dans le modèle en reprenant certains des points détaillés au chapitre 13.

Dans TROPES, les filtres ne sont pas considérés comme des classes mais plutôt comme des descriptions de classe. Un filtre est un objet qui peut être créé et manipulé (modifié, détruit) exactement comme la définition d'une classe. Un filtre n'apparaît pas dans le point de vue de sa classe et aucune instance ne peut lui être attachée (par un lien d'appartenance).

Il est possible de créer, de détruire, de lancer le filtrage ou de connaître l'ensemble des filtres qui ont une classe donnée comme base ou l'ensemble des filtres (fermeture transitive) des sous-classes d'une classe donnée.

Les filtres ont donc un usage pour les requêtes sur l'extension d'une classe mais peuvent être utilisés pour l'affinement d'un type d'attribut. L'emploi des filtres en tant que types est propre au modèle TROPES et est géré par METÉO (*cf.* section 3.4.1), le module de gestion de types associé à TROPES. Le filtre décrit alors une sous-classe (ou un sous-filtre) de la classe utilisée dans une utilisation antérieure de l'attribut (attachée à une sur-classe dans la hiérarchie).

3.2.4 La classification

Le mécanisme de raisonnement principal dans TROPES est la *classification* multi-points de vue d'une instance (*cf.* figure 3.3). La classification consiste à déterminer dans chaque point de vue d'un concept, la classe la plus spécialisée à laquelle l'instance peut être rattachée, en fonction des informations qu'elle contient. La classification utilise les informations des passerelles entre les points de vue pour *passer* d'un point de vue à un autre et accélérer ainsi la détermination des classes d'appartenance. L'algorithme de classification fournit comme résultat une étiquette pour chaque classe. Les classes sont étiquetées *possible*, *impossible*, *inconnu* , selon le degré de certitude de l'appartenance à la classe, et non plus respectivement *sûre, impossible, possible* pour bien insister sur le fait qu'il revient finalement à l'utilisateur (ou à un programme de substitution) de décider du rattachement. Une propagation des étiquettes *impossibles* vers les sous-classes permet d'accélérer le processus.

Le principe de l'algorithme repose sur la confrontation de l'instance à une classe en vue de déterminer l'étiquette de cette classe. La classification est un mécanisme interactif, en l'absence de valeurs pour un attribut de la classe, le système peut solliciter l'utilisateur. Afin d'informer le système sur ses compétences, l'utilisateur peut classer les points de vue en :

- points de vue principaux : lorsque la classification s'effectue sous un de ces points de vue, le système peut interroger l'utilisateur sur des valeurs manquantes d'attributs. Ce sont les points de vue prioritaires pour la descente de l'instance.
- points de vue auxiliaires : lorsque la classification est bloquée sous un point de vue principal, elle peut passer à un point de vue auxiliaire par le biais de passerelles, si elles existent, afin d'y compléter la connaissance, avant de revenir dans un point de vue principal.
- points de vue cachés : la classification dans ces points de vue ne peut être que le résultat de propagation de marques par l'utilisation d'éventuelles passerelles. Le système n'interroge pas l'utilisateur sur les attributs de ces points de vue.

Les passerelles jouent un rôle important dans la propagation d'étiquettes lors de la classification. Si toutes les sources d'une passerelle unidirectionnelle s'avèrent possibles, alors la destination est étiquetée possible. Si la destination d'une passerelle unidirectionnelle s'avère impossible, alors la source, si elle est unique, est étiquetée impossible. La classification d'objets composites [Mariño91]

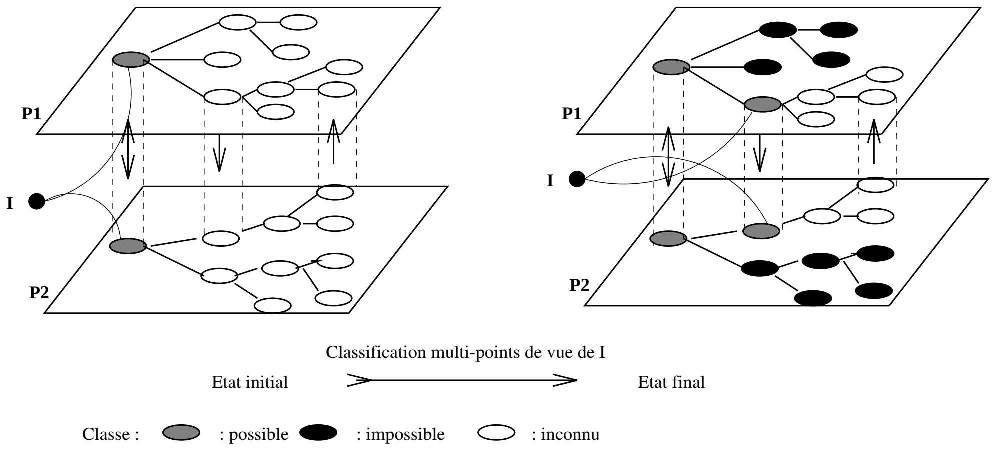


FIG. 3.3 - : La classification multi-points de vue (d'après [Mariño93]) est décrite comme le passage d'un état initial où une partie de l'information de l'instance (au moins son concept) est connue (dans le pire des cas (ici), les classes racines peuvent déjà être marquées possibles) à un état final où le maximum d'informations a été récupéré sur l'instance, classée le plus bas possible dans chaque point de vue. Ici, la classification s'est faite sur le point de vue P1, la passerelle unidirectionnelle a permis de propager les marques dans P2.

bénéficie de deux interprétations :

- la classification minimale d'un objet composite se contente de vérifier que les composants satisfont les conditions qui sont imposées par la description des attributs-composants. Un composant n'est descendu dans sa hiérarchie que si sa validation le requiert. C'est le fonctionnement par défaut choisi par TROPES.
- la classification maximale classe tous les composants le plus bas possible dans leurs hiérarchies.

Si une instance composite n'a pas de valeur pour un attribut composant, le système, après avoir vérifié la validité des valeurs des autres attributs, examine la description de l'attribut-composant non valué. Si la description est suffisamment précise (pour un multivalué, la cardinalité est fixée et l'attribut est totalement décrit par éclatement), l'instance composante peut être créée. Dans le cas contraire, la création du composant est repoussée.

3.3 Vérification de types

Dans Tropes, la vérification de types [Capponi95] intervient lors de la construction et de l'évolution de la hiérarchie de classes d'un point de vue afin de garantir la spécialisation de classes, ainsi que lors de la création et de la modification de classes ou d'instances.

Pour une classe dans un point de vue, on distingue son type local fourni par sa description complète (record des types des attributs hérités, affinés ou définis) de son type global (record de tous les attributs du concept ; les attributs n'appartenant pas à la classe ont le type donné par leur définition pour le concept). Si le premier est considéré lors de l'établissement de l'appartenance d'une instance à une classe, le second est utilisé lorsque la classe est donnée comme type d'un attribut et, dans la perspective d'une gestion dynamique des classes, pour vérifier la cohérence des arbres de points de vue.

Le type d'un attribut dans une classe est défini par la combinaison des facettes de typage. Il est soumis à une normalisation avant d'être utilisé pour le type de la classe. L'affinement d'un type d'attribut dans une classe est basé sur le sous-typage (ou inclusion des domaines de valeurs décrits par les types). Pour chaque attribut d'un point de vue du concept, Tropes maintient un graphe des différents sous-types du type général qui apparaissent dans la hiérarchie de classes. La vérification d'une valeur d'un attribut se réfère au nœud du graphe de types équivalent et est ainsi accélérée (économie d'un parcours du graphe de classe pour déterminer le type de l'attribut).

À la création d'une instance dont la structure globale inclut tous les attributs du concept, le type global initial d'une instance est défini comme le record de tous les types donnés par les définitions d'attribut pour le concept. Pour une instance d'une classe dans un point de vue, on considère son type local comme l'intersection du type global de la classe et du type global de l'instance (l'intersection sur les records de type est définie comme le record des intersections de chaque type). Vue du concept, l'instance est rattachée à une classe dans chaque point de vue : son type pour le concept est l'intersection des types d'instance pour chacune de ses classes d'appartenance.

Lors de la modification d'une valeur d'attribut d'une instance, si le lien taxinomique d'appartenance à la classe dans un point de vue peut être rompu, il n'en va pas de même du lien ontologique qui lie l'instance au concept. Il faut donc d'abord s'assurer que la valeur est bien du type donné par la description de l'attribut pour le concept. Si ce n'est pas le cas la modification est refusée. Si le lien ontologique est maintenu, deux situations peuvent se produire : le lien taxinomique est aussi maintenu (l'instance satisfait encore les conditions de la classe et une classification peut être ou non demandée) ou le lien taxinomique est rompu et alors une classification de l'instance doit être lancée depuis la sur-classe la plus directe à laquelle l'instance appartient avec certitude (par défaut, la racine).

Au niveau des types de classes sœurs, bien que l'hypothèse d'exclusivité régisse la disjonction de leur extension, il n'est pas opérée de vérification de type en ce sens. Exiger des records de types distincts pour deux classes sœurs serait trop restrictif et surtout reporterait la condition au niveau de l'intension.

Par contre, l'inclusion ensembliste imposée par une passerelle entre C et C' (l'extension de C est incluse dans celle de C') a pour effet d'imposer que le record des types des attributs de C présents dans C' soit un sous-record du record des types des attributs de C' au sens de [Cardelli et al.91]. De plus, si une passerelle est établie entre une classe C et une classe C', alors une passerelle ne peut être établie entre toute sous-classe de C et toute classe sœur ou sous-classe d'une classe sœur de C'.

3.4 Les extensions de TROPES

Le noyau présenté ci-dessus a été étendu vers l'utilisation de connaissances procédurales exprimées dans un modèle de tâches ([Orsier90, Gensel et al.92]) (assisté d'un système de maintien du raisonnement chargé de la cohérence des liens de dépendances établis par les tâches [Gensel90]). La gestion d'un raisonnement hypothétique étudiée dans [Gensel90] est prolongée par les travaux de Pierre Girard [Girard95]. Enfin, la gestion des types réalisée par le noyau est une des fonctionnalités d'un module général d'extension de types élaborés pour les objets, réalisé par Cécile Capponi [Capponi95]

Ces extensions sont fortement inter-connectées. Si les unes (modèle de tâche, gestion du raisonnement hypothétique) accroissent les fonctionnalités de représentation et d'inférence du système TROPES, les autres (système de maintien du raisonnement, gestion des types) visent à contrôler la cohérence de l'ensemble.

3.4.1 Le module de gestion de types

Un module extensible de types élaborés pour les objets (METÉO [Capponi95]) a été couplé à TROPES. Il est chargé, d'une part d'accélérer la vérification des types lors de la création, de la modification et de l'utilisation d'une base de connaissances TROPES et, d'autre part, de la gestion de types définis, introduits dans la base par l'utilisateur mais qui ne correspondent pas à des types de classes de la base.

Le système de types METÉO définit deux niveaux de types. Ainsi sont distinguées les notions de C-types et de δ-type (*cf.* figure 3.4).

La notion de C-type correspond à la description des types de base simples (entiers, réels, chaîne...), des types de base obtenus par l'emploi des constructeurs liste et ensemble, des types de base définis ou élaborés par l'utilisateur (date, séquences...), ainsi que les types records qui correspondent aux descriptions classes de la base.

Un C-type est une structure de données dédiée à la représentation d'un type abstrait T. Le C-type fournit l'ensemble maximal des valeurs de type T, le prédicat d'égalité entre les valeurs de ce type ($=_T$) et un prédicat d'appartenance au type ($\in_T$). Si le type abstrait décrit un type ordonné, la relation d'ordre ($\leq_T$) correspondante peut être également fournie. Cette description peut être complétée par des spécifications d'opérateurs sur le C-type correspondant.

METÉO offre à l'utilisateur la possibilité de définir de nouveaux C-types et de les organiser dans une hiérarchie basée sur le C-sous-typage. Le C-sous-typage est une relation subjective à la charge de l'utilisateur qui ne peut être établie automatiquement par le système (par exemple, pair est un C-sous-type de entier).

Un δ-type d'un C-type décrit un sous-ensemble de l'ensemble des valeurs du type T (donné par le C-type). Les δ-types décrivent les partitions de l'ensemble des valeurs du type T présentes dans la base.

Pour chaque C-type, est calculé un treillis de δ-types dont la relation d'ordre partiel est la relation de sous-typage (δ-sous-typage) préservant l'inclusion des domaines. Le sommet de ce treillis est un δ-type décrivant le C-type.

De plus, un Γ-sous-typage peut être établi à la demande entre un δ-type D2 d'un C-type C2 et un δ-type D1 d'un C-type C1 tels que C2 est un C-sous-type de C1.

METÉO procède à une normalisation des types d'attributs TROPES qui consiste à exprimer les domaines associés aux attributs sous forme de δ-types. Un nœud du treillis des δ-types contient ainsi la forme normalisée de l'ensemble de valeurs dénoté par le δ-type correspondant, et un lien avec les attributs TROPES dont il est le type.

Lors de la définition d'un nouveau type pour un attribut, METÉO classe le type proposé dans le treillis de types correspondant et s'assure du δ-sous-typage et, éventuellement, du Γ-sous-typage. Lors de la confrontation d'une valeur avec le type d'un attribut, le lien entre l'attribut TROPES et

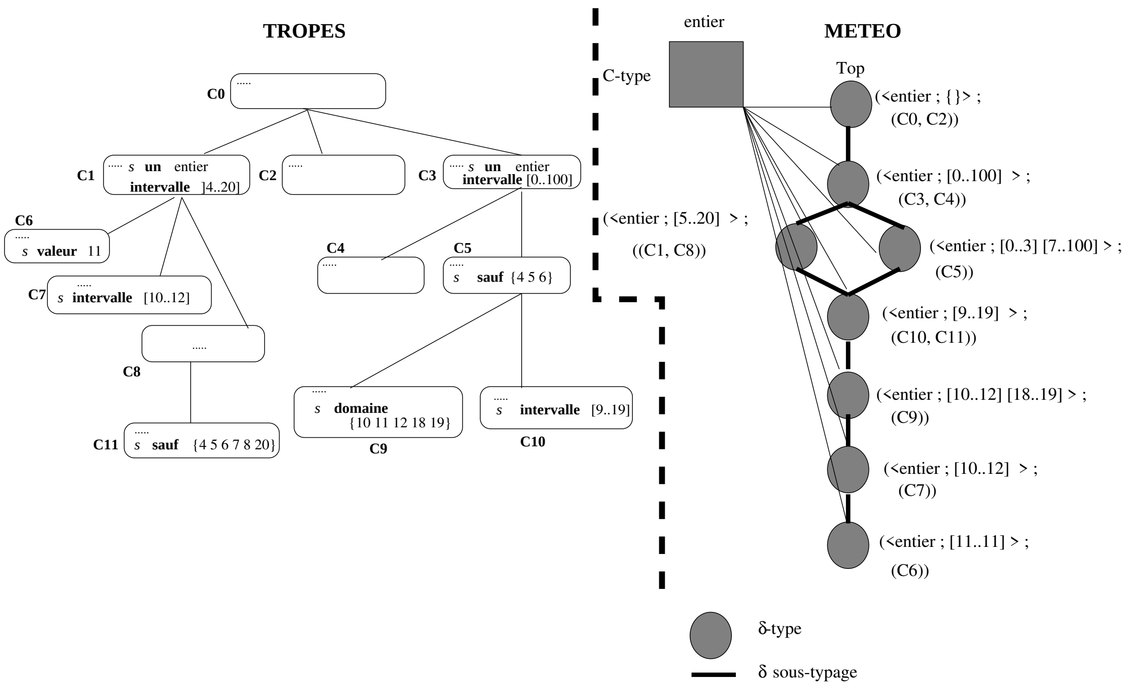


FIG. 3.4 - : (d'après [Capponi94]) À gauche le graphe de spécialisation de classes (réduit à celui de l'attribut s) et à droite le treillis du type C-entier (où seuls les nœuds concernant s sont représentés). Le treillis du type contient tous les liens de δ-sous-typage existants pour s.

son δ-type permet une vérification immédiate.

METÉO maintient également un graphe des types de classes. Les types de classes sont des δ-types du C-type *record*. Le type d'une classe est représenté sous la forme d'un record, où chaque élément réfère au δ-type de l'attribut correspondant, qu'il soit de type simple, construit, élaboré ou de classe. Il existe donc des liens entre les divers treillis de types.

La spécialisation entre classes repose sur le sous-typage des records [Cardelli et al.91] des types des attributs. Ainsi, pour établir qu'une classe D de description $[a_1 : t_{d1}, a_2 : t_{d2} \ldots, a_n : t_{dn}]$ est plus spécialisée qu'une classe C de description $[a_1 : t_{c1}, a_2 : t_{c2} \ldots, a_m : t_{cm}]$ où a_i est un attribut et t_{ji} son type dans la classe j, il faut établir que :

- $m \leq n$, ce qui est assuré par l'héritage, et
- $\forall i$ tel que $1 \leq i \leq m, t_{di} \leq_t t_{ci}$.

Lorsqu'un des t_{ji} est un type de classe, la même vérification est appliquée.

À partir des deux prédicats requis pour la définition d'un C-type et d'autres propriétés spécifiques (des opérations, des comportements, des actions...), METÉO est en mesure d'assurer la gestion dynamique (insertion et suppression) des treillis de types associés aux types présents dans une base.

3.4.2 Le modèle de tâches

Pour décrire la connaissance stratégique concernant la résolution d'un problème (ici, l'obtention de la valeur de l'attribut), un premier modèle a été proposé par Bruno Orsier [Rousseau88, Orsier90]. Ce modèle distingue les notions de *tâche*, de *méthode* et de *procédure* pour rendre compte des différents niveaux de description d'une connaissance stratégique dans la résolution d'un problème. Une tâche décrit une stratégie prédéfinie d'utilisation de procédures. Une méthode est le mode d'emploi d'une procédure. Une procédure est un simple morceau de code exécutable. Tâches et méthodes sont décrites par des classes et organisées en hiérarchies de spécialisation qui distinguent les tâches et les méthodes par le contexte d'évaluation. Associé à ce modèle de tâches, un système de maintien du raisonnement SMR [Gensel90], basé sur les principes du TMS, est chargé d'enregistrer les liens de dépendances entre un attribut calculé par une tâche et les attributs qui ont servi de paramètres. Comme dans SHIRKA, le caching est alors possible pour les valeurs inférées par une procédure. Le SMR est chargé de la mise à jour des justifications des attributs inférés et des descendances des attributs inférants avec les informations reçues du modèle de tâches. Il effectue également une propagation paresseuse des modifications afin de signaler qu'une tâche est à réactiver.

Un second modèle de tâches a été proposé ([Gensel et al.92]) dans lequel la tâche est intégrée totalement en termes d'objets et où la notion de méthode n'apparaît pas. Une tâche (*cf.* figure 3.5) est une entité de description de la résolution d'un problème (i.e. de la décomposition de ce problème en sous-problèmes). Un concept *Tâche* permet d'exprimer et d'organiser, à travers une hiérarchie de classes, l'éventail des tâches dédiées au calcul des attributs d'autres concepts de la base. Une tâche est décrite par :

- son but. L'attribut *but* de type chaîne décrit l'objectif que la tâche permet d'atteindre.
- ses entrées. Elles sont décrites via l'attribut *entrées*, de type ensemble de valeurs. Chaque entrée est plus précisément décrite au moyen de l'éclatement d'attribut.
- son traitement. L'attribut *traitement* décrit le lien de composition entre la tâche et ses sous-tâches. Cet attribut est une liste d'objets du concept TÂCHE. Là encore, l'éclatement de cet attribut permet de décrire la séquence de sous-tâches. Lorsque la tâche n'est plus décomposable, elle fait appel à une procédure externe.
- ses sorties. Elles sont décrites par l'attribut *sorties* de la même façon que les entrées.

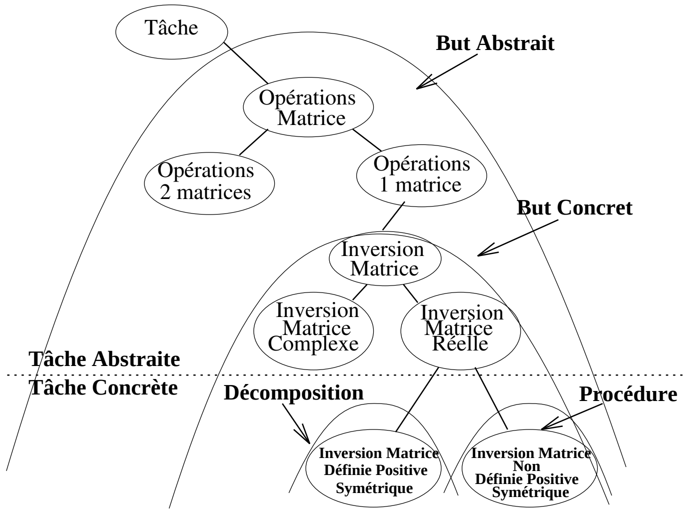


FIG. 3.5 - : Le concept TÂCHE. Description de la spécialisation des opérations sur les matrices.

Dans la hiérarchie de tâches, on distingue quatre types de tâches suivant le niveau de précision des divers composants. Une tâche est dite *abstraite avec but abstrait* lorsque sa classe ne fournit qu'une description générale. Une tâche est dite *abstraite avec but concret* lorsque sa classe indique explicitement le but atteint mais ne définit pas le traitement précisément. Une tâche est dite *concrète non terminale* lorsque sa classe décrit une décomposition en sous-tâches particulières par éclatement des attributs *traitement* et *sorties*. Une tâche est dite *concrète terminale* lorsque sa classe indique une procédure.

L'exécution d'une tâche débute par la construction d'une instance par instanciation de la classe racine de la hiérarchie correspondante. Le choix du traitement (décomposition en sous-tâches ou exécution d'une procédure externe) le plus adapté se traduit ensuite par la classification de cette instance de tâche vers la classe la plus spécifique (celle qui décrit le plus précisément le contexte d'appel de la tâche formé des entrées et du but). Puis, si la classe correspond à une tâche concrète non terminale, l'instanciation en séquence de chaque sous-tâche est lancée, si la classe est concrète terminale, la procédure est lancée. Ce processus récursif s'arrête lorsque l'activation des procédures est terminée (il y a échec si les résultats obtenus ne satisfont pas les conditions sur les sorties) ou lorsque la classification n'a pas pu déterminer une classe concrète non terminale (il y a échec).

Ce modèle permet donc de décrire une connaissance stratégique de résolution de problème (dans une classe), de la construire (par instanciation), de l'adapter à la configuration (par classification), de garder une trace de l'exécution (dans une instance). Cependant, l'une des principales difficultés de l'implantation d'un modèle de tâches réside en la description et la mise en œuvre du passage de paramètres entre tâches. Elle peut être exprimée par un attribut *flot-de-données* dont l'interprétation de la valeur est "codée en dur" comme dans les modèles de tâches SCAI [Poncabaré et al.91] et SCARP [Willamowski94], développés sur Shirka, le prédécesseur de Tropes. Nous proposons (*cf.* section 11.2) de mettre à profit la présence de contraintes pour décrire et contrôler cet échange entre tâches.

3.4.3 Le module de raisonnement hypothétique

Les premiers travaux pour la gestion d'un raisonnement hypothétique dans Tropes [Gensel90] ont été motivés par le besoin de juger de l'évolution d'instances incomplètes après émission d'hypothèses concernant les valeurs non définies. Deux sources d'hypothèses sont présentes dans Tropes : les domaines de valeurs possibles donnés par les types des attributs de l'instance et l'exécution indéterministe d'une tâche qui offre plusieurs résultats possibles. Une *instance hypothétique* est créée à partir d'une instance incomplète de la base (dite concrète) à laquelle on ajoute des valeurs hypothétiques d'attributs. L'ensemble des instances hypothétiques construites à partir d'une instance et d'ensembles disjoints d'hypothèses est appelé *bassin hypothétique* de l'instance. Les mécanismes de raisonnement sont applicables à l'instance hypothétique et permettent de la compléter plus encore (classification, inférence de valeurs d'attributs). La gestion du bassin hypothétique consiste à répercuter les effets de la fixation d'une valeur hypothétique, ou de la modification d'une valeur hypothétique, ou de la suppression d'une instance hypothétique dans le bassin hypothétique en modifiant ou en éliminant les instances hypothétiques concernées. Une utilisation d'hypothèses pour compléter une instance lors de la classification est également proposée.

Ces travaux sont étendus dans la thèse de Pierre Girard [Girard95] par la création d'un système d'assistance sous raisonnement hypothétique dont le rôle est de gérer des versions hypothétiques d'une instance. Cette étude tenant compte des propositions faites dans les chapitres suivants, elle sera décrite ultérieurement (*cf.* chapitre 10).

3.5 Vers l'introduction de contraintes dans Tropes

Les diverses entités de représentation de Tropes (concept, point de vue, classe, attribut, facette, instance) ont pour vocation de faire de ce système un outil de représentation suffisamment général pour s'adapter à la modélisation de divers domaines d'application. Cependant, dans l'état actuel de sa conception, il faut constater que Tropes ne permet pas l'expression de propriétés mettant en jeu plusieurs attributs.

Pourtant, il est souhaitable qu'un tel modèle à objets offre cette fonctionnalité, notamment parce qu'une telle propriété peut être un moyen de distinguer un individu (ou un ensemble d'individus) d'autres individus. Par exemple, on distingue les carrés des rectangles par le fait que ces quadrilatères ont une largeur égale à leur longueur. Aussi, la classe *carré* peut-elle être distinguée de sa sur-classe *rectangle* par la propriété $largeur = longueur$ portant sur les deux attributs $longueur$ et $largeur$.

Dans le modèle Shirka, la solution adoptée pour associer une telle propriété à une classe est d'associer la facette \$à-vérifier à un attribut prédéfini (l'attribut *lui-même*) qui a pour valeur l'instance elle-même. Le prédicat indiqué par la facette, à la différence de celui donné par la facette

$cond, n'est pas unaire mais n-aire et peut donc opérer sur divers attributs de l'objet.

```
{ carre
   sorte-de =     rectangle
   lui-même $à-verifier { egalite
              x $var <- largeur ;
              x $var <- longueur } ;
   largeur ;
   longueur }
```

```
{ egalite
   sorte-de = predicat ;
   nom-fct $valeur egalite ;
   x     $un reel ;
   y     $un reel ;
   epsilon $un reel
      $defaut 1.0e-8 }
```

```
(de egalite (inst)
   (<= (abs (- (val? 'x)
              (val? 'y))
          (val? 'epsilon)))
```

TAB. 3.9 - : La classe carré en SHIRKA : par l'intermédiaire de la facette *$à-vérifier*, on exprime la propriété des carrés. La facette décrit le passage de paramètres pour le prédicat *égalité* représenté par une classe qui réfère à une fonction LE-LISP *égalité* dont le code figure en partie droite.

Cette solution, applicable dans tous les SRPO disposant d'une telle facette, n'est que partiellement satisfaisante.

D'abord, elle implique de définir un attribut particulier – l'attribut *lui-même* – que chaque objet possède et qui lui fait référence. Si bien que la propriété est exprimée par une facette dans la définition d'un attribut alors qu'elle est propriété de la classe.

Ensuite, la facette utilisée ne garantit qu'une vérification dynamique de la propriété établie. De sorte que, ce n'est seulement lorsque l'information disponible est suffisante – les attributs impliqués ont *tous* une valeur – que le prédicat est activé pour tester la validité des valeurs proposées.

Statiquement, aucune information n'est déduite de cette propriété. Or, dans l'exemple choisi, il est déjà clair que seules les valeurs communes aux domaines des attributs *largeur* et *longueur* conviennent pour former des rectangles de cotés identiques. Autrement dit, une conséquence de la propriété est que la description d'un carré, pour être cohérente avec la propriété, doit présenter les attributs *largeur* et *longueur* avec le même domaine. Rien de tel n'est assuré par la seule facette $à-vérifier.

De plus, dans les instances incomplètes déclarées par l'utilisateur comme carrés, il suffit de connaître la valeur de l'attribut *largeur* (resp. *longueur*) pour déduire celle de l'attribut *longueur* (resp. *largeur*). La facette $à-vérifier n'a pour fonction que le test et ne peut inférer aucune valeur.

Aussi, confier la vérification d'une propriété à un prédicat comme le fait SHIRKA nous semble limitatif et ne rend pas compte de toute la portée de l'existence de cette propriété.

Cette constatation provient de l'interprétation que nous faisons de la propriété et qui est différente manifestement de celle de SHIRKA ou de tout autre SRPO qui permettrait l'expression d'une propriété impliquant plusieurs attributs à partir d'un prédicat.

Nous considérons que la propriété fait partie intégrante de la définition de la classe (et non de l'attribut) et que le fait d'avoir *attaché* l'instance à la classe doit *contraindre* son contenu à se plier à cette propriété. Aussi, dès lors qu'une propriété implique plusieurs attributs, nous considérons qu'elle doit revêtir la forme d'une relation (ou contrainte) entre ces attributs et non plus celle d'un simple prédicat.

En tant que contrainte, la propriété possède alors potentiellement deux aspects qui ne sont pas pris en compte par un prédicat :

- au niveau dynamique, le pouvoir inférant qui peut être celui de la relation, et,
- au niveau statique, le maintien de la cohérence des domaines des attributs impliqués dans la relation.

Une relation entre n variables a un pouvoir inférant ou est *productive* lorsque la donnée des valeurs de $n-1$ variables permet d'inférer la valeur manquante de la n-ième variable.

Considérons la propriété $a = b$ sur deux attributs a et b de domaine respectif $[6, 8]$ et $[3, 7]$. Si on associe à b la valeur 6 alors, en raison de la présence de la propriété, l'attribut a *doit* prendre la valeur 6 également (il y a inférence dynamique de valeur). Par contre, si on associe la valeur 4 à b, l'inférence échoue car l'attribut a n'a pas de valeur qui permette de vérifier la condition... Ceci était prévisible : seules les valeurs 6 et 7 peuvent convenir à a et b (il y a ici déduction statique).

La productivité d'une relation est donc liée, d'une part à l'opérateur de comparaison qui la compose – une égalité est potentiellement plus productive qu'une inégalité, par exemple – et, d'autre part, aux valeurs contenues dans les domaines des variables impliquées – une inégalité peut en certains cas être productive.

Pour parvenir dans les SRPO à l'inférence dynamique de valeurs en cas de propriété productive, des méthodes peuvent être utilisées. En effet, on peut concevoir que pour des propriétés basées sur des égalités, il est possible d'associer à chacun des attributs impliqués, une méthode de calcul de sa valeur en fonction des autres attributs. On rend ainsi compte de la *multi-directionnalité* de la propriété. Par exemple, dans SHIRKA, il s'agirait de mettre en place un ensemble de méthodes qui permette de simuler le caractère productif de la relation. Ainsi, pour la classe carré, on peut associer une méthode à l'attribut *largeur* qui lui affecte la valeur de l'attribut *longueur*, et réciproquement. Cependant, cette simulation est peu commode car elle oblige à associer *localement* à tout attribut la méthode déduite de la propriété. En quelque sorte, la propriété est dispersée.

Simuler le caractère inférant d'une propriété productive par des méthodes n'est donc pas très satisfaisant car cela induit une certaine redondance et un éparpillement contraire à l'un des principes premiers de la RPO qui est de regrouper les descriptions [Gensel93a]. De plus, cette simulation nécessite la mise en place d'un réseau de dépendances (solution de type TMS) entre les attributs des méthodes afin que le calcul soit relancé lorsque l'un des attributs-arguments d'une méthode est modifié.

Quant à la cohérence des domaines des attributs impliqués dans les propriétés, elle peut requérir la modification de ces domaines et donc, des types des attributs. Dans SHIRKA, la cohérence de la propriété n'est pas assurée. Dans le pire des cas (celui où la propriété est incohérente avec la définition des attributs), ce n'est que dynamiquement et après avoir essayé toutes les possibilités que l'utilisateur se rendra compte de celle-ci. Par exemple, que penser de la propriété $a = b$ dans une classe où le domaine de a est $[2, 4]$ et celui de b, $[5, 6]$?

De même, conserver des valeurs incohérentes dans les domaines peut occasionner des évaluations inutiles du prédicat associé à la propriété.

Dans SHIRKA, ni le pouvoir inférant d'une propriété, ni le maintien de la cohérence des domaines des attributs impliqués ne sont pris en compte. Simplement, parce que la propriété est vue comme une condition *stricto sensu* dont on s'assure qu'elle est ou n'est pas vérifiée.

Dans TROPES, nous souhaitons restituer à ces propriétés le pouvoir inférant qui peut être le leur *dynamiquement*, et la cohérence sur les domaines des attributs qui peut s'établir *statiquement*. Ces propriétés deviennent désormais des *contraintes* dans TROPES.

3.6 Conclusion

Ce chapitre a présenté les divers moyens de représentation et d'exploitation de la connaissance offerts par le modèle TROPES. Les deux entités qui distinguent TROPES des autres SRPO sont le concept et le point de vue. Elles obligent le concepteur d'une base de connaissances à réaliser une partition du domaine à modéliser, mais permettent en retour d'éviter des conflits d'héritage et offrent une lisibilité accrue des connaissances. Les facettes de TROPES sont essentiellement dédiées à la représentation des types des attributs. La classification est le mécanisme principal d'inférence. Pour une instance, elle informe sur l'ensemble des classes auxquelles il est possible, impossible ou non encore connu de l'attacher. Quant aux filtres, ce sont des pseudo-classes employées pour émettre des requêtes ou décrire des types d'attributs. Autour de ce noyau, trois extensions de TROPES sont développées :

- Un module de gestion de types, METÉO, qui permet de définir des types abstraits et gère les types des entités de représentation afin d'accélérer la vérification de type et la classification d'instance ou de classe.

- Un modèle de tâches dans lequel l'expression et l'activation de tâches se font à l'aide des entités de représentation et des mécanismes d'exploitation (instanciation, classification) de TROPES.
- Un module de raisonnement hypothétique qui permet de suivre l'évolution d'une base de connaissances grâce à la production d'hypothèses.

Bien qu'assez complet, le noyau de TROPES ne permet pas d'exprimer des propriétés dans les classes mettant en jeu plusieurs attributs. La solution notamment proposée par SHIRKA, qui consiste à associer un prédicat de test à cette propriété n'est pas satisfaisante : la propriété n'est pas réellement considérée comme faisant partie de la définition de la classe mais plutôt comme une condition. Afin qu'elle joue un rôle définitionnel, il faut en faire une *contrainte* pour en préserver le caractère multi-directionnel, éventuellement inférentiel, et imposer le maintien de la cohérence sur les attributs impliqués. C'est le point de départ de notre étude qui doit permettre, entre autres, la gestion de contraintes sur les attributs d'une classe.

Avant de nous intéresser aux diverses formes de ce raisonnement par contraintes dans le contexte d'un SRPO tel que TROPES, nous devons puiser les enseignements nécessaires à la réalisation de notre objectif, et nous intéresser aux tenants et aux aboutissants de la programmation par contraintes.

Chapitre 4

Contraintes

Le chapitre précédent a présenté un système de représentation de connaissances appelé TROPES et a mis en avant l'inadaptation de la gestion par des prédicats de propriétés entre attributs dans les objets. Nous avons conclu sur la nécessité de considérer ces propriétés comme des *contraintes* et nous nous proposons d'intégrer à TROPES les outils permettant un raisonnement sous contraintes. Aussi, dans ce chapitre, nous nous intéressons au domaine d'où cette notion de *contrainte* est issue : la *Programmation Par Contraintes* (PPC).

Il suffit de pénétrer un peu ce domaine pour s'apercevoir de sa vaste étendue. Cependant, l'objectif de coupler les deux paradigmes – Représentation de connaissances Par Objets et Programmation Par Contraintes – doit nous permettre de nous orienter dans la diversité des langages et techniques qui composent la PPC vers des choix répondant aux attentes de la RPO.

Après une présentation générale de la PPC (*cf.* section 4.1), nous abordons la théorie des Problèmes de Satisfaction de Contraintes (CSP[1]) (*cf.* section 4.2). Un CSP est défini comme un ensemble de contraintes (ou relations) portant sur des variables ayant chacune un domaine de valeurs associé. Le but de la satisfaction de contraintes est de déterminer les solutions de ces CSP.

Nous donnons les définitions relatives à cette théorie et nous intéressons aux diverses techniques et algorithmes de *maintien de la consistance* (*cf.* section 4.3) et de *satisfaction* (*cf.* section 4.4) des CSP. La plupart des résultats établis par cette théorie concerne des CSP binaires dont les variables ont un domaine fini de valeurs. Dans la perspective d'une intégration de ces principes dans un SRPO, l'étude des *CSP à intervalles* (ICSP), adaptés au traitement des CSP à domaines continus et infinis, nous semble alors pertinente (*cf.* section 4.5). Les notions théoriques sont ici similaires, ce qui change c'est l'approche dans la maintenance et la résolution des problèmes.

Toujours dans un souci d'anticiper les exigences d'un SRPO dynamique tel que TROPES où les valeurs des attributs, mais aussi les entités de représentation (concept, points de vue, classes, attributs, instances), peuvent être à tout moment ajoutées ou retirées d'une base de connaissances, il nous semble indispensable de nous pencher sur l'étude des *CSP dynamiques* (DCSP, *cf.* section 4.6).

À l'issue de ce chapitre, nous aurons une idée des types de CSP les mieux adaptés à notre problématique, mais aussi des techniques à mettre en œuvre pour mener à bien notre objectif.

4.1 La programmation par contraintes

4.1.1 Généralités

Dans le domaine de la résolution de problèmes, la notion de *contrainte* désigne un énoncé déclaratif exprimant une relation entre les données d'un problème. Une contrainte constitue donc un élément de représentation de la connaissance relative à un problème.

[1] Nous conserverons l'abrégé anglais de Constraint Satisfaction Problems.

La PPC est l'outil informatique qui permet d'exprimer dans un formalisme déclaratif les contraintes qui décrivent un problème, et qui fournit les algorithmes capables de le résoudre, lorsque cela est possible. La programmation par contraintes allie donc représentation de connaissances et techniques algorithmiques. Intuitivement, le terme de *programmation par contraintes* fait référence à l'utilisation de systèmes offrant à la fois :

1. un langage permettant de formuler des problèmes en termes de contraintes,
2. des algorithmes et des heuristiques permettant de résoudre ces problèmes.

Les divers langages de PPC existants peuvent être distingués :

- par le type des données – et donc des contraintes – qu'ils sont capables de gérer (contraintes sur des nombres entiers, réels ou flottants, rationnels, des booléens, des ensembles, des objets...) ;
- par la caractérisation des ensembles de valeurs possibles (appelés *domaines*) associés aux données des problèmes. Les domaines pris en compte peuvent être *finis* ou *infinis*, *discrets*, *denses* ou *continus* ;
- par la possibilité offerte à l'utilisateur d'étendre l'ensemble des contraintes proposé ;
- par leur langage ou système hôte (PPC logique, PPC concurrente, PPC orientée-objet...).

Les origines de la programmation par contraintes remontent au début des années soixante, avec le système SKETCHPAD [Sutherland63], une interface graphique capable de résoudre des contraintes géométriques. Il faudra attendre cependant le milieu des années soixante-dix pour que, dans le domaine de l'Intelligence Artificielle, soit forgée une théorie autour des *Problèmes de Satisfaction de Contraintes (CSP)*. À cette époque, sont apparues la notion de *réseaux de contraintes* [Montanari74, Mackworth77] d'un côté, et les premières techniques de *consistance* [Waltz72] de l'autre. Ces travaux sont à la source des différentes voies de recherche ouvertes vers la consistance des réseaux de contraintes, les techniques de résolution, l'étude des graphes de contraintes associées, les heuristiques visant à accélérer la détermination d'une solution, la satisfaction partielle des CSP, l'étude des CSP dynamiques, etc.

Ces voies de recherche sont toujours explorées aujourd'hui, notamment parce qu'il n'existe sans doute pas de méthode universelle de résolution, et que, pour un CSP particulier, on peut espérer trouver une méthode nouvelle et mieux adaptée que tout autre résolution, basée sur la spécificité même de ce CSP.

Si la théorie des problèmes de satisfaction de contraintes est une branche de l'Intelligence Artificielle, c'est sans doute parce que les contraintes ont infiltré bon nombre de domaines de l'IA. C'est aussi parce que l'une des vocations de l'IA est de s'attaquer à des problèmes réels, même s'ils sont par essence combinatoires ou NP-complets. Les techniques proposées par la théorie des CSP vont permettre d'élaguer l'espace de recherche de solutions initial, de considérer sa structure, de réduire sa profondeur, de déterminer un parcours heuristique performant afin de déterminer plus rapidement la ou les solutions [Mayoh94].

En IA, les contraintes sont apparues dans des domaines aussi divers que la conception de circuits [Sussman et al.80], la vision [Waltz72], l'aide à la décision [Benjamin et al.93] et au diagnostic [Geffner et al.87], la planification [Stefik81b, Stefik81a], le raisonnement temporel [Allen83, Dechter et al.91], le maintien de cohérence... Mais les domaines d'application de la PPC s'étendent bien au delà des confins de l'IA. Ainsi, les langages de programmation par contraintes sont souvent destinés à résoudre des problèmes de gestion d'emplois du temps, d'ordonnancement de tâches, d'allocation de ressources, de maintien de la cohérence...

En parallèle aux avancées de la théorie des CSP, des langages ou systèmes de programmation par contraintes sont apparus. Les premiers (SKETCHPAD [Sutherland63], ALICE [Laurière78], CONSTRAINTS [Sussman et al.80], THINGLAB [Borning81], BERTRAND [Leler88]...) furent des langages issus de laboratoires de recherche et ont servi de plate-formes de validation à l'étude de la théorie des problèmes de satisfaction de contraintes. Cependant, de plus en plus, fleurissent des langages de PPC commercialisés (CHIP [Dincbas et al.88], CHARME [Oplebodu89], PROLOG III [Colmerauer90], PECOS (ILOG-SOLVER) [Ilog92b]...) qui témoignent de la demande de l'industrie pour de tels outils et fait la preuve à la fois de leur puissance et de leur facilité d'utilisation. Ces langages non seulement intègrent des techniques propres à la théorie des problèmes de satisfaction de contraintes, mais également des techniques issues d'autres domaines scientifiques comme la recherche opérationnelle (algorithme du Simplexe [Dantzig63]) ou de l'analyse numérique (méthode de Newton).

Depuis 1988, deux groupes de langages de programmation par contraintes se distinguent : la Programmation Logique par Contraintes (ou PLC) et la Programmation par Contraintes avec Objets (ou PCO). Cette dernière classe de langages nous intéresse plus particulièrement ici et sera étudiée en détail au chapitre 5. Avant d'aborder la théorie des CSP et afin d'avoir une idée des possibilités offertes par un langage de PPC, nous présentons les grandes lignes de la PLC.

4.1.2 Programmation logique par contraintes

La Programmation Logique par Contraintes est une classe de langages déclaratifs de programmation qui combinent non-déterminisme et résolution de contraintes [Hentenryck91].

L'idée de la PLC est d'augmenter un langage à la Prolog avec les mécanismes de résolution de contraintes. Notamment, il s'agit d'introduire des opérateurs prédéfinis qui vont permettre d'enrichir la sémantique des termes ; par exemple, d'exprimer et de manipuler des expressions arithmétiques [Frühwirth et al.92]. Pour cela, il faut remplacer le moteur d'inférence d'un système de programmation logique qu'est *l'unification* par des algorithmes intégrés dans un *résolveur de contraintes* chargé de la propagation et de la satisfaction des contraintes.

Ce principe qui préside à la création de tout langage de programmation logique avec contraintes, a été énoncé par Jaffar et Lassez [Jaffar et al.87] sous le nom de *schéma* CLP(X) où X a été depuis instancié par divers domaines de calcul (réels CLP($\Re$), rationnels CLP($\mathbb{Q}$), entiers relatifs CLP($\mathbb{Z}$)). Pour chaque domaine X de calcul traité, CLP(X) fournit les opérations algébriques associées (par exemple, intersection ensembliste, conjonction booléenne, addition...). Des relations spécifiques au domaine de calcul sont également disponibles et forment les contraintes (égalité ensembliste, entre booléens, opérateurs de comparaison arithmétiques...).

4.2 Problèmes de Satisfaction de Contraintes

Le but de cette section est de rappeler les principales définitions de la théorie des problèmes de satisfaction de contraintes ainsi que les diverses techniques de consistance et de résolution.

4.2.1 Définitions

Définition 1 *Un problème de satisfaction de contraintes ou CSP [Montanari74, Mackworth77] est défini par la donnée d'un triplet (X, D, C) où :*

- $X = \{x_1, \ldots, x_n\}$ *est un ensemble fini de variables,*
- $D = \{d_1, \ldots, d_n\}$ *est l'ensemble fini des domaines de valeurs des variables de X (à chaque variable x_i de X est associé un domaine d_i de D),*
- $C = \{c_1, \ldots, c_m\}$ *est un ensemble fini de contraintes (chaque contrainte $c_i(x_{i1}, \ldots, x_{ik})$ de C porte sur un sous ensemble $\{x_{i1}, \ldots, x_{ik}\}$ de X). $c_i(x_{i1}, \ldots, x_{ik})$ représente en extension un sous-ensemble du produit cartésien $d_{i1} \otimes \ldots \otimes d_{ik}$). Chaque élément de ce sous-ensemble est un ik-uplet $(v_{i1}, \ldots, v_{ik})$ (une valeur pour chaque variable x_{ip} de c_i) qui représente une solution pour la contrainte c_i.*

Exemple 1 *On considère le CSP(X, D, C) numérique représenté par le système de contraintes suivant :*

$$\begin{cases} c_1: & x_1 < x_2 + x_3 \\ c_2: & x_5 = x_3 + x_4 \\ c_3: & x_5 = x_6 \\ c_4: & x_6 - x_5 \geq x_2 - x_7 \end{cases} \quad \begin{array}{l} \textit{avec } X = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7\} \\ \quad D = d_1, d_2, d_3, d_4, d_5, d_6, d_7 \textit{ avec } d_i \subseteq I\!N \\ \quad C = \{c_1, c_2, c_3, c_4\} \end{array}$$

Le type des variables d'un CSP détermine le type du CSP. On parle de CSP numérique sur des nombres (entiers, rationnels ou réels), booléen, ensembliste, etc.

Le domaine d'une variable peut être *fini* (par exemple, $\{1, 2, 4\}$) ou *infini* (par exemple, l'ensemble des entiers, l'intervalle réel $[2.7, 5.3]$).

Une contrainte $c_i(x_{i1}, \ldots, x_{ik})$ est une relation que les variables $x_{i1}, \ldots, x_{ik}$ de la contrainte doivent vérifier. Elle peut être donnée en :

- *extension* comme l'ensemble des ik-uplets $(v_{i1}, \ldots, v_{ik})$ (une valeur pour chaque variable x_{ij} de c_i) qui satisfont la contrainte c_i. De tels $(v_{i1}, \ldots, v_{ik})$ sont définis comme des *instanciations partielles* de X [Dechter92].
- *intention* sous la forme d'une relation décrite à l'aide des opérateurs définis sur le type des variables de X. Par exemple, une comparaison entre deux expressions arithmétiques pour un CSP numérique.

Exemple 2 *Soit le CSP (X, D, C) où*
$X = \{carroserie, habitacle, enjoliveurs\}$,
$D = \{\{“noir”, “rouge”, “blanc”\}, \{“noir”, “rouge”, “blanc”\}, \{“noir”, “rouge”, “blanc”\}\}$, *et*
C est donné en intention par les deux contraintes symboliques suivantes[2] *:*

c_1 *: l'habitacle est de couleur plus foncée que la carrosserie*

c_2 *: les enjoliveurs ont la même couleur ou sont plus foncés que la carrosserie*

c_3 *: les enjoliveurs et l'habitacle ont la même couleur*

C est donné en extension par :

$C = \{c_1(habitacle, carrosserie) = \{(“rouge”, “blanc”), (“noir”, “blanc”), (“noir”, “rouge”)\},$

$c_2(enjoliveurs, carrosserie) = \{(“rouge”, “blanc”), (“rouge”, “rouge”), (“rouge”, “noir”),$
$(“noir”, “blanc”), (“noir”, “rouge”), (“noir”, “noir”)\}$

$c_3(enjoliveurs, habitacle) = \{“noir”, noir”), (“rouge”, “rouge”), (“blanc”, “blanc”)\}\}$

Le *graphe associé* d'un CSP permet d'en donner une représentation graphique. Ce graphe peut être :

- un hypergraphe dont les nœuds représentent les variables du CSP et les arêtes représentent les contraintes. Cette représentation est plutôt adaptée aux CSP binaires ;
- un graphe biparti dont les nœuds représentent à la fois les variables et les contraintes. Toute arête de ce graphe relie donc un nœud circulaire *variable* à un nœud rectangulaire *contrainte*. Cette représentation est préférée pour les CSP n-aires (avec n > 2).

Nous donnons à présent les définitions d'instanciation partielle *localement et globalement consistante.*

Définition 2 (d'après [Dechter92]) *Par rapport à un CSP(X, D, C), une instanciation partielle (affectation de valeurs à un ensemble de variables Y tel que $Y \subseteq X$) est dite localement consistante si elle satisfait toute contrainte $c_i(x_{i1}, \ldots, x_{ik})$ telle que $\{x_{i1}, \ldots, x_{ik}\} \subseteq Y$.*

Définition 3 (d'après [Dechter92]) *Une solution du CSP (X, D, C) est une instanciation de X localement consistante. Cette instanciation est dite globalement consistante.*

Définition 4 (d'après [Dechter92]) *Une instanciation partielle est globalement consistante si elle peut être étendue à une solution.*

Le but de la satisfaction d'un CSP est de déterminer les solutions de ce problème, si elles existent. On peut se contenter d'une seule solution ou les rechercher toutes, ou encore chercher la

[2] *Exemple inspiré de [Bessière92].*

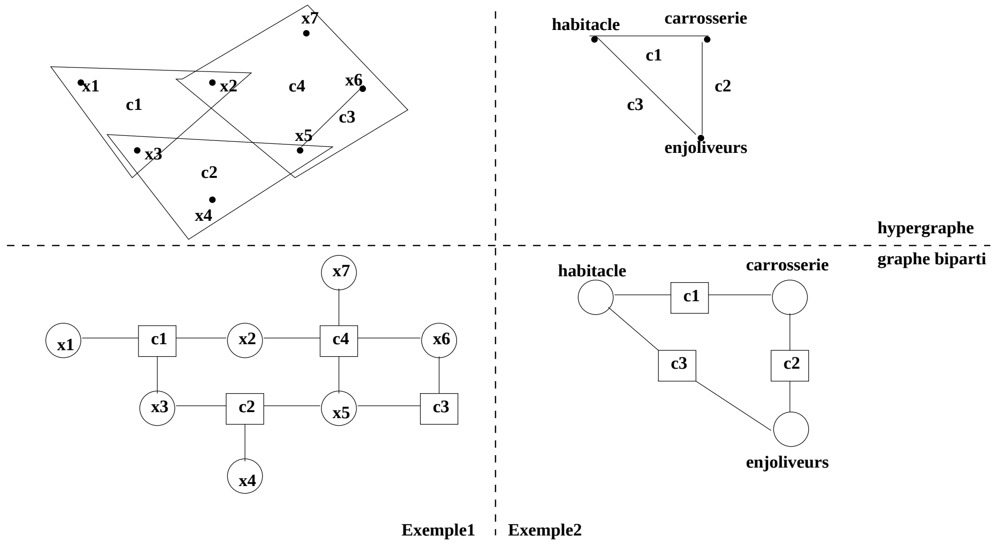


FIG. 4.1 - : Deux représentations pour les deux CSP donnés en exemple : en haut l'hypergraphe, en bas le graphe biparti associé

meilleure.

La recherche de solution est *a priori* une opération fortement combinatoire. En effet, si on considère l'espace de recherche de solutions associé à un CSP(X, D, C) à domaines finis, c'est un arbre qui contient toutes les instanciations partielles que l'on peut former à partir de X et de D. Il est donc composé de n niveaux où n est la taille de l'ensemble X des variables du CSP, et a une largeur au plus égale à d^n où d est la taille du plus grand domaine de D. Aussi, le parcours de cet arbre peut nécessiter un temps d'exécution exponentiel en la taille des données du CSP avant de déterminer si, oui ou non, une solution existe (*cf.* figure 4.2).

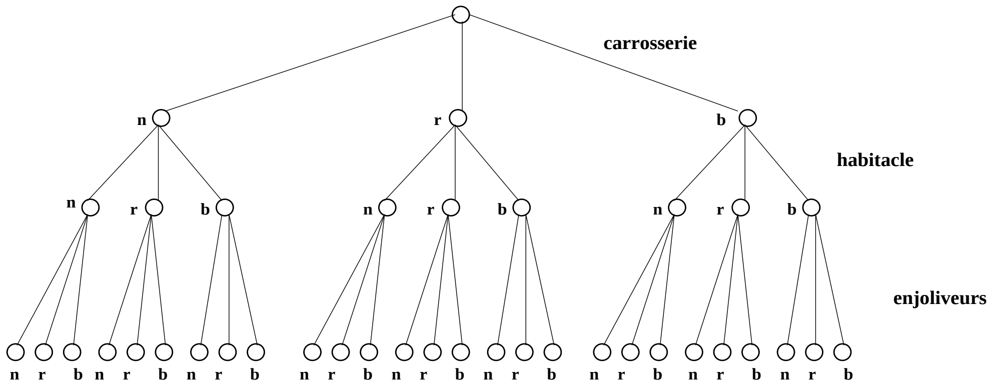


FIG. 4.2 - : Espace de recherche de solutions associé au second exemple, dans l'ordre d'instanciation *carrosserie, habitacle, enjoliveurs* avec les couleurs *noir (n), rouge (r), blanc (b).*

Il existe plusieurs techniques de recherche de solutions pour les CSP à domaines finis (*cf.* section 4.4). Ces techniques construisent un arbre de recherche de solutions dont les nœuds sont des états et les arêtes les actions – *choix d'une variable (à chaque niveau), choix d'une valeur (au même niveau)* – permettant d'atteindre ces états. Avant de se lancer dans une énumération du domaine de chaque variable pour construire une instanciation localement puis finalement globalement consistante, il est préférable de chercher à réduire la taille de l'espace de recherche de solutions. Pour ce faire, on opère un filtrage sur les domaines des variables afin de les rendre *consistants* (*cf.* section 4.3).

4.2.2 Notations

Dans la suite de l'exposé, pour tout CSP (X, D, C), nous introduisons les fonctions suivantes :

- *dom* : $X \rightarrow D$, la fonction qui à toute variable x de X associe son domaine $d = dom(x)$ de D ;
- *var* : $C \rightarrow 2^X$, la fonction qui à toute contrainte c de C associe l'ensemble $var(c)$ des variables de X sur lesquelles elle porte ;
- *contr* : $X \rightarrow 2^C$, la fonction qui à toute variable x de X associe l'ensemble $contr(x)$ des contraintes de C qui portent sur elle.

4.3 Maintenance de contraintes

Avant de chercher les solutions d'un CSP, une technique couramment employée est le filtrage des domaines. Elle consiste à réduire le CSP (X, D, C) en un CSP (X, D', C) *équivalent*, c'est-à-dire portant sur le même ensemble de n variables et possédant le même ensemble de n-uplets solutions [Tsang93] tel que le domaine dans D' de toute variable de X soit inclus ou égal à son domaine dans D.

Le principe de ce filtrage – appelé *maintenance de consistance* – est de supprimer des domaines les valeurs qui ne peuvent apparaître dans aucune des solutions. La *consistance* est la propriété que vérifie le CSP après application du filtrage associé.

En général, le filtrage constitue un prétraitement avant la recherche de solutions. Mais il peut aussi être appliqué durant la recherche même d'une solution. D'autre part, il est des cas particuliers de CSP pour lesquels ce prétraitement suffit à déterminer une solution. Enfin, si ce filtrage a pour effet de vider un domaine, alors une variable n'a plus de valeur qui puisse figurer dans une instanciation localement (et globalement) consistante ; il s'ensuit que le CSP n'a pas de solution, le filtrage peut donc être stoppé.

4.3.1 Arc-consistance

Il existe plusieurs consistances qui se distinguent par la puissance (ou degré) de filtrage et donc, par leur coût. Les *consistances de nœud, d'arc et de chemin*[3] ont été introduites par Alan Mackworth [Mackworth77]. Généralisant ces notions aux CSP n-aires, la notion de k-consistance a été introduite par Eugene Freuder [Freuder82].

Définition 5 *Un CSP est **nœud-consistant** ssi :*
$\forall x_i \in X, \forall\ v_i \in d_i,$
$\forall c_k \in C,$ *telle que* $var(c_k) = x_i$
v_i *satisfait* c_k

Définition 6 *Un CSP est **arc-consistant** ssi :*
$\forall x_i \in X, \forall v_i \in d_i, \forall\ d_i \neq \emptyset$
$\forall c_k(x_{k1}, \ldots, x_{kp}) \in C,$ *telle que* $x_i \in \{x_{k1}, \ldots, x_{kp}\}$
$\exists v_{k1} \in d_{k1}, \ldots, \exists v_{kp} \in d_{kp},$ *telles que* $(v_{k1}, \ldots, v_{kp}) \in c_k$

Définition 7 *Un CSP (binaire) est **chemin-consistant** ssi :*
$\forall$ *chemin* $(x_0, x_1, \ldots, x_m)$ *dans le graphe où les arêtes sont des contraintes binaires,*
$\forall\ v_0 \in d_0, \ldots, \forall\ v_m \in d_m,$
l'instanciation de deux valeurs $(x_0 = v_0, x_m = v_m)$ *est localement consistante et* $\forall\ k \in [1, m-1],$
$\exists\ v_k \in d_k$ *telle que toute instanciation de deux valeurs* $(x_{k-1} = v_{k-1}, x_k = v_k)$ *soit consistante.*

Définition 8 *Un CSP est **k-consistant*** $\forall\ k \in [1, n]$ *ssi :*
$\forall\ (x_1 = v_1, \ldots, x_{k-1} = v_{k-1})$*, instanciation partielle localement consistante de k-1 variables de X,*

[3] Par anglicisme, on parle aussi de nœud, d'arc et de chemin-consistance.

$\forall\ x_k \in X$ *telle que* $x_k \notin \{x_1, \ldots, x_{k-1}\}$,
$\exists\ v_k \in d_k$ *telle que* $(x_1 = v_1, \ldots, x_{k-1} = v_{k-1}, x_k = v_k)$ *soit localement consistante.*

La consistance de nœud s'applique aux contraintes unaires et n'a donc que peu d'intérêt. La consistance de chemin ne s'applique qu'aux CSP binaires. Plus forte que l'arc-consistance, elle est aussi plus coûteuse. De ces consistances, l'arc-consistance est celle qui est la plus utilisée : elle s'applique aux CSP de toute arité et affiche un bon rapport coût/efficacité en moyenne. La k-consistance est une définition généralisée des divers degrés de consistance atteignables.

Il existe à ce jour six versions d'algorithmes permettant d'établir l'arc-consistance d'un CSP, nommés AC-1 [Waltz72], AC-2, AC-3 [Mackworth77], AC-4 [Mohr et al.86], AC-5 [Deville et al.91] et AC-6 [Bessière et al.93]. L'algorithme de complexité optimale (*cf.* tableau 4.1) dans le pire des cas est AC-4 mais il s'avère moins efficace que AC-3 dans la plupart des cas [Wallace93]. AC-5 est une généralisation de AC-3 et de AC-4 qui présente une complexité de $O(ea)$ (e est le nombre de contraintes, a la taille du plus grand domaine) pour deux classes de contraintes (les contraintes fonctionnelles et les contraintes monotones). AC-6 diminue la place mémoire nécessaire par rapport à AC-4 pour une complexité égale dans le pire des cas mais meilleure en moyenne.

n = nombre de variables ; e = nombre de contraintes ; a = taille du plus grand domaine			
Consistance	Algorithme	Complexité	Taille mémoire
nœud-consistance	NC-1	$O(an)$	$O(an)$
arc-consistance	AC-1 [Mackworth77]	pire des cas : $O(a^3ne)$	$O(e + na)$
	AC-3 [Mackworth77]	borne inférieure : $\Omega(a^2e)$ borne supérieure : $O(a^3e)$	$O(e + na)$
	AC-4 [Mohr et al.86]	pire des cas : $O(a^2e)$	$O(a^2e)$
	AC-6 [Bessière94]	pire des cas : $O(a^2e)$	$O(ae)$
cohérence de chemin	PC-1 [Mackworth77]	pire des cas : $O(a^5n^5)$	$O(n^3a^2)$
	PC-2 [Mackworth77]	borne inférieure : $\Omega(a^3n^3)$ borne supérieure : $O(a^5n^3)$	$O(n^3 + n^2a^2)$
	PC-4 [Han et al.88]	pire des cas : $O(a^3n^3)$	$O(n^3a^3)$

TAB. 4.1 - : Complexité et taille mémoire des algorithmes des diverses consistances (d'après [Tsang93])

L'idée qui se trouve à la base de tout algorithme d'arc-consistance est la suivante[4] : pour chaque contrainte (binaire) ou arc (i, j) du CSP, on élimine du domaine de i les valeurs v_i pour lesquelles il n'existe pas de valeur v_j du domaine de j – appelée *support* de v_i – telle que (v_i, v_j) soit une instanciation localement consistante pour (i, j). Il faut alors *propager* la suppression de la valeur v_i du domaine de i. En effet, il se peut que la valeur v_i soit le *support* d'une valeur v_k d'une variable v_k différente de v_j. Dans le cas où v_i est l'unique support de v_k dans le domaine de i, v_k sera, à son tour, supprimée. Les différents algorithmes d'arc-consistance se distinguent par leur finesse de choix dans la remise en cause des arcs candidats à une telle révision. Si AC-1 est qualifié de *naïf*, car il remet en cause tous les arcs de C, AC-3 (AC-2 en est une variante moins efficace) ne remet en cause que les arcs (k, i) où $k \neq i$ et $k \neq j$, seuls *a priori* concernés par la disparition de v_i. AC-4 met en œuvre des structures de données plus complexes pour connaître l'ensemble des supports d'une valeur d'une variable dans un domaine, le nombre de supports d'une valeur d'une variable dans le domaine de la variable adjacente, et enfin si une valeur a déjà été supprimée d'un domaine ou non. AC-6 est supérieur à AC-4 dans la recherche d'un autre support pour tout v_k en instaurant un ordre sur le domaine de k et en ne conservant qu'une seule valeur support. Les algorithmes d'arc-consistance sont présentés généralement dans le cadre des CSP binaires, GAC-4 [Mohr et al.88] étend AC-4 aux CSP n-aires. Mais de l'aveu même de ses concepteurs, il est peu adapté à des CSP aux nombres de variables et de contraintes importants.

[4] Elle est donnée ici dans le cadre de CSP binaires.

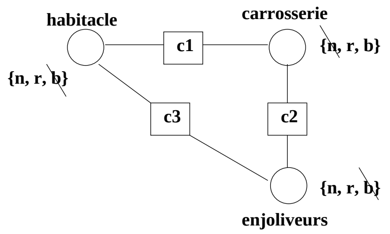


FIG. 4.3 - : Application de l'arc-consistance sur les domaines des variables du CSP du second exemple. La valeur "noir" dans le domaine de *carrosserie* n'a pas de valeur support, elle est donc éliminée. La valeur "blanc" dans le domaine de *habitacle* subit le même sort et entraîne la suppression de la valeur "blanc" dans le domaine de *enjoliveurs*.

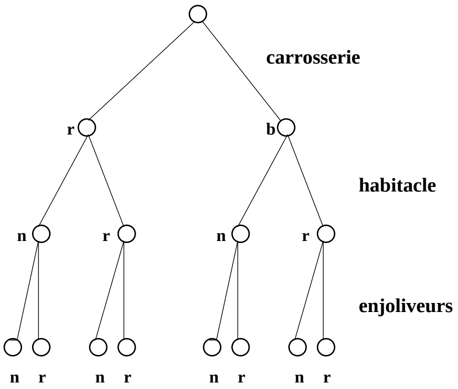


FIG. 4.4 - : L'espace de recherche de solutions après application de l'arc-consistance

procédure AC-3(X, D, C)
début
  Q $\leftarrow \{(i,j) \mid C_{i,j} \in C\}$
  /* arcs issus des contraintes de C*/
  tantque Q $\neq \emptyset$ **faire**
    enlever un élément de Q
    si Révise((i,j),X,D,C) **alors**
      Q $\leftarrow$ Q $\cup \{(k,i) \in C \wedge k \neq i \wedge k \neq j\}$
      finsi
  fintantque
fin

TAB. 4.2 - : La procédure AC-3

fonction Révise$((i,j), X, D, C) \rightarrow$ booléen
début
  Révise $\leftarrow$ faux
  pour tout $v_i \in dom(i)$ **faire**
    si $\nexists\ v_j \in dom(j)$ tel que $C_{i,j}$ soit satisfaite
      $dom(i) \leftarrow dom(i) - \{v_i\}$
      Révise $\leftarrow$ vrai
    finsi
  finpour
fin

TAB. 4.3 - : La fonction Révise

4.4 Satisfaction de contraintes

Nous avons présenté dans la section précédente les divers filtrages applicables en préambule d'une recherche de solutions. Il n'existe que deux configurations terminales de ces filtrages qui soient instructives.

1. Si le filtrage a vidé un domaine (aucune valeur n'a de support) alors on peut conclure que le CSP n'a pas de solution ; il est inutile de lancer une procédure de recherche de solutions.
2. Si tous les domaines sont réduits à une seule valeur, alors le CSP admet une solution unique ; l'instanciation formée de chacune de ces valeurs est globalement consistante.

Dans le reste des cas (tous les domaines sont non vides, au moins un n'est pas un singleton), il est impossible de conclure à l'existence d'une solution. Il convient alors de mettre en œuvre des stratégies de recherche de solutions afin de déterminer s'il existe ou non des instanciations globalement consistantes.

4.4.1 Stratégies de recherche de solutions

On considère dans la suite un CSP (X, D, C) à n variables et m contraintes, la taille des domaines finis étant majorée par d.

Les opérations de base de la recherche d'une solution sont, d'une part, l'affectation à une variable non encore instanciée d'une valeur de son domaine et, d'autre part, le test de la validité de l'instanciation partielle réalisée jusqu'alors.

La technique la plus primaire et la moins efficace consiste à générer les instanciations des n variables et à tester pour chacune d'elles si elle est globalement consistante (coût de l'ordre de d^n).

Les autres méthodes, plus efficaces, se divisent en deux classes :

1. les méthodes à base de retour-arrière qui, en cas d'échec, remettent en cause une variable précédemment instanciée. Elles se distinguent par la pertinence dans le choix de la variable à remettre en cause, c'est-à-dire, par leur capacité à éviter que des affectations ayant déjà mené à un échec ne se reproduisent.
2. les méthodes à base de recherche en avant qui cherchent à propager les conséquences de l'instanciation courante en éliminant des domaines des variables non encore instanciées, les valeurs d'ores et déjà incompatibles avec la valeur choisie pour la variable courante. Elles se distinguent par la puissance du filtrage qu'elles opèrent sur les domaines des variables non encore instanciées.

4.4.1.1 Stratégies de retour-arrière

Le *Backtrack* ou retour arrière chronologique consiste à choisir, d'après un ordre établi, une valeur dans le domaine de la prochaine variable à instancier et de tester si elle forme avec les valeurs des variables précédemment instanciées une instanciation partielle localement consistante. Si tel est le cas, alors on passe au choix d'une valeur pour la prochaine variable à instancier. Si ce n'est pas le cas, alors on essaie une autre valeur du domaine de la variable en cours d'instanciation. Si toutes les valeurs du domaine ont été essayées sans succès, on remet en cause la variable précédemment instanciée, d'où le nom de la méthode.

L'inefficacité de cette méthode réside justement dans le choix de la variable à remettre en cause. En effet, dans le cas où il n'existe pas de contraintes entre la variable couramment instanciée (et en situation d'échec) et la variable précédemment instanciée, on reproduit les mêmes situations d'échec quelle que soit la valeur de la variable précédente puisque celle-ci n'est pas immédiatement responsable. Il faut donc effectuer un retour arrière plus judicieux ou/et éviter de reproduire des instanciations dont le résultat est connu.

Le *Backjumping* [Gaschnig79] est une amélioration du *Backtrack* (chronologique) car, en cas d'échec de l'instanciation de la variable courante x_i, il remet en cause, non pas la variable précédemment instanciée, mais la dernière variable (dans l'ordre d'instanciation) x_j avec laquelle x_i partage une contrainte. Le *Backjumping* s'inscrit dans les méthodes dites de *retour-arrière dirigé par les dépendances*.

Parmi les méthodes qui évitent la redondance d'instanciation, le *Backchecking* [Haralick et al.80] consiste à se rappeler que la valeur v_i choisie pour la variable x_i en cours d'instanciation est incompatible avec la valeur v_j choisie pour une variable x_j précédemment instanciée. Tant que x_j aura pour valeur v_j, v_i ne sera plus considérée.

Une amélioration du *Backchecking* est apportée par le *Backmarking* [Gaschnig77] qui, pour un niveau (ou variable) d'instanciation courant i, enregistre le plus haut niveau j (la variable x_j précède x_i) dans l'arbre de recherche vers lequel un retour-arrière doit être opéré, mais aussi le plus bas niveau k pour lequel on peut garantir que v_i est compatible avec toute valeur $v \in \{v_1, \ldots, v_k\}$ des variables $\{x_1, \ldots, x_k\}$. Ainsi, ni les situations d'échec, ni les instanciations fructueuses déjà rencontrées ne sont reproduites.

La méthode dite de retour-arrière dirigé par les conflits [Dechter86] consiste à apprendre et à enregistrer les causes de l'échec d'une instanciation pour, à la fois éviter des branches dans l'arbre de recherche menant à un échec, et revenir au point de choix le plus judicieux lors d'un échec. Notamment, là où le *Backjumping* n'est capable que de sauts répétés, cette méthode assure l'élagage de l'arbre de recherche : les sauts que répète le *Backjumping* n'ont plus lieu d'être car leur cause est supprimée. Pour parvenir à ses fins, la méthode gère un ensemble de conflits ou *nogoods* qui est, pour une variable x_k, une instanciation partielle des variables x_i précédemment instanciées, telle qu'il n'existe aucune valeur dans $dom(x_k)$ compatible avec cette instanciation. Des ensembles minimaux de conflits sont maintenus qui empêchent la reconsidération d'instanciations conflictuelles. Cependant, la détermination de ces ensembles minimaux est coûteuse et un bon compromis doit être trouvé pour la taille des ensembles conflits.

Les études pour l'amélioration de ces stratégies de retour-arrière sont toujours en cours. Le *Backtrack dynamique* [Ginsberg93] et les algorithmes hybrides proposés par Patrick Prosser en témoignent. Le premier reprend le principe du *Backtrack* dirigé par les dépendances [Stallman et al.77] en économisant les informations qui permettent de faire un retour arrière à l'endroit le plus judicieux dans l'arbre. Cependant, cette technique s'est montrée moins efficace que le *Backjumping* sur certaines heuristiques [Baker94]. Prosser a combiné plusieurs de ces techniques en des algorithmes dits hybrides (par exemple, le *Backmarking* avec le *Backjumping*). Dans [Prosser93], Prosser présente une amélioration du *Backjumping* dirigée par les conflits et proche de celle proposée par Dechter dans [Dechter90]. Il étudie également l'effet de l'élimination de valeurs une fois pour toutes dans les domaines (proche des techniques proposées dans [Dechter et al.88b], [Dechter90]). Si cette modification donne de bons résultats sur les stratégies de recherche en avant, elle n'est satisfaisante qu'en moyenne sur les approches par retour arrière : la suppression définitive de valeurs peut entraîner une dégradation des performances de ces techniques. Frost et Dechter [Frost et al.94a] ont récemment proposé une définition supplémentaire d'ensembles conflits, supérieure aux précédentes [Dechter90] qui accroît l'efficacité du *Backjumping* lorsqu'il est associé à une heuristique statique (largeur minimum) ou une heuristique dynamique d'ordonnancement des variables (à base de *first-fail principle*), notamment pour les problèmes difficiles [Cheeseman et al.91, Mitchell et al.92].

4.4.1.2 Stratégies de recherche en avant

Dans les approches dites *prospectives* ou *Lookahead* [Haralick et al.80], l'idée est de vérifier si la valeur choisie pour la variable en cours d'instanciation est supportée par les variables non encore instanciées, c'est-à-dire s'il existe des valeurs compatibles dans les domaines de ces variables.

Avec les approches de retour arrière, l'espace de recherche n'est pas modifié, simplement son parcours est amélioré. Ici, il l'est à chaque étape puisque le principe sous-jacent aux techniques prospectives est de réduire les domaines des variables non encore instanciées (donc supprimer des branches situées plus bas dans l'arbre de recherche de solutions) afin de détecter les instanciations qui ne peuvent être complétées vers une solution.

Le *Forward-Checking* est le premier de ces algorithmes. Lorsqu'une valeur compatible avec les instanciations réalisées jusqu'alors est choisie pour une variable, on élimine des domaines des variables non encore instanciées, les valeurs qui ne sont pas compatibles avec la valeur choisie. Lorsque l'un des domaines des variables non encore instanciées est vidé par ce filtrage et qu'il ne reste plus de valeur candidate pour la variable en cours d'instanciation, le *Forward-Checking* remet en cause, comme le Backtrack, la valeur de la variable précédemment instanciée.

Dans le *Full-Lookahead*, un filtrage plus puissant, mais aussi plus coûteux, est effectué afin d'établir l'arc-consistance du réseau formé par les variables non encore instanciées. Plus le filtrage opéré est puissant, plus le nombre de tests opérés est grand. En revanche, l'espace de recherche est diminué par le filtrage, et les filtrages suivants porteront donc sur des domaines réduits. Là encore, il faut trouver un bon compromis entre la puissance du filtrage appliqué et le temps de parcours de l'arbre de recherche.

4.4.2 Heuristiques d'ordonnancement

Un ordonnancement des variables, voire des valeurs, peut avoir des répercussions sensibles sur la rapidité avec laquelle les techniques de résolution exposées précédemment parviennent à une solution. Nous présentons ici les divers critères d'ordonnancement.

4.4.2.1 Ordre sur les variables

L'ordre dans lequel sont instanciées les variables lors des différentes méthodes de recherche de solutions influe directement sur leurs performances. Tout d'abord, à chaque ordonnancement correspond un arbre de recherche de solution (chaque arbre ayant potentiellement le même nombre de nœuds). Pour les techniques basées sur un retour arrière, c'est le nombre de nœuds effectivement explorés qui varie selon l'ordonnancement. Pour les techniques prospectives, certains ordonnancements révèlent plutôt les échecs ou encore élaguent davantage l'arbre de recherche que d'autres. Dans les deux cas, l'ordonnancement a des conséquences directes sur le nombre de branches explorées dans l'arbre. Il s'avère donc un facteur prépondérant dans la rapidité avec laquelle la technique employée est capable de déterminer une, voire toutes les solutions.

On parle d'heuristiques d'ordonnancement car il n'est pas sûr que le critère choisi pour ordonner les variables donne le meilleur ordonnancement possible – en termes de branches à explorer pour trouver une ou pour toutes les solutions – quelque soit le CSP considéré.

Il existe deux types d'ordonnancement :

1. l'ordonnancement statique, qui est opéré *avant* que la procédure de recherche ne soit amorcée. Ces ordonnancements exploitent la topologie du graphe associé au CSP. Parmi les plus répandus, on trouve :
   - l'*ordonnancement selon la largeur minimale* [Freuder82], qui conduit à l'instanciation des variables les plus contraintes d'abord et donc réduit le nombre de retour-arrière
   - l'*ordonnancement selon la cardinalité minimale* [Dechter et al.89a], qui consiste à choisir arbitrairement la première variable et de choisir successivement la variable qui partage le plus de contraintes avec les variables déjà choisies
   - l'*ordonnancement selon le degré maximal* [Dechter et al.89a], qui classe les variables de la plus contrainte à la moins contrainte (c'est une approximation de l'ordonnancement selon la largeur minimale).

   On peut également procéder *statiquement* à *l'ordonnancement selon la taille des domaines*. La recherche d'une solution peut amener à considérer d'abord les variables qui ont les plus grands domaines pour se diriger vers les branches les plus prometteuses [Jégou91]. Mais développer au minimum l'arbre de recherche doit présider à la recherche de toutes les solutions. Dans ce cas, le *first fail principle* [Haralick et al.80], qui consiste à considérer d'abord les variables qui ont le plus petit domaine, s'avère souvent satisfaisant [Hentenryck89].
2. l'ordonnancement *dynamique*, qui est opéré *à chaque fois* que la technique de recherche employée a besoin de choisir une nouvelle variable pour progresser. Dans ce cas, une heuristique telle que le *first fail principle* [Haralick et al.80] ou réarrangement dynamique [Purdom83] – choix de la variable parmi celles non encore instanciées ayant le plus petit domaine – peut être combinée avec une méthode prospective de recherche de solution (le *Forward-Checking*, par exemple) susceptible de modifier à chaque choix de valeur les domaines des variables non encore instanciées.

4.4.2.2 Ordre sur les valeurs

L'ordonnancement des valeurs peut également avoir une incidence sur le nombre de retours en arrière effectués par une procédure de recherche, et donc sur la rapidité à trouver une solution. Il s'agit ici de s'engager dans les branches les plus prometteuses. Un des ordonnancements les plus connus est la méthode des *conflits minimaux* [Minton et al.90]. À partir d'une instanciation totale et aléatoire des variables du CSP, on choisit une des variables en conflit (qui, avec une autre variable, ne satisfait pas une contrainte) et on lui affecte la valeur qui entraîne le moins de conflits.

D'autres ordonnancements [Haralick et al.80] ont été proposés qui s'appuient sur des statistiques opérées sur les valeurs pour connaître celles qui ont le plus de support afin de les essayer en premier.

Enfin, des expériences menées par Wallace et Freuder [Wallace et al.93] ont montré qu'une combinaison de ces heuristiques (largeur, taille du domaine, degré, nombre de domaines ne supportant pas la valeur), employée sur une des techniques de recherche de solutions présentées, dépasse en performances l'utilisation d'une seule de ces heuristiques. La combinaison vise ici à départager les variables qui n'ont pu être ordonnées par l'heuristique précédente.

4.4.3 Performances

Les critères de performances qui permettent d'évaluer le coût d'un algorithme de satisfaction de contraintes sont divers (le nombre de nœuds rencontrés dans l'arbre de recherche, le nombre de tests de consistance effectués pour s'assurer que l'instanciation courante satisfait les contraintes, le nombre de retours-arrière nécessaires...). En général, la communauté scientifique spécialiste des CSP s'accorde à dire qu'à chaque CSP correspond un algorithme et des heuristiques d'ordonnancement plus efficaces que d'autres en raison même de la spécificité du CSP traité. Néanmoins des études ont été menées pour tenter de déterminer quelles sont les combinaisons algorithmes/heuristiques qui sont les plus efficaces. Consensuellement, elles sont effectuées sur des CSP (binaires) générés aléatoirement en contrôlant quatre paramètres : le nombre de variables, le nombre de valeurs dans chaque domaine, la force des contraintes (représentée par le rapport entre le nombre de couples de valeurs satisfaisant la contrainte et le nombre total de couples admissibles) et le nombre de contraintes sur les $n * (n-1)/2$ possibles (qui reflète la densité du graphe de contraintes). Alors que le *Forward-Checking*, combiné ou non avec des heuristiques d'ordonnancement de variables, a longtemps été considéré comme le plus satisfaisant [Nadel89, Wallace et al.93], une récente étude semble redonner l'avantage au Backjumping combiné avec une heuristique d'ordonnancement dynamique des variables [Frost et al.94b].

Quant à employer des méthodes de filtrages, Prosser a suggéré [Prosser93] que la suppression de valeurs incompatibles ne pouvait être bénéfique que pour les algorithmes à base de *Backtrack* chronologique en montrant qu'elle pouvait, au contraire, dégrader les performances des algorithmes de *Backtrack* ≪intelligents≫ comme le *Backjumping*. L'emploi d'heuristiques d'ordonnancement peut également interférer avec un filtrage préalable des domaines [Sabin et al.94]. C'est le cas notamment de la combinaison *Forward-Checking/first fail principle* qui se montre plus efficace en moyenne lorsqu'elle n'est pas précédée par une étape d'établissement de l'arc-consistance. Sabin et Freuder présentent également un algorithme, appelé *MAC* pour *Maintaining Arc-Consistency*, qui surpasse le *Forward-Checking*. *MAC* est un *Full-Lookahead* appliqué à un CSP rendu arc-consistant par AC-4[5].

Les performances du *Backtrack* et du *Forward-Checking* peuvent également être améliorées en considérant les diverses notions d'*interchangeabilité* proposée par Freuder (interchangeabilité pleine et de voisinage [Freuder91]), Haselböck (partition de domaines [Haselböck93]), Bellicha *et al.* (susbtituabilité [Bellicha et al.94]). L'idée commune à ces approches est de réduire l'espace de recherche des solutions en éliminant du domaine d'une variable les valeurs interchangeables (qui peuvent se substituer l'une l'autre dans une solution) pour n'en conserver qu'une seule en tant que représentante.

Enfin, Hubbe et Freuder [Hubbe et al.92] ont proposé une représentation des CSP qui consiste à factoriser l'écriture des solutions partielles obtenues durant la recherche des solutions en employant des *produits croisés*.

Exemple 3 *Soient les variables X et Y de domaine respectif $dom(X) = \{a, b, c\}$ et $dom(Y) = \{e, f, g\}$. Si la contrainte $c(X, Y)$ s'exprime en extension par l'ensemble des couples de valeurs $\{(ae), (af), (bg), (be), (ce), (cf)\}$, alors cet ensemble peut se réécrire à l'aide de deux produits croisés : $\{a, c\} \times \{e, f\}$ et $\{b\} \times \{g, e\}$*

Ce mode de représentation est appliqué aux divers algorithmes de recherche de solution (*Backtrack*

[5] Rappelons que le *Full-Lookahead* cherche à établir l'arc-consistance entre les variables non encore instanciées là où le *Forward-Checking* se contente de l'assurer vis-à-vis de l'instanciation réalisée jusqu'alors.

et *Forward-checking*), en les adaptant aux produits croisés et en les transformant en des versions dites CPR qui se montrent nettement plus économes en vérifications de contraintes que les versions standards, notamment dans les cas de CSP peu contraints. Ceci s'explique par le fait qu'à un niveau donné, les tests s'effectuent ici sur chacun des éléments du produit (qui représente les instanciations partiellement consistantes réalisées jusqu'alors) et non plus sur les différentes combinaisons de valeurs réalisées jusqu'alors (on parle de complexité additive pour les CPR au lieu de multiplicative pour les standards). Reprenant l'idée de factorisation, David Lesaint [LeSaint94] a lui aussi étendu le *Forward-Checking* en calculant les ensembles de solutions qui sont des produits cartésiens des sous-domaines de variables.

4.4.4 Conclusion

Nous avons présenté ici les principaux résultats de la théorie des problèmes de satisfaction de contraintes. Ce qu'il faut en retenir est que l'arc-consistance est un moyen peu coûteux de réduire l'espace de recherche de solutions. Quant aux techniques de résolution, qu'elles soient rétrospectives ou prospectives, avoir recours à un ordonnancement statique ou dynamique des variables ou des valeurs est souvent bénéfique. De nombreuses études de cette théorie n'ont pas été développées ici car, soit elles ont un caractère trop spécifique, soit elles ne répondent pas, au prime abord, à notre objectif. Parmi elles, citons les études concernant :

- Des classes de CSP dont la résolution peut se faire en un temps polynomial du fait qu'une solution peut être trouvée sans retour-arrière, après application d'un algorithme de filtrage. Ces études s'intéressent soit aux topologies particulières des graphes de contraintes [Freuder82, Dechter et al.88b, Freuder90, Jégou93], soit à la sémantique des contraintes (combinaison de valeurs permises entre les variables contraintes) [Deville et al.91, Beek92, Dechter92, David93, Cooper et al.94].
- Les méthodes de décomposition dont le but est de transformer un CSP en un CSP équivalent mais dont le graphe associé a une structure d'arbre, c'est-à-dire qui peut être résolu polynomialement (d'après [Dechter et al.88b]). On trouve deux méthodes de décomposition : le *coupe-cycle* [Dechter et al.87] et le *regroupement en arbre* [Dechter et al.89b] mais l'efficacité de ces méthodes dépend fortement de la topologie du graphe initial du CSP.
- Les autres techniques de résolution telles que les *méthodes stochastiques* qui cherchent à s'approcher d'une solution à partir d'une instanciation initiale prise au hasard (Hill-Climbing [Minton et al.90, Minton et al.92, Morris93, Yokoo94] ou approche connexionniste [Tsang93, Davenport et al.94]), la *synthèse de solutions* [Freuder78, Seidel81, Tsang et al.90] qui est une démarche progressive de construction de solutions ou l'*optimisation de CSP* qui consiste à déterminer parmi toutes les solutions d'un CSP celle qui est optimale au regard d'une fonction d'optimisation [Goldberg89, Davis91].
- La satisfaction partielle de CSP qui s'adresse aux problèmes sur-contraints et dont le but est d'exhiber une instanciation complète (non globalement consistante) qui satisfait au mieux le CSP [Freuder et al.92, Wallace et al.93].

Jusqu'à présent dans notre présentation, les définitions, les algorithmes de consistance et les méthodes de résolution décrits s'adressent à des CSP à domaines finis. Or, dans la perspective de construire des CSP dans les bases de connaissances gérées par les SRPO, nous ne pouvons nous contenter de cette limitation, notamment parce que les domaines des attributs impliqués dans des propriétés (contraintes) peuvent être infinis. Il nous faut donc nous intéresser aussi aux CSP à intervalles dont les domaines peuvent être infinis.

4.5 Les CSP à intervalles

4.5.1 Introduction

Deux idées motivent la représentation des domaines des variables réelles d'un CSP sous la forme d'intervalles :

1. approximer un nombre réel en le localisant entre deux bornes (inférieure et supérieure), elles-mêmes représentées par des flottants. Cette technique permet, dans une certaine mesure seulement, de limiter les erreurs dues au calcul en précision finie. Elle est à l'origine de la première proposition, faite par John Cleary [Cleary87], pour étendre le langage PROLOG vers la gestion d'intervalles de réels.
2. décrire un ensemble de valeurs que peut prendre une variable. Aussi, ce principe de représentation compacte des domaines peut être étendu aux CSP à variables entières et, plus généralement, à tout type ordonné de variables. Par conséquent, des CSP à intervalles peuvent être des CSP à domaines finis, infinis ou continus.

Le premier langage de programmation par contraintes permettant de manipuler des intervalles de réels fut BNR PROLOG [Older et al.90]. D'ailleurs, les efforts de recherche théorique les plus importants sur les CSP à intervalles ont été réalisés dans le domaine de la programmation logique par contraintes. Ils ont été concrétisés par l'apparition de nouveaux langages comme INTERLOG [Dassault electronique91], ECHIDNA [Havens et al.92], CAL, GDCC [Aiba et al.92], `Newton` [Benhamou et al.94] ou l'amélioration de langages existants comme CHIP [Lee et al.93], CLP(BNR) [Older et al.93], CLP($\mathcal{R}$) [Lee et al.93].

Hors du domaine de la programmation logique par contraintes, on trouve plus généralement des modules de programmation par contraintes sur des intervalles destinés à des langages comme LE-LISP (PECOS [Ilog92b]) ou C++ (INC++ [Hyvönen et al.93b] et SOLVER [Puget et al.93]).

Cette section a pour but de présenter les notions générales de la programmation par contraintes sur des intervalles. Après quelques définitions (*cf.* 4.5.2), nous présentons les principaux résultats de l'arithmétique des intervalles (*cf.* 4.5.3) qui sont repris par les langages cités plus haut pour traiter les problèmes de satisfaction de contraintes à intervalles ou ICSP (*cf.* 4.5.4). Nous étudions ensuite les divers degrés de consistance atteignables dans les ICSP (*cf.* 4.5.5), notamment dans le cas particulier des opérateurs non monotones. Nous signalons ensuite des techniques de l'analyse numérique qui peuvent réduire considérablement la taille des domaines (*cf.* 4.5.6). Dans les ICSP, la consistance est établie par une technique de propagation de contraintes dont nous soulignons les limites (*cf.* 4.5.7). Nous décrivons pour finir les principes de la résolution de contraintes dans les ICSP (*cf.* 4.5.8).

4.5.2 Définitions

Définition 9 *Un **intervalle** i sur* $\mathcal{T} = \{\mathbb{Z}, \mathbb{R}\}$ *est un élément de l'ensemble I défini par :*
$I = \{]a,b] | a \in \mathcal{T} \cup \{-\infty\}, b \in \mathcal{T}\} \cup \{[a,b[| a \in \mathcal{T}, b \in \mathcal{T} \cup \{+\infty\}\} \cup \{[a,b] | a,b \in \mathcal{T}\} \cup \{]a,b[| a \in \mathcal{T} \cup \{-\infty\}, b \in \mathcal{T} \cup \{+\infty\}\}$

où a (resp. b) est définie comme la borne inférieure (resp. supérieure) de I.

Définition 10 *Un **CSP à intervalles** ou ICSP (X, D, C) est la donnée de :*

- *un ensemble X de n variables* $\{x_1, \ldots, x_n\}$,
- *un ensemble D de n domaines* $\{d_1, \ldots, d_n\}$, *où* d_i *est le domaine associé à la variable* x_i, d_i *est un intervalle ou une union d'intervalles,*
- *un ensemble C de m relations numériques* $\{c_1, \ldots, c_m\}$ *données en intension.*

La plupart des systèmes évoqués (sauf ECHIDNA) plus haut se contentent d'associer à chaque variable un intervalle unique et non pas une union. Aussi, certains ne sont-ils en mesure que de fournir à chaque instant qu'un sur-domaine (au sens de l'inclusion ensembliste) de l'ensemble de valeurs effectif d'une variable. Dans ce cas, à la consistance est préférée l'efficacité. Également, la majorité de ces systèmes (sauf ICHIP et ECHIDNA) ne considèrent que des intervalles à bornes fermées de type [a,b].

4.5.3 Arithmétique des Intervalles

L'Arithmétique des Intervalles (ou AI) est un domaine des mathématiques dont les fondations ont été établies dès 1966 par Ramon Moore [Moore66]. La motivation de ces travaux est de donner des méthodes simples et efficaces de l'approximation de l'image de fonctions sur des réels. Le premier domaine d'application de l'AI est l'analyse numérique. Là, l'idée est d'encadrer le domaine de variation de valeur d'un réel par un intervalle de flottants et de contrôler les erreurs d'approximation dues à la précision des calculs, en effectuant les calculs des fonctions à l'aide de règles fournies par l'AI. Alors que les calculs en virgule flottante ne sont qu'une estimation du résultat réel, l'emploi d'intervalles garantit que l'encadrement du résultat est correct, même si le recours à des arrondis de bornes est nécessaire [Handsen88]. Ce résultat repose sur le *théorème fondamental de l'arithmétique des intervalles* qui s'énonce ainsi :

Théorème 1 (d'après [Moore66]) *Soit $f(x_1, \ldots, x_n)$ une fonction de n variables x_i définies chacune sur un intervalle X_i. L'intervalle $F(X_1, \ldots, X_n)$, obtenu à partir de $f(x_1, \ldots, x_n)$ en remplaçant les arguments x_i par l'intervalle X_i correspondant et en remplaçant chaque opération arithmétique de f par l'opération équivalente de l'arithmétique des intervalles, contient $f(x_1, \ldots, x_n)$, $\forall x_i \in X_i$. F est appelée extension de f aux intervalles.*

Ce théorème garantit que l'on peut calculer l'intervalle image d'une fonction sans perdre d'information.

Les règles de l'arithmétique des intervalles qui aident à construire F sont les suivantes :

$$\begin{aligned} X + Y &= [a,b] + [c,d] = [a+c, b+d] \\ X - Y &= [a,b] - [c,d] = [a-d, b-c] \\ X * Y &= [a,b] * [c,d] = [min(a*c, a*d, b*c, b*d), max(a*c, a*d, b*c, b*d)] \\ X/Y &= [a,b]/[c,d] = [a,b] * [1/d, 1/c], \text{ si } \ 0 \notin [c,d] \end{aligned}$$

Pour toute fonction $f(x_1, \ldots, x_n)$, ces règles vont permettre de calculer l'intervalle $I = F(X_1, \ldots, X_n)$ à partir des intervalles X_i des variables x_i et de garantir que cet intervalle contient l'ensemble des valeurs prises par f sur le domaine $X_1 \otimes X_2 \otimes \ldots \otimes X_n$.

Il est important de remarquer les différences entre les opérations de l'arithmétique des intervalles et les opérations de l'arithmétique classique. Ainsi :

$$\begin{aligned} 0 + X &= [0,0] + [a,b] = [a,b] = X \\ 1 * X &= [1,1] * [a,b] = [min(a,b), max(a,b)] = [a,b] = X \\ X + X &= [a,b] + [a,b] = [a+a, b+b] = [2*a, 2*b] = 2*X \\ \text{mais } X - X &= [a,b] - [a,b] = [a-b, b-a] \neq 0 = [0,0] \end{aligned}$$

Si l'addition et la multiplication de l'arithmétique des intervalles sont commutatives et associatives, la multiplication est *sous-distributive* par rapport à l'addition :

$$A * (B + C) \subseteq A * B + A * C$$

La distributivité n'est effective que lorsque $B * C > 0$

Pour d'autres fonctions de base telles que $log(X), exp(X), X^a, sin(X) \ldots$, il est possible également d'établir les règles de calcul des intervalles correspondantes.

4.5.4 ICSP et Arithmétique des Intervalles

Cleary [Cleary87] fut le premier à lier la programmation par contraintes à l'arithmétique des intervalles. Cette association repose sur trois points :

1. associer un intervalle à chaque variable contrainte ;
2. traduire chaque expression de contrainte *complexe* en contraintes *primitives* ;
3. fournir un mécanisme de division d'intervalles pour une recherche itérative des solutions.

À ces trois fonctionnalités, il faut ajouter également un mécanisme de propagation de contraintes chargé de propager dans le réseau de contraintes existant les réductions de domaines qu'a pu occasionner l'ajout d'une nouvelle contrainte.

Dans un tel système, un ICSP est défini par un système de contraintes complexes composées de contraintes primitives portant sur des variables dont le domaine associé est un intervalle ou une union d'intervalles.

Les contraintes primitives sont gérées par le système. Chacune d'elles représente une opération de comparaison ($=, <, >, \leq, \geq, \neq$) ou une opération arithmétique classique ($+, -, *, /, sin, log \ldots$) réalisée sur des intervalles en utilisant les règles de l'arithmétique des intervalles. Les contraintes primitives représentant une opération arithmétique produisent une variable dite *intermédiaire* qui correspond au résultat de l'opération. Bien que non apparentes pour l'utilisateur du système, ces variables intermédiaires permettent d'employer le résultat d'une opération comme argument d'une autre contrainte primitive (*cf.* tableau 4.4). Bien qu'elle s'appuie sur l'arithmétique des intervalles pour calculer notamment les intervalles résultats des opérations arithmétiques, l'arithmétique des contraintes sur des intervalles s'en démarque sur deux points. D'abord, l'arithmétique des contraintes sur des intervalles est applicable à des relations, là où l'arithmétique ne traite que des fonctions. Ainsi, l'arithmétique des intervalles est capable de calculer des intervalles de valeurs pour une fonction mais ne peut inverser ou même résoudre des équations, des inégalités ou des différences. Ensuite, une contrainte doit, par définition, être opérante toute la durée de son existence. Il s'agit ici de garantir le maintien de la cohérence d'un ensemble de variables sous l'effet d'un ensemble de contraintes, et, donc, de fournir un mécanisme – la propagation de contraintes – capable de répercuter toute modification le long du réseau de contraintes.

$$\left\{ \begin{array}{l} cc_1 : Z = X * Y \\ cc_2 : e^X + Z > 12.2 \end{array} \right. \iff \left\{ \begin{array}{l} cp_1 : Z = X * Y \\ cp_2 : e^X = X_1 \\ cp_3 : X_1 + Z = X_2 \\ cp_4 : X_2 > 12.2 \end{array} \right.$$

TAB. 4.4 - : Transformation d'un système de deux contraintes complexes (cc_i) en un système équivalent de quatre contraintes primitives (cp_i). Deux variables X_i intermédiaires sont nécessaires.

Afin de garantir le caractère relationnel des contraintes primitives, on leur associe des règles de maintien de la consistance locale, basées sur l'arithmétique des intervalles, qui garantissent qu'aucune valeur localement consistante n'est écartée des intervalles calculés (*cf.* tableau 4.5).

$$\begin{array}{c} x = y \\ \text{où } dom(x) = [a, b] \\ \text{et } dom(y) = [c, d]) \end{array} \qquad \left\{ \begin{array}{l} dom(x) = [a', b'] = [a, b] \cap [c, d] \\ dom(y) = [c', d'] = [a', b'] \end{array} \right.$$

$$\begin{array}{c} z = x + y \\ \text{où } dom(x) = [a, b], \\ dom(y) = [c, d] \\ \text{et } dom(z) = [e, f] \end{array} \qquad \left\{ \begin{array}{l} dom(z) = [e', f'] = [a + c, b + d]\{\cap[e, f] \text{ si } [e, f] \text{ défini }\} \\ dom(x) = [a', b'] = ([e', f'] - [c, d]) \cap [a, b] \\ dom(y) = [c', d'] = ([e', f'] - [a', b']) \cap [c, d] \end{array} \right.$$

TAB. 4.5 - : À gauche deux exemples de contraintes primitives. À droite, les règles de maintien de consistance associées.

Chacune des règles de maintenance permet de calculer, pour chaque variable de la contrainte, le nouvel intervalle qui constitue son domaine, en fonction des intervalles des autres variables de la contrainte.

L'ensemble des règles de maintenance associées à une contrainte primitive vérifie quatre propriétés essentielles [Benhamou et al.95] :

- il est *contractant*: après application des règles de maintenance, l'intervalle de chaque variable est inclus ou égal à l'intervalle précédent.

- il est *correct*: après application des règles de maintenance, toute valeur d'une instanciation satisfaisant la contrainte est toujours dans l'intervalle (s'il est non vide) de la variable correspondante.
- il est *monotone*: l'inclusion des intervalles est préservée par l'application des règles de maintenance.
- il est *idempotent*: une seule application des règles de maintenance suffit à calculer les nouveaux intervalles des variables. Toute autre application sur les mêmes intervalles ne les réduit pas.

Si la correction est prouvée – elle est garantie notamment par le théorème fondamental de l'arithmétique des intervalles –, les règles de l'arithmétique des intervalles ne donne qu'une approximation plus ou moins bonne de la réalité. Aussi, existe-t-il plusieurs degrés de consistance locale réalisables dans un système de programmation par contraintes sur des intervalles. Ils dépendent notamment de la représentation choisie pour les intervalles, du type des fonctions représentées par les contraintes primitives, du critère de consistance choisi, voire même de la méthode employée pour calculer les intervalles résultats pour des fonctions complexes. C'est ce que montrent les prochaines sections.

4.5.5 Consistances dans les ICSP

4.5.5.1 Approximations de l'arc-consistance

En raisonnant sur des intervalles de réels, c'est-à-dire des ensembles de valeurs non toutes représentables en machine, on peut considérer que l'arc-consistance n'est pas possible (par exemple, la contrainte arcsin(1) = x associe à x la valeur $\pi/2$ non représentable). Un intervalle de réels étant représenté en machine par un intervalle de nombres flottants, la notion d'arc-consistance ne peut donc être qu'approximée. Dès lors, un intervalle continu de réels est en réalité un intervalle fini de flottants ; la notion même d'intervalle à bornes infinies – que ce soit sur des entiers ou sur des réels – n'existe pas non plus en machine où ces bornes infinies sont capturées par le plus grand nombre représentable.

C'est à partir de ces limitations des machines que diverses études se sont attachées à définir divers degrés de consistance dans les ICSP. Notamment, les concepteurs du langage CLP(BNR) [Benhamou et al.94] ont défini l'*intervalle-consistance*, l'*enveloppe-consistance* et la *boîte-consistance*.

Notations On appelle :

- $\mathcal{F}$ tout sous ensemble fini de $I\!R \cup \{+\infty, -\infty\}$ ($\mathcal{F}$ est un ensemble de flottants),
- F-intervalle, tout intervalle $[a, b]$ où a, b $\in \mathcal{F}$
- $\mathcal{I}(\mathcal{F})$ l'ensemble des F-intervalles,
- $\mathcal{U}(\mathcal{F}) = \{D \subseteq I\!R \mid \exists I_1, \ldots, I_n \in \mathcal{I}(\mathcal{F})^n : D = I_1 \cup \ldots \cup I_n\}$

Définition 11 *L'**approximation**, notée appx, d'une relation r est le plus petit (au sens de l'inclusion ensembliste) élément de $\mathcal{U}(\mathcal{F})$ contenant r.*

Ici, on approxime l'ensemble des n-uplets de réels qui satisfont r par un ensemble d'union de F-intervalles.

Définition 12 (d'après [Benhamou et al.94]) *Un ICSP est **intervalle-consistant** si et seulement si pour toute variable X_i de domaine D_i et pour toute contrainte $C(X_1, \ldots, X_i, \ldots, X_n)$,*
$D_i = appx(D_i \cap \{a_i \in I\!R \mid \exists a_1 \in D_1, \ldots, \exists a_{i-1} \in D_{i-1}, \exists a_{i+1} \in D_{i+1}, \ldots \exists a_n \in D_n,$
avec $C(a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots a_n)\}$ *satisfait*$\})$

L'intervalle-consistance est observée dans ECHIDNA [Havens et al.92] à l'aide d'un algorithme d'arc-consistance hiérarchique [Mackworth et al.85], mais parce qu'elle demande la conservation d'unions d'intervalles pour des fonctions non monotones, elle peut présenter un coût élevé de maintenance. Aussi, dans des systèmes qui travaillent sur des intervalles uniques, on peut lui préférer une approximation moins fine – donc une consistance plus faible – appelée *enveloppe* ou *hull* consistance.

Définition 13 *L'**enveloppe**, notée envl, d'une relation r est le plus petit F-intervalle contenant r.*

Ici, on approxime l'ensemble des n-uplets de réels qui satisfont r par un ensemble de F-intervalles.

Définition 14 (d'après [Benhamou et al.94]) *Un ICSP est* ***enveloppe-consistant*** *si et seulement si pour toute variable X_i de domaine D_i et pour toute contrainte $C(X_1, \ldots, X_i, \ldots, X_n)$,*
$D_i = envl(D_i \cap \{a_i \in I\!R \mid \exists a_1 \in D_1, \ldots, \exists a_{i-1} \in D_{i-1}, \exists a_{i+1} \in D_{i+1}, \ldots \exists a_n \in D_n,$
avec $C(a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots a_n\})$ *satisfaite*$\})$

Cependant, cette consistance, si elle est satisfaisante pour les contraintes primitives, s'avère encore trop forte pour des contraintes contenant plusieurs occurrences de la même variable.

Par exemple [Benhamou et al.94], la contrainte $x_2 + x_1 - x_2 = 0$ (où $dom(x_1) = [-1, 1]$ et $dom(x_2) = [0, 1]$), n'est pas enveloppe-consistante, car -1 et 1 sont des valeurs impossibles pour x_1.

Dans ces cas, on réalise simplement une *consistance de boîte* :

Définition 15 (d'après [Benhamou et al.94]) *Un ICSP est* ***boîte-consistant*** *si et seulement si pour toute variable X_i de domaine D_i et pour toute contrainte $C(X_1, \ldots, X_i, \ldots, X_n)$,*
$D_i = envl(D_i \cap \{a_i \in I\!R \mid$ *l'extension de la contrainte C aux intervalles* $(D_1, \ldots, D_{i-1}, [a_i, a_i], D_{i+1}, \ldots, D_n)$ *est non vide* $\})$

Cette dernière consistance est plus faible encore que les précédentes (*cf.* figure 4.5). Il s'agit de la consistance obtenue dans la plupart des systèmes de programmation par contraintes à intervalles (CLP(BNR), SOLVER...) qui ne considèrent qu'un intervalle unique pour les variables contraintes et fondent le calcul des intervalles des variables impliquées dans les contraintes primitives (et donc aussi complexes) sur les règles de l'arithmétique des intervalles.

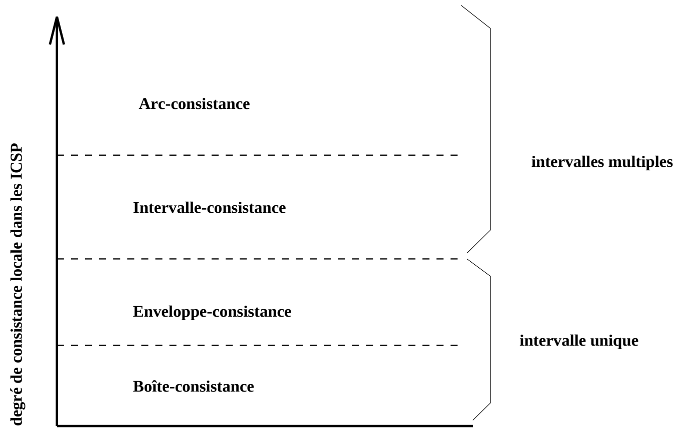


FIG. 4.5 - : Les différentes degrés de consistance locale des ICSP.

4.5.5.2 Conditions d'application et non monotonie

Il peut s'avérer que le calcul d'un intervalle ne soit pas possible lorsque l'opération à appliquer n'est pas définie. Par exemple, la contrainte $z = x/y$ ne peut être satisfaite lorsque 0 appartient au domaine de y. Il faut donc l'en exclure, ce que fait la règle correspondant à la division de l'arithmétique des intervalles. Mais, la même situation apparaît pour la contrainte $x * y = z$ qui n'est pas dénoncée par la règle de multiplication des intervalles. En effet, l'une des règles de maintenance associée à la contrainte est la règle $x = z/y$.

Par exemple, dans le cas où $dom(x) = [-2, 3]$, $dom(y) =]-\infty, +\infty[$, $dom(z) = [1, 1]$, alors 0 doit être supprimé du domaine de y (car il n'existe pas de nombre entre -2 et 3 qui multiplié par 0 donne 1). En fait, y ne peut prendre ici aucune valeur entre $-1/2$ et $1/3$. Finalement, $dom(y) =]-\infty, -1/2] \cup [1/3, +\infty[$ est un domaine intervalle-consistant pour y. Le domaine de y est ici représenté par une union d'intervalles.

Cette situation est problématique dans les systèmes n'acceptant qu'un intervalle unique où les domaines (comme celui de y sur l'exemple) peuvent devenir *non convexes.*

Définition 16 *Un domaine D_i = [a, b] d'une variable contrainte X_i est dit* ***convexe*** *si toutes les valeurs entre a et b sont localement consistantes.*

Définition 17 (D'après [Lhomme93]) *Une contrainte $c(X_1, \ldots, X_n)$ est dite* ***disjonctive*** *si au moins un des domaines D_i associés à la variable $X_i, \forall i \in [1, n]$, est non convexe.*

Cette propriété n'est pas seulement le fait de la contrainte, elle dépend également des domaines, comme le montre l'exemple suivant :

Par exemple, la contrainte $y = x^2$ est non disjonctive pour $dom(x) = [-2, 2], dom(y) = [0, 4]$, mais est disjonctive pour $dom(x) = [-2, 2], dom(y) = [1, 4]$, puisque $dom(x)$ contient la valeur 0 qui n'est pas localement consistante.

Pour qu'une contrainte disjonctive soit transformée en contrainte non disjonctive, la solution est de considérer des unions d'intervalles qui ne contiennent que les valeurs localement consistantes. De ce point de vue, il est possible d'obtenir une intervalle-consistance ou arc-consistance sur des nombres flottants. Cependant, cette solution est souvent dénoncée comme pouvant donner lieu à une explosion combinatoire du nombre d'intervalles à considérer.

Par exemple, soient x, y et z trois variables avec $dom(x) = dom(y) = [0, 3142], dom(z) = [0, +\infty[$ telles que :

$$\begin{cases} sin(x) = 0.8 \\ cos(y) = -0.4 \\ z = x * y \end{cases}$$

Rendre le domaine de x arc-consistant revient à le transformer en une union de 1000 intervalles singleton. De même pour y qui est arc-consistant au prix de 1000 intervalles. La dernière contrainte $z = x * y$ nécessite de considérer 10^6 intervalles pour rendre le domaine de z arc-consistant.

Cette situation extrême, mais atteinte avec simplement trois contraintes, est la motivation principale des systèmes de programmation par contraintes sur intervalles pour ne considérer que des intervalles uniques CLP(BNR), SOLVER, INTERLOG.... En contrepartie, l'intervalle-consistance ou l'arc-consistance sur les flottants n'est pas toujours possible dans ces systèmes.

Proche de la conservation d'unions de domaines, la solution proposée par Hyvönen [Hyvönen92] consiste à considérer des *divisions d'intervalles.* Une division est un ensemble d'intervalles (par exemple, $[1, 2 \mid 8, 9]$ est une division d'un domaine initial $[0, 10]$).

Les divisions des domaines initiaux sont obtenues en décomposant chaque contrainte complexe en contraintes primitives, et en considérant, pour chaque variable, son domaine de définition. Sur chaque division, la contrainte primitive doit être définie et monotone. La monotonicité garantit que la contrainte primitive représentée a son maximum et son minimum aux bornes de la division. Cependant, la contrainte primitive peut ne pas être monotone, et avoir quand même son maximum et son minimum aux bornes d'un intervalle d'une situation qui aura été (sur le critère de non monotonicité) décomposée inutilement en divisions.

Les critères de définissabilité et de monotonicité pris en compte par Hyvönen peuvent engendrer une croissance exponentielle du nombre de divisions (pour les mêmes cas critiques que celui exposé plus haut, par exemple). Afin d'y remédier, Hyvönen propose une opération dite de *non-éclatement*, notée $\cap_n$ et définie par :

$$D' \cap_n D = \bigcup \{[min(X_i \cap_n D'), max(X_i \cap_n D')] \mid X_i \in D\}$$

où D' est la nouvelle division évaluée par une règle de maintenance d'une contrainte primitive et D l'ancienne division.

Par exemple : $[1, 2 \mid 8, 9] \cap_n]-\infty, +\infty[= [1, 9]$.

Cette solution a l'avantage de stopper la décomposition des intervalles mais en contrepartie ne garantit plus que toute valeur dans les intervalles est localement consistante même si elle garantit encore que la ou les solutions se trouvent toujours dans cet ensemble. Autrement dit, l'idée d'arc-consistance est là aussi abandonnée pour une consistance plus faible.

4.5.5.3 Consistances de bornes

Pour les systèmes ne conservant qu'un intervalle unique, à défaut d'arc-consistance, Olivier Lhomme [Lhomme93] propose une consistance, plus faible, appelée *arc-consistance de bornes* ou *arc-B-consistance* qui agit seulement sur les bornes des domaines.

Définition 18 *Un ICSP(X, D, C) est arc-B-consistant si et seulement si* $\forall x \in X$ *telle que* D_x= *[a b],* $\forall c(x, x_1, \ldots, x_k) \in C$, *on a :*
$\exists v_1, \ldots, v_k \in D_1 \otimes \ldots \otimes D_k$ *telles que* $C(a, v_1, \ldots, v_k)$ *est satisfaite*
$\exists v_1, \ldots, v_k \in D_1 \otimes \ldots \otimes D_k$ *telles que* $C(b, v_1, \ldots, v_k)$ *est satisfaite.*

L'arc-consistance de bornes est donc une consistance d'arc limitée aux bornes des intervalles des variables. L'idée de l'arc-B-consistance est de ne conserver pour les contraintes disjonctives qu'un seul intervalle dont les bornes sont localement consistantes. Lhomme propose un algorithme d'arc-consistance de borne de complexité $O(Am)$ où A est le maximum de la taille des domaines et m le nombre de contraintes. Cette consistance s'apparente à la boîte-consistance décrite plus haut.

Exemple 4 (d'après [Lhomme93]) *Le CSP défini par* $(X = \{x, y\}, D = \{dom(x) = [-2, 2], dom(y) = [1, 4]\}, C = \{y = x^2\})$ *est arc-B-consistant mais pas arc-consistant (sur des entiers), car* 0 *dans* $dom(x)$ *n'est supporté par aucune valeur de* y.

Une consistance plus forte, appelée 3-B-consistance, car elle s'apparente à la 3-consistance forte de Freuder, a été proposée également par Lhomme [Lhomme93].

Définition 19 *Un ICSP est* ***3-B-consistant*** *si pour chaque domaine* $D_i = [a, b]$, *a et b font partie chacune d'une solution.*

Cette consistance peut être établie avec un coût de $O(mn^2A^2)$ où A est le maximum de la taille des domaines, m le nombre de contraintes, et n le nombre de variables.

Cette consistance de bornes s'avère très intéressante dans les ICSP pour lesquels les valeurs dans les solutions ne sont pas isolées mais, au contraire, forment des ensembles de valeurs continus.

4.5.6 Techniques algébriques de réduction

Les règles de l'arithmétique des intervalles données à la section 4.5.3 calculent pour l'intervalle résultat une *extension naturelle* de celui-ci. Cette extension naturelle est satisfaisante dans de nombreux cas, comme l'indique le théorème suivant :

Théorème 2 (d'après Moore [Moore79]) *Si une fonction* $f(x_1, \ldots, x_n)$ *ne comporte pas plusieurs occurrences de la même variable, alors* $F(X_1, \ldots, X_n)$, *l'extension de* f *aux intervalles, évalue l'intervalle exact des valeurs que peut prendre* $f(x_1, \ldots, x_n)$.

Autrement dit, pour des expressions simples comme $X + Y - (Z * T)$, l'intervalle calculé couvre bien l'étendue des valeurs admissibles. En revanche, comme l'atteste la sous-distributivité de l'arithmétique des intervalles ($X * (Y + Z) \subseteq X * Y + X * Z$), des problèmes surviennent lorsque l'on considère des fonctions avec des occurrences multiples de la même variable (ici, $X * Y + X * Z$). Ce que dit la sous-distributivité, c'est qu'il existe une expression équivalente $X * (Y + Z)$ dont l'extension naturelle peut-être incluse dans celle de $X * Y + X * Z$. Donc, que l'extension naturelle de $X * Y + X * Z$ couvre *plus* que l'étendue exacte de l'expression.

Le problème de déterminer la meilleure extension possible (celle qui donne un intervalle dont le minimum et le maximum sont les plus proches du minimum et du maximum réels de la fonction) est référencé en arithmétique des intervalles sous le terme d'*optimisation globale*.

Sous la garantie de déterminer des bornes d'intervalles qui contiendront toujours les solutions, la recherche en arithmétique des intervalles s'est dirigée à la fois vers les algorithmes numériques et vers les techniques algébriques.

Les algorithmes numériques d'évaluation de l'intervalle d'une fonction sont basés sur l'algorithme de Skelboe [Skelboe74] et des techniques de *Branch and Bound*. L'idée est d'explorer des parties des intervalles des arguments en jugeant de leur monotonicité en calculant la valeur de la dérivée première ou seconde.

Par des techniques algébriques, il est possible de restreindre l'intervalle des valeurs d'une fonction $f(x_1, \ldots, x_n)$ contenant des occurrences multiples d'une même variable et donc de fournir un encadrement inclus dans l'extension naturelle. Trois techniques peuvent être employées [Alander85] :

1. Le *schéma de Horner* qui est suggéré par la sous-distributivité. Ainsi, mettre un polynôme $P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0$ sous la forme $P_n(X) = a_0 + X * (a_1 + X * (a_2 + \ldots + X * a_n) \ldots)$ donne un encadrement qui n'est jamais plus grand que celui fourni par l'extension naturelle.
2. La *forme centrée de Moore* [Moore66] qui, pour une fonction $f(x_1, \ldots, x_n)$ est définie par $F_c(Y_1, \ldots, Y_n) = f(m_1, \ldots, m_n) + G(X_1 - m_1, \ldots, X_n - m_n)$ où m_i est le centre de l'intervalle X_i et $Y_i = X_i - m_i$. Elle s'obtient en remplaçant chaque X_i par $Y_i + m_i$. L'idée est de décaler les intervalles arguments de la fonction afin de les faire chevaucher 0, ce qui minimise les valeurs des limites de ces intervalles et peut donc produire un encadrement plus faible que l'extension naturelle.
3. La *forme de la valeur moyenne* qui, pour une fonction à un argument $f(x)$, est définie par $F_m(X) = f(m) + F'(X)(X - m)$ où m est le milieu de (ou plus généralement appartient à) X. Il est également possible de définir la valeur moyenne d'une fonction f à n-arguments en faisant intervenir les dérivées de f par rapport à chaque argument. La forme de la valeur moyenne est le développement de Taylor à l'ordre 1. Aussi d'autres formes peuvent être obtenues avec le développement de Taylor à des ordres supérieurs.

Nous reportons ici les extensions (à droite) fournies par ces techniques pour le polynôme $P(x) = x^4 - 10x^3 + 35x^2 - 50x + 24$ avec $X = [0, 4], m = 2, Y = [-2, 2]$ (d'après Alander) :

Somme des puissances	$x^4 - 10x^3 + 35x^2 - 50x + 24$	$[-816, 840]$
Schéma de Horner	$24 + x(-50 + x(35 + x(-10 + x)))$	$[-256, 384]$
Expansion des racines	$(x-1)(x-2)(x-3)(x-4)$	$[-72, 72]$
Forme centrée	$y^4 - 2y^3 - y^2 + 2y + P(2)$	$[-24, 36]$
Forme de la valeur moyenne	$P(2) + (4(y+2)^3 - 30(y+2)^2 + 70(y+2) - 50)y$	$[-1060, 1060]$
Forme de la valeur moyenne/centrée		$[-116, 116]$
Étendue exacte		$[-1, 25]$

Si, sur cet exemple, la forme centrée de Moore donne le plus petit encadrement, ce n'est pas toujours le cas, et il n'existe pas de méthode qui soit toujours la meilleure. L'efficacité de la méthode choisie dépend des propriétés de la fonction et des intervalles des arguments. Pour obtenir l'étendue réelle des valeurs d'une fonction, il est conseillé [Ratscheck et al.84] de combiner techniques algébriques (le second ordre de Taylor s'avère une très bonne heuristique) et algorithmes numériques. Le système INC++ [Hyvönen et al.93b] intègre de telles techniques. Mais celles-ci sont difficiles à mettre en œuvre et nécessitent un module dédié au calcul de dérivées des fonctions qui composent les expressions contraintes.

4.5.7 Propagation dans les ICSP

La propagation de contraintes est la capacité qu'a un système de répercuter sur un ICSP les changements (réductions) opérés sur les domaines des variables suite à l'introduction d'une contrainte[6]. L'algorithme approprié est un algorithme très proche des algorithmes d'arc-consistance. La différence réside dans le fait que les algorithmes d'arc-consistance ont été définis pour des CSP à domaines finis et qu'en conséquence, ils reposent sur une fonction qui teste la consistance lo-

[6] Nous en donnons ici une version incrémentale, mais il existe des versions qui opèrent sur un ensemble de contraintes.

cale de chaque valeur dans les domaines. Davis [Davis87] a été le premier à adapter l'algorithme d'arc-consistante (en fait, la procédure de Waltz [Waltz72]) aux intervalles mais seulement pour des intervalles représentant des ensembles finis d'entiers. Dans les domaines continus de réels – même si on peut considérer qu'ils sont en fait des domaines finis mais très grands de nombres flottants –, ce procédé n'est pas applicable. La solution adoptée le plus fréquemment (SOLVER, BNR-PROLOG, CLP(BNR), INC++...) consiste simplement à remplacer cette fonction de révision par l'application des diverses règles de maintenance de consistance associées à la contrainte ajoutée.

Nous donnons ici une version d'un algorithme de propagation de contraintes qui établit une $\mathcal{A}$-consistance où $\mathcal{A} = \{arc, intervalle, enveloppe, B, 3-B\ldots\}$. Il est appelé aussi *algorithme de réduction* [Benhamou et al.95] ou *algorithme de point fixe* [Benhamou et al.94], car après son application, la réduction devient idempotente.

Algorithme Propagation
entrée
  un ICSP (X, D, C) où C éventuellement vide
  une contrainte c
début
  C $\leftarrow$ C $\cup\{c\}$
  queue $\leftarrow \{c\}$
  tant que queue $\neq \emptyset$ **faire**
    c'$(x_1, \ldots, x_j) \leftarrow$ premier(queue)
    % premier(queue) retourne le premier élément de la queue et le supprime de la queue %
    $D_{c'} \leftarrow \{d_1, \ldots, d_j\}$
    $Dred_{c'} \leftarrow \mathcal{A}$-consistance(c', $D_{c'}$)
    % $\mathcal{A}$-consistance(c', $D_{c'}$) est un filtrage pour la $\mathcal{A}$-consistance des domaines de $D_{c'}$ %
    si $\exists d'_i \in Dred_{c'}$ tel que $d'_i = \emptyset$ **alors** STOP (inconsistance)
    sinon
      pour tout $d'_i \in Dred_{c'}$ **faire**
        si $d'_i \neq d_i \in D_{c'}$ **alors** queue $\leftarrow$ queue $\cup\{c'' \mid c'' \in constr(x_i) \wedge c'' \neq c'\}$
        finsi
      fin pour
    finsi
  fintantque
fin

4.5.7.1 Problèmes de terminaison et cycles

L'un des inconvénients majeurs de ce type d'algorithme de propagation est qu'il peut présenter certains problèmes de terminaison.

Considérons l'ICSP suivant :

$$\begin{cases} c_1 : y = x \\ c_2 : y = 2 * x \end{cases}$$

- si $x = [0, 100]$ et $y = [0, 100]$ sont deux variables entières, la propagation mène rapidement à la solution $x = y = 0$
- si $x = [0., 100.]$ et $y = [0., 100.]$ sont deux variables réelles, la propagation de c_2 donne $x = [0., 50.]$, celle de c_1 donne $y = [0., 50.]$, puis celle de c_2 donne $x = [0., 25.]$...au bout de 30 itérations, $x = [0., .003051758]$ et $y = [0., .003051758]$ et le processus n'est pas terminé. Il y a convergence asymptotique vers 0.

Ici, l'algorithme est entraîné dans une boucle "infinie" dont la longueur dépend des capacités de la machine, c'est-à-dire de la précision et donc de la plus petite largeur d'intervalle autorisée.

Pour maîtriser ce phénomène et finalement le stopper avant que la machine ne s'en charge, il est possible de définir un certain degré de précision (par exemple, la plus petite largeur d'intervalle autorisée). Cette idée est à la base de l'arc-B(w)-consistance présentée par Lhomme [Lhomme93] où w désigne un intervalle de valeurs autour des bornes dans lequel on souhaite trouver une valeur (très proche d'une borne) qui soit localement consistante.

Faltings [Faltings94] indique que ce comportement de l'algorithme n'est pas surprenant et qu'il est à suspecter dès que le graphe de contraintes associé à l'ICSP contient un cycle. Il montre également que si ce graphe est un arbre, il existe une itération de réductions, dont le nombre est linéairement proportionnel à la taille des variables, qui conduit à la consistance recherchée.

Plus décourageant est le fait que l'algorithme de propagation ne permet pas de résoudre les systèmes les plus simples :

Soit l'ICSP suivant :

$$\begin{cases} c_1 : a = b + c \\ c_2 : b = c + d \end{cases}$$

où initialement $dom(a) = [8,8], dom(d) = [4,4], dom(b) = dom(c) = [0,10]$ L'application de l'algorithme de propagation réalisant une consistance de bornes, par exemple, s'arrête avec $dom(a) = [8,8], dom(d) = [4,4], dom(b) = [4,8], dom(c) = [0,4]$. Comme prévu, les intervalles obtenus contiennent bien la solution mais, il est décevant de constater que par une simple élimination de Gauss, on obtient : $b = (a+d)/2 = 6$ et $c = (a-d)/2 = 2$. James Gosling [Gosling83] a proposé dans son système MAGRITTE une procédure de transformation de systèmes linéaires de contraintes d'égalité basée sur des éliminations de variables et la génération de contraintes implicites par substitution de variables. MAGRITTE peut résoudre un tel système. Notons que la 3-B-consistance permet ici de résoudre ce problème.

Enfin, au registre des approximations, il faut signaler que seul le système INC++ [Hyvőnen et al.93a] est en mesure d'éviter la dégénérescence de certains résultats lorsque de grands nombres sont manipulés.

Par exemple, $Z = (X - Y) - 1E32$ avec $X = 9.99E34, Y = 9.98E34$ donne 8.81E18 au lieu de 1E32 en nombres flottants double précision.

Ce système profite des facilités du langage hôte C++ pour encadrer tout intervalle $[min, MAX]$ par un plus grand intervalle $[min^-, MAX^+]$ où min^- est le plus grand nombre inférieur à min représentable en machine et MAX^+ est le plus petit nombre supérieur à MAX représentable en machine. Cette technique a été proposée par Jeffrey Ely [Ely90].

4.5.8 Résolution dans les ICSP

Pour des ICSP définis sur des réels, les domaines ne sont pas raisonnablement énumérables, même si on peut les imaginer comme de grands ensembles finis de flottants. Le même raisonnement peut s'appliquer sur des intervalles à bornes infinies d'entiers. La recherche des solutions d'un ICSP à domaines continus ou infinis ne relève donc pas du même processus d'énumération que la recherche de solutions dans un CSP à domaines finis.

Comme pour les CSP à domaines finis, il est des classes d'ICSP dont la consistance locale implique la consistance globale : les ICSP dont le graphe associé est acyclique [Hyvönen92]. Dans ce cas, on peut trouver une solution sans effectuer de retour-arrière. En affectant une valeur à une variable, on peut trouver des valeurs supports de cette valeur dans les domaines des variables qui lui sont liées (car l'ICSP est localement consistant) puisqu'elles ne sont pas liées elles-mêmes entre elles par d'autres contraintes (car l'ICSP est sans cycle).

Exemple 5 *Soit l'ICSP :*

$$\begin{cases} c_1 : z = x + y \\ c_2 : z + t = v \end{cases}$$

où $dom(x) = [0,2], dom(y) = [1,3], dom(z) = [4,6], dom(t) = [0,1], dom(v) = [4,6]$. La propagation de c_1 donne $dom(x) = [1,2], dom(y) = [2,3], dom(z) = [4,5], dom(t) = [0,1], dom(v) = [4,5]$. La propagation de c_2 donne $dom(x) = [1,2], dom(y) = [2,3], dom(z) = [4,5], dom(t) = [0,1], dom(v) = [4,5]$. L'ICSP est globalement consistant, une des solutions est $(x = 1, y = 3, z = 4, t = 1, v = 5)$.

Lorsque le graphe associé à l'ICSP comporte un cycle, Hyvönen a prouvé que si les variables de n'importe quel coupe-cycle du graphe ont un domaine réduit à un singleton, alors la consistance locale de cet ICSP implique la consistance globale.

Mais il est rare qu'une telle configuration se produise on peut s'attendre à ce que les domaines des variables du coupe-cycle ne soient pas en général réduits à un singleton. Aussi, pour poursuivre la recherche d'une solution dans un ICSP, on a recours à l'*éclatement* ou *splitting* des domaines. Il s'agit de diviser – le plus souvent par dichotomie mais le choix peut être nuancé (*cf.* [Cleary87]) – un ou plusieurs intervalles et de résoudre les (sous)-ICSP obtenus par un processus récursif.

Pour un ICSP cyclique – c'est le cas, par exemple, lorsqu'une contrainte présente plusieurs occurrences de la même variable – à n variables, Hyvönen propose de porter le choix des domaines à diviser vers une des variables coupe-cycle (de plus grand degré) de façon à obtenir la consistance globale au plus tôt.

L'idée du splitting est de s'intéresser à chacun des deux sous-ICSP formés par l'éclatement de l'intervalle de la variable et d'établir tout d'abord la consistance locale du sous-ICSP. S'il s'avère qu'un des sous-ICSP n'est pas consistant alors, le sous-domaine le concernant ne sera pas divisé davantage. Si le sous-ICSP est localement consistant mais pas globalement consistant ou que le critère de terminaison du *splitting* n'est pas satisfait, on procède à l'éclatement du sous-ICSP et ainsi de suite. La détermination de sous-ICSP inconsistants est le but principal car on cherche ici à réduire au maximum les domaines vers les solutions.

Exemple 6 (d'après [Hyvönen92]) *Soit l'ICSP formé de la seule contrainte* $x^3 - 9x + 4 = 0$ *où* $dom(x) = [0, +\infty[$ *initialement. Par propagation on obtient* $dom(x) = [0.45, 2.74]$*. La variable* x *est une variable coupe-cycle et en divisant son domaine en* $dom(x) = [0.45, 1.6[$ *et* $dom(x) = [1.6, 2.74]$*, on obtient les solutions globales* $x = 0.45$ *et* $x = 2.74$ *sans diviser davantage les domaines.*

Le problème principal de cette méthode d'éclatement est la détermination d'un critère de terminaison adapté (la condition d'avoir un domaine de variable coupe-cycle réduit à un singleton n'est pas une condition nécessaire [Hyvönen92]). De plus, même si les solutions sont en nombre fini, il se peut que de nombreux éclatements aient été faits avant que l'élimination de certains sous-domaines s'opère, ce qui rend la méthode lourde et coûteuse.

Il convient donc d'éviter le plus possible le recours à cette méthode. Dans ce but, Hyvönen a proposé une méthode appelé *propagation de la tolérance locale* qui consiste à :

1. déterminer les variables qui apparaissent dans les cycles de l'ICSP ;
2. exprimer chacune de ces variables en fonction uniquement des variables qui ne sont pas dans un cycle ;
3. éliminer, parmi les règles de maintien de la consistance associées aux contraintes, celles qui permettent le calcul d'une variable de cycle ;
4. ajouter aux règles de maintenance qui permettent le calcul des variables n'appartenant pas à un cycle, les règles obtenues en 2).

On obtient finalement un ensemble de règles de maintenance qui permet toujours de trouver une solution.

Exemple 7 (d'après [Hyvönen92]) *Soit l'ICSP :*

$$\begin{cases} c_1 : & x + t = y \\ c_2 : & y + t = z \end{cases}$$

En appliquant la propagation de la tolérance locale :

1. l'ensemble des variables de cycle est $\{y, t\}$
2. on a $y = (x + z)/2$ et $t = (z - x)/2$
3. on peut éliminer les règles $y = x + t, y = z - t, t = y - x, t = z - y$
4. les règles à considérer à présent sont : $y = (x + z)/2, t = (z - x)/2, x = y - t, z = y + t$

Avec $dom(x) = [1, 1], dom(y) = [1, 11], dom(t) = [0, 10], dom(z) = [11, 11]$, on trouve la solution $(x = 1, y = 6, t = 5, z = 11)$

Malheureusement, le recours à cette méthode nécessite de disposer d'un résolveur algébrique de type [Davenport et al.93], mais surtout, il n'est pas toujours possible de réaliser l'étape 2.

Pour les CSP à contraintes non linéaires, bien qu'il soit parfois possible de les résoudre simplement [Colmerauer93], certains systèmes spécialisés, tels que RISC-CLP, CAL [Hong93], ont recours à des méthodes algébriques de résolution telles que les *bases de Gröbner* [Buchberger85] ou *la décomposition algébrique cylindrique partielle* [Collins75]. D'autres ont recours à des techniques de l'arithmétique des intervalles telles que la *méthode itérative de Newton* comme `Newton` sur la base de CLP(BNR), ou l'*optimisation globale* combinée avec le *splitting* comme INC++, pour déterminer les solutions. Dans ce dernier système, une part importante est consacrée au calcul des plus petits intervalles contenant les solutions avant de se lancer dans une résolution.

4.5.9 Conclusion

Cette section a présenté les principes de la programmation par contraintes sur des intervalles dont l'origine est le traitement de CSP à variables réelles et à domaines continus. La théorie des CSP à intervalles s'appuie sur les règles de calcul d'intervalles issues de l'arithmétique des intervalles. Si l'arc-consistance n'est pas virtuellement possible sur les domaines réels continus, les langages traitant les ICSP atteignent des degrés de consistance plus ou moins forts selon qu'ils prennent en charge des intervalles simples ou des unions d'intervalles. Hormis les règles de base, l'arithmétique des intervalles fournit également des méthodes précises mais coûteuses de l'extension des intervalles. La propagation est l'algorithme clef de la programmation par contraintes sur des intervalles. Elle est destinée à établir la consistance de l'ICSP. Quant à la résolution en domaines réels continus, elle se base sur un éclatement des domaines et sur la résolution des sous-ICSP résultants de cette division. Des techniques plus performantes existent, comme la propagation de la tolérance locale, mais elles ne peuvent pas toujours être mises en œuvre.

À ce stade de la présentation, les techniques relatives aux CSP ont été passées en revue quelles que soient la finitude et la continuité de leurs domaines.

Parce que la dynamicité (propriété d'être modifiable à n'importe quel moment) est un caractère essentiel des bases de connaissances gérées par les SRPO, nous nous devons de nous intéresser à la façon d'appréhender cette propriété dans le contexte des CSP.

4.6 Les CSP dynamiques

Dans des applications réelles, les connaissances relatives à un problème, décrites en termes de CSP, peuvent évoluer avec le temps et conduisent alors à modifier l'énoncé de ce problème en ajoutant ou en supprimant des contraintes. Dans certains cas, il se peut que l'on n'ait du problème modélisé qu'une connaissance succincte, et qu'une démarche pragmatique basée sur l'ajout ou le retrait de contraintes soit nécessaire afin d'affiner la description du problème. De même, une fois le problème défini et résolu, il est possible que l'on constate, soit qu'il est *sur-contraint* et qu'il n'existe pas de solution, soit qu'il est *sous-contraint* et qu'il existe de nombreuses (voire une infinité de) solutions. Dans le premier cas, on peut souhaiter supprimer une ou plusieurs contraintes du problème afin de tenter de le rendre résoluble. Dans le second cas, on peut souhaiter ajouter une contrainte au problème qui tend à réduire l'ensemble des solutions.

Ces situations illustrent la nécessité, pour un système gérant des contraintes et modélisant des problèmes sujets à d'éventuelles modifications, de prendre en charge le caractère dynamique des CSP, c'est-à-dire, la possibilité d'effectuer à tout moment un ajout ou un retrait de contrainte. La définition d'un CSP *dynamique* est donnée à la section 4.6.1.

Face à la modification d'un CSP, le système peut considérer le CSP modifié comme un nouveau CSP, indépendant du précédent, et appliquer sur lui l'arsenal de techniques qui permettront de le résoudre. Mais, il est clair que cette façon de procéder est particulièrement inefficace car elle fait abstraction de l'évolution du CSP, de son passage d'un état à un autre, à travers les modifications successives qui lui sont apportées. Ainsi, pour un ajout de contrainte, il est souhaitable de disposer d'un système *incrémental* capable de propager la nouvelle contrainte sans avoir à reposer les contraintes précédentes. De même, pour un retrait de contrainte, il est souhaitable de disposer d'un système *décrémental* capable de supprimer la contrainte sans que cela consiste à reposer l'ensemble des contraintes du CSP avant le retrait, amputé de la contrainte supprimée. Pour cela, le système devra mémoriser ou être capable de retrouver les réductions opérées sur les domaines des variables au moment de la pose de la contrainte à supprimer. Ces principes régissent les diverses adaptations des algorithmes d'arc-consistance aux CSP dynamiques (*cf.* section 4.6.2).

Le passage d'un état à un autre d'un CSP à travers l'ajout ou le retrait d'une contrainte est également exploité dans la recherche d'une solution. Lorsqu'un CSP possède une solution avant l'ajout d'une contrainte, on cherche à déterminer, pour le nouveau CSP, une solution qui est "proche" de l'ancienne (*cf.* section 4.6.3).

4.6.1 Définition

Définition 20 (d'après [Dechter et al.88a] et [Janssen et al.88]) *Un **CSP dynamique** est une séquence $P_0, P_1, \ldots, P_i, P_{i+1}, \ldots$, de CSP statiques telle que deux CSP successifs $P_{j+1} = (X, D, C_{j+1})$ et $P_j = (X, D, C_j)$ vérifient soit $C_{j+1} = C_j + c$ (on dit que P_{j+1} est le résultat de l'**ajout** de la contrainte c à P_j), soit $C_{j+1} = C_j - c$ (on dit que P_{j+1} est le résultat du **retrait** de la contrainte c à P_j).*

Dans cette définition, l'ensemble X des variables impliquées par les contraintes est supposé connu. De même, l'ensemble D des domaines ne varie pas d'un CSP à l'autre, seuls les ensembles de contraintes déterminent en extension des ensembles de solutions qui varient.

Un ajout de contrainte est appelé *restriction*. Un retrait de contraintes est appelé *relaxation*.

Comme pour les CSP statiques à domaines finis, les études sur les CSP dynamiques à domaines finis portent, d'un côté, sur les algorithmes de filtrage (principalement d'arc-consistance) et, de l'autre, sur la recherche de solutions.

4.6.2 Arc-consistance dans les CSP dynamiques

La réalisation d'un algorithme d'arc-consistance dans les CSP dynamiques consiste à choisir un algorithme d'arc-consistance classique parmi les sept existants et à l'adapter à l'ajout et au retrait d'une contrainte.

- L'ajout d'une contrainte est la modification qui pose le moins de problèmes. Il suffit de modifier légèrement les algorithmes d'arc-consistance originaux établis pour des CSP statiques afin qu'ils deviennent incrémentaux.
- Le retrait d'une contrainte est l'opération la plus délicate. La relaxation consiste à supprimer une contrainte du CSP. Or, la contrainte à supprimer a pu être, au moment de sa pose mais aussi ultérieurement lors d'une propagation, la cause de la suppression de valeurs dans les domaines des variables qu'elle implique. De même, par propagation, les suppressions de valeurs qu'elle a occasionnées ont pu entraîner la suppression d'autres valeurs dans les domaines d'autres variables du réseau. Puisque la contrainte est supprimée du réseau, il faut veiller à réhabiliter les valeurs qu'elle a fait disparaître et s'assurer que ces valeurs sont toujours supportées par les autres contraintes avant de les réintégrer définitivement dans les domaines. La principale difficulté de la relaxation d'une contrainte est donc de déterminer l'ensemble des valeurs éliminées par la contrainte supprimée.

Une des premières solutions proposées par Philippe Janssen [Janssen90] pour assurer le maintien de l'arc-consistance consiste à mémoriser le résultat de chacun des filtrages opérés lors de la séquence $P_0, P_1, \ldots, P_{i+1}$ d'un CSP dynamique. Lors de la suppression d'un ensemble de contraintes[7] qui

[7] Qui correspond à une suite de suppressions d'une seule contrainte.

détermine le CSP P_{i+2}, on recherche le CSP P_j avec $j < i + 2$ qui est le plus proche de P_{i+2}. Le résultat du filtrage du problème P_j sert alors de point de départ au filtrage de P_{i+2}. Cette solution est jugée, par son auteur même, peu satisfaisante car elle ne mémorise que les résultats finals du filtrage (la fermeture par arc-consistance).

S'appuyant sur cette constatation, Philippe Jégou [Jégou91] a proposé d'enrichir la connaissance sur la cause exacte de la suppression d'une valeur lors d'un filtrage. Il s'agit ici de rendre compte de l'ensemble des contraintes à l'origine de la suppression de la valeur et des valeurs qui soutenaient cette valeur et qui ont disparu elles aussi. Cette approche vise donc à informer l'utilisateur sur les raisons d'une suppression lors d'un ajout de contraintes mais aussi à l'assister pour retrouver un état consistant pour un CSP dynamique devenu inconsistant. La méthode proposée consiste en une propagation d'*environnements* le long d'un graphe de propagation. Des environnements minimaux de suppression sont associés à chacun des sommets représentant les valeurs des variables. Ils contiennent les ensembles de contraintes qui ne soutiennent pas cette valeur. De même, des environnements minimaux sont associés aux sommets qui représentent les contraintes dans lesquelles chaque valeur d'une variable intervient.

Par la gestion des événements qu'elle met en place, l'approche précédente est semblable à celle des systèmes de type ATMS [Kleer86].

Nous ne détaillons pas ici les différentes méthodes de mises à jour des environnements qui ont lieu lors de l'ajout d'une contrainte donnant lieu à des suppressions de valeurs. La méthode de propagation des environnements parcourt le graphe de propagation en complétant de manière minimale les environnements de façon à connaître, pour chaque valeur supprimée, les causes de sa suppression. En tenant à jour ces environnements, le retrait d'une contrainte peut se faire simplement. La réhabilitation des valeurs appartenant à la fermeture par arc-consistance du nouveau CSP consiste à comparer les environnements de suppression avec l'environnement courant. Si ce dernier ne contient aucun environnement de suppression, alors la valeur figure bien dans le domaine de sa variable. Dans le cas contraire, la valeur ne peut être réintégrée.

Si la complexité des différentes méthodes de mises à jour des environnements – liée à la cardinalité maximum des environnements – est prohibitive devant le coût même de l'application de l'algorithme AC-4 lors des diverses modifications subies par un CSP dynamique, son auteur rappelle que son intérêt réside dans l'assistance qu'elle peut apporter à son utilisateur en fournissant les causes effectives des suppressions de valeurs. Il est possible de trouver un compromis temps de réponse du système/quantité et précision des informations fournies, en limitant la taille des environnements. Ceci se fait alors au détriment de la validité globale de la méthode (des valeurs n'étant pas considérées comme supprimées alors qu'elles le seraient sans cette limitation).

Se basant sur une approche de type TMS, c'est-à-dire sur l'utilisation de *justifications*, la première véritable adaptation d'un algorithme d'arc-consistance aux CSP dynamiques binaires a été réalisée par Christian Bessière [Bessière91] sur l'algorithme AC-4 [Mohr et al.86] sous le nom de DnAC-4.

Si AC-4 peut être adapté facilement à l'ajout incrémental de contraintes, il en va autrement pour la relaxation car l'algorithme n'a aucune mémoire des raisons des retraits des valeurs. L'idée de Bessière est de conserver, pour chaque valeur retirée, la trace ou justification de la première contrainte à l'origine de la suppression. Au moment de la relaxation, on peut alors, à l'aide des justifications, retrouver les valeurs candidates à une réintégration, c'est-à-dire celles dont la suppression est directement ou indirectement due à la contrainte. Afin de garantir que le CSP soit arc-consistant *maximal*, les justifications vérifient la propriété d'être bien *fondées* et sont donc acycliques (aucune valeur ne doit son retrait à elle-même).

Définition 21 (d'après [Bessière92]) *Les justifications mémorisées par DnAC-4 sont bien fondées si et seulement si dans tout ensemble E de valeurs retirées, il existe une valeur a pour la variable i telle que, sa justification étant la contrainte C_{ij}, aucune des valeurs de j supports de la valeur a de i pour la contrainte C_{ij} n'appartient à E.*

Définition 22 (d'après [Bessière92]) *Soit R l'ensemble des valeurs retirées. Les justifications mémorisées par DnAC-4 sont bien fondées si et seulement si il existe un ordre $<_o$ sur R tel que pour toute valeur a de la variable i appartenant à R; si C_{ij} est la contrainte justifiant le retrait de a, alors pour toute valeur b de j telle que $(i = a, j = b) \in C_{ij}, b <_o a$.*

DnAC-4 reprend à peu près les mêmes structures de données auxquelles AC-4 doit son efficacité, en y ajoutant les justifications de types $justif(i, a) = j$ lorsque le retrait de la valeur a pour la variable i est dû à la contrainte C_{ij} et en modifiant l'ensemble S_{jb} des couples variable-valeur (i, a) supportés par le couple (j, b).

Il comporte des procédures d'initialisation, d'ajout et de suppression de contrainte.

La procédure d'ajout d'une contrainte construit la liste des valeurs sans support dans les domaines des deux variables impliquées, puis propage récursivement la conséquence de ces retraits aux variables attenantes en mettant à jour la table des justifications.

La procédure de suppression d'une contrainte agit en trois phases. Elle construit d'abord la liste des valeurs dont le retrait était directement dû à la contrainte (cette information est fournie par les justifications). Puis, après avoir ajouté ces valeurs aux domaines concernés, elle informe les variables attenantes de la réhabilitation de ces valeurs en mettant à jour, le cas échéant, leur compteur de supports. Si, pour une des valeurs de ces variables, la contrainte était la cause de sa suppression, alors cette valeur doit être à son tour ajoutée. S'il s'avère qu'une valeur à rajouter reste sans support, elle est mise dans la liste des valeurs sans support. Enfin, la liste des valeurs sans support est parcourue comme lors de la procédure d'ajout afin de propager récursivement leur retrait.

DnAC-4 a une complexité en temps comme en espace de $O(ed^2)$ où e est le nombre des contraintes et d la taille maximum des domaines, ce qui est similaire à la complexité de AC-4 même si DnAC-4 réalise plus de travail que AC-4 lors d'une restriction (pour la mise à jour de ses structures de données propres). Lors d'une relaxation, il examine moins de valeurs qu'un AC-4 relancé sur le nouveau CSP. Bessière signale que DnAC-4 est d'autant plus efficace que les contraintes à retirer sont les plus récemment ajoutées et que le CSP est faiblement contraint.

De même que AC-4 a été généralisé aux CSP n-aires sous le nom de GAC-4 [Mohr et al.88], Bessière a proposé une généralisation de DnAC-4 appelée DnGAC-4 dont la complexité en espace comme en temps est semblable à celle de GAC-4, et de l'ordre de $O(ed^r r)$ où r est l'arité maximale des contraintes.

S'il a été prouvé qu'AC-4 est un algorithme optimal en temps, il n'en va pas de même de sa complexité en espace mémoire. AC-6 [Bessière et al.93], en instaurant un ordre – souvent naturel – sur les domaines, permet de réduire la taille des ensembles supports à un élément (la première valeur support trouvée dans le domaine). Dès lors, la complexité en espace est améliorée et devient, pour AC-6, $O(ed)$. Les idées de AC-6 ont été adaptées aux CSP dynamiques [Debruyne94] dans un algorithme appelé DnAC-6 qui préserve les complexités en temps et en espace de AC-6. De principe de fonctionnement relativement différent de DnAC-4, principalement parce qu'il ne dispose pas de compteurs de valeurs supports pour une valeur et qu'il doit scruter les listes des valeurs présentes et absentes d'un domaine, DnAC-6 présente des structures de données moins volumineuses que celles de DnAC-4. Ce dernier compensant ce défaut sur les CSP très contraints en effectuant moins de tests de consistance. À ce jour, nous n'avons pas connaissance d'une généralisation de AC-6 ou de DnAC-6 aux CSP n-aires.

Prenant le contre-pied des approches TMS présentées ci-dessus, Bertrand Neveu et Pierre Berlandier [Neveu et al.94, Berlandier94] ont proposé un algorithme d'arc-consistance pour les CSP dynamiques binaires non basé sur l'enregistrement de justifications des retraits, appelé AC|DC (Arc Consistency for Dynamic Constraint Problems). L'objectif avoué de ses auteurs est de ne pas alourdir l'emploi d'un algorithme d'arc-consistance par la gestion de structures d'enregistrements qui peuvent devenir imposantes sur les CSP de grande taille.

L'idée ici est de ne pas avoir à stocker les informations relatives à un retrait de valeurs lors d'une restriction ; on pourra donc employer n'importe quel algorithme d'arc-consistance lors de l'ajout d'une contrainte. En ce qui concerne la relaxation, il s'agit de proposer un sur-ensemble des valeurs "réhabilitables" et, comme pour les approches précédentes, de tester si les valeurs proposées ont des supports afin de les réintégrer dans leur domaine et de propager leur réintégration.

Au moment de la suppression d'une contrainte, l'ensemble des valeurs réhabilitables est construit en prenant, dans la différence entre le domaine initial d'une variable et son domaine courant, les valeurs qui n'ont pas de support pour cette contrainte. En effet, si elles en ont un, c'est qu'elles ont été retirées par une autre contrainte. Une fois construite cette liste de valeurs réhabilitables circonscrite à la contrainte supprimée, on cherche à la compléter en propageant récursivement cette réhabilitation provisoire auprès des valeurs écartées et non réhabilitables des variables attenantes.

Ayant obtenu la liste complète des valeurs réhabilitables, une procédure d'arc-consistance est chargée de ne conserver parmi elles que celles qui ont un support pour chaque contrainte et qui seront de nouveau admises dans leur domaine.

La complexité en espace de l'algorithme AC|DC est $O(e + nd)$. La complexité en temps est théoriquement comparable à celle d'AC-3 dont le schéma est utilisé lors du filtrage des valeurs réhabilitables, donc $O(ed^3)$. En règle générale, la taille des ensembles de valeurs propageables et réhabilitables confère de meilleures performances à l'algorithme.

Si sur les relaxations AC|DC est moins performant que DnAC-4 et a, dans certains cas, un moins bon comportement que AC3, ses auteurs soulignent qu'il a l'avantage de proposer un pas de relaxation beaucoup plus fin (rappelons-nous que DnAC-4 ne peut relaxer qu'une contrainte entière) en dictant la réhabilitation de certaines valeurs seulement. De même, l'extensibilité d'AC|DC est soulignée par la possibilité de choisir un algorithme d'arc-consistance plus performant qu'AC-3 et ne reposant pas sur des structures de données globales comme AC-5 ou AC-6, à la fois pour l'ajout de contrainte et pour le filtrage lors du retrait.

4.6.3 Résolution dans les CSP dynamiques

Dans le contexte d'un CSP dynamique, le recours aux méthodes classiques de résolution présentées précédemment est toujours possible. En effet, à chaque modification (par ajout ou retrait) du CSP précédent dans la séquence, on peut considérer le nouveau CSP comme un CSP statique standard, indépendant du CSP précédent. Cependant, cette méthode comporte deux défauts importants soulignés dans [Verfaillie et al.94a]

1. l'inefficacité : en considérant le nouveau CSP comme isolé, on recommence un processus de résolution dont certaines étapes ont déjà été effectuées lors de la résolution du CSP précédent. Ce défaut est assez critique dans les applications en temps réel où le temps imparti pour la résolution est limité ;
2. l'instabilité des solutions des CSP successifs qui peut s'avérer gênante dans le cadre d'une activité de planification ou lors d'un processus interactif de construction.

De plus, lors de la modification d'un CSP P_i en un CSP P_{i+1}, deux cas peuvent se présenter :

1. P_i est inconsistant (n'a pas de solution) :
    - L'ajout d'une contrainte conduit immédiatement à P_{i+1} inconsistant et n'a guère d'intérêt.
    - Le retrait d'une contrainte peut rendre P_{i+1} consistant.
2. P_i est consistant (a au moins une solution) :
    - Le retrait d'une contrainte transforme P_i en P_{i+1} qui lui aussi est consistant.
    - L'ajout d'une contrainte peut rendre P_{i+1} inconsistant.

Afin d'améliorer l'efficacité de la recherche de solution lors d'un ajout ou du retrait d'une contrainte dans un CSP dynamique, il est donc conseillé de réutiliser le raisonnement et/ou la solution précédente [Verfaillie93] en remarquant que :

- lors d'un ajout de contrainte, la solution du CSP précédent n'est peut être plus solution du CSP actuel, mais les échecs du *Backtrack* ainsi que les retraits de valeurs dus au filtrage effectués lors du CSP précédent seront reproduits pour le nouveau CSP. Il faut donc enregistrer les étapes du raisonnement précédent.
- lors d'un retrait de contrainte, la solution précédente est forcément solution du CSP actuel, le raisonnement n'est réutilisable que si ses différentes étapes ont été enregistrées.

Concernant la stabilité des solutions successives, elle induit une mesure de distance entre deux solutions – le plus communément, on choisit le nombre de variables instanciées différentes.

Les travaux visant l'efficacité et la stabilité dans la résolution d'un CSP dynamique se divisent aujourd'hui en quatre classes de méthodes dites de *révision de solution* [Verfaillie et al.94b].

1. les méthodes *heuristiques* [Hentenryck90] qui consistent à utiliser pour le CSP actuel comme instanciation (complète ou non), la première solution déterminée pour un CSP moins contraint ;
2. les méthodes de *réparations locales* [Minton et al.92, Selman et al.92, Verfaillie93, Verfaillie et al.94b] qui consistent à partir de n'importe quelle instanciation (complète ou non) et de la réparer par une séquence de modifications locales (une valeur d'une variable du CSP actuel à chaque étape) ;
3. les méthodes d'*enregistrement de contraintes* [Hentenryck et al.91, Schiex et al.93, Verfaillie et al.94a] qui, basées sur une approche de type ATMS, consistent à enregistrer lors du *Backtrack* (et ses améliorations : *Backjump, Backtrack* dynamique...) des *nogoods* (couples instanciation/ensemble de contraintes violées par l'instanciation proposée), afin d'éviter de reproduire les mêmes tests menant à un échec lors de la résolution du CSP actuel et, lors de la résolution d'un prochain CSP, si la même configuration apparaît ;
4. les méthodes de *calcul de la distance minimale entre deux solutions successives* [Bellicha93, Trombettoni92]

Les deux premières classes de méthodes visent à augmenter l'efficacité et la stabilité. La troisième classe est dédiée à la recherche d'efficacité. La quatrième s'intéresse avant tout à la solution la plus proche de la solution précédente.

Verfaillie et Schiex ont comparé leur méthode de réparation locale (changements locaux) avec la méthode de réparation heuristique de Minton, et diverses formes de *Backtrack* (standard, *Backjumping, Backtrack* dynamique) avec ou sans *Forward-Checking* sur des CSP binaires dynamiques générés aléatoirement. Il en résulte que la méthode de changements locaux est bien adaptée aux CSP de grandes tailles, sous-contraints, et sujets à de fréquents changements.

Une autre approche intéressante que nous ne classons pas dans les quatre catégories précédentes est l'approche logique choisie par Philippe Jégou qui propose de représenter les CSP sous la forme de *diagrammes binaires de décision* [Bryant86], adaptés à la dynamicité. Le maintien de la consistance dans les CSP dynamiques, ainsi que la recherche de solutions optimales, reviennent à la recherche d'un chemin dans le diagramme binaire de décision. Cependant, pour l'instant, cette étude n'a été réalisée que dans le cadre de variables de la logique propositionnelle.

4.6.4 Conclusion

Les CSP dynamiques sont particulièrement adaptés à des applications ou les informations descriptives du problème sont sujettes à des modifications observées – l'environnement du problème évolue – ou provoquées – par l'utilisateur pour explorer certaines configurations. Comme pour les CSP statiques, l'application d'un filtrage pour établir l'arc-consistance constitue un pré-traitement d'un coût raisonnable avant d'entamer une phase de résolution. Deux types d'algorithmes d'arc-consistance adaptés aux CSP dynamiques existent. D'un côté, les adaptations des algorithmes performants établis pour les CSP statiques sont basées sur des justifications de type TMS des retraits de valeurs ; efficaces, elles s'avèrent toutefois gourmandes en espace mémoire. De l'autre, un algorithme qui n'a pas recours aux justifications, moins efficace, mais plus extensible et moins gourmand que les précédents.

Dans le cadre des CSP dynamiques, les méthodes de révision de solutions permettent à la fois de réduire le temps de recherche d'une solution et de minimiser le travail à effectuer pour déterminer la solution du CSP actuel à partir du CSP précédent. Ces méthodes, à base de *Backtrack* plus ou moins performant, ont recours à des techniques empruntées aux ATMS, et/ou procèdent par modifications locales d'une instanciation non consistante.

À notre connaissance, les algorithmes d'arc-consistance ou les méthodes de révision de solutions n'ont pas été explorés dans le domaine des CSP dynamiques à intervalles continus ou infinis. En ce qui concerne les méthodes de résolution du type révision de solutions, ce n'est pas surprenant car cette approche est ici difficilement applicable. En effet, dans le cas de CSP dynamiques à domaines finis, elle se base sur une énumération des domaines à l'aide d'un *Backtrack* qui n'est pas envisageable dans le cas d'intervalles continus ou infinis où la résolution s'effectue soit par *splitting*, soit par méthodes adaptées de résolution. En ce qui concerne l'arc-consistance, ou un maintien de consistance s'en approchant, une justification des retraits n'est pas non plus applicable pour chacune des valeurs supprimées. En revanche, il reste à explorer le cas où l'adaptation d'un algorithme d'arc-consistance pour les CSP à intervalles dynamiques peut être faite, dans la représentation choisie n'impliquant que les bornes.

4.7 Conclusion

Le spectre des différents types de CSP et des différentes techniques de maintien de consistance et de résolution étant maintenant parcouru, nous sommes en mesure d'effectuer les choix qui vont présider à l'intégration d'un raisonnement par contraintes dans TROPES. Toutefois, il faut garder à l'esprit que cette intégration doit se soumettre aux principes de la représentation de connaissances par objets et non l'inverse. Avant de présenter ces choix, il convient de s'intéresser aux systèmes qui proposent également un couplage objets/contraintes afin d'observer des constantes d'intégration, mais aussi certainement des différences, voire même des manques, que nous nous proposons de combler dans le cadre de TROPES.

Chapitre 5

Objets et contraintes

Le chapitre 2 a décrit les différents types de langages élaborés autour de la notion d'"objet". Le chapitre 3 a présenté un système de représentation par objets en particulier, appelé TROPES et a montré le besoin d'exprimer et de gérer des contraintes dans TROPES. C'est pourquoi dans le chapitre précédent, ont été abordés les divers aspects de la théorie de la programmation par contraintes. Aussi, nous étudions ici les principaux systèmes qui allient objets et contraintes.

Nous écartons de ce chapitre les systèmes ou langages de contraintes qui n'utilisent pas la notion d'*objet*, ni en tant qu'entité de programmation orientée-objet, ni en tant qu'entité de représentation. Dans cette catégorie ignorée ici se trouvent, notamment, les langages de programmation logique par contraintes évoqués au premier chapitre.

On peut répertorier trois approches distinctes dans les systèmes ou langages alliant objets et contraintes :

1. étendre un langage de programmation orienté-objet existant avec des contraintes : THINGLAB, ILOG-SOLVER... ;
2. construire un nouveau langage de programmation basé sur les principes de la programmation orientée-objet et sur la programmation par contraintes : KALEIDOSCOPE, GARNET... ;
3. proposer une boîte à outils de programmation par contraintes en exploitant les principes de la programmation orientée-objet (représentation classe/instance, hiérarchies, abstraction) : CSPOO, PROSE .

Deux motivations différentes mais pas forcément incompatibles sont à l'origine de ces trois approches :

1. contraindre les objets utilisés dans un langage de programmation orientée-objet est l'objectif déclaré des deux premières approches ;
2. définir les contraintes par des objets selon les standards de la programmation orientée-objet est l'objectif déclaré de la troisième approche.

Dans le premier cas, ce sont les atouts de la programmation par contraintes – déclarativité et puissance de raisonnement – qu'il s'agit d'exploiter dans un environnement objet. L'idée est d'établir des relations (contraintes) entre les objets et d'assurer la maintenance et/ou la résolution de ces liens par un moteur de contraintes, c'est-à-dire un ensemble d'algorithmes empruntés à la programmation par contraintes. Autrement dit, le paradigme "objet" (des langages de programmation orientée-objet aux langages de représentation) se dote avec les contraintes d'un outil descriptif de liens entre objets, puis, selon les capacités du résolveur de contraintes, sinon d'un outil de résolution de problèmes, d'un outil de maintenance de consistance de ces liens.

Dans le second cas, ce sont les atouts de la programmation orientée-objet – organisation hiérarchique des objets, abstraction de données – qui sont sollicités pour la définition de langages ou de boîtes à outils de la programmation par contraintes. L'idée est de définir les contraintes en termes de classes et d'instances et de leur associer, en tant qu'objets de programmation, des méthodes (de consistance ou de résolution). Autrement dit, les langages (ou bibliothèques) de contraintes se dotent avec les objets d'un outil réputé de représentation, d'organisation et de factorisation de connaissances.

À travers ces deux visions de l'alliance objets/contraintes, il apparaît que chacun des deux paradigmes exposés jusqu'à présent peut tirer profit de l'autre. Loin d'être opposées ces deux visions deviennent complémentaires dans certains systèmes où, non seulement les objets sont contraignables, mais les contraintes sont également des objets. C'est le cas dans ThingLab mais ça peut l'être aussi pour tout langage (ou bibliothèque) de contraintes (*cf.* troisième approche) couplé avec un langage à objets.

Dans la première partie (*cf.* section 5.1) sont présentés les principaux représentants des trois approches distinguées. Pour chacun des systèmes, nous nous intéresserons aux différents aspects de l'alliance objets/contraintes qu'ils proposent :

- le domaine de calcul appréhendé par le moteur de contraintes (c'est-à-dire sur quel(s) type(s) de variables les contraintes peuvent porter) qui définit souvent aussi la finalité du système (les applications pour lesquelles il est le plus approprié)
- les capacités de résolution ou de maintenance offertes par le système qui, avec le domaine de calcul, définissent en général la finalité du système (les applications pour lesquelles il est le plus approprié)
- la nature du couplage objets/contraintes (pour les systèmes des deux premières approches, seulement)
- l'expression des contraintes dans les objets (pour les systèmes des deux premières approches seulement)
- la représentation des contraintes par des objets (pour les systèmes adhérant – uniquement ou non – à la troisième approche seulement)

En conclusion (*cf.* section 5.2), nous résumons les raisons et les avantages de lier objets et contraintes tels qu'ils nous sont apparus à l'étude des systèmes existants. Nous dressons enfin la liste des questions soulevées par un couplage RPO/contraintes auxquelles les chapitres suivants répondront.

5.1 Systèmes alliant objets et contraintes

5.1.1 Introduction

Nous reprenons ici la classification proposée plus haut en présentant les principaux représentants des langages à objets avec contraintes et des langages de contraintes avec objets. Nous rappelons que les catégories exhibées ne sont pas pas exclusives et que les idées prônées par la troisième approche peuvent être employées dans les deux autres.

5.1.2 Les langages à objets avec des contraintes

5.1.2.1 La famille ThingLab

ThingLab [Borning81] est un langage de programmation par contraintes, créé par Alan Borning, dont l'objectif est d'offrir un atelier de simulations de modèles dynamiques, notamment en géométrie ou en physique. ThingLab est une extension du langage Smalltalk-76 [Ingals78] dans lequel contraintes, hiérarchies d'objets composites et héritage sont combinés. Il est doté d'une interface graphique interactive qui permet à l'utilisateur de construire des objets, de les visualiser sous plusieurs formes, de les modifier ou de les déplacer à l'aide de la souris. Il s'inspire principalement des systèmes Sketchpad [Sutherland63] et Constraints [Sussman et al.80].

Trois types d'objets existent en ThingLab : les classes, les instances et les prototypes. Le prototype d'une classe est une instance particulière qui contient des valeurs par défaut ou à partir de laquelle la classe a été construite. Un objet ThingLab est composé d'autres objets – eux mêmes composés ou primitifs – qui forment ces parties (*cf.* tableau 5.1). Chaque partie est décrite par un nom, sa classe d'appartenance, un ensemble de contraintes et un ensemble de *fusions*. Le long des liens de spécialisation, une sous-classe hérite des contraintes de ses sur-classes.

Une contrainte est décrite par une *règle* (un prédicat qui indique si la contrainte est satisfaite ou non) et une liste de méthodes, ordonnées par préférence. Une contrainte définie pour une classe est appliquée à toutes les instances de cette classe. La règle exprime une relation entre des parties ou sous-parties atteignables à partir de l'objet par un *chemin*. Dès qu'une contrainte est définie en ThingLab, elle doit être satisfaite. La maintenance se fait à l'aide d'une propagation locale qui consiste à choisir pour la contrainte à satisfaire, une de ces méthodes (la première ou celle qui peut être activée). Les méthodes, en tant que procédures, sont applicables à tous les types de base du langage, aussi l'éventail des contraintes exprimables est-il large. Il est possible de déclarer la partie d'un objet en *référence* afin d'interdire sa modification, ou encore de l'ancrer comme constante fixe pour toutes les contraintes. Il est possible d'ajouter ou de modifier des contraintes existantes par la définition de nouvelles classes ou prototypes. Lorsque des parties sont fusionnées, il est établi

Class MidPointLine	**Class Quadrilateral**
Superclasses	**Superclasses**
Geometric Object	*Geometric Object*
Part Descriptions	**Part Descriptions**
line: *a Line*	*part1:* *a Line*
midpoint: *a Point*	*part2:* *a Line*
Constraints	*part3:* *a Line*
midpoint = (line point1 + line point2) / 2	*part4:* *a Line*
midpoint ← (line point1 + line point2) / 2	**Merges**
*line point1 ← (midpoint * 2 - line point2)*	*part2 point2 ≡ part3 point1*
*line point2 ← (midpoint * 2 - line point1)*	*part1 point1 ≡ part4 point2*
	part3 point2 ≡ part4 point1
	part1 point2 ≡ part2 point1

Tab. 5.1 - : Description de deux classes Thinglab. À gauche, la classe décrit le milieu d'une ligne ; la partie *Constraints* donne le prédicat et les règles assurant que le point est toujours au milieu de la ligne. À droite, la partie *Merges* décrit la fusion entre les sommets d'un quadrilatère.

entre elles des contraintes d'égalités. Dans une fusion, les parties impliquées sont remplacées par un seul objet, pour des raisons d'efficacité. Cette idée, déjà présente dans le système Sketchpad, permet un stockage plus compact des objets et accélère la satisfaction des contraintes impliquant les parties fusionnées.

La satisfaction des contraintes en ThingLab est incrémentale et s'effectue en deux phases :

1. une phase de planification durant laquelle sont regroupées toutes les contraintes concernées par la modification. Un plan de méthodes pour l'ajustement des valeurs des objets contraints est élaboré, chaque méthode de ce plan étant compilée.
2. une phase d'exécution qui exécute la méthode compilée avant de la ranger dans un dictionnaire où elle sera recherchée si la même modification est reproduite.

Il existe deux techniques pour construire le plan de méthodes :

- la propagation d'états connus : le résolveur cherche parmi les parties d'objets concernées par la modification, celles qui seront complètement connues à l'exécution (ou qui n'ont plus de degré de liberté). Les parties connues se propagent selon le principe qui veut que si un objet est connu alors ses parties sont connues et, inversement, si toutes les parties sont connues alors l'objet est connu.
- la propagation des degrés de liberté : le résolveur cherche une partie avec suffisamment de degrés de liberté (i.e. une seule contrainte portant sur elle) pour être modifiée. Ces parties sont ajustées les premières et leurs contraintes ne sont plus considérées. Il se peut alors qu'une autre partie ait à son tour suffisamment de degrés de liberté et le processus se poursuit.

Lorsque la planification rencontre un cycle (le résultat d'une méthode est employé pour recalculer une des entrées de cette méthode), il peut employer une méthode numérique de *relaxation*, limitée aux valeurs numériques et aux contraintes linéaires, qui consiste à minimiser l'erreur de satisfaction des contraintes. L'idée est de faire une supposition sur les valeurs des parties inconnues, de calculer l'erreur (par la méthode des moindres carrés) découlant de ces approximations, puis de recommencer jusqu'à obtenir une erreur minimale indiquée par la règle associée avec la contrainte. La relaxation est une méthode assez lente et coûteuse, aussi Borning suggère-t-il une autre technique empruntée à Steele et Sussman appelée *point de vue redondant*. Il s'agit ici de créer une contrainte redondante (en créant donc une nouvelle classe) et de fusionner les parties d'une instance de cette classe avec les objets du cycle.

ThingLab a été principalement utilisé dans des animations graphiques (Animus [Borning et al.86]) à base de contraintes temporelles. On peut lui reprocher de ne traiter que des contraintes fonctionnelles (à base d'égalité ou de fusions) et de s'appuyer seulement sur la propagation pour le maintien de solution, ce qui le rend fragile en cas de réseaux de contraintes circulaires non résolus par les méthodes de relaxation ou d'introduction de points de vue redondants. Dans ce cas, un simple *Backtrack* pourrait résoudre le problème et réajuster les valeurs des objets contraints. De plus, les contraintes sont définies à l'intérieur des classes : la même contrainte employée dans deux classes différentes (non liées par spécialisation directe ou indirecte) doit être décrite (règle + méthodes) deux fois. Cependant, ThingLab est un précurseur de nombreux langages à objets avec contraintes dont, ThingLab II et Kaleidoscope que nous présentons ci-dessous.

ThingLab II [Maloney et al.89] est le successeur de ThingLab. Écrit en SmallTalk-80 [Goldberg83], c'est également un système de programmation par contraintes orientée-objet dont la principale vocation est la construction d'interfaces-utilisateur.

Les objets dans ThingLab II sont des *Things* dont les parties sont inter-connectées par des contraintes. Une bibliothèque d'objets primitifs est fournie, à partir desquels l'utilisateur élabore ses objets. Parmi les objets primitifs, les variables à historique sont capables de conserver leurs états (valeurs) précédents permettant la déclaration de contraintes temporelles utiles, notamment pour les animations (Animus).

Les objets sont déclarés en tant que classes et sont constructibles directement à partir de l'interface graphique. Afin de minimiser les temps de réponse de l'interface, les concepteurs de ThingLab II proposent d'interpréter les plans de méthodes élaborés. La compilation est néanmoins possible lorsque les objets sont implémentés sous forme de *modules* [Freeman-Benson89].

L'une des autres différences de ThingLab II avec ThingLab est qu'il intègre un algorithme incrémental de satisfaction de contraintes conçu pour prendre en compte une *hiérarchie de contraintes* [Borning et al.87]. Dans une hiérarchie de contraintes, les contraintes sont regroupées dans des ensembles par niveaux de préférence (obligatoire, forte, faible...). Ainsi, dans un CSP(X, D, C), on a $C = \{C_0, C_1, \ldots, C_n\}$ où C_0 est l'ensemble des contraintes obligatoires, $C_1, \ldots, C_n$ sont les ensembles de contraintes ordonnées par ordre décroissant de préférence (fortes, faibles...).

Les contraintes obligatoires doivent être satisfaites alors que les autres (fortes, faibles, ...) peuvent ne pas l'être. Le but de l'algorithme, appelé *DeltaBlue*[1] [Freeman-Benson et al.90] est de fournir, lors de l'ajout ou du retrait d'une contrainte, la meilleure solution possible. Le critère de comparaison retenu ici est local et consiste à dire qu'une solution x du CSP à hiérarchie de contraintes est meilleure qu'une solution y si $\forall c \in C_1, \ldots, C_{k-1}$, x satisfait c si y satisfait c et, au niveau de préférence k, x satisfait au moins une contrainte de plus que y.

DeltaBlue travaille à partir d'un ensemble de contraintes hiérarchisées, d'un ensemble de variables, et de la solution courante représentée par un graphe acyclique de flots de données qui indique, pour chaque variable, quelle contrainte calcule sa valeur et, pour chaque contrainte, si elle est satisfaite ou non et quelle est la méthode utilisée.

Le principe de *DeltaBlue* est d'associer à chaque variable une information appelée *walkabout strenght* ou *force* (*cf.* figure 5.1). Lorsque la variable v est calculée par la méthode m d'une contrainte

[1] Position suggérée par ses auteurs après une classification des algorithmes de satisfaction de contraintes selon le spectre des couleurs.

c, sa force est le minimum entre le niveau de préférence de la contrainte c et les forces des entrées de la méthode m. Intuitivement, la force d'une variable est le niveau de la plus faible contrainte située en amont – que l'on peut donc atteindre en inversant les méthodes des contraintes qui y mènent – à révoquer (c'est-à-dire changer de méthode ou laisser insatisfaite) pour satisfaire une autre contrainte. Révoquer une contrainte consiste à choisir une autre méthode ou à laisser la contrainte non satisfaite s'il ne s'agit pas d'une contrainte obligatoire. Pour satisfaire une contrainte, *DeltaBlue* cherche une méthode de cette contrainte telle que la force de la variable de sortie de cette méthode soit plus faible que la force de la contrainte. Si une telle méthode n'existe pas, c'est que cette contrainte ne peut être satisfaite sans révoquer une contrainte de force supérieure (on aboutit à une solution moins bonne) ou égale (on aboutit à une solution aussi bonne). Lors de la suppression d'une contrainte satisfaite, la mise à jour des forces se fait vers les variables en aval où l'on tente de satisfaire les contraintes qui pourraient l'être. *DeltaBlue* est capable de détecter les cycles dans les

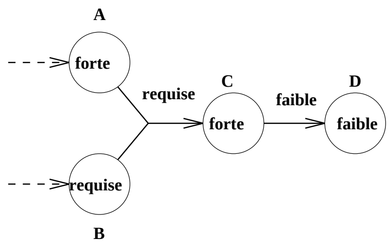


FIG. 5.1 - : La force de D est *faible* car la contrainte entre C et D est *faible*. La force de C est *forte* car A est *forte* et la contrainte entre A et C est *requise*.

flots de données, mais pas de les résoudre. Aussi, bien que THINGLAB II constitue une amélioration sensible, tant au point de vue de l'efficacité du temps de latence de l'interface graphique, que de la flexibilité des contraintes, il reste limité aux CSP sans cycles. Comme THINGLAB, il n'est capable que de propagation locale et ne gère que des contraintes fonctionnelles. Une caractéristique intéressante de THINGLAB II est la mise à la disposition de l'utilisateur d'outils pour analyser les contraintes et afficher les diverses façons (flots de données) de les résoudre.

Créé par Bjorn Freeman-Benson, dans l'équipe d'Alan Borning à l'Université de Washington, KALEIDOSCOPE [Freeman-Benson90] est un langage orienté-objet qui réalise la fusion de la programmation orientée-objet impérative et de la programmation par contraintes. L'idée est de permettre à l'utilisateur à la fois de définir des relations durables entre des objets et de décrire des séquences de relations entre les états successifs du programme. Ainsi, chaque variable garde un historique de ses valeurs. Le temps ne correspond pas à un cycle d'instruction mais est avancé à la fin de chaque instruction ; il est également possible de ne pas l'avancer entre deux instructions composant une instruction composée. La valeur d'une variable ne change qu'avec une affectation. KALEIDOSCOPE est à rapprocher de THINGLAB II par sa gestion d'une hiérarchie de contraintes à l'aide de l'algorithme *DeltaBlue*, et ne comporte pas d'algorithme de *Backtrack* pour la satisfaction. Les contraintes consistent en des égalités construites à partir d'opérateurs arithmétiques et booléens classiques ($+, -, =, < \ldots$). Des contraintes *complexes* (*cf.* tableau 5.2) peuvent être définies à l'intérieur des objets à l'aide de constructeurs. Un autre aspect intéressant du langage sont les *vues* (*cf.* tableau 5.3)– d'où KALEIDOSCOPE tire son nom – qui sont des objets du langage qui permettent à l'utilisateur non seulement de considérer un objet sous plusieurs points de vue (par exemple, une température en Celsius, Fahrenheit ou Kelvin), mais aussi de lier chacun des objets rassemblés sous une vue (ici, Celsius, Fahrenheit et Kelvin) par des contraintes de consistance. KALEIDOSCOPE est plus particulièrement adapté aux applications graphiques interactives où les objets évoluent au cours du temps. On peut faire à KALEIDOSCOPE les mêmes critiques que sur THINGLAB II concernant sa limitation aux contraintes fonctionnelles et l'absence de techniques de résolution (qui résulte, comme pour THINGLAB II d'un choix délibéré des créateurs de privilégier la maintenance incrémentale de solution à l'aide d'un algorithme de propagation locale inadapté aux cycles). La

`class Dash subclass of Object`
`   public var left, length, color;`
`   public virtvar right;`
`   initially`
`      always: rigth - left = length;`
`      always: length >= 1;`
`      always: length <= 127;`
`      weak color ← Black;`
`   end initially;`
`end Dash;`
	`class Point`
	`   constructor +(q, r)`
	`      always: self.x + q.x = r.x;`
	`      always: self.y + q.y = r.y;`
	`   end +;`
	`end Point;`

TAB. 5.2 - : Exemples de classes en KALÉIDOSCOPE. À gauche, déclaration de trois contraintes obligatoires *always* et d'une contrainte faible *weak*; à droite, définition d'une contrainte complexe.

```
view Celsius <--> Fahrenheit <--> Kelvin
   always: Celsius * 1.8 = Fahrenheit - 32;
   always: Celsius = Kelvin + 273;
end view;
```

TAB. 5.3 - : Définition d'une vue en KALÉIDOSCOPE

définition de contraintes et la prise en compte de vues constituent les originalités marquantes de ce langage relativement à l'intégration de contraintes dans les objets.

KALEIDOSCOPE90 a été suivi de deux nouvelles versions. Dans KALEIDOSCOPE91 [Freeman-Benson et al.92], l'historique des variables est réduit à deux valeurs, les contraintes sont compilées selon le principe des modules de THINGLAB II, un nouveau résolveur de contraintes permet de résoudre des contraintes de type (par exemple, `x + y = z`) en déterminant le constructeur de contraintes qui correspond à la valeur des variables (ici, `+(...)` sur les entiers si `x, y, z` sont entiers). KALEIDOSCOPE93 [Lopez et al.93] intègre des contraintes d'identité et s'ouvre à la programmation concurrente par contraintes. Surtout, il est basé sur un modèle de perturbation plutôt que de raffinement (une affectation de valeur à une variable est cette fois traitée comme un changement de valeur et non plus comme une contrainte supplémentaire qui peut ou non être encore satisfaite, le système le prend en compte et cherche la meilleure solution). Son résolveur est basé sur l'algorithme incrémental de propagation locale *SkyBlue* qui permet des contraintes à sorties multiples (maintenues par des méthodes qui permettent d'obtenir simultanément la valeur de plusieurs variables à partir d'une ou plusieurs variables d'entrée).

5.1.2.2 Garnet et Multi-Garnet

GARNET [Myers et al.92a, Myers et al.92b] est un environnement de développement d'interfaces-utilisateur dont les principales caractéristiques sont énumérées ci-dessous.

- Un système à objets basé sur une approche prototypique avec héritage structurel (lorsqu'un prototype est un *agrégat*, son instanciation entraîne l'instanciation de ses composants). Un prototype peut être modifié (ajout ou retrait d'attributs) simplement en un autre prototype.
- Un modèle de persistance d'objets, qui à chaque objet graphique de l'écran associe un objet en mémoire. Ce modèle libère l'utilisateur des tâches de maintenance. Notamment, lorsqu'un changement intervient sur l'un des attributs d'un objet graphique, il est immédiatement redessiné à l'écran.
- La définition et la maintenance automatique de contraintes pour lier les objets entre eux. À chaque attribut d'un objet, il est possible d'associer une contrainte ou *formule* (*cf.* tableau 5.4) qui calcule sa valeur. Cette caractéristique fait de GARNET un système orienté-objet où les méthodes ont un rôle secondaire, et dans lequel l'interface avec les objets se fait à travers les valeurs des attributs qui sont accédées ou calculées par les contraintes. Les contraintes sont utilisées pour propager tout changement de valeur. Les contraintes se décrivent à partir d'une expression Lisp. Dans une formule, un attribut d'un autre objet est accessible à partir

d'une liste de pointeurs ou *chemin*. La valeur de l'attribut peut changer, Garnet assure la propagation du changement et reconstruit l'objet.

- Un ensemble d'*interacteurs* qui évitent la programmation de réaction sur un événement.

De plus, Garnet inclut plusieurs outils qui permettent à l'utilisateur de définir et d'examiner des objets graphiques (Lapidary) et des contraintes (C32). Afin d'améliorer les performances du

```
(create-instance 'rect-proto rectangle
   (:left 0) (:top 0) (:width 10) (:height 10)
   (:right (formula (+ (gvl :left)
                 (gvl :width)))))

(create-instance 'rect1 rect-proto
   (:left 50) (:top 10))

(create-instance 'rect2 rect-proto
   (:main rect1)
   (:left (formula (gvl :main :left)))
   (:top 30))
```

```
(create-instance 'rect-type rectangle
   (:left 0) (:top 0) (:width 10) (:height 10) (:right 10)
   (:lrw-cn (m-constraint :max (left right width)
            (setf left (- right width))
            (setf right (+ left width))
            (setf width (- right left))))
   (:width-stay (m-stay-constraint :medium width)))
```

Tab. 5.4 - : À gauche, définition d'un rectangle type puis de deux rectangles issus de ce rectangle type. *formula* et *gvl* sont les mots-clefs pour la définition d'une formule et l'accès à une valeur d'attribut. À droite, une définition plus complète d'un rectangle contenant deux contraintes de force *max* et *medium*.

modèle, un certain nombre de techniques ont été employées, à savoir le *caching* pour les valeurs d'attributs qui évite une recherche dans la hiérarchie d'héritage lors de l'accès au même attribut, le recours à des agrégats virtuels pour éviter la surcharge de la mémoire, ou encore l'élimination de contraintes qui ne sont effectives qu'à la création de l'objet.

Lorsque la valeur d'un attribut est modifiée, toutes les valeurs des attributs calculées par une formule contenant l'attribut modifié sont invalidées. Le résolveur de contraintes de Garnet effectue cette propagation. Les formules des attributs dont les valeurs ont été invalidées ne sont ré-évaluées que paresseusement, au moment voulu[2]. En cas de cycles dans la propagation (on tente de lire la valeur d'un attribut résultat d'une formule alors même qu'on exécute cette formule), le résolveur rétablit pour l'attribut à calculer son ancienne valeur, ce qui peut se traduire par la satisfaction ou non des contraintes du cycle. La valeur d'un attribut contraint peut quand même être fixée par l'utilisateur (la contrainte n'est plus forcément satisfaite, mais le système accepte la modification). Le *caching* des valeurs écourte la propagation lorsque la nouvelle valeur est identique à l'ancienne.

On peut reprocher à Garnet de n'accepter qu'une contrainte unidirectionnelle – un attribut est lié à d'autres attributs, la réciproque n'est pas automatique. Les contraintes de Garnet ressemblent donc plus à des méthodes de calcul. Comme les autres outils précédents à base d'interfaces graphiques, il n'est pas orienté vers la résolution mais plutôt vers le maintien de solution, ce qui est justifiable dans ce contexte. Il présente une forme intéressante de chemin où des pointeurs permettent de recalculer une formule dès que l'un des attributs sur le chemin est modifié.

Signalons que pour remédier en partie à ses défauts, Michael Sanella a proposé en Multi-Garnet [Sanella94] une version de Garnet admettant des contraintes hiérarchisées et multi-directionnelles à méthodes mono ou multi-sorties. Le résolveur, basé sur l'algorithme *SkyBlue*, permet l'ajout ou le retrait de contraintes sous la forme d'un ajout ou d'un retrait d'attribut.

5.1.2.3 Pecos et Ilog-Solver

Toujours dans la famille des langages à objets avec contraintes, Pecos [Puget92b, Puget92a, Ilog92b] et Ilog-Solver [Puget94] sont deux bibliothèques de programmation par contraintes commercialisées par la société ILOG. Pecos est destiné à être utilisé avec le langage Le-Lisp [Chailloux et al.86] alors que Solver en est une ré-écriture pour le langage C++, présentant quelques différences avec Pecos. Nous décrivons ici, les principes de ces deux bibliothèques et, plus

[2] Il existe aussi un algorithme de ré-évaluation immédiate des méthodes en aval de l'attribut modifié.

particulièrement, l'intégration de contraintes dans les objets du langage hôte.

À l'origine de la création de PECOS, les motivations de son auteur, Jean-François Puget, étaient de combiner programmation par contraintes et programmation à base d'objets afin d'offrir un langage de haut niveau de programmation par contraintes. Inspiré de la programmation logique par contraintes, et plus particulièrement des techniques implantées dans le langage CHIP, PECOS permet de définir, de manière intentionnelle, des CSP, d'appliquer sur eux et de manière incrémentale un algorithme d'arc-consistance (*cf.* section 4.5.7), et de fournir un algorithme indéterministe de recherche de solution basé sur un *Backtrack* qui peut être contrôlé par la pose de points de choix. L'ordre d'énumération des variables et des valeurs peut être établi par des critères pré-existants ou définis par l'utilisateur.

En PECOS, un CSP est construit à partir de la définition de *variables contraintes* auxquelles est associé un domaine. Ces variables contraintes peuvent être de type quelconque, ce qui est une des particularités de PECOS. En particulier, le domaine d'une variable contrainte peut être un ensemble d'objets (au sens d'instances) utilisés dans la représentation du problème puisque PECOS étend un langage orienté-objet.

Cependant, pour un certain nombre de types particuliers, PECOS propose une définition de variable adaptée et un ensemble de contraintes prédéfinies s'y rapportant :

- les *variables contraintes entières* ont un domaine associé exprimable sous la forme d'un intervalle ou d'un ensemble énuméré. Un ensemble de contraintes représentant les opérateurs arithmétiques classiques ($+, -, *, / \dots$) est fourni, elles permettent de construire des expressions combinant variables contraintes et constantes entières.
- les *variables contraintes flottantes* ont un domaine défini sous la forme d'un intervalle délimité par deux flottants. Une *précision* permet de fixer l'amplitude minimum de l'intervalle. Lorsqu'elle est atteinte, la variable prend alors la valeur médiane de l'intervalle obtenu. Un ensemble de contraintes représentant les opérateurs arithmétiques classiques ($+, -, *, /, log, exp \dots$) est fourni qui permet de construire des expressions combinant variables contraintes et constantes flottantes.
- les *variables contraintes ensemblistes* ont toutes pour valeur un ensemble, leur domaine est donc un ensemble d'ensembles. Lorsqu'il n'est pas réduit à un singleton, ce domaine est représenté par sa borne inférieure (intersection de tous les éléments-ensembles du domaine) et par sa borne supérieure (union de tous les éléments-ensembles du domaine). La borne inférieure fournit les éléments *obligatoires* que l'on retrouvera dans une solution, si elle existe, la borne supérieure fournit les éléments *possibles*. Un ensemble d'opérateurs ensemblistes permet de manipuler ces variables. De même, la cardinalité de l'ensemble représenté par une telle variable peut être contrainte.

Il est non seulement possible de connaître le domaine (éventuellement vide) d'une variable contrainte, mais des fonctions de manipulation des variables de ces types sont fournies.

Les contraintes qui représentent des opérateurs (arithmétiques entiers, par exemple, comme `ct-add`) sont des fonctions dont le résultat est une variable contrainte (gérée par le système), qui peuvent donc être, à leur tour, employées dans une autre contrainte (ainsi, `(ct-add (ct-add x y) z)` est encore une variable contrainte). Pour les entiers, comme pour les flottants, le calcul de l'intervalle de la variable contrainte résultat se base sur les règles de l'arithmétique des intervalles. Sur l'exemple précédent, si le domaine de `x` est $[1, 3]$, celui de `y` est $[0, 6]$, le domaine de `(ct-add x y)` est $[1, 9]$.

Ces contraintes, en tant qu'opérateurs, sont appelées à être utilisées dans des contraintes qui représentent des comparateurs (égalité, inégalité, différence...) et qui sont fournies également selon le type des expressions contraintes (par exemple, pour les entiers, `(ct-eq x (ct-add y z))` ou `(ct-ge (ct-add x y) z))`. Contrairement aux opérateurs, ces comparateurs n'ont pas pour résultat une variable contrainte. Ils constituent des contraintes entre deux intervalles et agissent sur l'un ou/et l'autre de ces intervalles d'après la sémantique de l'opérateur. Par exemple, `(ct-eq x` `y)` procède aux modifications suivantes $dom(x) = dom(x) \cap dom(y)$ et $dom(y) = dom(x) \cap dom(y)$ alors que `(ct-ge x y)` procède aux modifications $dom(x) = [max(borne - inf(dom(x)), borne - inf(dom(y)), borne - sup(dom(x))] \cap dom(x)$ et $dom(y) = [borne - inf(dom(y)), min(borne - sup(dom(x)), borne - sup(dom(y))] \cap dom(y)$ où $borne - inf$ (respectivement $borne - sup$) donne la plus petite (respectivement grande) valeur d'un domaine.

Un des atouts majeurs de PECOS est la définition de nouvelles contraintes (*cf.* tableau 5.5) qui permet à l'utilisateur d'étendre l'ensemble des contraintes prédéfinies. Les contraintes sont des objets du langage hôte. Il est possible de définir une nouvelle classe de contrainte en indiquant son nom, la liste de ses paramètres (on donnera le type des variables contraintes sur lesquelles porte la contrainte) et une liste de *schémas* de contrainte. Les schémas de contraintes définissent les règles de la propagation. Celle-ci est basée sur un mécanisme de démons qui, dès qu'un événement de propagation survient (la modification du domaine d'une variable contrainte), déclenche le traitement approprié qui vise à rétablir l'arc-consistance du réseau. L'association entre programmation par

`(defctconstraint myeq ((x ct-var) (y ct-var))`
`   ((when (x = a) assert (y = a))`
`   (when (y = a) assert (x = a))`
`   ))`
	`(defctclass tache`
	`      nom`
	`      (duree () reversible)`
	`      (debut (ct-fix-tange-var 0 1000) constrained))`

TAB. 5.5 - : À gauche, définition d'une nouvelle contrainte d'égalité qui n'est activée que lorsqu'une des deux variables a une valeur. À droite, définition d'une classe d'objets contrainte.

contraintes et programmation par objets dans PECOS se traduit non seulement par le fait que les valeurs des variables contraintes peuvent être des objets, mais aussi et surtout par le fait que les objets peuvent être contraints. L'idée sous-jacente est de pouvoir contraindre directement au niveau des objets – plus précisément, des *champs* ou attributs de ces objets – en conservant donc la représentation intuitive et structurée du problème qu'offrent les objets. L'arsenal des contraintes qu'il est possible d'énoncer sur les objets est le même que celui disponible sur les variables contraintes. De fait, les algorithmes de résolution s'appliquent également sur les CSP dont les variables sont des champs d'objets.

Les contraintes définissables en PECOS sont des contraintes *génériques* ou contraintes de classes qui portent sur tous les objets d'une classe. En contrepartie, c'est au moment de la définition de la classe que doivent être définis les champs *réversibles* dont la valeur pourra être rétablie lors d'un retour en arrière et les champs *contraintes* auxquels on associe une variable contrainte. Les contraintes de classe facilitent l'écriture des programmes. Grâce au mécanisme d'héritage, elles sont héritées par les sous-classes et ne sont allouées qu'une seule fois.

Enfin, PECOS propose un ensemble de contraintes symboliques appelées *descriptions* qui permettent de décrire à un haut niveau d'abstraction un objet (ou plusieurs) d'une classe à travers la valeur d'un ou de plusieurs de ses (ou leurs) attributs. Parmi les objets de la classe désignée, les primitives de recherche de PECOS permettent d'énumérer les solutions lorsque plusieurs objets sont candidats à une description unique comme, par exemple, `(ct-the-object 'house 'color 'red l-house)`.

Le processus de base de la résolution de contraintes des CSP à domaines finis consiste en une énumération du domaine des variables basée sur un retour arrière classique. Des critères standards de choix des variables et des valeurs sont disponibles, mais d'autres peuvent être décrits par l'utilisateur. Une primitive d'optimisation permet de déterminer la meilleure solution vis-à-vis d'un critère donné.

En PECOS, il est possible de programmer de façon non déterministe, c'est-à-dire de dicter au résolveur un ensemble d'alternatives (construites à base de *ou* et de *et* logiques, de branchements conditionnels) à essayer. Pour ce faire, PECOS permet à l'utilisateur de guider le retour-arrière en fixant lui-même les points de choix vers lesquels l'algorithme retournera en cas d'échec.

Enfin, signalons, qu'il est possible d'inhiber une contrainte, c'est-à-dire de la rendre inactive pour la propagation : les effets de sa présence jusqu'à son inhibition ne sont pas annulés.

Solver correspond à l'implémentation d'une librairie similaire à Pecos dans le langage C++. Les caractéristiques de la bibliothèque Pecos sont aussi celles de Solver. On trouve de plus dans Solver, la classe des variables contraintes de type booléen (de domaine $\{0, 1\}$) qui sont traitées par la propagation. Sur les flottants ont été adaptés les algorithmes de la propagation d'intervalles présentés par Lhomme. En revanche, la définition de contraintes a disparu mais il est possible de définir de nouvelles contraintes en composant des contraintes prédéfinies à l'aide des opérateurs logiques *or, and, xor, not*. Les contraintes sur les objets obéissent aux mêmes principes.

Les succès industriels de Pecos et de Solver, notamment dans les problèmes d'allocations de ressources, d'optimisation de découpe, de planification, attestent de leur puissance de modélisation et de résolution. Ces bibliothèques très complètes se montrent en effet largement paramétrables et extensibles (choix dans la résolution, ajouts de contraintes, spécifications d'algorithmes particuliers de recherche...).

Concernant l'intégration de contraintes dans les objets, le reproche classique qui leur est fait est que les champs des classes doivent être définis comme contraignables dès la définition de la classe. On ne peut donc pas contraindre une classe à n'importe quel moment. De plus, il n'est pas possible de contraindre un objet dans une classe en particulier, à moins d'en faire l'objet unique d'une sous-classe. Parce que les contraintes de classes sont figées dans la définition de la classe, il n'est pas possible de les supprimer – seule une inhibition est possible – sans reconstruire tout le CSP. Dans Pecos, comme dans Solver, les domaines des variables ne peuvent qu'être réduits et ne sont jamais étendus (sauf dans le cas particulier de l'énumération).

5.1.3 Les langages de contraintes décrits par des objets

5.1.4 Prose

Prose [Berlandier92b, Berlandier92a] est une "boîte à outils" pour l'interprétation des contraintes qui propose un ensemble de fonctions pour la satisfaction d'un CSP à domaines finis et discrets et pour le maintien de solutions. Son auteur, Pierre Berlandier, a souhaité tout d'abord définir un schéma abstrait de représentation et de manipulation de contraintes. Aussi, Prose laisse une entière liberté au programmeur sur le choix de la représentation de son CSP. Prose communique simplement avec cette représentation au moyen d'une interface fonctionnelle composée, dans sa version 2.0 écrite en Le-Lisp v16, d'un ensemble de méthodes et de fonctions génériques qui permettent de manipuler les composants d'un CSP : variables, domaines et contraintes. Dans cette version, Prose comporte un générateur de problèmes-tests qui permettent à l'utilisateur de juger des performances des heuristiques ou méthodes de résolution qu'il propose.

Prose rend la satisfaction de contraintes en domaines finis très paramétrable. Si, par défaut, l'énumération des domaines des variables se fait à l'aide du *Forward-Checking*, le programmeur peut définir sa propre stratégie. Il est également possible d'appliquer un filtrage au préalable sur le CSP par arc-consistance totale ou orientée. Ce même filtrage peut être accéléré pour les contraintes actives (où la valeur d'une variable est établie en fonction des $n-1$ autres). Surtout, Prose permet un ordonnancement statique et/ou un ordonnancement dynamique des variables et des contraintes. Enfin, pour permettre la recherche d'une solution optimale d'un CSP, l'utilisateur a la possibilité d'exprimer un critère d'optimisation.

Le maintien de solution consiste, lors de la modification de la valeur d'un ensemble de variables et/ou de l'ajout d'un ensemble de contraintes, à déterminer une solution proche de la précédente. Plutôt qu'une approche locale qui consiste à répercuter récursivement la modification de valeur d'une variable auprès des variables attenantes, Prose propose une approche globale qui repose sur un parcours du graphe des contraintes afin de découvrir les chemins qui ne recalculent qu'une seule fois chaque variable. Cette phase de planification des graphes de propagation est alors suivie par une phase d'évaluation. L'avantage de cet algorithme est qu'il prend en compte des modifications simultanées (changements de valeurs ou ajouts de contraintes). L'idée est de rétablir la cohérence en ne modifiant au plus qu'une variable pour chaque contrainte sans former de cycle. Pour les réseaux contenant des cycles, l'algorithme ne fournit pas tous les graphes de propagation possibles. Gilles Trombettoni [Trombettoni92] a proposé une modification de cet algorithme en ce sens.

Dans sa thèse, Pierre Berlandier a proposé d'intégrer PROSE dans un modèle général d'Environnement de Représentation et d'Interprétation des Connaissances (ERIC) à base d'objets. La solution adoptée est de réaliser une intégration souple et uniforme dans laquelle les contraintes sont représentées par des objets standards, sans modification du modèle objet. Contraintes et objets se côtoient donc dans une base de connaissance de l'ERIC. C'est le générateur de systèmes experts SMECI qui a été choisi le premier pour l'implantation de PROSE, puis la couche objets TELOS de LE-LISP v16 [Ilog92a]. Dans les deux cas, les composants d'un CSP sont représentés par des objets du langage objet hôte à l'aide de trois classes[3] : *Variable, Constraint* et *Relation* (*cf.* figure 5.2).

- Les objets de la classe *Variable* pointent sur les variables du CSP (des attributs d'autres objets de la base). Chacun de ces objets stocke la liste des contraintes portant sur la variable qu'il représente.
- Les objets de la classe *Constraint* représente les contraintes effectives déclarées sur les objets. Un objet contrainte sert de lien entre chaque objet variable qui représente une variable du CSP qu'elle contraint et la relation correspondant à la contrainte.
- Les objets de la classe *Relation* sont liés aux contraintes qu'ils représentent. Pour chaque contrainte, un objet relation décrit la liste des types des paramètres, le prédicat de test de satisfaction de la contrainte et la liste des méthodes qui permettent d'effectuer le calcul des valeurs.

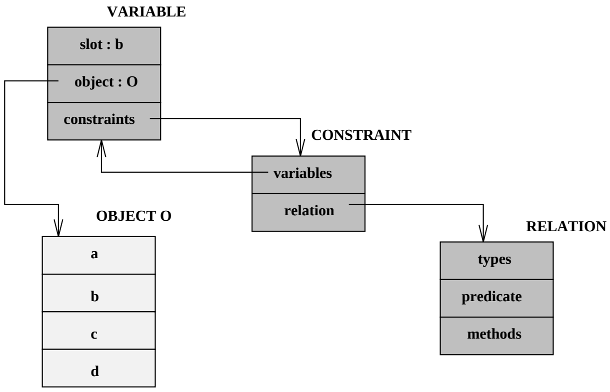


FIG. 5.2 - : L'architecture proposée pour contraindre les objets à l'aide de PROSE

Grâce à cette représentation, les objets du domaine (c'est-à-dire les objets qui ne sont pas des objets du système de contraintes) ne sont pas modifiés par la présence de contraintes. Indépendants, ils sont simplement accessibles par les objets du système de contraintes. N'importe quel objet du domaine peut être contraint à n'importe quel moment. Les contraintes, simples ou composées, peuvent être ajoutées ou supprimées à tout moment. L'arc-consistance peut être appliquée sur des domaines rétrécis par l'ajout de contraintes mais aussi sur des domaines relaxés, grâce à l'algorithme AC|DC. La satisfaction de contraintes peut être invoquée lorsque le CSP a été défini et que l'utilisateur souhaite en connaître une ou plusieurs solutions. Le maintien de solution est souhaitable dans des environnements de problèmes évolutifs, en conception notamment, lorsque des contraintes supplémentaires sont ajoutées au problème.

L'originalité du travail de Berlandier tient aussi à la proposition de deux types de contraintes particuliers :

- Les contraintes de *transition* dont le but est de lier la valeur d'une variable à la valeur qu'elle

[3] Six en réalité, les trois autres étant dédiées à la représentation des constantes, de contraintes et de relations complexes (composées d'autres contraintes ou relations).

avait dans un état précédent ou bien aux valeurs d'autres variables dans différents états. Profitant du fait que SMECI tient à jour un arbre d'états qui décrit les diverses modifications apportées à une base de connaissances, il est possible de retracer l'historique des changements de valeur d'une variable. Il est possible alors d'associer une horloge à chaque variable contrainte et de faire référence à un état (valeur) passé de la variable. De même, l'évolution de la base peut se faire relativement à plusieurs horloges, chacune étant avancée lors d'un événement particulier de transition du système d'un état à un autre. Les horloges sont représentées par des objets. La classe *Variable* est étendue afin de pouvoir enregistrer le temps choisi et le retour relatif à effectuer (nombre de coups d'horloge). En maintien de cohérence, les contraintes de transition doivent être déclenchées à chaque nouveau coup d'horloge. Aussi, l'attribut de la classe *Horloge* qui représente le temps écoulé est-il impliqué dans une contrainte, de sorte que, modifié, il déclenche le processus de propagation. Cette approche temporelle des relations trouve sa source d'inspiration dans le langage Lucid [Wadge et al.85], lui-même à l'origine des idées de KALEIDOSCOPE.

- Les *méta-contraintes* dont on distingue trois formes :
  1. en tant que contrainte impliquant des objets contraintes, une méta-contrainte peut, par exemple, imposer que ces contraintes portent sur les mêmes attributs, soient actives ou non en même temps, aient une relation différente...Mais l'auteur lui-même émet quelques réserves quant à l'intérêt réel de telles méta-contraintes.
  2. en tant que contrainte impliquant un ensemble d'attributs, une méta-contrainte permet de répondre aux exigences de cohérence d'une base de connaissances dynamiques. Par

Categorie Meuble	*Categorie Bureau*
hauteur : entier parmi une liste d'entiers	*surface : entier*
prix : reel dans un intervalle de reels	*mobilier : liste d'objets de categorie Meuble*
style : symbole parmi un ensemble de symboles	*cout : reel*
emplacement : objet de categorie Bureau	

TAB. 5.6 - : Deux catégories de SMECI.

exemple, en considérant les deux catégories SMECI du tableau 5.6, on souhaite établir une contrainte qui assure que le coût du mobilier d'un bureau b est égal à la somme des prix des meubles de ce bureau (C : *b.coût*$= \sum_{m_i \in b.mobilier} m_i.prix$). Si le mobilier change alors le coût du bureau doit être ré-évalué par la formule, mais aussi, si l'un des prix des meubles du mobilier du bureau change, le coût du bureau change aussi et doit être ré-évalué. La prise en compte de ces deux modifications conduit à transformer la contrainte C en deux contraintes C_1 et C_2. C_1 est une contrainte qui impose que *b.coût* $= \sum_{m \in b.mobilier} m.prix$ pour le mobilier du bureau (elle diffère de C en ce sens qu'elle n'agit pas sur la liste mais la parcourt simplement) et C_2 est une méta-contrainte qui gère la liste des attributs de la contrainte C_1 (autrement dit, le coût du bureau et la liste des meubles) et qui impose que $C_1.attributes$ = *b.coût* $\cup \bigcup_{m \in b.mobilier} m.prix$. C_1 assure que le coût du bureau est la somme des prix des meubles qui forment le mobilier du bureau alors que C_2 assure que la contrainte C_1 porte bien sur le mobilier actuel du bureau.
  3. en tant que contrainte impliquant l'activité d'autres contraintes, une méta-contrainte peut subordonner l'activité d'une contrainte (sa présence dans le réseau) à la satisfaction d'une condition. L'illustration de cette contrainte *conditionnelle* est fournie par l'exemple suivant :
On souhaite lier le budget du laboratoire l où se trouve le bureau b, de sorte que si le budget est faible, le coût de l'ameublement du bureau b n'excède pas une certaine limite. Autrement dit, C : si $l.budget = faible$ alors *b.coût* $<$ *limite*.
La solution avec méta-contrainte consiste à considérer la contrainte C_1 : *b.coût* $<$ *limite*

et de l'associer à la méta-contrainte $C_2 : C_1.activity = (l.budget = faible)$. Berlandier signale que l'on peut trouver cette contrainte conditionnelle dans la plupart des langages de contraintes sous la forme d'une contrainte unique $C_3(l.budget,\ b.coût,\ limite)$ dont l'extension est $\{(faible, v, w) \mid v < w\} \cup \{(u, v, w) \mid u \neq faible\}$. Cependant la méta-contrainte apporte plus de flexibilité en permettant l'ajout ou le retrait d'une condition à tout moment, et surtout, n'impose le recalcul de la contrainte que lorsque les variables impliquées par la condition sont modifiées.

Parce que les méta-contraintes peuvent modifier le nombre d'arguments d'une contrainte, la structure du graphe de contraintes n'est pas fixe. Aussi, l'algorithme de propagation implanté dans Prose n'était pas directement applicable. Il a donc été modifié en trois points : 1) évaluation par nécessité des contraintes le long du graphe de propagation pour déterminer la liste de leurs variables (attributes) et leur activité, 2) retour en arrière lorsque l'on cherche à remettre en cause un nœud déjà évalué (cycle) 3) remise en cause de la contrainte lors de la modification d'une variable de méta-contrainte. Cependant, cet algorithme ne garantit pas la terminaison de la propagation, notamment en cas de méta-contrainte absurde du type si $x + y \neq 0$ alors $x + y = 0$. Une solution pour éviter ces problèmes est d'interdire que la méta-contrainte partage des variables avec les contraintes sur lesquelles elle porte. Une autre solution est de partitionner l'ensemble des contraintes selon leur niveau dans la hiérarchie méta et de contrôler à chaque modification que les attributs ajoutés ont un niveau compatible (le même ou indéfini) avec celui de la contrainte. Si les possibilités d'expression des méta-contraintes sont limitées par ce contrôle, le maintien de la cohérence s'effectue simplement, en propageant des méta-niveaux les plus hauts aux plus bas.

Anticipant la critique selon laquelle Prose ne permet pas l'expression de contraintes de classes, Berlandier avance que c'est au langage objet hôte de proposer et de mettre en place à l'aide de Prose une telle fonctionnalité. Concernant l'absence de dynamisme dans les réseaux de contraintes, l'algorithme AC|DC [Neveu et al.94] semble aujourd'hui combler les lacunes originelles. Il reste que Prose n'est pas en mesure de traiter des CSP à domaines continus et infinis. Concernant l'intégration souple proposée de Prose dans un langage à objets, on peut regretter, comme Tibor Kökény [Kökény94], qu'elle ne tire pas d'avantage parti des caractéristiques du paradigme objet : hiérarchie, héritage. C'est pourquoi, si cette intégration réussit le pari de l'uniformisation en ce qui concerne la définition, ce ne semble pas être le cas (outre pour l'héritage et la hiérarchisation) pour la suppression des objets-contraintes. Les méta-contraintes sont une initiative très intéressante mais ne sont pas triviales à construire. Enfin, les conséquences de la modification (légitimée par l'uniformisation visée) des objets du langages de contraintes (les objets-variables, contraintes et relation) n'est pas non plus abordée par l'auteur de Prose.

5.1.5 CSPOO

CSPOO [Kökény94] est un système voué à la représentation et à la résolution de contraintes dans un environnement orienté-objet conçu par Tibor Kökény.

L'objectif de ce travail était de trouver une représentation, aussi bien pour les composants d'un CSP (variables, domaines, contraintes) que pour les algorithmes de résolution qui soit à la fois modulaire, extensible, indépendante du domaine d'application visé et garante de l'efficacité algorithmique. Le choix d'un environnement objet semble donc naturel pour les deux premiers critères, aussi deux classes *c-Variables* et *c-Contraintes* sont considérées qui suffisent à modéliser les composants d'un CSP. La prise en compte des deux derniers critères est rendue possible par une classe *c-Solveurs* destinée à représenter les spécificités des techniques de résolution employées pour un CSP particulier. Plus précisément (*cf.* figure 5.3) :

- les attributs de la classe *c-Variables* décrivent le domaine de la variable, les variables adjacentes, les contraintes qui portent sur la variable, la valeur courante. Parmi les méthodes qui sont attachées à cette classe, hormis une méthode d'instanciation, on dispose les méthodes qui permettent de trouver une affectation de valeur consistante pour la variable à partir d'une affectation partielle des variables du CSP.

- les attributs de la classe *c-Contraintes* décrivent les variables de la contrainte et les contraintes adjacentes. Les méthodes de cette classe, hormis la méthode d'instanciation, permettent le filtrage et la résolution globale selon la spécificité de la contrainte.
- les attributs de la classe *c-Solveurs* décrivent l'ensemble des variables du CSP traité, l'ensemble des variables déjà affectées, l'ensemble des contraintes, la méthode (*Backtrack, Backjumping*) et les heuristiques (ordonnancement statique ou dynamique) de résolution choisies. Cette classe fournit donc, pour un CSP donné, l'algorithme de pré-filtrage utilisé, le mode d'ordonnancement des variables à appliquer ainsi que l'algorithme d'énumération.

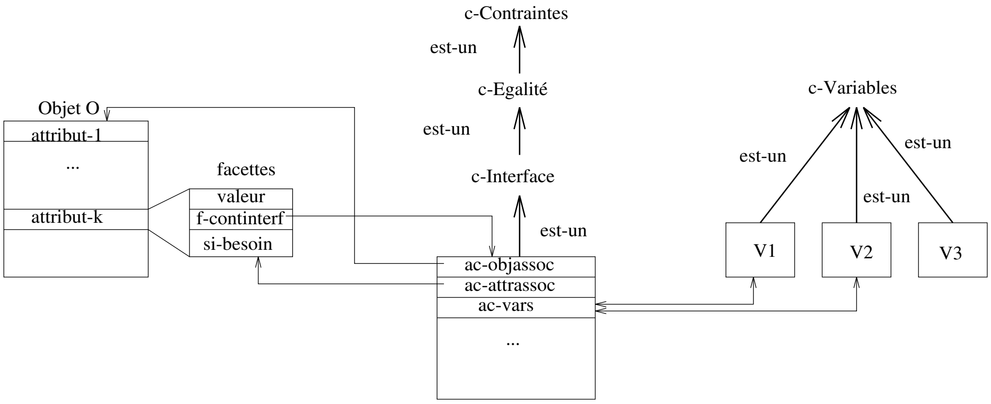


FIG. 5.3 - : L'architecture proposée pour contraindre les objets à l'aide de CSPOO

Si Kökény n'établit pas de hiérarchie sur les solveurs, il présente une hiérarchie de classes de contraintes distinguant les contraintes selon leur arité et leur expression (en intention ou en extension). Cette hiérarchie est utile car elle permet d'introduire pour chacune de ces classes une méthode de cohérence, de filtrage ou de propagation qui lui est propre.

Concernant les algorithmes de résolution et de filtrage (consistance), la représentation choisie et les méthodes attachées aux variables permettent de diriger la résolution (ordonnancement statique et dynamique de variables, points de choix dans le *Backtrack*, filtrage partiel ou sélectif des domaines selon la contrainte, etc.) et permettent de construire des algorithmes plus modulables que les algorithmes classiques.

Poursuivant ses investigations dans un univers orienté-objet, Kökény a proposé un algorithme d'arc-consistance, appelé HAC-6, qui est une adaptation de l'algorithme AC-6 au cas des CSP binaires à domaines hiérarchiques.

La structure hiérarchique des domaines est sous-jacente au monde des objets. Aussi, la pose d'une contrainte dont l'extension est le seul tuple {*(Evoit, Foncé)*} (où *Evoit* désigne la classe des voitures européennes et *Foncé* caractérise la couleur), représente, au regard des hiérarchies $Voitures$ et $Couleurs$, une simplification de l'extension réelle qui est (*cf.* figure 5.4) :

{*(Evoit, Noir), (Evoit, Brun), (Peugeot, Foncé), (BMW, Foncé),(Peugeot, Noir), (BMW, Noir), (Peugeot, Brun), (BMW, Brun),(Evoit, Foncé)* }

Dans ce contexte, l'hypothèse selon laquelle : *si un objet, valeur d'une variable contrainte, satisfait une contrainte c, alors toutes les spécialisations de cet objet satisfont aussi la contrainte c et inversement, s'il ne satisfait pas c, alors aucune de ses généralisations ne la satisfait*, est essentielle. Kökény montre qu'elle permet de simplifier la définition des contraintes, d'améliorer les algorithmes de filtrage (comme AC-6) mais aussi de résolution et de compacter l'écriture des solutions.

HAC-6 se base sur la recherche des *éléments maximaux* des domaines, ceux dont on peut dire que, s'ils ont un support, alors celui-ci est support de toutes leurs spécialisations. Il est alors inutile

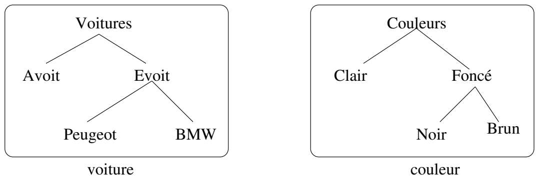


FIG. 5.4 - : Les domaines hiérarchiques de deux variables *voiture* et *couleur*

de chercher un support pour un élément inférieur (au sens de la hiérarchie établie) de ce domaine.

En raison de la structure hiérarchique, la vérification de la compatibilité de deux valeurs est dépendante de la situation de ces valeurs par rapport à la forme abrégée de la contrainte. Par exemple, si on cherche à satisfaire (*Evoit*, *Foncé*) pour les valeurs *Peugeot* et *Brun*, un parcours ascendant des hiérarchies est nécessaire qui peut être coûteux. Kökény préconise de calculer la fermeture par *h-compatibilité* des contraintes (c'est-à-dire accroître l'extension par les tuples solutions des éléments inférieurs).

Sur ce même principe d'élément maximal, la résolution d'un CSP à domaines hiérarchiques peut se réduire à la recherche de solutions maximales en choisissant les valeurs des domaines de l'élément le plus général au plus spécialisé. L'ordre établi sur les domaines des CSP à domaines hiérarchiques a été rapproché du travail réalisé par la même équipe "Contraintes" du LIRMM de Montpellier, concernant la notion de susbtituabilité.

Ayant proposé en CSPOO un modèle de représentation des CSP et des algorithmes de résolution dans un environnement orienté-objet, Kökény s'est également intéressé au moyen et à la signification de "contraindre des objets".

Afin de rassembler plusieurs contraintes dans un même objet, Kökény propose des *Contraintes Génériques Composées* (CGC) qui sont des *et* logiques d'autres CGC ou d'autres *Contraintes Génériques Simples*. Ces contraintes portent sur les attributs des objets. Mais, dès lors que la valeur d'un attribut d'objet peut être un objet, il est souhaitable que l'on puisse *indirectement* contraindre un attribut d'un autre objet à partir d'un objet. À ces fins, Kökény reprend la notion de *chemin d'attributs* (introduite par ThingLab). Un chemin d'attributs est une liste finie *(O attribut*$_1$ *attribut*$_2$*...)* où *O* est une classe ou une instance. Afin que tout chemin d'attributs puisse supporter l'absence mais aussi la modification de la valeur d'un des attributs, la notion de *variable flottante* est introduite. L'idée est d'associer à chaque attribut d'un chemin impliqué dans une contrainte un ensemble de variables qui est un sous-arbre de l'arbre d'attributs de l'objet de départ du chemin. Cet ensemble sera complété lors de la pose de nouvelles contraintes sur le même objet (*cf.* figures 5.5 5.6). Lors de la résolution – qui s'effectue dans un ordre compatible avec la hiérarchie de variables créée –, la même valeur peut être attribuée à différentes variables. Afin de garantir que ces variables aient la même valeur, une unification dynamique est réalisée à l'aide de contraintes d'égalité, appelées *contraintes d'interface* (*cf.* figure 5.7). Ces contraintes d'interface forment le lien entre un attribut contraint dans un objet et les variables (instances de c-Variables). Une implémentation de CSPOO a été réalisée dans l'environnement objets Y3 [Ducournau90] qui bénéficie de l'interface graphique YFEN qui facilite la visualisation lors de la définition et de la résolution des CSP.

CSPOO est un système innovant qui propose une représentation orientée-objet des CSP aussi bien que des algorithmes de résolution de ces CSP. La hiérarchie de contraintes proposée et la répartition des méthodes de filtrage en font un outil de résolution très modulable. Son auteur apporte une solution à la gestion de la dynamique pour les chemins d'attributs. Tout attribut du système objet hôte est contraignable à n'importe quel moment. Pour les domaines hiérarchiques, un algorithme d'arc-consistance adapté ainsi qu'une méthode de recherche de solutions maximales sont proposées qui tiennent compte de la spécialisation établie entre les éléments (objets).

On peut cependant critiquer le fait que tout attribut contraint soit représenté par un objet dans la base, ce qui peut considérablement encombrer celle-ci si elle contient des CSP importants

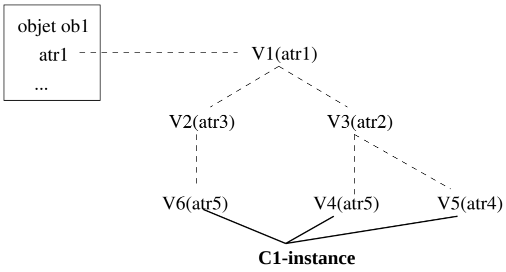


FIG. 5.5 - : Une contrainte C1 a trois paramètres qui sont les chemins *(ob1 atr1 atr2 atr4), (ob1 atr1 atr3 atr5)* et *(ob1 atr1 atr2 atr5)*. La structure ci-dessus est créée. Elle rend compte de l'arbre d'attributs à partir de $ob1$. Des variables sont attachées à chaque attribut impliqué dans un chemin : les variables flottantes. La contrainte porte sur les feuilles.

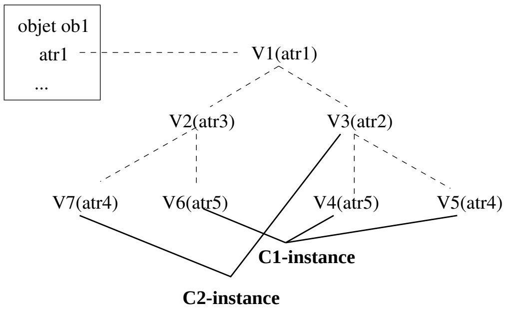


FIG. 5.6 - : On ajoute la nouvelle contrainte C2 qui porte sur les chemins *(ob1 atr1 atr3 atr4)* et *(ob1 atr1 atr2)*. Une seule variable est ajoutée à la structure, les autres sont unifiées car elles sont les préfixes de chemins déjà considérés.

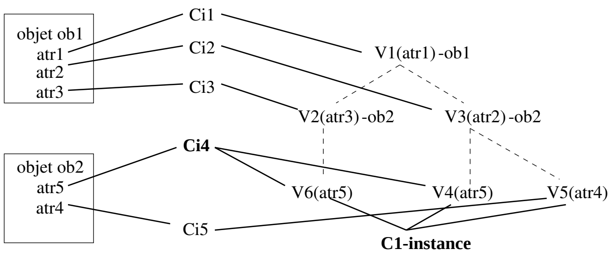


FIG. 5.7 - : L'attribut $atr1$ prend pour valeur $ob1$, les attributs $atr3$ et $atr2$ la valeur $ob2$. Alors les variables $V6(atr5)$ et $V4(atr5)$ ont même valeur. Pour contrôler l'égalité des variables pointant sur le même objet, des contraintes d'interface sont introduites auprès de chaque variable.

en taille. De plus, bien qu'une revue des diverses interprétations de la présence de contraintes entre (et sur) des objets soit proposée, elle ne constitue qu'une proposition non explorée. Ceci peut s'expliquer par le fait que CSPOO est avant tout un outil de programmation par contraintes destiné à un environnement orienté-objet, alors que ces choix seraient de l'ordre du couplage de CSPOO avec un système à objets particulier. Enfin, les conséquences sur les objets, sur la hiérarchie d'objets et sur la cohérence de la base d'ajouts et de retraits de contraintes ne sont pas évoquées.

5.1.6 COOL

COOL [Avesani et al.90] est un langage de contraintes qui a été implémenté sur le langage de frames KEE [Fikes et al.85]. Il est proche dans l'esprit des systèmes PROSE et CSPOO en ce sens que les contraintes d'un CSP sont représentées par des objets. Dans COOL, on trouve donc des objets KEE contraints et des objets KEE qui servent à la description des réseaux de contraintes établis.

En COOL, on considère trois types d'objets (*cf.* figure 5.8):

- les *attributs contraignables*: ce sont les attributs des objets KEE qui sont considérés comme les variables du CSP. Chaque attribut contraint possède deux facettes d'information. La facette *label* donne le domaine de l'attribut. La facette *constraints* donne l'ensemble des assertions de contraintes qui l'impliquent. Cette dernière facette est mise à jour automatiquement ou créée si elle n'existe pas.
- les *contraintes* sont des classes organisées en une hiérarchie qui servent à la description d'une contrainte. Au plus haut niveau de la hiérarchie sont définis trois attributs – *arité, test, test-class. arité* donne le nombre de variables de la contrainte. *test* est un prédicat qui vérifie si

les valeurs fournies pour les attributs impliqués par la contrainte sont satisfaisants. *test-class* vérifie le type des attributs impliqués. Plus bas dans la hiérarchie, la distinction est faite selon l'arité des contraintes. Dans ces classes où l'arité est fixée, les arguments (variables) de la contrainte sont décrits. La description consiste à donner le nom de l'attribut visé. C'est à ce niveau que l'on indique s'il s'agit d'une contrainte de classe (qui sera héritée par les sous-classes) ou d'une contrainte d'instance (les instances contraintes sont données). Cette hiérarchie de contraintes permet l'héritage des fonctions fournies par les différentes redéfinitions de l'attributs *test*.

- les *assertions de contraintes* rendent compte de la présence d'une contrainte et sont des objets, non pas des classes de contraintes présentées ci-dessus, mais des classes d'assertion de contraintes. Les classes d'assertion de contraintes sont organisées aussi hiérarchiquement. La classe racine comporte quatre attributs :
  - *constraint* donne le nom de la classe de contrainte référencée ;
  - *restriction* est un prédicat qui peut être associé à celui donné par *test-class* et qui vise à réduire encore l'ensemble des n-uplets qui satisfont la contrainte ;
  - *status* indique si la contrainte peut être considérée comme active ou non ;
  - *strength* a une valeur entre 0 et 1 qui donne la force de la contrainte. Il est ainsi possible d'établir une hiérarchie de préférence sur les contraintes.

  Plus bas dans la hiérarchie, la distinction est faite selon l'arité des contraintes. Les arguments sont définis comme dans les classes de contraintes équivalentes. Mais les ensembles de classes ou d'objets spécifiés indiquent quelles sont les classes (contraintes de classe) ou les objets (contraintes d'instances) dans lesquelles l'attribut est contraint. La facette *constraints* de cet attribut, dans ces classes ou dans ces objets, est alors mise à jour.

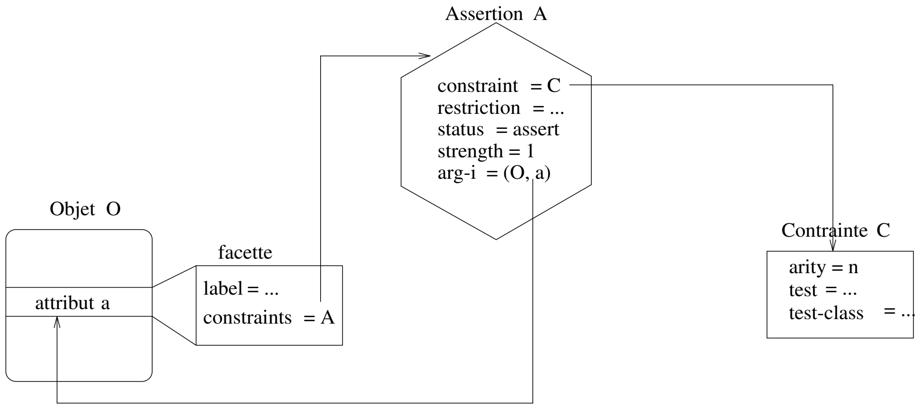


FIG. 5.8 - : L'architecture proposée pour contraindre les objets à l'aide de COOL

Grâce aux liens établis, il est possible, à tout moment, de connaître l'ensemble des assertions de contraintes portant sur un attribut ou l'ensemble des attributs contraints par une contrainte.

Une fois qu'un CSP a été défini par un ensemble d'assertions, COOL peut chercher une solution (une instanciation éventuellement non complète globalement consistante) au CSP pour un ensemble de paires (objet, attribut). Un algorithme d'arc-consistance est également intégré. Au moment de sa création, COOL exploitait les techniques et heuristiques de résolution les plus performantes.

COOL constitue sans doute la première immersion complète d'un langage de contraintes dans un environnement orienté-objet, puisqu'on y représente des contraintes par des objets. Les variables contraintes sont les attributs de ces objets et, par conséquent, un recours éventuel à des méta-contraintes est possible bien que non justifié. Comme pour PROSE et CSPOO, on peut reprocher à cette approche de mélanger les objets qui servent à la représentation du domaine d'application et ceux qui servent à la représentation des CSP établis entre les attributs.

Ensuite, une certaine redondance semble exister entre les informations détenues par les classes d'assertions de contraintes et celles de contraintes qui pourraient être aisément rassemblées dans une seule classe. De plus, la nécessité de décrire, dès la définition de la contrainte, les classes ou les objets des attributs qui sont susceptibles d'être la cible de cette contrainte, nuit à la flexibilité dans l'usage des contraintes. Ce typage des arguments de la contrainte a l'inconvénient de figer l'ensemble des attributs cibles. Ce procédé se prête mal à la définition dynamique de contraintes similaires à des contraintes existantes, mais destinées à de nouveaux objets. Concernant le retrait de contraintes, il est réduit à une désactivation que l'on peut rapprocher de celle de PECOS, et pour laquelle les valeurs écartées par la contrainte ne sont pas restituées.

5.1.7 Un langage hybride : SOCLE

SOCLE [Harris86] est un système hybride, créé par David Harris, qui couple les deux paradigmes contraintes et *frames*. La composante "objet" du système est assurée par le langage FRL [Roberts et al.77], alors que la composante "contrainte" repose sur le langage CONSTRAINTS. Pour son auteur, les avantages attendus de ce couplage sont:

- de la part du langage de représentation à base de frames, la structuration de la connaissance, les inférences par héritage, défaut ou attachement procédural ;
- de la part du langage de contraintes, l'expression et la maintenance de formules (multidirectionnelles) entre des attributs de frames génériques (classes) ou individuels (instances). Les contraintes de classes sont héritées.

Les contraintes sont donc utilisées dans SOCLE pour inférer des valeurs manquantes par propagation. De plus, des informations sur les dépendances entre valeurs, combinées avec un niveau de confiance *default, supposition, belief, constant* associé à ces valeurs, sont prises en compte pour tenter de résoudre une contradiction introduite par le changement de valeur d'un attribut.

La communication entre les deux composantes du système hybride – objet et contrainte – se fait à l'aide de *cellules* (*cf.* figure 5.9) attachées à la facette *valeur* des attributs contraints. Dès qu'une valeur est obtenue (par héritage, défaut, attachement procédural), elle est placée dans la cellule correspondante, et est considérée comme une valeur par défaut par le réseau de contraintes. Si la valeur est inférée par le réseau de contraintes, elle est également placée dans la cellule, les attachements procéduraux (démons) associés à l'attribut calculé sont déclenchés pour valider cette valeur.

Lorsque le système de contraintes a besoin de la valeur d'une variable attachée à un attribut, il peut solliciter le système de frames afin que celui-ci détermine la valeur par héritage, par défaut ou par attachement procédural.

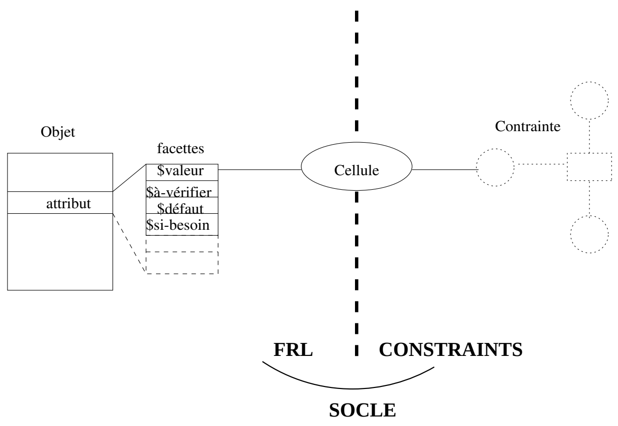


FIG. 5.9 - : L'architecture proposée pour contraindre les objets de FRL à l'aide de CONSTRAINTS dans SOCLE.

Deux cas extrêmes sont envisagés par Harris dans la répartition des rôles entre les deux systèmes couplés par cette hybridation. D'un côté, charger le système de contraintes de réaliser toutes les inférences du système de frames, notamment l'héritage. De l'autre, reporter toutes les informations concernant la propagation de contraintes dans les facettes des attributs. La solution réside dans la répartition des responsabilités des deux systèmes :

- La représentation de connaissances structurées et de formules est le fait du système de frames. L'instanciation d'un frame déclenchant les contraintes de classes associées, le réseau est installé par le système de contraintes.
- La propagation de valeurs est le fait du système de contraintes. Le système de frames vient épauler le système de contraintes lorsque des valeurs sont réclamées.
- Pour le système de frames, une valeur par défaut est utilisée lorsqu'aucune autre n'est disponible ou héritable. Si elle crée une situation contradictoire pour le système de contraintes, elle est rejetée.
- Facettes de types et formules définissent le domaine d'une valeur.
- La résolution de contradiction basée sur la remise en cause de variables s'appuie sur les niveaux de confiance que l'utilisateur a en ces valeurs. Les prémisses ne changent de valeur qu'au profit d'un plus fort niveau de confiance. Le système de contraintes se charge de la modification de valeurs non prémisses. En cas de contradiction, l'aide de l'utilisateur est sollicitée.
- Les dépendances entre attributs au travers des formules sont enregistrées par le système de contraintes et permettent des explications sur les valeurs stockées par le système de frames.

Les partisans du tout objet reprocheront à SOCLE de ne pas tirer parti de la présence de FRL pour représenter les contraintes par des objets. Un autre reproche est que SOCLE ne repose que sur la propagation de valeurs. Ceci est dû aux limites de FRL, le langage de frames, dans lequel un domaine de valeurs possibles ne peut être associé aux attributs. Aussi, un attribut d'un frame a-t-il une valeur (fournie ou inférée) ou n'en n'a pas et est alors en situation d'attente.

5.1.8 Autres systèmes

Il existe de nombreux autres systèmes intégrant objets et contraintes, nous avons voulu décrire ici les principaux représentants. Ainsi, à la liste des systèmes décrits ici, on peut ajouter :

- Equate [Wilk91] est un résolveur de contraintes qui préserve l'encapsulation des objets en programmation orientée-objet. Les contraintes accèdent aux objets par les méthodes de l'interface. Les contraintes, limitées aux équations, sont décomposées en contraintes plus petites dont les solutions connues sont combinées pour trouver une solution globale.
- Alien [Beaudoin-Lafon et al.91] est un modèle qui intègre contraintes et prototypes, destiné au développement d'interfaces graphiques. Les contraintes sont également des objets et sont organisées en hiérarchies, comme dans ThingLab II. Ici aussi, la résolution se limite à la propagation locale qui peut s'appliquer à n'importe quel type de contraintes (au contraire de la résolution, par exemple, qui ne s'applique qu'en cas de contraintes numériques linéaires) mais qui échoue en présence de cycles.
- Constraint-Lisp [Liu et al.92] est un langage de programmation orientée-objet qui est au couple (Common-Lisp, CLOS) ce que Pecos est au couple (Le-Lisp, Micro-Ceyx) puisqu'on y retrouve à peu près les mêmes fonctionnalités (ensembles de contraintes prédéfinies, variables contraintes de n'importe quel type, contraintes sur des objets, paramétrage de la résolution, optimisation...). Deux différences notables : les domaines des variables entières peuvent être des unions disjointes d'intervalles et une contrainte peut être posée sur une instance particulière, la définition d'une contrainte de classe impliquant une redéfinition de la fonction de création d'instance. Par contre, la définition de nouvelles contraintes n'est pas possible.
- Le système Laure [Caseau94] est un langage orienté-objet hybride qui permet à l'utilisateur de combiner règles, contraintes et méthodes sur les mêmes objets dans un même programme. Créé il y a dix ans par Yves Caseau, il a subi encore de nombreuses améliorations, sur chacune

des trois composantes qu'il intègre.

Les contraintes portent sur les attributs d'un ensemble d'objets, quel que soit leur domaine de valeurs. Une sous-classe hérite des contraintes de sa sur-classe. Il est possible de contraindre un seul objet (qui appartient alors à un singleton) et de représenter un domaine complexe par une *abstraction*. Par exemple, on peut définir, à l'aide d'une classe, l'abstraction de domaine *rectangle* et étendre l'ensemble des opérations disponibles sur ce nouveau domaine. Le résolveur sera alors capable de raisonner sur ces nouveaux domaines.

Ici, les contraintes sont décrites à l'aide d'un langage déclaratif et sont réécrites par le système dans une représentation algébrique en un ensemble de fonctions assurant la réduction des domaines et la propagation de valeurs. Cette technique d'expansion des contraintes, bien que coûteuse en place mémoire, procure à LAURE d'excellentes performances dans la résolution des CSP, qu'explique aussi un typage fort mis en place à l'aide de treillis, des techniques de compilation et d'optimisation des structures de données puissantes. Pour la propagation de contraintes, on trouve dans LAURE un algorithme semblable à celui de PECOS. Le *Backtrack* fourni pour la résolution des contraintes (sur des domaines finis exclusivement) est ouvert et peut être dirigé par des heuristiques définies à l'aide des règles de production ou des méthodes offertes par le système.

- LIFE [Aït-Kaci et al.93] est un langage de programmation qui est à la fois un langage de programmation logique (les structures de données de base sont des ψ-termes qui peuvent être employées dans des clauses à la Prolog dont l'unification est ici étendue), un langage de programmation fonctionnelle (il est possible de définir des fonctions) et un langage de programmation orientée-objet (un ψ-terme correspond à un ensemble d'objets ou *sorts*, qui peuvent être décrits par des attributs et organisés en hiérarchies sur lesquelles un héritage de propriétés, éventuellement multiple, est mis en place). C'est également un langage de programmation logique par contraintes puisque l'on peut contraindre des *sorts* ; les contraintes étant résolues par l'unification puissante de LIFE).
- Enfin, dans le domaine de la programmation logique par contraintes concurrente, nous citerons pour référence le langage OZ [Smolka et al.93].

5.2 Conclusion

Les systèmes présentés dans ce chapitre mettent en commun objets et contraintes. Cependant, il est possible de les distinguer les uns des autres selon plusieurs critères :

1. La finalité. On peut distinguer les systèmes dévoués à la gestion d'interface graphique (famille THINGLAB, GARNET, EQUATE, ALIEN), les bibliothèques de programmation par contraintes (PECOS, ILOG-SOLVER, CONSTRAINT-LISP) qui étendent un langage particulier, les boîtes à outils de programmation par contraintes (PROSE, CSPOO) qui sont destinées à être immergées dans n'importe quel langage à objets, les extensions de langage à objets vers les contraintes (COOL, SOCLE) et les langages hybrides (LAURE).
2. Le type des variables contraintes. Entiers et réels sont souvent représentés. En règle générale, les systèmes pour lesquels les contraintes sont à définir traitent des types quelconques. Des contraintes peuvent également être fournies sur les types prédéfinis (entiers, réels, booléens...).
3. Le domaine des variables contraintes. Tous ces systèmes n'associent pas forcément d'ensembles de valeurs possibles aux variables contraintes (famille THINGLAB, GARNET, SOCLE). Lorsque c'est le cas, il peut être fini (PROSE) et continu (PECOS, ILOG-SOLVER, CONSTRAINT-LISP), sous forme d'énumération ou d'intervalle. Les domaines peuvent également avoir une structure hiérarchique qui est exploitée par les algorithmes de traitement du CSP (CSPOO).

4. Les contraintes, selon le type des variables, peuvent être arithmétiques, booléennes, symboliques. Elles peuvent être fonctionnelles[4] (famille ThingLab, Garnet, Socle, Prose, CSPOO) ou non fonctionnelles (Pecos, Ilog-Solver, Laure), monodirectionnelles (Garnet) ou multidirectionnelles, linéaires ou non linéaires. Elles peuvent être également ordonnées par préférence (ThingLab II, Kaleidoscope).
5. Les techniques de consistance et de résolution employées. On peut distinguer divers traitements non forcément exclusifs :
   - la propagation de valeur qui consiste, lors de la modification de la valeur d'une variable à propager ce changement aux variables attenantes. Les systèmes basés sur un modèle de perturbation emploient cette technique. Lorsqu'une solution est perturbée par une modification, le système tente de retrouver une stabilité à travers un graphe d'activation de méthodes à exécuter. Les contraintes sont ici fonctionnelles. Ces systèmes (famille ThingLab, Garnet, Kaleidoscope, Prose) réalisent du maintien de solution.
   - la propagation de contraintes ou propagation de cohérence est appliquée dans les systèmes où les variables ont un domaine associé (Pecos, Ilog-Solver, Laure, CSPOO, Cool). Elle consiste, lors de la définition d'une contrainte, à établir la consistance des domaines des variables de la contrainte et à propager aux variables attenantes ces changements en réactivant les contraintes voisines.
   - la résolution ou satisfaction de contraintes (Prose, CSPOO, Cool, Pecos, Ilog-Solver, Laure) proprement dite consiste à déterminer les solutions du CSP. Diverses techniques peuvent être employées qui dépendent du type et du domaine des variables, mais aussi des contraintes.
6. Le type de liaison objets/contraintes. Les contraintes peuvent être définies et décrites à l'intérieur des objets (famille ThingLab, Garnet, Kaleidoscope). On peut alors parler d'intégration de contraintes dans les objets. Les contraintes peuvent être posées sur les objets. Plusieurs configurations sont alors possibles :
   - un couplage entre un langage de programmation par contraintes et un langage à objets. On distingue alors les couplages faibles (Socle) qui sont assurés par une interface entre les deux systèmes, et les couplages forts (Cool, Prose, CSPOO) dans lesquels les contraintes sont décrites elles-mêmes par des objets du langages à objets.
   - une extension d'un langage à objets existant par la définition d'une bibliothèque de programmation par contraintes (Pecos, Ilog-Solver, ConstraintLisp).
   - un langage hybride intégrant objets, contraintes et d'autres paradigmes (Laure)

Si l'on considère les avantages de chacun des paradigmes – objet et contraintes –, on peut trouver les motivations de chacun des systèmes présentés ici malgré leur diversité.

Le paradigme "objet" offre :

- des entités qui rassemblent les connaissances propres à une famille ou à un seul élément du domaine d'application.
  Dans les langages à objets intégrant des contraintes, ce principe est exploité dans la définition de contraintes à l'intérieur des classes. Les contraintes de classes induisent alors des contraintes sur chacun des instances de la classe.
  Dans les langages de contraintes utilisant les objets pour représenter les composants (voire les techniques de résolution) d'un CSP, la classe est un moyen de décrire les caractéristiques de ces composants – c'est le rôle des attributs –, mais aussi les méthodes à activer, comme la propagation, lors de la modification du domaine d'une variable ou la pose (instanciation) d'une contrainte.
- une organisation hiérarchique des descriptions sur laquelle se greffe un mécanisme d'héritage qui permet une factorisation de la connaissance.

[4] Une contrainte $c(x_1, \ldots, x_n)$ est fonctionnelle si $\forall x_i$ et $\forall (v_1, \ldots, v_{i-1}, v_{i+1}, v_n)$ où $vk \in dom(x_k)$, $\exists !\ v_i \in dom(x_i)$ tel que $(v_1, \ldots, v_{i-1}, v_i, v_{i+1}, v_n)$ soit localement consistante.

Dans les langages à objets intégrant des contraintes, ce principe est exploité à travers l'héritage des contraintes de classe, qui permet de factoriser la déclaration de contraintes.
Dans les langages de contraintes utilisant les objets comme éléments de représentation, l'organisation des divers composants en hiérarchies permet d'associer des comportements communs (par héritage) ou spécifiques (par redéfinition d'attributs, de méthodes) aux divers types de variables ou de contraintes. Lorsque les domaines sont eux-mêmes organisés hiérarchiquement, cet ordre est exploité pour accélérer les processus de maintien de cohérence et/ou de résolution.

- des mécanismes de création et de manipulation d'objets.
  Dans les langages à objets intégrant des contraintes de classe, les contraintes d'une classe sont associées à une instance dès sa création. La modification d'un objet donne lieu à une propagation.
  Dans les langages de contraintes utilisant les objets comme éléments de représentation, l'instanciation correspond à la définition d'une variable, à la pose d'une contrainte ou à la définition d'un CSP et d'un algorithme de résolution.
  Dans les langages qui font les deux, cela permet la définition de méta-contraintes en tant que contraintes sur des attributs de contraintes.
- des liens entre objets modélisés par les attributs de ces objets dont la valeur est un (ou plusieurs) autre(s) objet(s).
  Dans les langages à objets intégrant des contraintes, ces liens (relations), parcourus par des chemins, permettent de contraindre des attributs d'autres objets. Les contraintes viennent se greffer sur les relations.
- une représentation de la connaissance par divers niveaux d'abstraction ou de *décomposition* : un *triangle* est formé de *points* dont les coordonnées sont des *entiers*. Les contraintes portant sur les entités des plus hauts niveaux se décomposent récursivement en contraintes de niveaux inférieurs. Deux triangles sont égaux si leurs points sont égaux, deux points sont égaux si leur coordonnées sont égales. Retrouver ces niveaux d'abstraction sur les contraintes réduit considérablement l'effort de définition des CSP : la contrainte complexe n'est définie qu'une seule fois en fonction des contraintes de niveau inférieur qui la composent. À sa création, ses sous-contraintes sont créées également.
  Dans les langages à objets intégrant des contraintes, cela permet la définition de contraintes sur des objets ou sur des attributs dont la valeur est un objet. Les variables contraintes ne se limitent donc plus ici aux types simples habituels, mais peuvent être de n'importe quel type.
  Dans les langages de contraintes utilisant les objets comme éléments de représentation, la description de la contrainte composée doit faire référence à ses contraintes composantes. L'instanciation se charge de la pose des contraintes composantes de plus bas niveau.

Le paradigme "contrainte" offre :

- la possibilité de décrire un problème par l'énoncé de relations entre les données de ce problème. Dans les objets, on utilise la déclarativité pour définir les relations qui doivent être maintenues entre les attributs (d'un même objet, d'objets différents, de tous les objets d'une classe...) ou entre des objets ;
- un arsenal de techniques (propagation de valeur, propagation de contraintes, arc-consistance, algorithmes et heuristiques de résolution) sous-jacentes à ces énoncés, qui permettent, sinon de résoudre le problème, tout au moins de réduire son espace de solution. Les associations objets/contraintes se font sur un modèle de *perturbation* – c'est le cas la plupart du temps dans les systèmes pour le développement d'interface – ou sur un modèle de *raffinement*. Dans le premier cas, le système a une solution et doit réagir à une perturbation en établissant un plan de méthodes à exécuter. Dans le second cas, les contraintes sont posées et agissent par réduction sur les domaines des valeurs possibles des variables contraintes, la résolution constituant l'étape suivante.

Si les systèmes étudiés dans ce chapitre ont, au départ, la préoccupation commune de lier objets et contraintes, les moyens d'y parvenir sont assez distincts. Dans le cas particulier qui nous intéresse, la revue précédente ne révèle que trois[5] exemples d'extension d'un langage de représentation par objets vers les contraintes : SOCLE est le couplage faible de FRL et de CONSTRAINTS, COOL est un système hybride étendant KEE vers les contraintes que l'on peut considérer comme un couplage fort, et CSPOO est fortement couplé à Y3.

Pourtant, aucune de ces trois propositions n'étudie les conséquences de l'introduction de contraintes dans une base de connaissances. Ainsi, les questions suivantes – qui sont celles que tout concepteur d'un système de représentation de connaissances par objets doit se poser avant d'entreprendre une extension des objets vers les contraintes – restent en suspens :

- Quels sont les types des CSP ? Quelles techniques semblent plus appropriées ?
- Comment représenter les contraintes ? Quelles sont les conséquences de leur présence en tant qu'objets à part entière ? Quelle solution de couplage est la mieux adaptée compte tenu des exigences du SRPO ?
- À quels niveaux de représentation déclarer des contraintes ? Que signifie la présence d'une contrainte ? Quelle est sa portée ?
- Quelle est la priorité à accorder aux contraintes ? Sont-elles un moyen d'inférence supplémentaire ? Sont-elles des traits descriptifs de classe au même titre que les attributs ?
- Quelles sont les répercussions de la présence de contraintes sur l'instanciation, l'héritage, les autres moyens d'inférence, la classification ?
- Peut-on toujours garantir la consistance d'une base de connaissances contrainte, comme on le peut en leur absence ?
- Comment prolonger la dynamicité intrinsèque des bases de connaissances auprès des contraintes ?
- Les contraintes sont-elles d'un apport particulier pour le système lui-même ? Peuvent-elles aider au développement ou à l'expression d'autres fonctionnalités du modèle ?

C'est à ces questions que nous devons maintenant apporter une réponse, sinon dans le domaine des représentations à objets en général, tout au moins dans le contexte du système TROPES.

[5] Un seul, SOCLE, si on exclut les langages de RPO hybrides.

Deuxième partie

Intégration de contraintes à Tropes

Chapitre 6

Les CSP Tropes

Le chapitre 3 a révélé que le modèle TROPES ne fournit pas les outils permettant d'exprimer et de maintenir la cohérence de propriétés ou contraintes (numériques et symboliques) impliquant plusieurs attributs d'une classe. Dans la perspective de doter TROPES de tels outils, nous nous sommes intéressés aux principes de la programmation par contraintes (*cf.* chapitre 4) et aux systèmes où objets et contraintes cohabitent (*cf.* chapitre 5).

Permettre un raisonnement par contraintes dans les bases de connaissances doit consister, dans TROPES, d'une part à donner les moyens à un utilisateur de définir des contraintes, et, d'autre part, à disposer d'outils pour gérer ces contraintes. Les décisions à prendre ici pour l'expression et la gestion des contraintes doivent tenir compte des particularités du système, tant de ses principes de modélisation que de son utilisation.

À travers l'étude de l'expression et de la gestion de problèmes de satisfaction de contraintes entre les objets de TROPES (nous les appellerons CSP TROPES) que nous abordons ici, nous établissons un cahier des charges des contraintes, fonctionnalités et capacités techniques requises du module de programmation par contraintes couplé à TROPES.

Pour commencer cette étude, il nous faut tout d'abord définir la nature de chacun des composants d'un CSP TROPES : les variables, les domaines et les contraintes (*cf.* section 6.1).

Une fois les CSP définis, il reste à déterminer à quels niveaux de représentation une contrainte peut être posée dans TROPES (*cf.* section 6.2).

Afin de désigner un attribut à contraindre dans un objet TROPES, nous introduisons la notion d'*accès* (*cf.* section 6.3) qui étend la notion classique de *chemin* (présente, par exemple, dans THINGLAB et CSPOO) aux attributs multivalués.

Connaissant les composants des CSP TROPES, les niveaux de représentation des contraintes et le moyen de désigner des attributs à contraindre, nous débattons sur le choix d'une représentation interne ou externe des contraintes pour juger de la pertinence et de la cohérence de définir et de conserver des traces des CSP TROPES en tant qu'objets du modèle (*cf.* section 6.4).

Le choix étant fait d'une représentation externe, nous définissons le profil du module de gestion des CSP TROPES (*cf.* section 6.5) en énonçant les tâches qu'il doit accomplir.

6.1 Les composants d'un CSP dans Tropes

Au travers des besoins énoncés (*cf.* section 3.5), les attributs des objets TROPES apparaissent comme les premières variables contraintes à considérer dans TROPES. D'autres entités de représentation peuvent-elles être contraintes et faire également office de variables contraintes des CSP TROPES? Quelles sont les contraintes indispensables à considérer en fonction des variables contraintes et de leur domaine? Lesquelles sont envisageables? Cette section vise à répondre à ces questions.

6.1.1 Les variables contraintes

Dans TROPES, il existe sept niveaux distincts de représentation (*cf.* chapitre 3) : base de connaissances, concept, point de vue, classe, instance, attribut, facette. Afin de déterminer quelles sont les entités candidates à être variables d'un CSP dans le modèle, nous examinons pour chacune d'elles quel genre de contraintes pourrait lui être associé (*cf.* tableau 6.1).

Entité de représentation	**Contraintes unaires de contrôles**	**Contraintes de comparaison**
Base de connaissances	nombre de concepts, de points de vue, de classes par points de vue, d'attributs, d'instances, etc.	comparaisons entre les caractéristiques et les contenus de plusieurs bases de connaissances...
Concept	nombre de points de vue, de classes par points de vue, d'attributs, d'instances, la comparaison des points de vue, etc.	comparaisons entre les caractéristiques et les contenus de plusieurs concepts...
Point de vue	nombre de classes, les concepts visibles, les attributs de ces points de vue, etc.	comparaisons entre les caractéristiques et les contenus de plusieurs points de vue...
Classe	nombre d'instances, les attributs, les sous-classes, les sous-classes, le point de vue, le nombre d'instances, leur utilisation lors de la classification, le nombre d'instances satisfaisant une condition, etc.	comparaisons entre les caractéristiques et les contenus de plusieurs classes...
Instance	appartenance à une classe, la non-appartenance, le nom de l'instance, le nombre d'attributs valués, etc	comparaisons entre les caractéristiques et les contenus de plusieurs instances...
Attribut	caractéristiques de l'attribut (sa nature, son genre, ses facettes, etc.)	contraintes unaires portant sur sa valeur, comparaisons entre les caractéristiques et les valeurs de plusieurs attributs...
Facette	associées à l'attribut	associées à l'attribut

TAB. 6.1 - : Les contraintes envisageables sur les diverses entités de représentation de TROPES.

6.1.1.1 Choix des variables contraintes

Le tableau précédent montre que l'on peut *a priori* considérer la plupart des entités de représentation comme des variables contraintes : il suffit de disposer d'opérateurs de manipulation et de combinaison pour construire des contraintes.

Cependant, en considérant qu'il est possible de fournir une définition de TROPES en TROPES, réalisée dans un méta-modèle, dans laquelle les différentes entités de représentation – bases de connaissances, concepts, points de vue, classes, instances, attributs et facettes – sont décrites par des concepts et effectivement représentées par des instances, il est alors possible de restreindre l'ensemble des entités potentiellement contraignables aux seuls instances et attributs. En effet, exprimer une contrainte de contrôle unaire sur une entité peut alors être fait, moyennant des opérateurs adéquats, sur l'instance qui représente cette entité (cette base, ce concept, cette classe...). On agira ici directement sur les attributs de cette instance qui déterminent les caractéristiques de l'entité (par exemple, sur l'attribut *liste des attributs* d'une instance du concept Concept...).

Exprimer des contraintes n-aires de contrôle entre des entités de représentation (des bases, des concepts, des classes...) nécessite alors de disposer de contraintes portant sur des instances. Mais, la comparaison, qui s'opère au titre de la contrainte entre deux (ou plus) instances, s'effectue à un niveau de représentation inférieur puisqu'elle peut être traduite en une ou plusieurs comparaisons entre attributs de ces instances. Par exemple, si on veut que deux concepts aient le même nombre d'instances, on reportera la comparaison au niveau des attributs *instances* des deux concepts. Ceci suggère donc qu'au niveau le plus bas, ce sont toujours les attributs qui constituent les variables contraintes.

Pour une instance, nous avons envisagé des contraintes de contrôle qui ne régissent pas les valeurs des attributs de l'instance mais qui agissent directement sur les caractéristiques ontologiques (nom, lien d'appartenance) ou descriptives (nombre d'attributs valués, contraintes sur les facettes). C'est pourquoi nous les considérons plutôt comme des méta-contraintes (au sens de contraintes sur la structure de l'instance et non pas sur son contenu) qui ne participent pas à la description de relations entre les diverses caractéristiques de l'individu représenté, ou bien encore, entre certaines caractéristiques de l'individu représenté et les caractéristiques d'autres individus.

Exprimer une contrainte sur des instances qui ne soit pas une contrainte de contrôle revient à lier ces instances par le biais de leurs attributs. Aussi, une contrainte sur des instances se décompose-t-elle en fait en contraintes sur des attributs de ces instances. C'est pourquoi nous pouvons conclure que les seules variables contraintes qui sont considérées dans les CSP Tropes sont les attributs. Cependant, il peut être commode, dans un souci de factorisation et de déclarativité, de rassembler dans l'expression simple d'une contrainte sur des instances l'ensemble des contraintes sur attributs auquel elle est équivalente (par exemple, l'égalité de deux instances du concept Point se traduit en deux égalités sur les attributs représentant les abscisses et les ordonnées de ces points). La possibilité de définir des contraintes sur des instances à base de contraintes sur des attributs doit donc être offerte pour des besoins de déclarativité. Les instances sont dans ce cas des pseudo-variables contraintes, la contrainte les implique mais ce sont réellement leurs attributs qui sont contraints.

On peut remarquer que les contraintes sur des attributs dont la valeur est une instance sont équivalentes à des contraintes sur des instances qui, elles-mêmes, sont des contraintes sur des attributs.

Les variables des CSP définis dans Tropes sont donc les attributs quel que soit leur type, leur instance, leur classe, leur point de vue, ou leur concept d'appartenance.

6.1.2 Les domaines contraints

Les attributs ayant été choisis comme variables contraintes des CSP définis dans Tropes, les domaines contraints sont donc les domaines de ces attributs.

Un attribut Tropes est soit d'un type prédéfini (entier, réel, booléen...), soit d'un type élaboré (le type d'un concept, d'une classe ou d'un ensemble de classes présents dans la base de connaissances), soit d'un type défini (qui n'est pas modélisé dans la base mais défini à l'aide du langage de définition de types du module Metéo [Capponi95]).

La structure des domaines est un élément non négligeable dans le choix d'une bibliothèque de contraintes. Cette structure sera en effet manipulée par les règles de maintien de la consistance associées aux contraintes. Dans Tropes, cette structure est polymorphe puisqu'un domaine est exprimable sous la forme d'un ensemble (énumération) de valeurs mais aussi d'une union d'intervalles dans le cas de types ordonnés. De même, pour les attributs multivalués, un ensemble d'ensembles ou de listes peut être décrit, soit par le type de ses éléments[1] et, optionnellement, les facettes d'exclusion *sauf* et de cardinalité *card*, soit par une énumération de ces ensembles ou de ces listes. Les règles de maintien de la consistance devront prendre en compte ces différentes formes.

6.1.3 Les contraintes

Il s'agit ici de déterminer le type des contraintes mises à la disposition de l'utilisateur afin de contraindre les attributs de TROPES. À travers cet éventail des contraintes souhaitées, c'est aussi le profil du module chargé de la gestion des contraintes qui se dessine.

Considérer les types des variables contraintes, donc des attributs de TROPES, peut nous mettre sur la voie des contraintes à proposer aux utilisateurs. Toutefois, on ne peut préjuger ni des types définis à l'aide du module de gestion de types de TROPES, ni des concepts qui seront référencés par les attributs complexes dont la valeur est une ou un ensemble ou une liste d'instances. En revanche, TROPES intègre les types prédéfinis classiquement (entiers, réels, booléens...) qui sont ceux de son langage hôte mais aussi ceux prédéfinis dans le module de gestion des types.

Aussi, à première vue, il existe deux sortes de contraintes : celles que l'on peut prévoir et qui portent sur des attributs de types prédéfinis, et celles qui pourront être ajoutées et porteront sur des variables dont le type élaboré et défini – donc non prévisible – sera maintenu par le module de gestion de types. Ce type sera alors soit issu d'un concept, d'une classe ou d'un ensemble de classes, soit non décrit dans une base de connaissance mais défini grâce aux fonctionnalités de définition de types du module METÉO. Nous limitons notre étude à l'expression de contraintes sur les types prédéfinis mais donnerons quelques éléments de réponse pour l'expression de CSP sur des attributs de types définis ou élaborés (*cf.* chapitre 14).

Pour les attributs monovalués de type prédéfini, les contraintes qui peuvent être exprimées reposent sur des opérateurs arithmétiques ou booléens... Il faut donc proposer une bibliothèque de contraintes arithmétiques et booléennes. Fournir cet ensemble de contraintes paraît être la tâche la plus classique : elle est accomplie par la plupart des langages ou des bibliothèques de programmation par contraintes.

Pour les attributs multivalués, les contraintes qui peuvent être exprimées reposent sur des opérateurs ensemblistes ou sur des listes... Cette tâche nous paraît déjà plus difficile – aucun système à contraintes de notre connaissance ne propose un ensemble suffisamment complet de telles contraintes – et plus spécifique à la considération des attributs multivalués de TROPES. Un aspect intéressant est que les opérateurs d'ensembles ou de listes font le plus souvent abstraction des types des éléments des ensembles ou des listes.

Il semble également indispensable de disposer de contraintes d'égalité et de différence capables de traiter des attributs de tous types, si la comparaison de leurs valeurs est possible.

Il apparaît donc que la bibliothèque des contraintes destinées aux attributs de TROPES doit être adaptée à la fois :

- aux types prédéfinis : elle devra contenir des contraintes arithmétiques ou booléennes classiques ;
- à la structure des domaines : elle devra manipuler des domaines sous la forme d'énumération de valeurs, d'unions d'intervalles, d'ensemble d'ensembles ou de listes, et de descriptions d'ensemble d'ensembles ou de listes ;
- aux attributs multivalués : elle devra contenir des contraintes portant sur des listes et des contraintes portant sur des ensembles.

Il nous semble aussi souhaitable de proposer à l'utilisateur la définition de contraintes *composées* qui représentent un ensemble de contraintes prédéfinies ou elles-mêmes composées. Ceci afin de factoriser des expressions de contraintes et de permettre, par exemple, des contraintes dont les variables sont des instances mais qui sont équivalentes à un ensemble de contraintes sur des attributs

[1] les constructeurs *liste* et *ens-de* n'opèrent que sur un même type.

de ces instances (par exemple, l'égalité de deux instances du concept POINT correspond à l'égalité des attributs représentant les coordonnées).

Nous dressons dans la section suivante un inventaire des contraintes à mettre à la disposition des utilisateurs de TROPES.

6.1.4 Les contraintes attendues

6.1.4.1 Les contraintes d'égalité et de différence

On les retrouve opérantes pour chacun des types simples (entiers, réels, booléen, chaîne, ...) implantés en machine. Pour les valeurs d'un autre type, la comparaison a lieu sur les expressions de ces valeurs.

6.1.4.2 Les contraintes numériques

Disposer de contraintes numériques (sur des entiers ou sur des réels) est indispensable : les problèmes à contraintes (optimisation, allocation de ressources, etc.) l'attestent qui se modélisent sous la forme de problèmes mathématiques à données numériques.

Elles s'adressent plus particulièrement aux attributs monovalués de type entier ou réel. Les contraintes que l'on peut construire se basent sur les opérateurs classiques de comparaison ($=, \neq, <, \leq, >, \geq \ldots$) et les expressions qu'elles contiennent sont établies à partir des divers opérateurs unaires ($-, valeur\ absolue, log, exp, sin, cos \ldots$) et binaires ($+, -, *, / \ldots$) de l'arithmétique classique.

La distinction contraintes entières/contraintes réelles sera certainement à observer, d'une part parce que ces types ne sont pas représentés, ni traités de la même manière en machine et, d'autre part, parce que $\mathbb{R}$ est dense, ce qui n'est pas le cas de $\mathbb{N}$.

6.1.4.3 Les contraintes booléennes

Les contraintes booléennes ont déjà une application plus particulière. Elles doivent permettre de contraindre des expressions booléennes à être égales ou différentes. Ces expressions booléennes peuvent être construites à partir de variables booléennes en combinant les opérateurs unaires ($\neg$) ou binaires ($\wedge, \vee, \Rightarrow, \Leftrightarrow$) de l'algèbre de Boole. Ces expressions booléennes peuvent également résulter de comparaisons ($=, \neq, <, \leq, >, \geq \ldots$) entre des variables de types quelconques. L'utilité des contraintes booléennes est ici en majeure partie justifiée par la prise en compte de contraintes *conditionnelles*.

Les contraintes numériques ou booléennes sont des contraintes classiques et indispensables dans la plupart des langages de programmation par contraintes (*cf.* chapitre 4). Dans le cadre d'un SRPO tel que TROPES, ces contraintes sont nécessaires, mais il convient de compléter l'ensemble des contraintes mises à la disposition de l'utilisateur en tenant compte des spécificités du modèle. C'est pourquoi, dans les sections suivantes, nous proposons des contraintes particulières qui tiennent compte du type des attributs (contraintes ensemblistes, contraintes sur des listes), du caractère flexible et évolutif de la connaissance représentée (contraintes conditionnelles, contraintes disjonctives, contraintes de priorité, contraintes d'évolution et contraintes globales). Intégrer ce genre de contraintes dans TROPES accroît plus encore les capacités descriptives du modèle.

6.1.4.4 Les contraintes ensemblistes

Les contraintes ensemblistes ont pour but de contraindre des attributs dont la valeur est un ensemble de valeurs d'un même[2] type quelconque.

[2] C'est une des restrictions de cette première version due au fait que le langage d'expression des types des attributs de TROPES ne permet pas de constituer des ensembles ou des listes d'éléments de types différents.

La prise en compte de telles contraintes (au même titre que les contraintes de listes) est justifiée par la présence dans TROPES d'attributs multivalués et par le souci de permettre la définition de contraintes sur les attributs monovalués aussi bien que les attributs multivalués.

Les contraintes ensemblistes doivent être construites à partir des opérateurs ensemblistes de comparaison, d'appartenance, de construction d'ensembles.

6.1.4.5 Les contraintes de listes

Les contraintes de listes ont pour but de contraindre des attributs dont la valeur est une liste de valeurs d'un même type quelconque. Elles doivent être construites à partir des opérateurs classiques sur les listes, comparaison, accès à un élément, application d'une fonction à une liste, construction de liste. Plus particulièrement, les fonctions de manipulation de listes offertes par le langage hôte Lisp seront mises à profit.

Les composants d'un objet composite étant souvent groupés dans des ensembles (voire des listes), les contraintes sur des ensembles ou sur des listes paraissent donc d'un intérêt certain pour exprimer des contraintes entre un composite et ses composants.

6.1.4.6 Les contraintes conditionnelles

Dans le souci d'offrir une certaine flexibilité dans la définition des CSP, nous proposons de mettre à la disposition de l'utilisateur des contraintes s'inscrivant dans un mode non-déterministe de programmation par contraintes : les contraintes *conditionnelles*, les contraintes *disjonctives* et les contraintes *de priorité*.

Une base de connaissances TROPES est soumise à diverses modifications qui sont autant de transitions d'un état à un autre. Il paraît intéressant de lier l'existence d'une contrainte à la satisfaction d'une condition (et non pas d'une contrainte). On peut concevoir deux contraintes conditionnelles basées sur le modèle des «si *condition* alors *bloc1*» et «si *condition* alors *bloc1* sinon *bloc2* » des langages de programmation.

les deux contraintes conditionnelles sont :

- la contrainte de type *si alors* : si *condition* alors *contr1*
- la contrainte de type *si alors sinon* : si *condition* alors *contr1* sinon *contr2*

À la pose de la contrainte conditionnelle, la condition est évaluée, entraînant ou non la pose des contraintes contenues dans le bloc désigné. Durant toute la durée de vie de la contrainte conditionnelle, un démon d'activation est placé sur la condition. Si un événement survient qui modifie un des paramètres de la condition, cette dernière est ré-évaluée.

Pour la contrainte si *condition* alors *contr1*, le fonctionnement est donné par la procédure du tableau 6.2. De même, pour la contrainte si *condition* alors *contr1* sinon *contr2*, le fonctionnement est donné par la procédure du tableau 6.3. Ainsi, ce n'est pas la présence de la contrainte conditionnelle qui est remise en cause, mais les contraintes dont elle est à l'origine de la pose. La gestion de contraintes conditionnelles impose, d'une part d'être en mesure de ré-évaluer la condition lorsque un événement la concernant survient et, d'autre part, d'être capable d'annuler des contraintes dont la pose était justifiée par le résultat de l'évaluation précédente de la condition qui ne correspond plus au résultat de la dernière évaluation. Les contraintes conditionnelles ont donc un aspect dynamique.

6.1.4.7 Les contraintes disjonctives

Une contrainte disjonctive exprime une disjonction de deux contraintes C_1 et C_2. Au moment de la définition d'une contrainte disjonctive, la première contrainte C_1 est posée. Si elle est satisfaite alors la contrainte disjonctive est satisfaite. Si elle n'est pas satisfaite alors on supprime la contrainte C_1 et on pose la seconde contrainte C_2. Si C_2 est satisfaite alors alors la contrainte disjonctive est

```
si eval(cond) = t alors
    % la condition est évaluée à vrai %
    si  eval-prec = t alors rien
    % si elle avait été évaluée à vrai précédemment : rien à faire %
    sinon poser contr1
    % sinon, on pose la contraite %
sinon
    % la condition est évaluée à faux %
    si  eval-prec = t alors supprimer contr1
    % si elle avait été évaluée à faux précédemment : on supprime la contrainte qui avait été posée %
    sinon rien
    % sinon : rien à faire %
```

TAB. 6.2 - : Procédure d'évaluation de la contrainte si *condition* alors *contr1*. La fonction *eval(cond)* lance l'évaluation de la condition. La variable *eval-prec* conserve le résultat de l'évaluation précédente de la condition.

```
si eval(cond) = t alors
    si eval-prec = t alors rien
    sinon
        supprimer contr2
        poser contr1
sinon
    si eval-prec = t alors
        supprimer contr1
        poser contr2
    sinon rien
```

TAB. 6.3 - : Procédure d'évaluation de la contrainte si *condition* alors *contr1* sinon *contr2*. La fonction *eval(cond)* lance l'évaluation de la condition. La variable *eval-prec* conserve le résultat de l'évaluation précédente de la condition.

satisfaite, sinon C_2 est supprimée et la contrainte disjonctive n'est pas satisfaite.

Lorsque l'une des deux contraintes arguments d'une contrainte disjonctive n'est plus satisfaite suite à une modification dans la base de connaissances, on tentera de poser l'autre contrainte argument afin de satisfaire la contrainte disjonctive.

Ces contraintes disjonctives, comme les contraintes conditionnelles, ont pour arguments d'autres contraintes. C'est pourquoi nous appellerons ces contraintes des *méta-contraintes*.

6.1.4.8 Les contraintes de priorité

Une contrainte de priorité exprime des priorités dans la satisfaction de plusieurs contraintes. L'idée est d'associer à chaque contrainte d'une liste un coefficient (entre 1 exclu et 0 exclu) reflétant l'importance de la satisfaction de la contrainte. Si 1 est exclu, c'est parce qu'une contrainte à satisfaire à tout prix ne doit pas figurer ici. Si 0 est exclu, c'est qu'une contrainte qui ne doit pas être satisfaite n'est pas une contrainte.

Il s'agit donc ici de satisfaire le maximum de contraintes (en ce sens, une disjonction non-exclusive simple est exprimée par une liste de contraintes de même priorité). Toutes les contraintes sont posées. Si aucune d'elles n'est satisfaite, alors la contrainte de priorité n'est pas satisfaite ; ce n'est pas vraiment grave car le fait d'avoir exclu le 1 comme priorité absolue fait des contraintes de cette liste des contraintes ordonnées mais secondaires. Toutes les contraintes sont donc posées.

À toute contrainte de priorité $((c_1, p_1), (c_2, p_2), \ldots, (c_n, p_n))$, on associe un coefficient de satisfaction cs donné par la formule suivante :

$$cs = \frac{\sum_{i=1}^{n} satisf(c_i) * p_i}{\sum_{i=1}^{n} p_i}$$

où

$$satisf(c_i) = \begin{cases} 1 & \text{si } c_i \text{ satisfaite} \\ 0 & \text{si } c_i \text{ non satisfaite} \end{cases}$$

Puisque les contraintes sont toutes posées, la contrainte de priorité est chargée de contrôler l'échec éventuel d'une contrainte de sa liste : l'échec est ici accepté. Le coefficient de satisfaction de la contrainte de priorité est modifié lorsqu'un changement dans la base affecte une des contraintes de sorte que celle-ci soit satisfaite alors qu'elle ne l'était pas, ou l'inverse.

6.1.4.9 Les contraintes d'évolution

Les bases de connaissances TROPES pouvant être sujettes à de multiples modifications, il semble intéressant de disposer de contraintes d'évolution. Une contrainte d'évolution impose une condition entre la nouvelle valeur proposée pour un attribut et la valeur qu'il a actuellement (sa valeur courante). Les contraintes d'évolution ont donc au moins pour arguments la nouvelle et l'ancienne valeur. TROPES doit donc fournir les moyens de désigner la valeur précédente d'un attribut. Ce test de la satisfaction d'une telle contrainte ayant lieu lors de la proposition d'une nouvelle valeur, il n'est pas utile de conserver l'ancienne valeur au delà en cas de réussite.

6.1.4.10 Les contraintes globales

L'idée de contrainte globale est d'être en mesure de comparer un objet ou un ensemble (ou une liste) d'objets avec un ensemble (ou une liste) d'objets. Ainsi, il doit être possible de formuler une contrainte telle que : *le salaire de cet employé ne peut pas être supérieur à deux fois la moyenne des salaires des employés du même secteur.* Ce genre de contrainte nécessite la mise en place d'un mécanisme de sélection d'instances (ici, les employés du même secteur) qui sera abordé au chapitre 13.

6.1.5 Définition de contraintes

Il existe deux façons de définir des contraintes :

- soit, on fournit un langage complet de définition de contraintes qui permet de décrire les variables contraintes et les règles de maintenance associées à chaque nouvelle contrainte ;
- soit, on fournit un ensemble de contraintes de base qui peuvent être combinées pour construire des contraintes complexes. Ces contraintes complexes peuvent être, dans un souci d'économie d'écriture, à leur tour insérées dans l'ensemble des contraintes de base. Autrement dit, il ne s'agit pas ici de définir une contrainte en donnant sa sémantique à travers les règles de maintenance qui lui sont associées, mais bien d'intégrer une contrainte complexe – bâtie sur des contraintes de base dont les règles de maintenance auront été fixées par le concepteur du module de contraintes – dans l'ensemble des contraintes proposé par la bibliothèque.

Une tâche ardue est de proposer un langage pour la définition intégrale de nouvelles contraintes (première possibilité) afin d'étendre l'ensemble des contraintes de base et donc, celui des expressions contraintes utilisables. L'exemple d'une telle réalisation est la bibliothèque PECOS (*cf.* section 5.1.2.3) de programmation par contraintes du langage LE-LISP qui comprend une fonction de définition de contraintes. La définition d'une contrainte passe par la description du nom de la contrainte, de ses variables contraintes mais surtout des règles de maintien de consistance associées à la contrainte. Cette dernière description est rendue possible à la fois grâce à des primitives de reconnaissance d'événements de propagation et grâce à des primitives de manipulation de domaines. Mais, pour des règles nécessitant des calculs complexes, le recours à un langage de programmation (ici LE-LISP, le langage hôte de PECOS) est nécessaire.

Dans le contexte de l'utilisation d'un modèle de connaissances à objets intégrant des contraintes, il paraît contre-nature d'avoir ultimement recours à la programmation en un langage pour étendre la bibliothèque des contraintes disponibles. Aussi, une définition de nouvelles contraintes basée sur la seule intégration de contraintes complexes, construites à partir de contraintes de base, nous paraît être un premier objectif "honnête" à atteindre. En contrepartie, afin que l'absence d'un langage de définition de contraintes ne soit pas trop ressentie par les utilisateurs de TROPES, la bibliothèque de contraintes doit être initialement suffisamment complète.

6.2 Contraindre à différents niveaux de représentation

Dans la section 6.1.1, nous avons établi que les variables des CSP de TROPES sont les attributs des instances. Or, un attribut est la propriété d'un concept. Dans un point de vue, s'il est présent, il apparaît au niveau d'une classe et peut être affiné dans les sous-classes de celle-ci. Dans une instance, sa valeur peut-être connue ou non encore connue. Ces rappels nous amènent à considérer quatre niveaux de représentation – concept, point de vue, classe, instance – auxquels peut être attachée une contrainte.

Cette section décrit le comportement d'une contrainte selon le niveau de représentation auquel elle est déclarée. Dans les autres SRPO basés sur une approche classe/instance, cette étude se réduirait à la classe et à l'instance. Aussi, les décisions prises ici concernant la déclaration d'une contrainte au niveau d'une classe ou d'une instance sont généralisables à ces SRPO. Mais, de plus, dans le contexte de TROPES, il faut examiner les deux entités de représentation supplémentaires : le concept et le point de vue.

6.2.1 Contrainte de concept

Une contrainte de concept est une contrainte qui décrit une relation à maintenir entre des attributs de ce concept ou des attributs atteignables à partir de ce concept. En effet, puisqu'un attribut peut avoir pour valeur une instance, il peut permettre d'atteindre, par un moyen qui reste à définir, un attribut de cette instance ou de toute autre instance atteignable par plusieurs de ces indirections.

Une contrainte de concept est une caractéristique que doit observer chacune des instances du concept. À ce titre, elle fait partie de la définition du concept. La définition d'un concept est donc désormais constituée de deux parties : la partie concernant la déclaration des attributs du concept et la partie concernant les contraintes de ce concept (*cf.* figure 6.1).

En tant que trait définitionnel, chaque contrainte de concept est posée sur chacune des instances de ce concept (*cf.* figure 6.2).

Les contraintes de concept seront déclarées au niveau du concept. Elles seront rattachées à l'objet décrivant le concept. Un attribut particulier devra donc être disponible dans chaque concept qui fournira la liste des contraintes.

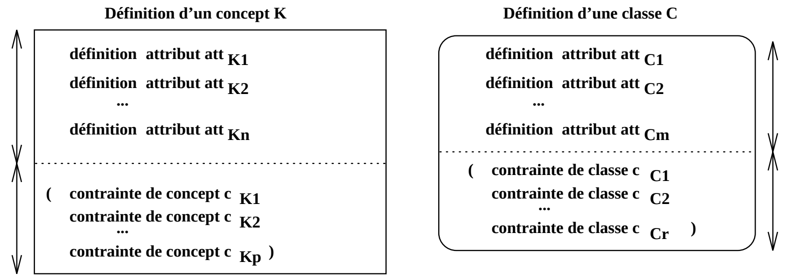


FIG. 6.1 - : Extensions des définitions de concept (respectivement de classe) par la partie contenant la liste des contraintes de concept (respectivement de classe).

6.2.2 Contrainte de point de vue

Une contrainte de point de vue est une contrainte qui décrit une relation à maintenir entre les attributs de ce point de vue (un point de vue masque certains attributs du concept).

Chaque instance du concept est rattachée à une classe dans ce point de vue. Aussi, chaque contrainte de point de vue est posée sur chacune des instances de ce concept. Dès lors, une contrainte de point de vue n'est autre qu'une contrainte de concept dont l'ensemble des attributs contraignables est circonscrit à l'ensemble des attributs du point de vue.

Une contrainte de point de vue pouvant être considérée comme un cas particulier de contrainte de concept, nous abandonnons ici cette distinction.

6.2.3 Contrainte de classe

Une contrainte de classe est une contrainte qui décrit une relation à maintenir entre les attributs de cette classe ou des attributs atteignables à partir de cette classe. Une contrainte de classe est une caractéristique que doit observer chacune des instances de la classe. À ce titre, elle fait partie de la définition de la classe. La définition d'une classe est donc désormais constituée de deux parties : la partie concernant la déclaration des attributs de la classe et la partie concernant les contraintes de cette classe (*cf.* figure 6.1).

En tant que trait définitionnel, chaque contrainte de concept est posée sur chacune des instances dont l'appartenance à cette classe est établie (*cf.* figure 6.2). Pour une instance attachée à une classe dans un point de vue, les contraintes de classe viennent s'ajouter aux contraintes de concept.

On associe donc aux contraintes de concept et aux contraintes de classe le même comportement, ce qui diffère ce sont les ensembles potentiels des attributs et des instances contraignables qui sont *a priori* plus réduits pour les contraintes de classe.

Les contraintes de classe seront déclarées au niveau de la classe. Elles seront rattachées à l'objet décrivant la classe. Un attribut particulier devra donc être ajouté à l'ensemble des attributs de l'entité utilisée pour décrire une classe.

6.2.4 Contrainte entre instances

Une contrainte entre instances est une contrainte qui décrit une relation à maintenir entre plusieurs instances quels que soient leurs classes et leur concept d'appartenance. Il peut s'agir soit d'une contrainte sur des attributs d'instances, soit d'une contrainte sur des instances.

Chaque instance impliquée dans une contrainte entre instances devra être informée de la présence d'une telle contrainte. Un champ particulier devra donc être ajouté à l'ensemble des attributs de l'entité utilisée pour décrire une instance, pour accueillir cette information.

6.2.5 Contrainte d'instance

Une contrainte d'instance est une contrainte qui décrit une relation à maintenir entre les attributs de cette instance ou des attributs atteignables à partir de cette instance.

Pour une instance de concept, les contraintes de ce type viennent s'ajouter, d'une part à l'ensemble des contraintes de concept et, d'autre part, aux contraintes des classes auxquelles elle est attachée dans chaque point de vue du concept.

De même que dans le cas de contraintes entre instances, les contraintes d'instance seront déclarées au niveau de l'instance. Elles seront rattachées à l'instance correspondante. Un champ particulier devra donc être ajouté à l'ensemble des attributs de l'entité utilisée pour décrire une instance, pour accueillir cette information.

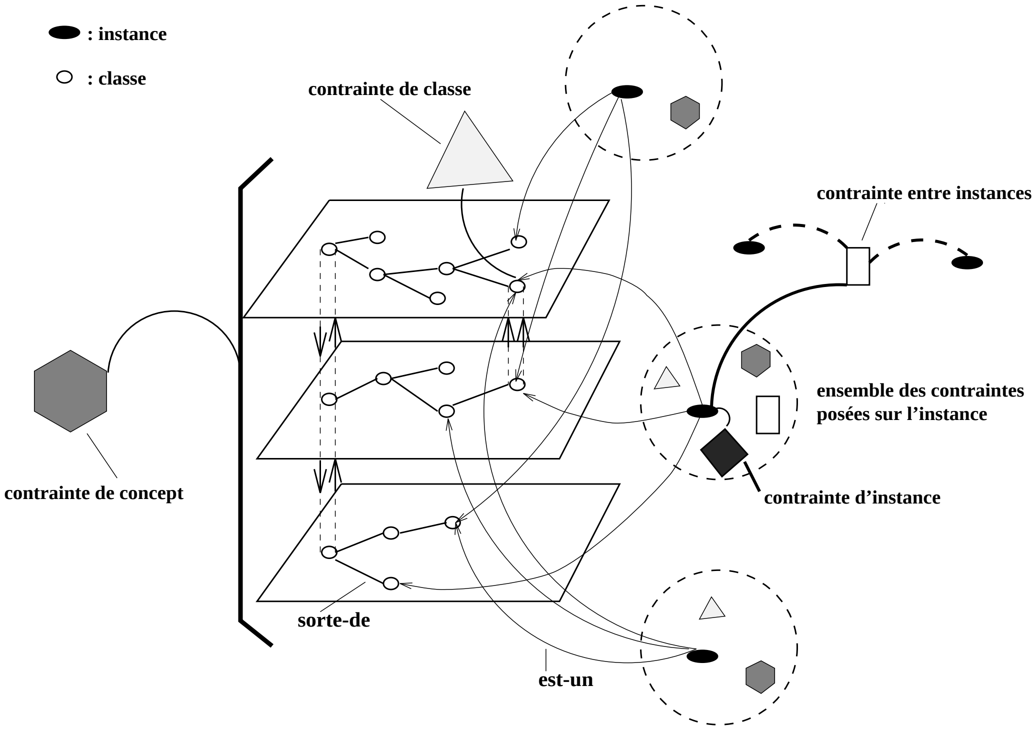


FIG. 6.2 - : Les différents niveaux de représentation (concept, classe, instance(s)) auxquels une contrainte peut être déclarée. Une contrainte de concept (respectivement de classe) est attachée à chacune des instances de ce concept (respectivement de cette classe)

6.2.6 Différents types de CSP indépendants

En considérant les diverses entités de représentation du modèle TROPES, nous venons d'exhiber quatre niveaux de déclaration de contraintes (concept, classe, entre instances, instance) (*cf.* figure 6.2). Pour chacun de ces niveaux, nous avons décidé d'un comportement, d'une sémantique à associer à la déclaration de la contrainte. Il est donc intéressant de remarquer qu'il existe quatre types distincts de réseaux de contraintes définissables en TROPES. En réalité, à un moment donné, si l'on considère un attribut d'une instance de TROPES, les contraintes qui portent sur lui ne sont pas distinguées du point de vue de la consistance de son domaine de valeurs possibles. Celui-ci est effectivement le domaine résultant de l'application des diverses contraintes (de concept, de classe, entre instances, d'instance) qui portent sur cet attribut. Le réseau effectif dans lequel il est plongé en tant que variable contrainte est donc le réseau formé indistinctement des quatre niveaux de déclaration de contraintes présentés ci-dessus.

6.3 La notion d'accès

Les variables des contraintes de TROPES sont des attributs d'objets (ou des objets dans le cas de contraintes composées). Or, il existe dans TROPES des liens entre objets qui sont modélisés par des attributs dont la valeur est une instance, un ensemble ou une liste d'instances. Aussi, nous trouvons là le moyen de contraindre, à partir d'un objet, un autre objet ou un attribut d'un autre objet. Il reste alors à désigner cet objet ou cet attribut dans une expression contrainte.

La désignation d'un attribut à contraindre peut être immédiate (lorsque l'attribut à contraindre est un attribut de l'objet auquel appartient la contrainte) ou indirecte (lorsque l'attribut à contraindre est atteignable – via une ou plusieurs indirections – de l'objet auquel appartient la contrainte).

Cette section vise à définir un moyen d'accéder à un ou plusieurs attributs à partir d'un objet pour le contraindre : la notion d'*accès*. Cette notion étend la notion de *chemin* rencontrée dans d'autres associations objets/contraintes [Borning81, Kökény94].

6.3.1 Définition d'un accès

Afin de désigner un attribut (à contraindre) dans TROPES, la notion d'accès (*cf.* figure 6.3) est introduite. Un accès permet, à partir d'un objet dit *source*, d'accéder, en parcourant éventuellement un ou plusieurs attributs dits *étapes*, un attribut dit *destination*.

Cette notion d'accès est introduite afin d'être utilisée comme dans l'expression d'une contrainte afin de désigner le ou les attributs à contraindre. Ainsi, un accès peut être vu comme une fonction qui délivre un ou plusieurs (liste ou ensemble) attributs.

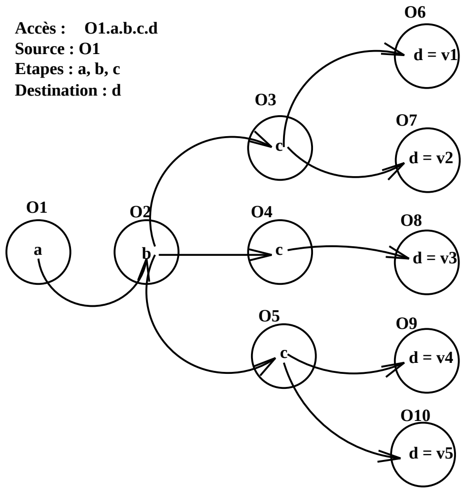


FIG. 6.3 - : Exemple d'accès : l'accès *O1.a.b.c.d.* Il y a diffraction au niveau des attributs étapes *b* et *c* qui sont des attributs multivalués.

Lorsque la source d'un accès n'est pas précisée, on considérera qu'il s'agit de l'objet courant dans lequel est utilisé cet accès. Par définition, un attribut étape est un attribut-lien. Sa valeur est donc une instance ou une liste ou un ensemble d'instances. Quant à l'attribut destination, il n'existe pas de restriction sur sa nature.

Cette notion d'accès n'est pas nouvelle. Elle fut sans doute utilisée pour la première fois dans le système THINGLAB sous le nom de *path* ou *chemin*. Mais les chemins définis dans THINGLAB sont de longueur 1 : ils sont donc équivalents à des accès ne comportant pas d'étapes mais simplement une destination. La notion de chemin a été étendue par Kökény dont le système gère les chemins de longueur supérieure à 1. Cependant, Kökény ne traite pas le cas où les attributs étapes ou/et destination sont des attributs multivalués. Ici, la notion d'accès est donc originale en ce sens qu'elle prend en compte le fait qu'un attribut (étape ou destination) puisse être multivalué.

Dans le cas d'un accès ne comportant que des attributs monovalués alors il n'existe qu'un seul chemin menant de l'objet source à l'objet destination.

Dans le cas d'un accès présentant au moins un attribut étape multivalué, on peut considérer qu'il y a *diffraction* en ce nœud étape du chemin suivi jusqu'alors depuis l'objet source : plusieurs chemins sont possibles pour atteindre les attributs destinations, autant que d'objets figurant dans la liste ou dans l'ensemble valeur de cet attribut étape.

6.3.2 Valeur d'un accès

La valeur d'un accès est entièrement définie lorsque tous les attributs étapes et l'attribut destination qui le composent possèdent eux-mêmes une valeur.

Dans les cas particuliers où l'accès ne comporte que des étapes monovaluées (un seul chemin mène à la destination) ou bien aucune étape (seulement une source et une destination), la valeur de l'accès est équivalente à la valeur de la destination. Plus généralement, si l'attribut destination n'a pas de valeur, ou si l'un des attributs étapes n'a pas de valeur, alors l'accès n'a pas de valeur non plus. Lorsque tous les attributs d'un accès (étapes et destination) sont valués mais que l'une des étapes au moins est multivaluée, la valeur de l'accès doit prendre en compte cette diffraction.

6.3.3 Interprétation d'un accès

L'interprétation d'un accès est l'opération qui consiste, à partir de l'expression d'un accès, à déterminer les attributs qu'il permet d'atteindre afin de calculer sa valeur comme l'ensemble ou la liste ou l'unique valeur de ou des attributs atteints.

L'interprétation qui permet de déterminer la valeur d'un accès est immédiate lorsqu'il ne contient aucune diffraction (aucun attribut étape multivalué). En revanche, lorsqu'il existe au moins deux diffractions, ou une diffraction et une destination multivaluée, l'interprétation de l'accès devient arbitraire, comme le montre l'exemple suivant :

Exemple 8 *Considérons l'accès de source l'objet S, constitué d'une étape e de type ensemble et d'une destination d également de type ensemble. e et d sont dont de type multivalué.*
Le parcours d'un accès s'effectue depuis la source, étape par étape, jusqu'à la destination. Ici, la diffraction s'effectue sur l'unique étape. Supposons que l'étape e ait pour valeur l'ensemble $\{O_1, O_2, O_3\}$*. Il a donc trois chemins à suivre à partir de e. Supposons que la destination d dans* O_1 *ait pour valeur l'ensemble* $\{v_{11}, v_{12}\}$*, que la destination d dans* O_2 *ait pour valeur l'ensemble* $\{v_{21}\}$*, que la destination d dans* O_3 *ait pour valeur l'ensemble* $\{v_{31}, v_{32}, v_{33}\}$*.*
La valeur finale de l'accès est-elle l'ensemble $\{v_{11}, v_{12}, v_{21}, v_{31}, v_{32}, v_{33}\}$ *des 6 valeurs atteintes ou bien l'ensemble* $\{\{v_{11}, v_{12}\}, \{v_{21}\}, \{v_{31}, v_{32}, v_{33}\}\}$ *des 3 ensembles de valeurs atteints par les 3 chemins présentés par la diffraction ?*

Choisir une interprétation au détriment d'une autre, c'est priver l'accès d'une certaine puissance d'expression. Aussi, nous prenons le parti de conserver ces deux interprétations sous le nom de *réduction* et d'*anti-réduction*. C'est l'expression même de l'accès qui devra déterminer, à chaque étape, quelle interprétation employer. La reconnaissance d'une interprétation sera possible grâce aux connecteurs d'étapes qui constitueront l'expression d'un accès :

- le connecteur d'étapes ≪.≫ correspondra à une réduction
- le connecteur d'étapes ≪!≫ correspondra à une anti-réduction

6.3.3.1 La réduction

La réduction, notée $\odot$, est une fonction définie par :

$$\odot : (\mathcal{C} \times T) \times (\mathcal{C} \times T) \rightarrow (\cap C \times T)$$

où $\mathcal{C} = \{un, liste, ens - de\}$ est l'ensemble des constructeurs de types et T l'ensemble des types présents dans la base de connaissances. Le tableau de valeurs associé à la réduction est le tableau 6.4.

La réduction possède les propriétés suivantes :

- elle est associative à gauche. Par exemple, l'accès $a.b.c$ est interprété comme $(cons_a \odot cons_b) \odot cons_c$ où $cons_i$ est le constructeur associé à l'attribut i.
- le constructeur *un* est élément neutre. Ceci traduit le fait qu'il n'y a qu'un seul chemin possible qui prolonge l'accès.
- le constructeur *ens* est élément absorbant. Ceci traduit le fait que la réduction a pour but de casser la structure établie jusqu'alors. C'est une fonction de désordre.

(un,τ) $\odot$ (un,τ') $\mapsto$ (un,τ')
(un,τ) $\odot$ (ens,τ') $\mapsto$ (ens,τ')
(un,τ) $\odot$ (list,τ') $\mapsto$ (list,τ')
(ens,τ) $\odot$ (un,τ') $\mapsto$ (ens,τ')
(ens,τ) $\odot$ (ens,τ') $\mapsto$ (ens,τ')
(ens,τ) $\odot$ (list,τ') $\mapsto$ (ens,τ')
(list,τ) $\odot$ (un,τ') $\mapsto$ (list,τ')
(list,τ) $\odot$ (ens,τ') $\mapsto$ (ens,τ')
(list,τ) $\odot$ (list,τ') $\mapsto$ (list,τ')

TAB. 6.4 - : Table des réductions : on donne ici le résultat de l'application de la réduction à 2 couples (*constructeur, type*) sous la forme d'un couple (*constructeur, type*) pour chacun des constructeurs *un, ens, list* de TROPES.

(un,τ) $\otimes$ (un,τ') $\mapsto$ (un,τ')
(un,τ) $\otimes$ (ens,τ') $\mapsto$ (ens,τ')
(un,τ) $\otimes$ (list,τ') $\mapsto$ (list,τ')
(ens,τ) $\otimes$ (un,τ') $\mapsto$ (ens,τ')
(ens,τ) $\otimes$ (ens,τ') $\mapsto$ (ens,ens(τ'))
(ens,τ) $\otimes$ (list,τ') $\mapsto$ (ens,list(τ'))
(list,τ) $\otimes$ (un,τ') $\mapsto$ (list,τ')
(list,τ) $\otimes$ (ens,τ') $\mapsto$ (list,ens(τ'))
(list,τ) $\otimes$ (list,τ') $\mapsto$ (list,list(τ'))

TAB. 6.5 - : Table des anti-réductions : on donne ici le résultat de l'application de l'anti-réduction à 2 couples (*constructeur, type*) sous la forme d'un couple (*constructeur, type*) pour chacun des constructeurs *un, ens, list* de TROPES.

6.3.3.2 L'anti-réduction

L'anti-réduction, notée $\otimes$, est une fonction définie par :

$$\otimes : (\mathcal{C} \times T) \times (\mathcal{C} \times T) \rightarrow (\cap C \times T)$$

où $\mathcal{C} = \{un, liste, ens - de\}$ est l'ensemble des constructeurs de types et T l'ensemble des types présents dans la base de connaissances. Le tableau de valeurs associé à l'anti-réduction est le tableau 6.5. L'anti-réduction possède les propriétés suivantes :

- elle est associative à droite. Par exemple, l'accès $a!b!c$ est interprété comme $cons_a \otimes (cons_b \otimes cons_c)$ où $cons_i$ constructeur associé à l'attribut i.
- le constructeur *un* est élément neutre. Ceci traduit le fait qu'il n'y a qu'un seul chemin possible qui prolonge l'accès.
- le constructeur *ens* n'est pas élément absorbant, contrairement à la réduction. Ceci traduit le fait que l'anti-réduction a pour but de conserver la structure établie jusqu'alors. C'est une fonction d'ordre.

Enfin, la réduction est prioritaire sur la non-réduction. Par exemple, l'accès $a!b.c$ est interprété comme $cons_a \otimes (cons_b \odot cons_c)$ où $cons_i$ constructeur associé à l'attribut i.

6.3.4 Exemples d'accès

Soient *père, mère, frères, amis, épouse, sœurs, âge* des attributs d'un concept P . Les attributs multivalués *frères, amis, sœurs* sont supposés avoir pour valeur des ensembles d'objets. On donne des exemples d'accès contenant ces attributs dans le tableau 6.6.

Le tableau 6.7 considère que les frères sont ordonnés (sous forme de liste).

ACCÈS	DÉSIGNATION	INTERPRÉTATION
père.mère.âge	âge de la mère du père	un(P)$\odot$ un(P)$\odot$ un($\mathbb{N}$) $\rightarrow$ un($\mathbb{N}$)
père.mère.frères	ensemble des frères de la mère du père	un(P)$\odot$ un(P)$\odot$ ens(P) $\rightarrow$ ens(P)
frères.amis.âge	ensemble des âges des amis des frères	ens(P)$\odot$ ens(P)$\odot$ un($\mathbb{N}$) $\rightarrow$ ens($\mathbb{N}$)
frères.amis	ensemble des amis des frères	ens(P)$\odot$ ens(P) $\rightarrow$ ens(P)
père.frères.âge	âge des frères du père	un(P)$\odot$ ens(P)$\odot$ un($\mathbb{N}$)$\rightarrow$ ens($\mathbb{N}$)
père.amis.mère.amis	ensemble des amis de la mère des amis du père	un(P)$\odot$ ens(P)$\odot$ un(P)$\odot$ ens(P) $\rightarrow$ ens(P)

TAB. 6.6 - : À gauche, on trouve l'expression de l'accès. Au centre, sa signification (ce que l'on cherche à atteindre). À droite, l'interprétation de l'accès (application d'une réduction $\odot$ ou d'une anti-réduction $\otimes$).

ACCÈS	DÉSIGNATION	INTERPRÉTATION
père.mère.frères	liste des frères de la mère du père	un(P)$\odot$ un(P)$\odot$ list(P) $\rightarrow$ list(P)
frères.amis.âge	ensemble des âge des amis des frères	list(P)$\odot$ ens(P)$\odot$ un($\mathbb{N}$) $\rightarrow$ ens($\mathbb{N}$)
frères!amis.âge	liste des âges des amis de chacun des frères	list(P)$\otimes$ens(P)$\odot$ un($\mathbb{N}$) $\rightarrow$ list(ens($\mathbb{N}$))
père.frères.âge	liste des âges des frères du père	un(P)$\odot$ list(P)$\odot$ un($\mathbb{N}$)$\rightarrow$ list($\mathbb{N}$)
père.frères.épouse.sœurs	ensemble des sœurs des épouses des frères du père	un(P)$\odot$ list(P)$\odot$ un(P)$\odot$ ens(P) $\rightarrow$ ens(P)
père.frères.épouse!sœurs	la liste des sœurs des épouses de chacun des frères du père	un(P)$\odot$ list(P)$\odot$ un(P)$\otimes$ens(P) $\rightarrow$ list(ens(P))

TAB. 6.7 - : À gauche, on trouve l'expression de l'accès. Au centre, sa signification (ce que l'on cherche à atteindre). À droite, l'interprétation de l'accès (application d'une réduction $\odot$ ou d'une anti-réduction $\otimes$).

6.3.5 Pose de contraintes et accès non défini

Lors d'une pose de contrainte, la validité de l'expression définissant l'accès sera testée (en s'assurant que les concepts (objets) atteints lors d'étapes contiennent bien les attributs désignés par l'accès). Lorsqu'une valeur d'étape n'est pas définie, la destination n'est pas connue. Cependant, la contrainte concerne l'objet qui la contient et sera rattachée à celui-ci. Un accès valide se comporte comme un accès valide ne comportant qu'une destination. Le fait qu'il ait une valeur ou pas, a seulement une conséquence sur la quantité d'information qu'est en mesure de vérifier la contrainte.

Toute étape doit avoir pour information le chemin dans laquelle elle est impliquée et la contrainte à poser. Ainsi, au niveau de chaque étape, on est en mesure de calculer la valeur de l'accès et de transmettre la contrainte à la prochaine étape. Si l'étape est la dernière, la contrainte est posée. Si la contrainte existe déjà (cas de la modification de la valeur d'une étape), ces informations permettent de se rendre aux attributs destinations (*cf.* section 10.7). La modification de la valeur d'une étape est acceptée lorsque l'accès n'est pas défini.

6.4 Représentation des contraintes

Après avoir décrit les composants des CSP TROPES, associé une portée à la présence d'une contrainte selon le niveau de représentation auquel elle est attachée, puis introduit la notion d'accès afin de désigner un attribut contraint, il reste à définir le mode de représentation des contraintes dans TROPES.

La représentation des contraintes dans un modèle de représentation par objets tel que TROPES peut se faire par deux moyens distincts (comme l'a montré le chapitre 5):

- Par intégration forte: les contraintes sont des connaissances exprimées grâces aux entités de représentation du modèle. Autrement dit, ce sont des objets à part entière décrits par des concepts et, au besoin, des points de vue, des classes et des instances. Une base de connaissances leur est donc dédiée.
- Par intégration faible : les contraintes ne sont pas représentées par des objets TROPES. Leur description est donc extérieure au modèle qui conserve néanmoins au sein des objets une trace de leur présence.

6.4.1 Les contraintes comme objets du modèle

La première solution a l'avantage de l'homogénéité puisque les contraintes sont des objets et que, en conséquence, leur création, activation et suppression peuvent s'effectuer à l'aide des primitives générales de manipulation d'objets.

Dans cette perspective, une contrainte est représentée par une classe contenant les descriptions des paramètres (les variables contraintes) mais aussi des méthodes à invoquer lors des phases de maintenance. Un concept CONTRAINTES (*cf.* figure 6.4) contient toutes les classes chargées de la description des contraintes de concept, des contraintes de classes, d'instances. Cette approche tout objet peut d'ailleurs être prolongée jusqu'aux méthodes de maintenance dont l'activation est déclenchée par instanciation de la classe équivalente.

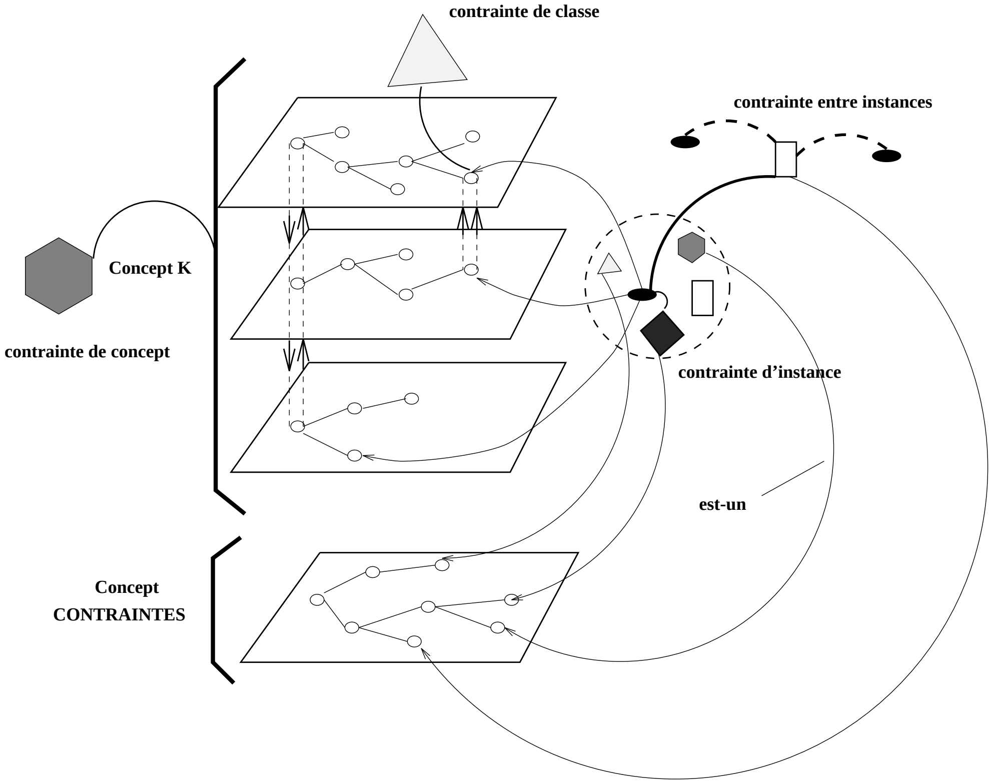


FIG. 6.4 - : Représentation des contraintes en tant qu'objets TROPES. Cette représentation a été abandonnée.

Si les contraintes sont décrites par des classes, la pose d'une contrainte correspond à l'instanciation. L'instance est une trace de la présence d'une contrainte. Il existe alors deux possibilités pour le contenu de ces instances de contraintes :

- les attributs de ces instances pointent – à l'aide d'accès, par exemple – sur les attributs de la base de connaissances sur lesquels la contrainte porte. On peut donc ici contraindre un attribut de contrainte ;

- les attributs de ces instances contiennent simplement des valeurs correspondant aux valeurs des attributs sur lesquels porte la contrainte et la liaison attributs contraints/attribut d'instance de contrainte se fait ailleurs...

Dans les deux cas, un attribut contraint de la base de connaissances doit connaître l'attribut d'instance de contrainte qui pointe sur lui ou qui contient sa valeur. Chaque attribut contraint doit posséder la liste des contraintes qui portent sur lui (information indispensable pour la liste de contraintes à construire lors de la propagation) et chaque contrainte doit contenir la liste des attributs qu'elle contraint. Ce qui est immédiat dans la première solution (donné par les valeurs des attributs) mais qu'il faut mettre en place dans la seconde solution (il faut enregistrer ce pointeur au niveau des attributs contraints)

Dans le premier cas, modifier la valeur d'un attribut d'une instance de contrainte revient à changer l'accès, et donc à contraindre un (ou plusieurs) autre(s) attribut(s) de la base. La modification d'une instance de contrainte revient donc à faire porter la même contrainte mais sur d'autres attributs. À la modification d'un attribut d'instance de contrainte, on doit donc associer un traitement particulier qui consiste à rompre le lien et à libérer les anciens attributs accédés de la contrainte, puis à construire le nouveau lien. L'homogénéité est ici rompue puisqu'il nous faut associer un traitement particulier à un attribut particulier.

Dans le second cas, il faut mettre en place un lien entre un attribut contraint et l'attribut d'instance de contrainte qui contient sa valeur. La modification d'une instance de contrainte revient à donner une autre valeur à un attribut d'instance de contrainte donc à changer par le biais d'une instance de contrainte, la valeur d'un attribut de TROPES. Cette solution présente tout d'abord un défaut de redondance dans l'enregistrement des valeurs et autorise la modification de la valeur d'un attribut contraint par une manière détournée (l'attribut d'instance de contrainte associé).

Pour prévenir ce genre de désagréments, il faut donc distinguer le concept CONTRAINTES en lui associant des primitives de manipulation particulières (*cf.* première solution), soit lui associer des droits d'accès afin d'éviter des effets de bord impromptus (*cf.* deuxième solution).

À ces deux cas problématiques s'ajoute un problème qui est de l'ordre de la sémantique de certaines contraintes exprimées dans le modèle. Par exemple, une contrainte de classe est identique sur chacune des instances de la classe. En donnant la possibilité à l'utilisateur de modifier une instance particulière d'une contrainte de classe – comme toute autre instance de la base –, on va à l'encontre de ce principe. Ce cas de figure peut se produire avec la première solution. De plus, si l'utilisateur est autorisé à supprimer une instance de contrainte de classe – il peut le faire avec les deux solutions – certaines instances de cette classe (ou de ce concept) peuvent donc être libérées. Pour éviter cela, une solution consiste encore à instaurer des droits d'accès sur certaines instances de contrainte de classe (ou de concept)...Encore-faut-il être en mesure de les reconnaître – ce qui peut être fait en passant par des concepts permettant de distinguer les différents niveaux de représentation contraignables – ou bien de les cacher à l'utilisateur...

Une intégration forte de contraintes dans un modèle de représentation de connaissances tel que TROPES entraîne des mesures discriminatoires à l'encontre des objets “contraintes” qui ne sont plus alors vraiment des objets à part entière du modèle de représentation. Dès lors une intégration faible, c'est-à-dire ne pas considérer les contraintes comme des objets présents dans la base de connaissances semble être une solution plus facile à mettre en œuvre. C'est aussi la solution choisie par le module de types. Elle a le mérite de bien distinguer ce qui est à représenter, de ce qui est livré avec le modèle et qui n'a pas à être inscrit dans les bases de connaissances.

6.4.2 Les contraintes hors du modèle

La seconde solution (intégration faible) a été choisie en vertu de la non adéquation de l'intégration forte. Une intégration faible signifie que les contraintes ne sont pas objets du modèle.

L'abandon d'une intégration forte a notamment pour conséquence que les contraintes ne sont pas définies ou supprimées par des mécanismes du modèle (comme l'instanciation et la suppression d'instances) mais par des fonctions de l'interface de programmation en TROPES : l'API (Application Programming Interface). Ceci n'est pas gênant dans la mesure où celle-ci est utilisée pour construire et manipuler des bases de connaissances : les contraintes seront construites et manipulées comme des entités du modèle, même si elles ne sont pas représentées à l'aide de celui-ci.

Les contraintes ne sont pas représentées en tant qu'objets TROPES (*cf.* figure 6.5), mais chaque objet (concept, classe ou instance) doit garder une trace des expressions de contraintes qui portent sur lui, afin que cette information soit disponible en accédant à cet objet. Le principe premier de la représentation par objets, selon lequel toute information relative à l'objet doit être accessible depuis celui-ci, est ainsi respecté.

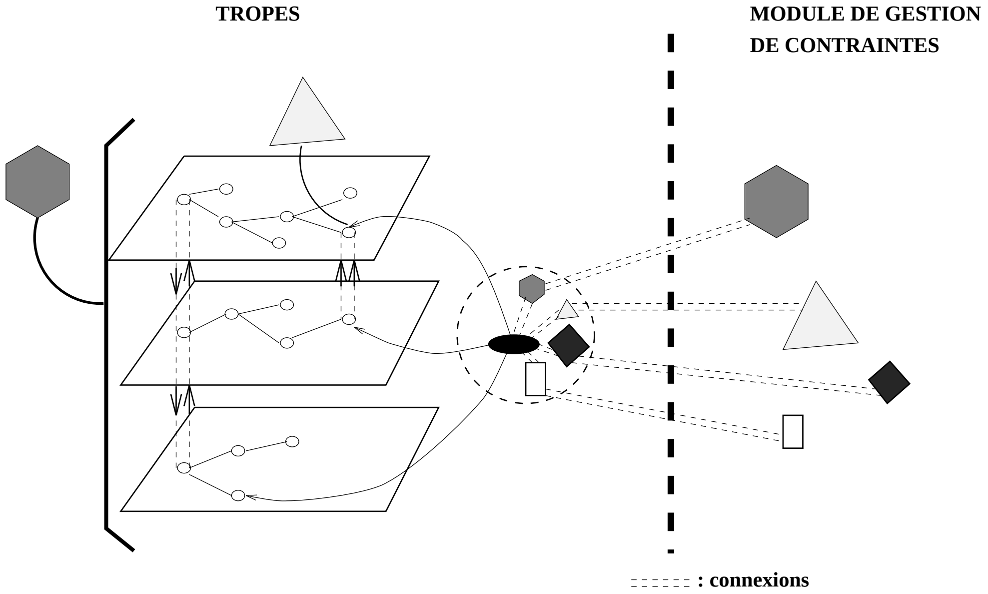


FIG. 6.5 - : La représentation choisie : les contraintes sont exprimées dans les objets ou à l'aide d'une interface de programmation d'application (API). La représentation de toute contrainte TROPES est déléguée à un module de gestion de contraintes. En contrepartie, il faut assurer l'échange d'informations entre les contraintes gérées par ce module et les objets contraints de TROPES.

La représentation des contraintes est donc externe à TROPES, elle concerne la bibliothèque ou le module de programmation par contraintes chargé de répondre aux spécifications des CSP TROPES. Ce module sera chargé de maintenir les CSP définis à l'aide des fonctions de l'API.

Les fonctions de l'API de manipulation des contraintes seront donc basées sur les fonctions de manipulation (définition, suppression, résolution) de contraintes offertes par ce module. Pour chacune des contraintes mises à la disposition de TROPES, ce module devra conserver les règles de maintenance de la consistance associées. Les points de connexion entre TROPES et ce module se situent au niveau des attributs contraints qui doivent être reliés aux variables contraintes équivalentes. Dès lors, les échanges d'informations (valeurs, domaines) seront possibles entre TROPES et ce module présenté au chapitre 8.

6.5 Gestion des CSP Tropes

La gestion des contraintes constitue l'aspect algorithmique de cette intégration. Il s'agit de déterminer quelles sont les caractéristiques techniques attendues du module de programmation par contraintes qui sera couplé à TROPES.

En vertu du mode d'utilisation et du maintien de la cohérence d'une base de connaissances, quatre tâches essentielles incombent au module chargé de la gestion des CSP TROPES :

1. Dès la définition d'une contrainte, un niveau de consistance suffisant doit être appliqué sur

les domaines des objets contraints afin de garantir, lorsque cela est possible, que ceux-ci ne contiennent pas de valeur localement inconsistante. Le module devra donc être en mesure de filtrer les domaines (nous avons vu au chapitre 4 que les degrés de consistance sont dépendants de la nature des domaines) et de propager dans les réseaux de contraintes établis toute modification susceptible d'avoir des répercussions. La propagation de contraintes est donc la première tâche à accomplir pour le module de gestion de contraintes.

Il est important que les réductions opérées par ce filtrage soient conservées dans la base de connaissances afin de disposer d'une image consistante des attributs contraints. Autrement dit, le domaine réduit d'un attribut contraint doit remplacer l'ancien domaine. Nous reviendrons sur les conséquences de la présence de contraintes sur le typage des attributs au chapitre suivant.

De plus, lorsque le domaine d'un objet contraint est réduit, par ce filtrage, à une seule valeur, alors on peut considérer que cette valeur est *effectivement* la valeur de l'objet contraint puisqu'elle est la seule admissible. Dans ce cas précis, il y a *inférence* de valeur, la contrainte est productive.

2. Toute modification (ou donnée) de la valeur d'un objet contraint doit être propagée aux objets concernés par cette modification. Là encore, la propagation de contraintes opérera le filtrage permettant de valider (localement) ou de refuser la valeur proposée. Concernant les attributs étapes d'accès, leur modification induit un changement de la valeur de l'accès. Cette particularité doit également être gérée par le couplage (*cf.* section 10.7).
3. L'ajout et le retrait d'une contrainte doivent être possibles à tout moment. Dans un modèle où la modification d'une valeur est déjà possible à tout moment et qui s'oriente vers le dynamisme des classes et des attributs, conférer la propriété de dynamicité (*cf.* section 4.6) aux CSP définis œuvre dans le sens de l'uniformité des opérations de manipulation des entités. Deux approches – l'une à base de justifications de types RMS, l'autre sans – ont été proposées à ce jour pour gérer des CSP dynamiques ; elles fournissent déjà une idée des structures à mettre en place. Pour tout ajout, il faut assurer qu'il est possible, c'est-à-dire que la phase de filtrage n'a vidé aucun domaine. Pour le retrait de contrainte, il faut assurer que la base de connaissances est libérée de toute trace de la présence de la contrainte. Le module de gestion de contraintes doit donc être adapté aux CSP dynamiques. Nous décrivons les conséquences de l'ajout et du retrait d'une contrainte au chapitre suivant.
4. La recherche des solutions du CSP doit être possible. Cette tâche est généralement l'objectif premier des langages de programmation par contraintes. Cependant, si elle est possible sur des CSP à domaines finis discrets, car reposant sur une énumération, nous avons vu qu'elle nécessite la mise en œuvre d'autres techniques pour les CSP numériques à domaines infinis. Or, ces deux types de CSP peuvent être définis sur des attributs de TROPES. Le module de gestion de contraintes doit donc notamment être en mesure de traiter des CSP numériques à intervalles, un intervalle permettant l'expression de ces deux types de domaines.

   Du fait même de son indécidabilité potentielle (rappelons que la propagation de contraintes sur intervalles peut boucler), la résolution de contraintes dans TROPES devient une tâche secondaire. Elle n'est qu'un prolongement possible de l'étape de maintien de consistance opérée à la définition d'une contrainte. La tâche principale dans la gestion des CSP TROPES est donc la maintenance qui se traduit par la recherche de la consistance locale des domaines des objets contraints. La résolution des contraintes s'inscrit dans un souci de rendre explicites des faits énoncés implicitement par les contraintes exprimées. Elle peut donc être vue comme l'un des mécanismes de complétion à solliciter lors d'un raisonnement prospectif ou hypothétique. En raison même de la possible infinitude des domaines manipulés, cette résolution de contraintes doit pouvoir être contrôlée. Par exemple, il n'est pas raisonnable d'exhiber les solutions de la seule contrainte $a < b$ pour a et b entiers de domaines $]-\infty+\infty[$ ...Aussi, le module de gestion de contraintes doit être en mesure de résoudre les CSP à domaines finis (de tous types) et de proposer pour les CSP numériques à domaines infinis une méthode acceptable de recherche de solutions.

6.6 Conclusion

L'introduction de contraintes dans TROPES doit permettre d'exprimer et de maintenir des propriétés sur un attribut et des relations entre plusieurs attributs. L'emploi de techniques de la programmation par contraintes (arc-consistance, propagation, résolution) garantit la consistance locale de ces expressions. Cette maintenance de contraintes peut également conduire à l'inférence de valeurs. Les attributs TROPES sont les seules variables contraintes effectives des CSP TROPES. L'éventuelle infinitude de leur domaine nous amène à faire, pour cette intégration, le choix des principes de la programmation par contraintes sur des intervalles. De ce fait, la résolution des contraintes doit être paramétrable afin de réduire la taille des domaines à explorer et rendre possible la recherche d'une solution.

Un éventail de contraintes a été proposé en tenant compte des types des attributs contraints. L'utilisateur doit disposer de contraintes arithmétiques ou booléennes classiques pour les attributs monovalués de type *nombre* ou *booléen*. En raison de la présence des attributs multivalués, des contraintes sur des listes ou des ensembles de valeurs sont également indispensables. Des contraintes plus spécifiques, qui tiennent compte de l'aspect dynamique des bases de connaissances, sont également envisageables (contraintes conditionnelles, contraintes de priorités, contraintes disjonctives, contraintes d'évolution et contraintes globales).

Plutôt que de fournir un langage de définition de contraintes, le choix d'un ensemble complet de contraintes de base a été fait. Celles-ci pourront être assemblées en des contraintes complexes, transformables à leur tour en contraintes de base. Ces contraintes de base doivent être suffisamment génériques pour s'adapter à la définition de nouveaux types gérés par le module de types METÉO.

N'importe laquelle de ces contraintes peut devenir une contrainte de concept, une contrainte de classe, une contrainte d'instance ou une contrainte entre instances. À chaque niveau de représentation est associé un comportement de la contrainte.

La désignation d'un attribut se fait grâce à un accès. L'originalité de cette approche réside dans les deux interprétations – réduction et anti-réduction – qui ont été associées à la notion d'accès et qui permettent d'accéder à un attribut contraint via plusieurs autres attributs éventuellement multivalués.

Si représenter les contraintes en tant qu'objets du modèle permet de manipuler les contraintes à l'aide des primitives du modèle, la sémantique associée aux contraintes, selon le niveau de représentation où elles apparaissent, implique de sérieuses restrictions qui conduisent à l'abandon de ce choix. L'intégration des contraintes réalisée est donc faible, les contraintes sont représentées hors du modèle, dans un module de gestion de contraintes chargé de la maintenance des CSP TROPES.

Enfin, nous avons pu établir un profil du module de gestion de contraintes souhaitable pour remplir le cahier des charges établi jusqu'alors. La propagation de contraintes doit en être le mécanisme de base qui permet d'obtenir un certain degré de consistance sur les domaines des CSP TROPES. Ce module doit également être adapté aux CSP dynamiques pour permettre une utilisation dynamique des contraintes dans TROPES qui soit en corrélation avec la gestion de l'évolution des bases de connaissances. Enfin, la résolution de CSP à domaines finis et de CSP numériques à domaines infinis sont des fonctionnalités que l'on est également en droit d'attendre d'un tel module.

Avant de définir ce module, nous étudions comment peut être gérée la dynamicité des contraintes dans TROPES, quelles sont les conséquences de l'ajout et du retrait de contraintes à la fois sur les types des objets concernés et sur les objets eux mêmes.

Chapitre 7

Contraintes, typage et dynamicité

TROPES est un modèle dynamique : à tout moment, il est possible de créer ou de supprimer une connaissance : un concept, un point de vue, une classe, une instance, un attribut ou une valeur.

Les contraintes permettent d'exprimer des propriétés sur des attributs ou des relations liant plusieurs attributs. Ce sont des éléments de connaissances greffées sur les entités (concept, classe, instance) à divers niveaux de représentation. Au même titre que les autres éléments de connaissances (descriptifs ou factuels) d'une base, les contraintes doivent pouvoir être ajoutées ou supprimées à tout moment. Dans les termes de la programmation par contraintes, cela signifie que les CSP TROPES sont dynamiques et qu'ils doivent être gérés en tant que tels.

Le chapitre précédent a montré que, dans TROPES, la tâche essentielle à réaliser pour le module chargé de la gestion des contraintes réside en l'établissement de la consistance locale des domaines contraints. Ceci peut être réalisé à l'aide d'un algorithme d'arc-consistance. Par ce filtrage, le domaine d'un attribut contraint peut être réduit lors de l'ajout d'une contrainte. À l'inverse, on peut s'attendre à ce que ce domaine retrouve, sinon sa taille initiale, une taille supérieure à sa taille courante, lors de la suppression d'une contrainte.

Or, les attributs TROPES sur lesquels sont ajoutées ou retirées ces contraintes sont typés. Un module de gestion de types, METÉO [Capponi95], stocke le type des attributs (des classes et des concepts, également) et, par là, une expression (normalisée) de leur domaine. Aussi, dans le cas d'une réduction (respectivement d'une extension) de domaine opérée par l'ajout (respectivement le retrait) d'une contrainte, le type de l'attribut contraint stocké par METÉO n'est donc plus à jour. Afin que le maintien de cohérence associé aux contraintes via l'établissement de la consistance locale soit répercuté dans la base de connaissances, il doit exister une coopération étroite entre METÉO et le module de gestion de contraintes chargé de la propagation de contraintes (donc, de l'éventuelle modification des domaines).

Ainsi, l'ajout et le retrait d'une contrainte donne lieu dans un premier temps à un calcul de types, puis, dans un second temps, à l'ajout ou au retrait effectif auprès des connaissances concernées dans la base.

Ce chapitre étudie comment doit être prise en compte la dynamicité des CSP. Dans la section 7.1, nous étudions les calculs de types induits par l'ajout et le retrait d'une contrainte selon son niveau de représentation (concept, classe ou instance). La section 7.2 est consacrée, dans un premier temps (section 7.2.1), à l'étude des répercussions de l'ajout et du retrait d'une contrainte sur les instances, puis, dans un second temps (section 7.2.2), à l'étude de l'impact sur les contraintes de la modification d'une valeur, d'un domaine ou d'une entité de représentation (concept, classe, attribut, instance).

7.1 Contraintes et types

7.1.1 Maintien de consistance et calcul de types

Assurer un certain niveau de consistance locale pour les domaines contraints consiste à éliminer des domaines des attributs contraints les valeurs qui ne pourront pas apparaître dans une solution. La section 6.2 a montré que les contraintes, selon le niveau de représentation auquel elles sont associées, possèdent une portée qui s'étend d'une simple instance (contrainte d'instance) à une extension complète (contrainte de concept, contrainte de classe), en passant par un ensemble quelconque d'instances (contraintes entre instances).

Le domaine d'un attribut contraint est rendu consistant vis-à-vis de l'ensemble de contraintes qui portent sur l'attribut. Dans la mesure où cet ensemble de contraintes varie, le domaine – donc le type[1] – 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 du retrait d'une contrainte). Nous étudions ici l'impact des contraintes sur les types des entités de représentation les supportant.

À chaque concept, chaque classe et chaque attribut dans un concept ou dans une classe, le module de gestion de types associe un type. Le type d'un concept (respectivement d'une classe) est le type record ([Cardelli84, Cardelli et al.91]) construit à partir des types des attributs de ce concept (respectivement de cette classe) (*cf.* chapitre 3). Ces types peuvent être considérés comme *initiaux*, car ils sont calculés à partir des définitions des attributs pour le concept ou pour la classe et ne tiennent pas compte de la présence de contraintes sur l'entité considérée. Nous en donnons à présent une formalisation.

Définissons tout d'abord les notions de type initial pour un attribut :

Définition 7.1 *Le type initial* $Tinit_{a_{iK}}$ *d'un attribut* a_{iK} *pour un concept* K *est le type décrit dans la définition de l'attribut* a_{iK} *pour le concept* K*.*

Définition 7.2 *Le type initial* $Tinit_{a_{jC}}$ *d'un attribut* a_{jC} *pour une classe* C *est le type décrit dans la définition de l'attribut* a_{jC} *pour la classe* C*. On a* $Tinit_{a_{jC}} \leq_T Tinit_{a_{iK}}$*, où* $\leq_T$ *est la relation de sous-typage associée au type* T*.*

Afin de définir le type initial d'un attribut pour une instance, nous définissons l'intersection de deux types.

Définition 7.3 *L'intersection de deux types* T *et* T'*, notée* $T \wedge T'$*, est le type* $T'' \in \mathcal{T} \cup \perp$ *tel que :*
si $T'' = \perp$ *alors* $\mathcal{D}(T'') = \emptyset$
sinon $\mathcal{D}(T'') = \mathcal{D}(T \wedge T') = \mathcal{D}(T) \cap \mathcal{D}(T')$
où $\mathcal{D}(T)$ *est la fonction qui à tout type associe son domaine de valeurs,* $\mathcal{T}$ *l'ensemble des types, et* $\cap$ *l'intersection ensembliste.*

Définition 7.4 *Le type initial* $Tinit_{a_{kI}}$ *d'un attribut* a_{kI} *pour une instance* I *du concept* K *est obtenu en faisant l'intersection des types initiaux de* a_{kI} *dans chacune des classes d'appartenance de l'instance, puis l'intersection avec le type initial de* a_{k_I} *pour le concept (afin d'obtenir le type si l'attribut ne figure dans aucune des classes d'appartenance).*

Il est défini par :

$$(\wedge_{i=1}^{p} Tinit_{a_{kCl}}) \wedge Tinit_{a_{kK}}$$

où p *est le nombre de points de vue du concept* K *et* Cl *la classe d'appartenance de* I *dans le point de vue* i *du concept* K*.*

Nous reprenons la notion de valeur record et de type record de Cardelli :

Définition 7.5 *Une valeur record de concept* K*, noté* $R_K =< a_{1K} = v_{a_{1K}}, a_{2K} = v_{a_{2K}}, \ldots, a_{nK} = v_{a_{nK}} >$*, est une association finie de valeurs à des étiquettes comportant* n *éléments (où* n *est le nombre d'attributs du concept) qui représente le contenu d'une instance*

[1] Rappelons que les types des attributs sont une représentation normalisée des domaines de valeurs de ces attributs.

Une étiquette a_{iK}*,* $\forall i \in [1, n]$*, est un nom d'attribut du concept K.*
Une valeur $v_{a_{iK}}$*,* $\forall i \in [1, n]$*, est la valeur de l'attribut* a_{iK} *du concept K.*

Nous exprimons à présent le type initial d'un concept et le type initial d'une classe à partir de la notion de type record :

Définition 7.6 *Le type initial d'un concept K comportant n attributs est le type record, noté* $Tinit_K$*, défini par :*
$< a_{1K} : Tinit_{a_{1K}}, a_{2K} : Tinit_{a_{2K}}, \ldots, a_{nK} : Tinit_{a_{nK}} >$
où $Tinit_{a_{iK}}$ *est le type initial pour le concept K de l'attribut* a_{iK}*.*
T_K *dénote l'ensemble potentiel de toutes les valeurs records de concept* R_K

Définition 7.7 *Le type initial d'une classe C comportant m attributs est le type record, noté* $Tinit_C$*, défini par :*
$< a_{1C} : Tinit_{a_{1C}}, a_{2C} : Tinit_{a_{2C}}, \ldots, a_{mC} : Tinit_{a_{mC}} >$
où $Tinit_{a_{jC}}$ *est le type initial pour la classe C de l'attribut* a_{jC}

Afin de définir le type initial d'une instance, nous adaptons dans la définition 7.8 l'intersection de types record de Cardelli.

Définition 7.8 *L'intersection de deux types record*
$< e_1 : T_1, e_2 : T_2, \ldots, e_m : T_m, e_{m+1} : T_{m+1}, \ldots, e_n : T_n >$ *et*
$< e_1 : T'_1, e_2 : T'_2, \ldots, e_m : T'_m >$*,*
notée $< e_1 : T_1, e_2 : T_2, \ldots, e_m : T_m, e_{m+1} : T_{m+1}, \ldots, e_n : T_n > \wedge < e_1 : T'_1, e_2 : T'_2, \ldots, e_m : T'_m >$
est le type record défini par $< e_1 : T_1 \wedge T'_1, e_2 : T_2 \wedge T'_2, \ldots, e_m : T_m \wedge T'm, e_{m+1} : T_{m+1}, \ldots, e_n : T_n >$
où $\wedge$ *est l'opérateur d'intersection des types*

Nous pouvons alors définir le type initial d'une instance :

Définition 7.9 *Le type initial d'une instance I du concept K est le type record, noté* $Tinit_I$*, défini par :*

$$(\wedge_{i=1}^{p} Tinit_{C_i}) \wedge Tinit_K$$

où p est le nombre de points de vue du concept K
C_i *est la classe d'appartenance de I dans le point de vue i du concept K*
et $\wedge$ *l'opérateur d'intersection définie sur les types records*

Un domaine contraint rendu consistant par la phase de maintenance de contraintes informe sur le type consistant de l'attribut. En présence ou en l'absence de contraintes, ce domaine consistant varie du domaine initial fourni par le type initial jusqu'au singleton formé d'une seule valeur du domaine initial. En théorie, le domaine d'un attribut contraint peut valoir n'importe quel ensemble de valeurs issu de la partition du domaine initial associé.

Afin de travailler avec des domaines et non plus avec des types, nous introduisons la notion de domaine record.

Définition 7.10 $< e_1 : D_{e_1}, e_2 : D_{e_2}, \ldots, e_n : D_{e_n} >$ *est le domaine record associé au type record* $< e_1 : T_{e_1}, e_2 : T_{e_2}, \ldots, e_n : T_{e_n} >$
avec $D_i = \mathcal{D}(T_i)$

En l'absence de contraintes, la validation d'une valeur proposée pour un attribut consiste, pour MÉTÉO, à s'assurer qu'elle figure bien dans le domaine fourni par le type (initial, ici) de l'attribut.

En présence de contraintes sur cet attribut, non seulement la valeur proposée doit se trouver dans le domaine consistant de l'attribut, mais, pour chaque contrainte, on doit également s'assurer que l'instanciation contenant la valeur de l'attribut et les valeurs des attributs attenants ne viole pas la contrainte.

Aussi, si le module de gestion de types ne conserve que les types initiaux, il peut entériner une valeur inconsistante (qui serait éliminée par le filtrage opéré par une contrainte) et ce n'est que lors de la phase de vérification de la contrainte que cette valeur sera écartée. En conservant le type courant d'un attribut contraint – dont le domaine associé est consistant – le module de gestion de types est en mesure de refuser immédiatement cette valeur.

C'est pourquoi tout domaine évalué par la phase de maintenance déclenchée à l'ajout ou au retrait d'une instance, sera conservé par le module de gestion de types. Ainsi, le domaine d'un type courant d'une entité contrainte (concept, classe, instance ou attribut) est un domaine consistant au regard de l'ensemble des contraintes (éventuellement vide) qui porte sur cette entité. Ce domaine a un degré de consistance établi par les diverses phases de maintenance déclenchées. Le type d'une entité reflète donc l'ensemble des contraintes qui l'impliquent, à partir du moment où le module de gestion de contraintes informe METÉO en cas de modifications sur les domaines des types des attributs contraints.

Afin que les types soient consistants avec les contraintes présentes, une collaboration étroite doit s'engager entre le module de gestion de types et le module de gestion de contraintes. Sans cette collaboration, le module de gestion de types conserve des informations non cohérentes avec l'état des connaissances ; son rôle en présence de contraintes serait alors limité. Le module de gestion de contraintes sera chargé du calcul de la restriction des types, le module de gestion de types de leur conservation (*cf.* figure 7.1). Le domaine réduit d'un type obtenu par le module de gestion de contraintes est passé à METÉO qui l'exprime alors sous la forme d'un nouveau type. Aussi, METÉO, grâce au module de gestion de contraintes, est capable de factoriser en une seule expression les facettes et les contraintes qui portent sur un attribut.

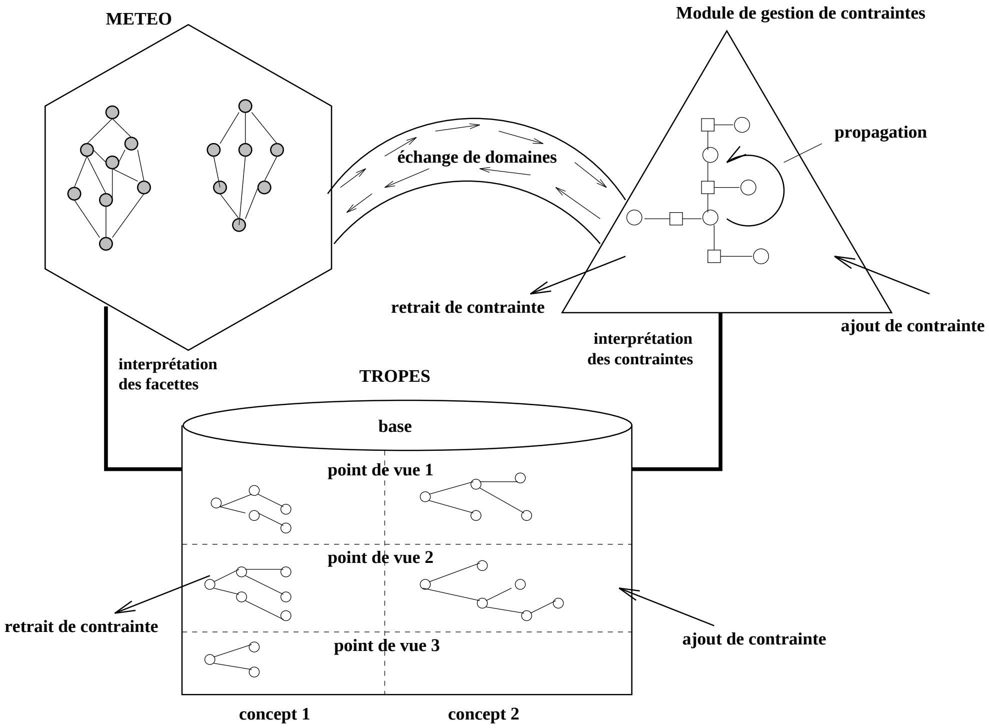


FIG. 7.1 - : Représentation de l'échange de domaines entre METÉO et le module de gestion de contraintes lors d'un ajout ou du retrait d'une contrainte.

Afin de formaliser l'effet d'une contrainte sur les domaines de ses attributs, nous adaptons à présent la notion de filtre proposée par Puget [Puget92b]

Définition 7.11 *Soit $r(x_1, x_2, \ldots, x_n)$ une contrainte du CSP (X, D, C) ($x_i \in X, \forall i \in [1, n], r \in C$). Le filtre associé à $r(x_1, \ldots, x_n)$ est une fonction, notée ϕ_r, qui, à tout domaine d_k de D de la variable x_k de X, associe le domaine d'_k tel que :*

$$\phi_r(d_k) = d'_k = \begin{cases} \textit{si } x_k \notin \{x_1, x_2, \ldots, x_n\} \textit{ alors} & d_k \\ \textit{sinon} & \{v_k \in d_k, \mid \forall j \in [1, k-1] \cup [k+1, n], \\ & \exists v_j \in d_j, (v_1, v_2, \ldots, v_{k-1}, v_k, v_{k+1}, \ldots, v_n) \\ & \textit{localement consistante}\} \end{cases}$$

Le filtre transforme le CSP (X,D,C) en un CSP (X,D',C) équivalent, avec $d'_k \in D'$.

Plusieurs auteurs [Mackworth77, Güsgen et al.88, Deville et al.91] ont montré que l'application d'une suite de filtres termine en un nombre fini d'étapes sur un CSP à domaines finis. Alors, un algorithme de propagation tel que celui défini au chapitre 4 résulte en une suite finie d'applications de filtres, notée ρ_r (c'est une composition de filtres), dont le premier est ϕ_r (propagation déclenchée par la contrainte r).

Notons que, pour les domaines infinis, une telle terminaison n'est pas garantie si ce n'est par les limites de la machine. Dans ce cas, qui se produit lors d'un ajout, la contrainte est refusée.

Nous sommes donc maintenant en mesure d'exprimer les types d'un concept, d'une classe et d'une instance lors de l'ajout et du retrait d'une contrainte de concept, de classe, d'instance ou entre instances.

7.1.2 Ajout de contrainte

7.1.2.1 Ajout d'une contrainte de concept

L'ajout d'une contrainte de concept r_K transforme le type d'un concept K $< a_{1K} : T_{a_{1K}}, \ldots, a_{nK} : T_{a_{nK}} >$ de domaine record associé $< a_{1K} : D_{a_{1K}}, \ldots, a_{nK} : D_{a_{nK}} >$ en $< a_{1K} : T'_{a_{1K}}, \ldots, a_{nK} : T'_{a_{nK}} >$ de domaine record associé $< a_{1K} : D'_{a_{1K}}, \ldots, a_{nK} : D'_{a_{nK}} >$ avec $D'a_{iK} = \rho_{r_K}(D_{a_{iK}}), \forall i \in [1, n]$

Les contraintes de concept s'appliquent sur des domaines issus de types prédéfinis (comme entier, réel) et n'ont généralement pas d'effet sur le type des attributs pour le concept.

Une contrainte de concept $r_K(a_{1K}, \ldots, a_{mK})$ doit être vérifiée par toute classe où apparaissent les attributs $a_{1K}, \ldots, a_{mK}$. La contrainte de concept agit donc comme une contrainte de classe.

L'ajout d'une contrainte de concept r_K transforme le type d'une classe C $< a_{1C} : T_{a_{1C}}, \ldots, a_{mC} : T_{a_{mC}} >$ de domaine record associé $< a_{1C} : D_{a_{1C}}, \ldots, a_{mC} : D_{a_{mC}} >$ en $< a_{1C} : T'_{a_{1C}}, \ldots, a_{mC} : T'_{a_{mC}} >$ de domaine record associé $< a_{1C} : D'_{a_{1C}}, \ldots, a_{mC} : D'_{a_{mC}} >$ avec $D'a_{iC} = \rho_{r_K}(D_{a_{iC}}), \forall i \in [1, m]$.

Il faut noter que le domaine d'un attribut impliqué dans la contrainte de concept, mais ne figurant pas dans la classe, est équivalent à son domaine pour le concept. Afin de détecter au plus tôt une incohérence, le calcul des types de classes est effectué à partir des feuilles de la hiérarchie de classes. En cas d'incohérence, la contrainte de concept est refusée.

Une contrainte de concept r_K doit être vérifiée par toute instance de ce concept. La contrainte de concept devient donc une contrainte d'instance.

L'ajout d'une contrainte de concept r_K transforme le type d'une instance I d'un concept K $< a_{1K} : T_{a_{1I}}, \ldots, a_{nK} : T_{a_{nI}} >$ de domaine record associé $< a_{1K} : D_{a_{1I}}, \ldots, a_{nK} : D_{a_{nI}} >$ en $< a_{1K} : T'_{a_{1I}}, \ldots, a_{nK} : T'_{a_{nI}} >$ de domaine record associé $< a_{1K} : D'_{a_{1I}}, \ldots, a_{nK} : D'_{a_{nI}} >$ avec $D'a_{iI} = \rho_{r_K}(D_{a_{iI}}), \forall i \in [1, n]$

Si le filtre associé à la contrainte de concept a pour effet de vider un des domaines du type d'un attribut pour l'instance, cela signifie que l'instance n'est pas viable (son type n'est pas défini). La contrainte de concept est alors refusée puisqu'elle empêche certaines configurations de rattachement.

7.1.2.2 Ajout d'une contrainte de classe

En vertu de la spécialisation de classes, toute caractéristique d'une classe apparaît intacte ou réduite dans ses sous-classes. S'agissant d'attributs, le sous-typage – correspondant à l'inclusion des extensions – est exigé lors d'une redéfinition dans une sous-classe. Si l'attribut n'est pas redéfini, alors sa définition sera récupérée en remontant la hiérarchie à la recherche de la première sur-classe où se trouve une définition de l'attribut. C'est le mécanisme d'*héritage* (*cf.* chapitre 2).

Une contrainte de classe est une autre forme de caractéristique d'une classe. Afin de se conformer à la relation de spécialisation sur laquelle est établie la hiérarchie de classes, on doit s'assurer que toute contrainte de classe est aussi contrainte de toutes ses sous-classes (directes ou indirectes). Si une contrainte pouvait ne pas apparaître dans une sous-classe, alors on aboutirait à une situation dans laquelle on ne pourrait garantir l'inclusion des extensions le long de la hiérarchie. Par exemple, une instance de sous-classe pourrait ne pas vérifier une contrainte de sa sur-classe et donc ne pas appartenir à sa sur-classe.

Dans TROPES, les notions de *concept* et de *point de vue* imposent aux classes-racines des concepts de ne pas être contraintes. C'est-à-dire que les définitions des attributs qu'elle définissent doivent être rigoureusement identiques aux définitions de ces attributs pour le concept ; ceci afin de garantir qu'une instance soit toujours rattachée, dans chaque concept, au moins à la classe-racine. C'est pourquoi toute instance d'un concept vérifie la description de chaque classe-racine des différents points de vue. Vis-à-vis de l'ajout de contraintes de classe, on peut donc dire que les contraintes du concept deviennent automatiquement contraintes de la classe-racine de chaque point de vue, mais qu'il n'est pas possible d'ajouter une contrainte de classe particulière à la classe-racine dans aucun point de vue. L'ajout d'une contrainte de classe concerne seulement les sous-classes d'une classe racine, afin de préserver le principe selon lequel une instance possède au moins une classe d'appartenance dans chaque point de vue.

Dans TROPES, l'inclusion des extensions de classes s'appuie sur le sous-typage des classes. Si une classe C est une sous-classe d'une classe C' alors le type de C est un sous type de C' (c'est une condition nécessaire). Ainsi, on garantit que l'extension de C est incluse dans l'extension de C'. Le sous-typage des classes est établi sur le sous-typage de chaque attribut.

L'introduction de contraintes dans une classe modifie son intension. Les contraintes d'une classe portent sur des attributs de cette classe ou des attributs atteignables par accès depuis cette classe. Assurer un certain niveau de consistance sur les domaines des attributs contraints de la classe consiste à filtrer dans les domaines les valeurs qui ne pourront pas apparaître dans une instance de la classe (simplement parce que, ne satisfaisant pas les contraintes de la classe, l'instance sera rejetée). Ce filtrage a un effet sur les domaines des attributs, donc sur leur type et donc sur le type de la classe. En calculant le type de la classe en présence de contraintes de classe, on cherche à décrire de manière intensionnelle une extension abstraite qui soit la plus proche et la moins coûteuse possible de l'extension réelle de la classe : l'ensemble de toutes les instances qui peuvent lui appartenir.

Puisque le type de la classe est établi notamment à partir des contraintes de classe et que l'on impose le sous-typage entre deux classes liées par le lien de spécialisation, l'héritage des contraintes, comme celui des attributs, doit être instauré le long des liens de spécialisation.

Une contrainte de classe doit porter sur toutes ses sous-classes. On assure de cette manière que les sous-classes ont pour type un sous-type du type de la classe. En effet, sans les contraintes, le module de gestion de types garantit la spécialisation des classes en vérifiant le sous-typage des types records de ces classes.

Soient deux classes C et C', telles que C' est sous-classe de C. Soient T_C et $T_{C'}$, les types de C et C', considérées sans contraintes. On a $T_{C'} \leq_T T_C$. Soient $\mathcal{C}_C$ et $\mathcal{C}_{C'}$, les ensembles de contraintes de C et C'. Soit $\rho_{\mathcal{C}_C}$, le filtre associé à $\mathcal{C}_C$.

On a $\rho_{\mathcal{C}_C}(\mathcal{D}(T_{C'})) \leq_T \rho_{\mathcal{C}_C}(\mathcal{D}(T_C))$, puisque le filtrage d'une contrainte sur les domaines de ces variables est une fonction monotone.

On a $\rho_{\mathcal{C}_C \cup \mathcal{C}_{C'}}(\mathcal{D}(T_{C'})) \leq_T \rho_{\mathcal{C}_{C'}}(\mathcal{D}(T_{C'}))$, puisque $\rho_{\mathcal{C}_C \cup \mathcal{C}_{C'}}(\mathcal{D}(T_{C'})) = \rho_{\mathcal{C}_C}(\rho_{\mathcal{C}_{C'}}(\mathcal{D}(T_{C'})))$

Donc on a $\rho_{\mathcal{C}_C \cup \mathcal{C}_{C'}}(\mathcal{D}(T_{C'})) \leq_T \rho_{\mathcal{C}_C}(\mathcal{D}(T_C))$

En présence de contraintes, afin de prolonger le sous-typage garant de la spécialisation de classes, on doit faire l'union de l'ensemble des contraintes attachées à la sous-classe avec l'ensemble des contraintes de sa sur-classe, qui, lui-même est défini par le même processus d'héritage.

L'ajout d'une contrainte de classe r_C transforme le type d'une classe C : $< a_{1C} : T_{a_{1C}}, \ldots, a_{mC} : T_{a_{mC}} >$ de domaine record associé $< a_{1C} : D_{a_{1C}}, \ldots, a_{mC} : D_{a_{mC}} >$ en $< a_{1C} : T'_{a_{1C}}, \ldots, a_{mC} : T'_{a_{mC}} >$ de domaine record associé $< a_{1C} : D'_{a_{1C}}, \ldots, a_{mC} : D'_{a_{mC}} >$ avec $D'a_{iC} = \rho_{r_C}(D_{a_{iC}}), \forall i \in [1, m]$.

Si le nouveau type de la classe n'est pas défini alors cela signifie qu'il existe une (sous-)classe dont l'extension ne peut contenir aucune instance qui satisfasse la contrainte. Plutôt que de recourir à l'élimination de cette branche morte de la hiérarchie, nous éliminons[2] la contrainte de la classe (et de ses sous-classes). En cas d'échec – la hiérarchie est sur-contrainte –, on revient à la situation initiale.

La contrainte est maintenue dans un premier temps si la propagation de contraintes déclenchée pour calculer le type de la classe s'arrête en ne vidant aucun domaine. Cela ne signifie pas pour autant qu'il existe effectivement des instances dans la classe capables de satisfaire l'ensemble des contraintes de classe. Simplement, en vertu du niveau de filtrage effectué, on peut espérer que de telles instances existent. Il faut ensuite tester si la contrainte peut être supportée par les sous-classes.

Le moyen le plus rapide pour se rendre compte de la viabilité de la descendance d'une classe après l'ajout d'une contrainte est de tester si les sous-classes les plus profondes (les feuilles de la hiérarchie) sont viables. Le calcul de type associé à la définition d'une contrainte doit donc se faire de *bas en haut* (*cf.* tableau 7.1) : depuis les sous-classes feuilles de la hiérarchie jusqu'à la classe de définition de la contrainte.

Si toutes les sous-classes feuilles sont viables alors toutes leurs sur-classes jusqu'à la classe de définition le sont aussi, puisque les feuilles de la hiérarchie présentent les plus petits types (domaines) à contraindre.

Si le filtrage d'une des sous-classes feuilles révèle une inconsistance, alors la classe feuille retrouve son type initial. Sinon, on procède au calcul des types des sur-classes, jusqu'à la classe de définition de la contrainte. À ce stade, la contrainte est intégrée dans la hiérarchie. Si elle présente une extension, il reste à poser la contrainte sur chacune des instances.

Il est à noter que la procédure *propage-ajout-contr-class* peut être interrompue au plus tôt par un exit et que si la condition :

$\{\mathcal{C}_c\} = \{\mathcal{C}_{c'}\} \wedge \forall a \in var(\mathcal{C}_c),\ type(a, c) = type(a, c')$

est respectée entre deux classes c et c', avec c' sous-classe de c, on peut se dispenser de propager la contrainte au niveau de c après l'avoir fait pour c'. Ici, on s'assure simplement que les attributs redéfinis dans c' ne sont pas dans le réseau de contraintes $\mathcal{C}_c$ ou que c' définit de nouveaux attributs non contraints.

Une contrainte de classe r_C doit être vérifiée par toute instance de cette classe. La contrainte de classe devient donc une contrainte d'instance.

L'ajout d'une contrainte de classe r_C transforme le type d'une instance I d'un concept K $< a_{1K} : T_{a_{1I}}, \ldots, a_{nK} : T_{a_{nI}} >$ de domaine record associé $< a_{1K} : D_{a_{1I}}, \ldots, a_{nK} : D_{a_{nI}} >$ en $< a_{1K} : T'_{a_{1I}}, \ldots, a_{nK} : T'_{a_{nI}} >$ de domaine record associé $< a_{1K} : D'_{a_{1I}}, \ldots, a_{nK} : D'_{a_{nI}} >$ avec $D'a_{iI} = \rho_{r_C}(D_{a_{iI}}), \forall i \in [1, n]$.

Si le type-record d'une instance calculé par MICRO présente un domaine vide, cela signifie que la contrainte de classe empêche certaines configurations de rattachement. Elle est alors refusée.

7.1.2.3 Ajout d'une contrainte d'instance

L'ajout d'une contrainte d'instance r_I transforme le type d'une instance I du concept K $< a_{1K} : T_{a_{1I}}, \ldots, a_{nK} : T_{a_{nI}} >$ de domaine record associé $< a_{1K} : D_{a_{1I}}, \ldots, a_{nK} : D_{a_{nI}} >$ en $< a_{1K} : T'_{a_{1I}}, \ldots, a_{nK} : T'_{a_{nI}} >$ de domaine record associé $< a_{1K} : D'_{a_{1I}}, a_{2K} : D'_{a_{2I}}, \ldots, a_{nK} : D'_{a_{nI}} >$

[2] Un système de versionnement peut permettre les deux solutions.

```
Procédure propage-ajout-contr-class(𝒞 : contrainte ; c : classe ; type-ok : booléen)
début
    si sous-classes(c) ≠ ∅ alors
        pour tout c' ∈ sous-classes(c) faire
            propage-ajout-contr-class(𝒞, c', type-ok)
        finpour
    finsi
    si type-ok alors
        type-ok ← calcul-type-ajout(𝒞, c)
    finsi
fin

Procédure calcul-type-ajout(𝒞 : contrainte, c : classe )
début
    propager-contrainte(𝒞, c)
    % la propagation se fait sur le réseau %
    % de contraintes, les domaines des attributs %
    % sont issus des types de ces attributs %
fin
```

TAB. 7.1 - : Algorithme de calcul des types des classes après ajout d'une contrainte de classe. La procédure de propagation *propager-contrainte* est semblable à celle donnée à la section 4.3.

avec $D'a_{iI} = \rho_{r_I}(D_{a_{iI}}), \forall i \in [1, n]$.

Si le type de l'instance ne peut être défini (un des domaines est vidé lors de la propagation de contraintes), alors la contrainte est refusée.

7.1.3 Retrait de contrainte

La suppression d'une contrainte doit tout d'abord permettre aux domaines de retrouver les valeurs éliminées par l'application des règles de consistance de la contrainte et par la propagation de contraintes qui a suivi. L'idée est de faire retrouver au réseau la configuration qu'il avait au moment où la contrainte a été posée. Nous appelons cette configuration le *contexte de pose*. Le contexte de pose d'une contrainte peut-être vue comme une photographie des domaines des attributs appartenant au réseau dans lequel est ajouté l'instance.

Une fois ce contexte de pose rétabli, on peut considérer que l'on se retrouve dans la situation où les contraintes antérieures à la contrainte supprimée sont prises en compte, il reste donc à reposer les contraintes postérieures à la contrainte supprimée.

Nous étudions à présent l'effet du retrait d'une contrainte selon son niveau de représentation.

7.1.3.1 Retrait d'une contrainte d'instance

Le retrait d'une contrainte d'instance r_{jI} transforme le type d'une instance I de concept K $< a_{1K} : T_{a_{1I}}, \ldots, a_{nK} : T_{a_{nI}} >$ de domaine record associé $< a_{1K} : D_{a_{1I}}, \ldots, a_{nK} : D_{a_{nI}} >$ en $< a_{1K} : T'_{a_{1I}}, \ldots, a_{nK} : T'_{a_{nI}} >$ de domaine record associé $< a_{1K} : D'_{a_{1I}}, \ldots, a_{nK} : D'_{a_{nI}} >$ avec $D'a_{iI} = \bigcirc_{k=j+1}^{r} \rho_{r_{kI}}(D''_{a_{iI}})$, $\forall i \in [1, n]$, avec $D''_{a_{iI}}$ domaine de l'étiquette a_{iK} du contexte de pose de r_{jI}, r nombre de contraintes posées sur l'instance (contraintes de classe et de concept incluses), et $\bigcirc$ opérateur de composition de filtres.

Autrement dit, le nouveau type de l'instance est calculé :

1. à partir du type de l'instance au moment de la pose de la contrainte,
2. en ajoutant les contraintes de concept, de classes et d'instances posées après r_{jI} sur I.

L'enregistrement d'un contexte de pose peut s'avérer coûteux en place mémoire. Néanmoins, il permet de gagner du temps lors de la suppression de contraintes jeunes (posées récemment).

7.1.3.2 Retrait d'une contrainte de classe

Le retrait d'une contrainte de classe r_{jC} transforme le type d'une classe C $< a_{1C} : T_{a_{1C}}, \ldots, a_{mC} : T_{a_{mC}} >$ de domaine record associé $< a_{1C} : D_{a_{1C}}, \ldots, a_{mC} : D_{a_{mC}} >$ en $< a_{1C} : T'_{a_{1C}}, \ldots, a_{mC} : T'_{a_{mC}} >$ de domaine record associé $< a_{1C} : D'_{a_{1C}}, \ldots, a_{mC} : D'_{a_{mC}} >$ avec $D'a_{iC} = \bigcirc_{k=j+1}^{r} \rho_{r_{kC}}(D''_{a_{iC}})$, $\forall i \in [1, m]$, avec $D''_{a_{iC}}$ domaine de l'étiquette a_{iC} du contexte de pose de r_{jC} , r nombre de contraintes de la classe (contraintes de concept incluses).

Autrement dit, le nouveau type de la classe est calculé :

1. à partir du type de la classe au moment de la pose de la contrainte,
2. en ajoutant les contraintes de classe et de concept posées après r_{jC} sur C.

En cas de suppression d'une contrainte de classe (ou relaxation), la viabilité de la descendance n'est pas mise en cause et on propose, au contraire, d'élargir potentiellement l'ensemble des instances viables.

La relaxation d'une contrainte de classe doit se faire à partir de la classe de définition de la contrainte. Une contrainte de classe devant être appliquée à toutes ses sous-classes, la suppression ne peut concerner qu'un sous-arbre de la hiérarchie dont la classe est racine.

Lors de la définition de la classe, la contrainte a été prise en compte pour calculer le type de la classe mais également les types de chacune de ses sous-classes. Aussi, il faut veiller à ce que le type de la classe soit relaxé, ainsi que le type de chacune des sous-classes.

La propagation du calcul des types de classes le long de la sous-hiérarchie dont la classe est racine s'effectue de *haut en bas* (*cf.* tableau 7.2). On relaxe d'abord le type de la classe puis celui de ses sous-classes, des sous-classes de celles-ci, ...jusqu'aux sous-classes feuilles.

La relaxation d'un type de classe se fait en deux étapes : d'abord réinstaurer le contexte de pose de la contrainte de classe, puis reposer toutes les contraintes de classes postérieures à la contrainte supprimée et se trouvant dans le même réseau. En effet, les contraintes postérieures n'appartenant pas à ce réseau n'ont pas à être repropagées : elles n'ont pas d'incidence sur les attributs du réseau de la contrainte supprimée, leur effet est déjà enregistré et n'est pas remis en cause par cette suppression.

Le contexte de pose de la contrainte de classe supprimée est un ensemble de paires *(nom d'attribut, domaine)*. Au moment de la pose de la contrainte, on sauve le domaine de tous les attributs contraints du réseau auquel va être connectée la contrainte. Ce domaine est issu du type de l'attribut pour la classe. Il a été établi après filtrage par l'ensemble des contraintes de classe posées jusqu'alors.

Dans le réseau, les attributs qui étaient déjà contraints au moment de la pose de la contrainte retrouvent leur domaine d'alors. Les autres attributs qui font partie du réseau actuel mais qui ont été introduits par les contraintes postérieures retrouvent le domaine fourni par leur définition dans la classe.

On enlève la contrainte du réseau. Et on propage chacune des contraintes postérieures. À l'issue de cette propagation, qui termine puisqu'on relaxe un CSP stable (sans domaine vide), on obtient les domaines (relaxés) des attributs pour la classe dont est formé le nouveau type de la classe.

On peut ensuite calculer le type de chacune des sous-classes selon le même principe.

Dans toutes les instances, il faut supprimer la contrainte de classe et donc procéder au recalcul du type pour l'instance. Le principe est de réinstaurer le type de l'instance au moment de la pose de la contrainte de classe puis d'ajouter les contraintes d'instances (donc aussi de concept et de classe) suivantes.

Le retrait d'une contrainte de classe r_C (considérée au niveau de l'instance comme une contrainte d'instance r_{jI}) transforme le type d'une instance I de concept K $< a_{1K} : T_{a_{1I}}, \ldots, a_{nK} : T_{a_{nI}} >$ de domaine record associé $< a_{1K} : D_{a_{1I}}, \ldots, a_{nK} : D_{a_{nI}} >$ en $< a_{1K} : T'_{a_{1I}}, \ldots, a_{nK} : T'_{a_{nI}} >$ de domaine record associé $< a_{1K} : D'_{a_{1I}}, \ldots, a_{nK} : D'_{a_{nI}} >$ avec $D'a_{iI} = \bigcirc_{k=j+1}^{r} \rho_{r_{kI}}(D''_{a_{iI}})$, $\forall i \in [1, n]$, avec $D''_{a_{iI}}$ domaine de l'étiquette a_{iK} du contexte de pose de r_{jI}, r nombre de contraintes

Procédure propage-retrait-contr-class($\mathcal{C}$: contrainte ; c : classe)
début
calcul-type-retrait($\mathcal{C}$, c)
si sous-classes(c) $\neq \emptyset$ **alors**
pour tout c' $\in$ sous-classes(c) **faire**
propage-retrait-contr-class($\mathcal{C}$, c')
finpour
finsi
fin

Procédure calcul-type-retrait($\mathcal{C}$: contrainte, c : classe)
début
restaurer-contexte-contr-class($\mathcal{C}$, c)
% restauration du contexte de pose %
enlever-contrainte
% on supprime physiquement la contrainte du réseau %
pour tout $\mathcal{C}' \in$ contr(c) t.q. $\mathcal{C}' >_{contr(c)} \mathcal{C} \wedge$ même-réseau($\mathcal{C}'$, $\mathcal{C}$) **faire**
% la contrainte $\mathcal{C}'$ est posée après $\mathcal{C}$ et est du même réseau %
propager-contrainte($\mathcal{C}'$, c)
fin pour
fin

TAB. 7.2 - : Algorithme de calcul des types des classes après retrait d'une contrainte de classe. Les procédures *propager-contrainte* et *restaurer-contexte-contr-class* sont décrites aux section 8.3.1 et 8.4 respectivement.

posées sur l'instance (contraintes de classe et de concept incluses).

Autrement dit, le nouveau type de l'instance est calculé :

1. à partir du type de l'instance au moment de la pose de la contrainte,
2. en ajoutant les contraintes de concept, de classes et d'instances posées après r_{jI} sur I.

7.1.3.3 Retrait d'une contrainte de concept

Dans la suite, on suppose que les contraintes sont ordonnées selon leur pose, de sorte qu'une contrainte r_i posée après une contrainte r_j vérifie $i > j$.

Le retrait d'une contrainte de concept r_{jK} transforme le type d'un concept K $< a_{1K} : T_{a_{1K}}, \ldots, a_{nK} : T_{a_{nK}} >$ de domaine record associé $< a_{1K} : D_{a_{1K}}, \ldots, a_{nK} : D_{a_{nK}} >$ en $< a_{1K} : T'_{a_{1K}}, \ldots, a_{nK} : T'_{a_{nK}} >$ de domaine record associé $< a_{1K} : D'_{a_{1K}}, \ldots, a_{nK} : D'_{a_{nK}} >$ avec $D'a_{iK} = \bigcirc_{k=j+1}^{r} \rho_{r_{kK}}(D''_{a_{iK}})$, $\forall i \in [1, n]$, avec $D''_{a_{iK}}$ domaine de l'étiquette a_{iK} du contexte de pose de r_{jK}, r nombre de contraintes du concept.

Autrement dit, le nouveau type du concept est calculé :

1. à partir du type du concept au moment de la pose de la contrainte,
2. en ajoutant les contraintes de concept posées après r_{jK} sur K.

Dans toutes les classes où elle apparaît, il faut supprimer la contrainte de concept et donc procéder au recalcul du type pour la classe. Le principe est de réinstaurer le type de la classe au moment de la pose de la contrainte de concept puis d'ajouter les contraintes de classes (donc aussi de concept) suivantes.

Le retrait d'une contrainte de concept r_K (considérée au niveau de la classe comme une contrainte de classe r_{jC}) transforme le type d'une classe C $< a_{1C} : T_{a_{1C}}, \ldots, a_{mC} : T_{a_{mC}} >$ de domaine record associé $< a_{1C} : D_{a_{1C}}, \ldots, a_{mC} : D_{a_{mC}} >$ en $< a_{1C} : T'_{a_{1C}}, \ldots, a_{mC} : T'_{a_{mC}} >$ de domaine record associé $< a_{1C} : D'_{a_{1C}}, \ldots, a_{mC} : D'_{a_{mC}} >$ avec $D'a_{iC} = \bigcirc_{k=j+1}^{r} \rho_{r_{kC}}(D''_{a_{iC}})$, $\forall i \in [1, m]$, avec $D''_{a_{iC}}$ domaine de l'étiquette a_{iC} du contexte de pose de r_{jC} et r nombre de contraintes de la classe (contraintes de concept incluses).

Autrement dit, le nouveau type de la classe est calculé :

1. à partir du type de la classe au moment de la pose de la contrainte,
2. en ajoutant les contraintes de concept et de classes posées après r_{jC} sur C.

Dans toutes les instances, il faut supprimer la contrainte de concept et donc procéder au recalcul du type pour l'instance. Le principe est de réinstaurer le type de l'instance au moment de la pose de la contrainte de concept puis d'ajouter les contraintes d'instances (donc aussi de concept et de classe) suivantes.

Le retrait d'une contrainte de concept r_K (considérée au niveau de l'instance comme une contrainte d'instance r_{jI}) transforme le type d'une instance I de concept K $< a_{1K} : T_{a_{1I}}, \ldots, a_{nK} : T_{a_{nI}} >$ de domaine record associé $< a_{1K} : D_{a_{1I}}, \ldots, a_{nK} : D_{a_{nI}} >$ en $< a_{1K} : T'_{a_{1I}}, \ldots, a_{nK} : T'_{a_{nI}} >$ de domaine record associé $< a_{1K} : D'_{a_{1I}}, \ldots, a_{nK} : D'_{a_{nI}} >$ avec $D'a_{iI} = \bigcirc_{k=j+1}^{r} \rho_{r_{kI}}(D''_{a_{iI}})$, $\forall i \in [1, n]$, avec $D''_{a_{iI}}$ domaine de l'étiquette a_{iK} du contexte de pose de r_{jI}, r nombre de contraintes posées sur l'instance (contraintes de classe et de concept incluses).

Autrement dit, le nouveau type de l'instance est calculé :

1. à partir du type de l'instance au moment de la pose de la contrainte,
2. en ajoutant les contraintes de concept, de classes et d'instances posées après r_{jI} sur I.

7.2 Contraintes et dynamicité

Nous étudions dans un premier temps les précautions à prendre, en ce qui concerne la cohérence de la base de connaissances, pour parvenir à gérer des CSP dynamiques dans TROPES.

Dans un deuxième temps, nous traitons de la dynamicité des entités de représentation. Dans sa forme la plus simple, elle correspond à la modification d'une valeur d'attribut et a également des conséquences sur la définition et sur la cohérence des CSP établis.

7.2.1 Dynamicité des CSP

Une fois calculé le nouveau type de l'entité concernée par la pose ou le retrait d'une contrainte, il faut procéder à l'ajout ou au retrait *effectif* sur l'extension de l'entité (dans le cas d'une contrainte de concept ou de classe) ou sur l'entité elle-même (c'est-à-dire d'une contrainte d'instance).

7.2.1.1 Ajout de contrainte

L'objectif est ici de disposer de CSP incrémentaux, c'est-à-dire auxquels on peut soumettre à tout moment l'ajout d'une contrainte. Nous étudions l'effet d'un ajout de contrainte selon l'entité de représentation sur laquelle est déclarée la contrainte ajoutée.

- Toute nouvelle contrainte de concept (respectivement de classe) doit être appliquée à toutes les instances de ce concept (respectivement de cette classe). En effet, la contrainte fait office de nouvelle propriété ajoutée à la définition du concept (respectivement de la classe). Pendant toute la durée de vie de la contrainte, chaque instance du concept (respectivement de la classe) doit y être soumise.
- Toute nouvelle contrainte d'instance doit être appliquée à cette instance.

Dans l'hypothèse qu'une contrainte est acceptable pour l'entité de représentation sur laquelle elle porte (c'est-à-dire qu'on peut supposer après le calcul de type qu'il existe des instances et des valeurs d'attributs satisfaisant cette contrainte et l'ensemble des contraintes établi jusqu'alors), il nous faut étudier les conséquences de cet ajout dynamique de contrainte.

Si tout se passe bien, cet ajout de contrainte peut n'avoir aucun effet ou peut être la cause de réductions de domaines dont l'effet extrême est une inférence de valeur. Mais il se peut également qu'une instance antérieure à la contrainte ne la satisfasse pas. Les décisions à prendre dans ce cas diffèrent selon le niveau de représentation auquel la contrainte est déclarée.

- Si une instance de concept existante ne satisfait pas la contrainte ajoutée à ce concept, alors deux solutions sont envisageables (*cf.* figure 7.2) : soit on supprime la contrainte, soit on supprime l'instance. La première solution peut être choisie si l'on souhaite, par exemple, affiner la description d'un concept tout en conservant les instances reconnues comme membres du concept, comme connaissances sûres. La seconde solution peut être adoptée dans le cas où la connaissance sur le concept n'est pas définitive et sûre au moment de la création de l'instance et que toute connaissance nouvelle est autorisée à remettre en cause des croyances (comme ici l'appartenance de l'instance au concept). La première solution consiste à confronter la contrainte à l'extension courante du concept avant de l'entériner définitivement. La seconde solution consiste donc à conformer l'extension du concept à sa nouvelle définition. L'arbitrage de l'utilisateur nous parait être dans ce cas, la solution la plus raisonnable.

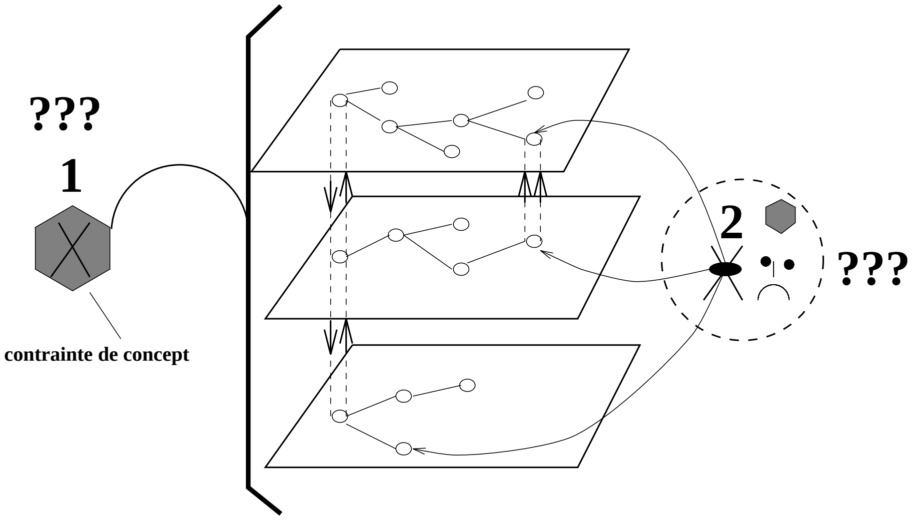


FIG. 7.2 - : Les deux possibilités de remise en cause lors de l'ajout d'une contrainte de concept : supprimer la contrainte ou supprimer l'instance ?

- Si une instance de classe existante ne satisfait pas la contrainte ajoutée à cette classe, alors trois solutions sont envisageables (*cf.* figure 7.3): soit on supprime la contrainte, soit on supprime l'instance, soit l'instance est reclassée. Les deux premières solutions donnent lieu à la même alternative que le cas précédent. La troisième solution proposée ici, qui n'est possible que si la classe en question n'est pas la classe racine, est plus conciliante. Elle consiste à rompre le lien d'appartenance et à tenter une nouvelle classification depuis la sur-classe et vers les classes sœurs. Ainsi, l'information modélisée n'est perdue – l'instance est conservée si elle trouve asile ailleurs – que si la classification échoue. Au cas où cette reclassification échoue, l'arbitrage de l'utilisateur nous parait être la solution la plus raisonnable.
- Si une instance ne satisfait pas la contrainte d'instance qui vient de lui être ajoutée, la contrainte est supprimée. En effet l'instance ne peut être supprimée puisqu'elle est l'objet de la contrainte.

  Lorsque des changements profonds, comme la suppression d'instances, sont imposés par l'ajout d'une contrainte et par choix de l'utilisateur, il est souhaitable de conserver l'état de la base avant que l'ajout ne soit pris en compte. Ceci est possible grâce à un mécanisme de versionnement [Tayar95]. Ainsi, si une contrainte posée se révèle peu judicieuse (car elle est trop forte ou ne rend finalement pas bien compte de la caractéristique à observer), l'utilisateur a la possibilité de revenir en arrière.

  D'autre part, si la contrainte se révèle trop forte car elle requiert la suppression de certaines instances, on pourrait envisager de supprimer une des contraintes du même réseau. L'utilisateur peut donc refuser dans un premier temps la contrainte ajoutée, en supprimer une autre

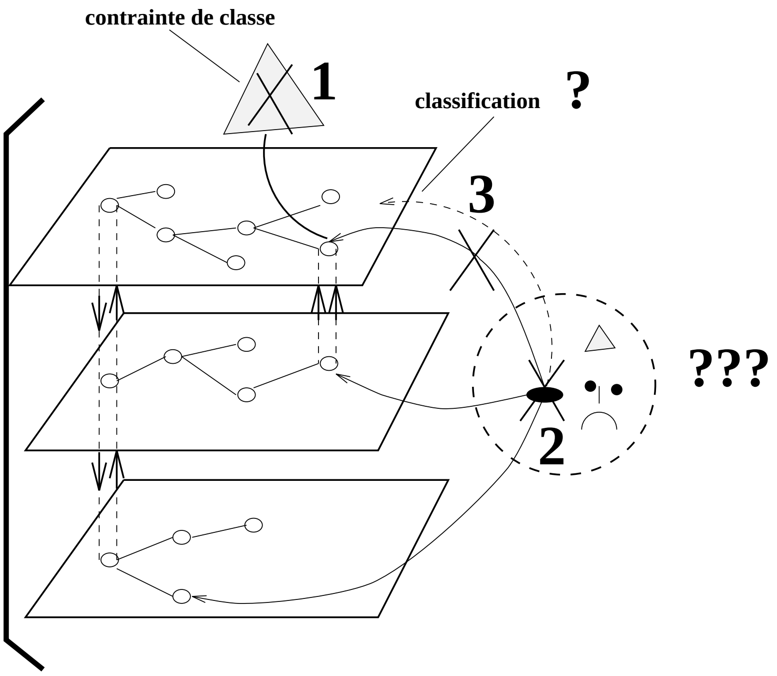


FIG. 7.3 - : Les trois possibilités de remise en cause lors de l'ajout d'une contrainte de classe : supprimer la contrainte ou supprimer l'instance ou reclasser l'instance?

de son choix et reposer la contrainte.

Si l'extension du concept (ou de la classe) est conséquente, l'ajout d'une contrainte de concept (ou de classe) peut se révéler à la fois coûteux en temps (calcul des types, propagation sur chaque instance) et en mémoire (pose sur chaque instance). L'utilisateur a donc intérêt à fournir l'ensemble des contraintes de concept (ou de classe) au moment de sa définition (avant la création de la première instance). L'ajout (comme le retrait) de contrainte de concept (ou de classe) trouve néanmoins son intérêt dans les bases de connaissances évolutives où rien n'est figé, même pas les descriptions des entités.

L'ajout (comme le retrait) d'une contrainte d'instances est en principe moins coûteux que l'ajout ou le retrait d'une contrainte de concept ou de classe, car portant sur un ensemble restreint d'objets. Ces fonctionnalités nous semblent essentielles pour juger à la fois de l'impact des modifications de la description de certains individus et non pas d'un groupe complet. De cette façon, deux types de raisonnement par contraintes sont disponibles dans TROPES : le raisonnement par contrainte collectif et le raisonnement par contrainte individuel.

7.2.1.2 Retrait de contrainte

L'objectif est ici de disposer de CSP décrémentaux, c'est-à-dire auxquels on peut soumettre à tout moment le retrait d'une contrainte. Nous étudions l'effet d'un retrait de contrainte selon l'entité de représentation sur laquelle porte la contrainte à retirer.

Le retrait d'une contrainte s'effectue en trois temps :

1. suppression de la contrainte du réseau
2. restauration du contexte de pose de la contrainte à supprimer
3. propagation des contraintes du réseau posées ultérieurement

- Toute contrainte de concept supprimée doit être relâchée sur toutes les instances de ce concept (puisqu'elle portait sur chacune d'elles). En effet, cette contrainte ne faisant plus partie de la description intrinsèque du concept, toute instance de ce dernier n'est plus censée la respecter. En réalité, on peut observer que ce qui est à remettre en cause ici, ce sont simplement les inférences qui ont pu être produites à partir de la contrainte. Cette dernière étant supprimée, elles n'ont plus de justification dans la base de connaissances.

- Toute contrainte de classe supprimée doit être relâchée sur toutes les instances de cette classe (puisqu'elle portait sur chacune d'elles). Pour les mêmes raisons qu'au cas précédent, il faut procéder à l'annulation des inférences qui ont été possibles grâce à la contrainte. La suppression d'une contrainte de classe a une autre conséquence : elle peut maintenant permettre le rattachement d'instances dont l'appartenance à la classe avait été refusée en raison de cette contrainte lors d'une classification postérieure à cette contrainte.
- Toute contrainte d'instance supprimée doit entraîner l'annulation des inférences qu'elle a permises.

Par inférences, il est entendu ici les divers retraits de valeurs inconsistantes des domaines des attributs contraints qui sont dus à l'application des règles de consistance de la contrainte et aux réductions entraînées par leur propagation, qu'elles se soient produites au moment de la pose de la contrainte ou ultérieurement, lors d'une modification de la base de connaissances.

Afin que ces inférences soient annulées, il faut conserver le contexte de pose de ces contraintes, c'est-à-dire l'état des domaines du réseau dans lequel elles ont été plongées, *avant* que sa présence ne soit prise en compte. Dans un premier temps, il s'agira de restaurer ce contexte, puis, dans un second temps, d'atteindre un état de la base de connaissances cohérent en l'absence de la contrainte : celui obtenu si la contrainte ne figurait pas dans l'ensemble des contraintes définies.

Le versionnement peut également permettre la sauvegarde des informations se trouvant dans la base avant la suppression de la contrainte, et se charger du stockage du contexte de pose de la contrainte.

7.2.2 Dynamicité des entités de représentation

7.2.2.1 Modification de valeur d'un attribut contraint

Comme il est d'usage dans une base de connaissances Tropes, il doit être possible de modifier à tout moment la valeur d'un attribut contraint. La nouvelle valeur proposée doit tout d'abord être présente dans le domaine – consistant au regard de l'ensemble des contraintes – de l'attribut contraint. Si tel est le cas, alors il faut tester la consistance de la valeur proposée. Pour cela, il faut procéder tout d'abord à l'annulation des inférences ou réductions de domaines permises par la valeur précédente, puis propager les contraintes de l'attribut affecté de la nouvelle valeur. Si cette dernière se révèle incohérente, alors il faut rétablir l'ancienne valeur.

7.2.2.2 Modification du domaine d'un attribut contraint

La modification du domaine d'un attribut contraint (dans un concept, une classe ou une instance) doit être répercutée sur le réseau de contraintes impliquant l'attribut. Le processus est identique à celui utilisé pour la modification de la valeur : il faut procéder à l'annulation des inférences ou réductions de domaines permises par le domaine précédent, puis propager les contraintes de l'attribut affecté du nouveau domaine. Si la propagation échoue alors l'ancien domaine devra être rétabli.

7.2.2.3 Ajout d'un concept

L'ajout d'un concept a pour seul effet le calcul du type des attributs impliqués dans des contraintes de ce concept.

7.2.2.4 Retrait d'un concept

Le retrait d'un concept doit entraîner la suppression des contraintes de ce concept, des contraintes des classes de ce concept, des contraintes d'instances de ce concept. Les instances d'autres concepts atteintes par des accès depuis le concept supprimé sont libérées de ces contraintes.

7.2.2.5 Ajout d'une classe

Lorsqu'une classe est ajoutée à la hiérarchie de classes, alors elle hérite des contraintes de ses sur-classes et fait hériter ses contraintes à ses sous-classes (*cf.* figure 7.4). La viabilité de la classe dépend donc du calcul de type effectué à partir des contraintes des sur-classes et des calculs de types effectués sur les sous-classes.

Afin d'accélérer plus encore l'héritage des contraintes, la propagation s'effectue depuis les feuilles de la hiérarchie situées sous la classe ajoutée jusqu'à ses sous-classes immédiates. La contrainte étant posée sur les sous-classes les plus spécifiques, si une inconsistance doit se produire, elle sera ainsi détectée au plus tôt.

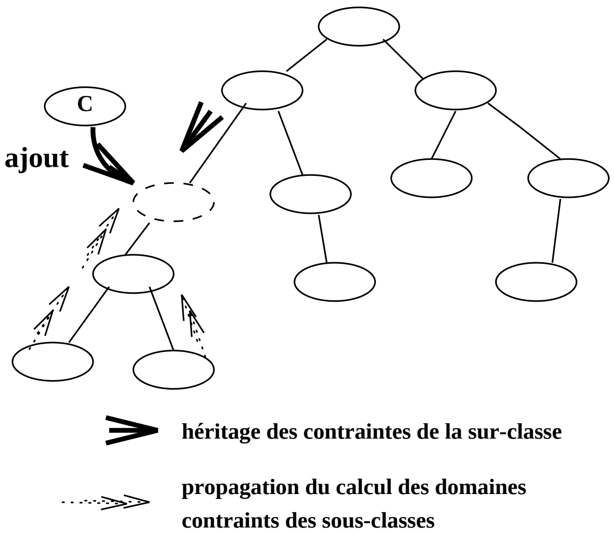


FIG. 7.4 - : Ajout d'une classe dans la hiérarchie : la classe C hérite les contraintes de sa sur-classe et propage ses propres contraintes depuis les feuilles de la hiérarchie jusqu'à elle.

7.2.2.6 Retrait d'une classe

Le retrait d'une classe dans la hiérarchie a pour effet de rattacher ses instances à sa sur-classe. En effet, contrairement à ce qui se passe pour le concept, une instance survit à sa classe. Il faut donc libérer ces instances des contraintes associées à la classe supprimée. Il faut également libérer de ces contraintes les instances des sous-classes de la classe supprimée qui ont hérité de ses contraintes.

7.2.2.7 Ajout d'une instance

À l'ajout (création) d'instance, les contraintes de concept lui sont appliquées. Si elle ne les satisfait pas, la création échoue. Sinon, le rattachement de l'instance dans chacun des points de vue doit être validé. Si, dans un point de vue, aucune classe n'est proposée, la classe racine est prise par défaut. Dans chaque point de vue, il faut donc appliquer toutes les contraintes de la classe candidate au rattachement. Lorsque celles-ci sont satisfaites, le rattachement est accepté.

Si celles-ci ne sont pas respectées, le rattachement est refusé. Le système peut alors, à la demande de l'utilisateur, effectuer une classification de l'instance dans le point de vue.

7.2.2.8 Retrait d'une instance

Le retrait d'une instance a pour effet de supprimer toutes les contraintes qui portent sur elle comme un tout – contraintes sur des instances – ou sur ses attributs (contraintes d'attributs définies sur un concept, une classe, une instance ou entre instances). Une contrainte ne survit donc pas à l'instance sur laquelle elle porte. Même dans le cas d'un partage de contraintes (contrainte d'instances), la suppression d'une des instances met fin à l'existence de la contrainte.

Si l'instance était valeur d'un attribut étape d'un accès, l'accès est conservé – il appartient à une contrainte d'une autre instance – mais il n'a plus de valeur.

7.2.2.9 Ajout d'un attribut de concept, de classe

L'ajout d'un attribut de concept ou de classe n'a pas d'effet direct sur les CSP présents tant qu'une contrainte le concernant n'est pas créée[3]. Ces ajouts visent à étendre la structure du concept ou de la classe.

7.2.2.10 Retrait d'un attribut de concept, de classe

Le retrait d'un attribut de concept ou de classe doit entraîner la suppression de toutes les contraintes portant sur lui ou l'utilisant dans un accès. Ces retraits ont pour objectif de réduire la structure du concept ou de la classe (donc de toutes les instances). Les contraintes de concept, de classe, d'instance ou entre instances impliquant l'attribut supprimé devront être ôtées.

7.3 Conclusion

Ce chapitre a décrit les diverses liaisons entre contraintes, typage et dynamicité.

Un type est associé à chaque entité de représentation dans Tropes. Aussi, les contraintes définies sur ces entités sont susceptibles d'en modifier le domaine et par là, le type. Afin de maintenir la cohérence (consistance) des entités contraintes, la pose et le retrait d'une contrainte doivent tout d'abord déclencher le calcul du type de l'entité concernée avant d'être effectivement réalisés sur l'entité ou son extension.

Après avoir formalisé la notion de domaine record, nous avons défini le calcul de type relatif à l'ajout et au retrait de contrainte de concept, de classe et d'instance. Bien que coûteux, ce calcul permet de reporter la consistance des CSP aux objets Tropes.

Dans une deuxième partie, nous nous sommes intéressés aux conséquences de l'ajout et du retrait d'une contrainte sur l'extension d'un concept et d'une classe, ainsi que sur une instance, en indiquant notamment quelles mesures pouvaient être prises lors d'une inconsistance. De même, nous avons indiqué quelles étaient les conséquences de l'ajout et du retrait d'une entité de représentation (concept, classe, attribut, instance) sur les contraintes rattachées à cette entité.

Ce chapitre suggère donc une étroite collaboration entre Metéo, le module de gestion de types de Tropes et le module dédié à la gestion des CSP. Il a montré que les opérations sur les contraintes (ajout ou retrait) se répercutent sur les types et sur les objets et, qu'inversement, la manipulation des objets Tropes (création, modification, suppression) se répercute sur les CSP.

Nous décrivons à présent le module de gestion des CSP Tropes, appelé Micro. Il est chargé de remplir le cahier des charges concernant les CSP Tropes et établi lors de ce chapitre et du chapitre précédent.

[3] Les contraintes impliquant des attributs non définis étant refusées.

Chapitre 8

Micro

Nous présentons dans ce chapitre un module de programmation par contraintes, appelé MICRO (Module pour l'Intégration de Contraintes et de Relations dans les Objets), destiné à être couplé à un modèle de connaissances à objets et, plus particulièrement, au modèle TROPES.

Ce module doit répondre aux divers besoins et exigences de TROPES en matière de CSP qui ont été décrits dans les deux chapitres précédents. Toutefois, avant de nous lancer dans la construction d'un tel module, nous avons tenté de coupler la première version de TROPES à un module de programmation par contraintes existant. Nos choix se sont portés vers PECOS [Ilog92b], le module de programmation par contraintes écrit pour le langage LE-LISP V15. Malgré la puissance et les fonctionnalités de ce module, la gestion d'unions d'intervalles et de la dynamicité dans TROPES nous a conduit à abandonner cette solution et à construire un module de programmation par contraintes pour TROPES. Parce que l'on apprend toujours d'un échec, nous relatons ici rapidement cette expérience (*cf.* section 8.1).

Bien que les caractéristiques de TROPES aient présidé aux spécifications de ce module, elles ont un caractère relativement général qui fait que le module présenté ici peut être utilisé dans un autre contexte, ou même de façon autonome.

MICRO est capable de maintenir et de résoudre :

- des CSP numériques dont les domaines, finis ou infinis, sont exprimés sous forme d'énumération de valeurs ou d'union d'intervalles ;
- des CSP booléens ;
- des CSP dont les variables ont pour valeur un ensemble ou une liste de valeurs de même type.

La définition de ces CSP peut être dynamique. La gestion de CSP dynamiques s'est imposée par le souci de considérer les contraintes comme des éléments de connaissances que l'on peut ajouter ou supprimer à tout moment. MICRO est capable de réagir à l'ajout mais aussi au retrait d'une contrainte, sans intervention extérieure : l'utilisateur ne doit pas procéder à la redéfinition du CSP lors de l'ajout ou du retrait d'une contrainte.

La suite de ce chapitre est constituée de quatre parties :

- Tout d'abord, nous décrivons, pour chaque composant d'un CSP traité par MICRO, les diverses formes qu'il peut revêtir : les types des variables, la forme des domaines, l'ensemble des contraintes prédéfinies. Nous montrons comment sont décrits ces composants dans le langage objet hôte de MICRO (*cf.* section 8.2).
- Nous découvrons alors la partie immergée de l'iceberg : les algorithmes de propagation et de résolution intégrés dans MICRO (*cf.* section 8.3).
- Ensuite, la gestion de la dynamicité est étudiée à travers la présentation des structures de données et des mises à jour nécessaires lors de la modification d'un réseau de contraintes (*cf.* section 8.4).
- Enfin, nous présentons un type particulier de contraintes dont les arguments sont des contraintes : les *méta-contraintes* (*cf.* section 8.5).

Nous concluons ce chapitre par une comparaison du module MICRO avec d'autres modules ou langages de programmation par contraintes (*cf.* section 8.7) étudiés aux chapitres 4 et 5.

8.1 Un couplage faible : l'expérience Pecos

Ayant constaté le besoin de contraintes dans TROPES (chapitre 3), nous devions, après avoir étudié leurs différents usages et localisations, choisir un système ou une bibliothèque de programmation par contraintes parmi les bibliothèques disponibles[1] dont les capacités, l'extensibilité et les facilités d'utilisation et de couplage soient les mieux adaptées aux objectifs à atteindre pour l'intégration à réaliser.

En réalité, peu de produits de programmation par contraintes étaient (et sont toujours) disponibles. Le choix déjà restreint devait donc être fait à partir des caractéristiques souhaitées des CSP de TROPES. La première des exigences que nous avions concernant la gestion de domaines numériques exprimés sous forme d'ensembles énumérés mais aussi d'intervalles, nous a conduit vers PECOS, la bibliothèque de programmation par contraintes du langage LE-LISP. La première version du noyau de TROPES ayant été développée dans ce langage, ce choix trouvait là une justification supplémentaire.

Un premier prototype d'intégration de contraintes dans TROPES a donc été construit avec PECOS [Gensel93b, Gensel95]. Le principe était d'associer à chaque attribut contraint de TROPES une variable contrainte de PECOS et de reporter chaque contrainte définie sur cet attribut dans TROPES sur la variable PECOS équivalente, les contraintes disponibles dans TROPES correspondant aux fonctions contraintes de la bibliothèque. Les fonctionnalités de PECOS nous permettaient d'attendre de PECOS les avantages suivants :

- disposer de primitives de manipulation (définition, modification) de variables contraintes entières, réelles (flottantes), ensemblistes ;
- disposer d'un ensemble prédéfini et extensible de contraintes numériques, ensemblistes, ou sur des objets ;
- disposer d'une recherche de solutions basée sur une énumération paramétrable par divers critères et un retour arrière contrôlable par divers points de choix.

Toutefois, d'autres attentes de notre intégration de contraintes ne pouvaient être simplement satisfaites et nécessitaient de recourir, à travers une interface, à des adaptations de PECOS. C'était notamment le cas pour les contraintes numériques, PECOS ne traitant pas d'unions d'intervalles, les domaines de ses variables n'étant exprimables que sous la forme d'un ensemble ou d'un seul intervalle. Une parade était envisageable : associer à un attribut contraint de TROPES dont le domaine est une union d'intervalles, plusieurs variables contraintes de PECOS et équiper l'interface d'une batterie d'opérateurs sur les intervalles (union, intersection...) en liaison avec les primitives de création et de suppression de variables contraintes de PECOS.

Enfin, il est apparu que certaines des prérogatives de l'intégration de contraintes à TROPES se montraient (encore plus) difficilement réalisables voire même irréalisables sur les CSP définis en PECOS :

- La panoplie de contraintes ensemblistes de PECOS nous paraissait d'usage assez complexe (recours aux *bitsets*) et ne recouvrait pas tout nos besoins.
- Aucune contrainte prédéfinie n'existait sur les listes.
- La modification de la valeur d'une variable contrainte dans PECOS implique de poser à nouveau l'ensemble des contraintes qui portent sur elle (encore faut-il le conserver). En fait, les domaines ne sont que réductibles (hormis au moment de la résolution) dans PECOS.
- La suppression d'une contrainte de PECOS consiste en un gel de celle-ci ; elle est rendue simplement inactive mais n'est pas réellement supprimée dans le sens où les réductions éven-

[1] Dans le domaine public, en freeware ou shareware, ou dans le commerce.

tuellement opérées lors de propagation la concernant ne sont pas annulées. Le réseau de contraintes considéré n'est pas le réseau sans cette contrainte, c'est le réseau courant où la contrainte est inaccessible.

Les deux premiers aspects mettent en cause une limite de l'ensemble des contraintes prédéfinies de PECOS à laquelle on peut remédier par l'outil de définition de contraintes de PECOS.

Les deux derniers révèlent les limites de PECOS lorsqu'il s'agit de maintenir des CSP dynamiques.

Devant les efforts à mettre en œuvre pour construire une interface capable de traduire correctement la configuration des attributs contraints en variables contraintes PECOS, mais aussi pour définir en PECOS les contraintes qu'il ne fournit pas, il nous a semblé préférable d'abandonner cette solution d'autant qu'elle n'aurait été que partiellement satisfaisante, notamment en ce qui concerne les listes et la dynamicité.

C'est pourquoi nous avons choisi de construire notre propre module de programmation par contraintes, en nous basant sur les caractéristiques attendues de l'intégration (décrites aux chapitres 6 et 7). Les caractéristiques de ce module, baptisé MICRO, sont présentées dans la suite de ce chapitre et illustrent son autonomie vis-à-vis de TROPES.

8.2 Composants d'un CSP Micro

Nous décrivons ici les trois composants – variables, domaines, contraintes – qui forment un CSP de MICRO.

8.2.1 Les variables de Micro

Les variables des CSP traités par MICRO sont appelées par la suite Variables Contraintes de MICRO ou VCM.

Les CSP de MICRO sont définis sur des variables entières, réelles (flottants), booléennes, ou dont les valeurs sont des ensembles ou des listes de valeurs, toutes du même type (entier, réel ou booléen). Les trois types possibles pour les variables monovaluées correspondent aux types prédéfinis les plus usuels.

Il existe trois types de VCM : les VCM monovaluées (de type entier, réel (flottant) ou booléen), les VCM multivaluées de type ensemble d'entiers, de réels ou de booléens, et les VCM de type liste d'entiers, de réels ou de booléens.

Il est également possible de définir des variables de type quelconque qui sont destinées à n'être impliquées que par des contraintes d'égalité ou de différence.

Les opérations disponibles sur les VCM sont[2] :

- la création : une variable est définie par la donnée de son nom (identificateur), de son type et, éventuellement de son domaine de valeurs. Lorsque le type n'est pas précisé, il s'agit de l'ensemble, possiblement infini, de toutes les valeurs de ce type ;
- la suppression d'une variable qui conduit à la suppression de toutes les contraintes qui portent sur elle ;
- la modification de son domaine ;
- l'affectation d'une valeur à la variable ;
- la consultation de la variable pour connaître son type, son domaine, l'ensemble des contraintes qui portent sur elle ;
- la liste des variables du même réseau de contraintes ;
- la liste des variables déclarées.

[2] Ces fonctions sont données en annexe B.1.

8.2.2 Les domaines de Micro

L'expression du domaine dépend du type de la VCM considérée :

- Le domaine d'une variable monovaluée peut être donné sous la forme d'une énumération de valeurs (numériques ou booléennes) ou sous la forme d'une union d'intervalles, lorsque le type de la variable est ordonné (VCM numériques seulement).
  Par exemple :
  (`mic-create-var` *'x 'integer*) crée la variable entière x.
  (`mic-create-var` *'y 'float* '(1.2 4.5 7.3)) crée la variable réelle (flottante) y de domaine de valeurs l'ensemble $\{1.2, 4.5, 7.3\}$.
  (`mic-create-var` *'z 'integer* '(((-4 t) (1 t)) ((5 t) (7 t)))) crée la variable entière z de domaine de valeurs l'union d'intervalles $[-4, 1] \cup [5, 7]$.
- La représentation du domaine d'une variable multivaluée est donnée sous la forme d'un quintuplet pour une variable ensembliste, étendu à un sextuplet pour une variable de type liste. La description d'une variable multivaluée comporte donc au moins cinq éléments (éventuellement vides) :
  - l'*extension* qui représente le domaine de la variable multivaluée, c'est-à-dire l'ensemble des valeurs (des ensembles ou listes) de la variable. Lorsque l'extension est définie et que sa cardinalité vaut 1, la valeur de la variable est connue ;
  - le *domaine de ses éléments* qui représente l'ensemble de valeurs que peut prendre chacun des éléments de l'ensemble ou de la liste ;
  - l'*ensemble de valeurs impossibles* ;
  - sa *cardinalité* (ou longueur pour une liste). Il s'agit d'une union d'intervalles ;
  - l'*ensemble des éléments requis* qui contient les valeurs du domaine des éléments que l'on doit obligatoirement retrouver dans les diverses valeurs possibles de la variable.

  Par exemple :
  (`mic-create-set` *'s 'integer* '(((1 3 5) (9 4)) () () () ())) crée la variable ensembliste s de domaine de valeurs l'ensemble d'ensembles d'entiers $\{\{1, 3, 5\}, \{9, 4\}\}$
  (`mic-create-set` *'v 'integer* '(() (((4 t) (7 t))) ((4 5)) ((2 t) (3 t)) ((5)))) crée la variable ensembliste v de domaine de valeurs l'ensemble d'ensembles d'entiers compris entre 4 et 7, de cardinalité 2 ou 3, contenant l'entier 5, $\{4, 5\}$ exclu.
  Pour les listes, la description du domaine comprend un élément supplémentaire : l'*ensemble des éléments requis à un certain rang dans la liste* qui décrit les valeurs du domaine des éléments que l'on doit obligatoirement retrouver à un certain rang dans les diverses valeurs possibles de la variable. Cet ensemble est constitué de paires (e, n) où e est une valeur du domaine des éléments de la variable et n le rang dans la liste auquel on doit trouver la valeur e.
  (`mic-create-list` *'l 'integer* '(() (((0 t) (4 t))) () ((3)) () ((5 1)))) crée la variable liste l de domaine de valeurs l'ensemble de listes d'entiers compris entre 0 et 4, de longueur 3 et contenant l'entier 5 au rang 1.
  Il est clair que l'unicité de l'expression d'un domaine de VCM n'est pas garantie. Par exemple, la variable v ci-dessus peut aussi bien être définie par :
  (`mic-create-set` *'v 'integer* '(((5 6) (5 7) (4 5 6) (4 5 7) (5 6 7))) () () () ())

Afin d'optimiser le traitement de ces variables, MICRO procède en interne à une normalisation de l'expression des domaines des VCM. Pour les variables monovaluées numériques, elle consiste à transformer dès que possible une énumération en unions disjointes d'intervalles et à faire l'union d'intervalles non disjoints. Pour les variables multivaluées, après la normalisation :

- une extension est finie lorsque le domaine des éléments et la cardinalité sont finis ;
- le domaine des éléments d'une variable multivaluée est toujours défini ;
- l'ensemble des valeurs impossibles est non vide seulement dans le cas d'une extension infinie ;

- la cardinalité est toujours définie (qu'elle soit finie ou infinie) ;
- l'ensemble des éléments requis (respectivement requis à un certain rang) n'est conservé que dans le cas d'une extension infinie.

Des opérations de consultation d'un domaine à partir d'une variable, ou de modification de domaines, sont disponibles. Les opérations de modification de domaines déclenchent la propagation des contraintes qui portent sur les variables de ces domaines.

8.2.3 La bibliothèque de contraintes

L'expression d'une contrainte MICRO est formée à partir de contraintes de base qui se divisent en deux catégories :

- les contraintes *principales*, à deux arguments au plus, qui correspondent aux contraintes proprement dites (contrainte d'égalité, d'inégalité, de différence, d'appartenance...). On trouve dans cette classe de contraintes les opérateurs de comparaison entre expressions arithmétiques, booléennes, ensemblistes ou à base de listes, les opérateurs d'appartenance et les opérateurs de vérification d'une condition sur les éléments d'un ensemble ou d'une liste.
  Les arguments des contraintes principales sont, soit des constantes, soit des variables MICRO définies par l'utilisateur (variables de CSP), soit des expressions de contraintes secondaires. Dans ce dernier cas, l'argument est lié à la variable résultat de la contrainte secondaire.
  Dans une représentation par arbre de l'expression d'une contrainte, les contraintes principales sont les nœuds racines de l'expression (*cf.* figure 8.1). Les méta-contraintes sont des contraintes

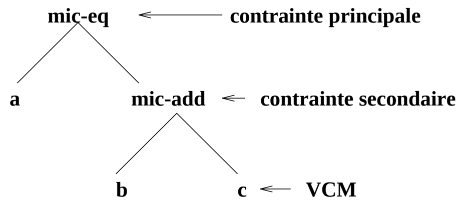


FIG. 8.1 - : Représentation arborescente de la contrainte $a = b + c$ en MICRO. À la racine de l'arbre, la contrainte principale (ici `mic-eq` pour l'égalité). Les nœuds internes (opérateurs des expressions contraintes) correspondent aux contraintes secondaires (ici `mic-add` pour $+$). Les feuilles correspondent aux constantes ou aux VCM (ici, a, b, c).

  principales particulières présentées à la section 8.5.
- les contraintes *secondaires*, à un argument ou plus, qui représentent les opérateurs employés dans les parties droite et gauche des contraintes principales, et permettent de combiner des sous-expressions contraintes. Elles correspondent aux opérateurs classiques arithmétiques, booléens, ensemblistes ou sur des listes.
  Contrairement aux contraintes principales, les contraintes secondaires ont un argument résultat. Ce dernier peut être donc à son tour utilisé comme argument d'une autre contrainte secondaire, ce qui permet l'expression de contraintes *complexes*. Il peut aussi être utilisé comme l'un des arguments d'une contrainte principale. Le résultat d'une contrainte secondaire est donc une variable gérée par MICRO qui n'est pas déclarée par l'utilisateur. Cette variable n'apparaît pas dans les expressions de contraintes à base de contraintes secondaires où elle demeure implicite. Pour MICRO, cette variable interne permet de lier une contrainte secondaire à la contrainte (principale ou secondaire) où elle apparaît.
  Les autres arguments d'une contrainte secondaire sont, soit des constantes, soit des variables MICRO définies par l'utilisateur (variables de CSP), soit des expressions de contraintes secondaires. Dans ce dernier cas, l'argument est lié à la variable résultat de la contrainte secondaire.
  Dans une représentation par arbre d'une expression contrainte, les contraintes secondaires correspondent aux nœuds intermédiaires. Leurs nœuds fils sont, soit d'autres nœuds de contraintes secondaires, soit des nœuds feuilles qui correspondent aux constantes ou à des variables de MICRO (*cf.* figure 8.1).

Ces contraintes principales et secondaires sont des contraintes prédéfinies qui traitent des variables monovaluées numériques, booléennes ou multivaluées. Nous les présentons dans les sections suivantes en donnant la condition ou l'opérateur équivalent.

8.2.3.1 Les contraintes numériques pour variables monovaluées

Les contraintes numériques s'appliquent aux entiers et aux réels (flottants) en manipulant des unions d'intervalles. Les principes de la prise en charge de CSP numériques à intervalles ont été exposés au chapitre 4 section 4.5. Il sont également appliqués dans MICRO, mais, au contraire des systèmes présentés alors, nous proposons de gérer des unions d'intervalles.

Les contraintes numériques principales correspondent aux opérateurs de comparaison arithmétiques classiques : égalité (`mic-eq`), différence (`mic-neq`), inégalités strictes (`mic-lt` et `mic-gt`), inégalités non strictes (`mic-le` et `mic-ge`). Nous avons ajouté à ces opérateurs de comparaison un opérateur unidirectionnel d'affectation (`mic-assign`).

Il faut noter que les contraintes d'égalité (`mic-eq`), de différence (`mic-neq`), et l'opérateur d'affectation (`mic-assign`, sont des contraintes génériques capables de prendre en argument des variables de types divers.

Les contraintes numériques secondaires correspondent aux opérateurs arithmétiques classiques. On distingue :

- les opérateurs unaires : valeur absolue (`mic-abs`), opposé (`mic-neg`), exponentielle (`mic-exp`), logarithme népérien (`mic-log`), sinus (`mic-sin`), cosinus (`mic-cos`) qui sont donc représentés par des contraintes secondaires à un argument et un résultat ;
- les opérateurs binaires : addition (`mic-add`), soustraction (`mic-sub`), multiplication (`mic--mul`), division (`mic-div`), puissance (`mic-power`) qui sont représentés par des contraintes secondaires à deux arguments et un résultat.

Nous donnons dans le tableau 8.1 l'ensemble des contraintes pour variables monovaluées numériques proposées par MICRO. Le nom des contraintes reste le même, que les VCM arguments soient entières ou réelles, mais le traitement peut, quant à lui, être différent. L'ensemble donné ici correspond

Contraintes	Symbole	Nom MICRO
Principales	$=$	`mic-eq`
	$\neq$	`mic-neq`
	$\leftarrow$	`mic-assign`
	$<$	`mic-lt`
	$\leq$	`mic-le`
	$>$	`mic-gt`
	$\geq$	`mic-ge`
Secondaires Unaires	$\|v\|$	`mic-abs`
	-v	`mic-neg`
	exp	`mic-exp`
	log	`mic-log`
	sin	`mic-sin`
	cos	`mic-cos`
Secondaires Binaires	$+$	`mic-add`
	-	`mic-sub`
	*	`mic-mul`
	/	`mic-div`
	^	`mic-power`

TAB. 8.1 - : Ensemble des contraintes pour variables monovaluées numériques proposées par MICRO.

aux opérateurs de comparaison ou arithmétiques disponibles dans la version actuelle de MICRO ; il n'est pas exhaustif (il manque, par exemple, *modulo, arc-sinus, arc-hyperbolique, etc.*) et sera étendu dans les prochaines versions.

8.2.3.2 Les contraintes booléennes pour variables monovaluées

Les contraintes booléennes principales sont au nombre de trois : égalité (`mic-eq`), différence (`mic-neq`), affectation (`mic-assign`).

L'ensemble des contraintes booléennes secondaires comporte :

- l'opérateur unaire de négation représenté par une contrainte secondaire à un argument et un résultat (`mic-not`).
- les opérateurs binaires booléens classiques : et (`mic-and`), ou (`mic-or`), implication (`mic--implies`), équivalence (`mic-equiv`). Ils sont représentés par des contraintes secondaires à deux arguments et un résultat.
- les opérateurs *semi-booléens* dérivés des opérateurs de comparaison ou d'appartenance qui sont des tests d'égalité (`mic-is-eq`), de différence (`mic-is-neq`), d'inégalité stricte ou non stricte (`mic-is-lt`, `mic-is-gt`, `mic-is-le`, `mic-is-ge`), d'inclusion stricte ou non (`mic--is-include`, `mic-is-include-eq`), de non-inclusion stricte ou non (`mic-is-not-include`, `mic-is-not-include-eq`), d'appartenance (`mic-in`) ou de non appartenance (`mic-not-in`), sur les éléments d'un ensemble ou d'une liste (`mic-is-every`, `mic-is-any`, `mic-is-all`), sur les éléments de deux listes pris deux à deux (`mic-is-compar`), sur la cardinalité d'un ensemble ou la longueur d'une liste (`mic-is-at-least`, `mic-is-at-most`). Ils sont représentés par des contraintes secondaires à deux arguments et un résultat. En général, ils correspondent à la forme prédicative (le test) d'une contrainte principale. Ces opérateurs sont dit semi-booléens parce que leur résultat est booléen mais leurs arguments doivent être du type attendu par le test effectué. Ces contraintes secondaires sont utilisées dans le cas de contraintes *conditionnelles* (*cf.* section 8.5). On obtient ainsi un comportement des expressions conditionnelles identiques à celui des contraintes : dès qu'un argument des conditions est modifié, la contrainte est propagée (*cf.* section 8.3.1)).

Nous donnons l'ensemble des contraintes pour variables monovaluées booléennes dans le tableau 8.2 proposées par MICRO. La liste donnée ici correspond aux opérateurs booléens disponibles dans la version actuelle de MICRO, elle n'est pas exhaustive (il manque, par exemple, *xor, etc.*) et sera étendue dans les prochaines versions.

8.2.3.3 Les contraintes de transformation

Les contraintes de transformation sont des contraintes secondaires que l'on peut utiliser pour construire des variables multivaluées (ensemble ou liste) à partir de variables monovaluées.

Ainsi, il est possible de transformer la valeur d'une variable monovaluée en un ensemble ou une liste qui est la valeur de la variable multivaluée donnée en résultat (`mic-mono-to-set`, `mic--mono-to-list`).

De même, il est possible de former la variable multivaluée à partir d'un ensemble ou d'une liste de variables monovaluées (`mic-set`, `mic-list`). Cette variable multivaluée a pour valeur la liste ou l'ensemble des valeurs des variables monovaluées qui la composent, si elles sont toutes définies.

Nous donnons l'ensemble des contraintes pour variables monovaluées booléennes dans le tableau 8.3 proposées par MICRO.

8.2.3.4 Les contraintes pour variables multivaluées

L'objectif ici est de fournir le moyen de définir des contraintes sur des variables multivaluées qui ont pour valeur un ensemble ou une liste de valeurs de même type (entier, réel ou booléen). La plupart des contraintes opérant sur des variables de type liste sont inspirées et reposent sur des fonctions du langage LISP qui est le langage hôte de MICRO.

Les contraintes principales pour variables multivaluées correspondent :

- aux opérateurs de comparaison entre les listes ou les ensembles : égalité (`mic-eq`), différence

Contraintes	Symbole	Nom MICRO
Principales	$=$	`mic-eq`
	$\neq$	`mic-neq`
	$\leftarrow$	`mic-assign`
Secondaires Unaires	$\neg$	`mic-not`
Secondaires Binaires	or	`mic-or`
	and	`mic-and`
	$\Rightarrow$	`mic-implies`
	$\leftrightarrow$	`mic-equiv`
Semi-Booléennes monovaluées	$\overset{?}{=}$	`mic-is-eq`
	$\overset{?}{\neq}$	`mic-is-neq`
	$<^{?}$	`mic-is-lt`
	$\overset{?}{\leq}$	`mic-is-le`
	$>^{?}$	`mic-is-gt`
	$\overset{?}{\geq}$	`mic-is-ge`
Semi-Booléennes multivaluées	$\overset{?}{\subset}$	`mic-is-include`
	$\overset{?}{\subseteq}$	`mic-is-include-eq`
	$\overset{?}{\not\subset}$	`mic-is-not-include`
	$\overset{?}{\not\subseteq}$	`mic-is-not-include-eq`
	$\overset{?}{\in}$	`mic-is-in`
	$\overset{?}{\notin}$	`mic-is-not-in`
	$\overset{?}{every}$	`mic-is-every`
	$\overset{?}{any}$	`mic-is-any`
	$\overset{?}{all}$	`mic-is-all`
	$\overset{?}{compar}$	`mic-is-compar`
	$at \overset{?}{-} least$	`mic-is-at-least`
	$at \overset{?}{-} most$	`mic-is-at-most`

TAB. 8.2 - : Ensemble des contraintes pour variables booléennes proposées par MICRO.

Contraintes	Symbole	Nom MICRO
	monovaluée $\rightarrow$ ensemble	`mic-mono-to-set`
	monovaluée $\rightarrow$ liste	`mic-mono-to-list`
	ensemble de monovaluées	`mic-set`
	liste de monovaluées	`mic-list`

TAB. 8.3 - : Ensemble des contraintes de transformation pour variables monovaluées proposées par MICRO.

(`mic-neq`), affectation (`mic-assign`), inclusion stricte ou non (`mic-include`), non-inclusion stricte ou non (`mic-not-include`, `mic-not-include-eq`);

- aux opérateurs de vérification d'une condition portant sur tous les éléments d'une liste ou d'un ensemble (`mic-every`), sur au moins un élément d'une liste ou d'un ensemble (`mic-any`), sur tous les éléments d'une liste ou d'un ensemble pris deux à deux (`mic-all`), ou sur tous les éléments de même rang de deux listes (`mic-compar`).

À chacune de ces contraintes principales correspond un opérateur semi-booléen représenté par une contrainte secondaire.

On trouve également des contraintes principales pouvant impliquer une variable monovaluée et une variable multivaluée, sous forme de contraintes :

- d'appartenance (`mic-in`) ou de non-appartenance (`mic-not-in`);

- de cardinalité (`mic-at-least`, `mic-at-most`).

Les contraintes secondaires pour variables multivaluées correspondent aux divers opérateurs dont l'un des arguments est une liste ou un ensemble.

Parmi les opérateurs unaires, on distingue :
- des opérateurs classiques sur les listes permettant d'obtenir le premier élément (`mic-car`), le reste (`mic-cdr`), l'élément minimum (`mic-min`) ou maximum (`mic-max`) (pour des ensembles ou des listes de nombres) ;
- des opérateurs permettant d'obtenir la cardinalité d'une variable de type ensemble (`mic-card`) ou la longueur d'une variable de type liste (`mic-length`) ;
- un opérateur permettant l'application d'une fonction du langage hôte (ou d'une contrainte MICRO) à chacun des éléments d'un ensemble ou une liste (`mic-map`) ;
- un opérateur permettant l'application d'une fonction ou d'une contrainte (binaire) aux éléments d'une liste (`mic-apply`). Le résultat de cet opérateur est une variable monovaluée interne.
- un opérateur permettant de sélectionner, dans une liste ou un ensemble, les éléments qui vérifient une condition ou une contrainte (`mic-select`).

Parmi les opérateurs binaires, on distingue :
- des opérateurs ensemblistes réalisant l'union (`mic-union`), l'intersection (`mic-inter`), ou la différence (`mic-diff`) de deux ensembles ;
- un opérateur de concaténation de listes (`mic-append`) ;
- un opérateur d'application d'une fonction ou d'une contrainte secondaires aux éléments de même rang de deux listes (`mic-combin`).

On trouve également des contraintes secondaires pouvant impliquer une variable monovaluée et une variable multivaluée, sous forme d'opérateurs :
- de concaténation (`mic-cons`). Le résultat de cet opérateur est une variable multivaluée interne ;
- d'obtention du nième élément d'une liste. Le résultat de cet opérateur est une variable monovaluée interne ;
- d'obtention des n premiers (`mic-firstn`) ou n derniers éléments (`mic-lastn`) d'une liste. Le résultat de cet opérateur est une variable multivaluée interne.

Enfin, des contraintes de transformation permettent de passer d'une variable multivaluée de type liste à une variable de type ensemble (`mic-list-set`) ou bien d'une variable multivaluée (ensemble ou liste) à une variable monovaluée (`mic-multi-to-mono`) lorsque la valeur de la variable multivaluée est un ensemble ou une liste d'une seule valeur (qui devient donc valeur de la variable monovaluée produite).

Nous donnons dans le tableau 8.4 l'ensemble des contraintes pour variables multivaluées proposées par MICRO.

8.2.3.5 Manipulation et définition de contraintes

MICRO offre un certain nombre de fonctions pour les contraintes :
- une contrainte peut être créée soit en donnant un nom et une expression de contrainte principale, soit en donnant simplement l'expression principale, puisque le nom d'une contrainte MICRO est le nom d'une classe d'objets mais aussi le nom d'une fonction du langage hôte de MICRO qui se charge de l'instanciation de la classe de la contrainte ;
- une contrainte principale peut être détruite à partir de son nom ou de son expression ;
- les diverses informations relatives à une contrainte peuvent être accessibles à tout moment, telles que le domaine courant des variables ou le contexte de pose (*cf.* section 8.4) ;
- il est possible de connaître à tout moment la liste des contraintes posées. Afin que soit possible une suppression des instances de contraintes (principales) correspondant aux contraintes po-

Contraintes	Symbole	Nom MICRO
Principales	$=$	`mic-eq`
	$\neq$	`mic-neq`
	$\leftarrow$	`mic-assign`
	$\subset$	`mic-include`
	$\subseteq$	`mic-include-eq`
	$\not\subset$	`mic-not-include`
	$\not\subset$	`mic-not-include-eq`
	every $\forall$	`mic-every*`
	any $\exists$	`mic-any`
	all	`mic-all*`
	vérif 2 à 2	`mic-compar*`
Principales Mono-multivaluées	$\in$	`mic-in`
	$\notin$	`mic-not-in`
	at-least	`mic-at-least`
	at-most	`mic-at-most`
Secondaires Unaires	car	`mic-car`
	cdr	`mic-cdr`
	min	`mic-min`
	max	`mic-max`
	card	`mic-card`
	length	`mic-length`
	map	`mic-map*`
	apply	`mic-apply*`
	selection	`mic-select*`
Secondaires Binaires	$\cup$	`mic-union`
	$\cap$	`mic-inter`
	-	`mic-diff`
	append	`mic-append`
	application 2 à 2	`mic-combin*`
Secondaires Mono-multivaluées	cons	`mic-cons`
	nth	`mic-nth`
	firstn	`mic-firstn`
	lastn	`mic-lastn`
Secondaires de Transformation	liste $\longrightarrow$ set	`mic-list-to-set`
	multivaluée $\longrightarrow$ monovaluée	`mic-multi-to-mono`

TAB. 8.4 - : Ensemble des contraintes pour variables multivaluées proposées par MICRO. Les contraintes dont le nom est suivi d'un astérisque ont pour argument une fonction ou une contrainte.

sées, MICRO gère une *table de correspondance* entre les expressions de contraintes principales qui ont été posées et les instances correspondantes.

Les fonctions de manipulation des contraintes opèrent seulement sur les contraintes principales. Les contraintes secondaires sont toutes liées à une contrainte principale à partir de laquelle elles ont été créées. Nous détaillons le principe de création d'une contrainte à la section 8.4.

Dans MICRO de nouvelles contraintes peuvent être définies et viennent grossir l'ensemble des contraintes disponibles pendant une session. Une nouvelle contrainte est simplement une expression formée à partir de contraintes prédéfinies ou d'autres contraintes définies. Cette définition permet d'utiliser une expression contrainte plusieurs fois sans avoir à la décrire à chaque fois ; il suffit simplement de faire référence à la contrainte définie correspondante.

Une fonction de MICRO permet de définir une nouvelle contrainte. Elle a en argument le nom de la nouvelle contrainte, la liste des arguments et leur type, puis l'expression contrainte construite à partir de contraintes déjà disponibles.

Par exemple, la contrainte `mic-triang-eq` qui reflète une inégalité triangulaire a trois arguments x, y, z de type numérique et est définie par :

(`mic-create-constraint` `mic-triang-eq` (x y z) ((`mic-le` (`mic-power` z 2) (`mic-add` (`mic-`

`-power` *x 2*) (`mic-power` *y 2*))))).

Il faut noter que le terme de définition est un peu abusif, car la définition d'une contrainte par la donnée de ses règles de maintien de consistance n'est pas possible dans cette version de MICRO. Elle nécessiterait un langage de définition de contraintes permettant de décrire ces règles, ce que nous n'avons pas jugé primordial dans un premier temps, préférant produire un ensemble assez complet de contraintes prédéfinies.

8.2.4 Représentation des composants

MICRO a été écrit en langage LE-LISP V16 [Ilog92a]. Le module de programmation orientée-objet TELOS de ce langage a été mis à profit pour représenter les variables et les contraintes MICRO. Cette représentation permet de regrouper au sein d'une même entité les informations liées à une variable ou à une contrainte. L'héritage est utilisé pour factoriser des informations structurelles communes à des classes de variables ou de contraintes. La méthode associée à une classe de contraintes contient le code destiné au maintien de la consistance lié à la contrainte.

8.2.4.1 Représentation des variables

Il existe trois classes de variables correspondant aux variables monovaluées et aux deux types de variables multivaluées. La déclaration d'une VCM donne lieu à la création d'un objet TELOS de la classe de variable correspondante. Les attributs des classes TELOS permettent de stocker le nom et le domaine fournis par la définition (après normalisation). Ils permettent également de stocker :

- le nom de la variable ;
- le domaine *de définition* ou domaine *initial* de la variable qui est l'ensemble des valeurs proposées pour cette variable ;
- la liste des contraintes associées à la variable ;
- la liste des attributs d'instance de contraintes ou AIC associés à la variable ;
- le domaine *courant* ou domaine *effectif* de la variable qui est le domaine tel qu'il est réduit par le réseau de contraintes. Initialement, sa valeur est égale à celle du domaine de définition.

8.2.4.2 Représentation des contraintes

Les attributs d'une classe de contrainte décrivent les divers arguments et le résultat (pour les contraintes secondaires) des fonctions.

Une hiérarchie de contraintes a été élaborée sous la classe racine CONTRAINTES (*cf.* figure 8.2). Elle permet de factoriser les attributs communs aux classes et aux sous-classes de contraintes. Les feuilles de cette hiérarchie correspondent aux divers contraintes décrites ci-dessus. À chacune des classes feuilles est associée une méthode. Cette méthode est chargée de la maintenance de la consistance de la contrainte. Elle est activée à la pose de la contrainte et lors de chaque réactivation de la contrainte durant une phase de propagation conséquente à la modification du domaine d'un des attributs argument de la contrainte. L'effet de ces méthodes est décrit à la section 8.2.5.

Lorsqu'une contrainte est posée, elle prend la forme d'une instance de la classe de contrainte correspondante. Les contraintes portant sur les résultats des contraintes secondaires, sur les VCM ou sur des constantes, un attribut d'une instance TELOS de classe de contrainte ou AIC contient (*cf.* figure 8.3) :

- dans le cas où l'argument correspond à une contrainte secondaire, le domaine courant de la variable résultat interne à MICRO et un lien avec l'AIC correspondant à cette variable résultat interne ;
- dans le cas où l'argument correspond à une VCM, le domaine courant de cette VCM, c'est-à-dire un sous-ensemble du domaine de définition de la VCM, si celui-ci a été réduit par la propagation de contraintes, et un lien (pointeur) vers cette VCM (qui est une instance) ;

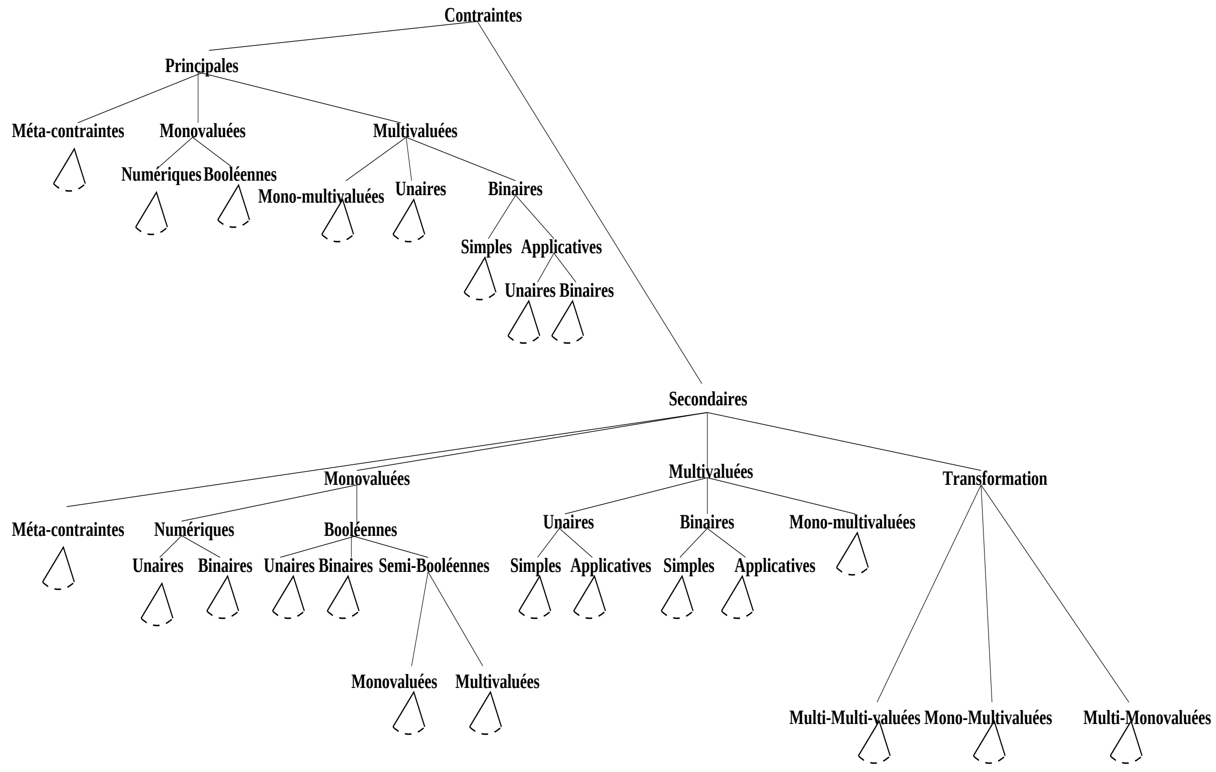


FIG. 8.2 - : La hiérarchie MICRO des classes de contraintes. Les classes feuilles (en pointillés) correspondent aux contraintes reportées dans les différents tableaux.

- dans le cas où l'argument correspond à une constante, la valeur de cette constante.

8.2.5 Règles de maintien de la consistance

Une méthode est associée à chaque classe feuille de la hiérarchie de contraintes, donc à chaque contrainte prédéfinie de MICRO, qu'elle soit principale ou secondaire. Cette méthode comporte un ensemble de règles de maintien de la consistance.

En général, pour chaque contrainte principale ou secondaire de MICRO, binaire ou ternaire, chaque variable dispose d'une règle de maintien de la consistance. La contrainte unidirectionnelle d'affectation `mic-assign` constitue une exception puisque seule la variable en partie gauche de l'affectation est calculée en fonction des variables apparaissant en partie droite de l'affectation.

8.2.5.1 Principe

Chacune des règles de consistance de la contrainte est activée lors de l'appel de la méthode. Chaque règle est destinée à calculer le domaine d'une variable (argument ou résultat) de la contrainte en fonction des domaines des autres variables de la contrainte et de l'ancien domaine associé à la variable traitée. Ce calcul fait intervenir la nature de la contrainte : l'opérateur qu'elle représente, qu'il soit de comparaison, arithmétique, booléen, ensembliste ou sur des listes.

Une fois ce calcul effectué, on obtient un domaine intermédiaire pour la variable qui est le plus grand domaine autorisé pour cette variable en fonction des domaines des autres variables de la contrainte. Ce domaine intermédiaire possède un degré de consistance inhérent aux propriétés de l'opérateur que représente la contrainte, aux types et aux domaines des arguments de la contrainte.

Chaque règle procède à l'intersection de ce domaine intermédiaire avec le domaine actuel de la variable :

- Si l'intersection est équivalente au domaine intermédiaire, c'est que la règle a opéré un fil-

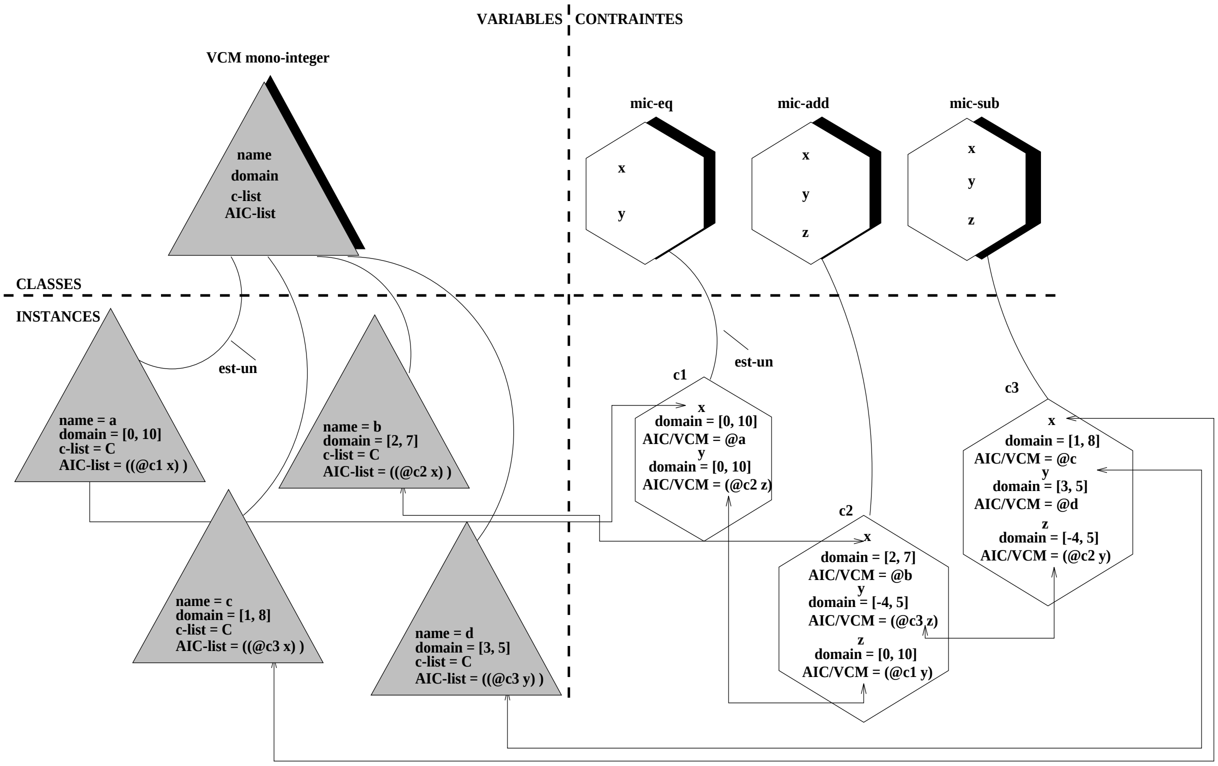


FIG. 8.3 - : Liens entre VCM et AIC (liens externes) et liens entre AIC (liens internes) pour la contrainte $C : a = b + (c - d)$ dont l'expression en MICRO est (`mic-eq` a (`mic-add` b (`mic-sub` c d))). Sont représentées ici les instances des quatre VCM a, b, c, d, les instances des trois contraintes (une principale et deux secondaires) qui forment la contrainte C, et les informations (domaine + lien) des VCM et des AIC.

trage sur le domaine. Le domaine intermédiaire devient le domaine actuel de la variable. La réduction de ce domaine doit être propagée aux autres contraintes qui impliquent la variable. Autrement dit, les méthodes de consistance associées à ces classes de contraintes vont être activées ; la propagation de contraintes se poursuit.

- Si l'intersection est équivalente au domaine actuel, c'est que celui-ci est déjà inclus dans le domaine intermédiaire. Ceci signifie que c'est le domaine de la variable qui, par sa modification, est à l'origine de l'activation de la méthode, ou que la modification à l'origine du déclenchement de la méthode est sans effet sur le domaine de la variable. Dans ce cas, le domaine est consistant vis-à-vis de la contrainte. Le domaine actuel de la variable n'est donc pas réduit, cette règle ne prolonge pas la propagation.
- Si l'intersection est vide, c'est que la contrainte ne peut pas être satisfaite. La propagation s'arrête. On peut conclure que l'événement à l'origine de l'activation de la méthode conduit à une inconsistance. Cet événement peut être l'ajout d'une contrainte, la modification d'une valeur ou d'un domaine de variable. Il faut prendre les mesures nécessaires à un retour vers un état consistant, c'est-à-dire l'état précédant le début de la propagation.

Nous donnons en annexe A.1 les règles associées à chacune des contraintes principales et secondaires.

- Pour les contraintes sur des variables monovaluées numériques, les règles de maintien de la consistance sont issues des règles de l'arithmétique des intervalles[3].
- Pour les contraintes sur des variables monovaluées booléennes, les règles de maintien de la consistance sont extraites des tableaux de valeurs des opérateurs booléens.
- Pour les contraintes sur des variables multivaluées, les règles de maintien de la consistance proviennent de la sémantique des opérateurs ensemblistes ou agissant sur des listes qui sont

[3] Toute énumération de valeurs étant transformée en unions d'intervalles disjoints.

traduits en contraintes.

8.2.5.2 Traitement des unions d'intervalles

MICRO se démarque des systèmes de programmation par contraintes à intervalles présentés en 4.5 par une gestion des unions d'intervalles. Nous expliquons ici son principe.

Soit $f(x)$ une fonction à un argument x. À x est associé le domaine de valeurs X sous la forme d'une union d'intervalles $X_1 \cup \ldots \cup X_n$. Soit $F(X)$, l'extension de $f(x)$ au domaine X. On a :
$\forall x \in X, f(x) \in F(X)$ d'après le théorème de l'arithmétique des intervalles. Si $x \in X_i \cup X_j$ alors $f(x) \in F(X_i \cup X_j)$
Or si $x \in X_i \cup X_j$ alors $x \in X_i$ ou $x \in X_j$
Si $x \in X_i$ alors $f(x) \in F(X_i)$ donc $f(x) \in F(X_i) \cup F(X_j)$
Si $x \in X_j$ alors $f(x) \in F(X_j)$ donc $f(x) \in F(X_i) \cup F(X_j)$
Donc $F(X_i \cup X_j) = F(X_i) \cup F(X_j)$

On peut généraliser ce résultat aux fonctions à n arguments. Dans le cas qui nous intéresse, les domaines intermédiaires sont calculés par des fonctions au plus binaires.

Soit une fonction binaire $f(x, y)$. À x est associé le domaine de valeurs X sous la forme d'une union d'intervalles $X_1 \cup \ldots \cup X_n$. À y est associé le domaine de valeurs Y sous la forme d'une union d'intervalles $Y_1 \cup \ldots \cup Y_m$. Soit $F(X, Y)$, l'extension de $f(x, y)$ aux domaines X et Y.

On a $F(X,Y) = \bigcup_{i=1,j=1}^{i=n,j=m} F(X_i, Y_j) = F(X_1, Y_1) \cup F(X_1, Y_2) \cup \ldots \cup F(X_1, Y_m) \cup \ldots \cup F(X_n, Y_1) \cup \ldots F(X_n, Y_m)$

$F(X, Y)$ est l'extension de l'application de F aux $n * m$ intervalles qui forment le produit cartésien de l'ensemble des intervalles de X et de l'ensemble des intervalles de Y.

Par exemple, la somme des unions d'intervalles $[-2, 5] \cup [8, 13]$ et $[-8, -6] \cup [3, 7]$ est équivalente à :
$([-2, 5] + [-8, -6]) \cup ([-2, 5] + [3, 7]) \cup ([8, 13] + [-8, -6]) \cup ([8, 13] + [3, 7])$, soit :
$[-10, -1] \cup [1, 12] \cup [0, 7] \cup [11, 20]$, soit :
$[-10, -1] \cup [0, 20]$

Ce principe est à la base du calcul du domaine intermédiaire lorsque celui-ci se fait à partir d'arguments qui ont pour domaine une union d'intervalles. Les contraintes étant au plus binaires, le domaine résultant est donc d'une taille de $O(m * n)$ dans le cas où les domaines des arguments utilisés ont une taille en $O(m)$ et $O(n)$ respectivement. Cet ordre de grandeur est généralement atténué par l'opération d'union d'intervalles qui procède à la fusion des intervalles non disjoints.

Le module MICRO dispose donc d'un ensemble d'opérations de manipulation d'intervalles (union, intersection, différence...) et est capable de distribuer l'application d'une fonction à une union d'intervalles, puis de procéder à une normalisation de chaque domaine modifié.

8.2.5.3 Divers degrés de consistance

Le filtrage opéré par une règle de maintien de la consistance dépend de la fonction utilisée pour calculer le domaine intermédiaire à partir des autres arguments ou du résultat de la contrainte et de la nature des domaines impliqués dans ce calcul.

Pour toutes les contraintes principales, secondaires unaires non périodiques et secondaires binaires portant sur des variables booléennes ou numériques, l'arc-consistance des domaines est garantie. Pour les variables de type réel représentées par des flottants, on peut atteindre jusqu'à l'intervalle-consistance (*cf.* section 4.5.5).

Les opérateurs périodiques comme *sin* et *cos* ont un traitement particulier. Le maintien d'une consistance d'arc sur les domaines des variables impliqués dans une telle contrainte peut entraîner une augmentation considérable et non forcément contrôlable des intervalles consistants [Lhomme93]. Aussi, lorsque l'étendue d'un domaine d'une variable impliquée dans une telle contrainte périodique est supérieure à la période (ici, 2π), aucun filtrage n'est opéré.

En ce qui concerne les contraintes portant sur des variables multivaluées, l'arc-consistance est garantie sur les extensions finies des variables multivaluées ainsi que sur les domaines finis de ces variables multivaluées. Lorsque le domaine des éléments de la variable est continu, les mêmes limitations sont rencontrées.

8.2.5.4 Gestion de bornes ouvertes

Pour les entiers et les réels, MICRO permet la gestion d'unions d'intervalles fermés et ouverts (pour les réels) à bornes finies ou infinies.

Dans le cas des entiers, les intervalles ouverts sont transformés en intervalles fermés. La distinction entre les bornes ouvertes et les bornes fermées sur les réels nécessite la prise en compte de la nature de la borne (ouverte ou fermée) dans les calculs d'intervalles.

Lors du calcul des bornes d'un intervalle, la première règle est que, lorsqu'une des bornes opérandes est ouverte, alors la borne résultat est ouverte aussi.

Cependant, cette seule règle n'est pas suffisante, comme le montre l'exemple suivant :

$$\begin{cases} c_1 : & X + Y = Z \\ c_2 : & X + Y = T \end{cases}$$

où $dom(X) = dom(Y) = [-2, 2.], dom(Z) = [2., 3.[, dom(T) =]3., 4]$

Il est évident que ce système de contraintes ne comporte pas de solutions. La seule propagation, basée sur l'application des règles de calcul de la nature des bornes, ne permet pas de conclure à l'absence de solution mais termine avec les domaines $dom(X) = dom(Y) =]1., 2.[$, $dom(Z) =]2.3.[$, $dom(T) =]3., 4.[$.

Cette insuffisance provient du fait qu'il n'a pas été tenu compte ici de la distance qui sépare la valeur approchée (lorsque la borne est ouverte) de la valeur exacte (lorsque la borne est fermée).

Afin de tenir compte de cette distance, toute borne d'intervalle est représentée par un doublet (v, d) ou v est la valeur et d la distance qui sépare la valeur v de la vraie valeur. Pour une borne fermée, la distance d est nulle. Pour une borne ouverte, elle peut être négative (il s'agit d'une borne ouverte supérieure) ou positive (il s'agit d'une borne ouverte inférieure).

Ainsi, l'intervalle $[2., 3.]$ est représenté par la paire de doublets $((2., 0)(3., 0))$, $]2., 3.]$ par $((2., d)(3., 0))$ avec $d > 0$, $[2., 3.[$ par $((2., 0)(3., d))$ avec $d < 0$, $]2., 3.[$ par $((2., d)(3., d'))$ avec $d > 0, d' = -d$.

Initialement, toute borne fermée est transformée en $(v, 0)$. Réciproquement, toute borne ouverte est transformée en (v, d) . La distance d peut être fixée par l'utilisateur et vaut par défaut 10^{-8}.

La solution qui consiste à remplacer tout intervalle ouvert par un intervalle fermé dont les bornes sont des valeurs arbitraires a été abandonnée car elle mène à des incohérences.

Par exemple, si l'intervalle ouvert $]2, 3]$ est remplacé par l'intervalle fermé $[2.00001, 3]$ alors, si on ajoute l'intervalle fermé $[0.99999, 0.99999]$, on obtient l'intervalle fermé $[3., 3.99999]$. Dans ce cas, il est difficile de reconnaître que cet intervalle correspond en fait à un intervalle ouvert. Par contre, avec la solution choisie, on obtient l'intervalle représenté par la paire $((2.99999, d), (3.99999, 0))$.

Avec l'introduction de cette distance d, les calculs pour évaluer les bornes des intervalles sont doublés. Il faut dans un premier temps, calculer la valeur résultante puis, dans un second temps, la distance résultante de l'application de l'opérateur. Le calcul de la distance est propre à chaque opérateur. Par exemple, la somme de $]2., 3.]$ représenté par $((2., d_1)(3., 0))$ et de $]4., 7.[$ représenté par $((4., d_2), (7., d_3))$ est $]6., 10[$ représenté par $((6., d_1 + d_2), (10., d_3))$.

De même, les comparaisons entre bornes se font à la fois sur les valeurs et sur les distances. Par exemple, $]2., 3.]$ représenté par $((2., d)(3., 0))$ est inclus dans $]2., 3.]$ représenté par $((2., d')(3., 0))$ si $d' < d$.

Si le temps de calcul des bornes d'intervalles est multiplié par deux, la capacité de détection d'inconsistance lors de la propagation des contraintes est augmentée. Ainsi, sur l'exemple précédent, l'échec est détecté.

En résumé, pour les flottants, un intervalle est représenté par $((v, d)(v', d'))$ avec, à tout moment,

$v+d \leq v'+d'$. De même, si on obtient la paire $((v,d)(v,d'))$ avec $d > d'$, alors l'intervalle représenté est vide.

Enfin, une précision p est considérée pour les calculs sur les domaines. Elle est, par défaut, fixée à 10^{-5} mais peut être contrôlée par l'utilisateur. Cette précision donne la plus petite étendue d'un intervalle. Dès que les deux bornes d'un intervalle sont placées à une distance inférieure à cette précision, on considère que l'intervalle ne comporte qu'une seule valeur, la valeur médiane des deux bornes. Cette précision intervient donc dans toutes les opérations sur les intervalles.

8.2.5.5 Gestion de bornes infinies

MICRO prend également en compte les bornes infinies $+\infty$ et $-\infty$. Pour chaque opérateur impliqué dans une contrainte, MICRO intègre les règles de calcul impliquant ces deux valeurs spéciales qui sont représentées en machine par le plus grand et le plus petit entier (respectivement flottant) représentable en machine. Lors du calcul des bornes des intervalles, ces règles sont appliquées pour chaque borne infinie rencontrée. Les cas d'erreurs où le résultat est indéfini sont pris en charge par MICRO. Nous donnons en annexe A.2 les règles de calcul avec bornes infinies.

8.3 Propagation et résolution

Nous présentons ici les algorithmes chargés de la propagation de contraintes et de la résolution des réseaux de contraintes dans MICRO.

8.3.1 La propagation

La propagation de contraintes est le processus qui répercute sur le réseau de contraintes, soit la modification du domaine d'une variable, soit l'ajout d'une contrainte.

Aussi, la propagation d'une contrainte a lieu à divers moments de la vie de la contrainte. À la pose de la contrainte, les méthodes de maintien de la consistance associée à la contrainte sont activées. Trois issues sont possibles :

- un des domaines des variables de la contrainte est vidé, le réseau est inconsistant (la contrainte est trop forte) ; la propagation s'arrête sur un échec ;
- l'application des méthodes de maintien de la consistance n'a aucun effet sur les domaines des variables de la contrainte ; la propagation s'arrête sur un succès pour cette contrainte ;
- au moins un des domaines des variables de la contrainte est réduit par l'application des règles de maintien de la consistance ; la propagation doit être poursuivie.

Dans le premier cas, la contrainte ne peut être acceptée dans ce réseau, MICRO revient alors à l'état précédant l'activation de la propagation de contraintes alors que la contrainte est refusée. Dans le deuxième cas, la contrainte est introduite dans le réseau, la propagation s'arrête. Dans le troisième cas, la propagation doit se poursuivre en activant les méthodes de maintien de la consistance des autres contraintes qui portent sur les domaines réduit par le filtrage opéré par les règles de consistance.

Comme on peut s'en douter, la propagation de contraintes est un processus récursif dont l'opération de base consiste à appliquer les règles de consistance associées à une contrainte. Si l'application de ces règles modifie un des domaines des variables de la contrainte alors les méthodes de consistance des contraintes impliquant les domaines modifiés – hormis la contrainte traitée – sont activées à leur tour.

La terminaison de la propagation est garantie dans le cas de domaines finis. Cependant, la terminaison sur un échec ou une réussite est tributaire des contraintes considérées (qui influent sur la rapidité de convergence vers la décision finale) et donc indirectement des capacités de la machine (pile d'exécution).

Pour les domaines infinis, le même constat peut être dressé même si la terminaison ne peut être théoriquement garantie. Lorsque la propagation ne termine pas dans les limites mémoire fixées par la machine, c'est qu'elle est confrontée à un réseau de contraintes cyclique. Dans ce cas, l'indécision, caractérisée ici par une pile d'exécution pleine, est considérée comme un échec de la propagation, et un échappement (récupération d'erreur) permet de revenir à l'état précédent.

```
Procédure propager-contrainte(X, D, C, c, propagees, reduits)
Entrées :
    X ensemble des variables du CSP
    D ensemble des domaines du CSP
    C ensemble des contraintes du CSP
    c ensemble des contraintes à propager
    propagees liste des contraintes propagées
    reduits liste des domaines réduits
début
    propagees ← propagees ∪ c
    % mise à jour de la liste des contraintes propagées %
    pour tout r ∈ règles(c) faire
    % toutes les règles de la contrainte sont appliquées %
        appliquer r(var(c))
        si ∃v ∈ var(c) t.q. dom(v) = ∅ alors
        % la règle a vidé un des domaines %
            echappement et restauration
            % la propagation se termine sur un échec, on restaure l'état précédent %
            % l'action à l'origine de la propagation (ajout de contrainte ou modification %
            % du domaine d'une variable) est refusée, l'état précédent (stable) est restauré %
        fin si
    fin pour
    pour tout v ∈ var(c) t.q. dom(v) réduit alors
    % pour toutes les variables de la contrainte dont le domaine a été réduit %
        reduits ← reduits ∪ dom(v)
        % mise à jour de la liste des contraintes propagées %
        pour tout c' ∈ contr(v) t.q. c' ∈ C et c' ≠ c faire
        % on propage chacune des contraintes impliquant cette variable, C exceptée %
            propager-contrainte(X, D, C, c', propagees, reduits)
        fin pour
    fin pour
fin
```

TAB. 8.5 - : Procédure de propagation d'une contrainte c dans un CSP (X, D, C). Cette procédure renvoie, pour les besoins de la résolution, l'ensemble des contraintes propagées et l'ensemble des domaines réduits.

Selon le même principe, la propagation d'une contrainte est lancée lorsque le domaine d'une de ses variables est modifié – c'est le cas, par exemple, lorsque la variable est affectée d'une valeur. Il sera donc fait appel à la même procédure de propagation.

Dans le cas où la modification d'une variable consiste simplement en une réduction de son domaine, elle est équivalente à l'ajout d'une contrainte (virtuelle) au réseau. Il suffit dans ce cas de lancer la propagation de toutes les contraintes qui portent sur la variable dont le domaine vient d'être réduit.

Si la propagation déclenchée par une modification du domaine d'une variable, et non pas par l'ajout d'une contrainte, échoue, la modification est refusée et MICRO revient à l'état précédant cette modification.

Le rôle de la propagation de contraintes est de rendre consistant le réseau de contraintes en réponse à une modification de celui-ci. La consistance établie dépend du degré du filtrage opéré et donc, comme nous l'avons vu, des contraintes, du type des variables et de la finitude de leur domaine. S'il est possible que la propagation révèle la solution d'un CSP en terminant sur une configuration où chaque variable du CSP ne possède qu'une seule valeur dans son domaine courant, son rôle n'est pas d'exhiber les solutions du CSP. Cette tâche est dévolue à un algorithme de résolution présenté ci-après.

8.3.2 La résolution

MICRO propose un algorithme de résolution de contraintes (*cf.* les différentes méthodes présentées en 4.4) capable à la fois de traiter les CSP dont les domaines sont finis et donnés sous forme d'énumération ou d'unions d'intervalles, et les CSP numériques dont les domaines sont infinis et donnés sous la forme d'unions d'intervalles (*cf.* tableau 8.6).

On peut donc distinguer deux parties dans cet algorithme :

1. La partie dédiée aux CSP à domaines finis qui s'appuie sur un algorithme de résolution semblable à celui de PECOS [Ilog92b] ou encore à l'algorithme *MAC* proposé par Sabin et Freuder [Sabin et al.94]. Lors de la pose ou du retrait d'une contrainte, l'arc-consistance est obtenue par les règles associées aux contraintes et activées lors de la propagation. La résolution en elle-même consiste en une énumération des domaines des variables du CSP. Lorsqu'une variable reçoit une valeur, le système réagit et propage les conséquences de cette modification, puis les conséquences de celle-ci et ainsi de suite. Aussi, le processus déclenché ici consiste en un *Full-Look-Ahead*. La résolution est paramétrable sur trois points :
   (a) La liste des variables du CSP à résoudre. Si cette liste est incomplète (il existe d'autres variables dans le réseau), elle est complétée par MICRO. Si la liste contient des variables de réseaux de contraintes distincts, les divers réseaux sont résolus.
   (b) La prochaine variable à choisir pour l'énumération. Par défaut, l'ordre donné par la liste des variables du CSP à résoudre est choisi. L'ordonnancement peut être statique ou dynamique (*first-fail-principle*) et divers critères d'ordonnancement des variables sont disponibles. Statiquement, les variables peuvent être préalablement ordonnées selon la largeur minimale, la cardinalité minimale, la taille des domaines. Ces critères peuvent être combinés pour départager en cas d'indécision. Dynamiquement, on peut avoir recours à un réarrangement selon le *first-fail-principle*.
   (c) La prochaine valeur à choisir pour affecter la variable en cours.

   Nous n'avons pas établi de critères d'ordonnancement des valeurs. Par défaut, sur les domaines ordonnés, l'ordre croissant est choisi ; sur les domaines non ordonnés l'ordre de définition est appliqué.
2. La partie dédiée aux CSP numériques à domaines infinis qui s'appuie sur un algorithme de résolution par division de domaines proche de celui proposé dans BNR-PROLOG [Older et al.90] ou encore dans ICSP [Hyvönen92]. Il consiste à diviser les domaines des variables du CSP considéré et de résoudre les sous-CSP résultants de ces divisions. Par défaut, un domaine est divisé par dichotomie, mais une autre forme de division est envisageable.

   Afin d'éviter une explosion combinatoire des sous-CSP résultants, il convient de choisir les variables à domaines infinis sur lesquelles va être opérée la division de domaine. Si le réseau est cyclique alors MICRO établit l'ensemble des variables coupe-cycles. Parmi ces variables, MICRO sélectionne la variable à domaine infini la plus contrainte afin de diviser son domaine et que les effets de cette division aient la plus forte probabilité d'avoir des répercussions sur les domaines des autres variables.

   Lorsque le CSP contient à la fois des variables à domaine fini et des variables à domaine infini, on opère d'abord un *Mixed-Full-Look-Ahead* qui consiste à tenter d'instancier d'abord les variables à domaine fini du CSP. Si l'instanciation des variables à domaine fini n'est pas suffisante pour déterminer les solutions du CSP, c'est qu'il reste des domaines infinis. Dans ce cas, on procède à la division de ces domaines, pour chaque sous-solution établie jusqu'alors sur les domaines finis. Comme précédemment, pour les réseaux cycliques, l'ensemble coupe-cycle est déterminé, la variable à domaine infini la plus contrainte étant choisie pour la division.

Les critères de terminaison du processus de division des domaines sont :

- d'une part, la précision avec laquelle sont effectués les calculs, de sorte que, lorsqu'un domaine a une étendue inférieure ou égale à celle-ci, on considère que la variable n'admet qu'une seule valeur possible ;
- d'autre part, si le filtrage de consistance locale opéré sur un sous-CSP ne réduit pas les domaines, alors on peut considérer que la division n'a pas eu d'effet et que les divisions à venir n'en n'auront pas non plus. On peut alors passer à une autre variable à domaine infini s'il en existe. La division est stoppée lorsqu'il n'y a plus de variables à domaine infini, ou lorsque la division de chaque domaine infini est sans conséquence sur le réseau des domaines.

La procédure *resoudre-CSP* est chargée de la résolution des CSP à domaines finis et/ou infinis. Elle délègue cependant le traitement des CSP à domaines finis à la procédure *Full-Look-Ahead* et celui des CSP à domaines finis et infinis à la procédure *Mixed-Full-Look-Ahead*.

Outre les conditions d'arrêt énoncées plus haut, la division de domaine peut être stoppée si elle est jugée non "rentable". Le principe appliqué ici est identique au *Branch and Bound* : la branche explorée par division est donc abandonnée, si elle n'est pas jugée prometteuse. La rentabilité de la poursuite d'une division est évaluée par la fonction *calcul-distance*. La mesure choisie est la somme de fractions de type x/y où x est la variable dont le domaine est en cours de division et y une variable dont le domaine a été réduit par la division de x.

L'hypothèse faite ici est que, si lors de deux divisions successives, le même ensemble de variables est atteint par la propagation de cette division et que les réductions de domaine opérées sont proportionnelles à la division, alors on peut considérer que diviser davantage est inutile et qu'un ensemble de solutions est déjà contenu dans les domaines courants. On peut alors passer à une autre variable à domaine infini.

Procédure resoudre-CSP($X, D, C, inertes, old\text{-}dist, old\text{-}propagees, old\text{-}reduits$)
Entrées :
  X ensemble des variables du CSP
  D ensemble des domaines du CSP
  C ensemble des contraintes du CSP
  $inertes$ ensemble des variables inertes
  $old - dist$ distance courante
  $old - propagees$ liste des contraintes propagées
  $old - reduits$ liste des domaines réduits
début
  $X_{inf} \leftarrow \{x \in X$ t.q. $dom(x)$ infini $\}$
  % calcul de l'ensemble des variables à domaine infini %
  si $X_{inf} = \emptyset$ **alors**
    % il n'y a pas de variable à domaine infini %
    $s \leftarrow \emptyset$
    % initialisation de la solution courante %
    $S \leftarrow \emptyset$
    % initialisation de l'ensemble des solutions %
    Full-Look-Ahead(X, D, C, s, S)
    % application d'un Full-Look-Ahead %
    imprimer(S)
    % impression de l'ensemble des solutions %
  sinon
    % le CSP comporte des variables à domaine infini %
    si $X_{inf} \neq X$ **alors**
      % le CSP comporte également des variables à domaine fini %
      $X_{fini} \leftarrow X - X_{inf}$

% on détermine l'ensemble des variables à domaine fini %
$s \leftarrow \emptyset$
% initialisation de la solution courante %
$S \leftarrow \emptyset$
% initialisation de l'ensemble des solutions %
Mixed-Full-Look-Ahead($X_{fini}, X, D, C, inertes, old\text{-}dist, old\text{-}propagees, old\text{-}reduits, s, S$)
% application d'un Full-Look-Ahead sur l'ensemble des variables à domaine fini %
imprimer(S) % impression de l'ensemble des solutions %

sinon
% le CSP n'a que des variables à domaine infini %
$CC \leftarrow$ coupe-cycle(X, C)
% on teste s'il comporte des cycles %
si $CC \neq \emptyset$ **et** $\exists y \in CC$ t.q. $y \notin$ inertes **alors**
% le CSP comporte des cycles et des variables coupe-cycle non inertes %
choisir $y \in CC$ t.q. $y \notin$ inertes $\wedge y \in X_{inf}$ $\wedge$ y a la plus grande largeur
% on choisit parmi les variables coupe-cycle non inertes la plus contrainte %
$D' \leftarrow$ diviser($dom(y)$)
% on divise son domaine %
sinon
% le CSP n'a pas de cycle % choisir $y \in X_{inf}$ t.q $y \notin$ inertes $\wedge$ y a la plus grande largeur
% on choisit parmi les variables non inertes la plus contrainte %
$D' \leftarrow$ diviser($dom(y)$)
finsi
si $D' = \emptyset$ **ou** $\forall d \in D', \mid d \mid < precision$ **alors**
% s'il n'y a pas de domaine à diviser ou si chaque division de domaine a une étendue inférieure à l'étendue minimale %
imprimer(D)
% l'ensemble des domaines est imprimé comme solution %
finsi
sauver-domaines(D)
% l'ensemble des domaines courants du CSP est sauvegardé %
pour tout $d \in D'$ t.q $\mid d \mid \geq precision$ **faire**
% pour chaque division %
si consistance-locale($X, (D - dom(y)) \cup d, C, contr(y), propagees, reduits$) **alors**
% on teste la consistance locale du CSP où le domaine de la variable choisie est remplacé par la division %
si $reduits = \emptyset$ **alors**
% il n'y a pas eu de réduction suite à la propagation %
inertes = inertes ∪ y
% la variable est donc inerte %
sinon
% il n'y a eu une réduction suite à la propagation %
inertes ← inertes - (reduits ∩ inertes)
% des variables inertes ont peut être fait l'objet d'une réduction %
$new\text{-}dist \leftarrow$ calcul-distance($y, X, ((D - dom(y)) \cup d) \cap reduits$)
% on calcule la nouvelle distance du CSP %
si $propagees = old\text{-}propagees$ **et** $reduits = old\text{-}reduits$ **alors**
% si les mêmes contraintes ont été propagées et les mêmes domaines réduits %
si $new\text{-}dist < old\text{-}dist$ **alors**
% on ne résout le sous-CSP que si sa distance est inférieure à l'ancienne %
resoudre-CSP($X, ((D - dom(y)) \cup d) \cap reduits, C,$
$inertes, new\text{-}dist, propagees, reduits$)
finsi
sinon
% si d'autres contraintes ont été propagées ou d'autres domaines réduits, on résout le sous CSP %
resoudre-CSP($X, ((D - dom(y)) \cup d) \cap reduits, C,$
$inertes, new\text{-}dist, propagees, reduits$)
finsi
finsi
finsi

fin pour restaurer-domaines(D)
% afin de poursuivre, les domaines sont restaurés %
finsi
finsi
fin

TAB. 8.6 - : Algorithme de résolution de contraintes de MICRO

La recherche d'un coupe-cycle dans le (hyper)graphe associé au CSP est effectuée par la procédure *coupe-cycle* (*cf.* annexe C, tableau C.1). Le principe est de tester, pour chaque sommet, si sa suppression ainsi que celles de toutes ses arêtes (ici ses contraintes), rend le graphe acyclique.

Le graphe est sans cycle, si on ne peut à partir d'un sommet trouver un chemin sur les contraintes qui mène à ce même sommet. La détection d'un cycle se fait par propagation de marques (*cf.* annexe C, tableau C.3) sur les sommets rencontrés.

La fonction *calcul-distance* (*cf.* tableau 8.7) permet de donner un poids à un nœud de l'arbre des divisions. Ce poids est la somme des rapports entre l'étendue du domaine de la variable dont le domaine est divisé et l'étendue de chaque variable dont le domaine a été réduit par cette division. Si pour deux nœuds successifs, ce poids et l'ensemble des domaines réduits sont identiques, alors cela signifie que la réduction du domaine divisé est proportionnelle à la réduction des autres variables. La propagation atteint les mêmes contraintes et les domaines sont modifiés proportionnellement à la division opérée. Aussi, on peut arrêter de diviser le domaine de x.

Fonction calcul-distance(x, X, D) : réel
début

$$\text{calcul-distance} \leftarrow \sum_{y \in X \wedge y \neq x} \mid D.dom(x) \mid / \mid D.dom(y) \mid$$

fin

TAB. 8.7 - : Fonction d'évaluation de la distance associée à une division de domaine.

La procédure *Full-Look-Ahead* (*cf.* annexe C, tableau C.4) n'est autre que la procédure classique proposée par [Haralick et al.80] mais adaptée à la consistance locale réalisée par la propagation de contraintes sur les intervalles de MICRO. Ceci explique, notamment, la sauvegarde et la restauration des domaines pour chaque instanciation d'une variable (réduction à un singleton du domaine associé).

La procédure *Mixed-Full-Look-Ahead* (*cf.* tableau 8.8) tente d'abord d'instancier les variables à domaine fini du CSP. Lors de cette phase, si des domaines infinis deviennent finis, ils sont à leur tour candidats à une instanciation. À l'issue de cette résolution lancée sur les variables à domaine fini, s'il reste des variables à domaine infini, celles-ci sont traitées par la procédure *résoudre-infinis*. Lorsqu'une solution est partiellement atteinte (tous les domaines finis sont réduits à des singletons), alors c'est à la procédure *résoudre-infinis* de la compléter en divisant les domaines infinis s'il y en a.

Procédure Mixed-Full-Look-Ahead($Y, X, D, C, inertes, old\text{-}dist, old\text{-}propagees, old\text{-}reduits, s, S$)
Entrées :
Y ensemble des variables à domaine fini du CSP
X ensemble des variables du CSP
D ensemble des domaines du CSP
C ensemble des contraintes du CSP
$inertes$ ensemble des variables inertes
$old - dist$ distance courante
$old - propagees$ liste des contraintes propagées
$old - reduits$ liste des domaines réduits
s solution courante
S ensemble des solutions

début
  si $Y = \emptyset$ **alors**
  % il n'y a plus de variables à domaine fini %
    resoudre-infinis($X, D, C, inertes, old\text{-}dist, old\text{-}propagees, old\text{-}reduits, s, S$)
    % on resout le sous-CSP à domaines infinis restant %
    $S \leftarrow S \cup s$
    % la solution est ajoutée à l'ensemble %
  sinon
  % il reste des variables à domaine fini %
    $y \leftarrow$ premier(Y)
    % on prend la première d'entre elles %
    répeter
      $v \leftarrow$ premier($dom(y)$)
      % on prend la première valeur de son domaine %
      $dom(y) \leftarrow dom(y) - v$
      % qu'on retire de ce domaine %
      sauver-domaines(D)
      % l'ensemble des domaines courants du CSP est sauvegardé %
      si consistance-locale($X, (D - dom(y)) \cup \{v\}, C, contr(y), propagees, reduits$) **alors**
      % la propagation lancée sur le CSP où la variable a la valeur choisie a réussi%
        $s \leftarrow s \cup (y, v)$
        % la solution courante est mise à jour %
        $newY \leftarrow \{x$ t.q. $dom(x) \in reduits \wedge dom(x)$ fini $\}$
        % on calcule l'ensemble des variables à domaine fini du CSP après cette propagation %
        Mixed-Full-Look-Ahead($(Y - y) \cup \{newY\}, X,$
          $((D - dom(y)) \cup \{v\}) \cap reduits,$
            $C, inertes, old\text{-}dist, propagees, reduits, s, S$)
        % on rappelle la résolution du CSP mixed sur la configuration courante %
        $s \leftarrow s - (y, v)$
        % on met à jour la solution courante %
      finsi
      restaurer-domaines(D)
      % afin de poursuivre avec une autre valeur, les domaines sont restaurés %
    jusqu'à $dom(y) = \emptyset$
    % on s'arrête lorsque toutes les valeurs de la variable ont été essayées %
  finsi
fin

TAB. 8.8 - : Le Mixed-Full-Look-Ahead créé pour MICRO.

La procédure *résoudre-infinis* (*cf.* tableau 8.9) reprend le même principe de division que celui appliqué dans *résoudre-CSP* pour des CSP à domaines infinis uniquement. Si un domaine infini devient fini, alors il est passé pour instanciation à la procédure *Mixed-Full-Look-Ahead*. Les procédures *Mixed-Full-Look-Ahead* et *résoudre-infinis* construisent et se transmettent progressivement les ensembles de solutions.

Procédure resoudre-infinis($X, D, C, inertes, old\text{-}dist, old\text{-}propagees, old\text{-}reduits, s, S$)
Entrées :
  X ensemble des variables du CSP
  D ensemble des domaines du CSP
  C ensemble des contraintes du CSP
  $inertes$ ensemble des variables inertes
  $old - dist$ distance courante
  $old - propagees$ liste des contraintes propagées
  $old - reduits$ liste des domaines réduits
  s solution courante
  S ensemble des solutions
début
  $X_{inf} \leftarrow \{x \in X$ t.q. $dom(x)$ infini $\}$
  % on calcule l'ensemble des variables à domaine infini %
  $CC \leftarrow$ coupe-cycle(X, C)

% on calcule l'ensemble coupe-cycle du graphe associé au CSP %
si $CC \neq \emptyset$ **et** $\exists y \in CC$ t.q. $y \notin$ inertes **alors**
% le CSP comporte des cycles et des variables coupe-cycle non inertes %
    choisir $y \in CC$ t.q. $y \notin$ inertes $\wedge y \in X_{inf} \wedge y$ a la plus grande largeur
    % on choisit, parmi les variables coupe-cycle non inertes, la plus contrainte %
    $D' \leftarrow$ diviser($dom(y)$)
    % on divise son domaine %
sinon
    % le CSP n'a pas de cycle %
    choisir $y \in X_{inf}$ t.q $y \notin$ inertes $\wedge$ y a la plus grande largeur
    % on choisit parmi les variables non inertes la plus contrainte %
    $D' \leftarrow$ diviser($dom(y)$)
finsi
si $D' = \emptyset$ **ou** $\forall d \in D', \mid d \mid < precision$ **alors**
% s'il n'y a pas de domaine à diviser ou % %
si chaque division de domaine a une étendue inférieure à l'étendue minimale %
    imprimer(D)
    % l'ensemble des domaines est imprimé comme solution %
finsi
sauver-domaines(D)
% l'ensemble des domaines courants du CSP est sauvegardé %
pour tout $d \in D'$ t.q $\mid d \mid \geq precision$ **faire**
    % pour chaque division %
    si consistance-locale($X, (D - dom(y)) \cup d, C, contr(y), propagees, reduits$) **alors**
        % on teste la consistance locale du CSP où le domaine de la variable choisie %
        % est replacée par la division %
        si $reduits = \emptyset$ **alors**
            % il n'y a pas eu de réduction suite à la propagation %
            inertes = inertes ∪ y
            % la variable est donc inerte %
        sinon
        % la propagation lancée a réduit des domaines %
            $new\text{-}dist \leftarrow$ calcul-distance($y, X, ((D - dom(y)) \cup d) \cap reduits$)
            % on calcule la distance associée à cette division %
            $newY \leftarrow \{x$ t.q. $dom(x) \in reduits \wedge dom(x)$ fini $\}$
            % la propagation ayant pu le modifier, on calcule l'ensemble des variables à domaine fini %
            si $propagees = old\text{-}propagees$ **et** $reduits = old\text{-}reduits$ **alors**
            % la propagation a propagé les mêmes contraintes et réduit les mêmes domaines %
                si $new\text{-}dist < old\text{-}dist$ **alors**
                % la nouvelle distance est inférieure à l'ancienne %
                    si $newY \neq \emptyset$ **alors**
                    % des variables à domaine fini sont apparues après la propagation %
                        Mixed-Full-Look-Ahead($newY, X, ((D - dom(y)) \cup d) \cap reduits, C, inertes$
                                        $new\text{-}dist, propagees, reduits, s, S$)
                        % on lance un Mixed-Full-Look-Ahead %
                    sinon
                    % il n'y a pas de variables à domaine fini %
                        resoudre-infinis($X, ((D - dom(y)) \cup d) \cap reduits, C, inertes$
                                        $new\text{-}dist, propagees, reduits, s, S$)
                        % on rappelle la procédure sur le sous-CSP où la division a été intégrée %
                    finsi
                finsi
            sinon
            % la propagation est différente de celle lancée sur le sur-CSP %
                si $newY \neq \emptyset$ **alors**
                % des variables à domaine fini sont apparues après la propagation %
                    Mixed-Full-Look-Ahead($newY, X, ((D - dom(y)) \cup d) \cap reduits, C, inertes$
                                        $new\text{-}dist, propagees, reduits, s, S$)
                    % on lance un Mixed-Full-Look-Ahead %
                sinon
                % il n'y a pas de variables à domaine fini %
                    resoudre-infinis($X, ((D - dom(y)) \cup d) \cap reduits, C, inertes$

new-dist, *propagees*, *reduits*, s, S)
% on rappelle la procédure sur le sous-CSP où la division a été intégrée %
finsi
finsi
finsi
finsi
fin pour restaurer-domaines(D)
% afin de poursuivre, les domaines sont restaurés %
fin

TAB. 8.9 - : La procédure de résolution pour les CSP à domaines infinis.

La fonction *premier* rend le premier élément de la liste argument. Il n'a pas été question ici d'ordonnancement dynamique – un ordonnancement statique étant supposé fait avant le premier appel de *résoudre-CSP*) – de ces listes, mais c'est ici qu'il peut être réalisé.

La fonction *consistance-locale* (*cf.* annexe C, tableau C.5) lance la propagation de plusieurs contraintes du CSP. Elle fait donc appel à la procédure *propager-contraintes* présentée à la section précédente.

La résolution est lancée à partir d'un ensemble de VCM X qui définit un ensemble D des domaines des VCM de X et un ensemble C des contraintes portant sur les variables de X. Il est possible que le triplet (X, D, C) détermine en fait plusieurs CSP (X_i, D_i, C_i) où $X_i \subset X$, $D_i \subset D$ et $C_i \subset C$. Dans ce cas, on résout chacun des (X_i, D_i, C_i) séparément.

Le résultat de la résolution est un ensemble de solutions (éventuellement vide) où une solution est donnée sous la forme de n couples (x_j, d_j) $(\forall j \in [1, n])$ où n est le nombre de variables du CSP traité, x_j une variable de ce CSP et d_j son domaine associé donné sous forme d'union d'intervalles : d_j peut être une valeur ou un ensemble de valeurs.

Si du point de vue algorithmique, l'algorithme proposé n'apporte aucun principe nouveau (il se base sur des techniques éprouvées et reconnues), son originalité réside essentiellement dans la prise en compte simultanée de CSP à domaines finis et infinis et dans la proposition d'un critère d'évaluation de la pertinence de la poursuite d'une division de domaine dans le cas de CSP à domaines infinis.

8.4 La dynamicité des CSP Micro

La dynamicité des CSP gérés par MICRO est reflétée par trois actions qui peuvent être réalisées à n'importe quel moment dans un CSP :

- l'ajout d'une contrainte est pris en charge par la procédure *propager-contrainte* présentée à la section 8.3.1. Cette opération modifie le graphe associé au CSP et nous nous proposons de montrer comment est étendu un réseau de contraintes MICRO.
- la suppression d'une contrainte consiste pour MICRO à amputer le CSP d'une contrainte, puis à atteindre l'état de stabilité qui aurait été atteint si toutes les contraintes du CSP avait été posées sans celle supprimée. L'objectif est d'atteindre cet état par un minimum de perturbations du réseau actuel.
- la modification du domaine d'une VCM ne modifie pas le graphe associé au CSP, mais nécessite la propagation de toutes les contraintes impliquant cette variable pour tester la consistance du nouveau CSP.

8.4.1 Création d'une contrainte

Une expression de contrainte est construite à partir d'une contrainte principale dont les arguments sont, soit des constantes, soit des VCM, soit des contraintes secondaires. À son tour, une contrainte secondaire a pour arguments; soit des constantes, soit des VCM, soit des contraintes secondaires.

Dans MICRO, l'ajout d'une contrainte dans un réseau revient à l'instanciation de la classe de contrainte principale équivalente. Celle-ci déclenche alors en cascade l'instanciation des classes de contraintes correspondant aux contraintes secondaires contenues par la contrainte principale.

Exemple 9 *Soit la contrainte* $a = b + (c - d)$*, elle est représentée en* MICRO *par l'expression contrainte (`mic-eq` a (`mic-add` b (`mic-sub` c d)))*

Sur l'exemple, trois instances de contraintes sont créées. Ces instances sont liées entre elles par des liens AIC/AIC et liées aux VCM par des liens AIC/VCM.

La création d'une contrainte est progressive et se fait par lecture de l'expression contrainte équivalente (sur l'exemple, sept pas de lecture sont nécessaires). Les attributs de l'instance de contrainte sont déterminés les uns après les autres lors de la lecture. Ainsi, pour chaque AIC se met en place son lien avec un autre AIC ou une VCM. Seuls les AIC liés à une VCM ont un domaine déterminé : celui de la VCM. Une fois que chaque AIC de l'instance de contrainte est déterminé – la construction de l'instance est achevée –, la propagation de la contrainte est lancée : les règles de maintien de la consistance sont activées. Une fois que l'expression de la contrainte principale est parcourue, la propagation déclenchée fournit un domaine aux AIC qui n'en n'avaient pas encore.

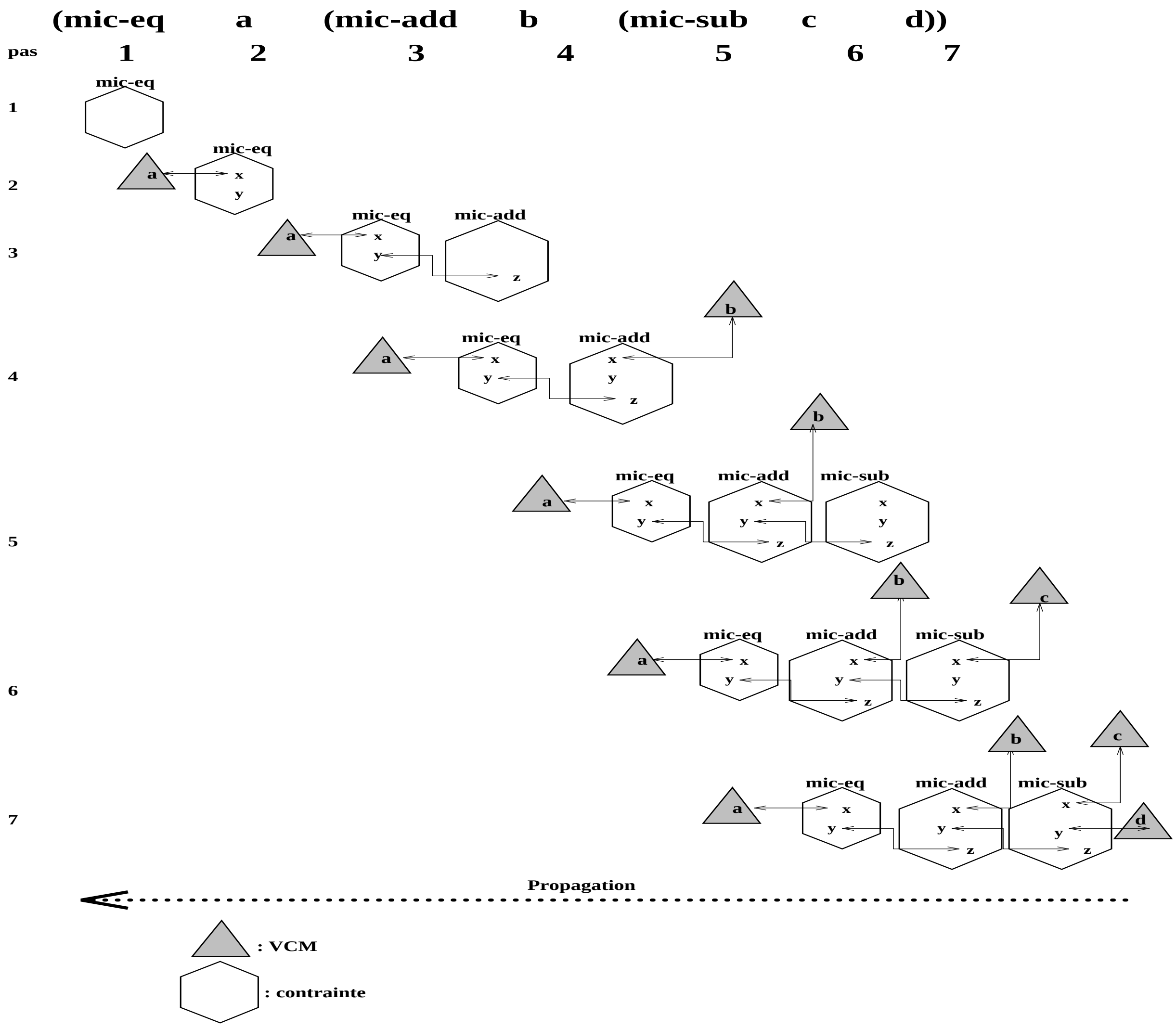


FIG. 8.4 - : Création d'une contrainte dans MICRO : la contrainte (`mic-eq` a (`mic-add` b (`mic-sub` c d))).

8.4.2 Retrait d'une contrainte

Les algorithmes d'arc-consistance étudiés en 4.6 sont adaptés à des CSP à domaines finis pour lesquels il est donc envisageable, soit de maintenir l'ensemble des supports pour chaque valeur dans les domaines, soit de reconstituer l'ensemble des valeurs supprimées. L'une des tâches que doit réaliser MICRO est de gérer des domaines continus et infinis. Aussi, les solutions précédentes n'ont pas été retenues. La gestion des retraits de contraintes dans MICRO (*cf.* tableaux 8.10 et 8.12) reprend l'idée intuitive de Janssen de conserver une photographie de l'état du réseau au moment de la pose de la contrainte et de repartir de cette photographie lors de la suppression de la contrainte pour atteindre le nouvel état de stabilité.

Au moment de la pose d'une contrainte, le module sauve dans l'instance de la contrainte son *contexte de pose*. Ce contexte est l'ensemble des paires *(VCM , dom(VCM))* des variables VCM qui se trouvent dans le réseau au moment de la pose et dont le domaine courant est *dom(VCM)*. À chaque instance de contrainte est associé un contexte de pose.

La représentation de ce contexte de pose pourrait être optimisée. On pourrait ainsi ne conserver que des contextes de pose minimaux en taille ne contenant que les variables dont le domaine a été réduit par la pose de la contrainte. La construction du contexte de pose complet (toutes les VCM du réseau retrouvent un domaine) nécessiterait alors un parcours des contextes de pose minimaux des contraintes posées avant. Nous n'avons pas fait ce choix dans l'implémentation actuelle de MICRO et le contexte de pose d'une contrainte n'est donc pas optimisé.

De même, on peut remarquer que si la contrainte n'a pas déclenché de propagation au moment de sa pose ni après (ce qui signifie qu'elle n'a pas été utilisée dans une propagation ultérieure), alors on peut faire l'économie de reposer toutes les contraintes ultérieures. En effet, vis-à-vis du réseau, la contrainte peut être considérée comme inactive. Cette considération n'est pas prise en compte par les algorithmes présentés ici.

Procédure supprimer-contrainte(c)
début
  restaurer-contexte(c)
  % on restaure le contexte de pose de la contrainte %
  enlever-contrainte(c)
  % la contrainte est physiquement supprimée du réseau %
  $C \leftarrow$ antérieures(c)
  % la propagation débute sur le réseau réduit aux contraintes antérieures %
  $X \leftarrow$ variables(C)
  $D \leftarrow$ domaines(X)
  c.contexte $\leftarrow$ construire-contexte(X, D)
  % le contexte de pose de la contrainte est mis à jour %
  propager-contrainte($X, D, C, c, \emptyset, \emptyset$)
  % on propage la contrainte %
  $C \leftarrow C \cup c$
  $X \leftarrow X \cup$ variables(c)
  $D \leftarrow D \cup$ domaines(c)
  % mise à jour du CSP courant %**pour tout** c' t.q même-réseau(c, c') **et** après(c, c') **faire**
  % on propage chaque contrainte postérieure à la contrainte supprimée et de même réseau %

    c'.contexte $\leftarrow$ construire-contexte(X, D)
    % le contexte de pose de la contrainte est mis à jour %
    propager-contrainte($X, D, C, c', \emptyset, \emptyset$)
    % on propage la contrainte %
    $C \leftarrow C \cup c'$
    $X \leftarrow X \cup$ variables(c')

$D \leftarrow D \cup$ domaines(c')
% mise à jour du CSP courant %
fin pour
fin

TAB. 8.10 - : La procédure *supprimer-contrainte* est chargée de la suppression d'une contrainte qui consiste à restaurer le contexte de pose, à supprimer la contrainte du réseau et à repropager chaque contrainte posée ultérieurement.

La suppression d'une contrainte (*cf.* tableau 8.10) opère en trois temps :

1. D'abord le contexte de pose de la contrainte est restauré (*cf.* procédure *restaurer-contexte*). Les VCM qui ont une valeur fixée (et non inférée par les contraintes) gardent cette valeur. Les VCM présentes au moment de la pose de la contrainte adoptent pour domaine courant le domaine qu'elles avaient alors. Les VCM qui ont été introduites par les contraintes postérieures retrouvent, quant à elles, leur domaine de définition.
Dans le réseau des instances de contraintes, les AIC prennent le même domaine que les VCM auxquelles ils sont liés. En revanche, les domaines des autres AIC (AIC internes liés à d'autres AIC) ne sont plus cohérents avec les changements opérés sur les VCM du réseau. Aussi, on procède (*cf.* procédure *propager-restauration*, tableau 8.12) à une propagation dans le réseau d'instances de contraintes qui vise à attribuer aux AIC internes un domaine indéfini que les propagations des contraintes ultérieures définiront à nouveau.

Procédure restaurer-contexte(c : contrainte)
début
pour tout $var \in$ réseau(c) **faire**
% pour toutes les variables du réseau actuel de la contrainte à supprimer %
si $val(var)$ indéfinie **alors**
% si la VCM n'a pas de valeur %

si $var \in contexte(c)$ **alors**
% si la variable était présente au moment de la pose de la contrainte %
$var.dom \leftarrow$ domaine-contexte(contexte(c), var)
% on lui restitue son domaine courant au moment de la pose %
sinon
% la variable était absente au moment de la pose de la contrainte %
$var.dom \leftarrow var.dom - init$
% son domaine courant redevient son domaine initial % **fin si**
fin si
pour tout $AIC \in$ liste-aic(var) **faire**
% pour chaque AIC correspondant à la variable %
$AIC.dom \leftarrow dom(var)$
% l'AIC reçoit le domaine de la VCM %
$propager - restauration(AIC, \emptyset)$
% invalidation des domaines des AIC internes %
fin pour
fin pour
fin

TAB. 8.11 - : La procédure *restaurer-contexte* nécessite de propager la restauration du domaine de la VCM vers les AIC correspondant puis, de là vers les AIC internes au réseau.

2. Puis, l'instance de contrainte principale correspondant à la contrainte à supprimer est détruite. La suppression d'une contrainte entraîne, outre une mise à jour des listes de liens VCM/AIC des variables impliquées par la contrainte, la destruction physique des instances de contraintes principales (et, éventuellement les instances des contraintes secondaires qui la composent) qui représentent cette contrainte. La base des instances de contraintes de MICRO est donc libérée de la contrainte. Cette opération utilise la table de correspondance entre les contraintes principales déclarées et posées et les instances de contraintes équivalentes.

3. Enfin, les contraintes postérieures à la contrainte supprimée et appartenant au même réseau sont propagées : leurs règles de maintien de la consistance sont activées. La propagation doit simuler une pose de contraintes. C'est pourquoi pour chacune des contraintes postérieures considérées, elle doit être circonscrite à l'ensemble des contraintes qui lui sont antérieures. De même, le contexte de pose de ces contraintes doit tenir compte de la suppression de la contrainte et doit être calculé.

Ainsi, on garantit que le système repart d'une configuration du CSP identique à celle présente au moment de la pose de la contrainte supprimée. Il n'y a plus qu'à poser les contraintes suivantes.

Cette technique impose de conserver la liste des contraintes du réseau. MICRO conserve la liste des contraintes définies mais également la liste des contraintes de chaque réseau de contraintes établi. De cette façon, il est possible de retrouver le réseau auquel appartient la contrainte et la liste des contraintes posées ultérieurement dans le même réseau.

Procédure propager-restauration(AIC : attribut, VISIT : liste des instances de contraintes visitées)
début
  $VISIT \leftarrow VISIT \cup inst(AIC)$
  % l'instance de contrainte de l'AIC est marquée comme visitée %
  pour tout $AIC \in inst(AIC)$ **faire**
  % pour chaque AIC de cette instance de contrainte %

    si $AIC.lien \neq \emptyset$ **alors**
    % si l'AIC ne correspond pas à une constante %
      si $type(AIC.lien) = AIC$ **alors**
      % si l'AIC est lié à un autre AIC : c'est un AIC interne%

        $AIC.lien.dom \leftarrow$ indéfini
        % son domaine devient indéfini %
        si $instance(AIC.lien) \notin VISIT$ **alors**
          % si l'instance de l'AIC auquel l'AIC est lié n'a pas été visitée %
          $propager - restauration(lien(AIC), VISIT)$
          % on propage la restauration %
        fin si
      fin si
    fin si
  fin pour
fin

TAB. 8.12 - : La procédure *propager-restauration* est chargée de la mise à jour du domaine des AIC internes (liés à d'autres AIC) au réseau de la contrainte à supprimer. Le domaine de ces AIC est déclaré indéfini et sera déterminé par les propagations des contraintes postérieures.

Le principe de l'algorithme de retrait de contrainte est simple mais on peut lui reprocher au moins deux désagréments.

- plus la contrainte à supprimer est vieille – il s'agit de l'une des premières contraintes déclarées dans le réseau – moins le gain de la procédure *supprimer-contrainte* est important par rapport à une redéfinition complète du CSP amputé de la contrainte. La seule économie réside alors dans la non redéfinition des instances de contraintes et des liens qui forment le réseau. La rapidité de la transition du CSP vers le prochain état de stabilité est donc dépendante de l'âge de la contrainte supprimée.
- les contextes sauvegardés au niveau des instances de contrainte sont gourmands en place mémoire. Leur taille dépend du nombre de variables du réseau et de la taille du domaine de ces variables. Ainsi, si n est le nombre de variables présentes dans le réseau au moment de la pose de la contrainte, si d est la taille de la représentation du plus grand domaine, un contexte de pose a une taille de l'ordre de $O(nd)$.

  S'il est possible d'optimiser la taille du contexte de pose et donc, de préserver l'espace mémoire, par un stockage de contextes de pose minimaux par exemple, c'est au prix d'un parcours des contraintes antérieures qui peut s'avérer coûteux. Aussi, nos avons préféré privilégier la rapidité d'accès à l'information au détriment de la place mémoire dans cette version de MICRO.

8.4.3 Modification d'une VCM

Les variables de MICRO possèdent un domaine de définition.

- Pour les variables monovaluées, ce domaine est représenté par une énumération de valeurs ou par des unions d'intervalles que la normalisation transforme en unions d'intervalles disjoints. Il n'y a donc qu'une façon de modifier le domaine d'une VCM monovaluée.
- Pour les variables multivaluées, ce domaine est représenté par cinq éléments descriptifs : l'extension, le domaine des éléments, l'ensemble des valeurs impossibles, la cardinalité et l'ensemble des éléments obligatoires, auxquels on ajoute l'ensemble des éléments obligatoires à certains rangs pour les listes. Il y a donc cinq, voire six, moyens de modifier le domaine d'une VCM multivaluée.

Avant de donner un algorithme général (que la VCM soit mono ou multivaluée) de réaction à la modification du domaine d'une VCM, nous nous intéressons aux répercussions de la modification d'un des éléments descriptifs au sein même du domaine de définition d'une VCM multivaluée. Ces divers changements sont pris en compte par les règles de maintien de la consistance (les réductions seulement) opérant sur des VCM multivaluées, mais aussi lors de modification directe par l'utilisateur d'un domaine de VCM multivaluée.

8.4.3.1 Modification d'une variable multivaluée

Nous donnons ici les répercussions de la modification – réduction ou relaxation – de chaque élément descriptif (extension, domaine des éléments, ensemble des valeurs impossibles, cardinalité, ensemble des éléments obligatoires, ensemble des éléments obligatoires à certains rangs) d'une variable multivaluée sur les autres éléments descriptifs selon que son extension est finie ou infinie.

L'extension d'une variable multivaluée de type ensemble est finie si et seulement si le domaine de ses éléments est fini. L'extension d'une variable multivaluée de type liste est finie si et seulement si le domaine de ses éléments est fini et si sa cardinalité est finie.

Modification de l'extension La réduction de l'extension (qui est, par conséquent, finie) peut entraîner une réduction du domaine des éléments, une relaxation de l'ensemble des valeurs interdites, et une réduction de la cardinalité et de l'ensemble des éléments obligatoires, ensemble des éléments obligatoires à certains rangs. On doit donc procéder au calcul de ces quatre éléments descriptifs à partir de l'extension réduite.

La relaxation de l'extension peut entraîner une relaxation du domaine des éléments, une réduction de l'ensemble des valeurs interdites, et une relaxation de la cardinalité. On doit donc procéder au calcul de ces trois éléments descriptifs à partir de l'extension étendue.

Modification du domaine des éléments Si l'extension est finie, la réduction du domaine des éléments peut entraîner une réduction de l'extension, une réduction de l'ensemble des valeurs interdites, une réduction de la cardinalité et une réduction de l'ensemble des éléments obligatoires. On doit donc procéder au calcul de ces éléments descriptifs à partir du domaine réduit des éléments.

Si l'extension est infinie, la réduction du domaine des éléments peut entraîner une réduction de l'extension (qui peut devenir finie si le domaine et la cardinalité sont finis), une réduction de l'ensemble des valeurs interdites, une réduction de la cardinalité et une réduction de l'ensemble des éléments obligatoires. On doit donc procéder au calcul de ces éléments descriptifs à partir du domaine réduit des éléments.

Si l'extension est finie, la relaxation du domaine des éléments peut entraîner une relaxation de l'extension. On doit donc procéder au calcul de cet élément descriptif à partir du domaine

étendu des éléments.
Si l'extension est infinie, la réduction du domaine des éléments n'a pas d'effet direct sur les autres éléments descriptifs de la variable multivaluée.

Modification de l'ensemble des valeurs impossibles Si l'extension est finie, la réduction de l'ensemble des valeurs impossibles peut entraîner une relaxation de l'extension, une relaxation du domaine des éléments, une relaxation de la cardinalité. On doit donc procéder au calcul de ces éléments descriptifs à partir de l'ensemble réduit des valeurs impossibles.
Si l'extension est infinie, la réduction de l'ensemble des valeurs impossibles n'a pas d'effet direct sur les autres éléments descriptifs de la variable multivaluée.
Si l'extension est finie, la relaxation de l'ensemble des valeurs impossibles peut entraîner une réduction de l'extension, une réduction du domaine des éléments, une réduction de la cardinalité, et une réduction de l'ensemble des éléments obligatoires. On doit donc procéder au calcul de ces éléments descriptifs à partir de l'ensemble étendu des valeurs impossibles.
Si l'extension est infinie, la réduction de l'ensemble des valeurs impossibles n'a pas d'effet direct sur les autres éléments descriptifs de la variable multivaluée.

Modification de la cardinalité Si l'extension est finie, la réduction de la cardinalité peut entraîner une réduction de l'extension, une réduction du domaine des éléments, une réduction de l'ensemble des valeurs impossibles et une réduction des éléments obligatoires dans le cas d'une liste. On doit donc procéder au calcul de ces éléments descriptifs à partir de la cardinalité réduite.
Si l'extension est infinie, la réduction de la cardinalité peut entraîner la réduction de l'extension (si le domaine des éléments est fini), la réduction de l'ensemble des valeurs impossibles, la réduction de l'ensemble des éléments obligatoires. On doit donc procéder au calcul de ces éléments descriptifs à partir de la cardinalité réduite.
Que l'extension soit finie ou non, la relaxation de la cardinalité n'a pas d'effet direct sur les autres éléments descriptifs de la variable multivaluée.

Modification de l'ensemble des éléments obligatoires Nous regroupons ici les répercussions des modifications des éléments obligatoires (pour un ensemble ou une liste) et des éléments localisés obligatoires.
Si l'extension est finie, la réduction de l'ensemble des éléments obligatoires peut entraîner une relaxation de l'extension. On doit donc procéder au calcul de cet élément descriptif à partir de l'ensemble réduit des éléments obligatoires.
Si l'extension est infinie, la réduction de l'ensemble des éléments obligatoires n'a pas d'effet direct sur les autres éléments descriptifs de la variable multivaluée.
Si l'extension est finie, la relaxation de l'ensemble des éléments obligatoires (sous réserve que l'élément requis appartienne au domaine des éléments et, le cas échéant, que la place indiquée respecte les conditions de cardinalité) peut entraîner une réduction de l'extension. On doit donc procéder au calcul de cet élément descriptif à partir de l'ensemble réduit des éléments obligatoires.
Si l'extension est infinie, la réduction de l'ensemble des éléments obligatoires (sous réserve que l'élément requis appartienne au domaine des éléments et, le cas échéant, que la place indiquée respecte les conditions de cardinalité) n'a pas d'effet direct sur les autres éléments descriptifs de la variable multivaluée.

8.4.3.2 L'algorithme de modification d'une VCM

Durant l'existence de la variable, le domaine courant de la variable est un ensemble de valeurs compris entre l'ensemble vide (état inconsistant et provisoire) et le domaine de définition. Le domaine de définition constitue donc une borne supérieure du domaine courant d'une variable contrainte, le domaine vide, une borne inférieure.

Il faut noter que ce domaine courant est également le domaine des AIC qui sont liés à la VCM. Aussi, sa présence dans l'objet dédié à la représentation de la VCM n'est pas obligatoire puisque les liens avec les AIC permettent de le retrouver.

Lorsque le domaine courant d'une VCM est modifié *indirectement*, c'est-à-dire lorsqu'une règle de consistance déclenchée par la propagation d'une contrainte vient modifier le domaine d'un des AIC auxquels la VCM est liée, il faut assurer que tous les autres AIC de la VCM prennent le même domaine avant de lancer les règles de consistance associées à leur contrainte. De même, lorsque le domaine courant d'une VCM est modifié *directement* par l'utilisateur, il faut assurer que tous les AIC de la VCM prennent le même domaine avant de lancer les règles de consistance associées à leur contrainte.

La modification du domaine de définition d'une VCM est autorisée sous réserve que le nouveau domaine fourni soit bien défini sur le type de la variable. Puisque le CSP est déjà construit, le principe est de valider la consistance locale du CSP avec ce nouveau domaine. Si le CSP n'est pas localement consistant, on retrouve l'ancienne configuration.

Cependant, si le graphe du CSP est inchangé, la définition du CSP est différente et on peut considérer que l'on a un nouveau CSP. Notamment, les contextes de pose des contraintes ne correspondent plus si on change les domaines des VCM, rendant incohérente la méthode employée lors de la suppression de contraintes.

La procédure *modifier-domaine* (*cf.* tableau 8.13) décrit comment MICRO réagit à la modification (réduction ou extension) du domaine d'une VCM.

Procédure modifier-domaine(VCM : variable, NEW-DOM : domaine)
début
  $VCM.dom-init \leftarrow NEW-DOM$
  % la variable a un nouveau domaine de définition %
  pour tout $AIC \in$ liste-aic(VCM) **faire**
  % pour chaque AIC correspondant à la VCM %
    $AIC.dom \leftarrow NEW-DOM$
    % l'AIC reçoit le domaine de la VCM %
    $propager-restauration(AIC, \emptyset)$
    % invalidation des domaines des AIC internes %
  fin pour
  pour tout $var \in$ réseau(VCM) **faire**
  % pour toutes les variables du réseau actuel de la VCM modifiée %
    si $val(var)$ indéfinie **alors**
    % si la VCM n'a pas de valeur %

      $var.dom \leftarrow var.dom-init$
      % son domaine courant redevient son domaine initial %
      pour tout $AIC \in$ liste-aic(var) **faire**
      % pour chaque AIC correspondant à la variable %
        $AIC.dom \leftarrow dom(var)$
        % l'AIC reçoit le domaine de la VCM %
        $propager-restauration(AIC, \emptyset)$
        % invalidation des domaines des AIC internes %
      fin pour
    finsi
  fin pour
  $c \leftarrow premier(contr(VCM))$
  % la propagation s'effectue à partir de la première contrainte de la VCM %
  $C \leftarrow$ antérieures(c)
  % la propagation débute sur le réseau limité aux contraintes antérieures %
  $X \leftarrow$ variables(C)
  $D \leftarrow$ domaines(X)
  pour tout c' t.q même-réseau(c, c') **et** après(c, c') **faire**
  % on propage chaque contrainte postérieure à la première contrainte de la VCM %

    c'.contexte $\leftarrow$ construire-contexte(X, D)

% le contexte de pose de la contrainte est mis à jour %
propager-contrainte($X, D, C, c', \emptyset, \emptyset$)
% on propage la contrainte %
$C \leftarrow C \cup c'$
$X \leftarrow X \cup$ variables(c')
$D \leftarrow D \cup$ domaines(c')
% mise à jour du CSP courant %
fin pour
fin

TAB. 8.13 - : La procédure *modifier-domaine* est chargée d'attribuer un nouveau domaine de définition à une VCM. La mise à jour des contextes de pose impose que soient propagées toutes les contraintes portant sur la VCM. Si une des propagations échoue, l'ancien domaine est rétabli.

La première chose à faire est d'affecter le nouveau domaine à la variable concernée alors que les autres variables du CSP retrouvent leur domaine de définition. Les variables dont la valeur a été fournie par l'utilisateur conservent cette valeur. Dans le réseau d'instances de contraintes, les AIC internes (qui sont liés à un autre AIC) reçoivent un domaine indéfini que les propagations de contraintes définiront.

Toutes les contraintes posées sur la VCM modifiée sont alors propagées dans l'ordre, mais de sorte que, pour un CSP à m contraintes, la propagation de la $i$ème contrainte se fasse en masquant dans le réseaux les contraintes $i+1, i+2, \ldots, m$. Autrement dit, seules sont susceptibles d'être activées pour des propagations inhérentes à la $i$ème contrainte, les contraintes $1, 2, \ldots, i-1$. Lors de l'activation de ces contraintes, le contexte de pose est recalculé.

La gestion de la dynamicité proposée par MICRO est adaptée au traitement des CSP à domaines infinis et répond aux modifications des variables contraintes. Cette fonctionnalité est importante car elle permet d'optimiser les temps de réponse (à défaut de l'espace mémoire) en cas de retrait d'une contrainte ou d'un changement sur une variable.

Une autre caractéristique intéressante de MICRO est la gestion de *méta-contraintes*. Nous les présentons dans la section suivante.

8.5 Les méta-contraintes

Une contrainte de MICRO est considérée comme *méta-contraintes* dès lors que l'un de ses arguments est une contrainte. Les méta-contraintes ne contraignent pas des contraintes mais procèdent à la pose de leurs contraintes arguments. On distingue deux types de méta-contraintes : les *semi-méta-contraintes* et les *méta-contraintes non déterministes*. Nous présentons ici ces deux classes de méta-contraintes.

8.5.1 Semi-méta-contraintes

Les semi-méta-contraintes sont des contraintes dont l'un des arguments est, soit une fonction, soit une contrainte. Elles sont au nombre de sept (trois principales et quatre secondaires) :

- la semi-méta-contrainte principale (`mic-every` C X Y) pose une contrainte principale C sur chaque élément d'un ensemble ou d'une liste de VCM X, Y étant le second argument de la contrainte C.
  Par exemple, (`mic-every` #'mic-gt (`mic-set` x y z) t) impose que les VCM x, y, et z soient chacune supérieure à la VCM monovaluée t. Ici, c'est un moyen de factoriser la pose de trois contraintes.
- la semi-méta-contrainte principale (`mic-all` C X) pose une contrainte principale binaire C sur les éléments d'un ensemble ou d'une liste X de VCM pris deux à deux. Par exemple, (`mic-all` #'mic-eq (`mic-set` x y z)) (*cf.* figure 8.5) impose que les VCM x, y, et z soient

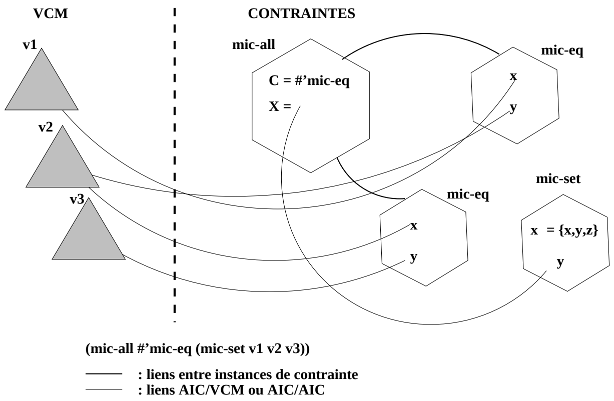


FIG. 8.5 - : Un exemple de semi-méta-contrainte principale, la contrainte (`mic-all` #'mic-eq (`mic-set` *v1 v2 v3*)) entraîne la pose de deux contraintes `mic-eq`. La semi-méta-contrainte garde trace des instances de contraintes dont elle est à l'origine de la pose. Ces liens sont utilisés lors de la suppression de la semi-méta-contrainte pour retrouver les contraintes posées.

toutes égales. C'est aussi un moyen de factoriser la pose de trois contraintes.

- la semi-méta-contrainte principale (`mic-compar` C X Y) pose une contrainte principale binaire sur chaque paire d'éléments de même rang de deux listes de VCM X et Y.
  Par exemple, (`mic-compar` #'mic-eq (`mic-list` x y z) (`mic-list` a b c)) impose que les VCM x et a (respectivement y et b, z et c) soient égales. Ici, c'est encore un moyen de factoriser la pose de six contraintes.
- la semi-méta-contrainte secondaire (`mic-map` C X Y) pose une contrainte secondaire binaire C sur chaque élément d'une liste de VCM X, Y étant le second argument de la contrainte C. Son résultat est une VCM multivaluée de type liste qui contient le résultat de l'application de l'opérateur contraint C à chaque élément de la liste avec Y.
  Par exemple, (`mic-map` #'mic-add (`mic-list` x y z) t) rend la VCM dont la valeur est la liste des résultats des contraintes (`mic-add` x t), (`mic-add` y t) et(`mic-add` z t).
- la semi-méta-contrainte secondaire (`mic-apply` C X) pose une contrainte secondaire binaire C sur les deux premiers éléments d'une liste de VCM X, puis pose C sur le résultat et sur l'élément suivant et ainsi de suite. Son résultat est une VCM monovaluée qui contient le résultat de l'application, de l'opérateur contraint C à chaque élément de la liste avec Y. Par

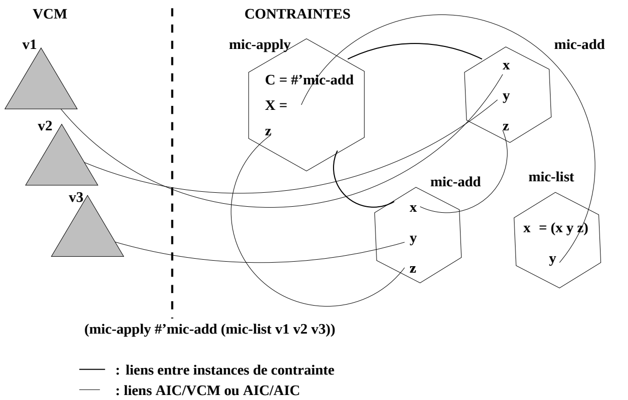


FIG. 8.6 - : Un exemple de semi-méta-contrainte secondaire, la contrainte (`mic-apply` #'mic-add (`mic-list` *v1 v2 v3*)) entraîne la pose de deux contraintes `mic-add`. Le résultat de la contrainte `mic-apply` peut être, à son tour, employé dans une autre contrainte (principale ou secondaire).

exemple, (`mic-apply` #'mic-add (`mic-list` x y z)) (*cf.* figure 8.6) rend la VCM dont la valeur est le résultat de la contrainte (`mic-add` (`mic-add` x y) z).

- la semi-méta-contrainte secondaire (`mic-select` C X Y) pose une contrainte secondaire binaire booléenne C sur chaque élément d'une liste de VCM X, Y étant le second argument de la contrainte C. Son résultat est une VCM multivaluée de type liste qui contient seulement les éléments de X pour lesquels la condition C est vérifiée avec Y.
  Par exemple, (`mic-select` #'mic-is-eq (`mic-list` x y z) v) rend la VCM dont la valeur est une liste contenant la valeur de x (respectivement y et z) si le résultat de la contrainte (`mic-is-eq` x v) vaut *vrai*.
- la semi-méta-contrainte secondaire (`mic-combin` C X Y) pose une contrainte secondaire binaire sur chaque paire d'éléments de même rang de deux listes de VCM X et Y. Son résultat est une VCM multivaluée de type liste qui contient le résultat de l'application de l'opérateur contraint C à deux éléments de même rang de X et Y respectivement.
  Par exemple, (`mic-combin` #'mic-add (`mic-list` x y z) (`mic-list` a b c) rend la VCM dont la valeur est la liste des résultats des contraintes (`mic-add` x a), (`mic-add` y b) et(`mic--add` z c).

Non seulement ces méta-contraintes permettent une factorisation de l'écriture de contraintes, mais elles permettent la construction de VCM multivaluées à partir d'applications d'opérateurs sur d'autres VCM. De plus, le fait que leur pose entraîne la pose de leur contrainte argument assure que toute modification, concernant la liste ou l'ensemble des VCM à contraindre, est soumise à vérification et déclenche une propagation de contraintes. Ce fonctionnement ne serait pas possible avec de simples fonctions.

8.5.2 Méta-contraintes non-déterministes

Conformément à l'intérêt suscité par une programmation par contraintes en mode non déterministe dans une base de connaissances, nous avons ajouté à MICRO trois contraintes que l'on peut considérer comme des méta-contraintes, c'est-à-dire des fonctions dont les arguments sont des contraintes :

- `mic-meta-or` qui prend deux contraintes principales en arguments : la pose de cette contrainte entraîne la pose de la contrainte qui est le premier argument. Si celle-ci ne peut être posée, alors MICRO tente de poser la seconde contrainte. Si cette tentative échoue aussi, la pose de la contrainte `mic-meta-or` échoue également. Si au cours de son existence, l'une des contraintes arguments n'est plus satisfaite alors MICRO tente de satisfaire l'autre contrainte argument. Si cette tentative échoue, l'action à l'origine de cette violation de la contrainte `mic-meta-or` est refusée.
- `mic-meta-if-then` qui admet deux arguments : une variable contrainte booléenne qui peut donc être une contrainte booléenne secondaire et une contrainte principale. La pose de cette contrainte entraîne la pose de la contrainte booléenne secondaire. Si le résultat de cette contrainte vaut *vrai* alors la contrainte principale donnée en second argument est posée, sinon la contrainte principale n'est pas posée. Le fait que la condition du *si ... alors* soit exprimée par une contrainte booléenne permet de réagir par une pose ou un retrait de la contrainte principale selon les modifications du résultat de la contrainte booléenne représentant la condition. Ainsi, si le résultat devient *vrai* alors qu'il était *faux* alors la contrainte est posée. Inversement si le résultat devient *faux* alors qu'il était *vrai*, la contrainte principale est retirée.
  Il faut noter que si le résultat de la contrainte booléenne est indéfini alors la contrainte principale n'est pas posée.
  Si une action est à l'origine de la violation de la contrainte principale et donc de la contrainte `mic-meta-if-then`, alors elle est refusée. De même, si la pose de la contrainte principale échoue, alors la pose de la méta-contrainte échoue.
- `mic-meta-if-else` qui admet trois arguments : une variable contrainte booléenne qui peut donc être une contrainte booléenne secondaire et deux contraintes principales. La pose de

cette contrainte entraîne la pose de la contrainte booléenne secondaire. Si le résultat de cette contrainte vaut *vrai* alors la contrainte principale donnée en second argument est posée, sinon la contrainte principale donnée en troisième argument est posée. Le fait que la condition du *si* ... *alors* ... *sinon* soit exprimée par une contrainte booléenne permet de réagir par une pose ou un retrait de l'une des deux contraintes principales selon les modifications du résultat de la contrainte booléenne représentant la condition. Ainsi, si le résultat devient *vrai* alors qu'il était à *faux*, alors la première contrainte principale est posée, la seconde est retirée. Inversement si le résultat devient *faux* alors qu'il était à *vrai*, alors la seconde contrainte principale est posée, la première est retirée.

Il faut noter que si le résultat de la contrainte booléenne est indéfini alors la contrainte principale n'est pas posée. Par contre, s'il passe d'une valeur indéfinie à *vrai* (respectivement *faux*) alors la première (respectivement la seconde) contrainte principale est posée.

Si une action est à l'origine de la violation d'une des contraintes principales et donc de la contrainte `mic-meta-if-then`, alors elle est refusée. De même, si la pose de la contrainte principale concernée échoue, alors la pose de la méta-contrainte échoue.

Les méta-contraintes sont des contraintes principales. Elles sont également représentées par des classes de contraintes. Les AIC des instances de ces méta-contraintes correspondant à des contraintes principales arguments de la méta-contrainte sont liés aux instances de ces contraintes principales.

La contrainte `mic-meta-or` nécessite un lien entre la contrainte principale argument posée et la méta-contrainte. Ce lien permet, lors de la violation ou le retrait de la contrainte principale, d'essayer l'autre contrainte principale argument de la méta-contrainte.

La résolution d'une méta-contrainte revient à la résolution de la contrainte qui la représente, s'il y en a une dans le cas de `mic-meta-if-then`. Dans le cas de la méta-contrainte `mic-meta-or`, toutefois, l'échec de la résolution de la contrainte principale représentant la méta-contrainte entraîne la tentative de résolution avec l'autre contrainte principale argument.

Quant au retrait d'une méta-contrainte, il consiste à retirer la contrainte principale, s'il y en a une dans le cas de `mic-meta-if-then`.

8.6 Comparaison avec d'autres systèmes

Nous comparons ici MICRO en tant que module de programmation par contraintes autonome avec les systèmes de programmation par contraintes présentées au chapitre 5.

MICRO est un module de programmation par contraintes sur des intervalles. Contrairement aux langages de programmation logique par contraintes comme CLP($\mathcal{R}$) [Jaffar et al.91] ou PROLOG III [Colmerauer90], il ne repose pas sur l'unification et ne permet pas de rendre des résultats symboliques. De tels résultats rendent explicites des relations mais ne résolvent pas pour autant le problème et n'ont de réelle utilité que pour les problèmes sous-contraints pour lesquels sont exhibées des relations implicites, déductibles de l'information présente.

MICRO ne comporte pas non plus d'algorithme ou de méthode spécialisés, comme le Simplexe pour CLP($\mathcal{R}$) et PROLOG III ou la méthode de Newton pour cc($\mathcal{FD}$) [Hentenryck et al.93], qui sont souhaitables dans un module destiné à des applications spécifiques en calcul scientifique ou pour des problèmes d'ordonnancement, mais qui ne sont pas *a priori* indispensables pour la résolution de CSP assez généraux et simples auxquels MICRO est destiné.

En revanche, MICRO inclut un algorithme de résolution capable de s'adapter aux domaines du CSP à résoudre, qu'ils soient finis ou infinis.

Pour des commodités de représentation et afin de traiter des variables entières et réelles, nous avons choisi de concevoir un module de programmation par contraintes sur intervalles. À ce titre, MICRO est à rapprocher davantage des langages tels que BNR PROLOG [Older et al.90], INTERLOG [Dassault electronique91], INC++ [Hyvönen et al.93b] ou PECOS/SOLVER [Ilog92b]. Comme dans ces langages, les contraintes sont construites à partir d'un ensemble prédéfini de contraintes de base (binaires ou ternaires) qui sont assemblées. Les résultats des contraintes de base représentant des opérateurs arithmétiques sont obtenus à partir des règles de l'arithmétique des intervalles. MICRO fournit un ensemble complet de contraintes (opérateurs de comparaison et opérateurs arithmétiques) pour les CSP à variables entières ou réelles mais aussi booléennes. Les opérateurs booléens permettent de définir des contraintes sur des variables de types différents. Cette caractéristique ne se retrouve que dans le langage CLP(BNR) [Older et al.93].

Le niveau de consistance locale atteint dans les CSP MICRO dépend du type des variables, de la finitude des domaines et de la monotonicité des opérateurs utilisés. Le plus fort niveau est l'arc-consistance (pour les CSP à domaines finis), puis l'intervalle consistance (pour les CSP à domaines infinis sans opérateurs périodiques et sans double occurrence de la même variable). Pour les opérateurs périodiques, le filtrage est retardé le plus possible aussi la consistance est-elle très faible. La consistance locale est établie lors de la propagation de contraintes, mécanisme central dans MICRO, qui consiste à activer une contrainte en appliquant les règles de maintien de consistance qui lui sont associées avant d'activer, à leur tour, les contraintes portant sur les variables dont les domaines ont été réduits par ces règles. La consistance locale dans MICRO est d'un degré en moyenne plus élevé que celle maintenue dans les langages de programmation par contraintes sur intervalles uniques comme PECOS et INTERLOG. Car, bien que plus coûteuse, la gestion d'unions d'intervalles permet d'obtenir une intervalle-consistance là où ces systèmes atteignent une boîte-consistance.

La propagation de contraintes, comme dans tous les systèmes cités, n'échappe pas aux cycles qui peuvent se comporter comme des boucles infinies au regard des limitations de la pile d'exécution de la machine. De ce point de vue, MICRO est à rapprocher davantage de la première version de INC++ [Hyvönen92].

Concernant le calcul de l'étendue des domaines, le recours à des méthodes la minimisant, comme c'est le cas dans INC++, n'a pas été retenu. Là encore, il ne s'agissait pas de faire de MICRO, dans sa première version, un outil pour traiter les fonctions spécifiques comme les polynômes, qui apparaissent souvent dans les formules d'un tableur et auxquels INC++ entend apporter une solution efficace. D'autre part, le recours à des méthodes comme les formes centrées de Moore, nécessite d'analyser l'expression entière de la contrainte, de calculer les formes de Taylor des parties droite et gauche des égalités ou inégalités. Des bibliothèques ou modules spécifiques de calcul doivent donc être associées au système afin de le rendre encore plus précis et performant. Cette recherche d'optimisation sur des contraintes somme toute particulières n'a pas été la motivation première lors de la conception de MICRO.

Dans MICRO, la résolution est basée sur deux principes, les domaines finis sont résolus à l'aide d'un algorithme de type *Full-Look-Ahead*. Cette méthode, dont les performances respectables ont été soulignées par Sabin et Freuder [Sabin et al.94], est retenue aussi dans PECOS. On est donc en droit d'attendre de MICRO des performances comparables à ce système sur de tels CSP. Les CSP à domaines infinis sont traités par division (comme dans INC++). Cette méthode est par essence combinatoire, mais nous avons proposé d'en contrôler le déroulement par un algorithme de *Branch and Bound* intégrant un critère d'évaluation de l'intérêt de la poursuite d'une division.

En fait, l'originalité de MICRO réside davantage dans le traitement des unions d'intervalles, traitement que les systèmes précédents ne proposent pas. Si les temps de calcul des intervalles sont accrus – ce qui constitue le principal argument des concepteurs des systèmes précédents pour raisonner en mono-intervalle –, l'expressivité et le degré de consistance locale effectivement maintenus sont beaucoup plus importants dans le pari multi-intervalles que tient MICRO.

Une autre facette intéressante de MICRO, qui n'est pas ou peu exploitée par les systèmes cités, est la mise à la disposition de l'utilisateur d'un ensemble de contraintes de base pour la définition de CSP sur des variables multi-valuées (ensembles et listes). Seul PECOS propose un éventail de contraintes ensemblistes n'opérant cependant que sur des ensembles finis et ne concernant, paradoxalement, pas les listes. Le langage hôte de MICRO étant le langage LE-LISP V16 dans sa première version, il nous a semblé naturel de nous intéresser également à ces dernières.

Le traitement de CSP dynamiques est, lui aussi, propre à MICRO. Pour ce faire, des structures de données (contextes) sont stockées et gérées par des algorithmes permettant le retrait d'une contrainte ou encore la modification du domaine d'une variable. On ne retrouve pas cette caractéristique dans les autres langages de programmation par contraintes à intervalles.

Enfin, une dernière particularité de MICRO, est la présence de contraintes qui permettent la pose d'autres contraintes sur les éléments de listes de variables, ou encore les méta-contraintes, qui permettent le non-déterminisme et la programmation par contraintes conditionnelles. PECOS propose également une programmation par contraintes non déterministe mais les expressions non déterministes élémentaires proposées ne sont destinées qu'à guider la résolution en définissant des points de choix. En l'occurrence, elles ne permettent pas de faire évoluer l'ensemble des contraintes du CSP en fonction de l'évolution des domaines des variables.

Les points sur lesquels MICRO se démarque des systèmes voisins nous ont été suggérés par l'application pour laquelle MICRO a été construit : la liaison avec le modèle de connaissances à objets TROPES. Aussi, les choix et la réalisation du module MICRO ont été faits pour répondre aux besoins en contraintes de ce modèle. La recherche de performances n'a donc pas été l'objectif premier. Souvent, elle a été sacrifiée au profit de la généricité, pour offrir un module de programmation par contrainte capable de traiter des CSP de natures diverses. Il reste que MICRO peut être utilisé comme un outil de programmation par contraintes indépendamment du couplage visé avec TROPES, voire être couplé avec d'autres SRPO.

8.7 Conclusion

MICRO est un module de programmation par contraintes destiné à traiter des CSP à variables de type entier, réel, ou booléen[4] dont les domaines sont finis ou infinis et exprimés en tant qu'énumération de valeurs ou unions d'intervalles. Afin de construire ces CSP, MICRO propose un ensemble de contraintes prédéfinies recouvrant la plupart des opérateurs de comparaison, arithmétiques et booléens à partir desquels des expressions complexes de contraintes peuvent être construites.

MICRO dispose également d'un ensemble d'opérateurs destinés à définir des CSP sur des variables multivaluées (dont la valeur est un ensemble ou une liste de valeurs).

La définition de contraintes complexes peut être conservée par MICRO permettant ainsi une économie d'écriture et une factorisation des traitements dans les utilisations multiples des mêmes contraintes.

MICRO inclut des fonctions permettant la pose de plusieurs contraintes sur des listes ou des ensembles de variables contraintes ainsi que des méta-contraintes (contraintes dont les arguments sont de contraintes) qui permettent de contraindre des ensembles ou des listes de variables ou encore d'introduire du non-déterminisme dans la définition et la résolution des CSP.

Une première version de MICRO a été écrite dans le langage LE-LISP V16 [Ilog92a] ; elle est limitée à la maintenance par propagation de contraintes de CSP à domaines finis ou infinis et à la résolution de CSP à domaine finis. Elle utilise le langage à objets TELOS pour décrire la classe des variables (appelées VCM) et la classe de chacun des opérateurs prédéfinis mis à la disposition de l'utilisateur. Les CSP sont construits à partir des expressions contraintes et sont matérialisés par des instances des classes de variables et de contraintes. Cette version de MICRO est destinée à la version 0 du système TROPES écrite en LE-LISP V16.

La seconde version de MICRO écrite en langage TALK [Ilog94] est en cours d'implémentation et comporte les algorithmes de gestion de la dynamicité ainsi l'algorithme général de résolution présenté ici. Elle est destinée à la version 1.0 [Sherpa95] du système TROPES écrite en TALK.

La maintenance des contraintes dans MICRO repose sur la propagation de contraintes. À chaque

[4] Ou de type quelconque avec des contraintes particulières (égalité, différence , affectation).

classe de contrainte est associée une méthode qui applique les règles de maintien de la consistance locale sur les domaines-arguments de la contrainte. Lorsque l'application de ces règles réduit le domaine d'un argument de la contrainte, la propagation consiste à activer la méthode de consistance associée à toutes les autres contraintes portant sur le domaine réduit.

Pour les domaines de variables monovaluées entières et réelles, ces règles sont issues de l'arithmétique des intervalles et garantissent sinon l'arc-consistance, tout au moins une intervalle-consistance pour les contraintes à base d'opérateurs non périodiques et les contraintes sans double occurrence de la même variable, sinon une boîte consistance. Pour les variables multivaluées, une représentation a été définie, les règles de consistance agissent sur chaque élément de la description du domaine (extension, domaine des éléments, éléments impossibles, cardinalité, éléments requis).

MICRO privilégie la gestion de CSP dynamiques et permet l'ajout comme le retrait de contraintes, sans qu'il ne soit nécessaire de redéfinir le CSP. Il est également capable de réagir à la modification du domaine d'une variable.

MICRO fournit un algorithme de résolution pour les CSP à domaines finis et/ou à domaines infinis. Les domaines finis sont traités par une énumération basée sur un *Full-Look-Ahead* s'appuyant sur la propagation de contraintes. Les domaines infinis sont traités par dichotomie et résolution des sous-CSP résultants. La division des domaines est contrôlée par un algorithme de type *Branch and Bound*, des critères décidant de la poursuite ou non de cette division.

Une interface fonctionnelle[5] permet de définir, de consulter et de modifier à tout moment des CSP (définition et consultation de variables, ajout, retrait de contraintes, modifications des domaines) et de demander la résolution de ces CSP. C'est par elle que l'utilisateur (ou un programme) communique avec MICRO.

Ces caractéristiques font de MICRO un module de programmation par contraintes dont l'originalité est qu'il concilie la définition et le traitement de CSP dynamiques, à domaines finis ou infinis, avec une gestion complète d'unions d'intervalles. Afin que MICRO soit en mesure d'accomplir la tâche pour laquelle il a été conçu, c'est-à-dire gérer les CSP définis dans le système TROPES, une interface a été conçue. Nous la décrivons dans le chapitre suivant.

[5] Décrite en annexe B.1.

Chapitre 9

Le couplage Tropes/Micro

Le chapitre précédent a présenté MICRO comme un module de programmation par contraintes dont les différentes classes de variables et de contraintes permettent son utilisation autonome, c'est-à-dire sans qu'il soit besoin de le rattacher à un autre système. Cependant, la motivation originale de la conception de MICRO étant le couplage du modèle de connaissances à objets TROPES avec un module de programmation par contraintes, nous décrivons ici ce couplage.

Comme le montre la section 9.1, le couplage réalisé est un couplage dans lequel les Attributs Contraints de TROPES (ACT) remplacent les VCM de MICRO.

Pour que soit opérée une telle substitution, les objets TROPES doivent contenir certaines informations, notamment celles détenues par les VCM lorsque MICRO est utilisé en mode autonome, que nous détaillons dans la section 9.2. Nous décrivons à quel moment ces informations sont mises à jour (*cf.* section 9.3).

Nous décrivons ensuite comment les types des attributs contraints peuvent être recalculés à partir de CSP gérés également par MICRO et l'interface (*cf.* section 9.4).

Les fonctionnalités de l'interface (*cf.* section 9.5) sont de traduire les opérations disponibles sur les CSP TROPES – définition, pose et suppression de contrainte dans TROPES, modification du domaine d'un ACT – en fonctions sémantiquement correspondantes de MICRO.

Enfin, nous comparons (*cf.* section 9.6) le couplage TROPES/MICRO avec les autres associations objets/contraintes présentées au chapitre 5.

9.1 Un couplage semi-faible

Si nous considérons MICRO en tant que module autonome de programmation par contraintes, le premier type de couplage avec le modèle TROPES qui vient à l'esprit est un couplage de type *faible* dans lequel chaque attribut (appelé désormais ACT) est lié avec une VCM (Variable Contrainte de MICRO), comme le montre la figure 9.1. Dans cette solution, les VCM servent de relais entre les instances TROPES et les instances MICRO : toute mise à jour d'un ACT doit être répercutée sur la VCM associée et, de là, dans le réseau de contraintes associé ; réciproquement, toute mise à jour d'une VCM (à l'issue d'une propagation) doit être transmise à l'ACT associé.

Ce couplage nécessite donc une interface entre TROPES et MICRO qui gère les échanges entre le modèle et le module, crée les CSP en déclarant variables et contraintes, déclenche les propagations, et réagit à toute modification d'un côté comme de l'autre. La première réalisation d'un couplage de TROPES avec un module de programmation par contraintes que nous avons effectuée – le couplage TROPES/PECOS décrit en 8.1 – était de ce type. Or, ce type de solution a le désavantage de dupliquer l'information puisque les VCM doivent être le miroir des ACT qui leur sont associés. Si l'économie de cette redondance ne peut, en général, être faite dans un couplage *faible* liant deux systèmes indépendants et se comportant l'un vis-à-vis de l'autre comme une boîte noire, il en est autrement

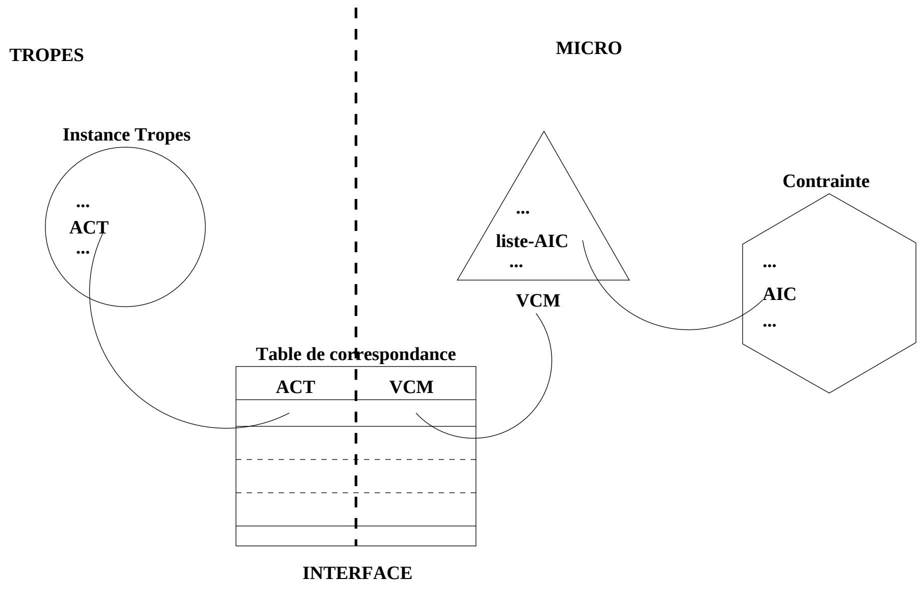


FIG. 9.1 - : Description d'un couplage faible entre TROPES et MICRO. Les VCM sont les miroirs des ACT. L'interface tient à jour une table de correspondance entre ACT et VCM. Les VCM seules sont liées aux AIC.

ici puisque, précisément, MICRO a, avant tout, été conçu pour TROPES.

Autrement dit, en tant que concepteur à la fois du modèle et du module, il nous est possible de shunter les VCM lors du couplage et de lier donc directement les ACT aux AIC de MICRO. L'idée sous-jacente est de faire jouer aux ACT le rôle des VCM. Dans MICRO, les VCM sont substituées par les ACT. De ce fait, l'information concernant les ACT n'est plus dupliquée dès lors que MICRO utilisent les ACT comme ses VCM.

De plus, il n'est plus besoin de déclarer (et de décrire) une VCM lorsqu'un nouvel ACT est contraint : il suffit de lier l'ACT à l'AIC de l'instance de contrainte de MICRO qui matérialise la présence de la contrainte. Aussi, la déclaration d'une contrainte sur un ACT équivaut à la définition d'une VCM pour MICRO.

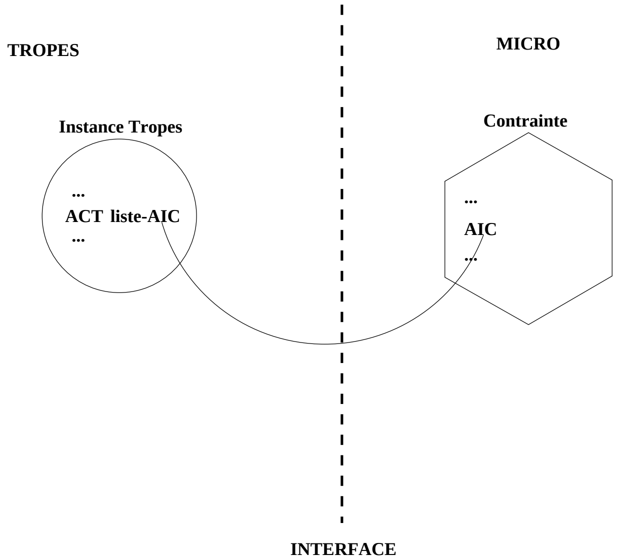


FIG. 9.2 - : Description du couplage semi-faible choisi pour TROPES et MICRO. Les VCM n'existent plus. La table de correspondance disparaît. Les ACT sont directement liés aux AIC.

En conclusion, le couplage réalisé ici n'est donc plus un couplage que l'on peut qualifier de *faible* car MICRO vient s'ancrer dans TROPES. Il ne s'agit pas non plus d'un couplage *fort* dans lequel les contraintes de MICRO seraient représentées par des objets du modèle TROPES. Pour TROPES et MICRO, le terme de couplage *semi-faible* semble approprié et dénote un couplage faible optimisé (sans redondance d'informations).

En contrepartie, ce couplage semi-faible impose de disposer dans les instances TROPES des informations nécessaires échangées avec MICRO. Nous décrivons à présent quelles sont ces informations.

9.2 Les structures informatives

9.2.1 Informations dans Tropes

Chaque instance TROPES – considérée du point de vue structurel comme formée de l'ensemble des attributs du concept – a dû être étendue afin de stocker un certain nombre de structures de données destinées au couplage semi-faible entre TROPES et MICRO:

- Pour chaque attribut susceptible d'être contraint, l'instance contient son domaine *effectif* qui est un sous-ensemble du domaine fourni par le type de l'attribut pour l'instance. Ce domaine est le plus grand ensemble de valeurs localement consistantes pour l'attribut. Il correspond au domaine effectif (ou courant) d'une VCM, et reflète l'état des modifications apportées par la propagation de contraintes sur le domaine de l'attribut contraint.
- Pour chaque attribut susceptible d'être contraint, l'instance stocke la liste des AIC auxquels il est associé. Il s'agit d'une liste et non pas d'un ensemble, car l'ordre de pose des contraintes est utilisé dans la gestion des CSP dynamiques. D'autre part, le type de la contrainte n'est pas considéré dans cette liste. Ainsi, on y trouve aussi bien des AIC d'instances de contrainte de concept, que des AIC d'instances de contraintes de classe, ou d'instances de contraintes d'instances. Cette information remplace la liste des AIC que contiennent les VCM de MICRO.
- Pour chaque attribut, un drapeau indique si la valeur de cet attribut a été inférée par une contrainte ou non. Cette information est utilisée lors du rétablissement d'un contexte de pose notamment. Elle se trouve également dans les VCM de MICRO.
- Chaque instance de TROPES doit contenir la liste des contraintes qui portent sur ses attributs. Ces contraintes sont donc aussi bien des contraintes de concept ou de classe que des contraintes d'instance car celles-ci sont également posées sur l'instance. Cette information vise à remplacer la liste des contraintes déclarées que MICRO tient à jour.
- Chaque instance de TROPES doit contenir la liste des instances de contraintes (de concept, de classe et d'instance) impliquant des attributs de l'instance et maintenues par MICRO. Cette information permet de reformer la table de correspondance contraintes/instances de contraintes que gère MICRO.

Parmi ces informations, seules sont accessibles pour un utilisateur de TROPES les informations "lisibles", c'est-à-dire son domaine effectif, la liste des contraintes qui portent sur l'instance et les drapeaux renseignant sur l'inférence des valeurs grâce aux contraintes. En raison des divers niveaux de représentation auxquels une contrainte peut être attachée, l'instance n'est pas la seule entité TROPES qu'il faille surcharger avec des informations permettant la gestion de CSP. La structure de classe et celle de concept ont dû également être étendues. Chaque classe de TROPES doit contenir (*cf.* figure 9.4) la liste des contraintes de classe mais aussi de concept qui portent sur ses attributs. Cette information est accessible aux utilisateurs de TROPES. Chaque concept de TROPES doit contenir (*cf.* figure 9.4) la liste des contraintes de concept qui portent sur ses instances. Cette information est accessible aux utilisateurs de TROPES.

9.2.2 Information dans Micro

Dans les instances de contraintes maintenues par MICRO, aucune information nécessaire au couplage n'a été instillée dans les AIC. En revanche, on trouve pour chaque AIC l'information dont

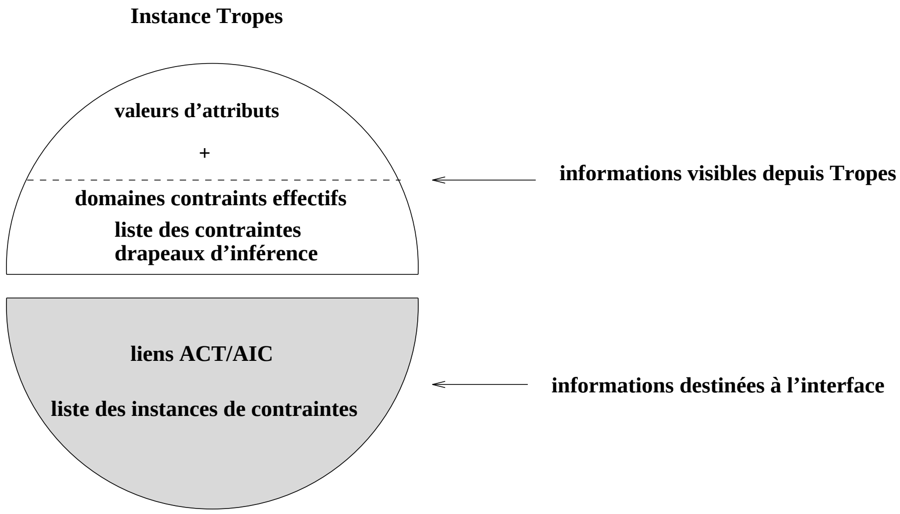


FIG. 9.3 - : Les structures de données ajoutées à une instance TROPES en vue du couplage avec MICRO.

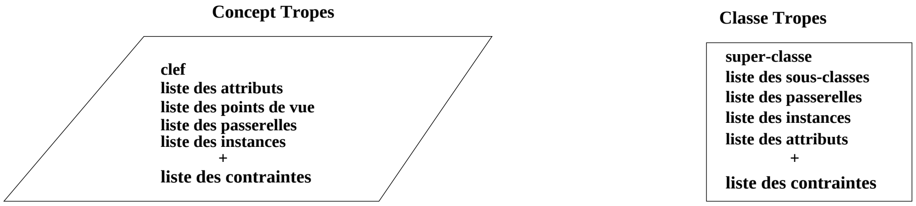


FIG. 9.4 - : Les structures de données ajoutées à une classe et à un concept TROPES en vue du couplage avec MICRO.

a besoin MICRO. Elle est utilisée pour le couplage comme en mode autonome (*cf.* figure 9.5) :

- Le domaine de l'AIC qui est établi par la propagation de contraintes. Lorsque l'AIC est lié à un ACT, ce domaine correspond au domaine effectif ou courant de l'ACT. Lorsque ce domaine est réduit à un singleton et que l'AIC est lié à un ACT, la valeur obtenue – qui devient valeur de l'ACT – est considérée comme inférée par la contrainte.
- Un lien qui peut être :
  - soit un lien avec un ACT. Typiquement, l'AIC modélise dans MICRO un ACT. Ce lien existe dans une utilisation classique de MICRO, mais lie alors l'AIC à une VCM. Dans le couplage semi-faible réalisé, les ACT remplacent les VCM ;
  - soit un lien avec un autre AIC. Typiquement, l'AIC est soit un argument de contrainte secondaire lié à un AIC résultat d'une autre contrainte secondaire, soit le résultat d'une contrainte secondaire utilisée comme argument d'une autre contrainte secondaire.

  Lorsque ce lien est vide, l'AIC représente une constante. Ce lien est unique, un AIC ne peut représenter plusieurs ACT, AIC ou constantes.

9.3 Gestion des liens et des structures informatives

Dans une instance, une classe ou un concept TROPES, les structures de données présentées dans la section précédente font désormais partie de la structure de l'entité. L'intégration de contraintes à TROPES se traduit donc par l'extension en place mémoire de la taille des entités de représentation. C'est le prix à payer pour gérer la consistance d'une base de connaissances en présence de contraintes dynamiques. Ces structures de données sont prévues dès la création de ces entités mais ne seront effectivement remplies ou modifiées que lors de la déclaration ou de la modification des CSP. Le

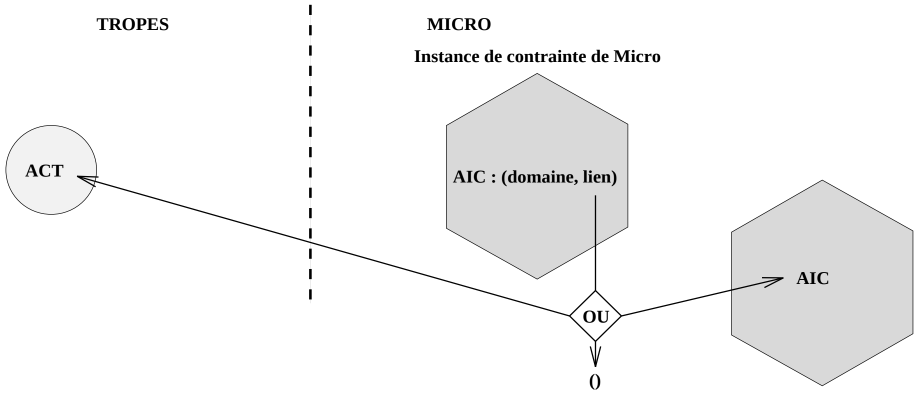


FIG. 9.5 - : Les structures de données d'une instance de MICRO. Chaque AIC comporte une information sur son domaine et un lien d'un des 3 types présentés.

contenu de ces structures informatives évolue donc avec les CSP de TROPES qu'elles permettent de traduire en CSP TROPES.

Ces informations sont destinées à l'interface. C'est l'interface entre TROPES et MICRO qui vient chercher et déposer dans ces structures les informations dont elle a besoin. La gestion de ces structures est donc immergée dans les primitives de manipulation des CSP TROPES qu'elle contient (*cf.* section 9.5).

Ainsi, au niveau d'une instance TROPES :

- la liste des contraintes est mise à jour lors de la déclaration et de la suppression d'une contrainte d'instance, de classe ou de concept ;
- le vecteur des drapeaux d'inférence par une contrainte est mis à jour lorsque la propagation de contraintes réduit un domaine à un singleton ;
- le vecteur des listes d'AIC par attribut est mis à jour lors de la création ou de la suppression d'une contrainte quel que soit son type ;
- la liste des instances de contraintes est mise à jour lors de la création ou de la suppression d'une contrainte.

Au niveau d'une classe (respectivement d'un concept), la liste des contraintes de la classe (respectivement du concept) est mise à jour lors de la création ou de la suppression d'une contrainte de la classe (respectivement du concept).

En ce qui concerne les informations conservées par les AIC de MICRO, les domaines sont mis à jour par la propagation de contraintes mais aussi par les fonctions de restauration de contexte. Quant aux liens, ils sont parcourus lors de la propagation de contraintes et sont mis en place lors de la pose de la contrainte.

9.4 Contraintes et calcul de types

Nous traitons ici des moyens à mettre en œuvre pour le calcul du type des entités contraintes, aussi bien les concepts, que les classes ou les instances. Lors de l'ajout ou de la suppression d'une contrainte, le nouveau type de l'entité est d'abord calculé avant de procéder à la pose ou au retrait effectif.

L'idée du calcul de type est simple : pour connaître l'effet de l'ajout ou du retrait d'une contrainte sur un ensemble d'attributs accessibles depuis un concept, une classe ou une instance, il suffit de propager l'ajout ou le retrait de la contrainte en remplaçant dans son réseau de contraintes les domaines effectifs des variables contraintes par leur domaine de définition.

Lors d'un ajout, ces domaines de définition correspondent aux types courants des attributs contraints. Lors d'un retrait de contrainte, ils correspondent aux types des attributs contraints au moment de la pose de la contrainte, et on procède à la pose des contraintes déclarées ultérieurement sur l'entité.

Cette propagation, si elle réussit, doit fournir les nouveaux domaines de définition des attributs contraints pour le concept, la classe ou l'instance, selon le niveau de représentation de la contrainte.

Les CSP destinés à calculer les types sont également maintenus par MICRO comme les autres CSP TROPES. Ces CSP ne sont pas rattachés à des ACT d'instances, pas plus qu'il n'est possible de les rattacher à une instance factice puisque les accès sont susceptibles d'atteindre un nombre imprévisible d'instances.

Autrement dit, il existe deux niveaux de CSP TROPES gérés par MICRO : ceux qui représentent les contraintes effectivement posées sur les attributs des objets et ceux qui ont été établis pour le calcul des types des concepts, des classes et des instances contraints.

Tous les avantages du couplage semi-faible sont ici perdus et il faut revenir à l'emploi d'outils permettant d'assurer un couplage faible. Aussi, une table de correspondance doit être mise en place par l'interface TROPES/MICRO entre les expressions des contraintes et les instances de contraintes créées pour le calcul de type, ainsi qu'une table de correspondance entre les accès des contraintes et les VCM de MICRO qui représentent ces accès.

Les contextes de pose de ces contraintes sont détenus par les instances de contraintes correspondantes. Les tables de correspondance contraintes/instance de contraintes et VCM/accès sont mises à jour lors de l'ajout ou du retrait d'une contrainte.

Lorsqu'une contrainte est déclarée, l'interface procède à la lecture de l'expression contrainte équivalente et crée pour chaque argument non constant une VCM dont le domaine est équivalent au domaine de la valeur de l'accès.

9.5 Les fonctionnalités de l'interface

L'interface TROPES/MICRO (*cf.* figure 9.6) contient l'ensemble des fonctions qu'il a fallu écrire afin que le couplage semi-faible adopté soit possible. Ces fonctions ont été écrites à partir des fonctions fournies par l'interface fonctionnelle de MICRO et des fonctions de l'API de TROPES. L'interface, à son tour, propose des fonctions permettant de manipuler des CSP dans TROPES et étend donc l'API de celui-ci. Les fonctions de l'API de TROPES destinées aux contraintes sont les seules fonctions visibles (celles de plus haut niveau) de l'interface. Les autres fonctions sont dédiées à la gestion des CSP MICRO équivalents aux CSP TROPES définis. Pour la plupart ce sont des fonctions permettant que les primitives de MICRO soient appliquées non plus sur des VCM définies et maintenues par MICRO, mais sur des ACT définis et maintenus par TROPES.

Les fonctions de l'interface TROPES/MICRO forment donc une couche intermédiaire entre l'API de TROPES et celle de MICRO. Ces fonctions sont chargées de la mise à jour des informations relatives aux contraintes, dans TROPES comme dans MICRO, lors des divers événements de nature à étendre, réduire ou perturber les CSP TROPES maintenus par MICRO.

9.5.1 Création de contrainte

La création d'une contrainte peut être explicitement demandée par une fonction de l'API de TROPES ou être la conséquence de la création d'une instance TROPES dans un concept contraint ou du rattachement d'une instance à une classe contrainte.

Selon le niveau de représentation de la contrainte (concept, classe ou instance), l'interface doit valider le calcul du type du concept, de la classe ou de l'instance avant de passer à la pose effective de la contrainte. Le calcul du type des entités nécessite l'utilisation des fonctions de consultation et de mise à jour du module de gestion de types METÉO.

Créer une contrainte pour l'interface consiste, du côté de MICRO, à appeler la fonction de création de contrainte (les liens entre AIC seront mis en place par les fonctions de MICRO) et, du

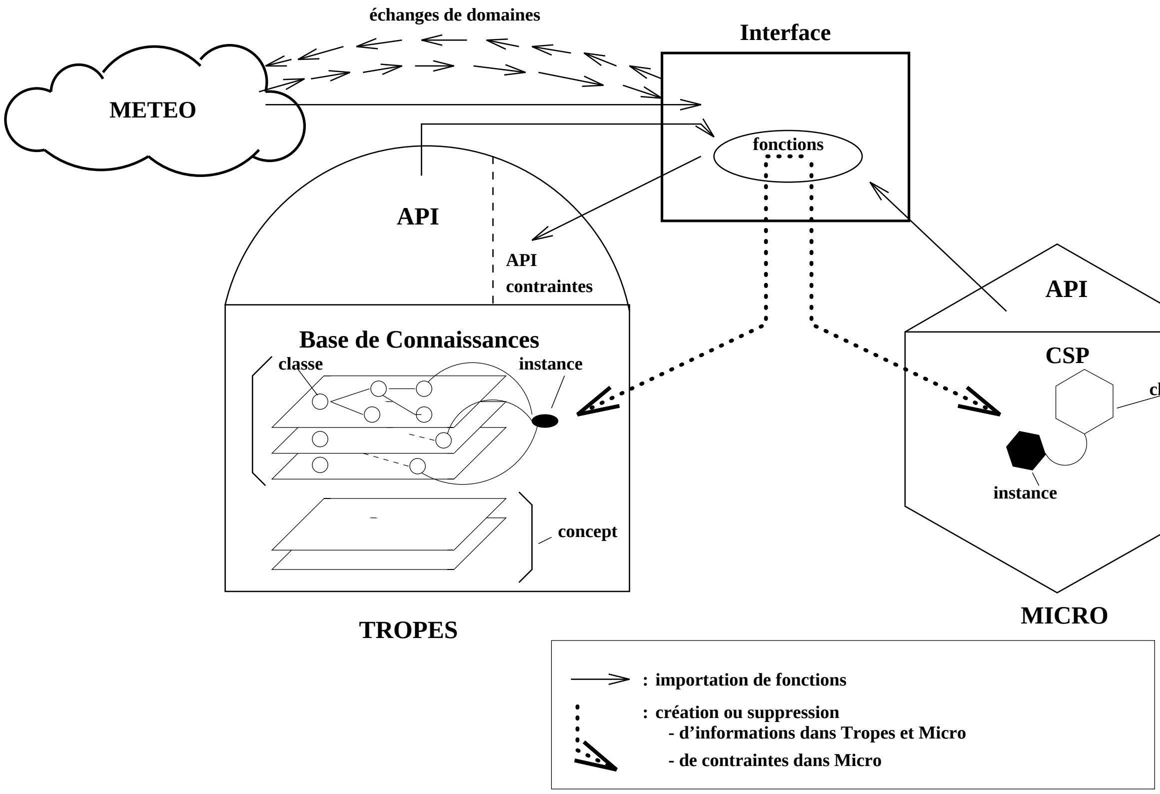


FIG. 9.6 - : Architecture du couplage TROPES/MICRO. L'interface se charge des communications entre TROPES et MICRO mais aussi des échanges de domaines entre METÉO et MICRO lors du calcul de type.

côté de TROPES, à mettre en place les liens ACT/AIC entre les attributs de l'instance de TROPES et ceux de l'instance de MICRO créée. La propagation peut alors être lancée (les méthodes de maintien de consistance associées à la contrainte sont activées) sur les CSP MICRO équivalents.

Si la propagation échoue, la contrainte n'est pas acceptée ; l'instance de contrainte est détruite dans MICRO, les liens ACT/AIC enregistrés dans TROPES sont supprimés.

Si la propagation réussit, les informations complémentaires (liste des contraintes, liste des instances de contraintes) sont placées dans les instances TROPES concernées.

Les fonctions de création de contrainte qu'utilise l'interface TROPES/MICRO ne sont pas exactement celles de MICRO, le code en a été légèrement modifié pour substituer les VCM par des ACT pour le couplage et pour récupérer les domaines des attributs auprès du module de gestion de types METÉO. Le principe de création d'une instance de contrainte dans MICRO (*cf.* section 8.4.1) reste cependant le même.

9.5.2 Suppression de contrainte

La suppression d'une contrainte peut être explicitement demandée par une fonction de l'API de TROPES ou être la conséquence de la suppression d'une entité TROPES.

Comme pour la création, selon le niveau de la contrainte, un calcul de type nécessitant une communication entre MICRO et METÉO est préalablement requis.

Supprimer une contrainte pour l'interface consiste, du côté de MICRO, à appeler la fonction MICRO de suppression de contrainte et, du côté de TROPES, à supprimer les liens ACT/AIC entre les attributs de l'instance de TROPES et ceux de l'instance de MICRO supprimée.

La suppression d'une contrainte est faite selon le principe exposé (*cf.* section 8.4.2). Les contraintes suivantes sont supprimées en bloc puis reposées à partir du contexte de pose de la contrainte supprimée qui aura été rétabli à partir du contexte de pose de l'instance de contrainte correspondante conservée par MICRO.

9.5.3 Modification d'un ACT

La modification d'un ACT, lorsqu'elle n'intervient pas pendant la propagation de contraintes, nécessite de restaurer des domaines des ACT non valués par l'utilisateur et appartenant au même réseau, avec le domaine fourni par le type de l'entité, et de vider les domaines des AIC internes. Dès lors, l'interface cherche à établir la consistance locale du réseau pour valider la modification (*cf.* section 8.4.3).

Les fonctions de restauration de contexte de MICRO sont utilisées et légèrement modifiées pour traiter des ACT et non pas des VCM.

9.5.4 Inférence de la valeur d'un ACT et domaine effectif

L'interface est chargée de mettre à jour la valeur d'un ACT et, par la même occasion, le drapeau booléen signalant une inférence, lorsque le domaine de l'ACT est réduit à un singleton par la propagation de contraintes.

Le domaine initial ou de définition d'un ACT est fixé par le calcul des types (qui tient compte des contraintes présentes sur l'ACT).

Le domaine effectif est un sous-ensemble du domaine de définition. À tout moment, il est le plus grand domaine localement consistant pour l'ACT en fonction des informations disponibles et dans la limite des règles de consistance de MICRO. Ce domaine effectif est commun à tous les AIC associés à l'ACT. Comme pour les VCM, bien que sa présence au niveau de l'ACT ne soit pas requise, il y a été placé pour information.

9.5.5 Définition de nouvelles contraintes

L'interface doit être également en mesure d'accepter des définitions de nouvelles contraintes. L'idée est la même que celle proposée par MICRO : factoriser et stocker une expression complexe de contraintes afin de pouvoir directement invoquer cette contrainte et non plus l'ensemble des contraintes complexes ou de base qui composent cette expression.

La fonction de l'API (**tr-create-constraint** *constraint-name constraint-arg-list constraint-expr-list*) est la fonction de création d'une nouvelle contrainte. Cette contrainte vient s'ajouter à l'ensemble des contraintes de base fournies par MICRO et à l'ensemble des contraintes déjà définies via l'interface.

Le rôle de l'interface consiste donc à substituer toute expression contrainte contenant cette contrainte par l'ensemble de contrainte complexes ou de base la composant, les expressions de contraintes complexes étant récursivement décomposées.

Par exemple[1], on peut signifier que deux objets du concept TRIANGLE sont égaux par la contrainte :

(`tr-create-constraint` `même-triangles` ((T1 : Triangle) (T2 : Triangle)) ((`même-points` T1.p1 T2.p1) (`même-points` T1.p1 T2.p1) (`même-points` T1.p1 T2.p1)))

où $p1, p2, p3$ sont trois attributs de la classe Triangle désignant les trois points d'un triangle et `même-points` est une contrainte, elle aussi construite, imposant l'égalité de deux objets du type Point

(`tr-create-constraint` `même-points` ((P1 : Point) (P2 : Point)) ((`mic-eq` P1.x P2.x) (`mic-eq` P1.y P2.y)))

où x, y correspondent aux attributs du concept POINT décrivant respectivement l'abscisse et l'ordonnée d'un point.

[1] Extrait de [Kökény94].

Lorsque la contrainte `même-triangles` est rencontrée, sous réserve d'être syntaxiquement correcte, l'expression est remplacée dans un premier temps par trois contraintes `même-points` puis, finalement par six contraintes `mic-eq`.

9.5.6 Contraintes et vérifications

L'interface est également chargée de la vérification syntaxique de toute expression de contrainte proposée dans TROPES.

L'analyse syntaxique d'une expression de contrainte consiste à s'assurer que chaque contrainte principale (et secondaire) qu'elle contient est bien dans l'ensemble des contraintes prédéfinies de MICRO ou dans l'ensemble des contraintes construites par le biais de la fonction *tr-create-constraint*. Lors d'une double définition, on considère que la dernière définition de l'utilisateur (gérée par l'interface) fait loi.

Une vérification est également opérée sur chaque accès argument d'une contrainte. Pour que l'accès soit bien défini, il faut que la suite des attributs proposée soit cohérente. Ainsi, à partir du niveau de représentation auquel est attachée la contrainte, il doit être possible d'atteindre l'attribut destination en suivant dans l'ordre les attributs étapes. Pour deux attributs étapes consécutifs, il suffit de s'assurer que le second attribut figure bien dans le concept, la classe ou l'ensemble de classe désigné par le type du premier.

Ensuite une vérification de type est effectuée sur chaque accès ou constante argument d'une contrainte. L'interface, qui gère également les accès, s'assure que le type de l'accès est le même que le type attendu pour l'argument de la contrainte.

Lorsqu'une de ces vérifications échoue, l'expression contrainte n'est pas valide et la fonction à réaliser est refusée.

9.6 Comparaison avec d'autres associations objets/contraintes

Le couplage TROPES/MICRO est un couplage semi-faible. Parmi les langages alliant contraintes et objets présentés au chapitre 5, seul le langage SOCLE est l'exemple d'un couplage entre un système de programmation par contraintes et un langage de représentation par objets. Cependant, le couplage en question est un couplage faible et nettement plus rudimentaire que celui proposé ici, CONSTRAINTS n'assurant que la propagation de valeurs sur des attributs à domaines finis.

Du point de vue de la représentation des contraintes, contraintes et objets ne sont pas mélangés au sein d'une base de connaissances TROPES. Ceci en raison de la préservation de l'uniformité des droits d'accès ou de manipulation des objets qui est incompatible avec la sémantique de la déclaration d'une contrainte à un certain niveau de représentation. Aussi, au premier abord et du point de vue du couplage, une comparaison avec les systèmes PROSE [Berlandier92b], CSPOO [Kökény94] ou COOL [Avesani et al.90] peut être écartée. Cependant, les contraintes étant représentées par des classes du langage hôte dans lequel est écrit MICRO, on retrouve pour MICRO une architecture pour contraindre les objets TROPES voisine de celles adoptées par ces systèmes. Les différences se situent au niveau de la description des variables. Alors que celles-ci sont représentées par des objets à part entière dans ces trois systèmes, elles sont shuntées à travers le couplage semi-faible réalisé ici, et ce sont les attributs de TROPES qui prennent en charge le stockage de l'information.

Alors que PROSE et COOL ne traitent que les CSP à domaines finis, MICRO gère des CSP à intervalles finis ou infinis. Alors que CSPOO est en quelque sorte un langage pour décrire des CSP et laisser à l'utilisateur le soin d'indiquer (et d'écrire !) les méthodes de consistance ou de résolution qui leur seront appliquées, MICRO, comme PROSE ou COOL, permet de décrire et de résoudre des CSP par des méthodes et algorithmes bien définis, dont l'extension se fait "en dur" dans le code du module, et non pas par spécification d'un attribut d'une classe *Solveurs*. MICRO n'a pas été non plus conçu pour la définition de contraintes par extension, ce qui, par exemple, le différencie de

CSPOO qui propose cette possibilité.

En revanche, comme CSPOO, MICRO exploite les méthodes attachées aux classes de contraintes pour établir la consistance locale. Enfin, la différence essentielle entre PROSE, CSPOO, COOL d'un côté, et le couplage TROPES/MICRO de l'autre, est que ce dernier n'est pas destiné à constituer une boîte à outils de programmation par contraintes dans laquelle les CSP (composants) sont décrits à l'aide d'objets qui sont appelés à cohabiter avec les objets sur lesquels portent les contraintes. MICRO est plutôt destiné à gérer la présence de contraintes dans une base de connaissances, sans introduire d'objets supplémentaires. Dans l'avenir, il est envisageable d'utiliser TROPES pour permettre la définition de nouvelles contraintes. On utilisera alors ce système pour ses vraies fonctions de représentation afin de décrire les arguments de la contrainte et ses règles de consistance. Mais cette fonctionnalité – un langage de définition de contraintes – peut également être ajoutée à MICRO indépendamment de TROPES. Outre les compétences techniques spécifiques (types des domaines traités, méthodes de résolution, dynamicité, etc.) des résolveurs de chacun de ces systèmes qui permettent déjà de distinguer le couplage TROPES/MICRO présenté des autres associations objets/contraintes, une autre différence importante réside dans le type de couplage choisi : semi-faible contre fort. L'objectif ici n'était pas de décrire des CSP à l'aide d'objets TROPES, mais de gérer des CSP sur les objets TROPES.

9.7 Conclusion

L'interface TROPES/MICRO est un ensemble de fonctions qui permettent de réaliser un couplage semi-faible entre TROPES et MICRO dans lequel les VCM sont remplacées par les ACT. La liaison TROPES/MICRO implique le stockage d'un certain nombre d'informations aux différents niveaux de représentation contraignables. Chaque instance de TROPES comporte un certain nombre de structures de données – accessibles ou non par l'utilisateur – destinées à la gestion des contraintes qui portent sur ses attributs. Ces champs d'informations sont mis à jour par l'interface. La partie visible de l'iceberg est constituée par les fonctions de l'API de TROPES qui permettent la définition et la suppression des contraintes. La partie immergée de l'iceberg comporte les fonctions de l'interface qui reposent essentiellement sur les fonctions de MICRO mais aussi sur les fonctions de l'API permettant la consultation et la modification des informations détenues par le module de gestion de types METÉO.

Cette interface fonctionnelle permet donc de définir et de modifier à tout moment des CSP et de demander la résolution de ces CSP. C'est par elle que l'utilisateur (ou un programme) communique avec MICRO qui est chargé de la représentation effective, de la maintenance et de la résolution des CSP définis sur les objets TROPES.

Avec ce couplage, nous obtenons un modèle TROPES étendu, dont les objets peuvent accueillir des contraintes. Maintenant que les contraintes sont dans TROPES un moyen d'enrichir la représentation des connaissances, nous devons nous intéresser aux conséquences pour le modèle de leur introduction dans les objets.

Chapitre 10

Contraintes et comportement du modèle

Dans les chapitres précédents, nous avons tout d'abord mis en évidence le besoin d'introduire des contraintes dans TROPES (*cf.* chapitre 3) pour en augmenter l'expressivité. Nous avons défini ensuite quels étaient les composants des CSP TROPES et avons attaché une portée à la contrainte selon le niveau de représentation auquel elle se trouve (*cf.* chapitre 6). Nous avons également étudié quelles pouvaient être les répercussions d'une utilisation dynamique des contraintes sur le module de types. Les choix effectués alors nous ont amené à construire un module de gestion de contraintes adapté aux différents aspects de l'intégration de contraintes proposée dans TROPES.

Aussi, il reste à mesurer les conséquences de l'intégration de contraintes sur le comportement du modèle. Ce chapitre vise à compléter les précédents en s'intéressant à l'adaptation du modèle TROPES aux contraintes.

Cette adaptation concerne tout d'abord le fonctionnement des mécanismes d'exploitation du système. L'instanciation (section 10.1), le rattachement (section 10.2), la classification (section 10.3) et le détachement procédural (section 10.4) sont maintenant décrits en présence de contraintes.

L'ajout ou le retrait de contrainte sur les classes appartenant à des passerelles doit lui aussi être considéré pour préserver la sémantique de ces relations entre entités de représentation (*cf.* section 10.5).

La notion d'accès a été introduite pour le besoin des contraintes. Il reste à définir son comportement en tant que nouvelle entité de description de TROPES (*cf.* section 10.7) lors de tout changement dans la base de connaissances le concernant.

Enfin, nous traitons des divers aspects (consistance, incohérence et redondance, résolution) relatifs aux CSP et ayant un effet direct sur la cohérence d'une base de connaissances (*cf.* section 10.8).

10.1 Contraintes et instanciation

Lors de l'*instanciation* (ou création d'une instance d'un concept), les seules contraintes *a priori* concernées sont les contraintes de concept. Autrement dit, la création de l'instance dépend, d'une part de la conformité des valeurs proposées aux types pour le concept de ces attributs et, d'autre part, de la satisfaction des contraintes de concept. Si une des contraintes de concept n'est pas satisfaite, la création de l'instance ne peut être acceptée car l'instance ne possède pas les caractéristiques du concept à instancier.

La vérification des contraintes de concept se fait en propageant les contraintes à partir du contenu de l'instance. Si cette propagation échoue c'est que deux au moins des valeurs d'attributs de l'instance impliquées dans une même contrainte ne font pas partie d'une instanciation localement consistance pour cette contrainte. Si la propagation n'échoue pas, l'instanciation est validée. Si la propagation donne lieu à une inférence de valeur d'attribut, l'instance est complétée avec cette valeur.

Le comportement du système doit être le même en cas de non respect des contraintes de concept qu'en cas de non conformité des valeurs au type calculé à partir des facettes. En cas d'échec, soit la création est refusée purement et simplement, soit l'utilisateur peut être sollicité pour fournir de nouvelles valeurs pour l'instance qui satisferont la description donnée dans le concept.

Si la création de l'instance est acceptée, l'instance a le droit d'existence dans le concept. Lors de l'instanciation, il peut être proposé des classes de *rattachement* pour l'instance, au plus une par point de vue. Par défaut – si aucune classe n'est proposée pour le rattachement dans un point de vue – la classe racine est choisie par le système, car son extension doit contenir toutes les instances du concept, son intension reprenant celle du concept. Nous étudions la part des contraintes dans le processus de rattachement.

10.2 Contraintes et rattachement

Le rattachement d'une instance à une classe suit donc immédiatement toute instanciation. Il peut aussi intervenir à n'importe quel moment de la vie d'une instance. Il reflète dans ce cas une tentative de création d'un nouveau lien d'appartenance pour l'instance qui correspond à une *migration* de l'instance (changement de classe d'appartenance dicté par l'utilisateur).

Pour valider le rattachement d'une instance à une classe dans le cas d'une instanciation, le système s'assure, d'une part que les valeurs fournies sont conformes aux types de ces attributs dans la classe et, d'autre part, que les contraintes de classe sont satisfaites. Si l'une de ces deux conditions n'est pas respectée, le rattachement est refusé.

La vérification des contraintes de classe se fait de la même manière que celle des contraintes de concept décrite plus haut.

Dans le cas d'un refus, nous devons considérer les deux contextes du rattachement :

1. L'instance vient d'être créée, une classe est proposée pour le rattachement. Si les valeurs ne sont pas conformes aux types, le rattachement est refusé. Il est possible dans ce cas de tenter une classification de l'instance plus haut dans la hiérarchie. Si les valeurs sont conformes aux types, mais que l'une au moins des contraintes de la classe n'est pas satisfaite, on peut dire que l'instance n'est pas une instanciation localement consistante des contraintes de classe. On peut tenter une classification de l'instance, car elle ne viole pas les contraintes du concept puisque l'instanciation a été validée.
2. On vient de rompre le précédent lien d'appartenance de l'instance dans le point de vue, sans changer ses valeurs. Si les valeurs de l'instance ne sont pas conformes à la nouvelle classe proposée, on peut tenter une classification de l'instance ou revenir à la situation précédente. L'ancien lien n'est rétabli que si le rattachement est possible et confirmé.

Dans le cas où l'instance est rattachée pour signifier une migration, les contraintes de l'ancienne classe de rattachement de l'instance sont supprimées. Par contre, les contraintes de concept et d'instances subsistent.

10.3 Contraintes et classification

Il existe deux types de classification dans TROPES : la classification d'instance et la classification de classe. Nous étudions pour chacune d'elles le rôle que doivent jouer les contraintes.

10.3.1 Contraintes et classification d'instance

Disposant d'une instance (avec éventuellement posées sur elle des contraintes de concept, des contraintes de classe et des contraintes d'instances), on cherche à déterminer pour cette instance, l'ensemble des classes *possibles*, l'ensemble des classes *inconnues* et l'ensemble des classes *impossibles*. Informé sur le contenu de ces trois ensembles, l'utilisateur pourra alors prendre la décision de rattacher l'instance à une des classes marquée *possible* ou *inconnue*.

La classification (mono-point de vue) consiste à faire descendre l'instance dans la hiérarchie. À chaque étape, on compare le contenu de l'instance avec la description de la classe.

En l'absence de contraintes, si le contenu de l'instance respecte la description de la classe (formée de l'ensemble des définitions des attributs), la classe est étiquetée *possible*, on cherche à déterminer les étiquettes des sous-classes (sous hypothèse d'exclusivité, les sœurs de cette classe sont marquées comme *impossible*).

Si le contenu de l'instance respecte la description de la classe mais n'est pas complet (des valeurs d'attributs sont manquantes), la classe est étiquetée comme *inconnue*, on cherche à déterminer les étiquettes (*inconnue, impossible*) des sous-classes.

Si le contenu de l'instance ne respecte pas la description de la classe, la classe et ses sous-classes sont étiquetées comme *impossible*, on peut passer à une classe sœur.

En présence de contraintes dans les classes, le même schéma de parcours s'applique, seul le test doit être modifié. La description de la classe est maintenant formée de l'ensemble des définitions des attributs de la classe (hérités, redéfinis ou introduits) et de l'ensemble des contraintes. Aussi,

- une classe est marquée *possible* si les valeurs de l'instance sont conformes aux types des attributs pour la classe – qui prennent eux-mêmes en compte la présence de contraintes – et si, de plus, elles satisfont les contraintes de la classe ;
- une classe est marquée comme *inconnue* si les valeurs de l'instance sont conformes aux types des attributs pour la classe et si, de plus, elles satisfont les contraintes de la classe qui les impliquent. C'est seulement l'absence de certaines valeurs qui empêche de trancher définitivement ;
- une classe est marquée comme *impossible* si une au moins des valeurs de l'instance n'est pas conforme au type attendu, ou si un sous-ensemble de ces valeurs ne constitue pas d'instanciation localement consistante pour une des contraintes de la classe.

On peut donc décomposer la confrontation du contenu d'une instance à la description d'une classe en deux temps :

1. la vérification des valeurs (vérification de types) ;
2. la vérification des contraintes de classes.

En supposant que la première étape ait réussi, nous nous intéressons plus particulièrement ici à la deuxième étape. Elle s'appuie sur la règle suivante :

Une contrainte *n-aire* de la classe peut être testée directement si les n attributs sur lesquelles elle porte ont une valeur. Autrement dit, s'il existe une telle contrainte dans la classe qui ne soit pas satisfaite, alors la classe est marquée comme *impossible*. Pour que la classe soit marquée *possible*, il faut donc que toutes les contraintes aient des attributs arguments valués et qu'elles soient toutes satisfaites. Pour les contraintes que l'on ne peut tester directement car certains de leurs attributs arguments n'ont pas de valeur, il faut procéder à une propagation à partir du contenu de l'instance qui déterminera si la classe est *possible* (la propagation termine sans vider de domaines, les valeurs forment une instanciation partielle consistante) ou *impossible* (la propagation a vidé un domaine ou ne termine pas, les valeurs forment une instanciation partielle mais déjà inconsistante).

À cette règle on peut ajouter l'inhibition du pouvoir inférant des contraintes qui consiste à restreindre le rôle des contraintes de classe lors de la classification à celui de simples prédicats. Sa justification est proche de celle avancée pour le détachement procédural. En effet, si la contrainte est fonctionnelle[1] (chaque argument peut être calculé en fonction des autres), alors la présence de $n-1$ valeurs pour une telle contrainte $n-aire$ est suffisante pour déduire la valeur manquante. Le contenu de l'instance peut donc être étendu de manière cohérente. Cependant, le statut de cette valeur inférée n'est reconnu que si l'instance peut appartenir à la classe – elle devra en satisfaire toutes les autres contraintes – et si, de plus, l'appartenance à cette classe est effectivement décidée.

Ainsi, autoriser le recours au pouvoir inférant des contraintes lors de la classification peut conduire à étendre le contenu de l'instance et à s'appuyer sur les valeurs inférées pour décider du caractère sûr de la classe vis-à-vis de l'appartenance. Nous pensons que ce recours ne peut et ne doit être effectué que lorsque le lien est fixé par l'utilisateur et validé par le système, et non pas lors d'une tentative de détermination. Enfin, un dernier argument plaide en faveur de cette utilisation prédicative des contraintes lors de la classification : si on tente de rattacher l'instance "ailleurs" (sur une branche n'appartenant pas au sous-arbre de la classe) dans la hiérarchie, il faut annuler ces inférences – ce qui peut s'avérer coûteux – pour, peut-être, les reproduire une fois le lien fixé définitivement.

En ce qui concerne la classification multi-points de vue, le principe de confrontation expliqué ici est repris dans tout point de vue auquel ce mécanisme accède.

10.3.2 Contraintes et classification de classe

La classification de classe consiste à déterminer les places possibles d'une classe dans chaque point de vue du concept. Ces positions résultent du placement du type de la classe dans le treillis des types des classes du concept.

Lorsque la description de la classe comporte des contraintes de classe, il faut procéder à la propagation de ces contraintes afin de déterminer le nouveau type de la classe.

Si cette propagation débouche sur une inconsistance (un domaine d'attribut contraint est vide), alors la classe ne peut exister. Sinon, la classification de la classe consiste à déterminer la position de son nouveau type (tenant compte des contraintes) dans le treillis des types des classes du concept.

Une fois une position choisie par l'utilisateur, la classe est insérée dans la hiérarchie, elle doit alors propager ses contraintes à ses sous-classes. S'il s'avère que cette propagation débouche elle aussi sur une inconsistance, la position ne peut être acceptée.

Que ce soit pour les instances ou pour les classes, il faut rappeler que les types des instances et des classes conservés par METÉO dans des treillis contiennent des domaines qui reflètent la présence de contraintes, c'est-à-dire les réductions ou extensions inférées par MICRO lors d'ajouts ou de retraits de contraintes. Aussi, vis-à-vis de la classification, un pré-filtrage des possibilités a été opéré à travers l'échange MICRO/METÉO.

10.4 Contraintes et détachement procédural

Le concept peut fournir un arbre de méthodes pour le calcul de la valeur d'un attribut. L'idée est d'organiser à l'aide de TROPES un ensemble de méthodes selon divers critères (information disponible, liens d'appartenance de l'instance...) permettant d'établir une hiérarchie de méthodes qui peut même être alors considérée depuis plusieurs points de vue (efficacité, précision...).

En l'absence de contraintes, ces méthodes sont déclenchées lorsque la valeur de l'attribut est demandée. Si l'utilisateur fournit une valeur valide pour un attribut, un tel appel de méthodes ne se produit pas. L'utilisateur est donc ici considéré comme un moyen d'inférence externe prioritaire sur les méthodes.

En présence de contraintes, l'ordre d'inférence de valeurs dans TROPES est le suivant : utilisateur, contraintes, méthodes. En effet, un attribut contraint a, soit une valeur, soit celle-ci n'est pas encore définie. S'il a une valeur, elle a pu être fournie par l'utilisateur ou bien inférée par le réseau de

[1] Ou si la configuration est favorable pour une contrainte non fonctionnelle.

contraintes associé à cet attribut parce que les informations pour qu'une complétion conforme aux contraintes ait lieu, étaient disponibles.

S'il n'a pas de valeur, c'est que l'utilisateur n'a pas pu en fournir et que le réseau de contraintes ne permet pas d'en inférer. Il reste alors la méthode. Celle-ci est activée et son résultat, s'il y en a un, doit à son tour être accepté par la propagation de contraintes.

Ce mode de fonctionnement peut néanmoins déboucher sur deux situations conflictuelles après le déclenchement de la méthode :

- soit on obtient une valeur – par hypothèse, quelle que soit la méthode employée, il n'y en a qu'une – qui ne se trouve pas dans le domaine de l'attribut contraint ou ne satisfait pas l'ensemble des contraintes de l'attribut – la méthode est inadaptée à l'ensemble de contraintes. La valeur est écartée, les méthodes bien qu'activées ont été inopérantes. Il n'y a pas là forcément d'erreur de conception, simplement la configuration de la base et l'introduction de relations entre les attributs font des méthodes un moyen d'inférence non garanti ;
- soit la méthode nécessite la valeur d'autres attributs (qui sont ses arguments) dont les méthodes sont à leur tour déclenchées. À un moment, une propagation de contraintes est déclenchée qui finalement détermine une valeur pour l'attribut. La valeur déterminée par la méthode, si elle existe, ne sert à rien et doit s'effacer devant la valeur inférée par la contrainte si celle-ci est viable.

La priorité a été donnée ici aux contraintes sur les méthodes en tant que moyen d'inférence. Elle n'est qu'arbitraire. Une étude approfondie sur la co-existence contraintes/méthodes en fonction d'un ordre de priorité quelconque est faite dans la thèse de Pierre Girard [Girard95] dans le cadre du contrôle de l'évaluation d'un attribut.

10.5 Contraintes et passerelles

Les passerelles peuvent être vues comme des contraintes sur des extensions de classes. Dans Tropes, trois types de passerelles ont été proposés dans [Mariño93].

- la passerelle *unidirectionnelle mono-source* impose que chaque instance de la classe source soit aussi instance de la classe destination ;
- la passerelle *unidirectionnelle multi-sources* impose qu'une instance appartenant à chacune des classes sources soit aussi instance de la classe destination ;
- la passerelle *bi-directionnelle mono-source* impose une égalité entre l'extension de la classe source et l'extension de la classe destination.

Les passerelles sont déclarées dans le concept. En l'absence de contraintes dans les classes (source(s) ou destination) de la passerelle, la cohérence de la passerelle est gérée tout d'abord par le module de types Metéo.

- Pour toute passerelle unidirectionnelle mono-source, Metéo assure que, pour un attribut apparaissant à la fois dans la classe source et dans la classe destination, le type de cet attribut dans la source est un sous-type du type de l'attribut dans la classe destination.
- Pour toute passerelle unidirectionnelle multi-sources, Metéo assure que, pour un attribut apparaissant à la fois dans une des classes sources et dans la classe destination, l'intersection des types de cet attribut dans ces classes sources est un sous-type du type de cet attribut dans la classe destination.
- Pour toute passerelle bi-directionnelle mono-source, Metéo assure qu'un attribut apparaissant à la fois dans la classe source et dans la classe destination a le même type dans les deux classes.

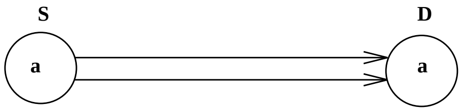
Nous proposons d'intégrer un quatrième type de passerelle : la passerelle *bi-directionnelle multi-sources* qui impose que chaque instance de la classe destination soit instance de chacune des classes sources. Le module de types Metéo devra assurer qu'un attribut apparaissant dans la classe destination a pour type un sous-type de l'intersection des types de cet attribut dans les diverses classes sources où il apparaît ;

En présence de contraintes, le type d'une classe est susceptible d'être réduit. Il nous faut étudier dans chacun des cas précédents comment la sémantique de la passerelle peut être préservée, dans le cas d'un ajout comme dans le cas d'un retrait de contrainte.

10.5.1 Passerelle et ajout de contrainte

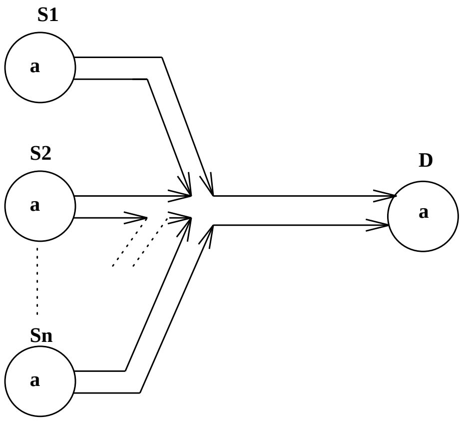
Nous étudions les conséquences d'un ajout de contrainte pour chacun des sept types de passerelles présentés. Les tableaux suivants reportent, dans ces cas, la répercussion de l'ajout sur la ou l'une des sources et la ou l'une des classes destinations.

Notation : Dans cette section, le type d'un attribut a dans une classe C est noté $T(a, C)$



$T(a, S) \subseteq T(a, D)$
si $c \in contr(S)$ **alors** ok
si $c \in contr(D)$ **alors** $c \in contr(S)$

FIG. 10.1 - : Passerelle unidirectionnelle mono-source. La pose d'une contrainte sur S n'a aucun effet sur D. Si une contrainte est posée sur D, elle est posée sur S également. Si elle n'est pas viable sur S, elle est refusée également sur D.



$\bigcap_{S_i | a \in attr(S_i)} T(a, S_i) \subseteq_T T(a, D)$
si $c \in contr(S_i)$ **alors**
  si $\bigcap_{S_i | a \in attr(S_i)} T(a, S_i) = \emptyset$ **alors** refusée
  sinon rien
si $c \in contr(D)$ **alors**
  si $\bigcap_{S_i | a \in attr(S_i)} T(a, S_i) \cap T(a, D) = \emptyset$ **alors** refusée
  sinon
    $i \in S_1 \wedge i \in S_2 \wedge \ldots \wedge i \in S_n \wedge$ (i satisfait c)$\Rightarrow i \in D$
    avec i instance

FIG. 10.2 - : Passerelle unidirectionnelle multi-sources. La pose d'une contrainte c sur une source S_i n'a aucun effet sur D mais peut être refusée. Si une contrainte est posée sur D, elle peut être refusée si l'attribut ne trouve pas de valeur commune dans les sources et la destination. La contrainte de D n'est pas posée systématiquement sur chacune des sources (la contrainte doit être vérifiée par l'intersection des sources, non pas chacune d'elles), mais s'ajoute à la condition d'appartenance multiple. En revanche, le type associé à l'instance i tient compte de la contrainte c.

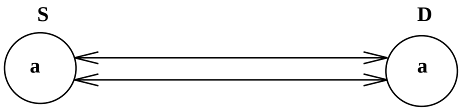
10.5.2 Passerelle et retrait de contrainte

Dans chacun des cas précédents où la pose d'une contrainte sur une classe entraîne la pose d'une contrainte sur l'autre extrémité de la passerelle, lorsque l'on procède au retrait de cette contrainte, il faut également la retirer de la classe sur laquelle elle a été posée pour les besoins de la sémantique de la passerelle. De même, le retrait d'une contrainte héritée par une passerelle est impossible (comme l'est le retrait d'une contrainte de classe dans une sous-classe).

Les passerelles n'admettent pas, pour l'instant, de définition dynamique. Elles sont fixées une fois pour toutes à la création du concept, ce qui permet à MÉTÉO d'opérer sur les types des classes pour maintenir la cohérence de la passerelle.

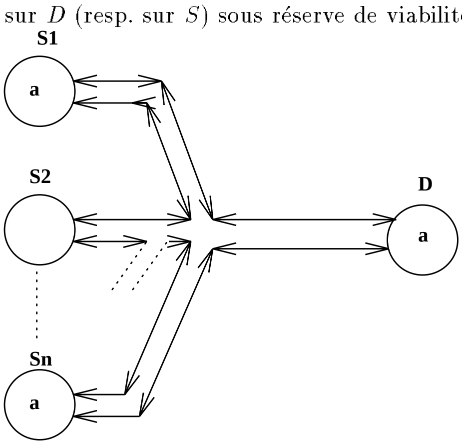
Il est important de noter que les algorithmes (donnés en 7.1) chargés du calcul des types des classes lors d'un ajout et d'un retrait d'une contrainte de classe, sont écrits sans tenir compte des passerelles pour des raisons de lisibilité.

Pour un ajout, à la rencontre d'une passerelle, il suffit de propager, s'il y a lieu, la contrainte vers l'autre point de vue. Pour un retrait, si la contrainte avait été héritée par une classe d'un autre



$T(a, S) = T(a, D)$
si $c \in contr(S)$ **alors** $c \in contr(D)$
si $c \in contr(D)$ **alors** $c \in contr(S)$

FIG. 10.3 - : Passerelle bi-directionnelle mono-source. La pose d'une contrainte sur S (resp. sur D) entraîne sa pose sur D (resp. sur S) sous réserve de viabilité.



$\bigcap_{S_i | a \in attr(S_i)} T(a, S_i) \subseteq T(a, D)$
si $c \in contr(S_i)$ **alors**
  si $\bigcap_{S_i | a \in attr(S_i)} T(a, S_i) = \emptyset$ **alors** refusée
  sinon $c \in contr(D)$
si $c \in contr(D)$ **alors**
  $i \in S_1 \wedge i \in S_2 \wedge \ldots \wedge i \in S_n \wedge (i \text{ satisfait c}) \Rightarrow i \in D$

FIG. 10.4 - : Passerelle bi-directionnelle multi-sources. La pose d'une contrainte c sur S (resp. sur D) entraîne sa pose sur D (resp. sur S) sous réserve de viabilité et sous réserve qu'elle conserve une intersection non vide des types des attributs communs dans les classes sources. La pose d'une contrainte sur D n'entraîne pas sa pose sur chacune des sources concernées mais la contrainte s'ajoute comme une condition à l'inclusion ensembliste. En revanche, le type associé à l'instance i tient compte de la contrainte c.

point de vue par l'intermédiaire d'une passerelle, il faut la supprimer dans ce point de vue aussi.

Afin de prévenir d'éventuels cycles dus aux passerelles entre les différents points de vue, on pourra adjoindre à ces algorithmes un marquage des classes visitées.

10.6 Méta-contraintes non déterministes

Il existe trois contraintes non-déterministes que MICRO met à la disposition de TROPES. Nous décrivons ici la sémantique de ces méta-contraintes et signalons les problèmes qu'elles introduisent au niveau du calcul du type des entités sur lesquelles elles portent.

10.6.1 Sémantique

Lorsqu'une méta-contrainte disjonctive `mic-meta-or` est présente dans un concept (respectivement dans une classe), alors toute instance de ce concept (respectivement de cette classe), doit vérifier l'une des deux contraintes. Aussi, il est possible que deux instances du même concept (respectivement de la même classe), contrainte par la méta-contrainte `mic-meta-or` vérifient chacune de leur côté une des deux contraintes induites par cette méta-contrainte.

Lorsqu'une méta-contrainte conditionnelle `mic-meta-if-then` est présente dans un concept (respectivement dans une classe), alors pour toute instance de ce concept (respectivement de cette classe), si la valeur de la variable booléenne qui reflète la condition vaut *vrai* alors l'instance doit vérifier la contrainte spécifiée par cette méta-contrainte `mic-meta-if-then` (dans le corps du *then*), sinon l'instance est libre. Aussi, il se peut qu'une méta-contrainte `mic-meta-if-then` ait un effet sur certaines instances du concept (respectivement de la classe) et pas sur d'autres, tout dépend du comportement de l'instance vis-à-vis de la condition.

Lorsqu'une méta-contrainte conditionnelle `mic-meta-if-then-else` est présente dans un concept (respectivement dans une classe), alors pour toute instance de ce concept (respectivement de cette classe), si la valeur de la variable booléenne qui reflète la condition vaut *vrai* alors l'instance doit vérifier la contrainte spécifiée par cette méta-contrainte `mic-meta-if-then-else` (dans le corps du *then*), sinon l'instance doit vérifier la contrainte spécifiée dans le corps du *else*. Aussi, une méta-contrainte `mic-meta-if-then-else` a un effet sur toutes les instances du concept (respectivement de la classe), mais la contrainte posée peut être différente d'une instance à l'autre et dépend de la condition.

10.6.2 Calcul de types

Les trois semi-méta-contraintes non-déterministes sont plus complexes à prendre en compte au niveau des types des entités de représentation qu'elles contraignent que les autres contraintes, en raison même de l'expression de la disjonction qu'elles sous-tendent.

En elles-mêmes, les méta-contraintes non déterministes n'ont pas d'effet sur le type des entités qu'elles contraignent, mais les contraintes auxquelles elles font référence, qui pourront être appliquées sur les extensions de ces entités, sont, elles, susceptibles de modifier ce type.

Trois comportements sont alors envisageables :

- ne pas tenir compte de l'incidence des contraintes référencées par les méta-contraintes dans le calcul de type. Du point de vue du type de l'entité contrainte par la méta-contrainte, cela signifie que la méta-contrainte (et ses contraintes) n'ont pas d'effet. La méta-contrainte (et la contrainte référencée) n'agira que comme un prédicat : on perd l'information statique que délivre la propagation de contraintes sur les types ;
- tenir compte de l'intersection des deux types issus des deux contraintes référencées (pour les méta-contraintes `mic-meta-or` et `mic-meta-if-then-else`) ou du type de la contrainte référencée (pour `mic-meta-if-then`). Se faisant, on va priver tout concept (ou classe) contraint par une telle méta-contrainte de certaines instances qui vérifient une des contraintes mais pas les deux (c'est le cas pour `mic-meta-or` et `mic-meta-if-then-else`) ou qui ne vérifient pas la contrainte mais qui n'ont pas non plus à la vérifier puisque la condition n'est pas respectée (c'est le cas pour `mic-meta-or` et `mic-meta-if-then-else`). En outre, cela contraint trop le type d'une instance et empêche certaines configurations. Ici, la disjonction se transforme en conjonction, ce qui n'est pas satisfaisant ;
- exprimer la disjonction induite par le non-déterminisme de ces méta-contraintes au niveau des types même des entités. S'il est possible d'avoir recours à une union de domaine lorsqu'un attribut de type simple (entier, réel, ...) est contraint par les contraintes apparaissant dans une méta-contrainte `mic-meta-or` ou `mic-meta-if-then-else`, pour les autres types cela est beaucoup plus compliqué. Une solution existe à travers la notion de type *variant* introduite par Cardelli *et al.* [Cardelli et al.85] qui désigne un type qui est en fait une disjonction d'autres types. La valeur d'un type variant est appelée à être valeur de l'un des types considérés par le type variant[2].

10.7 Gestion des accès

Un accès (*cf.* section 6.3) est à la fois une entité et une fonction de TROPES qui permet, à partir d'un objet, de désigner un attribut, appelé *destination*, dans un autre objet (mais pas forcément) en empruntant un chemin passant par un nombre (fini) d'attributs appelés *étapes*. En présence d'attributs étapes multivalués, l'accès diffuse, et des interprétations ont été associées aux opérateurs liant deux attributs étapes afin de définir la valeur finale de l'accès.

Cette section a pour but de compléter la présentation de la notion d'accès faite au chapitre 6, en s'intéressant plus particulièrement aux calculs de types mettant en jeu des accès, à la pose de contraintes avec accès, à la modification de valeurs d'attributs (étapes ou destinations d'accès) et aux inférences réalisées sur les attributs qui sont des destinations d'accès.

[2] À l'heure actuelle, cette fonctionnalité est à l'étude pour être prise en charge par METÉO.

10.7.1 Accès et calcul de type

Lorsqu'une contrainte est déclarée au niveau d'un concept, d'une classe ou d'une instance, une étape préliminaire à sa pose est le calcul du type de l'entité sur laquelle porte la contrainte. Le calcul du type de la valeur d'un accès est possible à partir de l'expression de l'accès et des interprétations données. Les opérateurs entre attributs étapes permettent de combiner les constructeurs de type, du premier attribut étape jusqu'à la destination.

Dans le cas qui nous intéresse ici, où la contrainte a pour argument un accès de longueur supérieure à un – l'accès comporte au moins une étape –, le calcul de type n'est pas aussi systématique que lorsque la contrainte implique uniquement des attributs de l'entité.

Pour s'en rendre compte, étudions le simple exemple suivant :

Exemple 10 *Soit la contrainte (`mic-eq` a.b c) posée sur une classe C contenant les attributs a et c tels que le type de a dans C est une classe C' (a est monovalué) et $dom(c) = [0, 12]$. C' contient l'attribut b tel que $dom(b) = [0, 30]$. Les règles de maintien de la consistance associées à la contrainte entraînent :*
$dom(b) = dom(b) \cap dom(c) = [0, 12]$*, et*
$dom(c) = dom(b) \cap dom(c) = [0, 12]$

En conclusion de cet exemple, le domaine (type) de c est inchangé alors que celui de b est réduit. Or, réduire le domaine (ou type) de b dans C' en raison d'une contrainte posée sur une autre classe C est abusif car c'est réduire les possibilités de valuation pour b aussi pour les instances de C' qui ne sont pas liées (via l'accès $a.b$) à une instance de la classe C.

Si, à présent, on considère la même contrainte avec $dom(b) = [0, 12]$ et $dom(c) = [0, 30]$, alors, cette fois, c'est le domaine de c (ou type) qui est réduit à $[0, 12]$. Mais le domaine réduit est cette fois celui d'un attribut de la classe qui doit vérifier cette condition pour toute instance. Autrement dit, dans toute instance i de C l'attribut c doit être égal à l'attribut b de l'objet valuant l'attribut a de i. Que la valeur de a soit définie ou non, c'est à cette condition que la valeur de c est acceptable.

De cet exemple on peut tirer une règle à appliquer pour le calcul de type inhérent à la déclaration d'une contrainte avec accès :

Le nouveau type d'un argument d'une contrainte sur une entité n'est pris en compte que lorsque la destination de l'accès correspondant appartient à l'entité. Pour tout argument ne vérifiant pas cette condition, le type est inchangé, même si la propagation de contraintes le réduit. Toutefois, si la propagation vide un des domaines, la contrainte est refusée.

10.7.2 Accès et pose de contraintes

La procédure de pose de contrainte doit traiter différemment les arguments qui sont des accès de longueur supérieure à un (ils sont dits *non-immédiats*) et ceux qui ne contiennent qu'un attribut destination. La principale raison de cette distinction est le fait que la valeur d'un accès peut être indéfinie lorsque la contrainte est posée.

Lorsque la valeur d'un accès est indéfinie, cela signifie que l'attribut (ou l'ensemble ou la liste d'attributs) destination(s) sur lequel porte la contrainte n'est pas connu ou qu'il est partiel parce que, suite à une diffraction, un des objets ne prolonge pas l'accès dans une des branches. On peut alors décider de retarder la pose de la contrainte jusqu'à ce que celui-ci soit déterminé. Nous faisons le choix de poser la contrainte et donc de créer l'instance de contrainte correspondante, puis de la compléter progressivement. Ainsi, lorsque pour un AIC, la valeur de l'accès non-immédiat correspondant devient déterminée, on connaît son domaine et son ou ses lien(s) vers le ou les ACT destination(s).

Le domaine d'un AIC correspondant à un accès non-immédiat est équivalent au domaine des valeurs de l'accès :

- lorsque l'accès est de type simple (ce n'est ni un ensemble, ni une liste), alors le domaine de l'AIC est le domaine de l'attribut destination de l'unique objet atteint ;

- lorsque l'accès est de type ensemble, alors le domaine de l'AIC est l'union des domaines de l'attribut destination dans l'ensemble des objets atteints. Il faut remarquer que ce domaine est alors un sur-domaine du domaine de valeurs réellement admissible, car les valeurs de l'attribut destination des objets atteints ne peuvent être distinguées dans un ensemble. Ainsi, si l'accès atteint un ensemble de trois objets $\{O_1, O_2, O_3\}$ dans lesquels l'attribut destination d a respectivement pour domaine $[0, 15], [0, 10], [0, 5]$, alors le domaine de l'AIC est l'union soit $[0, 15]$. Autrement dit, la valeur $\{12, 13, 14\}$ est acceptée pour l'AIC mais ne correspond pas à une instanciation possible de l'attribut destination des objets atteints ;
- lorsque l'accès est de type liste, alors le domaine de l'AIC est la liste des domaines de l'attribut destination dans l'ensemble des objets atteints.

Un AIC correspondant à un accès non-immédiat est lié, soit à l'attribut destination lorsque l'objet atteint est unique, soit à un ensemble d'attributs destinations lorsque l'accès atteint un ensemble d'objets, soit à une liste d'attributs destinations lorsque l'accès atteint une liste d'objets. Le lien AIC/ACT est donc un pointeur unique, un ensemble ou une liste de pointeurs.

Lorsque la valeur d'un accès non-immédiat n'est pas définie ou est incomplète, la contrainte est posée (l'instance correspondante est créée) mais l'AIC qui correspond à cet accès peut ne pas être lié à un ACT comme le montre la figure 10.5. Le lien AIC/ACT ne peut donc être complété que lorsque

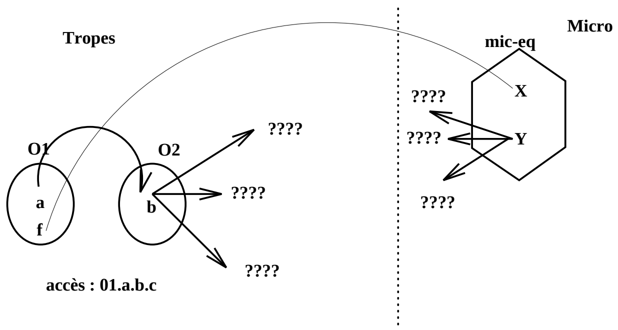


FIG. 10.5 - : La valeur de l'accès $a.b.c$ n'est pas définie à partir de l'objet $O1$ sur lequel porte la contrainte (`mic-eq` f $a.b.c$).

les objets atteints sont déterminés, que la valeur de l'attribut destination – donc de l'accès – soit définie ou non. Afin que ce lien soit mis à jour dès que possible, une propagation d'informations a lieu depuis chaque attribut étape valué en direction de l'attribut destination, et se prolonge tant que la valeur de chaque étape atteinte est définie. Lorsqu'un objet intermédiaire (contenant un attribut étape) est atteint, il reçoit et transmet une information à ses suivants. Cette information est un quadruplet contenant l'instance de contrainte contenant l'accès, l'AIC correspondant à l'accès, l'accès lui-même et le reste de l'accès à parcourir. Sur chaque attribut étape, cette information est utilisée lors d'une modification de domaine ou de valeur. En fin de chaîne, cette information est utilisée au niveau de la destination pour retrouver l'instance de contrainte à compléter.

Nous donnons ci-dessous la modification à apporter à la procédure de pose de contraintes au niveau du traitement de chaque argument de la contrainte, pour prendre en compte les accès non-immédiats :

dans la **Procédure** pose-contrainte(I : $instance, \mathcal{C}$: $contrainte$) insérer lors du traitement d'un *accès* argument de contrainte :

val $\longleftarrow$ $val(I.premier(acces))$
% on récupère la valeur du premier attribut %
liste-info-acces(I.premier(acces)) $\longleftarrow$ liste-info-acces(I.premier(acces))
% mise à jour du quadruplet d'information %
complet $\longleftarrow$ (val $\neq$?)
% la valeur de l'accès est-elle complète %
reste-acces $\longleftarrow$ reste(acces)

```
% on tient à jour le reste de l'accès %
tant que reste-acces et complet faire
    pour tout v ∈ val faire
        % mise à jour du quadruplet d'information %
        liste-info-acces(v.premier(reste-acces)) ← liste-info-acces(v.premier(reste-acces))
                ∪ (I_C, AIC, acces, reste-acces)
        si val(v.premier(reste − acces)) =? alors
        % la valeur de l'acces est incomplète %
            complet ← faux
        sinon
            val ← val cupval(v.premier(reste − acces))
        fin si
    fin pour
    reste-acces ← reste(reste-acces)
fin tant que
si ¬ reste-acces et complet alors
% on a atteint la destination %
    destination ← objets-atteints(acces)
    % calcul de la valeur de l'accès %
    si type(destination) = liste ou type(destination) = ensemble alors
        pour tout d ∈ destination faire
            % mise à jour de chacun des liens ACT/AIC des objets atteints avec l'AIC corres-
            pondant %
            lienACT/AIC(d) ← lienACT/AIC(d) ∪ AIC
        fin pour
    sinon
        lienACT/AIC(destination) ← lienACT/AIC(destination) ∪ AIC
    fin si
    % mise à jour du lien AIC/ACT de l'AIC avec la destination atteinte %
    lienAIC/ACT(AIC) ← destination.dernier(acces)
    % mise à jour du domaine de l'AIC (soit un domaine, soit une union) %
    dom(AIC) ← calcul-domaine(destination.dernier(acces))
fin si
```

Le contexte de pose d'une contrainte contenant des accès non-immédiats doit tenir compte des domaines de l'attribut destination dans l'objet ou l'ensemble d'objets ou la liste d'objets atteint(s) ainsi que des domaines des attributs présents dans le réseau au moment de l'activation de la contrainte.

Le contexte de pose d'une contrainte n'est donc calculé que lorsque celle-ci devient pour la première fois active, c'est-à-dire lorsque tous les AIC de l'instance correspondante ont un lien AIC/ACT défini.

Une instance de contrainte contenant des AIC correspondant à des accès non-immédiats peut être ou non active selon que les accès ont tous une valeur ou non. Lorsque tous les AIC correspondant à des accès non-immédiats ont une valeur, alors la contrainte est liée aux ACT destination de ses accès. Tant qu'il existe des accès non-immédiats sans valeur pour la contrainte, le contexte de pose de la contrainte ne peut être calculé (la contrainte existe mais n'est pas active). De même, la modification d'une valeur d'attribut étape peut entraîner la désactivation de la contrainte (si la destination ne peut être atteinte à partir de cette nouvelle étape). Tout se passe alors comme si la contrainte était supprimée.

10.7.3 Accès et modification de valeurs d'attributs

Nous traitons ici du cas où la valeur de l'un des attributs (étape ou destination) d'un accès est modifiée. Il faut distinguer le cas où la valeur fournie est la première valeur de l'attribut où s'il s'agit d'une réelle modification de la valeur.

Dans le premier cas, on cherche à atteindre l'attribut destination de l'accès, et à propager l'information nécessaire à l'établissement des liens AIC/ACT.

Dans le second cas (*cf.* figure 10.6), on cherche à éliminer l'information propagée par la valeur précédente, puis à atteindre la nouvelle destination, et à propager l'information nécessaire à l'établissement des liens AIC/ACT. En conséquence, nous devons modifier la procédure d'affectation

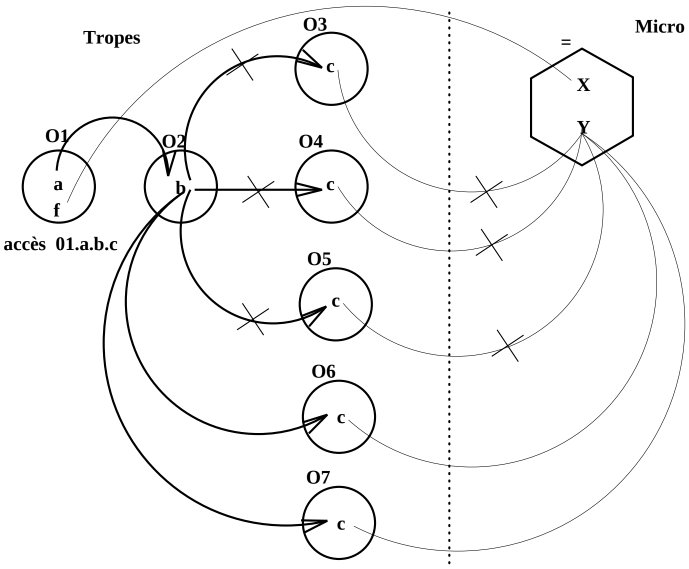


FIG. 10.6 - : La valeur de b est modifiée : l'accès *a.b.c* change de valeur. Les liens AIC/ACT et ACT/AIC sont mis à jour.

de valeur à un attribut de TROPES pour le cas particulier des attributs d'accès non-immédiats.

```
dans Procédure tr-set-value(I : instance, attr : attribut, valeur : valeur)
    pour tout info ∈ liste-info-acces(I.attr) faire
    % on utilise ici le quadruplet d'information associé à l'attribut %
        reste-acces ← info.reste-acces
        si ancienne-valeur alors
        % l'attribut avait une valeur %
            val ← ancienne-valeur
            complet ← vrai
            tant que complet et reste-acces faire
                pour tout v ∈ val faire
                % on supprime le quadruplet d'information relatif à cet accès %
                    liste-info-acces(v.premier(reste-acces)) ←
                        liste-info-acces(v.premier(reste-acces)) - info
                    si val(v.premier(reste − acces)) =? alors
                % la valeur de l'acces est incomplète %
                        complet ← faux
                    sinon
                        val ← val cupval(v.premier(reste − acces))
                    fin si
                fin pour
                reste-acces ← reste(reste-acces)
            fin tant que
            si ¬ reste-acces et completalors
                destination ← objets-atteints(acces)
                % on supprime les liens AIC/ACT et ACT/AIC %
                si type(destination) = liste ou type(destination) = ensemble alors
                    pour tout d ∈ destination faire
                        lienACT/AIC(d) ← lienACT/AIC(d) - (info.I_C,info.AIC)
```

```
                    fin pour
                sinon
                    lienACT/AIC(destination) ← lienACT/AIC(destination) - (info.I_C, info.AIC)
                fin si
                lienAIC/ACT(info.AIC) ← ()
                dom(info.AIC) ← ()
            fin si
            complet ← vrai
            reste-acces ← info.reste-acces
            val ← valeur
            tant que complet et reste-acces faire
                pour tout v ∈ val faire
                    liste-info-acces(v.premier(reste-acces)) ←
                        liste-info-acces(v.premier(reste-acces)) ∪ (info.I_C, info.AIC,info.acces, info.reste-
                acces)
                    si val(v.premier(reste − acces)) =? alors
                % la valeur de l'acces est incomplète %
                        complet ← faux
                    sinon
                        val ← val cupval(v.premier(reste − acces))
                    fin si
                fin pour
                reste-acces ← reste(reste-acces)
            fin tant que
        fin si
        complet ← vrai
        reste-acces ← info.reste-acces
        val ← valeur
        val(attr) ← val
        % on affecte à l'attribut la valeur val %
        % et on applique le même traitement que lors de la pose %
        tant que complet et reste-acces faire
            pour tout v ∈ val faire
                liste-info-acces(v.premier(reste-acces)) ←   liste-info-acces(v.premier(reste-acces))
                ∪ (info.I_C,info.AIC, info.acces, info.reste-acces)
                si val(v.premier(reste − acces)) =? alors
            % la valeur de l'acces est incomplète %
                    complet ← faux
                sinon
                    val ← val cupval(v.premier(reste − acces))
                fin si
            fin pour
            reste-acces ← reste(reste-acces)
        fin tant que
        si ¬ reste-acces et complet alors
            destination ← objets-atteints(acces)
            si type(destination) = liste ou type(destination) = ensemble alors
                pour tout d ∈ destination faire
                    lienACT/AIC(d) ← lienACT/AIC(d) ∪ (info.I_C, info.AIC)
                fin pour
            sinon
                lienACT/AIC(destination) ← lienACT/AIC(destination) ∪ (info.I_C, info.AIC)
            fin si
            lienAIC/ACT(info.AIC) ← destination.dernier(acces)
            dom(info.AIC) ← calcul-domaine(destination.dernier(acces))
        fin si
    fin pour
```

10.7.4 Accès et inférence

Selon le type de la contrainte, il est possible de réaliser des inférences à partir d'AIC correspondant à des accès non-immédiats sous certaines conditions :

- lorsque l'accès n'atteint qu'un seul objet, les conditions sur les inférences sont identiques au cas d'un accès immédiat : le domaine (ou la valeur) de l'AIC est transmis à l'ACT unique correspondant ;
- lorsque l'accès atteint un ensemble d'objets, l'inférence n'est possible que lorsque l'AIC a une valeur-ensemble de n éléments alors que seulement $n-1$ des n ACT auxquels il est lié ont une valeur. On peut, dans ce seul cas, déduire la valeur manquante, comme le montre la figure 10.7 ;

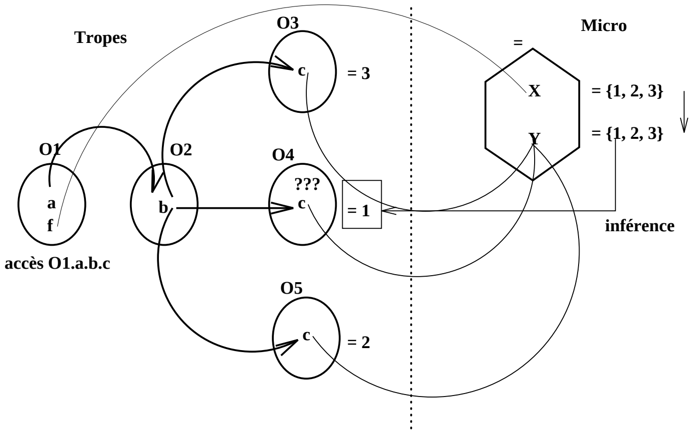


FIG. 10.7 - : La valeur de l'AIC est connue comme l'ensemble $\{1, 2, 3\}$. Les ACT liés à l'AIC sont déterminés. La valeur manquante de l'un d'eux peut, dans ce cas, être inférée.

- lorsque l'accès atteint une liste d'objets, dès que l'AIC a une valeur-liste, l'inférence est possible sur tous les ACT non encore valués, ceux-ci étant repérables par leur position dans la liste.

10.8 Contraintes et cohérence

Sans les contraintes, le maintien de la cohérence d'une base de connaissances est réparti entre les différentes fonctionnalités du modèle. Par exemple, la proposition d'une valeur est sujette à une vérification de type pour validation, la modification d'une valeur d'un attribut employé comme argument d'une méthode est propagée paresseusement le long des descendances de l'attribut qui sont maintenues par un système de maintien du raisonnement. Ces mesures garantissent que toute valeur présente dans la base est valide ou sera sujette à révision lors de la prochaine interrogation la concernant.

En présence de contraintes, si les services offerts par le modèle pour garantir la cohérence des bases de connaissances restent en place, il faut s'intéresser aux contributions du maintien de consistance opéré par MICRO sur les domaines des ACT, mais aussi sur les types des entités. Également, il faut s'intéresser à la vérification de la cohérence d'un ensemble de contraintes. Ces deux points sont débattus dans les sections suivantes. Pour terminer, nous décrivons les modes d'activation de la résolution dans TROPES, ultime étape vers la production d'instances cohérentes avec la description des entités contraintes.

10.8.1 Contraintes et consistance

Le niveau de consistance locale établi lors de la pose et de la propagation de contraintes par Micro n'offre statiquement pas toutes les garanties. En effet, en présence de CSP cycliques, ou d'opérateurs non monotones dans les contraintes, ou encore dans les CSP impliquant des variables mono ou multivaluées à domaines infinis, il peut subsister dans les domaines des valeurs qui, globalement, mais aussi localement, ne peuvent faire partie d'une instanciation satisfaisante.

Ceci est dû aux limitations de Micro qui sont les contre-parties de ses principes de fonctionnement et de l'éventail assez général de CSP qu'il est capable de traiter (prise en compte d'opérateurs monotones, CSP à intervalles, bornes d'intervalles infinies, domaines infinis ...).

Laisser des valeurs inconsistantes globalement, et même dans certains cas, localement, augmente pour Micro le temps nécessaire à l'étape de résolution pour énumérer ou diviser des domaines.

Du côté de Tropes, les insuffisances de Micro vis-à-vis du maintien de la consistance sont révélées au niveau des domaines des ACT, puisque la propagation de contraintes participe à l'élaboration du nouveau type des entités contraintes. Ceci signifie que les types des entités contraintes présentent des domaines de valeurs qui sont au mieux localement consistants, et non pas certainement globalement consistants. Exiger une telle garantie reviendrait à lancer la résolution des contraintes lorsqu'elles sont déclarées. Cette option est coûteuse et n'a pas été retenue. Le couplage Tropes/Micro opère un maintien de consistance sur les domaines des ACT selon les capacités de Micro. Aussi, pour toute entité contrainte de Tropes, son type est consistant localement dans les limites de Micro, au mieux – notamment sur les domaines finis – on a une arc-consistance.

Dans Tropes sans contrainte[3], cette considération n'intervient pas. Les domaines fournis dans les définitions des attributs offrent statiquement toutes les valeurs possibles pour ces attributs que l'on peut retrouver dans une instance : la condition tient dans le domaine lui-même. Avec les contraintes, les domaines expriment toujours cette condition mais d'autres conditions – les contraintes – viennent s'ajouter, qui font que l'ensemble des valeurs proposées par le domaine n'est pas réellement l'ensemble des valeurs possibles.

Cependant, la présence de contraintes se traduit par l'activation immédiate de celles-ci lors de toute modification dans la base les concernant. Ainsi, dynamiquement, dès qu'une valeur d'ACT est proposée, elle est propagée dans le réseau de contraintes dans lequel l'attribut est plongé. La propagation de contraintes prend le relais de la vérification de type, et sa terminaison dans un état stable du réseau de contraintes (pas de domaine vidé, pas de boucle infinie) est gage de la validité de la valeur proposée dans l'état actuel des connaissances.

Finalement, les domaines fournis par les types des attributs contraints peuvent statiquement fournir une information partiellement consistante ou inconsistante, mais la complétion d'une base de connaissances se fait sous le contrôle des contraintes présentes qui garantissent, à tout moment, que l'état actuel des connaissances n'est pas inconsistant.

Les limitations d'une simple maintenance de consistance locale sont alors évidentes : il se peut que les connaissances introduites dans la base, qu'il s'agisse de valeurs ou de contraintes, conduisent à une impasse globale. Ainsi, des concepts, des classes et des instances peuvent être contraints sans qu'en l'état actuel des connaissances, il soit possible de détecter statiquement une incohérence à terme inévitable.

Cette constatation montre que la maintenance de consistance assurée par Micro sur les domaines de Tropes n'est pas suffisante comme on s'y attendait, et que la résolution est indispensable pour tester l'existence de solutions aux CSP définis, et, par là même, la consistance de la base dans son état actuel.

Cependant, lorsque des CSP Tropes sont localement consistants mais globalement inconsistants (ils n'ont pas de solutions), ceci est dû à une incohérence dans l'ensemble des contraintes comme le montre la section suivante.

[3] Sans filtre et sans facette conditionnelle, non plus.

10.8.2 Contraintes, incohérences et redondances

La maintenance exercée par MICRO sur les CSP TROPES est insuffisante en elle-même pour détecter toutes les incohérences présentes dans les descriptions des entités (concept ou classe).

En général, une incohérence est détectée lors de la pose d'une contrainte ou de l'activation de l'une d'elles suite à une modification. L'activation d'une contrainte consistant en l'application des règles de consistance qui lui sont associées, la propagation de contrainte est déclenchée à partir de cette contrainte, si ces règles de consistance ont réduit un des domaines des attributs sur lesquels porte la contrainte.

Lorsqu'au cours de la propagation de contraintes, l'un des domaines est réduit à l'ensemble vide, c'est que l'action – l'activation de contrainte – à l'origine de la propagation de contraintes place le CSP dans un état inconsistant.

Pour TROPES, cela révèle une incohérence dans la base de connaissances. Cette incohérence a pu être introduite par l'ajout d'une contrainte ou par l'ajout ou la modification d'une valeur qui fait passer le CSP d'un état stable de consistance locale à un état instable d'inconsistance.

Or, il est des cas où l'ajout d'une contrainte transforme l'ensemble de contraintes en un ensemble inconsistant sans que cette incohérence ne puisse être détectée avec les moyens de maintenance de MICRO, qui se résument à l'application des règles de maintenance de la contrainte et, éventuellement, au déclenchement d'une propagation de contraintes.

Par exemple, considérons le système suivant formé des deux contraintes c_1 et c_2 où a et b sont tels que $dom(a) = [0, 10], dom(b) = [5, 15]$:

$$\begin{cases} c_1 : (\texttt{mic-eq}\ \mathrm{a\ b}) \\ c_2 : (\texttt{mic-neq}\ \mathrm{a\ b}) \end{cases}$$

La pose de c_1 réduit les domaines de a et b de sorte que $dom(a) = dom(b) = [5, 10]$. À la pose de c_2, les domaines demeurent inchangés. Or, manifestement aucune valeur de a et de b ne satisfera cet ensemble incohérent de contraintes.

Ce type d'incohérence n'est pas détecté par MICRO dont la maintenance, peu coûteuse, conclut à la consistance locale du CSP (ici l'arc-consistance) et montre encore une fois ses limites. Si bien qu'une base de connaissances TROPES peut être tacitement et statiquement incohérente. Dans l'exemple, l'incohérence sera signalée de manière dynamique lors de l'instanciation de l'un des deux ACT.

La leçon de cet exemple est qu'il est possible que l'ensemble des contraintes posées sur une entité soit incohérent. Pour combler ses lacunes, MICRO propose la résolution comme moyen de détecter l'incohérence. Cependant, ce recours peut être coûteux et ne résout pas le problème, simplement il conclut à la non existence d'une solution.

En réalité, il faut noter que ces incohérences relèvent bien d'un problème de modélisation – typiquement, un concept ou une classe est décrit par des contraintes incompatibles. Alors que, d'un côté les contraintes offrent un moyen d'étendre l'expressivité du modèle de connaissances et par là, le maintien de la cohérence au sein d'une base de connaissances – ce qui n'était absolument pas possible avant –, de l'autre, leur gestion ne suffit pas toujours à garantir cette cohérence et laisse la porte ouverte à des incohérences que MICRO peut ne pas détecter lorsqu'il ne dispose que des types des attributs. La puissance d'expression des contraintes n'est donc pas entièrement contrôlable par MICRO.

Dans la version actuelle du couplage TROPES/MICRO, nous avertissons préalablement l'utilisateur de l'occurrence éventuelle de ce genre de problèmes, à défaut de pouvoir les détecter et les résoudre. À l'avenir, il existe des solutions à apporter pour détecter ces incohérences sans même avoir à lancer la résolution sur le CSP pour convenir de son inconsistance.

Ces solutions, comme l'a notamment montré James Gosling à travers son système MAGRITTE [Gosling83], passent, par exemple, par le couplage de MICRO avec un système de réécriture des ensembles de contraintes pour se rendre compte, à partir de règles, des simplifications et des incompatibilités (comme ici l'incompatibilité des contraintes `mic-eq` et `mic-neq`) de contraintes portant sur les mêmes VCM (donc, ACT).

Un autre handicap de MICRO, dans sa version actuelle, est son inaptitude à déceler les contraintes redondantes et à opérer sur elles les simplifications qui s'imposent.

Par exemple, considérons le système suivant formé des deux contraintes c_1 et c_2 où les ATC a et b sont tels que $dom(a) = [0, 10], dom(b) = [5, 15]$:

$$\begin{cases} c_1 : (\texttt{mic-ge}\ \mathrm{a}\ \mathrm{b}) \\ c_2 : (\texttt{mic-gt}\ \mathrm{a}\ \mathrm{b}) \end{cases}$$

La pose de c_1 réduit les domaines de a et b de sorte que $dom(a) = dom(b) = [510]$. À la pose de c_2, les domaines sont changés en $dom(a) = [6, 10]$ et $dom(b) = [5, 9]$. La propagation de contraintes a fait son travail et, de ce point de vue, on ne peut rien reprocher à MICRO. Le seul reproche que l'on peut lui faire c'est de ne s'être pas rendu compte que les deux contraintes c_1 et c_2 sont redondantes et en relation de *subsomption*. c_1 est plus générale que c_2 au sens où le prédicat associé à c_2 implique (au sens de l'implication de la logique des prédicats) le prédicat associé à c_1 ou encore que l'extension (ou ensemble des solutions localement consistantes) de c_2 est incluse dans l'extension (ou ensemble des solutions localement consistantes) de c_1. C'est en ces termes que l'on peut définir la redondance entre deux contraintes portant sur le même ensemble de variables. Là encore, d'après cette définition, il serait possible de solliciter la résolution pour détecter ces redondances.

Dans TROPES, si la redondance entre certaines contraintes n'est pas source d'incohérence, elle se traduit malgré tout par une surcharge en termes de contraintes et donc de temps de réponse lors de la consultation ou de la modification d'une base de connaissances. Les contraintes redondantes n'étant pas simplifiées, elles sont considérées chacune comme des contraintes sans relation spécifique avec d'autres contraintes. La redondance et l'inefficacité du couplage se ressentent donc, soit lors de la pose, du maintien de la consistance, de la propagation et de la résolution, soit dans chacune des opérations concernant deux contraintes redondantes. Ceci s'explique par le fait que deux contraintes redondantes sont impliquées dans les mêmes propagations et processus de résolution.

Il serait souhaitable de détecter également les redondances afin de les éliminer et d'empêcher ainsi l'héritage inutile de contraintes de classes. Le fait qu'une contrainte de concept soit considérée comme une contrainte de certaines classes et comme une contrainte de toutes les instances du concept, qu'une contrainte de classe soit considérée comme contrainte de toutes les instances de la classe donne une estimation du temps de calcul consacré à des poses de contraintes inutiles.

Une solution envisageable à terme serait d'employer des méthodes de simplifications de contraintes redondantes comme celles proposées dans [Lassez et al.93]. Ces règles de simplifications sont également à la base du fonctionnement des systèmes de réécriture. Aussi, le couplage ou l'intégration de MICRO avec un tel système permettrait de simplifier les systèmes de contraintes, voire même de détecter des incohérences. Cette solution peut toutefois s'avérer coûteuse (traduction des contraintes, application des règles systématiques...) et dépend toujours de la puissance et de la complétude des règles de simplification.

Une autre solution, toujours partielle mais plus déclarative et moins coûteuse, a été proposée dans [Capponi et al.95]. Elle consiste à déclarer au niveau de TROPES les contraintes qui sont incompatibles entre elles (qui ne peuvent porter sur les mêmes attributs) et les contraintes qui sont redondantes entre elles (portant sur les mêmes attributs mais dont les extensions sont liés par des inclusions). Lorsqu'une contrainte est définie, cette information, gérée par l'interface TROPES/MICRO est exploitée pour valider sa pose. Si une contrainte est incompatible ou redondante (ou plus faible) avec l'une des contraintes déjà présentes, sa pose est refusée.

En l'état actuel de MICRO, la détection ultime des incohérences et des redondances repose donc sur la résolution des systèmes de contraintes. La prochaine section décrit à partir de quelles entités ce processus peut être lancé dans TROPES.

10.8.3 Contraintes et résolution

Dans TROPES, les contraintes peuvent être considérées selon le niveau de représentation (concept, classe, instance) auquel elles sont déclarées. Aussi, on peut imaginer que la résolution s'applique sur un réseau de contraintes de concept, un réseau de contraintes de classe ou bien un réseau de contraintes d'instances.

Cependant, au niveau des instances, cette distinction entre les contraintes ne se fait plus. Ceci parce que définir une contrainte à un certain niveau de représentation n'est qu'une façon de décrire l'ensemble des instances sur laquelle porte la contrainte. Aussi, au niveau d'une instance, toutes les contraintes sont présentes et sont considérées sans distinction de niveau de déclaration. Ainsi, une contrainte de concept peut être associée, dans le même réseau de contraintes, avec d'autres contraintes de concept mais aussi des contraintes de classes et des contraintes d'instances. Pour une instance, s'il existe plusieurs CSP indépendants, c'est bien qu'ils portent sur des ensembles distincts d'attributs et non pas qu'ils sont composés de contraintes de niveau de représentation distincts. De même au niveau des classes, les CSP sont constitués des contraintes de la classe et des contraintes du concept.

Cette constatation nous a conduit à proposer que la résolution se fasse sur un ensemble d'attributs d'une instance, sur une instance, sur un ensemble d'instances d'une classe (énuméré ou sélectionnable par un filtre), sur l'extension entière d'une classe, sur un ensemble d'extensions de classes, sur un ensemble d'instances d'un concept ou sur toutes les instances d'un concept, dans l'ordre croissant du coût de résolution.

La résolution sur un ensemble d'attributs consiste à résoudre le ou les CSP dans lesquels sont impliqués tous les attributs de l'ensemble. Implicitement, il est possible que d'autres attributs soient impliqués, notamment si un des attributs qui doivent figurer dans la solution est la source d'un accès non-immédiat. L'ensemble d'attributs fourni est donc minimal et indicatif, il détermine le nombre minimal de n-uplets d'une solution. La résolution sur une instance, et, à partir de là, sur les diverses tailles d'ensembles d'instance proposées, est une généralisation de ce principe.

Le résultat de la résolution est un, plusieurs ou aucun (pas de solution) n-uplet solution dont chaque élément est un triplet $(instance, attribut, domaine)$.

Ces n-uplets solutions sont donnés comme information des évolutions ou complétions possibles et cohérentes de la base à partir de son état actuel. L'utilisation de ces résultats est laissée au libre-arbitre de l'utilisateur ayant sollicité cette résolution.

10.9 Conclusion

Ce chapitre s'est intéressé aux conséquences de la présence de contraintes dans le modèle TROPES.

D'un côté, le système repose sur un certain nombre de principes de représentation (approche classe/instance, description hiérarchisée de la connaissance, observation selon divers points de vue, définition de passerelles...). Afin d'exploiter la connaissance décrite, TROPES propose divers mécanismes tels que l'héritage, l'instanciation, le détachement procédural et la classification.

De l'autre côté, les contraintes se présentent à la fois comme un moyen déclaratif d'exprimer des relations entre des attributs d'objets et comme un moyen d'inférence de valeur de ces attributs.

L'intégration de contraintes dans TROPES ne doit pas remettre en cause la sémantique attachée à chaque principe de représentation ou d'inférence du modèle. Aussi, nous avons décrit ici comment la présence de contraintes devait être interprétée par le modèle lors de l'instanciation et de la tentative de rattachement d'une instance à une classe. La déclaration de contraintes au niveau des classes doit, elle aussi, bénéficier du mécanisme d'héritage comme d'un moyen de factoriser des connaissances communes entre classes et sous-classes. Vis-à-vis des passerelles, les contraintes ne doivent pas aller à l'encontre mais renforcer les relations ensemblistes déduites sur les extensions des classes jointes. Lors de la classification, le pouvoir inférant des contraintes doit être inhibé, elles doivent se contenter d'avoir un rôle prédicatif, la propagation de contraintes pouvant toutefois être sollicitée. La présence simultanée d'une méthode et d'une contrainte sur un attribut peut être source d'incohérence. Mais chacun de ces mécanismes a une tâche bien définie : à la méthode, il revient de fournir la valeur lorsque celle-ci n'est pas disponible, à la contrainte, il revient de veiller à la maintenance d'une relation entre cet attribut et d'autres.

L'ajout et le retrait d'une contrainte a été étudié selon le niveau de représentation auquel est attachée la contrainte. Afin que soient désignés les attributs à contraindre, la notion d'accès a été introduite en même temps que les contraintes dans TROPES. Nous avons donc également étudié la gestion des accès dans les diverses étapes du cycle de vie d'une contrainte.

Cette gestion des accès proposée ici est une contribution par rapport aux travaux sur la notion de *chemin* des systèmes THINGLAB [Borning81] et CSPOO [Kökény94]. Nous avons défini un comportement de l'accès lors du calcul de type. Nous avons proposé une gestion de l'absence et de la modification de la valeur d'un attribut utilisé dans un accès. La gestion proposée ici est différente de celle de CSPOO (*cf.* section 5.1.5) : l'information n'est pas maintenue par des variables flottantes et des contraintes d'interface, mais est stockée au niveau des attributs et propagée le long de l'accès. De plus – et c'est là une contribution innovante –, les attributs multivalués sont pris en charge dans la gestion des accès.

Enfin, un constat quelque peu décevant mais prévisible a été dévoilé quant à la cohérence d'une base de connaissances en présence de contraintes. Le couplage TROPES/MICRO ne pouvant garantir qu'une consistance locale des domaines d'ACT, la définition (types) des attributs est bien souvent une description trop générale, qui contient bien toutes les solutions possibles, mais peut contenir également des valeurs, qui globalement n'apparaîtront jamais dans une instance (solution). Si statiquement la présence de contraintes peut donc introduire une certaine inconsistance dans les définitions des objets, dynamiquement, toute tentative d'instanciation d'un attribut contraint avec une valeur ne pouvant figurer dans une solution du CSP de l'attribut, sera refusée. Aussi, l'introduction de contraintes ne garantit plus la consistance globale des faits descriptifs d'une base de connaissance. Par contre, la cohérence des faits effectifs (les valeurs) contenus dans une base de connaissances est toujours préservée. C'est le rôle de la propagation par contraintes assurée par le module de programmation par contraintes MICRO.

Une autre limite dans la gestion de CSP TROPES est l'incapacité à déceler les redondances, et pire, certaines incohérences dans un ensemble de contraintes sans avoir recours à la résolution. Les premières sont synonymes de perte de temps, les secondes d'absence de solutions mais surtout d'un problème de modélisation. Des solutions sont envisageables pour prévenir ce genre de désagréments. La moins coûteuse serait d'avoir recours à TROPES pour exprimer les incompatibilités et les redondances prévisibles.

Le premier objectif du couplage de TROPES avec un module de programmation par contraintes tel que MICRO est la mise à la disposition des utilisateurs du modèle d'un moyen déclaratif d'expression de relations entre objets possédant, le cas échéant, une certaine capacité d'inférence. Mais la présence de contraintes s'est montrée également exploitable par le modèle lui-même, pour gérer l'expression de propriétés entre objets impliqués dans une relation de composition, ou bien encore dans l'expression du flots de données entre tâches, mais aussi pour entamer l'exploration de la base sous raisonnement hypothétique. Ces trois aspects sont détaillés dans le chapitre suivant.

Troisième partie

Extensions du modèle Tropes par les contraintes

Chapitre 11

Exploitation des contraintes par Tropes

Les chapitres précédents se sont attachés à décrire comment un utilisateur de TROPES peut définir des contraintes sur les attributs des instances d'un concept, des instances d'une classe, ou d'instances particulières. Les CSP sont maintenus et résolus par MICRO, un module de programmation par contraintes qui décharge l'utilisateur des problèmes de maintenance. Si bien que pour un utilisateur, les contraintes apparaissent comme un moyen d'énoncer de manière déclarative des relations entre attributs d'objets maintenues automatiquement par TROPES lui-même (par l'intermédiaire de MICRO).

Si c'est pour ce premier usage qu'a été conçu MICRO, la disposition d'un système de gestion automatique de contraintes s'est révélée être également un atout précieux dans la description de la sémantique de certaines fonctionnalités de représentation ou opérations proposées comme extensions[1] du modèle TROPES, à savoir :

- la gestion du partage de propriétés entre les objets composites et leurs objets composants (section 11.1) ;
- la gestion du flot de données dont a besoin le modèle de tâches pour assurer l'activation des objets tâches (section 11.2) ;
- l'exploration des connaissances en mode de raisonnement hypothétique (section 11.3).

Ces trois illustrations des bénéfices que peut retirer TROPES lui-même (ses développeurs) des contraintes afin de contrôler l'implantation de mécanismes nouveaux sont décrites dans les sections suivantes.

11.1 Contraintes et objets composites

11.1.1 Objets composites et partage de propriétés

Dans le modèle TROPES, une relation peut être modélisée par un attribut de nature *relation*. La description de la signature de la relation ne peut se faire alors que par les facettes de l'attribut lien en question. Toutefois, la relation de composition jouit d'un traitement particulier dans TROPES (*cf.* section 3.1.10) et il lui a été attaché un comportement des objets précis qui sont contrôlés par le modèle (c'est-à-dire codés "en dur"). Un concept dont l'un des attributs est de nature *composant* est alors un concept composite. Parmi les six interprétations de la composition proposées par Winston *et al.* [Winston et al.87] (*cf.* section 2.1.1.6), TROPES adopte la relation *composant/tout* qui a les propriétés de fonctionnalité et de séparabilité, et qui est la plus souvent rencontrée en pratique. Elle autorise la décomposition transitive (mais non réflexive) des objets : un composite

[1] Ces propositions sont restées au stade conceptuel.

peut être composé de composites eux-mêmes composites, etc. Les composants d'un objet composite O' composant d'un objet composite O sont composants de O.

Bien que la relation de composition puisse être considérée comme orthogonale à la relation de spécialisation, les attributs *composants* se comportent comme tout autre attribut vis-à-vis de l'héritage dans TROPES.

Kim *et al* [Kim et al.89] ont proposé quatre types de comportement, selon la dépendance composite/composant et le partage composant/composite. Dans TROPES, la dépendance d'un objet composant vis-à-vis de son objet composite est exprimée comme suit : à la destruction de l'objet composite, la fermeture transitive des composants qui ont été créés par lui est détruite. Concernant le partage, un composant créé par et pour son composite ne peut être partagé par plusieurs composites, simplement parce que l'identification de ce composant doit être gérée par le système. Autrement dit, un composant peut avoir une existence antérieure et postérieure à son composite, ou être partagé par plusieurs composites s'il a été défini par l'utilisateur, et non pas par le système (comme c'est le cas lors de la création d'instance de composite opérée pendant la classification d'un objet composite).

La spécificité de la relation de composition fait qu'il est intéressant de faire partager des propriétés entre un composite et ses composants, ou bien encore de lier certaines des caractéristiques qu'ils partagent.

Le partage de propriétés entre un objet composite et ses composants dans TROPES (*cf.* figure 11.1) a été proposé par Olivier Schmeltzer et vise à adjoindre à la relation de composition des mécanismes d'inférence de propriétés d'attributs. Cette notion est ici reprise et étendue. Initialement présenté comme un mécanisme par défaut, le partage de propriétés est pris en charge par les contraintes, ce qui inhibe la sémantique initiale puisque le partage de propriétés est maintenant immédiatement pris en compte par les démons de déclenchement de la propagation de contraintes [Gensel et al.94]. L'expression de contraintes a un autre avantage : le partage n'est plus exprimé lourdement à l'aide de facettes spécifiques, comme cela était le cas à l'origine, mais uniformisé et inséré dans la liste des contraintes du concept ou de la classe du composite ou du composant, selon le sens de ce partage.

De même, cette étude du partage de propriétés, ou bien encore l'expression de relations entre les caractéristiques d'objets composites, peut être généralisée à tous les objets complexes – et non plus seulement composites – qu'ils soient liés par une relation[2] à travers un attribut de nature *relation* ou *propriété*.

Dans le cadre des objets composites, le partage de propriété est dénommé *co-héritage*. Il s'agit d'exprimer le moyen par lequel un attribut d'objet composite hérite de la valeur d'une propriété (d'un attribut) de ses composants ou bien qu'un attribut d'objet composant hérite de la valeur d'un attribut (d'une propriété) de son objet composite. Ce partage de propriétés devient plus complexe quand on veut faire des opérations sur les valeurs des attributs des composants afin d'en déduire la valeur d'un attribut du composite (par exemple, exprimer que le prix d'une chaîne stéréo est la somme des prix de ses éléments). Aussi, c'est notamment dans la perspective de l'expression d'un partage de propriétés ou de relations complexes entre les caractéristiques de divers objets qu'ont été construites les fonctions contraintes de MICRO destinées aux attributs multivalués.

Dans le sens composite vers composant, la propagation d'une valeur d'attribut ne pose pas de problème de compatibilité, dans l'hypothèse où un composant n'est composant que d'un seul composite, puisque une seule valeur est propagée (celle de la caractéristique du composite dont le composant hérite). Dans le cas où cette hypothèse ne prévaudrait pas, il s'agirait de lever les conflits de noms dans le co-héritage par le composant d'une caractéristique possédée par plusieurs de ces composites. Là, l'entité accès de TROPES permettrait d'ôter toute ambiguïté.

Dans le sens composant vers composite, il doit être possible de spécifier de quel(s) composant(s) l'attribut du composite co-hérite sa valeur. Là encore, les accès semblent tout indiqués. De même,

[2] Le chapitre 12 s'intéresse au partage de valeur entre objets impliqués dans une relation.

il doit être possible d'exprimer ces inférences à l'aide d'opérateurs plus complexes que la simple identité (dans le cas d'un partage usuel) comme la somme, le minimum, la moyenne... – ces opérateurs agissant alors sur l'ensemble ou la liste des composants du composite. Les opérateurs fournis par MICRO vont permettre de construire ces expressions de partage, là où des opérateurs prédéfinis et spécifiques à cet usage avaient été envisagés initialement, qui alourdissaient considérablement l'expression désirée.

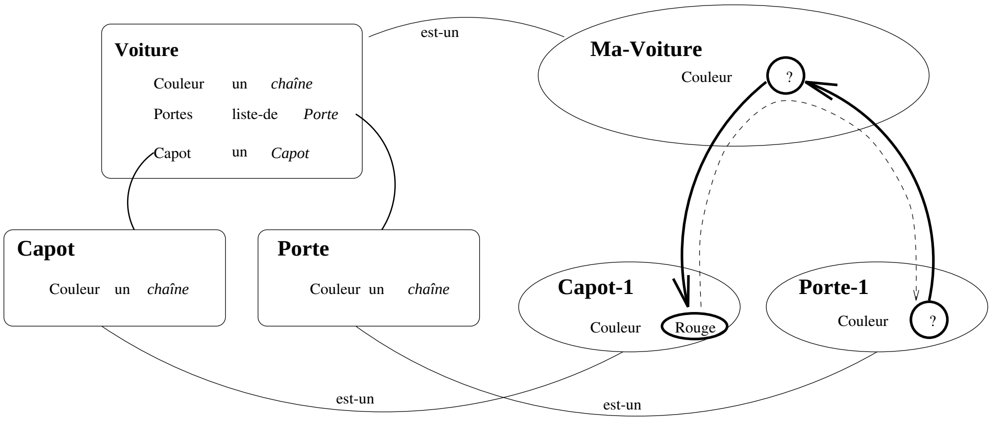


FIG. 11.1 - : (d'après [Gensel et al.94]). On souhaite exprimer que la couleur d'une voiture peut être héritée des composants (ici réduits au capot et aux portes) alors que la couleur de chacune des portes est héritée de la couleur de la voiture. Ainsi, dans la configuration présente, la couleur inconnue de la porte *Porte-1* peut être héritée de *Ma-voiture*. Or, la couleur de *Ma-voiture* n'est pas connue mais est héritable de son composant capot *Mon-capot*.

11.1.2 Expression du partage de propriétés par les contraintes

La panoplie des contraintes fournies par MICRO, mais aussi la notion d'accès, permettent d'exprimer ce partage de propriétés.

Tout d'abord, le partage de propriétés doit être spécifié au niveau du concept (ou de la classe) de l'objet composite. Ceci s'explique aisément par le fait, d'une part, que l'information concernant le composite – même si le partage a lieu du composite vers le(s) composant(s) – est propriété de celui-ci et, d'autre part, qu'un composant recevant une valeur ne la reçoit *seulement* que dans le contexte de cette relation de composition, c'est-à-dire à travers l'existence d'un objet composite le référençant en tant que composant.

Ensuite, le sens du partage peut être facilement indiqué à l'aide des contraintes d'égalité `mic-eq` – le partage s'effectue dans les deux sens – ou d'affectation `mic-assign` – le partage suit le sens de l'affectation.

Enfin, pour les expressions complexes d'obtention de la valeur d'un attribut du composite à partir des valeurs de l'attribut dans les composants, les fonctions contraintes particulières telles que `mic-apply`, `mic-all`, `mic-min`... ainsi que les contraintes conditionnelles, trouvent là un emploi privilégié pour appliquer un opérateur sur un ensemble ou une liste de composants. Il est à noter que les contraintes permettent, de surcroît, via les accès, de spécifier un partage de propriétés entre des composants via le composite. Toutefois, les contraintes exprimant le sens et la sémantique du partage de propriétés entre composants sont exprimées à nouveau au niveau du composite. Ici, on souhaite indiquer que ce partage de propriétés a lieu seulement dans le cadre d'une relation de composition qui unit deux objets composants. Hors de ce contexte, ces objets ne seraient peut être jamais liés ou ne partageraient peut-être pas de la même façon une caractéristique commune.

Initialement, le partage de propriétés était décrit à l'aide de deux facettes ajoutées à l'ensemble des facettes de description d'un attribut.

- La facette *co-héritage* prenait la valeur *composant* pour indiquer dans un composite que la valeur était héritée d'un des composants ; la valeur *composite* pour indiquer que la valeur

```
Voiture
        couleur     un        chaîne

        portes      liste-de       Porte
                    card 4

        capot       un        Capot

 contrainte={(mic-assign couleur capot.couleur)
              (mic-if  (mic-all-eq portes.couleur)
                       (mic-assign couleur
                           (mic-car portes).couleur
                       )
              )

              (mic-map-c #'mic-assign
                              portes.couleur
                              couleur
              )

             }
```

la voiture herite de la couleur de ses composants

la porte hérite de la couleur de la voiture

FIG. 11.2 - : On déclare dans la classe (ou concept) voiture les contraintes qui permettent le partage de valeur pour l'attribut *couleur*. La voiture hérite la couleur du capot et la couleur des portes, si elles ont toutes la même couleur ; chaque porte hérite la couleur de la voiture, si elle n'a pas de couleur. On notera l'utilisation d'un résultat d'une fonction contrainte (ici `mic-car`) pour déterminer l'objet source d'un accès.

qu'un composite héritait la valeur du composant (sous réserve que ce composant soit unique, sans quoi un risque de conflit peut se produire et aucun moyen de préciser le composant n'est fourni) ; la valeur *indifférent* pour indiquer que l'héritage se faisait dans les deux sens.

- La facette *co-opérateur* spécifiait l'opérateur à appliquer pour déterminer la valeur. Un ensemble prédéfini de propriétés lui était associé (égalité, moyenne, somme, min, max...)

Cette solution n'est pas satisfaisante car elle implique de reporter, dans chaque concept concerné, la facette *co-héritage* selon le sens voulu. De fait, cette solution supporte mal, par exemple, l'utilisation d'un concept comme composant de plusieurs composites, pour peu que le même attribut ne soit pas l'objet du même partage dans deux concepts composites différents. De plus, l'ensemble des valeurs de la facette *co-opérateur* ne recouvre pas l'ensemble des possibilités de calcul de la valeur d'un attribut à partir d'un ensemble ou d'une liste de propriétés d'autres attributs. Aussi, si l'intention première de l'utilisation de ces facettes était d'offrir à l'utilisateur une expression plus agréable et plus courte du partage de propriétés, elle souffre d'une certaine lourdeur et d'une incomplétude dans les traitements. La spécification intégrale du partage de propriétés par des contraintes oblige l'utilisateur à rassembler les contraintes dans l'objet composite, et à définir avec précision, à travers l'expression des contraintes, le sens et la valeur transmise dans le partage de propriétés.

Enfin, quant à la motivation originale de considérer ce partage de propriétés agissant comme un mécanisme de défaut, il faut signaler que les contraintes conditionnelles de MICRO permettent de n'effectuer le partage (par le biais de contraintes) que lorsque certaines conditions sont réunies (*cf.* figure 11.2).

11.2 Contraintes et tâches

11.2.1 Modèle de tâches et flot de données

Un modèle de tâches a été conçu pour TROPES [Gensel et al.92] dont le principe est de considérer les tâches comme des entités (objets) du modèle à part entière (il existe un concept général et prédéfini de tâches), afin de décrire la stratégie de résolution d'un problème qu'est censée représenter la tâche. Une tâche est alors exécutée par la construction d'une instance de ce concept. L'instance créée est incomplète et ne comporte que certaines entrées de la tâche. Il reste à déterminer la valeur du traitement, attribut qui décrit la décomposition de la tâche en sous-tâches et/ou procédures (c'est-à-dire tâches élémentaires non décomposables). Cette complétion se fait par une classification de l'instance de tâche dans une hiérarchie afin d'identifier la classe qui lui correspond. Lorsque l'instance de tâche est suffisamment complète – la classe déterminée par la classification représente une tâche concrète – on peut lancer l'exécution des éventuelles sous-tâches ou procédures dont elle se compose. La tâche est donc un objet composite dont la classification reprend les principes énoncés dans [Mariño91].

Le flot de données ou passage de paramètres (*cf.* figure 11.3) décrit comment une sortie de tâche est l'entrée d'une ou de plusieurs de ses sous-tâches. Les entrées et les sorties des tâches étant des attributs d'instances de tâches, l'expression du flot de données peut se faire à l'aide de contraintes.

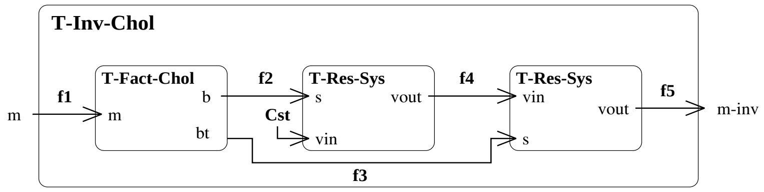


FIG. 11.3 - : (d'après [Gensel et al.94]). Description du flot de données dans la décomposition de la tâche *T-Inv-Chol* en une séquence de trois sous-tâches *T-Fact-Chol, T-Res-Sys, T-Res-Sys. f1, f2, f3, f4, f5* sont les liaisons entre les entrées et sorties de ces tâches.

11.2.2 Expression du flot de données par des contraintes

La description du flot de données consiste à décrire dans la classe de la tâche composite chacune des liaisons entre ses attributs *sorties* et les attributs *entrées* de ses sous-tâches qui sont fournies par l'éclatement (ou description détaillée) de son attribut *traitement* de nature *composant*. Il s'agit de l'opération la plus délicate à exprimer et à maintenir. Dans TROPES, on confie la gestion du flot de données aux contraintes [Gensel et al.94]. Chacune des liaisons entre attributs de tâches est exprimée par une contrainte d'affectation entre un attribut *entrée* d'une sous-tâche et un attribut *sortie* d'une tâche. Cette contrainte vient s'inscrire dans l'ensemble des contraintes de classe de la tâche et assure automatiquement, lors de l'exécution, le passage de la valeur d'un paramètre à l'autre.

Afin de rester proche du formalisme des tâches et de permettre au concepteur d'exprimer le flot de données en termes de liaisons entre attributs, une nouvelle facette appelée *flot* est introduite. Cette facette, qui a pour valeur une chaîne de caractères, agit comme une ancre et permet de lier deux attributs de deux tâches différentes entre eux. Elle est traduite par le système en contrainte d'égalité. Tous les attributs qui ont le même nom de liaison sont contraints à être identiques (*cf.* figure 11.4). La description des flots de données se fait au niveau des tâches concrètes non terminales. C'est dans ces classes que sont ajoutées à la liste des contraintes de classe, les contraintes qui assurent le passage de paramètres.

Ces contraintes sont ici des contraintes d'affectation. Mais, dans la perspective d'une utilisation du modèle de tâches pour la reconnaissance de plans ou la reconnaissance d'intentions, des contraintes d'égalité seraient plus appropriées et permettraient de reconstituer une tâche à partir de ses sous-tâches.

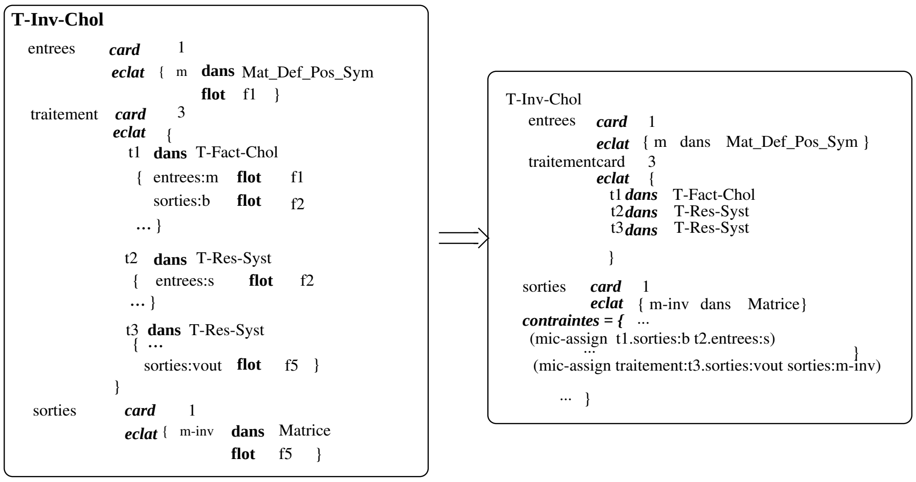


FIG. 11.4 - : Description du passage de paramètres par la facette *flot* et son équivalent en termes de contrainte. Par exemple, le flot *f1* est réécrit en la contrainte (`mic-assign` entrées:m, traitement:t1.entrées:M) où ':' signifie l'accès à un sous-attribut d'un attribut éclaté.

11.2.3 L'exécution de la tâche

L'exécution d'une tâche (*cf.* figure 11.5) correspond à la construction complète d'une instance du concept *Tâche*. C'est-à-dire que les quatre attributs *but*, *entrées*, *traitement* et *sorties* doivent être connus. Dans le cas contraire, l'instance restant incomplète, l'exécution échoue. Le processus d'exécution se décompose en trois étapes majeures: la demande d'exécution correspond à l'instanciation d'une classe du concept *Tâche*, le but et les entrées étant fournis; si la classe ne caractérise pas une tâche concrète – les attributs *traitement* et *sorties* ne sont pas spécifiés – un processus de classification effectue une descente dans la hiérarchie de tâches en fonction des entrées et du but. Une fois la classification achevée, si une classe correspondant à une tâche concrète a été trouvée, soit l'instanciation en séquence de chaque sous-tâche décrite dans l'attribut traitement est lancée et les contraintes définissant le flot de données sont activées (cas d'une tâche non-terminale), soit l'exécution de la procédure externe attachée à la classe est lancée (cas d'une tâche terminale).

11.3 Contraintes, évaluation et raisonnement hypothétique

Parallèlement à ces travaux sur l'intégration de contraintes au modèle TROPES, une étude est menée par Pierre Girard [Girard95] sur la construction d'instance hypothétique par exploration de la base de connaissances. Cette étude définit un contrôle d'évaluation des attributs dans lequel les contraintes apparaissent comme un moyen d'inférence.

11.3.1 Contrôle d'évaluation

Il est considéré ici que plusieurs moyens d'inférence – encore appelés connaissances de production – (utilisateur, détachement procédural, contraintes, valeur par défaut) sont disponibles dans TROPES et, en conséquence, pourraient se trouver en concurrence pour l'évaluation de la valeur d'un attribut. Ce risque est envisagé dans un processus de raffinement d'instance où l'on cherche à compléter une instance en la faisant descendre classe par classe dans la hiérarchie. Afin de ne pas compromettre les possibilités de descente de l'instance vers certaines classes par l'attribution hâtive d'une valeur à un attribut, il convient de déterminer un ordre de priorité sur les moyens d'inférences qui peuvent être employés à un niveau donné de la hiérarchie. Conceptuellement, l'idée est d'exprimer explicitement la stratégie d'évaluation à suivre pour un attribut dans une classe. Cet ordre de priorité est défini au niveau du concept. Il est établi entre les moyens d'inférence par des

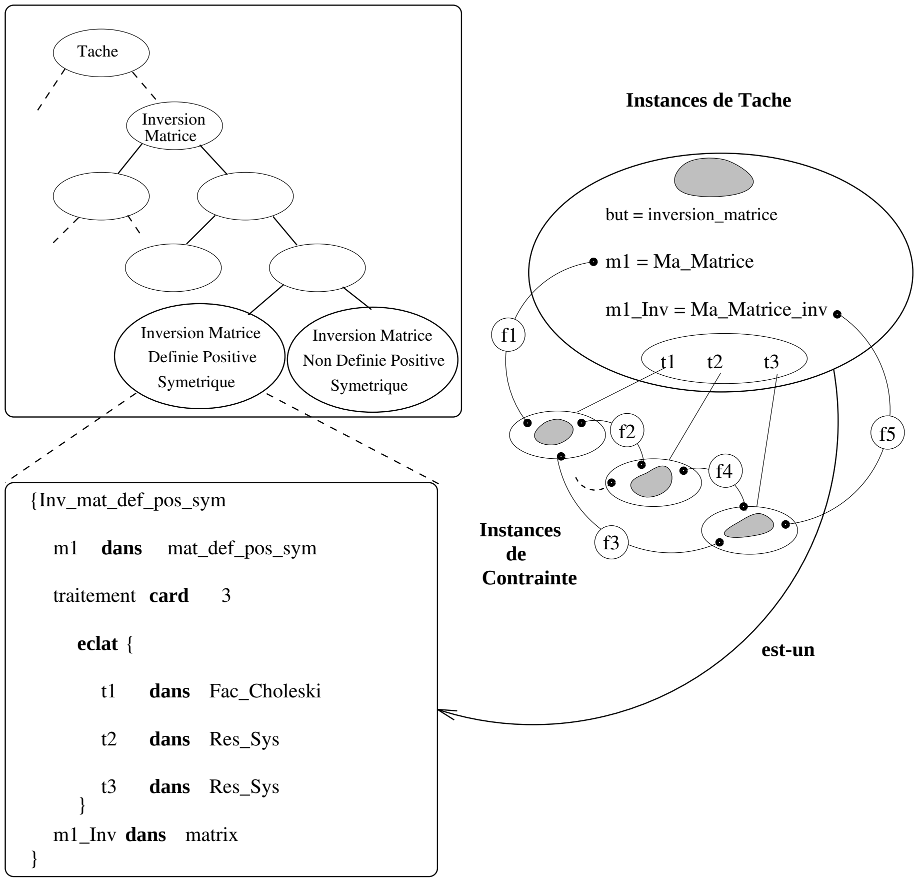


FIG. 11.5 - : L'exécution d'une tâche dont le but est d'inverser une matrice. Initialement, la classe *Inversion de Matrice* est instanciée, puis, suite au processus de classification, l'instance de tâche est complétée suivant le moule de la classe *Inversion de matrice symétrique définie positive*. Le rattachement de l'instance à la classe *Inversion de matrice symétrique définie positive* provoque la création de trois instances des tâches apparaissant dans l'attribut *traitement*, ainsi que l'activation des contraintes entre les attributs de ces instances.

règles de précédence. Ces règles expriment que, au niveau d'une classe, une source de connaissance disponible (détachement procédural, réduction de domaine ou défaut) est prioritaire devant une autre source de connaissance disponible dans une sous-classe ou/et une sur-classe ou/et la même classe.

Le contrôle d'évaluation a pour but d'assurer le développement de la stratégie d'évaluation d'un attribut au rythme du raffinement de l'instance dans la hiérarchie de spécialisation. À chaque attribut d'une instance en cours de raffinement (complétion et descente dans la hiérarchie de classes) est associé un état (inconnu, bloqué, demandeur, indécis ou évalué) qui caractérise l'étape d'évaluation dans lequel il se trouve. Le contrôle d'évaluation d'un attribut doit sélectionner le moyen d'inférence le plus prioritaire, gérer les échecs des tentatives d'évaluation ainsi que les éventuelles phases d'attente et de reprise dans l'évaluation.

La part la plus importante du travail effectué par le contrôle d'évaluation réside dans la sélection du moyen d'inférence à activer mais également dans la gestion des reprises et des attentes dans l'évaluation d'un attribut. Celle-ci repose sur la gestion d'une partition des moyens d'inférences selon que leur activation est possible, impossible ou sûre. La sélection finale s'effectue sur les moyens d'inférence en fonction des règles de précédence évoquées plus haut.

Lorsqu'elles sont prioritaires, les inférences dues à des réductions de domaines reposent sur les résultats fournis par la propagation de contraintes gérée par MICRO. Les AIC correspondant à l'ACT en cours d'évaluation sont consultés par le contrôle d'évaluation pour savoir si la propagation a réduit le domaine à un singleton. Dans ce cas, l'inférence a réussi. Il se peut également que l'un des ACT soit en situation d'attente d'évaluation, auquel cas, l'inférence par réduction de domaine est également considérée en situation d'attente. Si ces conditions ne sont pas réunies, le contrôle d'évaluation conclut à une situation d'échec.

11.3.2 Raisonnement hypothétique

La mise en place d'un raisonnement hypothétique dans TROPES consiste à gérer des versions hypothétiques d'une instance. L'idée est la suivante : l'utilisateur fournit à un système d'assistance à base de raisonnement hypothétique une *contrainte de productivité* concernant une instance. Cette contrainte de productivité est un ensemble de *buts* à atteindre par le système d'assistance au raisonnement hypothétique. Chaque but consiste à évaluer un attribut de l'instance non encore évalué. Le système d'assistance au raisonnement hypothétique est en mesure de fournir une solution à la contrainte de productivité lorsque les attributs dont l'évaluation constituait un but ont une valeur et que l'instance est cohérente vis-à-vis des diverses contraintes données par les descriptions du concept, de ses classes de rattachement ou encore des contraintes d'instance qui lui sont relatives.

Le système d'assistance au raisonnement hypothétique produit deux types d'hypothèses : celles issues d'un choix du moyen d'inférence à employer pour atteindre un but, et celles issues d'un raffinement de l'instance.

Le système d'assistance au raisonnement hypothétique consulte le contrôle d'évaluation pour connaître les moyens d'obtention de la valeur des attributs désignés comme buts. À partir des différentes possibilités, le système d'assistance au raisonnement hypothétique propose au système de raisonnement des versions hypothétiques de l'instance.

Le système d'assistance au raisonnement hypothétique élabore ainsi une hiérarchie de versions hypothétiques de l'instance pour chaque position particulière de l'instance dans la hiérarchie de classes. Dans cette hiérarchie de versions, on trouve au même niveau des versions de l'instance qui sont distinguées par l'hypothèse qui a conduit à leur création (choix d'un mécanisme d'inférence) (*cf.* figure 11.6). Lorsqu'à un niveau dans la hiérarchie de classes, l'évaluation des attributs de la contrainte de productivité n'est pas possible (les attributs sont bloqués), le système d'assistance au raisonnement hypothétique tente une descente (un raffinement) de l'instance. Une version à un niveau i de la hiérarchie de classe devient alors configuration initiale (racine) d'une hiérarchie de versions associée à un niveau j inférieur (*cf.* figure 11.7).

À un niveau inférieur, on trouve une version plus complète. Les feuilles de cette hiérarchie correspondent, soit à des versions solutions (on est au niveau n de la hiérarchie, il y avait n attributs buts à atteindre), soit à des versions incohérentes (un des buts ne peut être atteint), soit à des versions complètes non satisfaisantes (au niveau n, elles satisfont la contrainte de productivité mais pas les contraintes relatives à l'instance selon sa position dans la hiérarchie de classes), soit encore à des versions incomplètes (des buts sont encore non valués) qu'il reste à traiter.

11.3.3 Discussion

La mise en place d'un contrôle d'évaluation sur les moyens d'inférence apporte quelques nuances dans le fonctionnement du couplage TROPES/MICRO. Par exemple, il est capable d'inhiber le pouvoir inférant d'une contrainte. Ainsi, lorsque le domaine d'un attribut est réduit à un singleton par la propagation de contraintes, le contrôle d'évaluation est seul juge de l'acceptation de la valeur comme valeur effective de l'attribut. C'est donc à lui d'entériner l'inférence alors que nous l'avons considérée comme automatique dans notre étude. Toutefois, si la réduction de domaine n'est pas le moyen d'inférence privilégié d'un attribut mais un détachement procédural ou une valeur par défaut, il est vérifié que la valeur inférée figure bien dans le domaine contrôlé par MICRO. Aussi, la présence de contrainte n'est pas inhibée, elle reste au second plan pour réapparaître lors de la vérification de la cohérence de l'inférence.

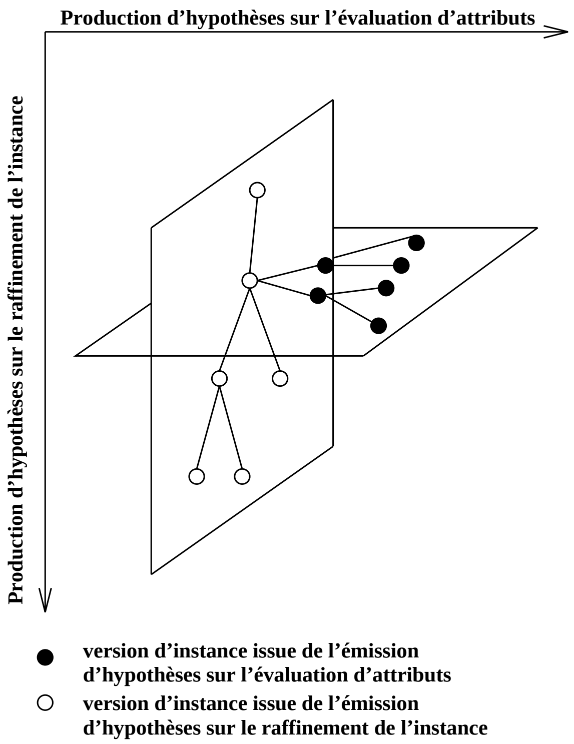


FIG. 11.6 - : Le système d'assistance explore toutes les évolutions possibles d'une version. Ces évolutions peuvent être le fruit de deux types de production d'hypothèses.

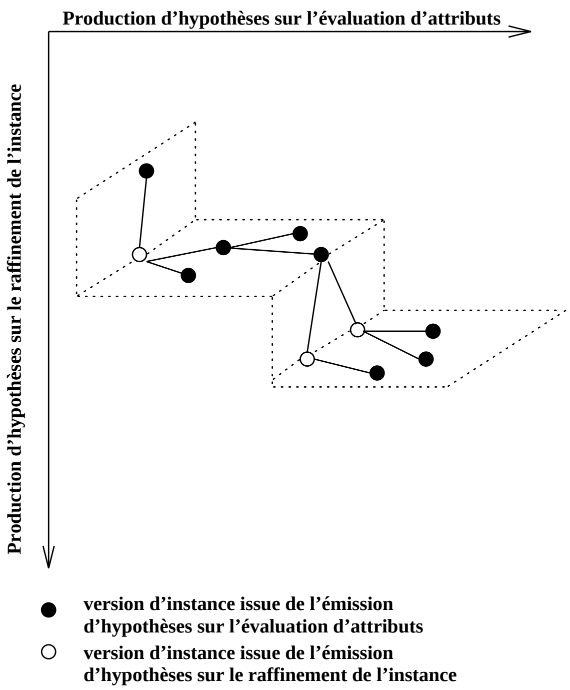


FIG. 11.7 - : Schéma d'alternance des deux types de production d'hypothèses.

Le contrôle d'évaluation a également le pouvoir d'orienter à un certain moment une contrainte puisque, en fonction des priorités données aux moyens d'inférence sur les attributs, il se peut que seuls certains attributs d'une contrainte aient la permission d'être évalués par cette contrainte.

Lorsqu'un ATC est évalué par un autre moyen d'inférence que la propagation de contrainte, MICRO est placé dans la situation d'une modification (affectation) de la valeur. Les contraintes portant sur cet attribut sont activées. Si la propagation de contraintes vide un domaine d'un ACT du réseau, la valeur s'avère incohérente, l'instance est considérée comme incohérente.

Tels qu'ils ont été conçus, le contrôle d'évaluation et le système d'assistance au raisonnement hypothétique n'utilisent que les résultats de la propagation de contraintes effectuée par MICRO. Les domaines des AIC sont consultés pour décider de la réussite d'une inférence par contrainte et, plus généralement, pour la cohérence des valeurs inférées.

La résolution de contraintes n'est pas employée ici. Pourtant, elle pourrait l'être notamment en situation de raisonnement hypothétique afin de compléter des versions d'instances pour lesquelles il reste des buts en situation de blocage (il n'est pas possible de déterminer leur valeur) ou même pour des versions d'instances qui sont solutions de contraintes de productivité mais non encore complètes (des attributs non buts sont encore sans valeur).

Réciproquement, le système d'assistance au raisonnement hypothétique se charge de la complétion des instances de manière automatique par les moyens d'inférence – résolution de contraintes exclue – fournis dans la description des objets. Aussi, son travail est celui qui serait effectué à un moment donné par la résolution de contraintes dans la phase d'énumération ou de division des domaines. Par conséquent, le raisonnement hypothétique choisit lui même des branches de l'arbre de recherche de solutions jusqu'à la complétion totale (satisfaction de la contrainte de productivité) ou jusqu'à ce qu'une incohérence soit détectée. L'emploi du système d'assistance au raisonnement hypothétique peut donc être vu comme un moyen d'exploiter au maximum les connaissances sur les instances constructibles, et donc, comme un moyen de retarder d'autant le recours à la résolution de contraintes. Ce système se charge alors lui-même de l'élagage de l'arbre de recherche de solutions.

La résolution peut être alors ultimement utilisée lorsque toute inférence est impossible en employant les moyens d'inférence présents. Elle serait lancée sur la configuration courante d'une version hypothétique d'instance pour tout but contraint non atteint. Pour ce faire, il suffit d'augmenter le langage de contraintes de productivité d'un opérateur de résolution.

Il est également envisageable d'étendre l'émission d'hypothèses en commandant l'énumération du domaine d'un ACT but. À un niveau de la hiérarchie de versions hypothétiques, se trouvent alors les différentes valeurs du domaine de l'ACT, chaque version hypothétique de l'instance rendant compte de la propagation de contraintes dans un réseau où l'attribut a une valeur donnée de son domaine. De même, pour passer à un niveau inférieur de la hiérarchie, on demande à MICRO l'énumération ou la division du domaine d'un autre attribut but.

11.4 Conclusion

Ce chapitre a montré trois études conceptuelles qui font usage des contraintes.

- Bien qu'initialement conçu comme un mode d'inférence de valeur par défaut, le partage de propriétés entre les attributs d'un objet composite et ceux de ses objets composants peut être pris en charge par les contraintes. Le recours à des facettes n'est pas nécessaire, les contraintes indiquent le sens dans lequel se fait le partage et peuvent limiter les inférences possibles. La description du partage n'est plus éclatée dans les objets héritiers, mais elle est regroupée au niveau de l'objet composite par l'intermédiaire de contraintes de classe. Toute l'information inhérente à l'objet composite est donc accessible dans cet objet – ce qui est un principe essentiel préservé de la RPO. Bien que l'étude ait été faite sur les objets composites, le recours aux contraintes pour représenter le partage de propriétés entre attributs d'objets distincts peut se faire dans le cadre plus général des objets complexes.
- Dans le modèle de tâches, c'est aux contraintes qu'est déléguée la responsabilité de décrire et d'assurer le passage de paramètres d'une tâche vers ses sous-tâches. Là aussi, si des facettes peuvent être employées pour faciliter la description du flot de données, leur sémantique est prise en charge par des contraintes gérées par MICRO.
- La présence de plusieurs moyens d'inférence de la valeur d'un attribut, et le recours au raffinement d'une instance pour la compléter, sont les principales raisons de la mise en place d'un contrôle d'évaluation des attributs, qui repose alors sur l'établissement de priorités entre ces moyens d'inférence dans une hiérarchie de classes. Un réseau de contraintes est considéré comme inférant s'il a réduit le domaine d'un attribut à un singleton et si la priorité est effectivement à l'inférence par réduction de domaine pour cet attribut. Néanmoins, les résultats de la propagation de contraintes interviennent au moment de valider toute autre inférence. Ce principe est repris lors de l'exploration d'une base TROPES sous le contrôle d'un système d'assistance au raisonnement hypothétique. Le recours à un raisonnement hypothétique est un moyen de faire reculer la combinatoire sous-jacente à la résolution de contraintes. Mais, réciproquement, ce raisonnement hypothétique pourrait être ultimement étendu par la résolution de contraintes, ou même incorporé une résolution progressive.

Chacun des deux chapitres suivants propose une étude conceptuelle de deux notions – la notion de relation et la notion de filtre – dans le cadre de TROPES, et montrent comment ces notions exploitent également la présence de contraintes.

Chapitre 12

La notion de relation dans Tropes

Les chapitres de la partie précédente ont décrit comment exprimer (chapitre 6) et gérer (chapitres 7 et 8) des CSP dynamiques dans TROPES. Nous avons dressé la liste des conséquences de la présence de contraintes dans TROPES (chapitre 10) et présenté comment celles-ci peuvent être exploitées par le modèle lui-même pour étendre les capacités de ses fonctionnalités (chapitre 11).

Dans ce chapitre, nous poursuivons l'exploitation que peut faire le modèle de la présence de contraintes : nous faisons la première de nos deux propositions d'extensions de TROPES. Elle concerne la notion de *relation*.

Le terme de relation a ici un sens très général et rejoint celui usité dans les autres SRPO (*cf.* section 2.1.1.7). Il fait référence à tout lien entre objets que peut modéliser un attribut. Nous sommes plus particulièrement intéressés par l'expression et la maintenance de ces liens ou relations. Nous décrivons comment, ici encore, les contraintes sont mises à profit pour mener à bien cette tâche.

Nous commençons par faire un état des lieux des relations dans TROPES (*cf.* section 12.1) qui dévoile les limites des outils classiques de représentation pour exprimer la sémantique et les propriétés attachées à une relation. Autrement dit, la notion de relation est à l'état embryonnaire dans TROPES. Nous faisons alors une proposition pour combler ces lacunes à grand renfort de contraintes (*cf.* section 12.2). La façon dont les contraintes sont exploitées dans l'expression et la maintenance d'informations relatives à la relation est alors plus précisément décrite (*cf.* section 12.3). Nous proposons de construire des relations à partir de relations existantes et de la panoplie d'opérateurs délivrée par MICRO (*cf.* section 12.4), avant de comparer la gestion des relations proposée dans TROPES à celle effectuée dans les autres SRPO concernés (*cf.* section 12.5).

12.1 L'état actuel des relations dans Tropes

12.1.1 Représentation des liens

Dans la spécification originelle du modèle TROPES, trois natures d'attributs ont été retenues – propriété, composant, relation. Dans un objet, un attribut de nature *relation* est destiné à modéliser une *relation* dans laquelle peut être impliqué cet objet. Lorsque cet attribut a une valeur alors il existe un lien entre cet objet et un ou plusieurs objets du même concept ou de concepts différents.

Il n'y a pas ici de limite dans l'acception du terme *relation* à partir du moment où cette relation peut être représentée par un attribut. Toutefois, la relation de composition dispose d'une sémantique particulière dans TROPES et la définition d'un attribut de nature *composant* indique au modèle que cet attribut doit être placé sous le contrôle associé à la composition.

Un attribut de nature *relation*, comme tout autre attribut de TROPES, peut être mono ou multivalué. Sa valeur est une instance ou un ensemble ou une liste d'autres instances.

Des exemples de tels attributs liens sont, par exemple, *père-de, a-pour-voiture, a-pour-fils, fils-*

de... Les exemples de liens choisis ont des noms suffisamment explicites pour qu'il soit permis de comprendre quelle relation est associée à chacun. Ainsi, on conçoit que :

- le lien *père-de* permet de modéliser la relation de paternité entre une personne (objet du concept PERSONNE) et une ou plusieurs autres personnes (objets du même concept) ;
- le lien *a-pour-voiture* permet de modéliser la relation de propriété entre une personne (objet du concept PERSONNE) et une ou plusieurs voitures (objets d'un autre concept VOITURE). Il apparaît clairement ici que le lien est orienté d'un concept (PERSONNE) vers un autre concept (VOITURE) et que, sans doute, un lien *voiture-de* entre un objet du concept VOITURE et un objet du concept PERSONNE permettrait de modéliser la relation inverse.
- les deux liens *a-pour-fils* et *fils-de* sont des liens inverses opérant sur le même concept PERSONNE, par exemple. La présence de ces deux liens dans un même objet est destinée à informer, de manière bi-directionnelle, sur le ou les objets avec le(s)quel(s) un objet est en relation. La description des liens inverses permet d'exprimer deux relations orientées et inverses, représentées par les liens.

  L'existence de liens inverses définis sur le même concept permet de disposer, dans le même objet O, d'une vision "transitive" de la relation. Par ce moyen, par exemple, on sait que l'objet O est en relation par le lien l avec un objet O', mais aussi en relation par le lien l^{-1} (lien inverse de l) avec un objet O''. Autrement dit O'' est en relation par le lien l avec O.

  L'existence de liens inverses définis entre deux concepts distincts permet de connaître, quel que soit l'objet que l'on considère dans un des deux concepts, le ou les objets de l'autre concept avec le(s)quel(s) il est en relation.

  Notons qu'il faut distinguer ici la relation de filiation (modélisée par les liens *fils-de, a-pour-fils*) de la relation de paternité (modélisée par les liens *père-de, a-pour-père*). La première lie des individus de sexe masculins à deux individus mâle et femelle, la seconde un individu de sexe masculins à plusieurs individus de sexes indifférents. C'est une façon d'exprimer que la relation de filiation (au sens décrit ici) est une sous-relation (au sens de l'inclusion ensembliste des graphes de ces relations) de la relation de paternité[1].

Ces exemples donnent une idée de la fonctionnalité des attributs de nature *relation*. Les attributs liens, comme tout autre attribut de TROPES, font l'objet d'une définition où sont décrits le type et/ou la cardinalité de l'attribut. Ces contraintes de typage exprimées par des facettes suffisent à contrôler en type et en nombre l'objet ou les objets avec le(s)quel(s) l'objet courant est en relation.

Mais les limitations des outils de représentation du noyau de TROPES énoncées au chapitre 3 resurgissent également dans la modélisation des relations.

12.1.2 Les limites

Il faut se rendre à l'évidence que la panoplie de facettes disponible ne permet en somme qu'une vérification du type des objets en relation avec l'objet courant à travers le lien. Les insuffisances des outils actuels de représentation des relations portent sur trois points.

1. Dans l'optique d'une représentation bi-directionnelle des relations, l'objectif est de gérer les liens et leurs inverses, le couple (lien, lien-inverse) assurant la description d'une relation bi-directionnelle entre deux objets. Or, il est clair que si un objet O' apparaît dans la valeur d'un attribut lien l d'un objet O, alors O apparaît dans la valeur de l'attribut l^{-1} lien inverse de l dans O'. Dans l'exemple, si un objet O apparaît dans la valeur de l'attribut *a-pour-fils* d'un objet O', alors O' apparaît dans la valeur de l'attribut *fils-de* de l'objet O.

   Dans TROPES, rien n'est prévu pour signaler que deux attributs liens sont inverses l'un de l'autre et, qu'en conséquence, l'obtention de la valeur de l'un deux permet de déduire des informations sur la valeur de l'attribut lien inverse dans chacun des objets mentionnés par le

[1] Cette sous-relation peut s'apparenter à la notion de hiérarchie de rôles des langages terminologiques [Haton et al.91, Napoli92].

lien. Il nous semble que cette propriété d'inverse doit être signalée afin que, dans ce cas, les inférences de valeurs sous-jacentes soient réalisées automatiquement par le système.

2. Bien que les relations représentées ne soient pas, *a priori*, des relations de nature mathématique, un certain nombre de propriétés mathématiques intéressantes telles que la réflexivité, la symétrie, l'anti-symétrie, la transitivité, etc., indiquent comment compléter et contrôler la connaissance relative à une relation. Rien n'est prévu dans TROPES pour la spécification de telles propriétés dans les attributs liens, encore moins pour la gestion inhérente au maintien de ces propriétés.
   Par exemple, savoir qu'une relation est symétrique (entre objets du même concept) permet d'inférer que si un objet O a son lien l valué avec O' alors O' a son lien l valué avec O.
3. L'expression de la sémantique d'une relation se limite aux facettes de ses attributs liens dans le modèle TROPES initial. Or, il existe sans doute des propriétés (ou des contraintes) sous-jacentes à la déclaration d'un lien entre deux objets, qui sont inhérentes à la sémantique ou/et à la cohérence de la relation.
   Par exemple, on peut exiger, pour la cohérence, que l'âge d'un père soit toujours supérieur ou égal de 10 années à l'âge indiqué pour ses fils.
   Là encore, le noyau de TROPES est insuffisant pour exprimer et gérer de telles informations contingentes (propres ou immanentes) à la signification de la relation.

De ces insuffisances, il se dégage trois objectifs à atteindre. Avant de proposer un moyen de représenter les relations qui comblent ces insuffisances, nous décrivons le type des relations qui sont appelées à être représentées désormais en TROPES grâce à cette proposition.

12.2 Proposition d'extension de la notion de relation

12.2.1 Détermination des relations à exprimer

James Rumbaugh [Rumbaugh87] affirme que considérer des relations d'arité supérieure à trois est rarement utile et que l'on peut représenter des relations $n - aires$ par des opérations binaires. Nous nous rallions à ce constat et limitons donc notre proposition à la représentation de relations d'arité deux ou trois, c'est-à-dire aux relations dont un élément du graphe est un doublet ou un triplet d'objets (de concepts ou de classes, disjoint(e)s ou non).

12.2.1.1 Relations binaires

Les exemples donnés jusqu'à présent sont des illustrations de relations binaires entre deux concepts ou classes non forcément distincts. Selon la cardinalité du lien, il faut plutôt parler d'*associations* comme il est entendu dans le *modèle Z* des bases de données, par exemple [Delobel et al.82].

Un attribut lien monovalué représente une association 1 : 1 entre deux objets.

Par exemple, l'attribut *a-pour-fils* représente une association 1 : n dans le concept PERSONNE, l'attribut *a-pour-père* représente une association 1 : 1 dans le même concept.

Une association 1 : n est une relation binaire entre deux concepts ou classes non forcément distinct(e)s, car son graphe est constitué de n associations 1 : 1. Elle est représentée par un attribut lien multivalué dont la valeur est un ensemble de n objets. Par exemple, les n couples (père,fils) issus des n valeurs de l'attribut *a-pour-fils* d'un objet père reflètent n associations 1 : 1. La relation est bien binaire : elle lie un objet d'un concept à un autre objet du même concept ou d'un autre concept, son extension ne contient que des couples.

Pour les associations 1 : n symétriques qui portent sur les objets d'un même concept, le lien et son inverse sont confondus. Le nom est le même car la sémantique est identique.

Par exemple, la relation de fraternité entre deux personnes est représentée par le seul lien *frère-de*, qui est également son inverse.

12.2.1.2 Relations ternaires

Les relations ternaires ont pour but de lier trois objets de concepts (ou classes) non forcément distincts.

Un exemple d'une telle relation est l'action de don qui lie le donneur, le receveur et l'objet donné (et reçu). Supposons qu'elle lie trois objets : deux du concept PERSONNE (le donneur et le receveur) et un du concept OBJET. Trois liens (et leurs inverses) peuvent servir à représenter cette relation :

- *donne-à* lie le donneur au receveur ; son inverse *reçu-de* lie le receveur au donneur ;
- *donne* lie le donneur à l'objet donné ; son inverse *donné-par* lie l'objet donné au receveur ;
- *donné-à* lie l'objet donné au receveur ; son inverse *a-reçu* lie le receveur à l'objet donné.

Six liens assurent la bi-directionnalité de cette relation ternaire, qui peut donc se décomposer en trois relations binaires : une entre le donneur et le receveur, une entre le donneur et l'objet donné, une entre l'objet donné et le receveur.

12.2.2 Utilisation de contraintes

Si l'on considère les trois objectifs fixés pour étendre et assurer l'expression et la maintenance de relations dans TROPES, il apparaît immédiatement que le troisième – expression et maintenance de propriétés (ou contraintes) contingentes à la relation – doit être dévolu aux contraintes. Il s'agit simplement ici d'exprimer à l'aide des contraintes fournies par MICRO des relations numériques ou symboliques qui doivent être tenues entre les attributs des objets liés. Aux contraintes est donc confiée la réalisation du troisième objectif.

Concernant la complétion automatique d'un lien inverse à partir de l'affectation d'un lien, MICRO offre un mécanisme de propagation basé sur des démons qui semble tout à fait désigné pour procéder aux mises à jour attendues. Aux contraintes est donc confiée la réalisation du premier objectif.

Enfin, les propriétés mathématiques des relations (réflexivité, symétrie...) induisent un comportement similaire du système dans la complétion automatique d'attributs liens. Aussi, là encore, les contraintes se montrent tout à fait désignées pour assurer ce genre d'inférences. Aux contraintes est donc confiée la réalisation du second objectif.

Dans cette proposition de représentation de relations, les contraintes ont donc une place importante. Elles interviennent à la fois dans la description de la sémantique de la relation, mais aussi dans le maintien de la cohérence et l'inférence des informations.

Les moyens de parvenir aux trois objectifs fixés sont donc répertoriés, il reste à décrire leur mise en place.

12.3 Expression et maintenance de relations

Cette section présente comment peut être étendue dans TROPES la notion de relation dans les directions fixées par les trois objectifs énoncés. Pour chacun des points à satisfaire, une solution est proposée qui met largement les contraintes de MICRO à contribution.

Nous partons ici des fonctionnalités de représentation prévues initialement dans TROPES pour représenter les relations, qui se résument aux attributs de nature *relation*, définis avec les seuls outils de définition (descripteurs ou facettes) du noyau de TROPES.

Les attributs liens sont attributs d'un concept. Ils représentent une relation orientée entre ce concept et un autre concept non forcément distinct. Ces attributs, comme les autres, apparaissent dans les hiérarchies des points de vue où leur définition peut être précisée et affinée à l'aide de facettes. Ainsi, la définition d'un attribut lien dans le concept décrit les caractéristiques générales de la relation orientée (type du concept lié, arité de l'association), alors que la (re)définition dans une classe précise le type et le nombre d'objets liés.

12.3.1 Expression et maintenance des liens inverses

Rien ne permet d'indiquer dans TROPES qu'un attribut lien est l'inverse d'un autre attribut lien du même concept. Or, nous avons montré que la présence d'un lien et de son inverse dans une classe permet de considérer la relation comme bi-directionnelle et de connaître, à partir d'un concept ou d'une classe, tous les objets auxquels l'objet est lié par la relation (ceux qui sont ses images par la relation), mais aussi ceux dont il est l'image par la relation inverse.

Pour rendre cette information (relation d'inverse) explicite, on peut avoir recours à une facette (comme c'est le cas dans YAFOOL [Ducournau88]) qu'il faudrait alors ajouter à l'ensemble des descripteurs d'un attribut, soit au niveau de la définition pour le concept, soit au niveau de la définition pour la classe. Dans TROPES, cette solution ne nous apparaît pas satisfaisante car elle consiste à introduire une facette qui n'est utilisée que par un attribut de nature *relation* et rompt avec l'uniformité de définition des attributs.

En tant qu'outils pour l'expression et la maintenance de relations, les contraintes semblent tout à fait désignées pour exprimer et maintenir la relation inverse qui existe entre deux attributs d'un concept. Dans un premier temps, nous montrons comment déclarer que deux liens sont inverses. Dans un second temps, la maintenance de deux liens inverses est décrite en termes de contraintes impliquant ces liens.

12.3.1.1 Déclaration de liens inverses

Une contrainte de l'interface TROPES/MICRO est chargée d'exprimer que deux liens sont inverses. Cette contrainte spécifique, appelée `mic-inv-link`, est déclarée comme une contrainte de concept du lien dont elle est chargée d'indiquer l'inverse.

Si $l1$ est le nom d'un attribut lien du concept K_1 et $l2$ est le nom d'un attribut lien du concept K_2 (avec K_1 et K_2 non forcément identiques) alors la contrainte (`mic-inv-link` *l1 l2*) déclare les attributs liens $l1$ et $l2$ comme inverses l'un de l'autre.

Si les concepts K_1 et K_2 sont identiques alors il suffit de placer cette contrainte dans le concept.

Si les liens inverses sont confondus au sein d'un même concept (c'est le cas pour les associations $1 : n$ symétriques) alors la contrainte (`mic-inv-link` *l l*) est posée également de façon à ce que s'effectue la mise à jour.

Si les deux concepts K_1 et K_2 sont distincts, alors la contrainte devient orientée. On trouve dans K_1 la contrainte (`mic-inv-link` *l1 l2*) et dans K_2 la contrainte (`mic-inv-link` *l2 l1*).

Nous avons vu que les relations ternaires sont décomposables en trois relations binaires entre chacun des concepts liés. Dès lors, les liens inverses entre ces trois concepts se déclarent de la même façon.

La contrainte `mic-inv-link` est une contrainte complexe de l'interface TROPES/MICRO. Elle est en fait composée d'un ensemble de contraintes de base de MICRO impliquant également les liens, et dont le rôle est d'assurer la maintenance des liens inverses. Nous décrivons à présent cet ensemble.

12.3.1.2 Maintenance de liens inverses

Une fois que deux liens ont été déclarés inverses, il faut se donner les moyens d'assurer le partage d'informations entre ces liens. Cette tâche est confiée à la contrainte `mic-inv-link` qui se charge de la maintenance des liens inverses, grâce notamment aux accès qui permettent, depuis un objet, de désigner le lien inverse dans le ou les objets au(x)quel(s) il est lié.

- Pour une association de type 1 : 1, la contrainte est que le lien inverse ait pour valeur l'objet contenant le lien.
  Si la relation concerne deux concepts distincts K_1 et K_2 (K_1 contient l'attribut *lien* et K_2 contient l'attribut *lien-inverse* (lien inverse de l)), alors la contrainte `mic-inv-link` de K_1

est équivalente à (`mic-assign` *lien.lien-inverse self*) et la contrainte `mic-inv-link` de K_2 est équivalente à la contrainte (`mic-assign` *lien-inverse.lien self*) afin d'assurer le partage de valeur.
Si les concepts K_1 et K_2 sont confondus, la contrainte `mic-inv-link` du concept est formée de ces deux contraintes.

- Pour une association de type 1 : n non symétrique (les liens inverses sont distingués), il faut veiller à ce que l'objet apparaisse dans la valeur de l'attribut lien de chacun des objets auxquels il est lié. Inversement, le lien de l'objet-valeur (unique) de tout objet contenant le lien inverse doit contenir l'objet lié.
Si la relation concerne deux concepts distincts K_1 et K_2 (K_1 contient l'attribut *lien* et K_2 contient l'attribut *lien-inverse*), alors la contrainte `mic-inv-link` de K_1 est équivalente à la contrainte (`mic-map` *#'mic-assign lien.lien-inverse self*) puisque l'objet (dénoté par *self*) est valeur de l'attribut *lien-inverse* dans chaque objet de K_2 auquel il est lié. Réciproquement, la contrainte `mic-inv-link` de K_2 est équivalente à la contrainte (`mic-assign` *lien-inverse.lien* (`mic-cons` *self lien-inverse.lien*)) afin d'assurer le partage de valeur.
Si les concepts K_1 et K_2 sont confondus, la contrainte `mic-inv-link` du concept contient ces deux contraintes.
- Pour une association de type 1 : n symétrique (les liens inverses sont confondus au sein d'un même concept), il faut veiller à ce que l'objet apparaisse dans la valeur de l'attribut lien de chacun des objets auxquels il est lié. Inversement, le lien de l'objet-valeur (unique) de tout objet contenant le lien inverse, doit contenir l'objet lié.
La contrainte `mic-inv-link` est équivalente à l'ensemble des deux contraintes {(`mic-map` *#'mic-assign lien.lien self*), (`mic-assign` *lien.lien* (`mic-cons` *self lien.lien*))} afin d'assurer le partage de valeur.

Quelle que soit la relation binaire ou ternaire représentée, la propagation de valeurs entre un attribut lien et son attribut lien inverse est donc assurée par les contraintes composant la contrainte `mic-inv-link`. L'utilisateur n'a pas à se préoccuper de quelles contraintes est effectivement formée la contrainte `mic-inv-link`, le système s'en charge selon les caractéristiques des attributs liens donnés comme arguments de cette contrainte.

La transmission d'information sur les attributs liens multivalués montrent l'intérêt de disposer de contraintes spécifiques à ce type d'attributs, telles que la contrainte `mic-map` agissant sur des listes.

12.3.2 Expression et maintenance de propriétés

L'idée ici est d'exprimer et de maintenir les propriétés mathématiques classiques qui peuvent être requises pour une relation.

12.3.2.1 Expression des propriétés

Indiquer la propriété que doit posséder une relation est une information contingente d'un type particulier puisque la relation est obligée d'observer cette propriété. Aussi, dans le cadre de TROPES, les propriétés telles que la réflexivité, la symétrie, etc., ne sont plus déduites de l'examen du graphe de la relation, mais au contraire contrôlent sa formation.

Les attributs destinés à représenter des liens entre objets sont déclarés de nature *relation*. Nous proposons d'affiner la description de la nature de cet attribut. Ainsi le descripteur *nature* admet désormais comme valeur soit une des chaînes {*"propriétés", "composant", "relation"*}, soit un sous-ensemble de l'ensemble {*"relation-réflexive", "relation-symétrique", "relation-anti-symétrique", "relation-transitive"*}[2].

[2] L'ensemble { *relation-symétrique, relation-anti-symétrique étant exclu* } ; il est possible de factoriser ces ensembles

Ainsi, la nature de l'attribut indique également les propriétés que doit observer la relation qu'il représente.

Il faut noter que les propriétés exprimables (réflexivité, symétrie, anti-symétrie, transitivité...) ne concernent que les relations binaires, ce que représentent justement les liens.

12.3.2.2 Maintenance des propriétés

En ce qui concerne la maintenance, elle peut être confiée aux contraintes en utilisant également la notion d'accès pour passer, à travers les liens d'un objet O à l'objet auquel O est lié. Ainsi :

- la *réflexivité* impose que tout objet apparaisse dans la valeur de son lien. Cette propriété s'exprime par la contrainte (`mic-assign` *lien self*) si le lien est monovalué, ou (`mic-assign` *lien* (`mic-cons` *self lien*)) si le lien est multivalué, pour signaler que l'objet est valeur (ou dans la valeur) du lien.
- la *symétrie* impose que si un objet O fait partie de la valeur du lien dans un objet O', alors O' fait partie de la valeur du lien dans O. Cette propriété s'exprime par la contrainte (`mic-assign` *lien.lien self*) si le lien est monovalué, ou (`mic-assign` *lien.lien (*`mic-cons` *self lien.lien*)) si le lien est multivalué.
- l'*anti-symétrie* impose que si un objet O fait partie de la valeur du lien dans un objet O', alors O' ne fait pas partie de la valeur du lien dans l'objet O. Cette propriété s'exprime par la contrainte (`mic-neq` *lien.lien self*) si le lien est monovalué, ou (`mic-not-in` *self lien.lien*) si le lien est multivalué.
- la *transitivité* impose que si un objet O' fait partie de la valeur du lien dans un objet O, et si un objet O'' fait partie de la valeur du lien dans l'objet O', alors O'' fait partie de la valeur du lien dans O. Cette propriété s'exprime par la contrainte (`mic-assign` *lien.lien lien*). Il faut remarquer que cette contrainte assure la propagation des valeurs sur toute la fermeture transitive de la relation.

D'autres propriétés (anti-réflexivité, anti-transitivité...) peuvent également être exprimées et contrôlées par des contraintes.

Ainsi, en étendant l'ensemble des valeurs du descripteur *nature* de la définition d'un attribut pour le concept, il est possible d'exprimer les propriétés d'un attribut lien. À ces propriétés sont alors automatiquement associées des contraintes de concept, qui garantissent leur respect.

12.3.3 Expression et maintenance de la sémantique des relations

L'idée ici est d'étendre la sémantique de la relation en indiquant par l'intermédiaire des contraintes, des propriétés induites par la signification de la relation.

Contraindre les attributs de deux ou trois objets liés par une relation binaire ou ternaire peut se faire aisément à l'aide des accès qui permettent de désigner un attribut dans un autre objet. En effet, si l'on veut contraindre un attribut d'un objet à satisfaire une relation avec d'autres attributs des objets auxquels il est lié, il suffit d'accéder à ces attributs, via les liens.

Par exemple, si on souhaite contraindre un enfant à avoir une différence d'âge d'au moins dix ans avec son père, on peut l'exprimer par la contrainte (`mic-ge` (`mic-add` *10 age*) *a-pour-père.âge*). Dès qu'un objet enfant et qu'un objet père sont liés par la relation, la contrainte est posée. Son non-respect conduit au refus de l'établissement du lien : le contenu des instances n'est pas compatible avec la relation.

Lorsque la relation porte sur des concepts distincts contenant chacun un lien inverse, la contrainte n'est à associer qu'à un seul concept, puisque déclarer la contrainte dans les deux concepts serait inutile (les deux contraintes ont même effet) car redondant. On peut considérer que cette solution revient à considérer une orientation de la relation, ou la dominance d'un concept sur l'autre

---

en considérant les chaînes "relation d'équivalence" et "relation d'ordre", par exemple.

(relation maître/esclave). Il pourrait être envisagé de stocker cette contrainte dans une entité englobante puisqu'elle a trait aux deux concepts liés. Nous avons mené en ce sens des études sur la réification des relations, en tant qu'objets de TROPES. Il s'est avéré que cette réification présente dans TROPES plus de problèmes que les solutions qu'elle est censée apporter (création d'un concept prédéfini de relation qui rend la définition de relation peu modulable, difficulté d'établissement de la propagation de valeur entre liens, etc.). Finalement, ce problème – qui n'en est pas un – se résume au choix du concept d'accueil de la contrainte et on peut, après tout, toujours trouver des raisons pour décider du concept maître ou du concept esclave. En fait, une fois la contrainte posée – et il n'est nécessaire de la poser qu'une seule fois –, la maintenance est assurée quel que soit le concept choisi. Ceci est garanti par le caractère multi-directionnel des contraintes.

Les semi-méta-contraintes (*cf.* section 8.5) et les accès (*cf.* section 6.3) se révèlent être de précieux atouts dans l'expression et la maintenance de relations inter-attributs entre des instances de concepts liés. Sur l'exemple de la figure 12.1, l'attribut *nb-habitants* du concept COMMUNAUTÉS est lié à chacun des attributs *nb-habitants* des instances du concept ÉTATS qui apparaissent dans la liste des valeurs de l'attribut *membres* du concept COMMUNAUTÉS. La contrainte (`mic-assign` *nb-habitants* (`mic-apply` *#'mic-add membres.nb-habitants)*) est chargée de la maintenance de cette relation. Cette contrainte est en mesure de calculer le nombre d'habitants d'une communauté d'États à partir de la somme des habitants de chaque État. Mais de plus, elle est capable de réagir, grâce à la gestion des accès, à toute modification de la valeur de l'attribut *membres* (par exemple, si on ajoute ou retire des membres), ainsi qu'à la modification du nombre d'habitants d'un des États membres. On obtient ici un fonctionnement analogue aux méta-contraintes proposées dans

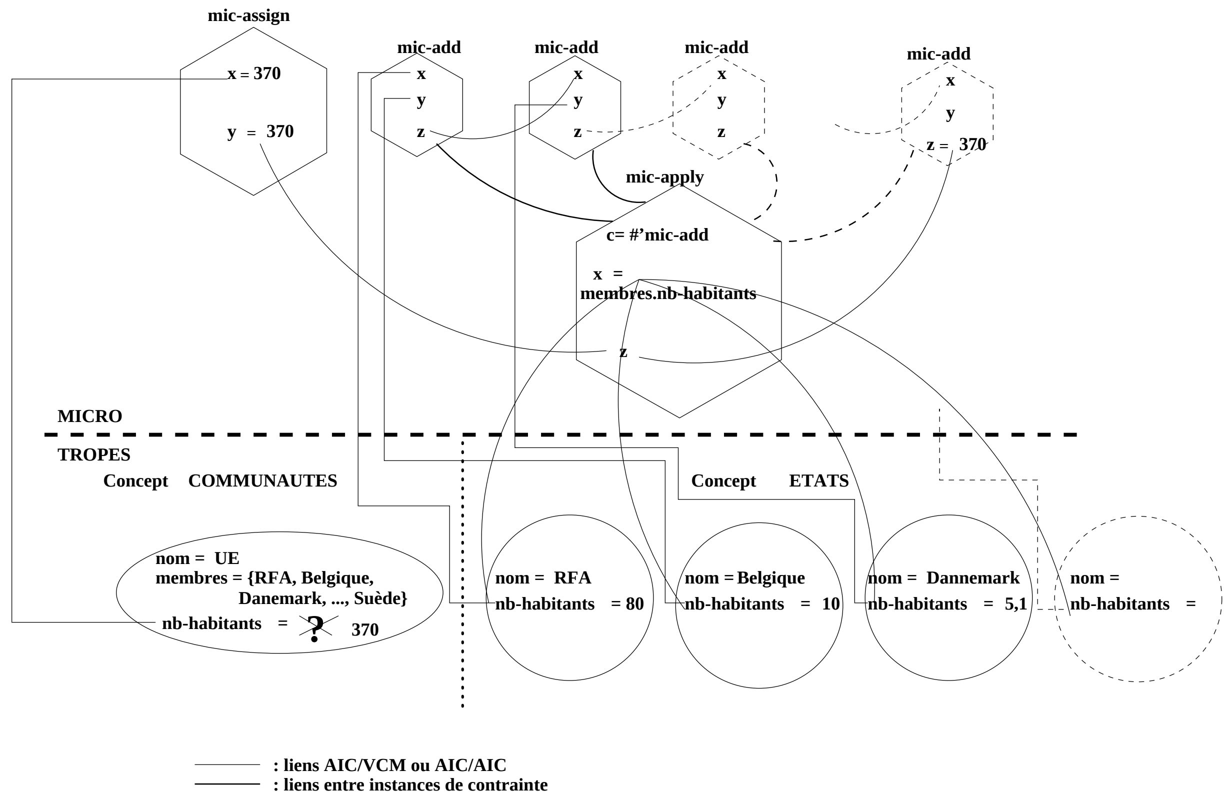


FIG. 12.1 - : Exemple de l'utilisation de semi-méta-contraintes pour la sémantique d'une relation. Ici, la contrainte (`mic-assign` *nb-habitants* (`mic-apply` *#'mic-add membres.nb-habitants*)) contrôle la valeur de l'attribut *nb-habitants* du concept COMMUNAUTÉS à travers un accès par le lien *membres* au concept ÉTATS. Les contraintes `mic-add` sont posées sur les éléments de l'accès *membres.nb-habitants*, deux à deux. Le résultat final est transmis par la contrainte `mic-assign` à l'attribut *nb-habitants*.

PROSE [Berlandier92b] (*cf.* section 5.1) mais les méta-contraintes sont d'une construction et d'une expression plus facile, qui notamment ne nécessitent pas la structuration en niveaux “méta” des contraintes.

12.4 Construction de relations

Ayant étendu la notion de relation dans TROPES vers l'expression et la maintenance à la fois des liens inverses, des propriétés et de la sémantique, il est tentant à présent de s'intéresser à l'extension de l'ensemble des relations lui-même, c'est-à-dire à la construction de nouvelles relations dans TROPES à partir des relations existantes.

L'algèbre relationnelle fournit à de telles fins une gamme d'opérateurs dont nous avons cherché la correspondance (expression et maintenance) dans TROPES. De même, la composition de relations a été étudiée.

12.4.1 Contraintes et algèbre relationnelle

Les opérateurs de l'algèbre relationnelle, tels qu'on les trouve notamment dans les bases de données [Delobel et al.82] sont le complément, la projection, l'anti-projection et la sélection (opérateurs unaires), la somme, le produit, l'union, l'intersection, la différence, la division, le produit cartésien et le θ-produit (opérateurs binaires).

La somme et le produit n'ont d'intérêt que si les relations portent sur les mêmes domaines (si les liens sont définis sur le même concept et portent sur le même concept). Dans ce cas, ils sont équivalents respectivement à l'union et à l'intersection. La remarque vaut aussi pour le produit cartésien de deux relations qui équivaut au produit de deux relations dans les conditions qui nous intéressent.

Nous cherchons ici à exprimer et maintenir les relations résultant de ces opérateurs relationnels afin de construire des relations dans TROPES. Plutôt que d'introduire dans un premier temps un langage d'opérateurs relationnels destinés à combiner des attributs liens de TROPES, nous cherchons une correspondance entre la sémantique de ces opérateurs et les contraintes dont dispose TROPES. Ainsi, si un lien peut être construit à partir d'un ou deux autre(s) lien(s) par application d'un opérateur relationnel, il suffit de l'impliquer dans l'expression de contraintes qui correspond à cet opérateur. La construction de relations en TROPES s'effectue alors en deux temps : 1) déclaration dans le concept de l'attribut lien associé à la relation et 2) pose de la contrainte de concept correspondant à l'opérateur relationnel utilisé pour construire la relation.

12.4.1.1 Union de relations

Soient $l1$ et $l2$ deux attributs liens d'un concept K_1 vers un concept K_2. Soit l'attribut lien $l3$ de K_1 vers K_2 défini comme l'union de $l1$ et $l2$ ($l3 = l1 \cup l2$). $l3$ est destiné à modéliser la relation qui est l'union des relations modélisées par $l1$ et $l2$. Cette union de liens peut être exprimée et maintenue par la contrainte (`mic-assign` *l3* (`mic-union` *l1 l2*)).

12.4.1.2 Intersection de relations

Soient $l1$ et $l2$ deux attributs liens d'un concept K_1 vers un concept K_2. Soit l'attribut lien $l3$ de K_1 vers K_2 défini comme l'intersection de $l1$ et $l2$ ($l3 = l1 \cap l2$). $l3$ est destiné à modéliser la relation qui est l'intersection des relations modélisées par $l1$ et $l2$. Cette intersection de liens peut être exprimée et maintenue par la contrainte (`mic-assign` *l3* (`mic-inter` *l1 l2*)).

12.4.1.3 Différence de relations

Soient $l1$ et $l2$ deux attributs liens d'un concept K_1 vers un concept K_2. Soit l'attribut lien $l3$ de K_1 vers K_2 défini comme la différence de $l1$ et $l2$ ($l3 = l1 - l2$). $l3$ est destiné à modéliser la relation qui est la différence des relations modélisées par $l1$ et $l2$. Cette différence de liens peut être exprimée et maintenue par la contrainte (`mic-assign` *l3* (`mic-diff` *l1 l2)*).

12.4.1.4 Division de relations

La division de relations ne peut s'exprimer par un lien dans le contexte de TROPES. En effet, le résultat de la division d'une relation binaire par une relation (unaire) est une relation unaire. Par contre, le résultat de cet opérateur peut être obtenu par un filtre (*cf.* chapitre 13) qui consiste à rechercher parmi les instances du concept de départ (respectivement d'arrivée) de la relation binaire, celles dont le lien contient un ensemble donné d'instances du concept d'arrivée (respectivement de départ).

12.4.1.5 θ-produit de relations

Soient $l1$ et $l2$ deux attributs liens d'un concept K_1 vers un concept K_2. Soit l'attribut lien $l3$ de K_1 vers K_2 défini comme le θ-produit de $l1$ et $l2$ où θ représente un opérateur de comparaison entre deux attributs de K_2. La valeur de $l3$ est formée des instances de K_2 apparaissant à la fois dans la valeur de $l1$ et de $l2$, et pour lesquelles la condition $a\ \theta\ b$ est respectée.

Aussi, le θ-produit de liens peut être exprimé et maintenu par la contrainte (`mic-assign` *l3* (`mic-select` *#'mic-θ* (`mic-inter` *l1 l2)))* où `mic-`θ est la contrainte secondaire exprimant la condition θ. Le fonctionnement de la contrainte secondaire `mic-select` est détaillé ci-dessous pour l'opérateur de sélection.

12.4.1.6 Complément d'une relation

La valeur de l'attribut lien complément $\neg l$ d'un attribut lien l d'un concept K_1 vers un concept K_2 est formée de l'ensemble des instances de K_2 qui n'apparaissent pas dans la valeur de l. Ce qui peut se faire par la contrainte (`mic-assign` $\neg l$ (`mic-diff` *(filter* K_2 { }*)))* où *(filter* K_2 { }*)* est un filtre vide sur les instances de K_2 (qui retient donc toutes les instances de K_2).

Il faut remarquer ici qu'un filtre est argument d'une contrainte, ce qui constitue également un emploi intéressant des filtres développé au chapitre suivant.

12.4.1.7 Projection d'une relation

La projection (comme l'anti-projection) est un opérateur qui travaille sur le graphe des relations. Il ne peut donc pas donner lieu à de nouveaux liens.

La projection sur K_1 d'une relation d'un concept K_1 vers un concept K_2 est l'ensemble des instances de K_1 pour lesquelles le lien associé à la relation a une valeur.

La projection sur K_2 d'une relation d'un concept K_1 vers un concept K_2 est l'ensemble des instances de K_2 pour lesquelles le lien associé à la relation inverse – donc, le lien inverse – a une valeur.

Ces ensembles peuvent donc être obtenus par filtrage.

12.4.1.8 Anti-projection d'une relation

L'anti-projection sur K_1 (respectivement K_2) d'une relation d'un concept K_1 (respectivement K_2) vers un concept K_2 (respectivement K_1) consiste à déterminer les instances de K_1 (respectivement K_2) qui apparaissent dans la valeur du lien associé à la relation inverse – donc, le lien inverse – dans chaque instance de K_2 (respectivement K_1) où ce lien inverse est défini.

Il suffit de filtrer le concept K_2 ((respectivement K_1) pour n'en retenir que les instances où le lien associé à la relation inverse est défini, puis de filtrer à son tour le concept K_1 (respectivement K_2) pour ne retenir que les instances où le lien associé à la relation a pour valeur le premier filtre obtenu. Ici, on imbrique donc deux filtres.

12.4.1.9 Sélection dans une relation

La sélection consiste à chercher dans le graphe d'une relation les éléments (ici, doublets, triplets) qui satisfont une propriété exprimée à l'aide d'opérateurs logiques.

Par exemple, à partir du lien *frères-de*, il est possible de sélectionner les frères d'une personne qui ont le même âge qu'elle, que nous appellerons frères jumeaux par abus de langage.

Dans le contexte de Tropes, il est clair que cette sélection s'opère dans chaque objet à partir d'un attribut lien (sur l'exemple, dans chaque objet personne, à partir du lien *frères-de*), et met en relation un ou des attributs autres que ce lien (sur l'exemple, l'attribut *âge*) avec un ou des attributs (sur l'exemple, l'attribut *âge*) des objets auxquels il est lié par le lien.

Il y a donc une idée de filtrage parmi l'ensemble ou la liste des instances valeurs du lien. En ce qui concerne la sélection de valeurs dans un ensemble ou une liste de valeurs, il faut s'en remettre à Micro qui propose un opérateur pour l'expression de contraintes : l'opérateur `mic-select`.

L'opérateur `mic-select` agit sur une variable multivaluée et forme le sous-ensemble ou la sous-liste des valeurs de cette variable qui satisfont un ensemble de conditions contrôlées par des prédicats ou par des contraintes booléennes, afin d'assurer un maintien de cohérence immédiat.

Or, dans le cas d'une sélection de relation, il s'agit de sélectionner des instances, parmi l'ensemble ou la liste d'instances qui forme la valeur d'un attribut lien, qui satisfont une condition. L'expression est donc un peu plus compliquée et nécessite un niveau d'indirection supplémentaire pour exprimer des conditions-test entre les attributs des instances données par la valeur de l'attribut lien et les attributs de l'objet contenant le lien. Cette indirection peut se faire à travers la définition d'une nouvelle contrainte basée sur la currification.

Ainsi, il est possible de créer un lien *frères-jumeaux-de* à partir du lien *frères-de*. La valeur de ce lien est contrôlée par la contrainte (`mic-assign` *frères-jumeaux-de* (`mic-select` #'mic-même-âge *frères-de*)) où la contrainte `mic-même-âge` est définie par :

```
(mic-create-constr mic-même-âge (X)
   (mic-is-eq X.âge self.âge)
)
```

Ici, l'accès *X.age* permet d'atteindre l'âge de chaque frère, alors que *self.âge* fait référence à l'âge de la personne (de l'instance) sur laquelle porte la contrainte principale `mic-assign`.

Cet exemple met en évidence la nécessité d'un contexte de pose de contrainte. En effet, la contrainte `mic-is-eq` porte sur chacun des objets atteignables à partir du lien *frères-de* mais lie chacun d'eux avec l'objet de départ – celui qui porte la contrainte principale `mic-assign`. Au moment de la pose des contraintes `mic-is-eq`, cette information doit être présente afin que les contraintes `mic-is-eq` soient considérées comme des propriétés de l'objet contenant le lien, et non pas comme des propriétés des objets liés. Dans ce contexte, l'accès *self.âge* désigne l'attribut *âge* de l'objet courant, pas celui des objets liés.

Puisqu'une contrainte contrôle à la fois le lien *frères-de* et l'attribut *âge* de chaque frère, il est garanti que toute modification de l'ensemble ou la liste des frères, toute modification de l'âge de l'objet courant, ainsi que toute modification de l'âge de l'un des frères, conduiront à la réactivation de la contrainte `mic-assign`.

12.4.2 Composition de relations

La composition de relations est la manière la plus simple de créer un lien à partir de deux autres liens, simplement en les juxtaposant.

Soient deux liens $l1$ et $l2$, pour que la composition de liens soit définie, il faut que le concept d'arrivée de $l1$ soit le concept de départ de $l2$. Dès lors on peut construire le lien $l3 = l1 \circ l2$ dont la valeur est calculable et contrôlée par la contrainte (`mic-assign` *l3 l1.l2*).

La composition de relation est donc représentée dans TROPES par un accès. Par exemple, le lien *grand-père-de* peut être évalué par la contrainte (`mic-assign` *grand-père-de* (`mic-union` *père-de.père-de père-de.mère-de)*).

Encore une fois, le fait que l'expression du lien composé se fasse à partir de contraintes garantit la cohérence de la valeur de ce lien avec les valeurs des liens qui le composent.

12.5 Comparaison avec d'autres SRPO avec relations

À la différence de la plupart des SRPO dédiés à la représentation de relations (SRL [Wright et al.84], Shood [Escamilla et al.90], Views [Davis87], Othelo [Fornarino et al.90b], DBMS [Rumbaugh87]), les relations ne sont pas des objets dans TROPES.

Les relations, à l'instar de YAFOOL [Ducournau88], sont intégrées dans les objets sous la forme d'attributs destinés à représenter les liens entre objets. Dans TROPES, le pari est fait d'augmenter l'expressivité de ces attributs liens (reposant uniquement sur l'ensemble des facettes) par les contraintes.

Dans TROPES, les relations de spécialisation et de composition sont gérées par le système. Aussi, la définition de relations dans TROPES a-t-elle un caractère moins général que celle proposée dans SRL, où cinq attributs d'un objet relation permettent de spécifier l'héritage (partage d'attributs) entre les objets liés. Dans TROPES, le partage de valeurs est assurée par les contraintes alors que les accès sont une façon de considérer le partage d'attributs.

Bien que les relations ne soient pas des objets, TROPES est capable des mêmes fonctionnalités que DBMS, notamment l'ajout ou le retrait d'un lien entre objets (par suppression de l'instance dans la valeur de l'attribut lien) ou la visualisation de tous les n-uplets d'une relation (par filtrage).

Si les diverses dépendances entre objets exprimables dans SHOOD ne sont pas reprises dans TROPES (excepté pour les objets composites où elles sont codées), le point commun entre SHOOD et TROPES se trouve dans la gestion des propriétés mathématiques et la construction de relations à partir d'opérateurs de l'algèbre relationnelle. Mais, d'une part TROPES propose un plus large éventail de propriétés et d'opérateurs, et d'autre part ces propriétés et ces opérateurs sont décrits en termes de contraintes et bénéficient donc d'une mise à jour immédiate garantie par les démons de la propagation de contraintes, alors qu'elles sont codées en SHOOD. De plus, ces propriétés et ces opérateurs sont exprimés dans les objets par les contraintes équivalentes alors que SHOOD propose d'en faire des instances de classes décrivant leur comportement.

Bien que VIEWS permette de contraindre les parties et les relations d'une vue, ces contraintes sont en fait des prédicats et n'ont pas de pouvoir inférentiel.

Enfin, les diverses classes de liens proposées dans OTHELO permettent le partage de valeur, conditionnel ou non, entre attributs d'objets liés. Des liens peuvent être déduits d'autres liens. Là encore, les mêmes résultats sont obtenus dans TROPES en impliquant les attributs liens dans des contraintes ; mais alors que la sémantique du partage ou de la construction de liens est contenue dans des méthodes associées à des classes prédéfinies de liens en OTHELO, dans TROPES, la sémantique de chaque lien de dépendance (attribut lien associé à une relation) est définie simplement et extensible à volonté, grâce à l'ensemble des contraintes de MICRO.

S'il ne représente pas les relations par des objets, TROPES est capable d'exprimer et de contrôler à la fois la valeur du lien inverse, les propriétés et les vérifications contingentes à la relation par un ensemble de contraintes portant sur l'attribut lien associé à la relation. Aussi, il remplit simplement l'ensemble des fonctionnalités que l'on trouve dispersées dans les divers SRPO traitant les relations.

12.6 Conclusion

Ce chapitre a présenté une proposition pour étendre la notion de relation dans TROPES. Après avoir montré les limites des outils mis à la disposition de l'utilisateur par le noyau du modèle TROPES dans l'expression et la maintenance de liens (relations) entre objets, nous avons proposé de mettre à profit la présence de contraintes dans TROPES afin de gérer :

- le lien associé à la relation inverse afin d'établir la bi-directionnalité et donc l'accès à l'information concernant une relation quel que soit le concept courant ;
- les propriétés mathématiques qui caractérisent une relation ;
- les contraintes sémantiques que doivent observer les objets liés.

Dans chacun de ces cas, les contraintes offrent un moyen d'inférence des informations implicites mais garantissent surtout, à tout moment, la cohérence des informations exprimées et, contrôlent de ce fait la validité de l'établissement d'un lien entre objets.

Nous avons poursuivi nos investigations vers la construction de relations. Là encore, les contraintes sont un support pour l'expression et le contrôle de la sémantique de la plupart des opérateurs de l'algèbre relationnelle. La notion d'accès est exploitée dans la composition de relations alors que les filtres compléteraient les opérations envisageables. Cette dernière notion a donc un intérêt certain. C'est pourquoi nous faisons dans le chapitre suivant une proposition visant à intégrer des filtres dans TROPES.

Chapitre 13

La notion de filtre dans Tropes

Comme le montre la section 2.1.2.3, la notion de filtre est exploitée sous diverses formes dans de nombreux systèmes de représentation de connaissances à objets. Cette étude a révélé deux caractères du filtre :

- le filtre permet d'associer à une entité de représentation (le plus souvent une classe) un ensemble de conditions en étendant simplement la définition (l'intension) de celle-ci : c'est là son caractère descriptif ;
- le filtrage est le mécanisme sous-jacent associé dont le rôle est de parcourir l'extension de la base fournie afin de déterminer les instances qui satisfont les conditions spécifiées par le filtre : c'est là son caractère inférentiel.

Ce chapitre formule un ensemble de propositions pour intégrer à TROPES la notion de filtre, absente des spécifications initiales du modèle.

Nous rappelons tout d'abord la définition du filtre et distinguons les filtres par l'ensemble d'instances qu'ils sont destinés à filtrer (*cf.* section 13.1).

Nous proposons alors de mettre à profit la présence du module de gestion de types METÉO dans TROPES pour utiliser le filtre, non plus seulement en tant que moyen d'inférence ou de requête, mais aussi en tant que descripteur de type d'un attribut (*cf.* section 13.2). Également, la présence de contraintes dans TROPES nous incite à étendre la description d'un filtre par des contraintes, d'une part, et à utiliser les filtres en tant qu'arguments de contraintes (*cf.* section 13.4), d'autre part.

Nous montrons alors que le mécanisme de filtrage peut s'appuyer sur les hiérarchies de types maintenues par MÉTÉO en proposant un pré-filtrage par les types afin d'accélérer la recherche des instances retenues par le filtre (*cf.* section 13.5).

Nous indiquons comment assurer la cohérence d'une base de connaissances en présence de chacun des types de filtre présentés (*cf.* section 13.6) avant de comparer les filtres TROPES aux filtres des autres SRPO (*cf.* section 13.7).

13.1 Définition et représentation des filtres

L'acception la plus commune du rôle d'un filtre dans le domaine des représentations de connaissances est de déterminer, parmi un ensemble d'instances, celles qui vérifient un ensemble de conditions énoncées par le filtre. La *base* du filtre est l'ensemble des instances à filtrer. Cette base est communément l'extension d'une classe qui forme dans tout système de représentation de connaissances un ensemble fini et éventuellement vide. Dans TROPES, nous faisons l'hypothèse que tout ensemble d'instances sur lequel est appliqué un filtre est constitué d'instances du même concept.

Or, un ensemble d'instances dans le même concept peut être désigné par :

- un concept : l'ensemble d'instances correspond à l'extension de ce concept ;
- une classe : l'ensemble d'instances correspond à l'extension de cette classe ;

- un ensemble de classes du concept chacune d'un point de vue différent : l'ensemble des instances correspond alors à l'intersection des extensions de chacune des classes (instances qui sont reliées à chacune des classes spécifiées) ;
- la valeur[1] d'un attribut dont le type est un concept, une classe, ou un ensemble de classes du concept, chacune d'un point de vue différent ;

Une base de connaissances TROPES est donc capable de fournir quatre types de base pour un filtre. Nous limitons ici notre étude à l'expression de filtre aux trois premiers types de base qui sont, en fait, la transposition à TROPES du cas classique. Le quatrième est un cas très particulier dû à la gestion d'attributs multivalués dont la valeur est un ensemble ou une liste d'instances. Son étude peut être intéressante dans le cas notamment de filtres sur les composants d'un objet composite ou de requêtes émises sur la valeur d'un attribut représentant une relation. Toutefois, on peut remarquer qu'un filtre d'attribut n'est rien d'autre qu'un raccourci dans l'expression d'un filtre de concept (de classe ou d'un ensemble de classes) avec lequel l'objet est en relation. On utilise simplement le fait que la relation est modélisée par un attribut pour directement filtrer les instances de ce concept (*i.e.* celles qui sont dans la valeur de l'attribut relation) avec lesquelles l'objet est en relation. Aussi, nous ne considérerons pas ici ce quatrième type de filtre.

Ces trois types de base nous permettent de parler désormais de *filtre de concept*, de *filtre de classe* et de *filtre d'un ensemble de classes.*

On peut voir un filtre (et le filtrage) comme une fonction de la base du filtre vers un sous-ensemble de cette base.

Pour les filtres de concept, on a :

$$\begin{aligned} filtre_K &: \mathcal{K} \rightarrow \mathcal{I} \\ & K \mapsto filtre_K(K) = I \mid I \subseteq extension(K) \end{aligned}$$

Pour les filtres de classe, on a :

$$\begin{aligned} filtre_C &: \mathcal{C} \rightarrow \mathcal{I} \\ & C \mapsto filtre_C(C) = I \mid I \subseteq extension(C) \end{aligned}$$

Pour les filtres d'un ensemble de classes, on a :

$$\begin{aligned} filtre_S &: P(\mathcal{C} \otimes \mathcal{P}) \rightarrow \mathcal{I} \\ & C_1/P_1 \ldots C_n/P_n \mapsto filtre_S(C_1/P_1 \ldots C_n/P_n) = I \mid I \subseteq extension(C_1) \cap \ldots \cap extension(C_n) \end{aligned}$$

Les conditions portent sur les attributs des instances à filtrer. Les descripteurs associés aux définitions des attributs pour un concept ou pour une classe sont les premiers moyens d'imposer des conditions sur ces attributs ; la présente étude a défini un autre moyen d'imposer des conditions : les contraintes. Nous proposons donc d'étendre l'expression des conditions d'un filtre par des contraintes. De même que les contraintes augmentent la déclarativité du modèle en permettant l'expression de relations entre attributs, elles doivent ici augmenter les moyens de description des conditions des filtres.

La définition d'un filtre se fait en étendant la définition de sa base par un ensemble de conditions. On peut donc voir un filtre comme le doublet $(base, \{conditions\})$. Aussi, la description d'un filtre passe par la description de la base et des conditions.

Dans TROPES, nous avons vu qu'il est possible de considérer quatre types de bases de filtres : concept, classe, ensemble de classe d'un concept, chacune d'un point de vue distinct, attribut (composant ou relation). La description de la base d'un filtre est donc la description d'un concept, d'une classe ou d'un ensemble de classes selon le type du filtre.

Quant à l'expression des conditions portant sur la base du filtre, elle peut se faire par les facettes de restrictions de types pour les conditions propres à un attribut et par les contraintes pour les conditions entre plusieurs attributs.

Ainsi, il apparaît que la définition d'un filtre de concept ou d'un filtre de classe se fait suivant le même schéma que la définition d'un concept ou d'une classe. Dans le cas d'un filtre sur un

[1] Cette valeur pourra également être une liste.

ensemble de classes, la similitude de définition est moins immédiate, notamment car il n'existe pas dans TROPES d'entité de représentation équivalente à un ensemble de classes. Nous étudions la définition de chacun des types de filtre :

- La définition d'un filtre de concept consiste à donner le concept qui sert de base, et un ensemble de redéfinitions d'attributs de ce concept – on garantira qu'il s'agit bien d'affinements – ainsi qu'un ensemble de contraintes entre les attributs du concept.

filtre-concept filtr1 =
{ *concept:* ÉTAT,
conditions: km-côtes > 3000,
contraintes: km-frontières < km-côtes
}

TAB. 13.1 - : Description d'un filtre de concept : on filtre le concept ÉTAT de façon à connaître les états du globe qui ont plus de 3000 kilomètres de côtes et ont plus de kilomètres de côtes que de frontières.

- La définition d'un filtre de classe consiste à donner la classe qui sert de base, et un ensemble de redéfinition d'attributs de cette classe – on garantira qu'il s'agit bien d'affinements – ainsi qu'un ensemble de contraintes entre les attributs de cette classe.

filtre-classe filtr2 =
{ *classe : Europe/géographique,*
conditions: km-côtes > 3000,
contraintes: km-frontières < km-côtes
}

TAB. 13.2 - : Description d'un filtre de classe : on filtre les États d'Europe qui ont plus de 3000 kilomètres de côtes et ont plus de kilomètres de côtes que de frontières.

- La définition d'un filtre sur un ensemble de classes consiste à donner l'ensemble de classes qui sert de base, et un ensemble de redéfinition d'attributs de ces classes – on garantira qu'il s'agit bien d'affinements – ainsi qu'un ensemble de contraintes entre les attributs de ces classes, redéfinis par le filtre ou non. Finalement, cette définition revient à redéfinir chacune des classes qui forment la base, à lui associer éventuellement des contraintes de classe supplémentaires et à associer éventuellement à cet ensemble de classes, un ensemble de contraintes entre les attributs de ces classes.

filtre-ens-classe filtr3 =
{ *classe : Europe/géographique & Riche/économique*
conditions: km-côtes > 1000,
contraintes: km-frontières < km-côtes, production-pêche > nb-habitants
}

TAB. 13.3 - : Description d'un filtre d'un ensemble de classes : on filtre les États riches européens qui ont plus de 1000 kilomètres de côtes, qui ont plus de kilomètres de côtes que de frontières, et qui pêchent plus d'une tonne de poissons par habitant.

13.2 Trois usages possibles d'un filtre dans Tropes

Un filtre TROPES n'est pas un objet TROPES bien qu'il puisse en avoir la forme, comme les filtres de concept ou les filtres de classes. Conserver les filtres dans TROPES reviendrait à conserver des concepts qui n'en sont pas vraiment, ou des classes qui n'en sont pas vraiment. L'utilité de la conservation des filtres réside notamment dans leur réutilisation dans une base peu ou pas modifiée. Dans TROPES, nous n'avons pas misé sur une telle hypothèse et ne souhaitons pas alourdir la base de connaissances avec des objets qui n'ont pas été créés comme tels par le concepteur ou l'utilisateur mais insérés par le système à des fins de réutilisabilité. En revanche, nous allons voir que, dans chacun des trois usages que nous proposons pour ces filtres dans TROPES, il correspond, à un moment donné, un type géré par le module de gestion de type.

13.2.1 Affinement de type

Le premier usage du filtre est suggéré par la définition du type d'un attribut dont la valeur est une instance, un ensemble ou une liste d'instances. Dans ce cas, le type est référencé par un concept, une classe ou un ensemble de classe d'un concept, chacune d'un point de vue distinct.

Aussi, on peut considérer que le type (la base, ici) peut être affiné davantage par un filtre de concept, un filtre de classe ou un filtre d'ensemble de classes. L'objectif de la donnée d'un filtre pour décrire le type d'un tel attribut peut donc être mise à profit pour raffiner les définitions de cet attribut sans pour autant faire référence à des concepts ou des classes existants, mais bien à des sous-ensembles de leur extension qui n'ont pas de définition dans TROPES. La propriété des filtres de représenter des concepts, ou classes, ou ensembles d'instances virtuels peut être ici exploitée dans la définition d'un type.

Dès lors, METÉO, le module de gestion de types de TROPES, intègre dans la hiérarchie de types correspondant à l'attribut, le type issu du filtre, c'est-à-dire le type du concept, de la classe, ou de l'intersection des types de classes désignés par le filtre, réduit éventuellement par la propagation de contraintes effectuée par MICRO.

L'emploi du filtre en tant que descripteur de type implique une utilisation particulière du filtre au moment de la vérification de type. Le principe est le suivant :

- Si le filtre ne comporte pas de contraintes dans les conditions, alors ces conditions ont un caractère *statique* pour le module de gestion de type et il suffit pour METÉO de tester si la valeur fournie pour l'attribut est bien du type du filtre.
- Si le filtre comporte des contraintes alors l'intension du filtre a un caractère dynamique. Non seulement, METÉO doit vérifier que la valeur est bien dans le domaine filtré, mais qu'elle satisfait aussi les contraintes. Et pour cela, MICRO sera sollicité. Ce n'est qu'après la vérification de la satisfaction des contraintes que la valeur sera acceptée.

L'usage d'un filtre avec contraintes en tant que descripteur du type d'un attribut est une proposition qui n'a pas d'équivalent dans les autres systèmes de représentation de connaissances par objets. Dans TROPES, elle est naturelle de par la présence simultanée de METÉO et de MICRO.

13.2.2 Inférence de valeur d'attribut

Dans SHIRKA, le prédécesseur de TROPES, les filtres sont employés comme moyen d'inférence par défaut de la valeur d'un attribut. Dans la définition d'un attribut, un filtre est déclaré par la facette `sib-filtre`. La description du filtre est semblable à la description d'une classe. Le principal emploi d'un filtre est la détermination d'un ensemble d'instances qui sont en relation avec l'instance courante. Cette relation est d'ailleurs souvent représentée par l'attribut d'attache du filtre :

- soit entièrement, c'est le cas dans l'exemple où l'on recherche les enfants d'une personne afin d'évaluer l'attribut *pere-de* à partir des personnes dont l'attribut *a-pour-pere* est valué avec la personne ;
- soit partiellement, c'est le cas dans l'exemple de filtres imbriqués où l'on recherche les petits-fils d'une personne en filtrant les personnes qui ont pour père une personne qui a pour père la personne.

Un autre emploi du filtre dans SHIRKA, moins usité, consiste à simuler une méthode de calcul. Le filtre balaye des sous-ensembles d'instances et affecte dans chaque sous-ensemble une valeur à un attribut. C'est le cas dans l'exemple où l'on cherche à associer à des tranches de salaires un nombre de points de base correspondant à la tranche. Un filtre est déclaré pour chaque tranche à qui il revient d'affecter l'attribut *nb-points-de-base* avec la valeur adéquate.

La description des filtres est facilitée en SHIRKA par deux facettes `var->` et `var<-` qui indiquent comment sont transmises les valeurs de l'instance d'attache du filtre aux instances de la base du filtre. D'autre part, l'attribut prédéfini *lui-même* permet la référence à l'objet d'attache du filtre. Cet outillage syntaxique est indispensable pour que s'opère le lien entre l'instance du filtre et les instances à filtrer, puis la sélection et enfin, la transmission du résultat du filtre à l'attribut.

Dans l'optique d'un usage du filtre dans TROPES en tant que moyen d'inférence, il convient de s'intéresser à la sémantique de l'utilisation des filtres en tant que moyens d'inférence par défaut.

Tout d'abord, le fait que ce moyen d'inférence soit par défaut implique que le filtrage doit être activé que lorsque la valeur de l'attribut est demandée mais n'est pas présente.

Au niveau de la cohérence du filtre, il est nécessaire que le résultat du filtre soit bien en accord, en type (et donc en cardinalité), avec le type de l'attribut. Une contrainte de typage serait d'exiger que la base du filtre soit l'ensemble des valeurs décrit par les facettes de typage. La contrainte de cardinalité est, en revanche, difficilement exprimable statiquement sur le filtre et nécessite une vérification dynamique de l'accord en nombre du résultat du filtre avec toute contrainte de cardinalité dans le cas d'un attribut multivalué.

Au cours du filtrage, le contenu d'une instance appartenant à l'ensemble à filtrer est comparé avec la description du filtre. Les valeurs contenues dans l'instance doivent satisfaire à la fois la partie statique de l'intension du filtre (les définitions d'attributs, celles données par la base ou redéfinies par le filtre) et la partie dynamique (les contraintes associées à la base du filtre et les contraintes déclarées dans le filtre). Ce processus de comparaison et d'identification est semblable à celui mis en œuvre lors de la classification. Aussi, l'instance en cours de filtrage peut s'avérer être :

- *sûre* lorsqu'elle appartient à la base du filtre et que son contenu satisfait l'intension (les conditions) du filtre ;
- *incomplète* lorsqu'elle appartient à la base du filtre mais que son contenu est incomplet et ne permet pas de tester toutes les conditions du filtre ;
- *impossible* lorsqu'elle appartient à la base du filtre mais que son contenu ne satisfait pas les conditions du filtre.

Le résultat de l'application d'un filtre à une base est donc constitué *a priori* de trois ensembles d'instances : les instances qui satisfont le filtre de façon sûre, les instances dont on considère qu'elles satisfont le filtre par manque d'informations (hypothèse du monde ouvert), les instances dont le contenu ne satisfait pas le filtre.

Dans l'utilisation du filtre dans la description d'un type, les instances sûres et incomplètes seront validées par la vérification de type.

Dans l'utilisation du filtre en tant que moyen d'inférence, l'objectif est de fournir la valeur d'un attribut. Il est possible d'adopter trois modes de délivrance du résultat du filtrage :

1. un mode semblable à la classification où le système restitue l'ensemble des instances sûres et l'ensemble des instances incomplètes. L'utilisateur doit alors choisir quelle(s) valeur(s) il associe à l'attribut ;
2. un mode où seules les instances sûres constituent le résultat du filtrage ;
3. un mode où l'ensemble des instances sûres et l'ensemble des instances incomplètes constituent le résultat du filtrage ;

Le premier mode laisse la validation du résultat final à l'utilisateur. Le filtre n'infère plus la valeur mais les possibilités de valeur. Ce mode peut donc paraître aller à l'encontre de ce que l'on peut attendre d'un mécanisme d'inférence puisqu'il sollicite l'utilisateur. Les deux autres reviennent à décider du niveau de confiance que l'on accorde au résultat du filtrage (strict dans le second mode, lâche dans le troisième). Le troisième mode semble correspondre davantage au mode de délivrance associé au filtrage pour une vérification de type.

Cependant, il demeure un problème essentiel à l'emploi de filtre en tant que mécanisme d'obtention de la valeur d'un attribut : l'indéterminisme lié à la satisfaction de la contrainte de cardinalité.

- pour un attribut monovalué, la contrainte de cardinalité spécifie qu'une seule valeur est attendue. Or, si le filtrage propose plusieurs instances (sûres ou/et incomplètes), alors un choix s'impose pour lequel l'utilisateur est sans doute le plus sûr critère (sinon, il serait exprimé par des conditions du filtre). Aussi, le premier mode de fonctionnement s'avère le plus approprié. Dans SHIRKA, la première instance est choisie, les autres écartées (le filtrage stoppe), ce qui peut sembler un peu cavalier.
- pour un attribut multivalué, s'il n'y a pas de contrainte de cardinalité, les deux derniers modes de délivrance sont les plus adéquats et la valeur est immédiatement disponible puisqu'elle ne nécessite pas de choix. En revanche, si une contrainte de cardinalité porte sur l'attribut, le premier mode de délivrance ne convient que dans le cas où le nombre d'instances retenues par le filtre n'est pas en deçà de la cardinalité minimale requise. En effet, s'il s'avère que le nombre d'instances (sûres ou incomplètes) retenues par le filtre est inférieur à la cardinalité requise, on peut considérer que l'inférence a échoué. D'autre part, dans le cas d'un attribut dont la valeur est une liste d'instances, il revient à l'utilisateur, si la contrainte de cardinalité est satisfaite par le filtre, d'établir un ordre, en plus d'un choix éventuel, sur les instances retenues. Dans SHIRKA[2], lorsque la contrainte de cardinalité est satisfaite, le filtrage stoppe.

Si le filtrage ne retient aucune instance de la base, on peut considérer que le résultat est l'ensemble vide.

L'utilisation du filtre comme mécanisme d'inférence implique donc la mise en place d'un contrôle sur le résultat du filtre qui peut nécessiter l'intervention de l'utilisateur dans les cas où le système ne peut de lui-même inférer la valeur de l'attribut. Ceci rend cette fonctionnalité finalement peu attractive pour un utilisateur, à moins que des décisions arbitraires, comme pour les filtres de SHIRKA, et non toujours satisfaisantes (si le nombre d'instances retenues ne satisfait pas les contraintes de cardinalité) soient imposées par TROPES sur l'application des filtres.

13.2.3 Formulation de requête

Les deux utilisations de filtre proposées ci-dessus – en tant que type ou en tant que moyen d'obtention de la valeur d'un attribut – attachent la déclaration du filtre à la définition de l'attribut dans la classe. Le filtre est donc une entité descriptive placée à l'intérieur des objets.

La troisième et dernière utilisation proposée ici vise, au contraire, à sortir les filtres des objets, et à les considérer comme des *requêtes*.

L'idée est d'établir une requête à l'aide d'un filtre afin de rechercher dans une base de connaissances les instances qui satisfont certaines conditions que le filtre exprimera. La requête prend pour cible un ensemble d'instances correspondant à la base du filtre. Aussi, une requête peut être formulée sur un concept entier (filtre de concept), sur une classe (filtre de classe) ou sur un ensemble de classes d'un concept, appartenant chacune à un point de vue distinct (filtre d'un ensemble de classes).

Comme indiqué précédemment, le résultat d'un filtrage peut être formulé sous la forme de trois ensembles d'instances de la base du filtre. L'extension de la base est donc partitionnée en instances sûres, incomplètes et impossibles.

Au regard de la requête, seules les instances sûres et, éventuellement incomplètes, sont pertinentes. La décision d'inclure ou non les instances incomplètes à la réponse peut être paramétrée. Les instances impossibles répondent, quant à elles, à la négation de la requête.

Cette utilisation des filtres pour l'expression de requêtes suggère que le nombre d'instances répondant à la requête (*i.e.* retenues par le filtre correspondant(s)) puisse être limité. Cette idée rejoint le concept de contrainte de cardinalité et se plie à la même sémantique de fonctionnement.

[2] Dans SHIRKA, la cardinalité n'est pas un intervalle mais une valeur fixe.

Alors que jusqu'ici TROPES ne proposait que la visualisation de la définition ou de l'extension des diverses entités de représentation (concept, point de vue, classe, instance, attribut), la formulation de requête étend les possibilités d'interrogation du contenu d'une base de connaissances en permettant une sélection.

Il est possible d'établir un langage pour l'expression de requêtes complexes à base d'opérateurs unaires ou binaires (union, intersection, différence, complément...) en les traduisant en filtres à partir d'opérateurs ensemblistes et logiques.

Soient $r_1 : f_1 = (b_1, c_1), r2 : f_2 = (b_2, c_2)$, deux requêtes et leur filtres associés représentés par l'ensemble des instances sûres (et éventuellement incomplètes) calculé par le filtrage.

- l'union de deux requêtes est équivalente à deux requêtes formulées séparément, donc à deux filtres distincts :
  $r_1 \cup r_2 : f_1 \cup f_2 = (b_1, c_1) \cup (b_2, c_2)$;
- l'intersection de deux requêtes consiste à trouver les instances qui vérifient les deux requêtes simultanément. Il doit donc exister une relation ensembliste d'inclusion non stricte entre les bases des deux requêtes. Il suffit de prendre la plus petite base et de transformer les deux filtres en un seul contenant les conditions des deux :
  $r_1 \cap r_2 : f_1 \cap f_2 = (b_1, c_1) \cap (b_2, c_2) = (b_1 \cap b_2, c1 \wedge c_2)$;
- la différence de deux requêtes consiste à trouver parmi les instances qui vérifient la première requête celle qui ne vérifie pas la seconde :
  $r_1 - r_2 : f_1 - f_2 = (b_1, c_1) - (b_2, c_2)$;
- le complément ou négation d'une requête est formé à partir de l'ensemble des instances impossibles du filtre associé :
  $r : f = (b, c), \neg r : \neg f$ où $\neg f$ correspond à l'ensemble des instances impossibles calculé par le filtrage.

13.3 Représentation des filtres en Tropes

Les filtres TROPES ne sont pas l'équivalent des entités de représentation qui leur servent de base. Ainsi un filtre de concept n'est pas un concept et aucune instance n'est instance de filtre au sens du lien ontologique entre une instance et un concept. Une instance ne peut qu'appartenir à un ensemble d'instances résultat du filtrage. De même, un filtre de classe n'est pas une classe et n'est donc pas inséré dans la hiérarchie de classes. Une instance n'est pas rattachée par le lien d'appartenance à un filtre comme elle l'est à une classe. Ces distinctions sont importantes. Si le rôle d'un filtre de concept ou de classe ne se limite pas à une sélection parmi un ensemble d'instances mais bien à une description d'un ensemble d'instances qu'il convient de distinguer en général, alors c'est que ce filtre doit exister en tant que concept ou classe. Seule la définition d'un filtre reprend les principes de définition de sa base.

13.3.1 Implémentation actuelle

À ce jour, les filtres de TROPES sont essentiellement des filtres de classe et ont deux usages possibles : en tant que descripteurs de type (ils décrivent donc le type d'une sous-classe virtuelle de leur base) ou en tant que requêtes (la description du filtre est utilisée pour filtrer l'extension de la base).

Un filtre est un objet de représentation TROPES à part entière que l'on peut créer, supprimer, modifier, interroger, mais qui n'a pas les prérogatives et le statut d'une classe.

Il existe deux manières de décrire un filtre de classe en TROPES :

- soit à partir d'un fichier de description d'une base qui est parcouru par un analyseur lexical et un analyseur syntaxique afin que soient créées les entités correspondantes ;
- soit à partir de fonctions de création, de modification et de suppression, fournies par l'API de

TROPES. Dans ce cas, les fonctions de définition des attributs des filtres sont les mêmes que les fonctions équivalentes de définition des attributs associées aux concepts et aux classes.

Il peut exister plusieurs niveaux d'imbrication de filtres qui induisent un déclenchement en cascade des filtrages. Des filtres sont imbriqués lorsque l'expression d'un filtre contient un autre filtre.

13.3.2 Représentations possibles pour les autres types de filtres

Nous nous intéressons ici à la représentation des autres types de filtres et proposons un moyen de les définir.

13.3.2.1 Filtres de concept

La description d'un filtre de concept peut se faire sur le même principe que la représentation d'un filtre de classe en utilisant les fonctions de définition de concept du modèle pour, notamment spécifier les contraintes du filtre[3]. Ainsi, un filtre de concept se définit comme un concept mais n'en n'est pas un puisque, par exemple, la clef n'est pas requise.

Un filtre de concept n'est pas équivalent à un filtre sur une classe-racine. Notamment, parce qu'il permet d'exprimer une contrainte (condition) sur des attributs du concept, donc non forcément du même point de vue.

13.3.2.2 Filtres d'ensemble de classes

La définition d'un filtre d'ensemble de classes doit comporter deux parties distinctes :

- les définitions des classes de l'ensemble. Pour chacune d'elles, les restrictions d'attributs et les contraintes de classe sont déclarées en utilisant les fonctions de création de classes de TROPES.
- les définitions des contraintes entre des attributs de ces différentes classes. Ces contraintes sont les propriétés du filtre. Elles sont à distinguer des contraintes propres à chacune des classes de l'ensemble. Elles peuvent être considérées seulement dans le contexte de ce filtre, puisqu'elles lient des attributs de classe distinctes non forcément reliées par des relations.

Un filtre de concept n'est pas non plus équivalent à un filtre sur l'ensemble des classes-racines de chaque point de vue car les conditions peuvent porter sur des attributs qui ne sont pas encore définis au niveau des classes-racines.

13.4 Les filtres comme arguments de contraintes

Dans la perspective d'utiliser les filtres comme moyens d'inférence, plutôt que d'introduire une facette supplémentaire comme décrit plus haut, nous proposons de faire d'un filtre un argument d'une contrainte.

Cette possibilité étend aussitôt les modalités d'emploi du filtre qui peut désormais servir à calculer la valeur d'un attribut (comme argument d'une contrainte `mic-eq` ou `mic-assign`, par exemple) ou simplement à vérifier sa validité (comme argument d'une contrainte de comparaison, par exemple). Ainsi, la valeur d'un attribut dans un objet peut prendre pour valeur, ou faire intervenir dans son calcul, le résultat du filtrage de l'extension d'un concept, d'une classe ou d'un ensemble de classes.

Par exemple, si dans le concept PERSONNE muni de l'attribut *a-pour-pere*, on souhaite obtenir l'ensemble des petits-fils (*cf.* exemple de filtre SHIRKA, section 2.1.2.3), il suffit de contraindre l'attribut *petits-fils* par la contrainte :

(`mic-assign` *petits-fils filtre(*PERSONNE, {*a-pour-père.a-pour-père = self*}*)*)

[3] Les contraintes sont essentiellement ce qui distingue un filtre de concept de sa base.

où *self* est une valeur qui fait référence à l'objet d'attache du filtre (et joue le rôle des facettes `var ->` et `var <-` de Shirka).

Le filtre est un objet Tropes. Si on désire en faire un argument d'une contrainte, le filtre doit alors être considéré comme une variable multivaluée – son résultat est un ensemble d'instances – dont les éléments vérifient le type d'un concept, une classe ou un ensemble de classes d'un concept.

Le filtre n'a pas de représentation en tant qu'objet d'une base de connaissances Tropes. Il est géré par Tropes comme un objet privé ayant sa propre représentation. Or, faire d'un filtre un argument de contrainte nécessite de le lier à un AIC, non pas comme ACT, car il ne correspond pas à un attribut de Tropes, mais en tant que variable contrainte spéciale. On ne peut pas considérer non plus qu'un filtre soit un argument constant car la valeur du filtre est fortement dépendante des modifications de sa base.

Afin que toute modification d'un filtre soit propagée dans le réseau de contraintes dans lequel il est impliqué, il faut que le filtre possède la liste des AIC (donc des contraintes) auxquels il est lié.

Lorsqu'un filtre est impliqué dans une contrainte, c'est en fait l'extension de ce filtre qui est contrainte. Chaque modification de l'intension ou de l'extension du filtre doit être répercutée dans le réseau en réactivant les contraintes dont le filtre est argument.

L'emploi de filtre dans des contraintes est sujet à quelques précautions.

Tout d'abord le filtre ne peut être employé comme partie gauche d'une affectation ou d'une contrainte d'égalité. En effet, puisque c'est l'extension du filtre qui est sollicitée et que celle-ci est évaluable dynamiquement et dépend du contenu de la base, elle ne peut être évaluée que par filtrage de la base du filtre[4]. L'idée est bien d'utiliser le filtre pour contrôler la validité ou inférer des connaissances, et non pas de forcer l'évaluation de leur extension ; le filtrage étant le seul mécanisme pour déterminer cette extension.

Enfin, en tant que variable multivaluée, le filtre possède une cardinalité. Cette cardinalité est variable et dépend du contenu de la base du filtre. Elle intervient dans les tests de maintenance des contraintes où est impliqué le filtre. D'autre part, lorsque l'extension d'un filtre se réduit à un singleton, il est possible de transformer cet ensemble-valeur en une valeur par la fonction contrainte `mic-set-to-value`. Le résultat de l'application de cette fonction est alors une valeur, on a transformé une variable multivaluée dont la valeur est un singleton en une variable monovaluée dont la valeur est l'élément du singleton.

13.5 Mise en œuvre du filtrage

Lorsqu'un filtre est utilisé comme moyen d'obtention d'une valeur d'attribut ou lorsqu'il est associé à une requête, un mécanisme de filtrage sous-jacent permet d'obtenir les trois ensembles d'instances décrits plus haut. Lorsque le filtre est employé dans la description d'un type, le filtrage n'est pas nécessaire : seul le type du filtre géré par Metéo (et éventuellement calculé par Micro) intervient dans la vérification des valeurs proposées.

Cette section décrit comment le typage systématique des filtres effectué par Metéo est utilisé par le mécanisme de filtrage pour réaliser un pré-filtrage de la base du filtre.

13.5.1 Typage d'un filtre

À chaque filtre créé est associé un type géré par Metéo.

- Le type d'un filtre de concept est construit depuis la description du filtre. Le sur-type du type du filtre est le type du concept. Un (sous)-type intermédiaire est obtenu en substituant, dans le record de types du concept, les types des attributs redéfinis par les nouveaux types issus des affinements imposés par le filtre. Enfin, le type final du filtre est calculé en activant les

[4] Dans une extension, on pourrait imaginer que le résultat connu d'un filtre engendre la création d'instances manquantes.

contraintes du filtre. Comme pour le calcul du type d'un concept en présence de contraintes, MICRO est chargé de la propagation des contraintes et de fournir les nouveaux types à METÉO. Le record de type du filtre est un sous-type du record de types du concept. Il est donc situé sous celui-ci dans la hiérarchie du type du concept.
- Le type d'un filtre pour une classe est obtenu selon les mêmes étapes d'élaboration. Le record de type du filtre est un sous-type du record de types de la classe. Il est donc situé sous celui-ci dans la hiérarchie du type de la classe.
- Le type d'un filtre d'ensemble de classes est obtenu en trois étapes également, mais par des opérations différentes. Tout d'abord, il est procédé à l'intersection des types de chacune des classes afin d'obtenir le type initial du filtre. Le type intermédiaire est obtenu en remplaçant dans le type initial les attributs redéfinis par le filtre, s'il s'avère que la redéfinition produit un sous-type du type initial (calculé après l'intersection de la première étape). Enfin, le type final du filtre est obtenu en propageant les contraintes. Le record de types du filtre est un sous-type du record de types du concept. Il est donc situé sous celui-ci dans la hiérarchie du type du concept.

Si lors du calcul du type d'un filtre, un domaine est vidé (lors de la propagation ou lors de l'intersection réalisée pour un filtre d'ensemble de classes), le type du filtre ne peut être établi. Le filtre étant inopérant, sa création est refusée.

Si dans une base de connaissances, les filtres ne sont pas représentés comme le sont les concepts ou les classes, on peut remarquer que METÉO, en revanche, représente le type d'un filtre, c'est-à-dire l'inclut dans la hiérarchie de type correspondante.

Cette représentation est persistante lorsque le filtre est utilisé dans la description du type d'un attribut, car le filtre est alors assimilé à un type.

En revanche, cette représentation peut être provisoire pour les deux autres utilisations du filtre qui reposent sur un mécanisme de filtrage. Pour l'obtention de la valeur d'un attribut comme pour la réponse à une requête, le type du filtre peut être construit, inséré dans la hiérarchie de types correspondante, utilisé pour le filtrage, et supprimé lorsque le filtrage est terminé. Le principe du filtrage est à présent détaillé.

13.5.2 Le mécanisme de filtrage

Le mécanisme du filtrage est assez intuitif : chaque instance de la base du filtre doit être passée au tamis des conditions du filtre.

Selon ce principe, dans le cas d'un filtre de concept, toutes les instances du concept sont examinées ; dans le cas d'un filtre de classe, toutes les instances de la classe sont examinées ; dans le cas d'un filtre d'un ensemble de classes, toutes les instances appartenant à la fois à chacune des classes sont examinées.

Dans TROPES, un *pré-filtrage* a été conçu en collaboration avec Cécile Capponi, l'auteur de METÉO. Cet algorithme est décrit dans son mémoire de thèse [Capponi95]. Nous en donnons ici le principe.

Il s'agit de mettre à profit la maintenance d'une hiérarchie de types des diverses entités de représentation afin de limiter le nombre de tests de satisfaction du filtre. Afin de limiter ce nombre, il faut écarter du test les instances appartenant à la base dont on peut dire avec certitude qu'elles ne satisfont pas le filtre (elles peuvent donc être placées directement dans l'ensemble des instances impossibles pour le filtre). Or, cette estimation peut être faite en comparant le type du filtre avec le type de l'instance qui est le type obtenu après intersection des types des classes d'appartenance de l'instance (qui peut donc également être le type commun à plusieurs instances de la base de connaissances). Si l'intersection deux à deux des types d'attributs qui constituent respectivement le record de types du filtre et le record de type de l'instance est vide, alors l'instance peut être écartée sans être testée, d'où le nom de pré-filtrage (*cf.* figure 13.1).

En effet, si les types du filtre et le type de l'instance sont disjoints, cela signifie que l'instance, pour au moins un attribut, ne peut recevoir aucune valeur qui soit acceptée par le filtre. Sans la constatation de cette disjonction, tant que cette valeur n'est pas connue, l'instance peut demeurer dans l'ensemble des instances incomplètes alors que sa place est dans l'ensemble des instances impossibles. Le pré-filtrage écarte donc les instances intensionnellement incompatibles avec le filtre. Celles qui ne seront pas écartées ont un type compatible avec celui du filtre : pour chaque attribut, l'instance peut recevoir une valeur acceptée par le filtre. Le principe de l'algorithme de pré-filtrage

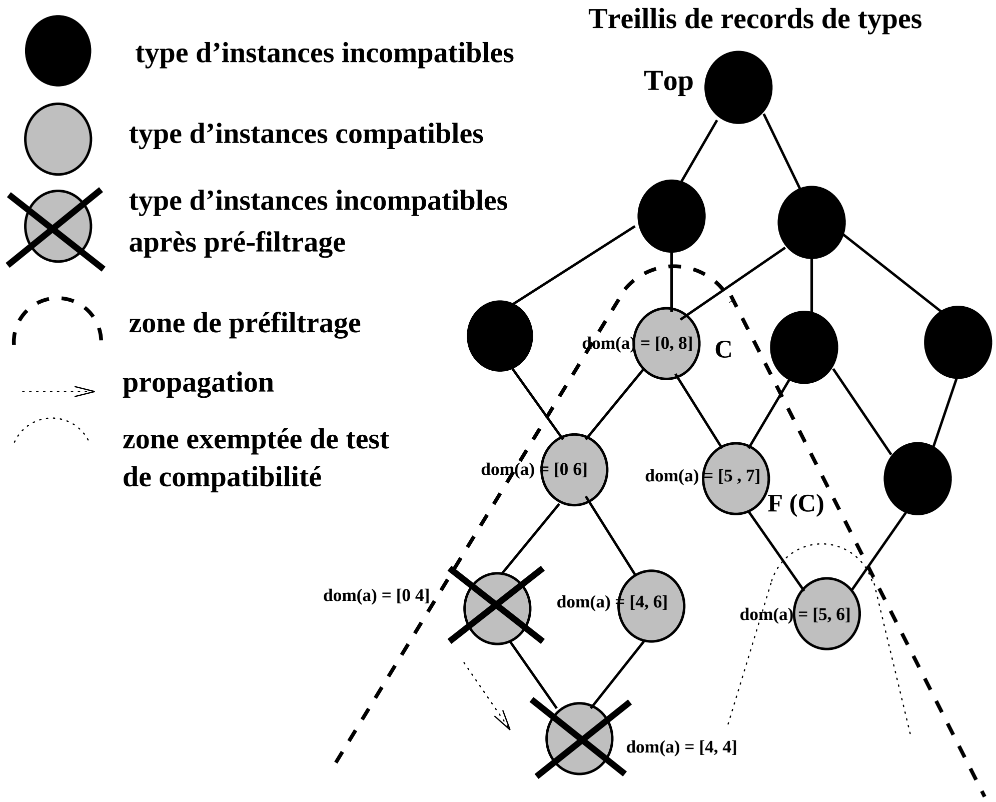


FIG. 13.1 - : Le résultat d'un pré-filtrage des types d'instances. L'appariement se fait selon le seul attribut a. Les types situés au dessus du type de la base du filtre sont écartés. En dessous du type de la base du filtre, on procède aux intersections avec le type du filtre. Dès qu'une incompatibilité est détectée, elle est propagée vers les sous-types du treillis.

est donc de déterminer, parmi les sous-types de celui de la base du filtre, ceux qui ne sont pas disjoints au type du filtre et correspondent à des types d'instances.

Tout d'abord le type du filtre est calculé et placé dans la hiérarchie de types (sous celui de la base). Lorsque des contraintes interviennent dans le filtre, le type du filtre résulte d'une propagation de contraintes et nécessite donc un échange entre METÉO et MICRO.

Une fois le type du filtre calculé, une zone de pré-filtrage est définie : elle contient tous les types situés en dessous du type de la base. Les sous-types du type du filtre sont jugés compatibles sans autre forme de procès. Les autres types dans la hiérarchie ne sont pas pertinents pour le filtre.

La sous-hiérarchie définie par la zone de pré-filtrage est alors parcourue, du haut vers le bas, depuis le type de la base. À chaque type rencontré, une intersection avec le type du filtre est calculée. Si elle s'avère vide, alors le type courant est incompatible (disjoint avec le type du filtre). Par conséquent, tous les sous-types de ce type sont également incompatibles, sans même qu'il soit besoin de procéder à l'intersection. On propage donc jusqu'aux sous-types feuilles une incompatibilité décelée à un niveau donné.

Ensuite, le filtrage peut s'opérer. Les instances dont le type a été jugé compatible sont récupérées dans la base de connaissances par METÉO.

Hormis pour les filtres de concept qui s'appliquent sur l'extension du concept, indépendamment des rattachements d'instances dans les points de vue, si des instances ont un type compatible avec celui du filtre, rien ne prouve qu'elles appartiennent en effet à l'extension de sa base. Il faut donc tester l'appartenance de l'instance à la base du filtre.

Une fois que l'instance est déterminée comme pouvant subir le test du filtre, son contenu est confronté aux conditions du filtre selon le principe d'une vérification de type : un test d'appariement est réalisé entre chacune des valeurs de l'instance et la description (type de chacun des attributs) du filtre. Si ces appariement réussissent, la partie statique de l'intension du filtre est satisfaite, sinon l'instance est placée dans l'ensemble des instances impossibles (d'après leur contenu et non plus leur type). Le travail de METÉO est terminé. La main est alors passée à MICRO afin de tester la partie dynamique, c'est-à-dire vérifier si le contenu de l'instance satisfait les contraintes du filtre.

Si cette dernière étape est franchie avec succès, l'instance est placée dans l'ensemble des instances sûres seulement si son contenu est complet (tous les attributs du filtre sont valués). Si une contrainte au moins n'est pas satisfaite par le contenu de l'instance, cette dernière est placée dans l'ensemble des instances impossibles. Sinon, l'instance est incomplète mais ne viole aucune des contraintes du filtre : elle est considérée comme inconnue vis-à-vis du résultat du filtre.

Le pré-filtrage s'avère payant dans la mesure où le rapport entre le nombre d'instances sûres ou incomplètes 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.

13.6 Filtrage et cohérence

Introduire des filtres dans TROPES et les utiliser à diverses fins nécessite de s'intéresser également au maintien de la cohérence de la connaissance contrôlée ou produite par eux. Le maintien de la cohérence est étudié ici dans deux situations distinctes : la modification de la base du filtre et la modification de la description du filtre lui-même.

13.6.1 Modification de la base d'un filtre

Par modification de la base d'un filtre, il est entendu ici tout ajout, retrait ou modification d'une instance de la base d'un filtre. Chacune de ces modifications est susceptible d'avoir un effet sur les filtres inférentiels, c'est-à-dire ceux utilisés pour formuler des requêtes ou apparaissant dans des contraintes destinées à obtenir la valeur d'un attribut. Seuls ces filtres, en effet, appliquent un filtrage (et non une simple vérification de type comme le font les filtres employés comme descripteurs de type d'attribut) sur un ensemble d'instances. Aussi, toute modification de cet ensemble est susceptible de ne plus être cohérente avec le résultat du précédent filtrage opéré par le filtre.

Deux solutions peuvent être mises en place pour garantir la cohérence d'une base de connaissances en présence de filtres :

- une solution procédant au recalcul systématique du filtre à chaque interrogation. Cette solution dispense d'une gestion de liens de dépendances entre base et attribut filtré, mais s'avère peu efficace dans une base de connaissances peu modifiée ;
- une solution de type *caching* où l'on conserve le résultat de l'application du filtrage (dans l'objet représentant le filtre, par exemple). Tant que l'extension de la base du filtre n'est pas modifiée, il n'y a pas lieu de réactiver le filtre. Si une modification se produit dans l'extension de la base alors on propage une information vers le filtre signifiant qu'il doit être réactivé. Aussi, cette solution requiert que tout concept ou classe connaisse l'ensemble des filtres inférentiels dans lesquels elle est impliquée. Une gestion de liens de dépendances entre la base et le filtre de type TMS doit alors être mise en place. Cette solution est avantageuse

pour les bases de connaissances peu modifiées (simplement consultées).

Chacune de ces solutions veillant au maintien de la cohérence des résultats de filtres est préférable à l'autre dans une utilisation particulière d'une base de connaissances.

Pour une solution de type *caching*, la modification de la base conduit à la réactivation systématique de tous les filtres. Or, il existe un moyen de sélectionner seulement les filtres dont la réactivation est nécessaire. Pour cela, il suffit de tester, pour chaque filtre de la base, si le type de l'instance ajoutée, supprimée ou modifiée est compatible avec lui. Si l'instance n'a pas un type compatible avec celui du filtre (l'intersection des types est vide) alors elle n'en modifie pas le résultat, on peut donc s'épargner la réactivation de ce filtre.

Réactiver les filtres concernés par la modification de la base d'un filtre n'est malheureusement pas suffisant si l'on considère les accès. En effet, la modification d'un attribut étape d'un accès ayant sa source dans la base du filtre est également susceptible de modifier le résultat du filtrage. Or, cet attribut étape ne figure pas forcément dans la base du filtre. Autrement dit, la base peut être inchangée mais un des accès qui s'en échappe peut être modifié. Si cet accès est utilisé dans les conditions d'un filtre posé sur la base, il faut réactiver le filtre. Dans une solution de type *caching*, cela impose de propager la modification de l'accès vers sa source, donc vers la base.

13.6.2 Modification dans la description d'un filtre

Un filtre est composé de deux parties : une base et un ensemble de conditions constitué de descriptions d'attributs de la base et/ou de contraintes. Modifier la description d'un filtre revient à en changer soit la base, soit la définition d'un des attributs affinés, soit une contrainte.

La modification d'un filtre utilisé en tant que descripteur du type d'un attribut doit entraîner un déclenchement de la vérification de type des valeurs précédemment validées. La modification d'un filtre inférentiel doit conduire à sa réactivation, c'est-à-dire au nouveau filtrage de sa (éventuellement nouvelle) base.

Pour le module de gestion des types, la modification d'un filtre entraîne la suppression du type de l'ancien filtre, si celui-ci est conservé, et le positionnement du nouveau type du filtre dans la hiérarchie de types correspondante.

13.6.3 Modification d'un filtre imbriqué

Un filtre peut être imbriqué dans un autre filtre. Aussi, toute modification sur l'intension ou l'extension de ce filtre doit être propagée au(x) filtre(s) englobant(s).

La réactivation d'un filtre imbriqué doit donc déclencher la réactivation en cascade des filtres englobants auxquels il doit être lié dans une solution de type *caching*.

Une solution de type *caching* peut paraître lourde pour maintenir la cohérence des filtres dans une base de connaissances. En effet, elle impose une gestion de liens de dépendances entre les filtres et les entités qui leur servent de base. Elle nécessite également une propagation lors de la modification d'un accès prenant sa source dans la base du filtre afin de signaler qu'une réactivation est possible. De même, un réseau de dépendances doit être mis en place entre les filtres imbriqués. Enfin, le *caching* est pénalisant lorsque de nombreuses modifications ont lieu sur la base de filtre. En contre partie, les filtres à réactiver peuvent être sélectionnés, ce qui n'est pas le cas dans la solution sans *caching* où les filtres sont systématiquement réactivés.

13.7 Comparaison avec d'autres SRPO avec filtres

La notion de filtre en TROPES recouvre les divers usages rencontrés dans les SRPO intégrant des filtres :

- Les filtres dans TROPES sont un moyen de formuler des requêtes. La formulation de ces filtres ne s'accompagne pas de critères qualitatifs permettant de juger de la ressemblance comme dans KRL [Bobrow et al.77], mais les contraintes de MICRO permettent d'obtenir un langage de description de filtres assez puissant comparable à celui de MERING [Ferber84].
- Les filtres sont un moyen d'obtention de la valeur d'un attribut comme dans SHIRKA [Rechenmann et al.90] et FROME [Dekker94] mais le filtre n'est plus introduit à l'aide d'une facette mais à l'aide d'une contrainte.

L'information rendue par le filtrage – trois ensembles d'instances jugées sûres, incomplètes ou impossibles – est semblable au résultat fourni par le filtrage dans YAFOOL [Ducournau88] et permet à l'utilisateur ou au système de juger des instances à conserver. Conserver les instances jugées possibles (donc incomplètes) introduit un degré de liberté dans le filtrage.

À la différence de langages comme FROME ou OBJLOG+ [Faucher91], les filtres ne sont pas insérés dans les hiérarchies de classes ou bien en tant que concepts. Ils correspondent à des objets "cachés", gérés par TROPES, mais ne viennent pas encombrer les entités destinées à la représentation. En clair, les filtres sont distingués des concepts et des classes, même si leurs définitions sont semblables, car ils n'ont pas la même finalité : le concept ou la classe décrivent, le filtre distingue. L'insertion du type du filtre dans les treillis de types records réalisée par METÉO rejoint le principe de positionnement du filtre dans une hiérarchie de classes qu'adoptent ces deux langages mais exploite davantage, pendant le filtrage, la spécificité de chacune des instances de l'ensemble d'instances à filtrer.

Les filtres de TROPES remplissent au moins les mêmes fonctions que les filtres de SHIRKA, mais leur description est rendue plus aisée par les contraintes ; la cohérence est également renforcée.

Enfin, l'utilisation des filtres dans la description du type des attributs est une utilisation propre à TROPES, naturelle dans un SRPO fortement typé comme TROPES. Dans ce mode de fonctionnement, le filtre n'est activé que sur une seule instance : la valeur proposée pour l'attribut.

13.8 Conclusion

En considérant les divers ensembles d'instances définis et identifiables dans une base de connaissances, nous avons été amenés à proposer trois types de filtre : les filtres de concept, de classe et d'un ensemble de classes. L'ensemble d'instances sur lequel porte un filtre est défini comme la base de filtre. Pour chacun de ces types de filtre, la description se fait de manière identique à la description de la base.

La présence de contraintes dans TROPES nous conduit naturellement à étendre la description des filtres par des contraintes. Ces contraintes interviennent à la fois dans la partie statique (elles interviennent dans le calcul du type du filtre) et dynamique (le résultat du filtrage repose sur leur satisfaction) de l'intension du filtre. Lors de l'appariement entre le contenu d'une instance et la description du filtre, les contraintes ont, comme lors de la classification, un rôle qui se limite à celui d'un prédicat, la propagation de contraintes pouvant toutefois être mise à contribution.

Le résultat du filtrage peut être fourni sous la forme de trois ensembles : l'ensemble des instances sûres qui satisfont avec certitude le filtre, l'ensemble des instances inconnues dont le contenu incomplet satisfait le filtre, l'ensemble des instances impossibles dont le contenu ne satisfait pas le filtre.

Il peut être fait des filtres trois utilisations : description de type d'un attribut, requête ou moyen d'obtention de la valeur d'un attribut. Dans ce dernier cas, le filtre est argument d'une contrainte.

Nous avons proposé un pré-filtrage de l'ensemble d'instances qui tire profit de la présence du module de gestion de types MÉTÉO et peut considérablement accélérer le mécanisme. Le principe est de comparer la compatibilité du type de l'instance avec le type du filtre par une opération d'intersection.

Nous avons également étudié quels sont les tenants d'un maintien de cohérence sur les connaissances contrôlées par les filtres. Nous avons décrit les actions opérées sur la base de connaissances qui nécessitent la réactivation des filtres. Une solution à base de *caching* s'avère sans doute efficace pour les bases de connaissances en lecture seulement, mais coûteuse et complexe (stockage et maintenance des liens de dépendance) à mettre en œuvre ; on peut lui préférer la réactivation systématique des filtres.

Enfin, il faut signaler que les filtres et les relations peuvent être étroitement liés comme l'a montré la section 12.4. En effet, les filtres peuvent être mis à contribution pour construire une nouvelle relation à partir de relations existantes. Ainsi, des opérateurs de l'algèbre relationnelle comme la division, le complément, la projection et l'anti-projection ont été décrits en termes de filtre. Aussi, les deux extensions présentées ici se montrent finalement liées l'une à l'autre par l'intermédiaire des contraintes.

Avec cette proposition se termine la présentation des extensions de TROPES envisageables grâce à l'intégration de contraintes.

Chapitre 14

Conclusion et perspectives

Pour terminer ce mémoire, nous faisons ici le bilan de l'étude réalisée. Nous résumons les apports de ce travail et dégageons un certain nombre de perspectives de recherche pour la poursuite de l'intégration de contraintes réalisée dans TROPES.

14.1 Conclusion

C'est en constatant le manque d'adaptation des moyens de représentation du modèle TROPES pour exprimer et maintenir des relations numériques, voire symboliques, entre les attributs de ses objets, que nous avons choisi d'avoir recours à la programmation par contraintes afin d'intégrer les principes de ce paradigme dans TROPES.

En déterminant les besoins en contraintes du modèle TROPES, nous avons élaboré un cahier des charges pour un module de programmation par contraintes chargé de gérer les CSP TROPES. Nous avons également introduit un moyen de désigner les attributs contraints en étendant la notion classique de *chemin* aux attributs multivalués, à travers la notion d'*accès*. La cohérence des connaissances représentées a mis l'accent sur les liens étroits que devaient entretenir types et contraintes au sein du modèle. Aussi, nous avons décrit comment les contraintes interviennent dans le calcul du type des entités. De même, le souci d'uniformiser la manipulation des entités de représentation, nous a conduit à exiger que les CSP TROPES soient dynamiques, c'est-à-dire qu'à tout moment une contrainte puisse être ajoutée ou retirée, ou que le domaine d'une variable contrainte (attribut) puisse évoluer.

Ces exigences ont été prises en compte dans la conception du module de programmation par contraintes MICRO. Ce module est chargé du traitement de CSP numériques à intervalles, de CSP booléens et de CSP ensemblistes ou sur des listes.

Son originalité réside dans :

- sa gestion simultanée de domaines numériques finis ou infinis représentés sous forme d'unions d'intervalles ;
- sa gestion des domaines infinis tant au niveau de la maintenance que de la résolution ;
- sa capacité à réagir lors de l'ajout ou du retrait d'une contrainte ou de la modification du domaine d'une variable contrainte ;
- sa gestion de contraintes sur des ensembles et des listes issues des opérateurs du langage LISP ;
- des méta-contraintes permettant de poser des contraintes sur des ensembles ou des listes de variables contraintes, ou de poser der contraintes non déterministes.

On peut reprocher à MICRO d'avoir choisi une gestion des CSP numériques à intervalles qui est coûteuse en place mémoire, et en temps de calcul alloué aux règles de maintien de consistance puisque MICRO maintient des unions d'intervalles. Mais il faut remarquer qu'en contrepartie, le niveau de consistance est supérieur à celui des modules travaillant sur des intervalles uniques. MICRO privilégie la consistance à l'efficacité.

La même critique peut être formulée à l'égard de la gestion des CSP dynamiques qui s'avère également coûteuse en place mémoire en raison du stockage des contextes de pose, même si, nous l'avons signalé, des optimisations sont toujours possibles. En présence de domaines infinis, il semble que cela soit le prix à payer pour que le module n'ait pas à reconstruire tout le CSP lors d'une modification.

Le fait d'avoir construit notre propre module de programmation par contraintes nous a permis de réaliser un couplage *semi-faible* entre TROPES et MICRO, plus économe et plus rapide qu'un couplage faible qui entraînerait la duplication des informations.

La présence de contraintes dans TROPES augmente sa déclarativité en permettant aux utilisateurs d'exprimer des relations entre des attributs d'objets. En outre, par leur mécanisme de maintenance sous-jacent, les contraintes permettent de garantir un certain niveau de cohérence dans les domaines des attributs contraints, en accord avec les relations exprimées. Enfin, les contraintes constituent un nouveau moyen d'inférence pour TROPES qui permet, dans leurs limites, de suppléer les méthodes. Les contraintes, par leur caractère multidirectionnel, sont alors un moyen déclaratif de factoriser en un seul énoncé, l'expression de plusieurs méthodes.

Cependant, la présence de contraintes dans une base de connaissances TROPES n'est pas sans conséquence sur les autres mécanismes d'exploitation du modèle et sur la cohérence même de ces bases. En ce qui concerne le premier point, nous avons pris soin de détailler comment les contraintes sont prises en compte par les autres mécanismes. Quant à la cohérence de la base, elle est désormais statiquement liée au niveau de consistance locale maintenu par MICRO sur les CSP TROPES.

De plus, nous avons montré que TROPES pouvait tirer parti des contraintes dans le contrôle du partage de valeurs entre attributs. L'idée est que TROPES propose un ensemble de primitives pour exprimer ce partage, auxquelles sont associées, de façon sous-jacente, des contraintes.

C'est cette même utilisation des contraintes que nous avons suggérée dans les deux propositions qui terminent cette étude. Nous avons montré que l'expression de relations (dans le sens général du terme) et de filtres peut se faire à l'aide des contraintes. Non seulement les contraintes sont garantes de la sémantique associée à une relation, mais elles peuvent, de plus, être employées dans la construction de relations en en faisant l'équivalent d'une algèbre relationnelle. Quant aux filtres, après avoir proposé un pré-filtrage tirant parti de la présence du module de types METÉO dans TROPES, nous en avons étudié les divers usages possibles et montré que leur expressivité pouvait être accrue par les contraintes.

14.2 Perspectives

Au chapitre des perspectives concernant cette intégration de contraintes dans TROPES, certaines voies sont à achever alors que d'autres nous paraissent intéressantes à explorer. Nous détaillons ici ces travaux à court ou à long terme.

La première version de couplage TROPES/MICRO concernait la version 0 de TROPES écrite en langage LE-LISP V16. Par rapport aux descriptions faites ici, la version du module MICRO ne disposait pas encore de l'algorithme de résolution proposé, ni de la gestion des méta-contraintes non-déterministes. En revanche, les primitives de dynamicité et les diverses contraintes décrites étaient opérationnelles. De même, l'interface ne gérait que des accès immédiats (de longueur 1).

Depuis peu, la version 1.0 de TROPES est disponible. Elle nécessite une traduction de MICRO dans le langage ILOG TALK. Il nous reste donc à mettre en place le couplage avec cette nouvelle version, à inclure dans MICRO l'algorithme de résolution proposé et à munir l'interface des fonctions de gestion des accès que nous avons décrites.

Nous avons vu que les contraintes permettent d'étendre les possibilités de maintien systématique de la cohérence des relations qu'elles permettent d'exprimer. Cependant, si le niveau de vérification est augmenté, il n'est pas complet. Autrement dit, il se peut, non seulement que des valeurs figurent dans des domaines d'attributs contraints alors qu'elles devraient en être écartées (ce qui révèle un faible degré de consistance locale) mais aussi, ce qui est plus gênant, que des contraintes soient acceptées alors qu'elles entraînent une incohérence que Micro n'a pu détecter. Ces limites de la consistance locale sont bien connues des spécialistes de la programmation par contraintes, les utilisateurs de Tropes doivent l'apprendre aujourd'hui à leurs dépens.

Aussi, nous nous proposons à plus long terme d'étendre les capacités du module Micro sur deux points, afin de renforcer la cohérence des bases de connaissances dans Tropes.

1. Le calcul des extensions des intervalles qui résultent des expressions contraintes peut être renforcé par des techniques de l'analyse numérique, comme nous l'avons vu. Cependant, il faut signaler que cette amélioration ne vaut que pour les expressions arithmétiques de contraintes non linéaires, et particulièrement pour des expressions polynomiales. Le gain attendu est un niveau de consistance locale plus fort et donc un temps de résolution plus court.
2. La détection des incompatibilités et des redondances dans les CSP gérés par Micro peut se faire de deux manières, comme nous l'avons vu :
   - soit en laissant le soin à l'utilisateur de Tropes ou de Micro de fournir des informations sur l'incompatibilité et la redondance de certaines contraintes. Cette solution est partielle, déclarative et peu coûteuse ;
   - soit en couplant Micro avec un système de ré-écriture [Dershowitz et al.89] – existant ou à concevoir. Cette solution “boîte noire” est coûteuse mais sans doute plus complète.

   Ces solutions permettraient de simplifier des systèmes d'équations ou d'inéquations (donc, de contraintes) et de détecter les incohérences dans les CSP. Le gain attendu ici se situe au niveau de la place mémoire, de par la simplification des CSP, mais surtout au niveau de la détection de l'incohérence des CSP construits. Les répercussions au niveau de Tropes seraient considérables : non seulement les redondances entre contraintes seraient supprimées, mais les incohérences seraient détectées au plus tôt, augmentant du coup le niveau de fiabilité dans la cohérence d'une base de connaissances en présence de contraintes.

Du côté de Tropes, certaines contraintes envisagées, comme les contraintes de priorité, d'évolution ou les contraintes globales, semblent présenter un intérêt certain et méritent d'être intégrées à l'ensemble des contraintes disponibles. Par ailleurs, nous avons volontairement écarté ici les contraintes de contrôle (contraintes dont les arguments sont des entités de représentation comme les concepts, les classes, les points de vue...). Il y a là aussi matière à s'interroger sur la façon dont les contraintes pourraient finalement contrôler jusqu'à la configuration des bases de connaissances. Concernant la notion d'accès, elle pourrait également être utilisée pour déterminer la valeur d'un attribut et sous-tendrait alors une contrainte d'égalité.

Enfin, et surtout, nous pensons qu'une adaptation (et donc une extension) de Micro à des attributs de Tropes de type quelconque – et donc, autres que ceux pour lesquels Micro a été conçu (entier, réel, booléen, ensemble, liste)[1] –, est possible à travers le couplage de Micro avec Metéo, le module de gestion de types associé à Tropes. Dès lors la définition de contraintes dans Tropes ne se limiterait plus aux attributs dont les types peuvent être contraints, mais serait étendue à n'importe quel attribut.

Cette collaboration renforcée entre Micro et Metéo est suggérée par trois constatations.

1. Tout d'abord, les opérations de normalisation que Micro intègre, par exemple, sur les domaines des variables numériques ou multivaluées, sont identiques à celles opérées par Metéo pour uniformiser l'expression des types. Il y a donc ici possibilité d'un partage de code entre les deux modules.
2. Ensuite, si l'on observe de plus près les règles de maintenance associées aux opérateurs arithmétiques binaires ($+, -, *, / \ldots$) et données en annexe A.1, on s'aperçoit qu'une seule contrainte de Micro, appelons-la `mic-op-bin`, est en mesure de factoriser l'effet de toutes les autres (`mic-add, mic-sub`...) moyennant un paramétrage. En effet, en indiquant le type des

[1] Rappelons que des variables de type quelconque peuvent être arguments de contraintes d'égalité, de différence ou d'affection.

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 MICRO.

3. Enfin, METÉO est un module de gestion de types *extensible* dans lequel 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 à loisir par l'utilisateur. Il y a donc dans METÉO tous les ingrédients nécessaires à MICRO pour fabriquer de nouvelles contraintes sur des variables de types nouveaux.

Des premières recherches que nous avons entreprises dans cette direction, il ressort qu'une classification des opérateurs selon leurs propriétés (monotonicité, parité, périodicité...) permet de factoriser l'écriture des contraintes associées à ces opérateurs. Cette voie mérite, encore plus sans doute que les précédentes, d'être approfondie. Aussi, il nous faut à présent aller vers un couplage MICRO/METÉO qui pourrait déboucher sur un module capable de traiter des CSP de types divers et dont l'ensemble de contraintes serait extensible via la définition de nouveaux types.

Annexe A

Maintenance dans Micro

A.1 Règles de maintenance

Nous donnons ici les règles de maintenance de chacune des contraintes disponibles dans MICRO.

A.1.1 Contraintes sur variables monovaluées

Pour des raisons de lisibilité, les règles sont présentées avec des variables dont le domaine n'est formé que d'un seul intervalle. Pour le cas d'unions d'intervalles, il suffit d'appliquer la distribution décrite dans le chapitre 8.

Soit X une variable monovaluée. Dans la suite $dom(X)$ désigne son domaine (formé d'un seul intervalle).

Soit X une variable multivaluée. Dans la suite, $ext(X)$ désigne son extension, $dom(X)$ désigne le domaine de ses éléments, $sauf(X)$ désigne l'ensemble des valeurs impossibles, $card(X)$ désigne la cardinalité (ou la longueur dans le cas de listes), $required(X)$ désigne l'ensemble des éléments obligatoires, $required-at(X)$ désigne l'ensemble des éléments obligatoires à certains rangs de la liste X.

Dans la suite, on appelle $inf(X)$ (resp. $sup(X)$) la borne inférieure (resp. supérieure) de l'intervalle (domaine) de X. Lorsque $inf(X) = sup(X)$, le domaine de X est de cardinalité 1 et la valeur de X est désignée par $val(X)$.

A.1.1.1 Contraintes principales

Contraintes génériques

- $X = Y$ (`mic-eq X Y`)
  $newdom(X) = dom(X) \cap dom(Y)$
  $newdom(Y) = dom(X) \cap dom(Y)$
- $X \neq Y$ (`mic-neq X Y`)
  si $\mid dom(Y) \mid = 1$ **alors**
    $newdom(X) = dom(X)/val(Y)$
  si $\mid dom(X) \mid = 1$ **alors**
    $newdom(Y) = dom(Y)/val(X)$
- $X \leftarrow Y$ (`mic-assign X Y`)
  $newdom(X) = dom(X) \cap dom(Y)$

Contraintes numériques

- $X < Y$ (`mic-lt X Y`)
  $newdom(X) = [min(inf(X), inf(Y)), min(sup(X), sup(Y)^-)] \cap dom(X)$
  $newdom(Y) = [max(inf(X)^+, inf(Y)), max(sup(X), sup(Y))] \cap dom(Y)$
  où $sup(Y)^-$ et le plus grand entier ou réel inférieur à $sup(Y)$
  et $inf(X)^+$ et le plus grand entier ou réel supérieur à $inf(X)$
- $X \leq Y$ (`mic-le X Y`)
  $newdom(X) = [min(inf(X), inf(Y)), min(sup(X), sup(Y))] \cap dom(X)$
  $newdom(Y) = [max(inf(X), inf(Y)), max(sup(X), sup(Y))] \cap dom(Y)$

- $X > Y$ (`mic-gt X Y`)
  équivalent à (`mic-lt Y X`)
- $X \geq Y$ (`mic-ge X Y`)
  équivalent à (`mic-le Y X`)

A.1.1.2 Contraintes secondaires

Contraintes unaires numériques

- $Y = \mid X \mid$ (`mic-abs X`)
  $newdom(Y) =$
  si $inf(X) < 0$ **alors**
  si $sup(X) > 0$ **alors**
  $[0, max(-inf(X), sup(X))] \cap dom(Y)$
  sinon $newdom(X) = [-sup(X), -inf(X)] \cap dom(Y)$
  finsi
  sinon $dom(X) \cap dom(Y)$
  finsi

  $newdom(X) =$
  si $inf(Y) = 0$ **alors**
  $[-sup(Y), sup(Y)] \cap dom(X)$
  sinon
  $([-sup(Y), -inf(Y)]$
  $\cup$
  $[sup(Y)), inf(Y)]) \cap dom(X)$
  finsi
- $Y = -X$ (`mic-neg X`)
  $newdom(Y) = [-sup(X), -inf(X)] \cap dom(Y)$
  $newdom(X) = [-sup(Y), -inf(Y)] \cap dom(X)$
- $Y = exp(X)$ (`mic-exp X`)
  $newdom(Y) = [exp(inf(X))^*, exp(sup(X))^*] \cap dom(Y)$
  $newdom(X) = [log(inf(Y))^*, log(sup(Y))^*] \cap dom(X)$
  où * réfère à des calculs de partie entière pour les entiers
- $Y = log(X)$ (`mic-log X`)
  si $inf(X) \leq 0$ **alors** $newdom(Y) = \emptyset$
  sinon $newdom(Y) = [log(inf(X))^*, log(sup(X))^*] \cap dom(Y)$
  finsi $newdom(X) = [exp(inf(Y))^*, exp(sup(Y))^*] \cap dom(X)$
  où * réfère à des calculs de partie entière pour les entiers
- $Y = sin(X)$ (`mic-sin X`)
  si $(sup(X) - inf(X) > 2\pi)$ **alors**
  $newdom(Y) = [-1, 1] \cap dom(Y)$
  sinon
  si $(0 \leq inf(X) < \pi/2)$ **alors**
  si $(0 \leq sup(X) < \pi/2)$ **alors**
  si $sin(inf(X)) > sin(sup(X))$ **alors** $newdom(Y) = [-1, 1] \cap dom(Y)$
  sinon $newdom(Y) = [sin(inf(X)), sin(sup(X))] \cap dom(Y)$
  finsi
  sinon
  si $(\pi/2 \leq sup(X) < \pi)$ **alors**
  $newdom(Y) = [min(sin(inf(X)), sin(sup(X))), 1] \cap dom(Y)$
  sinon
  si $(\pi \leq sup(X) < 3\pi/2)$ **alors** $newdom(Y) = [sin(sup(X)), 1] \cap dom(Y)$
  sinon
  si $(3\pi/2 \leq sup(X) < 2\pi)$ **alors** $newdom(Y) = [-1, 1] \cap dom(Y)$
  finsi
  finsi
  finsi
  sinon
  si $(\pi/2 \leq inf(X) < \pi)$ **alors**
  si $(0 \leq sup(X) < \pi/2)$ **alors**
  $[-1, max(sin(inf(X)), sin(sup(X)))] \cap dom(Y)$
  sinon
  si $(\pi/2 \leq sup(X) < \pi)$ **alors**
  si $sin(inf(X)) < sin(sup(X))$ **alors** $[-1, 1] \cap dom(Y)$
  sinon $[sin(sup(X)), sin(inf(X))]$
  finsi
  sinon
  si $(\pi \leq sup(X) < 3\pi/2)$ **alors**
  $[sin(sup(X)), sin(inf(X))] \cap dom(Y)$

```
                  sinon
                      si (3π/2 ≤ sup(X) < 2π) alors
                          [−1, sin(inf(X))] ∩ dom(Y)
                      finsi
                  finsi
              finsi
          finsi
      sinon
          si (π ≤ inf(X) < 3π/2) alors
              si (0 ≤ sup(X) < π/2) alors
                  newdom(Y) = [−1, sin(sup(X))] ∩ dom(Y)
              sinon
                  si (π/2 ≤ sup(X) < π) alors
                      newdom(Y) = [sin(inf(X)), sin(sup(X))] ∩ dom(Y)
                  sinon
                      si (π ≤ sup(X) < 3π/2) alors
                          si sin(inf(X)) < sin(sup(X)) alors
                              newdom(Y) = [−1, 1] ∩ dom(Y)
                          sinon newdom(Y) = [sin(sup(X)), sin(inf(X))]
                          fin si
                      sinon
                          si (3π/2 ≤ sup(X) < 2π) alors
                              newdom(Y) = [−1, max(sin(inf(X)), sin(sup(X)))] ∩ dom(Y)
                          finsi
                      finsi
                  finsi
              finsi
          sinon
              si (3π/2 ≤ inf(X) < 2π) alors
                  si (0 ≤ sup(X) < π/2) alors
                      newdom(Y) = [sin(inf(X)), sin(sup(X))] ∩ dom(Y)
                  sinon
                      si (π/2 ≤ sup(X) < π) alors
                          newdom(Y) = [sin(inf(X)), 1] ∩ dom(Y)
                      sinon
                          si (π ≤ sup(X) < 3π/2) alors
                              newdom(Y) = [min(sin(inf(X)), sin(sup(X))), 1] ∩ dom(Y)
                          sinon
                              si (π ≤ sup(X) < 3π/2) alors
                                  si sin(inf(X)) > sin(sup(X)) alors
                                      newdom(Y) = [−1, 1] ∩ dom(Y)
                                  sinon
                                      newdom(Y) = [sin(inf(X)), sin(sup(X))]
                                  finsi
                              finsi
                          finsi
                      finsi
                  finsi
              finsi
          finsi
      finsi
      si (dom(Y) ≠ [−1, 1]) alors
          si (sup(X) − inf(X) < 2π) alors
              inf1(X) ← (inf(X)div2π) ∗ 2π + arcsin(inf(Y))
              inf2(X) ← (inf(X) div 2π) ∗ 2π + (π − arcsin(inf(Y)))
              inf3(X) ← (sup(X) div 2π) ∗ 2π + arcsin(inf(Y))
              inf4(X) ← (sup(X) div 2π) ∗ 2π + (π − arcsin(inf(Y)))
              sup1(X) ← (inf(X) div 2π) ∗ 2π + arcsin(inf(Y))
              sup2(X) ← (inf(X) div 2π) ∗ 2π + (π − arcsin(inf(Y)))
              sup3(X) ← (sup(X) div 2π) ∗ 2π + arcsin(inf(Y))
              sup4(X) ← (sup(X) div 2π) ∗ 2π + (π − arcsin(inf(Y)))
              newdom(X) = [min(inf1(X), sup1(X)), max(inf1(X),sup1(X))] ∩ dom(X)
                              ∪
              newdom(X) = [min(inf2(X), sup2(X)), max(inf2(X),sup2(X))] ∩ dom(X)
                              ∪
              newdom(X) = [min(inf3(X), sup3(X)), max(inf3(X),sup3(X))] ∩ dom(X)
                              ∪
              newdom(X) = [min(inf4(X), sup4(X)), max(inf4(X),sup4(X))] ∩ dom(X)
          finsi
      finsi
finsi
```

finsi
finsi

- $Y = cos(X)$ (`mic-cos` X)
  se construit sur le même principe que (`mic-sin` X)

Contraintes secondaires numériques

- $Z = X + Y$ (`mic-add` X Y)
  $newdom(Z) = [inf(X) + inf(Y), sup(X) + sup(Y)] \cap dom(Z)$
  $newdom(X) = [inf(Z) - sup(Y), sup(Z) - inf(Y)] \cap dom(X)$
  $newdom(Y) = [inf(Z) - sup(X), sup(Z) - inf(Y)] \cap dom(Y)$
- $Z = X - Y$ (`mic-sub` X Y)
  $newdom(Z) = [inf(X) - sup(Y), sup(X) - inf(Y)] \cap dom(Z)$
  $newdom(X) = [inf(Z) + inf(Y), sup(Z) + sup(Y)] \cap dom(X)$
  $newdom(Y) = [inf(X) - sup(Z), sup(X) - inf(Z)] \cap dom(Y)$
- $Z = X * Y$ (`mic-mul` X Y)
  $newdom(Z) = [min(inf(X)*inf(Y), inf(X)*sup(Y), sup(X)*inf(Y), sup(X)*sup(Y)), max(inf(X)*inf(Y), inf(X)* sup(Y), sup(X)*inf(Y), sup(X)*sup(Y))] \cap dom(Z)$
  si $inf(Y) = 0$ **alors**
  si $sup(Y) <> 0$ **alors**
  $newdom(X) = [min(inf(Z)/sup(Y), sup(Z)/sup(Y)), +\infty[\cap dom(X)$
  finsi
  sinon
  si $sup(Y) = 0$ **alors**
  $newdom(X) =]-\infty, max(inf(Z)/inf(Y), sup(Z)/inf(Y)] \cap dom(X)$
  sinon
  $newdom(X) =$
  $[min(inf(Z)/inf(Y), inf(Z)/sup(Y), sup(Z)/inf(Y), sup(Z)/sup(Y)),$
  $max(inf(Z)/inf(Y), inf(Z)/sup(Y), sup(Z)/inf(Y), sup(Z)/sup(Y))] \cap dom(X)$
  finsi
  finsi

si $inf(X) = 0$ **alors**
si $sup(X) <> 0$ **alors**
$newdom(Y) = [min(inf(Z)/sup(X), sup(Z)/sup(X)), +\infty[\cap dom(Y)$
finsi
sinon
si $sup(X) = 0$ **alors**
$newdom(Y) =]-\infty, max(inf(Z)/inf(X), sup(Z)/inf(X)] \cap dom(Y)$
sinon
$newdom(Y) =$
$[min(inf(Z)/inf(X), inf(Z)/sup(X), sup(Z)/inf(X), sup(Z)/sup(X)),$
$max(inf(Z)/inf(X), inf(Z)/sup(X), sup(Z)/inf(X), sup(Z)/sup(X))] \cap dom(Y)$
finsi
finsi

- $Z = X/Y$ (`mic-div` X Y)
  si $inf(Y) \leq 0$ **et** $sup(Y) \geq 0$ **alors**
  $newdom(Z) =]min(inf(X)/0^-, sup(X)/0^-), max(inf(X)/inf(Y), sup(X)/inf(Y))]$
  $\cup[min(inf(X)/sup(Y), sup(X)/sup(Y)], max(inf(X)/0^+, sup(X)/0^+)[\cap dom(Z)$
  sinon
  $newdom(Z) = [min(inf(X)/inf(Y), inf(X)/sup(Y), sup(X)/inf(Y), sup(X)/sup(Y)),$
  $max(inf(X)/inf(Y), inf(X)/sup(Y), sup(X)/inf(Y), sup(X)/sup(Y))] \cap dom(Z)$
  finsi
  $newdom(X) = [min(inf(Z) * inf(Y), inf(Z) * sup(Y), sup(Z) * inf(Y), sup(Z) * sup(Y)),$
  $max(inf(Z) * inf(Y), inf(Z) * sup(Y), sup(Z) * inf(Y), sup(Z) * sup(Y))] \cap dom(X)$
  si $inf(Z) \leq 0$ **et** $sup(Z) \geq 0$ **alors**
  $newdom(Y) =]min(inf(X)/0^-, sup(X)/0^-), max(inf(X)/inf(Z), sup(X)/inf(Z))]$
  $\cup[min(inf(X)/sup(Z), sup(X)/sup(Z)], max(inf(X)/0^+, sup(X)/0^+)[\cap dom(Y)$
  sinon
  $newdom(Y) = [min(inf(X)/inf(Z), inf(X)/sup(Z), sup(X)/inf(Z), sup(X)/sup(Z)),$
  $max(inf(X)/inf(Z), inf(X)/sup(Z), sup(X)/inf(Z), sup(X)/sup(Z))] \cap dom(Y)$
  finsi
- $Z = X^Y$ (`mic-power` X Y)
  si $sup(X) \leq 0$ **alors**
  $dom(Y) \leftarrow dom(Y) -]-1, 1[$
  finsi
  $newdom(Z) = [min(inf(X)^{inf(Y)}, inf(X)^{sup(Y)}, sup(X)^{inf(Y)}, sup(X)^{sup(Y)}),$
  $max(inf(X)^{inf(Y)}, inf(X)^{sup(Y)}, sup(X)^{inf(Y)}, sup(X)^{sup(Y)})] \cap dom(Z)$
  si $inf(Y) = 0$ **alors**
  si $sup(Y) \neq 0$ **alors**
  $newdom(X) = [min(inf(Z)^{1/sup(Y)}, 1, sup(Z)^{1/sup(Y)}),$
  $max(inf(Z)^{1/sup(Y)}, 1, sup(Z)^{1/sup(Y)})] \cap dom(X)$

finsi
sinon
si $sup(Y) = 0$ **alors**
$newdom(X) = [min(inf(Z)^{1/inf(Y)}, 1, sup(Z)^{1/inf(Y)}),$
$max(inf(Z)^{1/inf(Y)}, 1, sup(Z)^{1/inf(Y)})] \cap dom(X)$
finsi
$newdom(X) = [min(inf(Z)^{1/inf(Y)}, inf(Z)^{1/sup(Y)}, sup(Z)^{1/inf(Y)}, sup(Z)^{1/sup(Y)}),$
$max(inf(Z)^{1/inf(Y)}, inf(Z)^{1/sup(Y)}, sup(Z)^{1/inf(Y)}, sup(Z)^{1/sup(Y)})] \cap dom(X)$
finsi

si $inf(X) \neq 0$ **et** $sup(X) \neq 0$ **alors**
si $inf(Z) \neq 0$ **et** $sup(Z) \neq 0$ **alors**
$newdom(Y) = [min(log(\mid inf(Z) \mid)/log(\mid inf(X) \mid),$
$log(\mid inf(Z) \mid)/log(\mid sup(X) \mid), log(\mid sup(Z) \mid)/log(\mid inf(X) \mid),$
$log(\mid sup(Z) \mid)/log(\mid sup(X) \mid)),$
$max(log(\mid inf(Z) \mid)/log(\mid inf(X) \mid), log(\mid inf(Z) \mid)/log(\mid sup(X) \mid),$
$log(\mid sup(Z) \mid)/log(\mid inf(X) \mid), log(\mid sup(Z) \mid)/log(\mid sup(X) \mid))] \cap dom(Y)$
finsi
finsi

Contrainte secondaire booléenne unaire

- $Y = not(X)$ (`mic-not` X)

si $\mid dom(X) \mid = 1$ **alors**
$newdom(Y) = [not(val(X)), not(val(X))] \cap dom(Y)$
finsi

si $\mid dom(Y) \mid = 1$ **alors**
$newdom(X) = [not(val(Y)), not(val(Y))] \cap dom(X)$
finsi

Contraintes secondaires booléennes

- $Z = X or Y$ (`mic-or` X Y)

si $\mid dom(X) \mid = 1$ **alors**
si $\mid dom(Y) \mid = 1$ **alors**
$newdom(Z) = [val(X)\ or\ val(Y), val(X)\ or\ val(Y)] \cap dom(Z)$
sinon
si $val(X) = vrai$ **alors**
$newdom(Z) = [vrai, vrai] \cap dom(Z)$
finsi
finsi
finsi

si $\mid dom(Y) \mid = 1$ **alors**
si $\mid dom(Z) \mid = 1$ **alors**
si $val(Y) = faux$ **alors**
$newdom(X) = [val(Z), val(Z)] \cap dom(X)$
finsi
finsi
sinon
si $\mid dom(Z) \mid = 1$ **alors**
si $val(Z) = faux$ **alors**
$newdom(X) = [faux, faux] \cap dom(X)$
finsi
finsi

si $\mid dom(X) \mid = 1$ **alors**
si $\mid dom(Z) \mid = 1$ **alors**
si $val(X) = faux$ **alors**
$newdom(Y) = [val(Z), val(Z)] \cap dom(Y)$
finsi
finsi
sinon
si $\mid dom(Z) \mid = 1$ **alors**
si $val(Z) = faux$ **alors**
$newdom(X) = [faux, faux] \cap dom(Y)$
finsi
finsi

- $Z = X and Y$ (`mic-and` X Y)

si $\mid dom(X) \mid = 1$ **alors**
si $\mid dom(Y) \mid = 1$ **alors**
$newdom(Z) = [val(X)\ and\ val(Y), val(X)\ andval(Y)] \cap dom(Z)$
sinon
si $val(X) = faux$ **alors**
$newdom(Z) = [faux, faux] \cap dom(Z)$
finsi

finsi
sinon
si $\mid dom(Y) \mid = 1$ **alors**
si $val(Y) = faux$ **alors**
$newdom(Z) = [faux, faux] \cap dom(Z)$
finsi
finsi
finsi
si $\mid dom(Z) \mid = 1$ **alors**
si $\mid dom(Y) \mid = 1$ **alors**
si $val(Y) = vrai$ **alors**
$newdom(X) = [val(Z), val(Z)] \cap dom(X)$
finsi
sinon
si $val(Z) = vrai$ **alors**
$newdom(X) = [vrai, vrai] \cap dom(X)$
finsi
finsi
finsi
si $\mid dom(Z) \mid = 1$ **alors**
si $\mid dom(X) \mid = 1$ **alors**
si $val(X) = vrai$ **alors**
$newdom(Y) = [val(Z), val(Z)] \cap dom(Y)$
finsi
sinon
si $val(Z) = vrai$ **alors**
$newdom(Y) = [vrai, vrai] \cap dom(Y)$
finsi
finsi
finsi

- $Z = X \Rightarrow Y$ (`mic-implies` X Y)
  équivalent à (`mic-or` (`mic-not` X) Y)
- $Z = X \Rightarrow Y$ (`mic-equiv` X Y)
  équivalent à (`mic-and` (`mic-implies` X Y) (`mic-implies` Y X))

Contraintes secondaires semi-booléennes

- $Z = X \stackrel{?}{=} Y$ (`mic-is-eq` X Y)

si $dom(X) \cap dom(Y) = \emptyset$ **alors**
$newdom(Z) = [faux, faux] \cap dom(Z)$
sinon
si $\mid dom(X) \mid = 1$ **et** $\mid dom(Y) \mid = 1$ **alors**
$newdom(Z) = [val(X) = val(Y), val(X) = val(Y)] \cap dom(Z)$
finsi
finsi
si $\mid dom(Z) \mid = 1$ **et** $val(Z) = vrai$ **alors**
si $\mid dom(Y) \mid = 1$ **alors**
$newdom(X) = dom(Y) \cap dom(X)$
finsi
finsi
si $\mid dom(Z) \mid = 1$ **et** $val(Z) = vrai$ **alors**
si $\mid dom(X) \mid = 1$ **alors**
$newdom(Y) = dom(X) \cap dom(Y)$
finsi
finsi

- $Z = X \stackrel{?}{\neq} Y$ (`mic-is-neq` X Y)

si $dom(X) \cap dom(Y) = \emptyset$ **alors**
$newdom(Z) = [vrai, vrai] \cap dom(Z)$
sinon
si $\mid dom(X) \mid = 1$ **et** $\mid dom(Y) \mid = 1$ **alors**
$newdom(Z) = [val(X) \neq val(Y), val(X) \neq val(Y)] \cap dom(Z)$
finsi
finsi
si $\mid dom(Z) \mid = 1$ **et** $val(Z) = vrai$ **alors**
si $\mid dom(Y) \mid = 1$ **alors**
$newdom(X) = (dom(X)/val(Y)) \cap dom(X)$
finsi
finsi
si $\mid dom(Z) \mid = 1$ **et** $val(Z) = vrai$ **alors**
si $\mid dom(X) \mid = 1$ **alors**
$newdom(Y) = (dom(Y)/val(X)) \cap dom(Y)$

finsi
finsi

- $Z = X <^? Y$ (`mic-is-lt` X Y)
  si $dom(X) \cap dom(Y) = \emptyset$ **alors**
  si $inf(Y) > sup(X)$ **alors**
  $newdom(Z) = [vrai, vrai] \cap dom(Z)$
  sinon
  si $inf(X) > sup(Y)$ **alors**
  $newdom(Z) = [faux, faux] \cap dom(Z)$
  finsi
  finsi
  sinon
  si $\mid dom(X) \mid = 1$ **et** $\mid dom(Y) \mid = 1$ **alors**
  $newdom(Z) = [inf(X) < inf(Y), inf(X) < inf(Y)] \cap dom(Z)$
  finsi
  finsi
  si $\mid dom(Z) \mid = 1$
  si $inf(Z) = vrai$ **alors**
  si $\mid dom(Y) \mid = 1$ **alors**
  $newdom(X) = [inf(X), val(Y)^-] \cap dom(X)$
  finsi
  sinon
  si $\mid dom(Y) \mid = 1$ **alors**
  $newdom(X) = [val(Y), sup(X)] \cap dom(X)$
  finsi
  finsi
  finsi
  si $\mid dom(Z) \mid = 1$
  si $\mid dom(X) \mid = 1$ **alors**
  si $inf(Z) = vrai$ **alors**
  $newdom(Y) = [val(X)^+, sup(Y)] \cap dom(Y)$
  finsi
  sinon
  si $\mid dom(Z) \mid = 1$
  $newdom(Y) = [inf(Y)), val(X)] capdom(Y)$
  finsi
  finsi
  finsi
- $Z = X \overset{?}{\leq} Y$ (`mic-is-le` X Y)
  principe semblable à (`mic-is-lt` X Y)
- $Z = X >^? Y$ (`mic-is-gt` X Y)
  équivalent à (`mic-is-lt` Y X)
- $Z = X \overset{?}{\geq} Y$ (`mic-is-ge` X Y)
  équivalent à (`mic-is-le` Y X)
- $Z = X \overset{?}{=} Y$ (`mic-is-eq` X Y)
  si $dom(X) \cap dom(Y) = \emptyset$ **alors**
  $newdom(Z) = [faux, faux] \cap dom(Z)$
  sinon
  si $\mid ext(X) \mid = 1$ **et** $\mid ext(Y) \mid = 1$ **alors**
  $newdom(Z) = [val(X) = val(Y), val(X) = val(Y)] \cap dom(Z)$
  finsi
  finsi
  si $\mid dom(Z) \mid = 1$ **et** $val(Z) = vrai$ **alors**
  si $\mid ext(Y) \mid = 1$ **alors**
  $newext(X) = ext(Y) \cap ext(X)$
  finsi
  finsi
  si $\mid dom(Z) \mid = 1$ **et** $val(Z) = vrai$ **alors**
  si $\mid ext(X) \mid = 1$ **alors**
  $newext(Y) = ext(X) \cap ext(Y)$
  finsi
  finsi
- $Z = X \overset{?}{\neq} Y$ (`mic-is-neq` X Y)
  si $dom(X) \cap dom(Y) = \emptyset$ **alors**
  $newdom(Z) = [vrai, vrai] \cap dom(Z)$
  sinon
  si $\mid ext(X) \mid = 1$ **et** $\mid ext(Y) \mid = 1$ **alors**
  $newdom(Z) = [val(X) \neq val(Y), val(X) \neq val(Y)] \cap dom(Z)$

finsi
finsi
si $\mid dom(Z) \mid = 1$ **et** $val(Z) = vrai$ **alors**
si $\mid ext(Y) \mid = 1$ **alors**
$newext(X) = (ext(X)/val(Y)) \cap ext(X)$
finsi
finsi
si $\mid dom(Z) \mid = 1$ **et** $val(Z) = vrai$ **alors**
si $\mid ext(X) \mid = 1$ **alors**
$newext(Y) = (ext(Y)/ext(X)) \cap ext(Y)$
finsi
finsi

- $Z = X \overset{?}{\subset} Y$ (`mic-is-include X Y`)

si $dom(X) \cap dom(Y) = \emptyset$ **ou** $card(X) > card(Y)$ **alors**
$newdom(Z) = [faux, faux] \cap dom(Z)$
sinon
si $\mid ext(X) \mid = 1$ **et** $\mid ext(Y) \mid = 1$ **alors**
$newdom(Z) = [val(X) \subset val(Y), val(X) \subseteq val(Y)] \cap dom(Z)$
finsi
finsi
si $\mid dom(Z) \mid = 1$
si $val(Z) = vrai$ **alors**
si $ext(X) \neq \emptyset$ **et** $ext(Y) \neq \emptyset$ **alors**
$newext(X) = \{e \in ext(X) \mid \exists f \in ext(Y), e \subset f\}$
sinon
$newdom(X) = dom(X) \cap dom(Y)$
finsi
finsi
finsi
si $\mid dom(Z) \mid = 1$
si $inf(Z) = vrai$ **alors**
si $ext(X) \neq \emptyset$ **et** $ext(X) \neq \emptyset$ **alors**
$newext(Y) = \{e \in ext(Y) \mid \exists f \in ext(X), f \subset e\}$
sinon
$newcard(Y) = [min(inf(card(X)), inf(card(Y))), min(sup(card(X)), sup(card(Y))^{-})] \cap card(X)$
finsi
finsi
finsi

- $Z = X \overset{?}{\subseteq} Y$ (`mic-is-include-eq X Y`)

même principe que (`mic-is-include X Y`)

- $Z = X \overset{?}{\not\subset} Y$ (`mic-is-not-include X Y`)

si $dom(X) \cap dom(Y) = \emptyset$ **ou** $card(X) > card(Y)$ **alors**
$newdom(Z) = [vrai, vrai] \cap dom(Z)$
sinon
si $\mid ext(X) \mid = 1$ **et** $\mid ext(Y) \mid = 1$ **alors**
$newdom(Z) = [val(X) \not\subset val(Y), val(X) \not\subset val(Y)] \cap dom(Z)$
finsi
finsi
si $\mid dom(Z) \mid = 1$
si $val(Z) = vrai$ **alors**
si $\mid ext(X) \mid = 1$ **et** $ext(Y) \neq \emptyset$ **alors**
$newext(Y) = ext(Y)/val(X)$
finsi
sinon
si $\mid ext(X) \mid = 1$ **et** $ext(Y) \neq \emptyset$ **alors**
$newext(Y) = \{e \in ext(Y) \mid \exists f \in val(X), f \subset e\}$
finsi
finsi
finsi
si $\mid dom(Z) \mid = 1$
si $val(Z) = vrai$ **alors**
si $\mid ext(Y) \mid = 1$ **et** $ext(X) \neq \emptyset$ **alors**
$newext(X) = \{e \in ext(X) \mid e \not\subset val(Y)\}$
finsi
sinon
si $\mid ext(Y) \mid = 1$ **et** $ext(X) \neq \emptyset$ **alors**
$newext(X) = \{e \in ext(X) \mid \exists f \in val(Y), f \subseteq e\}$

finsi
finsi
finsi

- $Z = X \overset{?}{\in} Y$ (`mic-is-not-include-eq` X Y)
même principe que `mic-is-include-eq`

- $Z = X \overset{?}{\in} Y$ (`mic-is-in` X Y)
si $dom(X) \cap dom(Y) = \emptyset$ **alors**
$newdom(Z) = [faux, faux] \cap dom(Z)$
sinon
si $\mid dom(X) \mid = 1$ **et** $\mid ext(Y) \mid = 1$ **alors**
$newdom(Z) = [val(X) \in val(Y), val(X) \in val(Y)] \cap dom(Z)$
finsi
finsi
si $\mid dom(Z) \mid = 1$
si $val(Z) = vrai$ **alors**
si $\mid dom(X) \mid = 1$ **et** $ext(Y) \neq \emptyset$ **alors**
$newext(Y) = \{e \in ext(Y) \mid val(X) \in e\}$
finsi
sinon
si $\mid dom(X) \mid = 1$ **et** $ext(Y) \neq \emptyset$ **alors**
$newext(Y) = \{e \in ext(Y) \mid val(X) \notin e\}$
finsi
finsi
finsi
si $\mid dom(Z) \mid = 1$
si $val(Z) = vrai$ **alors**
si $\mid ext(Y) \mid = 1$ **alors**
$newdom(X) = \{e \in dom(X) \mid e \in val(Y)\}$
finsi
sinon
si $\mid ext(Y) \mid = 1$ **et** $ext(X) \neq \emptyset$ **alors**
$newval(X) = \{e \in dom(X) \mid e \notin val(Y)\}$
finsi
finsi
finsi

- $Z = X \overset{?}{\notin} Y$ (`mic-is-not-in` X Y)
si $dom(X) \cap dom(Y) = \emptyset$ **alors**
$newdom(Z) = [vrai, vrai] \cap dom(Z)$
sinon
si $\mid dom(X) \mid = 1$ **et** $\mid ext(Y) \mid = 1$ **alors**
$newdom(Z) = [val(X) \notin val(Y), val(X) \notin val(Y)] \cap dom(Z)$
finsi
finsi
si $\mid dom(Z) \mid = 1$
si $val(Z) = vrai$ **alors**
si $\mid dom(X) \mid = 1$ **et** $ext(Y) \neq \emptyset$ **alors**
$newext(Y) = \{e \in ext(Y) \mid val(X) \notin e\}$
finsi
sinon
si $\mid dom(X) \mid = 1$ **et** $ext(Y) \neq \emptyset$ **alors**
$newext(Y) = \{e \in ext(Y) \mid val(X) \in e\}$
finsi
finsi
finsi
si $\mid dom(Z) \mid = 1$
si $val(Z) = vrai$ **alors**
si $\mid ext(Y) \mid = 1$ **alors**
$newdom(X) = \{e \in dom(X) \mid e \notin val(Y)\}$
finsi
sinon
si $\mid ext(Y) \mid = 1$ **et** $ext(X) \neq \emptyset$ **alors**
$newdom(X) = \{e \in dom(X) \mid e \in dom(Y)\}$
finsi
finsi
finsi

- $Z = \overset{?}{every}\ X\ Y$ (`mic-is-every` f X Y)
si $\mid ext(X) \mid = 1$ **alors**
$newdom(Y) = (every\ f\ val(X)\ val(Y))$

finsi
si $\mid dom(Z) \mid = 1$
  si $val(Z) = vrai$ **alors**
    si $ext(X) \neq \emptyset$ **alors**
      $newext(X) = \{e \in ext(X) \mid \forall x \in e, f(x, val(Y)) = vrai\}$
    sinon
      si $dom(X)\ fini$ **alors** $newdom(X) = \{e \in dom(X) \mid f(e, val(Y)) = vrai\}$
    finsi
  sinon
    si $ext(X) \neq \emptyset$ **alors**
      $newext(X) = \{e \in ext(X) \mid \forall x \in e, f(x, val(Y)) = faux\}$
    finsi
  finsi
finsi

• $Y \overset{?}{=} any\ X$ (`mic-is-any` f X)
si $\mid ext(X) \mid = 1$ **alors**
  $newdom(Y) = (any\ f\ ext(X))$
finsi
si $\mid dom(Y) \mid = 1$
  si $val(Y) = vrai$ **alors**
    si $ext(X) \neq \emptyset$ **alors**
      $newext(X) = \{e \in ext(X) \mid \exists x \in e, f(x) = vrai\}$
    finsi
  sinon
    si $ext(X) \neq \emptyset$ **alors**
      $newext(X) = \{e \in ext(X) \mid \forall x \in e, f(x) = vrai\}$
    sinon
      si $dom(X) fini$ **alors**
        $newdom(X) = \{e \in dom(X) \mid f(e) = vrai\}$
      finsi
    finsi
  finsi
finsi

• $Y \overset{?}{=} all\ X$ (`mic-is-all` f X)
si $\mid ext(X) \mid = 1$ **alors**
  si $\forall x, y \in ext(X)\ f(x, y) = vrai$ **alors**
    $newdom(Y) = [vrai, vrai] \cap dom(Y)$
  sinon
    $newdom(Y) = [faux, faux] \cap dom(Y)$
  finsi
finsi
si $\mid dom(Y) \mid = 1$
  si $val(Y) = vrai$ **alors**
    si $ext(X) \neq \emptyset$ **alors**
      $newext(X) = \{e \in ext(X) \mid \forall x, y \in e, f(x, y) = vrai\}$
    finsi
  finsi
sinon
  si $ext(X) \neq \emptyset$ **alors**
    $newext(X) = ext(X) - \{e \in ext(X) \mid \forall x, y \in e, f(x, y) = vrai\}$
  finsi
finsi

• $Z \overset{?}{=} compar\ f\ X\ Y$ (`mic-is-compar-f` f X Y)
si $\mid ext(X) \mid = 1$ **alors**
  si $\mid ext(Y) \mid = 1$ **alors**
    $newdom(Z) = [(compar\ f\ e$ val(Y)),(comparfe$val(Y)$)]
  finsi
finsi
si $\mid dom(Z) \mid = 1$
  si $val(Z) = vrai$ **alors**
    si $ext(X) \neq \emptyset$ **alors**
      si $\mid ext(Y) \mid = 1$ **alors**
        $newext(X) = \{e \in ext(X) \mid (compar\ f\ e\ ext(Y)) = vrai\}$
      finsi
    finsi
  sinon
    si $ext(X) \neq \emptyset$ **alors**
      si $\mid ext(Y) \mid = 1$ **alors**
        $newext(X) = \{e \in ext(X) \mid (compar\ f\ e\ ext(Y)) = faux\}$

finsi
finsi
finsi
finsi

si $\mid dom(Z) \mid = 1$
si $val(Z) = vrai$ **alors**
si $ext(Y) \neq \emptyset$ **alors**
si $\mid ext(X) \mid = 1$ **alors**
$newext(X) = \{e \in ext(X) \mid (compar\ f\ ext(X)\ e) = vrai\}$
finsi
finsi
sinon
si $ext(Y) \neq \emptyset$ **alors**
si $\mid ext(X) \mid = 1$ **alors**
$newext(Y) = \{e \in ext(Y) \mid (compar\ f\ ext(X)\ e) = faux\}$
finsi
finsi
finsi
finsi

où $(comparfxy)$ est une fonction qui compare selon f les éléments de même rang dans x et y.

- $Y \stackrel{?}{=} at - least\ n\ X$ (`mic-is-at-least` n X)
  si $\mid ext(X) \mid \neq 0$ **alors**
  si $\forall e \in ext(X)$ t.q. $\mid e \mid \geq n$ **ou** $card(X) > n$ **alors**
  $newdom(Y) = [vrai, vrai] \cap dom(Y)$
  sinon
  $newdom(Y) = [faux, faux] \cap dom(Y)$
  finsi
  finsi

  si $\mid dom(Y) \mid = 1$
  si $val(Y) = vrai$ **alors**
  si $ext(X) \neq \emptyset$ **alors**
  $newext(X) = \{e \in ext(X) t.q. \mid e \mid \geq n\}$
  finsi
  finsi
  sinon
  si $ext(X) \neq \emptyset$ **alors**
  $newext(X) = ext(X) - \{e \in ext(X) t.q. \mid e \mid \geq n\}$
  finsi
  finsi

- $Y \stackrel{?}{=} at - most\ n\ X$ (`mic-is-at-most` n X)
  même principe que (`mic-at-least` n X)

A.1.1.3 Contraintes de transformation

- $Y = \{X\}$ (`mic-mono-to-set` X)
  $newext(Y) = \{val(X)\}$
  $newdom(Y) = dom(X)$
  $card(Y) = 1$
  si $ext(Y) = 1$ **alors**
  $newdom(X) = [element - of(ext(Y)), element - of(ext(Y))]$
  fin si

- $Y = (X)$ (`mic-mono-to-list` X)
  même principe que mono-to-set

- $Y = \{S\}$ (`mic-set` S)
  $newext(Y) = (mapcar\#'valS)$
  $newdom(Y) = \cup_{i=1}^{i=card(S)} dom(S_i)$
  $card(Y) = card(S)$

- $Y = (L)$ (`mic-list` X)
  $newext(Y) = (mapcar\#'valL)$
  $newdom((nthiY)) = dom((nthiL))$
  $card(Y) = card(L)$

A.1.2 Contraintes sur variables multivaluées numériques

A.1.2.1 Contraintes principales

- $X = Y$ (`mic-eq` X Y)
  - **si** $ext(X) \neq \emptyset$ **alors**
    - **si** $ext(Y) \neq \emptyset$ **alors**
      - $newext(X) = ext(X) \cap ext(Y)$
    - **finsi**
  - **sinon**
    - $newdom(X) = dom(X) \cap dom(Y)$
    - $newsauf(X) = sauf(X) \cup sauf(Y)$
    - $newcard(X) = card(X) \cap card(Y)$
    - $newrequired(X) = required(X) \cap required(Y)$
  - **finsi**
  - **si** $ext(Y) \neq \emptyset$ **alors**
    - **si** $ext(X) \neq \emptyset$ **alors**
      - $newext(Y) = ext(X) \cap ext(Y)$
    - **finsi**
  - **sinon**
    - $newdom(Y) = dom(Y) \cap dom(X)$
    - $newsauf(Y) = sauf(Y) \cup sauf(X)$
    - $newcard(Y) = card(Y) \cap card(X)$
  - **finsi**
- $X \neq Y$ (`mic-neq` X Y)
  - **si** $\mid ext(Y) \mid = 1$ **alors**
    - **si** $\mid ext(X) \mid \neq \emptyset$ **alors**
      - $newext(X) = ext(X) - val(Y)$
      - $newsauf(X) = sauf(X) \cup sauf(Y)$
    - **finsi**
  - **finsi**
  - **si** $\mid ext(X) \mid = 1$ **alors**
    - **si** $\mid ext(Y) \mid \neq \emptyset$ **alors**
      - $newext(Y) = ext(Y) - ext(X)$
      - $newsauf(X) = sauf(X) \cup sauf(Y)$
    - **finsi**
  - **finsi**
- $X \leftarrow Y$ (`mic-assign` X Y)
  - **si** $ext(X) \neq \emptyset$ **alors**
    - **si** $xt(Y) \neq \emptyset$ **alors**
      - $newext(X) = ext(X) \cap ext(Y)$
    - **finsi**
  - **sinon**
    - $newdom(X) = dom(X) \cap dom(Y)$
    - $newsauf(X) = sauf(X) \cup sauf(Y)$
    - $newcard(X) = card(X) \cap card(Y)$
    - $newrequired(X) = required(X) \cap required(Y)$
  - **finsi**
- $X \subset Y$ (`mic-include` X Y)
  - $newdom(X) = dom(X) \cap dom(Y)$
  - $newcard(X) = [min(inf(card(X)), inf(card(Y))), min(sup(card(X)), sup(card(Y))^-)] \cap card(X)$
  - $newsauf(X) = sauf(X) \cup sauf(Y)$
  - $newrequired(X) = required(X) \cap required(Y)$
  - $newdom(Y) = dom(Y) \cap dom(Y)$
  - $newcard(Y) = [max(inf(card(X)^+), inf(card(Y))), max(sup(card(X)), sup(card(Y))^-)] \cap card(Y)$
- $X \subseteq Y$ (`mic-include-eq` X Y)
  - même principe que (`mic-include` X Y)
- $X \not\subset Y$ (`mic-not-include` X Y)
  - **si** $\mid ext(X) \mid = 1$ **et** $ext(Y) \neq \emptyset$ **alors**
    - $newext(Y) = ext(Y) - val(X)$
    - $newsauf(Y) = sauf(Y) \cup val(X)$
  - **finsi**
  - **si** $\mid ext(Y) \mid = 1$ **et** $ext(X) \neq \emptyset$ **alors**
    - $newext(X) = ext(X) - \{e \mid e \subset val(Y)\}$
    - $newsauf(X) = sauf(X) \cup \{e \mid e \subset val(Y)\}$
  - **finsi**
- $X \subseteq Y$ (`mic-include-eq` X Y)
  - même principe que (`mic-include` X Y)

- $X \not\subseteq Y$ (`mic-not-include-eq` X Y)
  même principe que `mic-not-include-eq`
- $X \in Y$ (`mic-in` X Y)
  $newdom(X) = dom(X) \cap dom(Y)$

  si $ext(Y) \neq \emptyset$ **alors**
      si $\mid dom(X) \mid = 1$ **alors**
          $newext(Y) = \{e \in ext(Y) \mid val(X) \in e\}$
          $newrequired(Y) = required(X) \cup val(X)$
      finsi
  finsi
- $X \notin Y$ (`mic-not-in` X Y)
  si $ext(Y) \neq \emptyset$ **alors**
      si $\mid dom(X) \mid = 1$ **alors**
          $newext(Y) = ext(Y) - \{e \in ext(Y) \mid val(X) \in e\}$
      finsi
  finsi
- $every\ f\ X$ (`mic-every-f` f X Y)
  si $dom(X) fini$ **alors**
      $newdom(X) = \{e \in dom(X) \mid f(e, val(Y)) = vrai\}$
  finsi
- $any\ f\ X$ (`mic-any` f X Y)
  si $ext(X) \neq \emptyset$ **alors**
      $newext(X) = \{e \in ext(X) \mid \exists x \in e \mid f(x, val(Y)) = vrai\}$
  finsi
- $all\ f\ X$ (`mic-all-f` f X)
  si $ext(X) \neq \emptyset$ **alors**
      $newext(X) = \{e \in ext(X) \mid \forall x \in e, \forall y \in e, f(x, y) = vrai\}$
  finsi
- $compar\ f\ X\ Y$ (`mic-compar-f` f X Y)
  si $ext(X) \neq \emptyset$ **alors**
      si $ext(Y) \neq \emptyset$ **alors**
          $newext(X) = \{i \in ext(X) \mid \exists i \in ext(Y), (compar\ i\ j)\}$
          $newext(Y) = \{i \in ext(Y) \mid \exists j \in ext(Y), (compar\ i\ j)\}$
      finsi
  finsi

où $(comparfxy)$ est une fonction qui compare selon f les éléments de même rang dans x et y.

- $at - least\ n\ X$ (`mic-at-least` n X)
  $newcard(X) = [max(n, inf(X)), max(n, sup(X))] \cap card(X)$
- $at - most\ n\ X$ (`mic-at-most` n X)
  $newcard(X) = [min(n, inf(X)), min(n, sup(X))] \cap card(X)$

A.1.2.2 Contraintes secondaires

- $Y = list - to - set\ X$ (`mic-list-to-set` X)
  si $ext(X) \neq \emptyset$ **alors**
      $newext(Y) = \{list - to - set(f) f \in ext(X)\}$
  sinon
      $newdom(Y) = dom(X) \cap dom(Y)$
      $newcard(Y) = [min(inf(card(Y)), inf(card(X))), max(inf(card(Y)), inf(card(X)))] \cap dom(Y)$
      $newsauf(Y) = \{list - to - set(f), f \in sauf(X)\}$
      $newrequired(Y) = required(X) \cap required(Y)$
  finsi

  si $ext(Y) \neq \emptyset$ **alors**
      si $ext(X) \neq \emptyset$ **alors**
          $newext(X) = \{e \mid \exists f \in ext(Y) \mid f = list - to - set(e)\}$
      finsi
  sinon
      $newdom(X) = dom(X) \cap dom(Y)$
      $newsauf(X) = \{e \mid f = list - to - set(e), f \in sauf(X)\}$
      $newrequired(X) = required(X) \cap required(Y)$
  finsi
- $Y = multi - to - mono\ X$ (`mic-multi-to-mono` X)
  si $\mid ext(X) \mid = 1$ **alors**
      si $(cardext(X)) = 1$ **alors**
          $newdom(Y) = [(caarext(X)), (caarext(X))]$
      finsi

finsi
si $\mid dom(X) \mid= 1$ **alors**
  si type-of(Y) = list **alors**
    $newext(Y) = (val(X))$
  sinon
    $newext(Y) = \{val(X)\}$
  finsi
finsi
$newdom(X) = dom(X) \cap dom(Y)$

- $Y = car\ X$ (`mic-car` X)
  si $ext(X) \neq \emptyset$ **alors**
    $newdom(Y) = \{e \mid e = (car\ l), l \in ext(X)\}$
  finsi
  $newdom(Y) = dom(X) \cap dom(Y)$
  si $\mid dom(Y) \mid= 1$ **alors**
    $required - at(X) = required - at(X) \cup (inf(Y), 0)$
  finsi
  si $ext(X) \neq \emptyset$ **alors**
    $newext(X) = \{e \in ext(X) \mid (care) = inf(Y)\}$
  finsi
- $Y = cdr\ X$ (`mic-cdr` X)
  si $ext(X) \neq \emptyset$ **alors**
    $newdom(Y) = \{e \mid e = (cdr\ l), l \in ext(X)\}$
  finsi
  $newdom(Y) = dom(X) \cap dom(Y)$
  $newcard(Y) = card(X) - 1$
  $newsauf(Y) = \{e \mid e = (cdrl), l \in sauf(X)\}$
  $newrequired(Y) = \{(e, n-1)\forall(e,n) \in required(X)\}$
  si $ext(Y) \neq \emptyset$ **alors**
    si $ext(X) \neq \emptyset$ **alors**
      $newext(X) = \{e \in ext(X) \mid \exists f \in ext(Y), (cdr\ e) = f\}$
      $newcard(X) = card(Y) + 1$
      $newsauf(X) = \{l \mid e = (cdr\ l), e \in sauf(Y)\}$
      $newrequired(X) = \{(e, n+1)\forall(e,n) \in required(Y)\}$
    finsi
  finsi
- $Z = cons\ X\ Y$ (`mic-cons` X Y)
  si $ext(Y) \neq \emptyset$ **alors**
    si $\mid ext(Y) \mid= 1$ **alors**
      si $\mid dom(X) \mid= 1$ **alors**
        $newext(Z) = (consval(X)val(Y))$
      sinon
        $newext(Z) = \{(consxval(Y))avecx \in dom(X)\}$
      fin si
    sinon
      $newext(Z) = \{(consxy)avecx \in dom(X), y \in ext(Y)\}$
    fin si
  fin si
  $newdom(Z) = dom(X) \cup dom(Y)$
  $sauf(Z) = \{(consxy)avecx \in dom(X), y \in sauf(Y)\}$
  $card(Z) = card(Y) + 1$
  $required - at(Z) = required - at(Y) + 1$
  $newdom(X) = (dom(Z) \cup dom(Y)) \cap dom(X)$
  si $ext(Z) \neq \emptyset$ **alors**
    $newext(Y) = \{(cdrz)avecz \in ext(Z)\}$
  fin si $newdom(Y) = (dom(Z) \cup dom(X)) \cap dom(Y)$
  $sauf(Y) = \{(cdrz)avecz \in sauf(Y)\}$
  $card(Y) = card(Z) - 1$
  required-at(Y) = required-at(Z) - 1
- $Z = cons\ X\ Y$ (`mic-nth` X Y)
  si $ext(Y) \neq \emptyset$ **alors**
    si $dom(X) fini$ **alors**
      $newdom(Z) = \{(nthxy)avecx \in dom(X), y \in ext(Y)\}$
    fin si
  fin si
  $newdom(Z) = dom(Y)$
  $newdom(X) = [0, max(card(Y))]$
  si $dom(Z) fini$ **alors**
    si $dom(X) fini$ **alors**
      $newext(Y) = \{y \mid (nthxy) = zavecx \in dom(X), y \in ext(Y), z \in dom(Z)\}$

fin si
fin si
$card(Y) = max(dom(X))$

- $Y = lastn\ n\ X$ (`mic-lastn` n X)
  si $ext(X) \neq \emptyset$ **alors**
  $newdom(Y) = \{e \mid e = (lastn\ n\ l), l \in ext(X)\}$
  finsi
  $newdom(Y) = dom(X) \cap dom(Y)$
  $newcard(Y) = n$
  $newsauf(Y) = \{e \mid e = (lastn\ n\ l), l \in sauf(X)\}$
  si $ext(Y) \neq \emptyset$ **alors**
  si $ext(Y) \neq \emptyset$ **alors**
  $newext(X) = \{e \in ext(X) \mid \exists f \in ext(Y)(lastn\ n\ e) = f\}$
  finsi
  finsi
- $Y = firstn\ n\ X$ (`mic-firstn` n X)
  si $ext(X) \neq \emptyset$ **alors**
  $newdom(Y) = \{e \mid e = (firstn\ n\ l), l \in ext(X)\}$
  finsi
  $newdom(Y) = dom(X) \cap dom(Y)$
  $newcard(Y) = n$
  $newsauf(Y) = \{e \mid e = (firstn\ n\ l), l \in sauf(X)\}$
  si $ext(Y) \neq \emptyset$ **alors**
  si $ext(X) \neq \emptyset$ **alors**
  $newext(X) = \{e \in ext(X) \mid \exists f \in ext(Y)(firstn\ n\ e) = f\}$
  finsi
  finsi
- $Y = card\ X$ (`mic-card` X)
  $newdom(Y) = card(X)$
$newcard(X) = dom(Y)$
- $Y = length\ X$ (`mic-length` X)
  $newdom(Y) = card(X)$
$newcard(X) = dom(Y)$
- $Y = max\ X$ (`mic-max` X)
  si $ext(X) \neq \emptyset$ **alors**
  $newdom(Y) = \{e \mid e = max(l), l \in ext(X)\}$
  sinon
  $newdom(Y) = dom(X) \cap dom(Y)$
  finsi
  si $ext(Y) \neq \emptyset$ **alors**
  si $ext(X) \neq \emptyset$ **alors**
  $newext(X) = \{e \mid \exists f \in ext(Y), max(e) = f\}$
  finsi
  finsi
- $Y = min\ X$ (`mic-min` X)
  si $ext(X) \neq \emptyset$ **alors**
  $newdom(Y) = \{e \mid e = min(l), l \in ext(X)\}$
  sinon
  $newdom(Y) = dom(X) \cap dom(Y)$
  finsi
  si $ext(Y) \neq \emptyset$ **alors**
  si $ext(X) \neq \emptyset$ **alors**
  $newext(X) = \{e \mid \exists f \in ext(Y), min(e) = f\}$
  finsi
  finsi
- $Y = apply\ f\ X$ (`mic-apply-f` f X)
  si $ext(X) \neq \emptyset$ **alors**
  $newdom(Y) = \{e \mid e = (apply\ f\ l),\ l \in ext(X)\}$
  finsi
  si $ext(Y) \neq \emptyset$ **alors**
  si $ext(X) \neq \emptyset$ **alors**
  $newext(X) = \{l \in ext(X) \mid \exists e \in dom(Y) e = (applyfl)\}$
  finsi
  finsi
- $Z = map\ f\ X\ Y$ (`mic-map-f` f X Y)
  si $ext(X) \neq \emptyset$ **alors**
  $newext(Z) = \{e \mid e = (map\ f\ l\ val(Y)), \forall l \in ext(X)\}$
  finsi

si $dom(X)\ fini$ **alors**
$newdom(Z) = \{f(d, val(Y), d \in dom(X)\}$
finsi
$newsauf(Z) = \{(mapfeval(Y)), e \in sauf(Y)\}$
$newcard(Z) = card(X)$
si $ext(Y) \neq \emptyset$ **alors**
si $ext(X) \neq \emptyset$ **alors**
$newext(X) = \{e \in ext(X) \mid (map\ f\ e\ val(Y)) = ext(Y)\}$
finsi
finsi

- $Z = select\ f\ X\ Y$ (`mic-select-f` f X Y)

si $ext(X) \neq \emptyset$ **alors**
$newext(Z) = \{e \mid \forall x \in e(fxval(Y)) = true, \forall l \in ext(X) \mid e \in l\}$
finsi
si $dom(X)\ fini$ **alors**
$newdom(Z) = \{d \mid f(d, val(Y)) = true, d \in dom(X)\}$
finsi
$newcard(Z) = [0, max(card(X))]$
si $ext(Y) \neq \emptyset$ **alors**
si $ext(X) \neq \emptyset$ **alors**
$newext(X) = \{e \in ext(X) \mid (selectfeval(Y)) \in ext(Z)\}$
finsi
finsi

- $Z = combin\ f\ X\ Y$ (`mic-combin-f` f X Y)

si $ext(X) \neq \emptyset$ **alors**
si $ext(Y) \neq \emptyset$ **alors**
$newext(Z) = \{(combin\ f\ e\ l),\ e \in ext(X), l \in ext(Y)\}$
finsi
finsi
si $ext(Z) \neq \emptyset$ **alors**
si $ext(Y) \neq \emptyset$ **alors**
$newext(X) = \{e \in ext(X) \mid \exists l \in ext(Y)(combin\ f\ e\ l) \in ext(Z)\}$
finsi
si $dom(X) fini$ **alors**
si $dom(Y) fini$ **alors**
$newdom(X) = \{f(a, b)a \in dom(X), b \in dom(Y)\}$
finsi
finsi
finsi
si $ext(Z) \neq \emptyset$ **alors**
si $ext(X) \neq \emptyset$ **alors**
$newext(Y) = \{e \in ext(Y) \mid \exists l \in ext(X)(combinfle)) \in ext(Z)\}$
finsi
si $dom(X) fini$ **alors**
si $dom(Y) fini$ **alors**
$newdom(Y) = \{f(a, b)a \in dom(X), b \in dom(Y)\}$
finsi
finsi
finsi

- $Z = X \cup Y$ (`mic-union` X Y)

si $ext(X) \neq \emptyset$ **et** $ext(Y) \neq \emptyset$ **alors**
$newext(Z) = (ext(X) \cup ext(Y)) \cap ext(Z)$
finsi
$newdom(Z) = (dom(X) \cup dom(Y)) \cap dom(Z)$
$newcard(Z) = (card(X) \cup card(Y)) \cap card(Z)$
$newsauf(Z) = (sauf(X) \cup sauf(Y)) \cap sauf(Z)$
$newrequired(Z) = (required(X) \cup required(Y)) \cap required(Z)$
si $ext(Z) \neq \emptyset$ **et** $ext(Y) \neq \emptyset$ **alors**
$newext(X) = (ext(Z) - ext(Y)) \cap ext(X)$
finsi
$newdom(X) = (dom(Z) - dom(Y)) \cap dom(X)$
$newcard(X) = (card(Z) - card(Y)) \cap card(X)$
$newsauf(X) = (sauf(Z) - sauf(Y)) \cap sauf(X)$
$newrequired(X) = (required(Z) - required(Y)) \cap required(X)$
si $ext(Z) \neq \emptyset$ **et** $ext(X) \neq \emptyset$ **alors**
$newext(X) = (ext(Z) - ext(X)) \cap ext(Y)$
finsi
$newdom(Y) = (dom(Z) - dom(X)) \cap dom(Y)$
$newcard(Y) = (card(Z) - card(X)) \cap card(Y)$

$newsauf(Y) = (sauf(Z) - sauf(X)) \cap sauf(Y)$
$newrequired(Y) = (required(Z) - required(X)) \cap required(Y)$

- $Z = append\ XY$ (`mic-append` X Y)
  même principe que (`mic-union` X Y)
- $Z = X \cap Y$ (`mic-inter` X Y)
  si $ext(X) \neq \emptyset$ **et** $ext(Y) \neq \emptyset$ **alors**
  $newext(Z) = (ext(X) \cap ext(Y)) \cap ext(Z)$
  finsi
  $newdom(Z) = (dom(X) \cap dom(Y)) \cap dom(Z)$
  $newcard(Z) = (card(X) \cap card(Y)) \cap card(Z)$
  $newsauf(Z) = (sauf(X) \cap sauf(Y)) \cap sauf(Z)$
  $newrequired(Z) = (required(X) \cap required(Y)) \cap required(Z)$
  si $ext(Z) \neq \emptyset$ **et** $ext(Y) \neq \emptyset$ **alors**
  $newext(X) = (ext(X) - (ext(X) \cap ext(Y) - ext(Z)))$
  finsi
  $newdom(X) = (dom(X) - (dom(X) \cap dom(Y) - dom(Z)))$
  $newcard(X) = (card(X) - (card(X) \cap card(Y) - card(Z)))$
  $newsauf(X) = (sauf(X) - (sauf(X) \cap sauf(Y) - sauf(Z)))$
  $newrequired(X) = (required(X) - (required(X) \cap required(Y) - required(Z)))$
  si $ext(Z) \neq \emptyset$ **et** $ext(X) \neq \emptyset$ **alors**
  $newext(Y) = (ext(Y) - (ext(Y) \cap ext(Y) - ext(Z)))$
  finsi
  $newdom(Y) = (dom(Y) - (dom(X) \cap dom(Y) - dom(Z)))$
  $newcard(Y) = (card(Y) - (card(X) \cap card(Y) - card(Z)))$
  $newsauf(Y) = (sauf(Y) - (sauf(X) \cap sauf(Y) - sauf(Z)))$
  $newrequired(Y) = (required(Y) - (required(X) \cap required(Y) - required(Z)))$
- $Z = X - Y$ (`mic-diff` X Y)
  équivalent à $X = Z \cup Y$
- $Z = nth\ X\ Y$ (`mic-nth` X Y)
  si $ext(Y) \neq \emptyset$ **alors**
  si $dom(X)\ fini$ **alors**
  $newdom(Z) = \{(nthxy), x \in dom(X), y \in ext(Y)\}$
  finsi
  sinon
  $newdom(Z) = dom(Y) \cap dom(Z)$
  finsi
  $newdom(X) = (card(Y) - [1,1]) \cap dom(X)$
  si $ext(Y) \neq \emptyset$ **alors**
  si $dom(X)\ fini$ **alors**
  $newdom(Y) = \{y \in ext(Y) \mid \exists z \in dom(Z) \mid z = (nthxy)\}$
  finsi
  finsi
  si $\mid dom(Z) \mid = 1$ **et** $\mid dom(X) \mid = 1$ **alors**
  $newrequired - at(Y) = required - at(Y) \cup (val(Z), val(X))$
  finsi

A.1.3 Les multi-contraintes

A.1.3.1 Semi-méta-contraintes principales

- $every\ c\ X\ Y$ (`mic-every-c` c X Y)
  $\forall x \in X (c\ x\ val(Y))$
- $all\ c\ X$ (`mic-all-c` c X)
  $\forall (a,b) \in X^2, (c\ a\ b)$
- $compar\ c\ X\ Y$ (`mic-compar-c` c X Y)
  $\forall a \in X, b \in Y \mid rank(a) = rank(b), (c\ a\ b)$

A.1.3.2 Semi-méta-contraintes secondaires

- $Z = map\ c\ X\ Y$ (`mic-map-c` c X Y)
  $newext(Z) = \{(c\ a\ val(Y)), \forall a \in X\}$
- $Z = apply\ c\ X$ (`mic-apply-c` c X)
  $newdom(Z) = (c ...(c(c(cx_0x_1)x_2)x_3)...x_n)$
  ou
  $newext(Z) = (apply\ c\ X)$
- $Z = select\ c\ X\ Y$ (`mic-select-c` c X Y)
  $newext(Z) = \{a \mid (c\ a\ val(Y)) = vrai, \forall a \in X\}$

- $Z = combin\ c\ X\ Y$ (`mic-combin-c c X Y`)
  $newext(Z) = \{(c\ a\ b), \forall a \in X, b \in Y \mid rank(a) = rank(b)\}$

A.2 Calcul de bornes avec bornes infinies

Nous reportons ici les résultats des calculs de bornes s'effectuant avec des bornes infinies.

- $+$
  $+\infty + a = a + +\infty = +\infty, \forall a \in \{\boldsymbol{R}^+, \boldsymbol{N}^+\} \cup \{+\infty\}$
  $-\infty + a = a + -\infty = -\infty, \forall a \in \{\boldsymbol{R}^-, \boldsymbol{N}^-\} \cup \{-\infty\}$
  (les cas $+\infty + -\infty$ ou $-\infty + +\infty$ ne peuvent se produire).
- $-$
  $+\infty - a = +\infty, \forall a \in \{\boldsymbol{R}^-, \boldsymbol{N}^-\} \cup \{-\infty\}$
  $a - +\infty = -\infty, \forall a \in \{\boldsymbol{R}^-, \boldsymbol{N}^-\} \cup \{-\infty\}$
  $-\infty - a = -\infty, \forall a \in \{\boldsymbol{R}^+, \boldsymbol{N}^+\} \cup \{+\infty\}$
  $a - -\infty = +\infty, \forall a \in \{\boldsymbol{R}^+, \boldsymbol{N}^+\} \cup \{+\infty\}$
  (les cas $+\infty - +\infty$ ou $-\infty - +\infty$ ne peuvent se produire).
- $*$
  $+\infty * a = a * +\infty = (sign(a)) * \infty, \forall a \in \{\boldsymbol{R}, \boldsymbol{N}\} \cup \{+\infty, -\infty\}$ et $\mid a \mid \geq 1$
  $-\infty * a = a * -\infty = -(sign(a)) * \infty, \forall a \in \{\boldsymbol{R}, \boldsymbol{N}\} \cup \{+\infty, -\infty\}$ et $\mid a \mid \geq 1$

  $+\infty * 0 = 0 * +\infty = 0$
  $-\infty * 0 = 0 * -\infty = 0$
- $/$
  $+\infty / a = (sign(a)) * \infty, \forall a \in \{\boldsymbol{R}, \boldsymbol{N}\}\{+\infty, -\infty\}$ et $\mid a \mid \leq 1$ et $a \neq 0$
  $a / + \infty = (sign(a)) * 0, \forall a \in \{\boldsymbol{R}, \boldsymbol{N}\}\{-\infty, +\infty\}$
  $-\infty / a = -(sign(a)) * 1 * \infty, \forall a \in \{\boldsymbol{R}, \boldsymbol{N}\}\{+\infty, -\infty\}$ et $\mid a \mid \leq 1$ et $a \neq 0$
  $a / - \infty = -(sign(a)) * 0 \forall a \in \{\boldsymbol{R}, nit\}\{-\infty, +\infty\}$
  $+\infty / + \infty =$ indéfini
  $+\infty / - \infty =$ indéfini
  $-\infty / - \infty =$ indéfini
  $-\infty / + \infty =$ indéfini
- *power*
  $+\infty^a = +\infty$ si $a \geq 1$
  $+\infty^a = 0^+$ si $a < 0$
  $a^{+\infty} = +\infty$ si $a \geq 1$
  $a^{+\infty} =$indéfini si $a < 0$
  $a^{+\infty} = 0^+$ si $a > 0$ et $a < 1$
  $+\infty^0 = 1$
  $-\infty^0 = 1$
  $0^{+\infty} = 0$
  $0^{-\infty} =$ indéfini
  $-\infty^a = 0^+$ si $a \leq -1$ et a pair
  $-\infty^a = 0^-$ si $a \leq -1$ et a impair
  $-\infty^a = -\infty$ si $a \geq 1$ et a impair
  $-\infty^a = +\infty$ si $a \geq 1$ et a pair
  $a^{-\infty} = 0^+$ si $a \geq 1$
  $a^{-\infty} = +\infty$ si $a < 1$ et $a > 0$
  $a^{-\infty} =$ indéfini si $a < 0$
  $0^{-\infty} =$ indéfini
  $+\infty^{+\infty} = +\infty$
  $+\infty^{-\infty} = 0^+$
  $-\infty^{+\infty} = indefini$
  $-\infty^{-\infty} = indefini$

Dans MICRO, on impose que le domaine de Y dans $(mic - powerXY)$ soit fini

- $-$ (opposé)
  $-(+\infty) = -\infty$
  $-(-\infty) = +\infty$
- *exp*
  $exp(+\infty) = +\infty$
  $exp(-\infty) = 0^+$

Annexe B

Interface fonctionnelle

B.1 L'interface fonctionnelle de Micro

Afin de créer, de supprimer, de modifier et de résoudre des CSP, MICRO propose un ensemble de fonctions que nous décrivons ici.

B.1.1 Fonctions pour les variables

- La création d'une variable monovaluée est réalisée par la fonction :
  (`mic-create-var` *var-name var-type var-dom*) → VCM
  avec
  var-type ::= boolean | integer | real
  var-dom ::= int-list | val-list | nil
  int-list ::= '(' { int }$^+$ ')'
  val-list ::= '(' { val }$^+$ ')'
  int ::= '("(' val borne ')' '(' val borne ')")'
  borne ::= t | nil
  val ::= valeur
- La création d'une variable multivaluée de type ensemble est réalisée par la fonction :
  (`mic-create-var-set` *var-name var-type var-ext var-dom var-sauf var-card var-required*) → VCM
  avec
  var-type ::= boolean | integer | real
  var-ext ::= set-list | nil
  var-dom ::= int-list | val-list | nil
  var-sauf ::= set-list
  var-card ::= var-dom
  var-required ::= val-list
  set-list ::= '(' { set }$^+$ ')'
  set ::= val-list
  int-list ::= '(' { int }$^+$ ')'
  val-list ::= '(' { val }$^+$ ')'
  int ::= '("(' val borne ')' '(' val borne ')")'
  borne ::= t | nil
  val ::= valeur
- La création d'une variable multivaluée de type liste est réalisée par la fonction :
  (`mic-create-var-set` *var-name var-type var-ext var-dom var-sauf var-card var-required var-required-at*) → VCM
  avec
  var-type ::= boolean | integer | real
  var-ext ::= set-list | nil
  var-dom ::= int-list | val-list | nil
  var-sauf ::= set-list
  var-card ::= var-dom
  var-required ::= val-list
  var-required-at ::= required-at-list
  required-at-list ::= '(' { required-at }$^+$ ')'
  required-at ::= '(' val position ')'
  set-list ::= '(' { set }$^+$ ')'
  set ::= val-list
  int-list ::= '(' { int }$^+$ ')'
  val-list ::= '(' { val }$^+$ ')'
  int ::= '("(' val borne ')' '(' val borne ')")'

borne ::= t | nil
val ::= valeur
position ::= integer

- La suppression d'une variable est réalisée par la fonction :
  (`mic-rem-var` *var-name*) → boolean
- La modification du domaine d'une variable monovaluée est réalisée par la fonction :
  (`mic-set-dom-var` *var-name var-dom*) → boolean
- La modification de l'extension d'une variable multivaluée est réalisée par la fonction :
  (`mic-set-ext-var` *var-name var-ext*) → boolean
- La modification du domaine des éléments d'une variable multivalué est réalisée par la fonction :
  (`mic-set-dom-var` *var-name var-dom*) → boolean
- La modification de l'ensemble des valeurs impossibles d'une variable multivaluée est réalisée par la fonction :
  (`mic-set-sauf-var` *var-name var-sauf*) → boolean
- La modification de la cardinalité d'une variable multivaluée est réalisée par la fonction :
  (`mic-set-card-var` *var-name var-card*) → boolean
- La modification de l'ensemble des éléments requis d'une variable multivaluée est réalisée par la fonction :
  (`mic-set-required-var` *var-name var-required*) → boolean
- La modification de l'ensemble des éléments requis à un certain rang d'une variable multivaluée est réalisée par la fonction :
  (`mic-set-required-at-var` *var-name var-required-at*) → boolean
- L'affectation d'une valeur à une variable est réalisée par la fonction :
  (`mic-set-value` *var-name val*) → boolean
- La consultation d'une variable est réalisée par la fonction :
  (`mic-get-var` *var-name*) → VCM
- Le type d'une variable est donné par la fonction :
  (`mic-get-type-var` *var-name*) → var-type
- Le domaine d'une variable monovaluée est donné par la fonction :
  (`mic-get-dom-var` *var-name*) → var-dom
- L'extension d'une variable multivaluée est donnée par la fonction :
  (`mic-get-ext-var` *var-name var-ext*) → boolean
- Le domaine des éléments d'une variable multivalué est donné par la fonction :
  (`mic-get-dom-var` *var-name var-dom*) → boolean
- L'ensemble des valeurs impossibles d'une variable multivaluée est donné par la fonction :
  (`mic-get-sauf-var` *var-name var-sauf*) → boolean
- La cardinalité d'une variable multivaluée est donnée par la fonction :
  (`mic-get-card-var` *var-name var-card*) → boolean
- L'ensemble des éléments requis d'une variable multivaluée est donné par la fonction :
  (`mic-get-required-var` *var-name var-required*) → boolean
- L'ensemble des éléments requis à un certain rang d'une variable multivaluée est donné par la fonction :
  (`mic-get-required-at-var` *var-name var-required-at*) → boolean
- La valeur d'une variable est donnée par la fonction :
  (`mic-get-value-var` *var-name*) → var-dom
- L'ensemble des contraintes d'une variable est donné par la fonction :
  (`mic-get-constr-var` *var-name*) → princ-constr-expr-list
- La liste des variables déclarées est donnée par la fonction :
  (`mic-var-list`) → var-list

B.1.2 Fonctions pour les contraintes

- La définition d'une nouvelle contrainte est réalisée par la fonction :
  (`mic-create-constr` *constr-name arg-list princ-constr-expr*) → princ-constr-expr
  avec
  arg-list ::= '('{ arg }$^+$ ')'
  arg ::= var-name
  où *princ-constr-expr* est formée à partir de la fonction :
  (`mic-princ-constr-expr` *princ-constr-name sec-constr-expr*) → princ-constr-expr
  où *sec-constr-expr* est formée à partir de la fonction :
  (`mic-sec-constr-expr` *sec-constr-name sec-constr-arg-list*) → sec-constr-expr
  où
  sec-constr-arg-list ::= '('{ sec-constr-arg }$^+$ ')'
  sec-constr-arg ::= sec-contr-expr | VCM | val
- L'ajout d'une contrainte est réalisé par la fonction :
  (`mic-add-constr` *constr-name princ-constr-expr*) → boolean
- La suppression d'une contrainte est réalisée par la fonction :
  (`mic-rem-constr` *constr-name*) → boolean
- La consultation d'une contrainte est réalisée par la fonction :
  (`mic-get-constr` *constr-name*) → VCM
- Le contexte de pose d'une contrainte est donné par la fonction :
  (`mic-get-ctxt-constr` *constr-name*) → ctxt
  avec :
  ctxt ::= '(' { '(' var-name var-dom ')' } ')'

- L'ensemble des variables d'une contrainte est donné par la fonction :
  (`mic-get-var-constr` *constr-name*) → VCM-list
- La liste des contraintes déclarées est donnée par la fonction :
  (`mic-constr-list`) → constr-list

B.1.3 Fonctions sur les CSP

- L'établissement de la consistance locale d'un CSP est réalisé par la fonction :
  (`mic-propagate` *princ-constr-expr-list*) → boolean
- La résolution d'un CSP est réalisée par la fonction :
  (`mic-solve` *var-name-list static-ord-list dynamic-ord*) → solution-list
  avec : static-ord-list ::= '(' { static-ord }$^+$ ')'
  static-ord ::= 'min-width' | 'min-card' | 'dom-min' | 'dom-max'
  dynamic-ord ::= 'first-fail-principle' solution-list ::= '(' { solution }$^+$ ')'
  solution ::= '(' { '(' var-name var-dom ')' }$^+$ ')'

B.2 Les contraintes et l'API de Tropes

L'interface fonctionnelle de TROPES (ou API) permet de créer et de manipuler des bases de connaissances à l'aide d'un ensemble de primitives qui forme un langage d'interactions avec TROPES.

L'intégration de contraintes à TROPES a permis d'étendre l'API vers les fonctions de création et de suppression de contraintes à trois niveaux de représentation. Six fonctions sont donc disponibles :

- (**tr-concept-add-constraint** *concept constraint-expr*) qui ajoute une contrainte de concept.
- (**tr-concept-rem-constraint** *concept constraint-expr*) qui supprime une contrainte de concept.
- (**tr-class-add-constraint** *class constraint-expr*) qui ajoute une contrainte de classe.
- (**tr-class-rem-constraint** *class constraint-expr*) qui supprime une contrainte de classe.
- (**tr-object-add-constraint** *object constraint-expr*) qui ajoute une contrainte d'instance.
- (**tr-object-rem-constraint** *object constraint-expr*) qui supprime une contrainte d'instance.

où *constraint-expr* est une expression de contrainte principale construite à partir des fonctions de contraintes principales de MICRO dont les arguments sont, soit des constantes, soit des accès, soit des expressions de contraintes secondaires. Ces dernières sont construites à partir des fonctions de contraintes secondaires de MICRO dont les arguments sont, soit des constantes, soit des accès, soit des expressions de contraintes secondaires.

Ces fonctions sont la partie visible de l'iceberg ; les fonctions de l'interface TROPES/MICRO font le lien entre l'API de TROPES et celui de MICRO.

La fonction (**tr-set-value** *object slotname value rest:views*) correspond à la modification d'un ACT et agit également sur les CSP de TROPES maintenus par MICRO.

Annexe C

Algorithmes

Nous donnons ici les procédures utilisées dans l'algorithme de résolution de MICRO (*cf.* section 8.3).

Fonction coupe-cycle(X, C) : ensemble de variables
début
    coupe-cycle ← ∅
    pour tout $x \in X$ **faire**
        si dans-cycle(x, X, C) **alors**
            coupe-cycle ← coupe-cycle ∪$\{x\}$
            X ← X - x
            C ← C - { C ∩ contr(x)}
        finsi
    fin pour
fin

TAB. C.1 - : Fonction de détermination d'un ensemble de variables coupe-cycle du graphe associé aux CSP.

Fonction dans-cycle(X, C) : *booleen*
début
    cycle ← faux
    VISIT ← ∅
    C' ← contr(x) ∩ C
    tant que C' ≠ ∅ **et** ¬ cycle **faire**
        c ← premier(C')
        C' ← C' - c
        si c ∉ VISIT **alors**
            VISIT ← VISIT ∪{c}
            V ← var(c) - x
            tant que V ≠ ∅ **et** ¬ cycle **faire**
                y ← premier(V)
                V ← V - y
                cycle ← atteint$(x, y, VISIT)$
            fin tant que
        finsi
    fin tant que
    dans-cycle ← cycle
fin

TAB. C.2 - : Fonction de test d'appartenance à un cycle.

Fonction atteint$(x, y, VISIT)$: *booleen*
début
    atteint ← faux
    C' ← contr(y) ∩ C

tant que C' $\neq \emptyset$ **et** $\neg$ atteint **faire**
c $\leftarrow$ premier(C')
C' $\leftarrow$ C' - c
si c $\notin$ VISIT **faire**
VISIT $\leftarrow$ VISIT $\cup$\{c\}
V $\leftarrow$ var(c) - x
tant que V $\neq \emptyset$ **et** $\neg$ atteint **faire**
z $\leftarrow$ premier(V)
V $\leftarrow$ V - z
si z $=$ x **alors**
atteint $\leftarrow$ vrai
sinon
atteint $\leftarrow$ atteint($x, z, VISIT$)
finsi
fin tant que
finsi
fin tant que
atteint $\leftarrow$ cycle
fin

TAB. C.3 - : Fonction de parcours du graphe.

Procédure Full-Look-Ahead(X, D, C, s, S)
Entrées :
X ensemble des variables du CSP
D ensemble des domaines du CSP
C ensemble des contraintes du CSP
s solution courante
S ensemble des solutions
début
si $X = \emptyset$ **alors**
% il n'y a plus de variable à instancier, la solution courante est ajoutée %
% à l'ensemble des solutions (il se peut qu'il n'y en ait pas) %
$S \leftarrow S \cup s$
sinon % il reste des variables à instancier %

$x \leftarrow$ premier(X)
% on prend la première %
répeter
$v \leftarrow$ premier($dom(x)$)
% on prend la première valeur de son domaine %
$dom(x) \leftarrow dom(x) - v$
% cette valeur est supprimée du domaine %
sauver-domaines(D)
% on sauve l'ensemble des domaines courants du CSP %
si consistance-locale($X, (D - dom(x)) \cup \{v\}, C, contr(x), propagees, reduits$) **alors**
% si le CSP où le domaine de la variable ne contient que la valeur choisie est consistant %
$s \leftarrow s \cup (x, v)$
% cette valeur est ajoutée à la solution courante %
Full-Look-Ahead($X - x, ((D - dom(x)) \cup \{v\}) \cap reduits, C, s, S$)
% on poursuit la recherche %
$s \leftarrow s - (x, v)$
% on enlève cette valeur de la solution courante %

finsi
restaurer-domaines(D)
% l'ensemble des domaines courants est restauré %

jusqu'à $dom(x) = \emptyset$
% toutes les valeurs du domaines sont testées %

finsi
fin

TAB. C.4 - : Le *Full-Look-Ahead* de MICRO

Fonction consistance-locale($X, D, C, C', old-propagees, old-reduits$) : booléen
Entrées :
X ensemble des variables du CSP
D ensemble des domaines du CSP
C ensemble des contraintes du CSP
C' ensemble des contraintes à propager
$old-propagees$ liste des contraintes propagées
$old-reduits$ liste des domaines réduits
debut
consistant $\leftarrow$ vrai
tant que $C' \neq \emptyset$ **et** consistant **faire**
$c \leftarrow$ premier(C')
$C \leftarrow C' - c$
consistant $\leftarrow$ propager-contrainte($X, D, C, c, propagees, reduits$)
old-propagees $\leftarrow$ old-propagees $\cup$ propagees
old-reduits $\leftarrow$ old-reduits $\cup$ reduits
fin tant que
fin

TAB. C.5 - : La fonction de consistance locale des CSP MICRO. La fonction *propager-contrainte* est ici modifiée pour fournir les contraintes propagées à partir de c et les domaines réduits.

Annexe D

Description formelle de Tropes

Cette annexe intègre les contraintes dans la description formelle de TROPES proposée par Cécile Capponi.

D.1 Domaines

Nous considérons les alphabets de symboles suivants :

- $\mathcal{B}$ est l'ensemble des symboles de bases de connaissances (dont les éléments sont B, B_1...)
- $\mathcal{O}$ est l'ensemble des symboles d'individus (dont les éléments sont o, o_1...)
- $\mathcal{K}$ est l'ensemble des symboles de concepts (dont les éléments sont K, K_1
- $\mathcal{A}$ est l'ensemble des symboles d'attributs (dont les éléments sont a, a_1...)
- $\mathcal{C}$ est l'ensemble des symboles de classes (dont les éléments sont C, C_1..)
- $\mathcal{V}$ est l'ensemble des symboles de valeurs (dont les éléments sont v, v_1)
- $\mathcal{D}$ est l'ensemble des symboles de domaines de valeurs (dont les éléments sont d, d_1...)
- $\mathcal{I}$ est l'ensemble des symboles d'intervalles de valeurs (dont les éléments sont i, i_1,...)
- $\mathcal{T}$ est l'ensemble des symboles de types abstraits de données (dont les éléments sont T, T_1...)
- $\mathcal{H}$ est l'ensemble des symboles de points de vue (dont les éléments sont P, P_1)
- $\mathcal{R}_n$ est l'ensemble des symboles de contraintes (dont les éléments sont R, R_1...)
- $\mathcal{G}$ est l'ensemble des symboles de constructeurs de types (dont les éléments sont γ, γ_1,...)
- $\mathcal{F}$ est l'ensemble des symboles de conditions des méta-contraintes conditionnelles (dont les éléments sont $cond$, $cond_1$,...)

Soit $\mathcal{E}$ le domaine des objets à modéliser. Soient $\mathcal{E}_1, \cdots, \mathcal{E}_n$ les domaines respectifs des types $T_1, \cdots, T_n$. Nous définissons Δ comme $\Delta = \mathcal{E} \cup (\bigcup \mathcal{E}_i)_{i \geq 1}$.

Soit $\mathcal{I} = \langle \Delta, \|.\|^{\mathcal{I}} \rangle$ une interprétation sur Δ où $\|.\|^{\mathcal{I}}$ est une fonction telle que :

$$\|.\|^{\mathcal{I}} : \begin{cases} \mathcal{B} \rightarrow 2^{\mathcal{E}} \\ \mathcal{O} \rightarrow \mathcal{E} \\ \mathcal{A} \rightarrow (\mathcal{E} \rightarrow \Delta) \\ \mathcal{K} \rightarrow 2^{\mathcal{E}} \\ \mathcal{C} \rightarrow 2^{\mathcal{E}} \\ \mathcal{T} \rightarrow \{\mathcal{E}_i\}_{i \geq 1} \\ \mathcal{V} \rightarrow \Delta \\ \mathcal{I} \rightarrow 2^{\Delta} \\ \mathcal{D} \rightarrow 2^{\Delta} \\ \mathcal{H} \rightarrow 2^{\mathcal{E}} \\ \mathcal{R}_n \rightarrow (\bigotimes^n \Delta) \\ \mathcal{G} \rightarrow (2^{\Delta} \rightarrow \Delta) \\ \mathcal{F} \rightarrow \{vrai, faux, \diamondsuit\} \end{cases} \qquad (D.1)$$

Par exemple, à chaque élément de $\mathcal{K}$, $\Vert.\Vert^{\mathcal{I}}$ associe un sous-ensemble de $\mathcal{E}$.

Nous notons f_γ la fonction qui donne le domaine résultant de l'application du constructeur γ à un domaine. Par exemple, si $\gamma{=}ens$ alors $f_{ens}(D) = 2^D$.

Nous utilisons une notation fonctionnelle pour les attributs :

$$\Vert a\Vert^{\mathcal{I}}(d) = \{e \mid (d,e) \in \Vert a\Vert^{\mathcal{I}}\} \tag{D.2}$$

et pour les constructeurs :

$$\Vert\gamma\Vert^{\mathcal{I}}(d) = f_\gamma(d) \tag{D.3}$$

D.2 Syntaxe abstraite et sémantique extensionnelle

Forme syntaxique abstraite	*Sémantique*
B formée de $K_1, ..., K_n$	$\Vert B\Vert^{\mathcal{I}} \equiv \Vert K_1\Vert^{\mathcal{I}} \cup ... \cup \Vert K_n\Vert^{\mathcal{I}}$
$K.a$ const γ dans $\left\{\begin{array}{c} K' \\ T \end{array}\right\}$	$\Vert K\Vert^{\mathcal{I}} \subseteq \{e \in \mathcal{E} \mid \Vert a\Vert^{\mathcal{I}}(e) \in f_\gamma(\Vert K'\Vert^{\mathcal{I}})\}$ $\Vert K\Vert^{\mathcal{I}} \subseteq \{e \in \mathcal{E} \mid \Vert a\Vert^{\mathcal{I}}(e) \in f_\gamma(\Vert T\Vert^{\mathcal{I}})\}$
C est racine de P	$\Vert C\Vert^{\mathcal{I}} \equiv \Vert P\Vert^{\mathcal{I}}$
C possède a	$\Vert C\Vert^{\mathcal{I}} \subseteq \{e \in \mathcal{E} \mid \Vert a\Vert^{\mathcal{I}}(e) \neq \perp\}$
$C.a\ t$	$\Vert C\Vert^{\mathcal{I}} \subseteq \{e \in \mathcal{E} \mid \Vert a\Vert^{\mathcal{I}}(e) \in \Vert t\Vert^{\mathcal{I}}\}$
$K.R(a_1, \cdots, a_n)$	$\Vert K\Vert^{\mathcal{I}} \subseteq \{e \in \mathcal{E} \mid (\Vert a_1\Vert^{\mathcal{I}}(e), \cdots, \Vert a_n\Vert^{\mathcal{I}}(e)) \in \Vert R\Vert^{\mathcal{I}}\}$
$K\nabla(P_1, \cdots, P_n)$	$\Vert P_1\Vert^{\mathcal{I}} \equiv \cdots \equiv \Vert P_n\Vert^{\mathcal{I}} \equiv \Vert K\Vert^{\mathcal{I}}$
$t \left\{\begin{array}{c} \text{domaine } d \\ \text{domaine } \gamma(d) \\ \text{sauf } d \\ \text{sauf } \gamma(d) \\ \text{intervalle } i_1, \cdots, i_n \\ \text{intervalle } \gamma(i_1, \cdots, i_n) \\ \text{valeur } v \\ \gamma-\text{card } [n_1..n_2] \end{array}\right\}$	$\Vert d\Vert^{\mathcal{I}}$ $\Vert\gamma\Vert^{\mathcal{I}}(\Vert d\Vert^{\mathcal{I}})$ $\Delta \setminus \Vert d\Vert^{\mathcal{I}}$ $\Delta \setminus \Vert\gamma\Vert^{\mathcal{I}}(\Vert d\Vert^{\mathcal{I}})$ $\Vert i_1\Vert^{\mathcal{I}} \cup \cdots \cup \Vert i_n\Vert^{\mathcal{I}}$ $(\bigcup \mathcal{E}_i)_{i\geq 1} \setminus \Vert\gamma\Vert^{\mathcal{I}}(\Vert i_1\Vert^{\mathcal{I}} \cup \cdots \cup \Vert i_n\Vert^{\mathcal{I}})$ $\Vert v\Vert^{\mathcal{I}}$ $\{e \in \Vert\gamma\Vert^{\mathcal{I}}((\bigcup \mathcal{E}_i)_{i\geq 1}) \mid n_1 \leq_{int} \lvert e\rvert_\gamma \leq_{int} n_2\}$
$C.a$ const γ dans $\left\{\begin{array}{c} C_1, \cdots, C_n \\ T \end{array}\right\}$	$\Vert C\Vert^{\mathcal{I}} \subseteq \{e \in \mathcal{E} \mid \Vert a\Vert^{\mathcal{I}}(e) \in \Vert\gamma\Vert^{\mathcal{I}}(\Vert C_1\Vert^{\mathcal{I}} \cap \cdots \cap \Vert C_n\Vert^{\mathcal{I}})\}$ $\Vert C\Vert^{\mathcal{I}} \subseteq \{e \in \mathcal{E} \mid \Vert a\Vert^{\mathcal{I}}(e) \in \Vert\gamma\Vert^{\mathcal{I}}(\Vert T\Vert^{\mathcal{I}})\}$
$C.R(a_1, \cdots, a_n)$	$\Vert C\Vert^{\mathcal{I}} \subseteq \{e \in \mathcal{E} \mid (\Vert a_1\Vert^{\mathcal{I}}(e), \cdots, \Vert a_n\Vert^{\mathcal{I}}(e)) \in \Vert R\Vert^{\mathcal{I}}\}$
$C_1 \leq_\sigma C_2$	$\Vert C_1\Vert^{\mathcal{I}} \subseteq \Vert C_2\Vert^{\mathcal{I}}$
$(C_1, \cdots, C_n) \rhd C$	$\Vert C_1\Vert^{\mathcal{I}} \cap \cdots \cap \Vert C_n\Vert^{\mathcal{I}} \subseteq \Vert C\Vert^{\mathcal{I}}$
$o.R(a_1, \cdots, a_n)$	$\Vert o\Vert^{\mathcal{I}} \in \{e \in \mathcal{E} \mid (\Vert a_1\Vert^{\mathcal{I}}(e), \cdots, \Vert a_n\Vert^{\mathcal{I}}(e)) \in \Vert R\Vert^{\mathcal{I}}\}$

Forme syntaxique abstraite	*Sémantique*
$o \in K$	$\lVert o \rVert^{\mathcal{I}} \in \lVert K \rVert^{\mathcal{I}}$
$o \in C_1 \wedge \cdots \wedge C_n$	$\lVert o \rVert^{\mathcal{I}} \in \lVert C_1 \rVert^{\mathcal{I}} \cap \cdots \cap \lVert C_n \rVert^{\mathcal{I}}$
$o.a = \left\{ \begin{array}{c} v \\ o' \\ ? \end{array} \right\}$	$\lVert a \rVert^{\mathcal{I}}(\lVert o \rVert^{\mathcal{I}}) = \lVert v \rVert^{\mathcal{I}}$ $\lVert a \rVert^{\mathcal{I}}(\lVert o \rVert^{\mathcal{I}}) = \lVert o' \rVert^{\mathcal{I}}$ $\lVert a \rVert^{\mathcal{I}}(\lVert o \rVert^{\mathcal{I}}) = \Diamond$
$d \quad \{v_1, ..., v_n\}$	$\{\lVert v_1 \rVert^{\mathcal{I}}, \cdots, \lVert v_n \rVert^{\mathcal{I}}\}$
$i \quad [v_1..v_2]$	$\{e \in (\bigcup \mathcal{E}_i)_{i \geq 1} \mid v_1 \leq_{T_i} e \leq_{T_i} v_2\}$
v	v
$K.\text{meta-or}(R_1, R_2)$	$\lVert K \rVert^{\mathcal{I}} \subseteq \{e \in \mathcal{E} \mid e.R_1 \cup e.R_2 \neq \emptyset\}$
$C.\text{meta-or}(R_1, R_2)$	$\lVert C \rVert^{\mathcal{I}} \subseteq \{e \in \mathcal{E} \mid e.R_1 \cup e.R_2 \neq \emptyset\}$
$o.\text{meta-or}(R_1, R_2)$	$\lVert o \rVert^{\mathcal{I}} \in \{e \in \mathcal{E} \mid e.R_1 \cup e.R_2 \neq \emptyset\}$
$K.\text{meta-if-then}(cond, R)$	$\lVert K \rVert^{\mathcal{I}} \subseteq \{e \in \mathcal{E} \mid \neg(\lVert cond \rVert^{\mathcal{I}} = vrai) \vee e.R_1 \neq \emptyset\}$
$C.\text{meta-if-then}(cond, R)$	$\lVert C \rVert^{\mathcal{I}} \subseteq \{e \in \mathcal{E} \mid \neg(\lVert cond \rVert^{\mathcal{I}} = vrai) \vee e.R_1 \neq \emptyset\}$
$o.\text{meta-if-then}(cond, R)$	$\lVert o \rVert^{\mathcal{I}} \in \{e \in \mathcal{E} \mid \neg(\lVert cond \rVert^{\mathcal{I}} = vrai) \vee e.R_1 \neq \emptyset\}$
$K.\text{meta-if-then-else}(cond, R_1, R_2)$	$\lVert K \rVert^{\mathcal{I}} \subseteq \{e \in \mathcal{E} \mid \neg(\lVert cond \rVert^{\mathcal{I}} = vrai) \vee \lVert cond \rVert^{\mathcal{I}} = \Diamond \vee e.R_1 \neq \emptyset\}$ $\lVert K \rVert^{\mathcal{I}} \subseteq \{e \in \mathcal{E} \mid \neg(\lVert cond \rVert^{\mathcal{I}} = faux) \vee \lVert cond \rVert^{\mathcal{I}} = \Diamond \vee e.R_2 \neq \emptyset\}$
$C.\text{meta-if-then-else}(cond, R_1, R_2)$	$\lVert C \rVert^{\mathcal{I}} \subseteq \{e \in \mathcal{E} \mid \neg(\lVert cond \rVert^{\mathcal{I}} = vrai) \vee \lVert cond \rVert^{\mathcal{I}} = \Diamond \vee e.R_1 \neq \emptyset\}$ $\lVert C \rVert^{\mathcal{I}} \subseteq \{e \in \mathcal{E} \mid \neg(\lVert cond \rVert^{\mathcal{I}} = faux) \vee \lVert cond \rVert^{\mathcal{I}} = \Diamond \vee e.R_2 \neq \emptyset\}$
$o.\text{meta-if-then-else}(cond, R_1, R_2)$	$\lVert o \rVert^{\mathcal{I}} \in \{e \in \mathcal{E} \mid \neg(\lVert cond \rVert^{\mathcal{I}} = vrai) \vee \lVert cond \rVert^{\mathcal{I}} = \Diamond \vee e.R_1 \neq \emptyset\}$ $\lVert o \rVert^{\mathcal{I}} \in \{e \in \mathcal{E} \mid \neg(\lVert cond \rVert^{\mathcal{I}} = faux) \vee \lVert cond \rVert^{\mathcal{I}} = \Diamond \vee e.R_2 \neq \emptyset\}$

où $\lVert R \rVert^{\mathcal{I}}$ représente l'ensemble des n-uplets de valeurs qui satisfont la contrainte R.

meta-or représente la méta-contrainte disjonctive, meta-if-then et meta-if-then-else, les méta-contraintes conditionnelles.

cond représente une variable (un attribut) ou une contrainte secondaire booléenne. $\lVert cond \rVert^{\mathcal{I}}$ correspond à l'évaluation de *cond* et $\Diamond$ signifie que l'évaluation n'est pas possible (valeur indéfinie).

Bibliographie

[Adam-Nicolle et al.88] A. Adam-Nicolle, B. Victorri, J. Madelaine, C. Porquet et M. Revenu. – *Airelle: rapport de présentation et manuel d'utilisation.* – Les cahiers du LIUC n° 88-3/4, Caen, France, Laboratoire d'Informatique de l'Université de Caen, 1988.

[Aiba et al.92] A. Aiba et R. Hasegawa. – Constraint Logic Programming Systems - CAL, GDCC and Their Constraint Solvers. *FGCS 92*, pp. 113–131. – Tokyo, Japon, 1992.

[Aït-Kaci et al.93] H. Aït-Kaci et A. Podelski. – Towards a meaning of Life. *Jounal of Logic Programming*, vol. 3-4, n° 16, 1993, pp. 195–234.

[Alander85] J. Alander. – On interval arithmetic range approximation methods of polynomials and rational functions. *Computer and Graphics*, vol. 9, n° 4, 1985, pp. 365–372.

[Allen83] J.F. Allen. – Maintaining knowledge about temporal intervals. *Communications of the ACM*, vol. 26, n° 11, novembre 1983, pp. 832–843.

[Avesani et al.90] P. Avesani, A. Perini et F. Ricci. – COOL: An Object System with Constraints. *TOOLS90*, pp. 221–228. – Washington, DC, USA, 1990.

[Baker94] A.B. Baker. – The hazards of fancy backtracking. *AAAI'94*, pp. 288–293. – Seattle, WS, USA, 1994.

[Beaudoin-Lafon et al.91] M. Beaudoin-Lafon et E. Cournarie. – *ALIEN: A Prototype-Based Constraint System.* – Rapport de Recherche n° 662, Laboratoire de Recherche en Informatique, Orsay, France, Université de Paris-Sud, avril 1991.

[Beek92] P. Van Beek. – On the minimality and decomposability of constraint networks. *Proc. of AAAI'92*, pp. 447–452. – San José, CA, USA, 1992.

[Bellicha et al.94] A. Bellicha, C. Capelle, M. Habib, T. Kökeny et M-C. Vilarem. – CSP techniques using partial orders on domain values. *ECAI'94 workshop on constraint reasoning raised by practical applications.* – Amsterdam, Pays-Bas, 1994.

[Bellicha93] A. Bellicha. – Maintenance of Solution in a Dynamic Constraint Satisfaction Problem. *Applications of Artificial Intelligence in Engineering VIII*, pp. 261–274. – 1993.

[Benhamou et al.94] F. Benhamou, D. McAllester et P. VanHentenryck. – *CLP(Intervals) revisited.* – Technical report n° CS-94-18, Providence, RI, USA, Brown University, avril 1994.

[Benhamou et al.95] F. Benhamou et W. Older. – Applying Interval Arithmetic to Real, Integer and Boolean Constraints. *Journal of Logic Programming*, 1995. – (à paraître).

[Benjamin et al.93] M. Benjamin, T. Viana, K. Corbett et A. Silva. – Satisfying multiple rated constraints in a knowledge based decision aid. *9th Conference on Artificial Intelligence Applications*, pp. 227–283. – 1993.

[Berlandier92a] P. Berlandier. – *Étude de mécanismes d'interprétation de contraintes et de leur intégration dans un système à base de connaissances.* – France, Thèse de doctorat, Université de Nice Sophia Antipolis, 1992.

[Berlandier92b] P. Berlandier. – *PROSE: Une boîte à outils fonctionnelle pour l'interprétation de contraintes; guide d'utilisation.* – Rapport Technique n° 145, Unité de Recherche Sophia Antipolis, INRIA, novembre 1992.

[Berlandier94] P. Berlandier. – *Deux variations sur le thème de la consistance d'arcs: maintien et renforcement.* – Rapport technique n° 2426, Sophia-Antipolis, INRIA, décembre 1994.

[Bessière et al.93] C. Bessière et M. O. Cordier. – Arc-consistency and Arc-consistency again. *Eleventh National Conference on Artificial Intelligence.* pp. 108–113. – Washington, DC, USA, juillet 1993.

[Bessière91] C. Bessière. – Arc-consistency in Dynamic Constraint Satisfaction Problems. *Proc. of AAAI'91*, pp. 221–226. – Anaheim, CA, USA, 1991.

[Bessière92] C. Bessière. – *Systèmes à contraintes évolutifs en Intelligence Artificielle.* – Montpellier, France, Thèse de doctorat en informatique, Université des Sciences et Techniques du Languedoc, septembre 1992.

[Bessière94] C. Bessière. – Arc-consistency and Arc-consistency again. *Artificial Intelligence*, vol. 65, 1994, pp. 179–190.

[Bobrow et al.77] D. G. Bobrow et T. Winograd. – An Overview of KRL, a Knowledge Representation Language. *Cognitive Science*, vol. 1, n° 1, 1977, pp. 3–46.

[Bobrow et al.83] D. G. Bobrow et M. Stefik. – *The LOOPS manual : a data and object-oriented programming system for Interlisp.* – Rapport technique n° Memo KB-VLSI-81-13, Xerox-PARC, Palo Alto, CA, USA, Knowledge-Based VLSI Design Group, 1983.

[Borning et al.86] A. Borning et R. Duisberg. – Constraint-Based Tools for Building User Interfaces. *ACM Transactions on Graphics*, vol. 5, n° 4, octobre 1986, pp. 345–374.

[Borning et al.87] A. Borning, R. Duisberg, B. Freeman-Benson, A. Kramer et M. Woolf. – Constraint Hierarchies. *Proc. of OOPSLA'87*, pp. 48–60. – Orlando, FL, USA, 1987.

[Borning81] A. Borning. – The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, vol. 3, n° 4, octobre 1981, pp. 353–387.

[Brachman85] R. J. Brachman. – "I lied about the trees", Or Defaults and Definitions in Knowledge Representation. *AI Magazine*, vol. 3, n° 6, 1985, pp. 80–93.

[Bryant86] R. Bryant. – Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computer*, vol. 35, n° 6, 1986, pp. 677–691.

[Buchberger85] B. Buchberger. – *Gröbner bases : An algorithmic method in polynomial ideal theory*, chap. 6, pp. 184–232. – New-York, NY, USA, D. Reidel Publishing, 1985, *Recent Trends in Multidimensional Systems Theory.*

[Capponi et al.95] C. Capponi, J. Euzenat et J. Gensel. – Objects, types and constraints as classification schemes. *International Conference on Knowledge Re-use, Storage and Efficiency KRUSE95*, pp. 69–73. – Santa Cruz, CA, USA, août 1995.

[Capponi93] C. Capponi. – Classification des classes par les types. *Représentations Par Objets*, éd. par EC2, pp. 215–224. – La Grande Motte, France, 1993.

[Capponi94] C. Capponi. – Interactive class classification using types. *New Approaches in Classification and Data Analysis*, éd. par E. Diday, Y. Lechevallier, M. Schader, P. Bertrand et B. Burtschy, pp. 204–211. – Springer-Verlag, juillet 1994.

[Capponi95] C. Capponi. – *Identification et exploitation des types dans un modèle de connaissances à objets.* – Grenoble, France, Rapport de thèse, Université Joseph Fourier, octobre 1995.

[Cardelli et al.85] L. Cardelli et P. Wegner. – On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, vol. 17, n° 4, décembre 1985, pp. 471–522. – D11.

[Cardelli et al.91] L. Cardelli et J.C. Mitchell. – Operations on records. *Mathematical Structures in Computer Science*, vol. 1, n° 1, mars 1991, pp. 3–48.

[Cardelli84] L. Cardelli. – A semantics of multiple inheritance. *Lecture Notes in Computer Science*, vol. 173, 1984.

[Carré et al.88] B. Carré et G. Comyn. – On Multiple Classification, Point of View and Object Evolution. *Artificial Intelligence and Cognitive Sciences*, éd. par J. Demongeot, T. Hervé, V. Rialle et C. Roche, pp. 49–62. – New-York, Manchester University Press, 1988.

[Carré89] B. Carré. – *Méthodologie orientée-objet pour la représentation des connaissances. Concepts de points de vue, de représentation multiple et évolutive d'objet.* – France, Thèse, Université des Sciences et Techniques de Lille Flandres Artois, 1989.

[Caseau94] Y. Caseau. – Constraint Satisfaction with an Object-Oriented Knowledge Representation Language. *Journal of Applied Intelligence*, vol. 4, 1994, pp. 157–184.

[Chailloux et al.86] J. Chailloux, M. Devin, F. Dupont, J.-M. Hullot, B. Serpette et J. Vuillemin. – *Le-Lisp version 15.2 : the reference Manual.* – INRIA, France, mai 1986.

[Cheeseman et al.91] P. Cheeseman, Kanefsky B. et W. M. Taylor. – Where the really hard problems are. *12th International Joint Conference on Artificial Intelligence*, pp. 331–337. – Sydney, Australie, 1991.

[Cleary87] J.C. Cleary. – Logical arithmetic. *Future Computing System*, vol. 2, n° 2, 1987, pp. 125–149.

[Collins75] G.E. Collins. – Quantifier elimination for real closed fields by cylindrical algebraic decomposition. *2nd GI Conference on Automata Theory and Formal Languages.* pp. 134–183. – Springer-Verlag, 1975.

[Colmerauer90] A. Colmerauer. – An Introduction to PROLOG. *Communications of the ACM*, vol. 33, n° 7, juillet 1990, pp. 69–90.

[Colmerauer93] A. Colmerauer. – *Naive Solving of Non-linear Constraints*, chap. 6, pp. 89–112. – MIT Press, 1993, *Constraint Logic Programming : Selected Research.*

[Cooper et al.94] M.C. Cooper, D.A. Cohen et P.G. Jeavons. – Characterising tractable constraints. *Artificial Intelligence*, vol. 65, 1994, pp. 347–361.

[Dantzig63] G. B. Dantzig. – *Linear Programming and Extensions.* – Princeton University Press, 1963.

[Dassault electronique91] Dassault Electronique, Saint-Cloud, France. – *INTERLOG : Guide d'utilisation*, 1991.

[Davenport et al.93] J.H. Davenport, Y. Siret et E. Tournier. – *Computer-Algebra : Systems and Algorithms for Algebraic Computation.* – New-York, NY, USA, Academic Press, 1993.

[Davenport et al.94] A. Davenport, E. Tsang, J. Wang et K. Zhu. – GENET : a connectionnist architecture for solving constraint satisfaction problems by iterative improvement. *AAAI'94*, pp. 325–330. – 1994.

[David93] P. David. – When functional and bijective constraints make a CSP polynomial. *13th International Joint Conference on Artificial Intelligence*, pp. 224–229. – Chambéry, France, 1993.

[Davis87] H.E. Davis. – *VIEWS : Multiple Perspectives and Structured Objects in a Knowledge Representation Language.* – Cambridge, MA, USA, Bachelor and master of science thesis, MIT, 1987.

[Davis91] L. Davis. – *Handbook of genetic algorithms.* – Van Nostrand Reinhold, 1991.

[Debruyne94] R. Debruyne. – *DnAC-6.* – Rapport Technique n° 94054, Université du Languedoc, Montpellier, France, LIRMM, 1994.

[Dechter et al.87] R. Dechter et J. Pearl. – The cycle-cutset method for improving search performance in AI applications. *3rd IEEE conference on AI applications*, pp. 224–230. – Orlando, Fl, USA, 1987.

[Dechter et al.88a] A. Dechter et R. Dechter. – Belief Maintenance in Dynamic Constraints Networks. *AAAI'88*, pp. 37–42. – Saint Paul, Mn, USA, 1988.

[Dechter et al.88b] R. Dechter et J. Pearl. – Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, vol. 34, 1988, pp. 1–38.

[Dechter et al.89a] R. Dechter et I. Meiri. – Experimental evaluation of preprocessing techniques in constraint satisfaction problems. *Proc. of the 11 th IJCAI*, pp. 271–277. – Detroit, MI, USA, 1989.

[Dechter et al.89b] R. Dechter et J. Pearl. – Tree clustering for constraint networks. *Artificial Intelligence*, vol. 38, 1989, pp. 353–366.

[Dechter et al.91] R. Dechter, J. Pearl et I. Meiri. – Temporal constraint networks. *Artificial In-*

telligence, vol. 49, 1991, pp. 61–95.

[Dechter86] R. Dechter. – Learning while searching in constraint satisfaction problems. *Proc. of AAAI'86*, pp. 178–183. – Philadelphia, PA, USA, 1986.

[Dechter92] R. Dechter. – From local to global consistency. *Artificial Intelligence*, vol. 55, 1992, pp. 87–107.

[Dechter90] R. Dechter. – Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Artificial Intelligence*, vol. 41, 1989/90, pp. 273–312.

[Dekker94] L. Dekker. – *FROME: représentation multiple et classification d'objets avec points de vue.* – Lille, France, Thèse de PhD, Université des Sciences et Techniques de Lille Flandres Artois, 1994.

[Delobel et al.82] C. Delobel et M. Adiba. – *Bases de données et systèmes relationnels.* – Paris, France, Dunod, 1982.

[Dershowitz et al.89] N. Dershowitz et J.-P. Jouannaud. – *Rewrite systems.* – Rapport de Recherche n° RR-478, Université de Paris-Sud, Orsay, France, avril 1989.

[Deville et al.91] Y Deville et P. Van Hentenryck. – An efficient arc-consistency algorithm for a class of CSP problems. *Proc. of the 12 th IJCAI*, pp. 325–330. – Sidney, Australie, 1991.

[Dincbas et al.88] M. Dincbas, P. VanHentenryck, H. Simonis, A. Aggoun, Graf T. et F. Berthier. – The Constraint Logic Programming Language Chip. *International Conference on Fifth Generation Computer Systems.* – Tokio, Japon, 1988.

[Doyle79] J. Doyle. – A Truth Maintenance System. *Artificial Intelligence*, vol. 12, n° 3, 1979, pp. 231–272.

[Ducournau et al.89] R. Ducournau et M. Habib. – La multiplicité de l'héritage dans les langages à objets. *Technique et Science Informatiques*, vol. 1989, n° 1, 1989, pp. 41–62.

[Ducournau88] R. Ducournau. – *YAFOOL version 3.22 manuel de référence.* – SEMA.METRA, Montrouge, France, 1988.

[Ducournau90] R. Ducournau. – *Y3 manuel de référence.* – SEMA GROUP, Montrouge, France, 1990.

[Dugerdil87] P. Dugerdil. – Les mécanismes d'héritage d'OBJLOG : vertical et sélectif multiple avec point de vue. *6ème CARFIA*, pp. 259–273. – Antibes, France, 1987.

[Ely90] J.S. Ely. – *Prospects for Using Variable Precision Interval Software in C++ for Solving some Contemporary Scientific Problems.* – Colombus, OH, USA, Phd thesis, The Ohio State University, 1990.

[Escamilla et al.90] J. Escamilla et P. Jean. – Relationships in an Object Knowledge Representation Model. *2nd International Conference on TOOLS for Aritificial Intelligence.* – Washington, DC, USA, septembre 1990.

[Euzenat87] J. Euzenat. – *Un système de maintenance de la vérité pour une représentation de connaissances centrée-objet.* – France, Mémoire de dea d'informatique, Institut National Polytechnique de Grenoble, 1987.

[Euzenat93a] J. Euzenat. – Définition abstraite de la classification et son application aux taxonomies d'objets. *2ndes journées Représentation par Objets RPO'93*, éd. par EC2, pp. 235–246. – La Grande Motte, France, 1993.

[Euzenat93b] J. Euzenat. – On a purely taxonomic and descriptive meaning for classes. *13th IJCAI Workshop on Object-Based Representation Systems*, pp. 81–92. – Chambéry, France, 1993.

[Euzenat94] J. Euzenat. – Classification dans les représentations par objets : produits de systèmes classificatoires. *9ième congrès Reconnaissance des Formes et Intelligence Artificielle*, éd. par AFCET, pp. 185–196. – Paris, France, janvier 1994.

[Faltings94] B. Faltings. – Arc-consistency for continuous variables. *Artificial Intelligence*, vol. 65, 1994, pp. 363–376.

[Faucher91] C. Faucher. – *Élaboration d'un langage extensible fondé sur les schémas, le langage Objlog+.* – France, Thèse, Université d'Aix-Marseille III, juillet 1991.

[Ferber84] J. Ferber. – MERING : un langage d'acteurs pour la représentation des connaissances et la compréhension du langage naturel. *4ième CARFIA*, pp. 179–189. – Paris, France, 1984.

[Ferber88] J. Ferber. – *PtitLoo Manuel d'utilisation.* – LAFORIA, Université de Paris 6, Paris, France, 1988.

[Fikes et al.85] R. Fikes et T. Keller. – The role of frame-based representation in reasoning. *Communications of the ACM*, vol. 28, n° 9, septembre 1985, pp. 904–920.

[Fornarino et al.90a] M. Fornarino et A. M. Pinna. – *Expression des relations et maintien de la cohérence : le concept de lien.* – Rapport de Recherche n° 1346, Unité de Recherche de Sophia Antipolis, France, INRIA, décembre 1990.

[Fornarino et al.90b] M. Fornarino et A. M. Pinna. – *Un modèle objet logique et relationnel : le langage OTHELO.* – France, Doctorat en informatique, Université de Nice, avril 1990.

[Fornarino91] M. Fornarino. – *Mise en œuvre de l'héritage au moyen de relations.* – Rapport de Recherche n° 1417, Unité de Recherche de Sophia Antipolis, France, INRIA, avril 1991.

[Fox et al.86] M. S. Fox, J. M. Wright et D. Adam. – Experiences with SRL: An Analysis of a Frame-based Knowledge Representation. *Expert Data Systems*, pp. 161–172. – 1986.

[Freeman-Benson et al.90] B. N. Freeman-Benson, J. Maloney et A. Borning. – An Incremental Constraint Solver. *Communications of the ACM*, vol. 33, n° 1, janvier 1990, pp. 54–63.

[Freeman-Benson et al.92] B. N. Freeman-Benson et A. Borning. – Integrating Constraints with an Object-Oriented Language. *Proc. of ECOOP'92.* pp. 268–286. – Utrecht, Pays-Bas, 1992.

[Freeman-Benson89] B. N. Freeman-Benson. – *A Module Mechanism for Constraints in Smalltalk.* – Technical Report n° 89-05-03, Department of Computer Science and Engineering, University of Washington, Seattle, WA, USA, mai 1989. in Proceddings of OOPSLA'89.

[Freeman-Benson90] B. N. Freeman-Benson. – Kaleidoscope: Mixing Objects, Constraints and Imperative Programming. *ACM SIGPLAN Notices ECOOP/OOPSLA'90*, vol. 25, n° 10, octobre 1990, pp. 77–88.

[Freuder et al.92] E. C. Freuder et R. J. Wallace. – Partial Constraint Satisfaction. *Artificial Intelligence*, vol. 58, décembre 1992, pp. 21–70.

[Freuder78] E. C. Freuder. – Synthesizing constraint expressions. *Communications of the ACM*, vol. 21, n° 11, novembre 1978, pp. 958–966.

[Freuder82] E. C. Freuder. – A sufficient condition for backtrack-free search. *Journal of the ACM*, vol. 29, n° 1, 1982, pp. 24–32.

[Freuder90] E. C. Freuder. – Complexity of K-tree structured constraint satisfaction problems. *Proc. of AAAI'90*, pp. 4–9. – Boston, MA, USA, 1990.

[Freuder91] E. C. Freuder. – Eliminating Interchangeable Values in Constraint Satisfaction Problems. *Proc. of AAAI'91*, pp. 227–233. – Anaheim, CA, USA, 1991.

[Frost et al.94a] D. Frost et R. Dechter. – Dead-end driven learning. *Proc. of AAAI'94*, pp. 294–300. – Seattle, WA, USA, 1994.

[Frost et al.94b] D. Frost et R. Dechter. – In search of the best constraint satisfaction search. *Proc. of AAAI'94*, pp. 301–306. – Seattle, WA, USA, 1994.

[Frühwirth et al.92] T. Frühwirth, A. Herold, V. Küchenhoff, T. Le provost, P. Lim, E. Monfroy et M. Wallace. – Constraint Logic Programming - An Informal Introduction. *2nd International Logic Programming Summer School LPSS'92*, éd. par N. E. Fuchs et M. J. Ratcliffe. pp. 3–35. – Zurich, Switzerland, septembre 1992.

[Gaschnig77] J. Gaschnig. – A general backtrack algorithm that eliminates most redundant tests. *Proc. of the 5 th IJCAI*, pp. 457–471. – Cambridge, MA, USA, 1977.

[Gaschnig79] J. Gaschnig. – *Performance measurement and analysis of certain search algorithms.* – Rapport technique n° CMU-CS-79-124, Pittsburg, PA, USA, Carnegie-Mellon University, 1979.

[Geffner et al.87] H. Geffner et J. Pearl. – An improved constraint-propagation algorithm for diagnosis. *10th International Joint Conference on Artificial Intelligence.* – Milan, Italy, 1987.

[Gensel et al.92] J. Gensel et P. Girard. – Expression d'un modèle de tâches à l'aide d'une repré sentation par objets. *Représentation Par Objets*, éd. par EC2, pp. 225–236. – La Grande-Motte, France, juin 1992.

[Gensel et al.94] J. Gensel, P. Girard et O. Schmeltzer. – Intégration de contraintes, d'objets composites et de tâches dans un modèle de représentation par objets. *Neuvième congrès Reconnaissance des Formes et Intelligence Artificielle, RFIA'94*, éd. par AFCET-AFIA, pp. 281–292. – Paris, France, janvier 1994.

[Gensel90] J. Gensel. – *Gestion des dépendances et des hypothèses dans un modèle de connaissances à objets.* – Grenoble, France, dea d'informatique, Institut National Polytechnique de Grenoble, juin 1990.

[Gensel93a] J. Gensel. – Expression et satisfaction de contraintes dans TROPES, un modèle de repré sentation de connaissances par objets. *Représentation Par Objets*, pp. 51–62. – La Grande-Motte, France, juin 1993.

[Gensel93b] J. Gensel. – Integrating Constraints in an Object-Based Knowledge Representation System. *International Workshop on Constraint Processing at CSAM'93*, éd. par M. Meyer. pp. 51–64. – Kaiserslautern, Germany, août 1993.

[Gensel95] J. Gensel. – Integrating constraints in a knowldge representation system. *Constraint Processing*, éd. par M. Meyer, pp. 67–77. – Springer-Verlag, 1995.

[Ginsberg93] M.L. Ginsberg. – Dynamic Backtracking. *JAIR*, vol. 1, 1993, pp. 25–46.

[Girard95] P. Girard. – *Construction hypothétique d'objets complexes.* – Grenoble, France, Rapport de thèse, Université Joseph Fourier, octobre 1995.

[Goldberg83] A. Goldberg. – SMALLTALK-80 *: the Interactive Programming Environment.* – Addison-Wesley, 1983.

[Goldberg89] D.E. Goldberg. – *Genetic algorithm in search, optimization and machine learning.* – Addison-Wesley, 1989.

[Gosling83] J. Gosling. – *Algebraic Constraints.* – Pittsburgh, PA, USA, Phd thesis, Carnegie-Mellon University, 1983.

[Güsgen et al.88] H. Güsgen et J. Hertzberg. – Some Fundamental Properties of Local Constraint Propagation. *Artificial Intelligence*, vol. 36, 1988, pp. 237–247.

[Han et al.88] C. Han et C. Lee. – Comments on Mohr and Henderson's Path Consistency Algorithm. *Artificial Intelligence*, vol. 36, 1988, pp. 125–130.

[Handsen88] E. Handsen. – *An Overview of Global Optimization Using Interval Analysis*, pp. 289–307. – New-York, USA, Academic Press, 1988.

[Haralick et al.80] R. M. Haralick et G. L. Elliot. – Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, vol. 14, 1980, pp. 263–313.

[Harris86] D. R. Harris. – A Hybrid Structured Object and Constraint Representation System. *Proc. of AAAI'86*, pp. 986–990. – Philadelphia, PA, USA, 1986.

[Haselböck93] A. Haselböck. – Exploiting Interchangeabilities in Constraint Satisfaction Problems. *Proc. of the 13 th IJCAI*, pp. 282–287. – Chambéry, France, 1993.

[Haton et al.91] J.P. Haton, N. Bouzid, F. Charpillet, M.C. Haton, B. Lâasri, H. Lâasri, P. Marquis, T. Mondot et A. Napoli. – *Le raisonnement en Intelligence Artificielle.* – Paris, France, 1991.

[Havens et al.92] W. Havens, S. Sidebottom, G. Sidebottom, J. Jones et R. Ovans. – ECHIDNA: A Constraint Logic Programming Shell. *1992 Pacific Rim International Conference on Artificial Intelligence.* – Séoul, Corée du Sud, 1992.

[Hentenryck et al.91] P. Van Hentenryck et T. Le Provost. – Incremental Search in Constraint Logic Programming. *New Generation Computing*, vol. 9, 1991, pp. 257–275.

[Hentenryck et al.93] P. Van Hentenryck, V. Saraswat et Y. Deville. – *The design, Implementation, and Evaluation of the Constraint Language* `cc(FD)`*.* – Technical report n° CS-93-02, Providence, RI, USA, Brown University, janvier 1993.

[Hentenryck89] P. Van Hentenryck. – *Constraint Satisfaction in Logic Programming.* – MIT Press, 1989.

[Hentenryck90] P. Van Hentenryck. – Incremental Constraint Satisfaction in Logic Programming. *ICLP'90*, pp. 189–202. – 1990.

[Hentenryck91] P. Van Hentenryck. – The CLP language CHIP: Constraint Solving and Applications. *36th Computer Society International Conference*, éd. par IEEE, pp. 382–387. – 1991.

[Hong93] H. Hong. – *RISC-CLP(Real): Logic Programming with Non-linear Constraints over the Reals*, chap. 8, pp. 133–160. – MIT Press, 1993, *Constraint Logic Programming: Selected Research*.

[Hubbe et al.92] P.D. Hubbe et E.C. Freuder. – An efficient cross product representation of the constraint satisfaction problem search space. *Proc. of AAAI'92*, pp. 421–427. – San José, CA, USA, 1992.

[Hyvőnen et al.93a] E. Hyvőnen, S. De Pascale et A. Lehtola. – Interval Constraint Programming in C++. *Proceedings NATO ASI on Constraint Programming*, éd. par Brian Mayoh. pp. 343–360. – Parnu, Estonie, août 1993.

[Hyvönen et al.93b] E. Hyvönen, S. DePascale et A. Lehtola. – Interval Constraint Satisfaction Tool INC++. *1993 IEEE International Conference on Tools with AI*, pp. 298–305. – Boston, MA, USA, 1993.

[Hyvönen92] E. Hyvönen. – Constraint reasoning based on interval arithmetic: the tolerance propagation approach. *Artificial Intelligence*, vol. 58, décembre 1992, pp. 71–112.

[Ilog92a] ILOG, Gentilly, France. – LE-LISP *v16 : manuel de référence*, 1992.

[Ilog92b] ILOG, Gentilly, France. – *PECOS : manuel de référence Version 1.1*, 1992.

[Ilog94] ILOG, Gentilly, France. – *Ilog Talk version 3.01 : manuel de référence*, 1994.

[Ingals78] D.H.H. Ingals. – The SMALLTALK-76 Programming System Design and Implementation. *Fifth POPL*, pp. 9–17. – Tucson, AR, USA, 1978.

[Jaffar et al.87] J. Jaffar et J. L. Lassez. – Constraint Logic Programming. *SIGPLAN Notices Symposium on Principles of Programming Languages*, pp. 111–119. – janvier 1987.

[Jaffar et al.91] J. Jaffar, S. Michaylov, P. J. Stuckey et R. H. C. Yap. – The CLP(R) language and System: An Overview. *36th Computer Society International Conference*, éd. par IEEE, pp. 376–381. – 1991.

[Janssen et al.88] P. Janssen et M-C. Vilarem. – *Problèmes de satisfaction de contraintes : techniques de résolution et application à la synthèse de peptides.* – Rapport de Recherche n° 54, France, Centre de Recherche en Informatique de Montpellier, 1988.

[Janssen90] P. Janssen. – *Aide à la conception : une approche basée sur la satisfaction de contraintes.* – Montpellier, France, Thèse de doctorat en informatique, Université des Sciences et Techniques du Languedoc, janvier 1990.

[Jégou91] P. Jégou. – *Contribution à l'étude des problèmes de satisfaction de contraintes : algorithme de propagation et de résolution, propagation de contraintes dans les réseaux dynamiques.* – Montpellier, France, Thèse de doctorat en informatique, Université des Sciences et Techniques du Languedoc, janvier 1991.

[Jégou93] P. Jégou. – Decomposition of domains based on the micro structure of finite constraint satisfaction problems. *Eleventh National Conference on Artificial Intelligence.* pp. 731–736. – Washington, DC, USA, juillet 1993.

[Kim et al.89] W. Kim, E. Bertino et J. Garza. – Composite Objects Revisted. *ACM/SIGMOD International Conference on the Management of Data*, pp. 337–347. – Portland, OR, USA, 1989.

[Kleer86] J. De Kleer. – An Assumption-Based TMS. *Artificial Intelligence*, vol. 28, n° 1, 1986, pp. 127–162.

[Kökény94] T. Kökény. – *Satisfaction de Contraintes dans un Environnement Orienté Objets.* – France, Rapport de thèse, Université de Montpellier II, décembre 1994.

[Lassez et al.93] J-L. Lassez, T. Huyinh et K. McAloon. – *Simplifications and Elimination of*

Redundant Linear Arithmetic Constraints, chap. 5, pp. 73–88. – MIT Press, 1993, *Constraint Logic Programming : Selected Research.*

[Laurière78] J-L. Laurière. – A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, vol. 10, n° 1, 1978, pp. 29–127.

[Lee et al.93] J.H.M. Lee et M.H. VanEmden. – Interval Computation as deduction in CHIP. *Journal of Logic Programming*, vol. 16, 1993, pp. 255–276.

[Leler88] W. Leler. – *Constraint Programming Languages.* – Addison-Wesley, 1988.

[LeSaint94] D. LeSaint. – Maximal Sets of Solution for Constraint Satisfaction Problems. *11th ECAI'94*, éd. par A.G. Cohn. pp. 110–114. – Amsterdam, Pays-Bas, 1994.

[Lhomme93] O. Lhomme. – Consistency Techniques for Numeric CSPs. *13th International Joint Conference on Artificial Intelligence*, pp. 232–238. – Chambéry, France, 1993.

[Liu et al.92] B. Liu et Y. Ku. – Constraint-Lisp: An Object-Oriented Constraint Programming Language. *ACM SIGPLAN Notices*, vol. 27, n° 11, novembre 1992, pp. 17–26.

[Lopez et al.93] G. Lopez, B. Freeman Benseon et A. Borning. – KALEIDOSCOPE *: A Constraint Imperative Programming Language.* – Technical report n° 93-09-04, University of Washington, WA, USA, Department of Computer Science and Engineering, septembre 1993.

[Mackworth et al.85] A.K. Mackworth, J.A. Mulder et W.S. Havens. – Hierarchical arc consistency : exploiting structured domains in constraint satisfaction problems. *Computational Intelligence*, vol. 1, 1985, pp. 118–126.

[Mackworth77] A. K. Mackworth. – Consistency in Networks of Relations. *Artificial Intelligence*, vol. 8, 1977, pp. 99–118.

[Maloney et al.89] J. H. Maloney, A. Borning et B. N. Freeman-Benson. – *Constraint Technology for User-Interface in ThingLab II.* – Technical Report n° 89-05-02, Department of Computer Science and Engineering, University of Washington, Seattle, WA, USA, mai 1989.

[Marcke87] K. Van Marcke. – KRS : An Object-Oriented Representation Language. *Revue d'Intelligence Artificielle*, vol. 1, n° 4, 1987, pp. 43–68.

[Mariño et al.90] O. Mariño, F. Rechenmann et P. Uvietta. – Multiple perspectives and classification mechanism in object-oriented representation. *9th European Conference on Artificial Intelligence*, pp. 425–430. – Stockholm, Suède, août 1990.

[Mariño91] O. Mariño. – Classification d'objets composites dans un système de représentation de connaissances multi-points de vue. *8ième congrès Reconnaissance des Formes et Intelligence Artificielle*, pp. 25–29. – Lyon, France, novembre 1991. Notes pour RFIA'91.

[Mariño93] O. Mariño. – *Raisonnement classificatoire dans une représentation à objets multi-points de vue.* – Grenoble, France, Thèse, Université Joseph Fourier, 1993.

[Masini et al.89] G. Masini, A. Napoli, D. Colnet, D. Léonard et K. Tombre. – *Les langages à objets.* – Paris, France, InterEditions, 1989.

[Mayoh94] B. Mayoh. – Constraint Programming and Artificial Intelligence. *Proceedings NATO ASI on Constraint Programming*, éd. par B.Mayoh, E. Tyugu et J. Penham. pp. 18–53. – Parnu, Estonie, août 1994.

[Michalski et al.84] R.S. Michalski et R.E. Stepp. – Learning from Observation : Conceptual Clustering. *Machine Learning, an Artificial Intelligence Approach*, éd. par R.S. Michalsky, J.G. Carbonell et T.M. Mitchell, pp. 331–363. – Berlin, Germany, Springer-Verlag, 1984.

[Minsky75] M. Minsky. – A Framework for Representing Knowledge. *The Psychology of Computer Vision*, éd. par P. Winston, pp. 211–281. – New-York, NY, USA, McGraw-Hill, 1975.

[Minton et al.90] S. Minton, M. D. Johnston, A. B. Philips et P. Laird. – Solving Large-Scale Constraint Satisfaction and Scheduling Problems Using a Heuristic repair Method. *Proc. of AAAI'90*, pp. 17–24. – Boston, MA, USA, 1990.

[Minton et al.92] S. Minton, M. D. Johnston, A. B. Philips et P. Laird. – Minimizing conflicts : a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, vol. 58, n° 1-3, 1992, pp. 161–205.

[Mitchell et al.92] D. Mitchell, B. Selman et H. Levesque. – Hard and easy distributions of SAT problems. *Proc. of AAAI'92*, pp. 459–465. – San José, CA, USA, 1992.

[Mohr et al.86] R. Mohr et C. Henderson. – Arc and Path Consistency Revisited. *Artificial Intelligence*, vol. 28, 1986, pp. 225–233.

[Mohr et al.88] R. Mohr et G. Masini. – Good old discrete relaxation. *Proc. of the 8 th ECAI*, pp. 651–656. – Munich, Allemagne, 1988.

[Montanari74] U. Montanari. – Networks of Constraints: Fundamental Properties and Applications to Picture Processing. *Information Sciences*, vol. 7, n° 3, 1974, pp. 95–132.

[Moore66] R. Moore. – *Interval Arithmetic.* – Englewoods Cliff, NJ, USA, Prentice-Hall, 1966.

[Moore79] R. Moore. – *Methods and Applications of Interval Analysis.* – Philadelphie, USA, SIAM, 1979, *SIAM Studies in Applied Mathematics.*

[Morris93] P. Morris. – The breakout method for escaping from local minima. *Proc. of AAAI'93*, pp. 40–45. – Washington, DC, USA, 1993.

[Myers et al.92a] B. A. Myers, D. A. Giuse et B. Vander Zanden. – Declarative Programming in a Prototype-Instance System: Object-Oriented Programming Without Writing Methods. *ACM SIGPLAN Notices*, vol. 27, n° 10, octobre 1992, pp. 184–200. – Proceedings of the OOPSLA'92.

[Myers et al.92b] B. A. Myers, A. Mickish et D. A. Giuse. – *GARNET.* – Carnegie Mellon University, novembre 1992. ftp(128.2.242.7).

[Nadel89] B. Nadel. – Constraint satisfaction algorithms. *Computational Intelligence*, vol. 5, 1989, pp. 188–224.

[Napoli et al.91] A. Napoli, C. Laurenço et R. Ducournau. – Techniques de classification pour modéliser la synthèse de molécules organiques. *1st International Confernce on Knowledge Modeling and Expertise Transfer (KMET'91), Sophia Antipolis, France*, éd. par D. Hérin-Aime, R. Dieng, J.P. Regourd et J.P. Angoujard. pp. 241–252. – Amsterdam, Pays-Bas, 1991.

[Napoli92] A. Napoli. – *Représentation à objets et raisonnement par classification en Intelligence Artificielle.* – CRIN-INRIA Lorraine, France, Doctorat d'état ès sciences mathématiques, Université de Nancy I, janvier 1992.

[Neveu et al.94] B. Neveu et P. Berlandier. – Arc-Consistency for Dynamic Constraint Satisfaction Problems: An RMS free Approach. *Workshop on Constraint Satisfaction Issues Raised by Practical Applications at ECAI'94.* – Amsterdam, Pays-Bas, août 1994.

[Older et al.90] W. Older et A. Vellino. – Extending Prolog with constraint arithmetic on real intervals. *Canadian Conference on Electrical and Computer Engineering*, 1990.

[Older et al.93] W. Older et A. Vellino. – *Constraint Arithmetic on Real Interval*, chap. 10, pp. 175–196. – MIT Press, 1993, *Constraint Logic Programming : Selected Research.*

[Oplebodu89] A. Oplebodu. – CHARME : un langage industriel de programmation par contraintes. *Proc. of AVIGNON'89*, éd. par EC2. – Avignon, France, 1989.

[Orsier90] B. Orsier. – *évolution de l'attachement procédural : intégration de tâches, méthodes et procédures dans une représentation de connaissances à objets.* – Grenoble, France, Rapport de dea, Institut National Polytechnique de Grenoble, juin 1990.

[Poncabaré et al.91] T. Poncabaré et F. Rechenmann. – SCAI : un environnement de développement de systèmes à bases de connaissances en calcul scientifique et technique. *Troisième convention intelligence artificielle*, pp. 491–509. – Paris, France, 1991.

[Prosser93] P. Prosser. – Domain filtering can degrade intelligent backjumping. *13th International Joint Conference on Artificial Intelligence*, pp. 262–267. – Chambéry, France, 1993.

[Puget et al.93] J. F. Puget et P. Albert. – SOLVER: Constraints + Objects = Descriptions. *Workshop on Object Representation at the 13th International Joint Conference on Artificial Intelligence*, pp. 93–101. – Chambéry, France, septembre 1993.

[Puget92a] J.-F. Puget. – PECOS : a High Level Constraint Programming Language. *SPICIS92.* – Singapour, septembre 1992.

[Puget92b] J. F. Puget. – Programmation par contraintes orientée objet. *Proc. of AVIGNON'92*, pp. 129–138. – Avignon, France, 1992.

[Puget94] J.-F. Puget. – *A C++ Implementation of CLP.* – Technical report, Gentilly, France, Ilog, janvier 1994.

[Purdom83] P.W. Purdom. – Search rearrangement backtracking and polynomial average time. *Artificial Intelligence*, vol. 21, n° 1-2, 1983, pp. 117–133.

[Quillian68] M.R. Quillian. – Semantic Memory. *Semantic Information Processing*, éd. par M. Minsky, pp. 227–270. – Cambridge, MA, USA, MIT Press, 1968.

[Ratscheck et al.84] H. Ratscheck et J. Rokne. – *Computer Methods for the Range of Functions.* – Chichester, Grande Bretagne, Ellis Horwood, 1984.

[Rechenmann et al.90] F. Rechenmann, P. Fontanille et P. Uvietta. – *SHIRKA : Système de gestion de bases de connaissances centrées-objets.* – INRIA et laboratoire Artémis / IMAG, Grenoble, France, octobre 1990.

[Rechenmann92] F. Rechenmann. – Integrating procedural knowledge: procedural attachment revisited. *25th INRIA anniversary workshop on knowledge representation.* – Paris, France, 1992.

[Rechenmann93] F. Rechenmann. – Integrating procedural and declarative knowledge in object-based knowledge models. *IEEE International Conference on Systems, Man and Cybernetics*, pp. 98–101. – Le Touquet, France, octobre 1993.

[Rieu et al.92] D. Rieu, G. T. Nguyen et J. Escamilla. – SHOOD: An Object Model for Engineering Design. *Proc. of AVIGNON'92*, éd. par EC2, pp. 295–303. – Avignon, France, 1992.

[Roberts et al.77] R.B Roberts et I.P. Goldstein. – *The FRL Manual.* – AI Lab, MIT, Cambridge, MA, USA, 1977.

[Rousseau88] B. Rousseau. – *Vers un environnement de résolution de problèmes en biométrie – apports des techniques de l'intelligence artificielle et de l'interaction graphique.* – Lyon, France, Thèse de biométrie, Université Claude Bernard, 1988.

[Rumbaugh87] J. Rumbaugh. – Relations as Semantic Constructs in an Object-Oriented Language. *Proc. of OOPSLA'87*, pp. 466–481. – Orlando, FL, USA, octobre 1987.

[Sabin et al.94] D. Sabin et E.C. Freuder. – Contradicting Conventional Wisdom in Constraint Satisfaction. *ECAI'94*, pp. 125–129. – Amsterdam, Pays-Bas, 1994.

[Sanella94] M. Sanella. – *Constraint Satisfaction and Debugging for Interactive User Interfaces.* – Ph-D Dissertation n° 94-09-10, University of Washington, WA, USA, Department of Computer Science and Engineering, septembre 1994.

[Schiex et al.93] T. Schiex et G. Verfaillie. – Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. *International Conference on Tools with Artificial Intelligence*, pp. 48–55. – Boston, MA, USA, novembre 1993.

[Seidel81] R. Seidel. – A new method for solving satisfaction problems. *Proc. of the 7th IJCAI*, pp. 338–342. – Vancouver, Canada, 1981.

[Selman et al.92] B. Selman, H. Levesque et D. Mitchell. – A New Method for Solving Hard Satisfiability Problems. *Proc. of AAAI'92*, pp. 440–446. – San José, CA, USA, 1992.

[Sherpa95] SHERPA, IMAG-LIFIA, Grenoble, France. – *Tropes 1.0 : reference manual*, juillet 1995.

[Skelboe74] S. Skelboe. – Computation of rational interval functions. *BIT*, vol. 14, 1974, pp. 87–95.

[Smolka et al.93] G. Smolka, M. Hentz et J. Würtz. – *Object-Oriented Constraint Programming in Oz.* – Research Report n° RR-93-16, Saarbrücken, Germany, DFKI, avril 1993.

[Stallman et al.77] R.M. Stallman et G.J. Sussman. – Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, vol. 9, n° 2, 1977, pp. 135–196.

[Stefik81a] M. Stefik. – Planning and Meta-Planning (MOLGEN: Part 2). *Artificial Intelligence*, vol. 16, 1981, pp. 141–170.

[Stefik81b] M. Stefik. – Planning with Constraints (MOLGEN: Part 1). *Artificial Intelligence*, vol. 16, 1981, pp. 111–140.

[Sussman et al.80] G. J. Sussman et G. L. Steele Jr. – CONSTRAINTS - A Language for Expressing

Almost-Hierarchical Descriptions. *Artificial Intelligence*, vol. 14, 1980, pp. 1–39.

[Sutherland63] I. Sutherland. – SKETCHPAD *: A Man-Machine Graphical Communication System.* – Cambridge, MA, USA, Phd thesis, Massachussetts Institute of Technology, 1963.

[Tayar95] N. Tayar. – *Gestions des versions pour la construction incrémentale et concourante de bases de connaissances.* – Grenoble, France, Rapport de thèse, Université Joseph Fourier, septembre 1995.

[Trombettoni92] G. Trombettoni. – *Conception d'un algorithme de maintien de solutions dans un réseau de contraintes.* – Rapport de Recherche n° 1784, Unité de Recherche Sophia Antipolis, INRIA, novembre 1992.

[Tsang et al.90] E. Tsang et N. Foster. – *Solution synthesis in the constraints satisfaction problem.* – Technical Report n° CSM-142, University of Essex, Grande Bretagne, Department of Computer Science, 1990.

[Tsang93] E. Tsang. – *Foundations of Constraint Satisfaction.* – Academic Press, 1993, *Computation in Cognitive Science.*

[Verfaillie et al.94a] G. Verfaillie et T. Schiex. – Dynamic Backtracking for Dynamic Constraint Satisfaction Problems. *Workshop on Constraint Satisfaction Issues Raised by Practical Applications at ECAI'94.* – Amsterdam, Pays-Bas, août 1994.

[Verfaillie et al.94b] G. Verfaillie et T. Schiex. – Solution Reuse in Dynamic Constraint Satisfaction Problems. *Proc. of AAAI'94*, pp. 307–312. – Seattle, WA, USA, 1994.

[Verfaillie93] G. Verfaillie. – Problèmes de satisfaction de contraintes : production et révision de solution par modifications locales. *Thirteenth International Conference Avignon'93*, éd. par EC2, pp. 277–286. – Avignon, France, mai 1993.

[Wadge et al.85] W.W. Wadge et E.A. Ashcroft. – *Lucid, the Dataflow Programming Language.* – Londres, Grande-Bretagne, Academic Press1985, 1985.

[Wallace et al.93] R. J. Wallace et E. C. Freuder. – Conjunctive width heuristics for maximal constraint satisfaction. *Eleventh National Conference on Artificial Intelligence.* pp. 762–768. – Washington, DC, USA, juillet 1993.

[Wallace93] R.J. Wallace. – Why AC-3 is almost always better than AC-4 for establishing arc-consistency in CSPs. *13th International Conference on Artificial Intelligence*, pp. 239–245. – Chambéry, France, 1993.

[Waltz72] D. Waltz. – *Generating semantic description from drawing of scenes with shadows.* – A.I. Memo n° 271, MIT AI Laboratory, 1972.

[Wilk91] M. R. Wilk. – Equate: An Object-Oriented Constraint Solver. *ACM SIGPLAN NOTICES*, vol. 26, n° 11, novembre 1991, pp. 286–298. – Proceedings of OOPSLA'91.

[Willamowski94] J. Willamowski. – *Modélisation de tâches pour la résolution de problèmes en coopération système-utilisateur.* – Grenoble, France, Thèse d'informatique, Université Joseph Fourier, 1994.

[Winston et al.87] M. E. Winston, R. Chaffin et D. Herrmann. – A taxonomy of Part-Whole Relations. *Cognitive Science*, vol. 11, 1987, pp. 417–444.

[Wright et al.84] J.M. Wright, M.S. Fox et D.L. Adam. – *SRL2 User's Manual.* – Robotics Institute, Carnegie Mallon University, Pittsburgh, PA, USA, 1984.

[Yokoo94] M. Yokoo. – Weak-commitment search for solving constraint satisfaction problems. *Proc. of AAAI'94*, pp. 313–318. – Seattle, WA, USA, 1994.