50376
1991
236

N° d'ordre : 818

THESE

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE FLANDRES ARTOIS

pour l'obtention du titre de

DOCTEUR

en Productique : Automatique et Informatique Industrielle

par

Marie-Hélène CHILOUP-BEKAERT

UTILISATION DE LA NOTION D'OBJETS AVEC CONTRAINTES POUR LA MODELISATION ET LA SIMULATION DES SYSTEMES DE PRODUCTION

Soutenue le 5 décembre 1991 devant la commission d'examen :

Messieurs et Madame

P. VIDAL	*Président*	*Professeur à l'U.S.T.L.F.A.*
P. MILLOT	*Rapporteur*	*Professeur à l'Université de Valenciennes.*
A. DERYCKE	*Rapporteur*	*Professeur à l'U.S.T.L.F.A.*
J.M. TOULOTTE	*Co-Directeur de thèse*	*Professeur à l'U.S.T.L.F.A.*
B. BAUDEL-CANTEGRIT	*Co-Directrice de thèse*	*Maître de Conférence à l'U.S.T.L.F.A.*
A. HAURAT	*Examinateur*	*Professeur à l'Université d'Annecy.*

SOMMAIRE

CHAPITRE 3 : EXPRESSION DE LA CONTRAINTE

CHAPITRE 4 : LA SATISFACTION DES CONTRAINTES

CHAPITRE 5 : DOMAINES D'APPLICATION

INTRODUCTION

INTRODUCTION

La conception, l'apprentissage ou le développement des systèmes de production industriels impliquent un investissement humain et matériel souvent très coûteux. L'intégration de la simulation dans le domaine industriel permet d'abaisser considérablement ces coûts.

Dans la phase de conception d'un système de production, la simulation permet de tester puis de valider l'architecture de l'atelier et d'expérimenter à moindre frais les différents systèmes de conduite envisageables pour une production donnée. Lors de l'apprentissage d'un pilote de conduite, la simulation permet à celui-ci d'acquérir une certaine expérience sans risque d'accident ni de dégât matériel, toujours coûteux et parfois catastrophiques. Enfin, dans les phases de développement ou de réorganisation du système de production, les essais de validation et de conception des commandes pourront être entrepris sans nuire à l'installation actuelle ni à son fonctionnement.

L'utilisation des langages de programmation orientée objet pour la conception des postes de simulation présentent des avantages aujourd'hui reconnus. C'est un moyen simple et naturel pour écrire des simulations. Les concepts de classe et d'instance offrent la possibilité de modéliser puis de créer les machines ou les outils à loisir, l'envoi de message permettant d'autre part de définir les interactions entre les différentes entités crées. Le concept tout-objet mène donc souvent à une structure de représentation uniforme, aussi précise que celle du domaine exploité. De plus, l'extrême modularité des langages de programmation orientée objet leur permet, contrairement aux langages plus classiques, de suivre très facilement les évolutions et les modifications du site d'exploitation, sans remettre en cause la validité du système existant.

Le travail présenté dans ce mémoire a été réalisé au centre d'automatique de l'Université des Sciences et Techniques de Lille Flandres-Artois, sous la direction du Professeur Jean-Marc Toulotte et de Madame Brigitte Baudel Cantegrit. Il fait suite aux travaux effectués au sein de notre équipe [CAN] ayant permis la définition d'un outil de modélisation des processus par lot en vue de la simulation de leur comportement.

Ces premières recherches ont abouti à la définition d'une première plate-forme de modélisation basée sur une décomposition en objets des systèmes industriels automatisés. La technique de décomposition retenue a montré l'intérêt de la programmation orientée objet pour la modélisation des systèmes de production, tout en soulignant la nécessité et surtout la difficulté d'établir précisément les relations régissant les objets de l'application.

Définir un ensemble d'objets n'est pas suffisant à la description d'un système, encore faut-il décrire les liens existant entre ces différents objets. La solution la plus simple serait de définir le système de production comme un seul objet. Si cette solution parait des plus simples, elle n'est plus envisageable quand on considère la complexité de l'objet à définir et l'absence de flexibilité du produit ainsi obtenu. La composition d'objets apparaît alors comme le chemin le plus sage, puisque le plus naturel, vers la modélisation et la simulation.

La principale difficulté dans la composition d'objets est de donner une cohérence et une fonctionnalité à l'objet composé à partir de la seule définition des objets-composantes et de leurs interactivités. S'il fallait d'écrire le fonctionnement de l'objet composé, après en avoir décrit toutes les composantes, la composition d'objets ne serait qu'un leurre, un moyen détourné de définir le système comme un seul et unique objet.

Face à cet impératif, modéliser précisément les relations entre toutes les composantes d'un système, notre équipe étudie actuellement différents moyens pour définir les objets, leurs interactions, leur environnement. L'utilisation d'objets contraints est l'une des méthodes que nous avons développées puis intégrées dans la nouvelle plate-forme de modélisation et de simulation interactive des processus par lots.

Bon nombre de travaux tendent à considérer le problème de la modélisation comme un ensemble de contraintes à définir, puis à résoudre, sur un certain nombre de données. Pourtant peu de ces travaux ont réuni les concepts de contrainte et d'objet. Le but de notre recherche a été de définir la notion d'objet contraint et d'en étudier les différents domaines d'application.

L'exposé de nos travaux comprend cinq parties.

La première partie décrit divers types de relations pouvant être établies sur les objets. La liste fournie n'est pas exhaustive mais rend compte de la diversité des relations tant dans leur fonctionnalité que dans leur complexité.

La deuxième partie définit les notions de contrainte et d'objet contraint. Nous y verrons qu'une contrainte décrit une relation parfois complexe, multidirectionnelle sur un ensemble de données et peut être classée suivant deux critères selon qu'elle soit statique ou dynamique, de classe ou d'instance. Enfin, un objet contraint y est défini comme l'association d'un objet (ou d'un ensemble d'objets) et d'une contrainte. Les objets contraints seront classés selon quatre catégories.

La troisième partie est consacrée à l'expression des différents types de contraintes. Le mode d'expression des contraintes dépend beaucoup du langage de programmation utilisé, même si les notions de base restent les mêmes. Nous verrons qu'il est possible de dégager deux courants, l'un basé sur une expression mathématique de la contrainte, l'autre sur une description formelle.

Le quatrième chapitre présente plusieurs méthodes de satisfaction des contraintes. Ces méthodes considèrent la propagation des contraintes sous deux angles différents. Certaines guident l'instanciation des variables par l'évaluation des contraintes, d'autres à l'inverse guident l'évaluation des contraintes par l'instanciation des variables.

Enfin, la cinquième partie expose succinctement l'implantation d'un mécanisme de satisfaction des contraintes en Smalltalk-80 puis donne un aperçu des différents domaines d'utilisation des objets contraints. Parmi ces domaines nous retrouvons bien-sûr la simulation des systèmes de production industriels. Nous verrons dans quelles mesures l'utilisation des objets contraints simplifie la définition d'une simulation interactive et aide à la construction de son interface. D'autres domaines d'utilisation sont également envisagés comme l'animation d'algorithmes ou l'Enseignement Assisté par Ordinateur.

CHAPITRE 1 : OBJETS ET RELATIONS

CHAPITRE 1 : OBJETS ET RELATIONS

En programmation orientée objet, toute donnée est objet, des objets primitifs comme les entiers, aux objets plus complexes comme les textes structurés ou les fichiers. Mais comment situer ces objets les uns par rapport aux autres ? Comment les utiliser, les mettre en rapport pour définir telle ou telle application ?

Considérant acquise la notion d'objet, ce chapitre décrit les relations les plus couramment définies sur un objet ou sur un ensemble d'objets.

La première partie expose brièvement les relations de base de la programmation orientée objet comme la relation d'héritage ou l'envoi de message. Les deux parties suivantes détaillent deux structures particulières de relations, la structure M.V.C (Modèle Vue Contrôleur) et la structure M.V.C étendue (Modèle Vue Contrôleur Routine-Graphique).

Smalltalk-80 ayant été choisi comme langage de programmation pour développer notre plate-forme de simulation des systèmes de production, la terminologie employée dans ce chapitre se réfère essentiellement à ce langage.

1.1. OBJETS ET RELATIONS

En programmation orientée objet, l'objet n'est pas considéré comme une simple donnée. S'il permet de stocker les informations qui lui sont propres (comme toute donnée en programmation classique), il est aussi capable de répondre à un ensemble de messages. L'objet est à la fois une description structurelle et fonctionnelle de la donnée.

De nombreux écrits ont déjà été publiés sur la "méthodologie objet". Au lecteur désirant de plus amples détails sur les notions de base de la programmation orientée objet, nous ne pouvons que recommander les publications, dont les références [STE] [STR] [WEG] [BOO] [HAI] [ZDO] figurent en bibliographie, .

Un système est constitué par un ensemble d'objets sur lesquels sont définies des relations statiques ou dynamiques. Ces relations peuvent permettre de situer l'objet dans un environnement (une classe d'objets, une hiérarchie ou une structure d'objet composé) et gérer ainsi l'accès à certaines informations. Elles peuvent aussi tout simplement définir un mode de communication entre objets ou une dépendance.

Ce paragraphe dresse une liste, non exhaustive, des relations pouvant être définies sur un objet ou entre un ensemble d'objets. Certaines de ces relations, inhérentes à la programmation orientée objet, comme la relation d'appartenance à une classe ou la relation d'héritage, peuvent paraître simples, voire naturelles. Toutefois, ne pas les citées ici eut été une erreur car ces relations contribuent pour beaucoup à la clarté, à la logique et à la réutilisabilité des applications orientées objet.

1.1.1) L'appartenance à une classe

Les concepts de classe et d'objet instance d'une classe ont été introduits dans de nombreux langages de programmation orientée objet. Le concept de classe permet de regrouper sous une seule et même structure tous les objets de définition et de comportement identiques (par exemple la classe des points, la classe des entiers...). Ces objets sont alors appelés instances de la classe .

Une classe décrit sous quelle forme seront stockées les informations concernant ses instances . Les variables caractérisant les objets de la classe peuvent être propres à chaque instance ou partagées par toutes les instances de la classe. La réponse d'une instance à un message est décrite au niveau de la classe dans une méthode.

Les classes offrent ainsi une description générique des objets, leurs instances sont les objets.

La relation d'appartenance à une classe est une relation statique, de l'objet instance vers sa classe, générée systématiquement à la création de l'objet. C'est une relation simple, de type fils-père, n'intégrant aucune autre information. Elle est mise en place par un pointeur de l'objet vers le nom de sa classe.

Regrouper sous une seule classe les objets de même nature crée une relation sur ces objets. Concrètement, cette relation est davantage exploitée par le système interne au langage de programmation que par l'utilisateur :

- L'utilisateur n'utilise la relation de classe que pour identifier ou caractériser les objets. Les classes lui permettent d'organiser ses données, de les regrouper selon des critères qu'il aura choisis.

- Le système utilise la relation d'appartenance à une classe, entre autre, pour trouver dans quelle classe a été définie la méthode correspondant au message envoyé à un objet. En effet, il n'existe aucun lien direct entre un objet et l'ensemble des méthodes auxquelles il répond. La recherche d'une méthode à partir d'un sélecteur se fait lors de l'exécution. Le système regarde à quelle classe appartient l'objet et trouve dans celle-ci, ou dans sa hiérarchie (Cf. relation d'héritage), la méthode correspondante. Cette recherche est souvent assimilée à une liaison dynamique entre l'objet et l'ensemble des méthodes auxquelles il répond, mais ce lien est purement fictif. Le fait d'associer les méthodes aux classes et non aux instances est un atout important des langages à objets, qui leur assure la modularité (un objet peut être modifié ou remplacé sans remettre en cause la validité du système) et la réutilisabilité (l'utilisateur peut à loisir réutiliser les même classes sans utiliser les même objets).

1.1.2) La relation d'héritage

Le concept de classe a permis de simplifier considérablement la programmation et de diminuer de beaucoup la taille des applications. L'utilisateur ne redéfinit pas pour chaque objet sa structure de définition ni l'ensemble des méthodes auxquelles il répond, l'objet est créé selon un modèle défini par sa classe. Dans la même optique, le concept d'héritage permet de définir une classe comme la modification ou la spécialisation d'une classe déjà existante [GOL] [SNY].

Lorsqu'une classe hérite d'une autre classe (la première est appelée sous-classe et la seconde super-classe), elle possède par défaut toutes les variables et toutes les méthodes de sa super-classe. Les spécifications qui lui sont propres seront définies par l'ajout de nouvelles variables, l'ajout de nouvelles méthodes ou la redéfinition de méthodes héritées.

La relation d'héritage est une relation statique, d'une sous-classe vers sa super-classe, établie à la création de la sous-classe. Il s'agit là encore d'une simple relation de type fils-père souvent bidirectionnelle : la sous-classe possède un pointeur vers sa super-classe et inversement la super-classe possède un pointeur vers la liste de ses sous-classes. La relation inverse, de la super-classe vers ses sous-classes n'est pas nécessaire pour définir la relation d'héritage proprement dite mais est indispensable au système interne pour gérer d'éventuels conflits. Un conflit apparaît, par exemple, lorsque l'utilisateur définit au niveau de la super-classe des variables d'instance déjà nommées dans une de ses sous-classes.

La relation d'héritage est utilisée par le système interne au langage de programmation pour :

- rechercher dans la hiérarchie des super-classes la méthode correspondant au message envoyé à un objet, si cette méthode n'a pas été trouvée dans la classe même de l'objet ;
- connaître l'ensemble des variables définies sur une classe donnée (ses propres variables et les variables définies dans la hiérarchie de ses super-classes).

Pour l'utilisateur, la relation d'héritage permet de donner à un objet différents niveaux de définition, du plus général au plus précis. Un exemple concret sur l'utilisation de l'héritage est la définition en Smalltalk-80 des ensembles de données, comme le montre la hiérarchie de classes illustrée par la figure 1.1.

Cet exemple définit de façon générale tout ensemble de données comme une collection. La classe COLLECTION est une classe abstraite (ne possédant pas d'instance), dans laquelle sont écrites les méthodes communes à tout ensemble de données.

Les classes BAG et SEQUENCEABLECOLLECTION font une première distinction entre les ensembles non ordonnés de données et les ensembles ordonnés, et ainsi de suite, chaque niveau d'héritage correspondant à un degré de spécialisation.

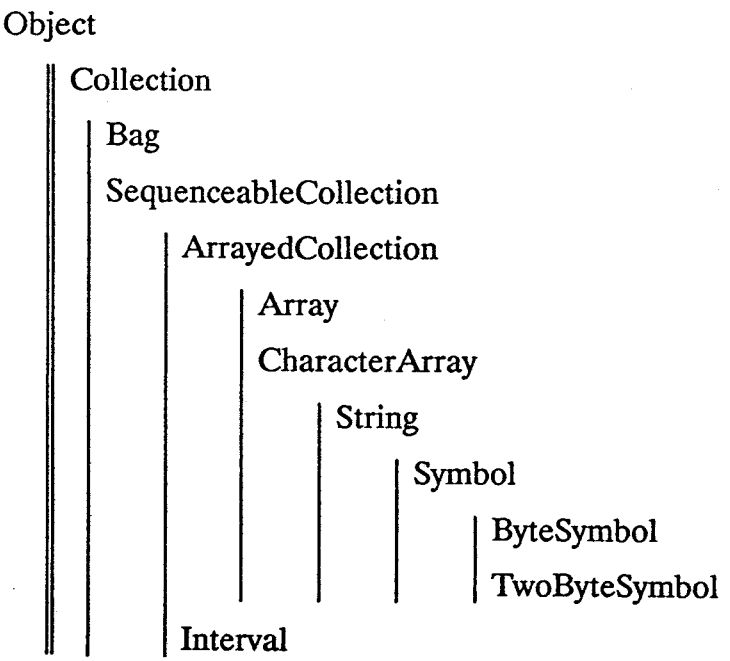


Figure 1.1

1.1.3) Le multiple héritage

Certains langages de programmation orientée objet ont étendu le concept d'héritage à celui de multiple héritage [MEV] [CAR] [DUC]. Le principe en est le même sauf qu'il permet à une classe d'hériter, au même niveau, des variables et des méthodes de plusieurs super-classes (figure 1.2).

Supposons à modéliser une population selon son activité professionnelle. L'utilisateur classe chaque personne selon quatre critères (salarié, étudiant, retraité, sans-emploi) et définit donc quatre classes de même niveau. Puis, après une enquête plus approfondie, il se révèle que certaines personnes répondent à deux critères comme les étudiants-salariés. En héritage simple pour construire la classe des étudiants-salariés, l'utilisateur peut soit créer une sous-classe de la classe ETUDIANT, soit créer une sous-classe de la classe SALARIE. Supposons que la classe ETUDIANT-SALARIE soit définie sous-classe d'ETUDIANT, chacune de ses instances hérite des caractéristiques de l'étudiant, reste à définir au niveau de cette nouvelle classe les caractéristiques (variables et méthodes) d'un salarié. En héritage multiple, la classe ETUDIANT-SALARIE peut être à la fois définie comme sous-classe de la classe ETUDIANT et sous-classe de la classe SALARIE. Ses instances héritent donc au même niveau des caractéristiques de l'étudiant et du salarié, sans que l'utilisateur n'est à redéfinir aucun des deux comportements.

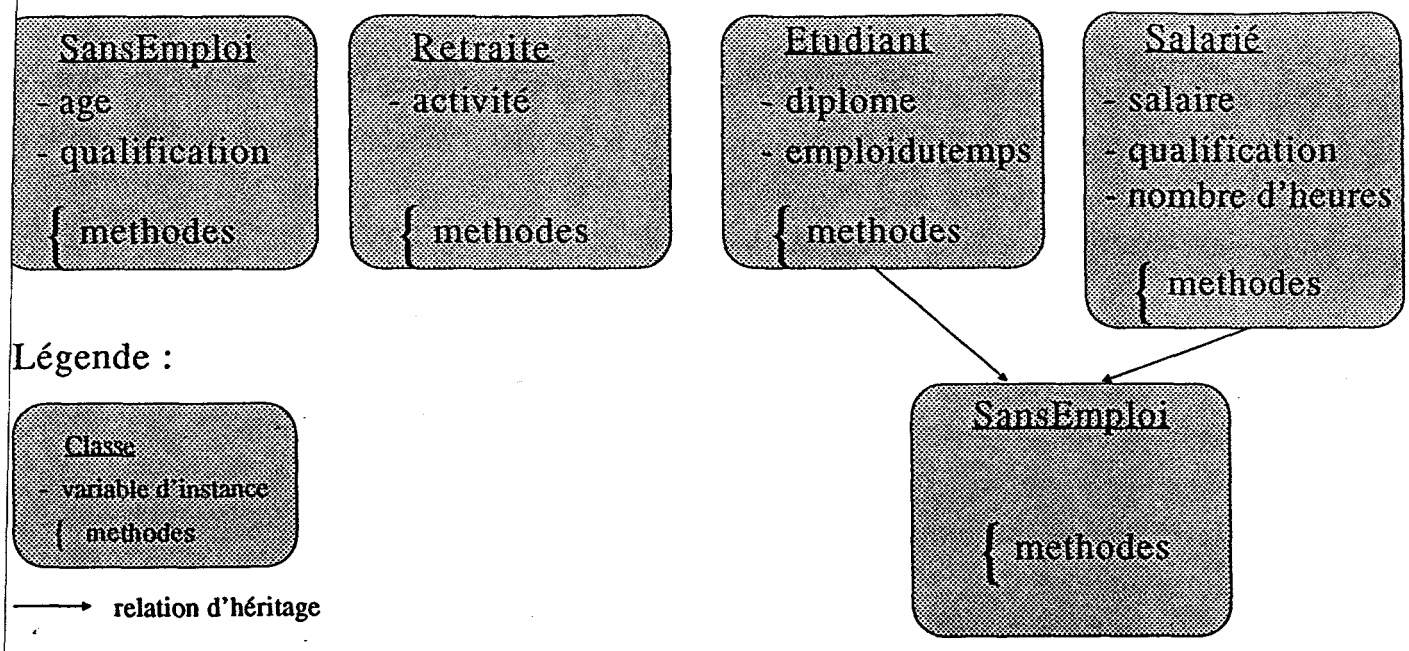


Figure 1.2

Le multiple héritage comme l'héritage simple est une relation statique implantée à l'aide de pointeurs: un pointeur de la super-classe vers la liste de ses sous-classes, un pointeur de la sous-classe vers la liste de ses super-classes.

S'il présente des aspects attractifs, le multiple héritage est beaucoup plus difficile à gérer que l'héritage simple : il oblige à exclure tout conflit entre super-classes de même niveau pour la dénomination des sélecteurs de méthodes et entre super-classes, tous niveaux confondus, pour la dénomination des variables.

Le multiple héritage offre à l'utilisateur, comme nous l'avons vu avec l'exemple cité précédemment, la possibilité de définir plusieurs comportements pour un même objet .

1.1.4) Délégation et prototypage

Pour les langages basés sur le prototypage et la délégation [ALM] [BOR] [STEI] [LIE], un objet n'est plus instance d'une classe, mais simplement une liste de champs nommés. Chaque champ peut contenir un autre objet ou une méthode. Quand un message est envoyé à l'objet, celui-ci regarde si un de ses champs porte pour nom le sélecteur du message, si oui il renvoie la valeur du champ ou exécute la méthode qu'il contient. Si l'objet ne trouve pas de champ correspondant au message reçu, il délègue celui-ci vers un autre objet susceptible d'y répondre. De façon générale, le message est transmis à l'objet contenu dans un champ précis désignant le "père" du receveur (s'apparente à la recherche d'une méthode dans la hiérarchie des classes).

Dans cette approche, un prototype est un objet servant de modèle, de référence, à un ensemble d'objets ayant une définition et un comportement identique ou voisin. C'est vers son prototype qu'un objet délègue les messages auxquels il ne peut répondre, le prototype pouvant à son tour déléguer ces messages vers d'autres objets.

L'usage du prototypage et de la délégation permet de suppléer d'une part à la relation d'appartenance à une classe (prototypage) et d'autre part à la relation d'héritage (délégation). Nous ne développerons pas davantage ces techniques qui selon nous n'offrent pas une classification des objets aussi claire que les concepts de classe et d'héritage.

1.1.5) L'utilisation des variables d'instance

L'utilisation des variables d'instance assure aux applications orientées objet leur modularité et leur réutilisabilité. Toute information sur l'état d'un objet n'est accessible que de l'objet lui-même, toutefois l'utilisateur peut manipuler les données sans aucune difficulté : les détails de fonctionnement d'un objet ne lui sont pas nécessaires pour le manipuler, seule la connaissance du comportement externe de l'objet lui est utile.

Dans les langages de programmation orientée objet, le concept d'encapsulation est sous-jacent à la structure des données [SNY]. L'état d'un objet est déterminé par la valeur de ses variables d'instance (et variables de classe). Ces valeurs sont elles même des objets dont l'état est déterminé par la valeur de leurs variables d'instance et ainsi de suite jusqu'à l'obtention d'objets primitifs.

Dans l'exemple présenté par la figure 1.3, une instance de la classe RECTANGLE possède deux variables d'instance, **corner** et **origin**, chacune pointant sur une instance de la classe POINT. Un point possède lui même deux variables d'instance, l'abscisse **x** et l'ordonnée **y**, pointant chacune vers un nombre, objet primitif.

Cette figure met en évidence un réseau de relations sur les objets, mis en place par l'ensemble des pointeurs que constituent les variables d'instance. Ce réseau peut être complexe et de forme non arborescente, comme celui présenté par la figure 1.4, relatif à la définition d'une fenêtre graphique de type Choix-Binaire (activée , non-activée) en Smalltalk-80.

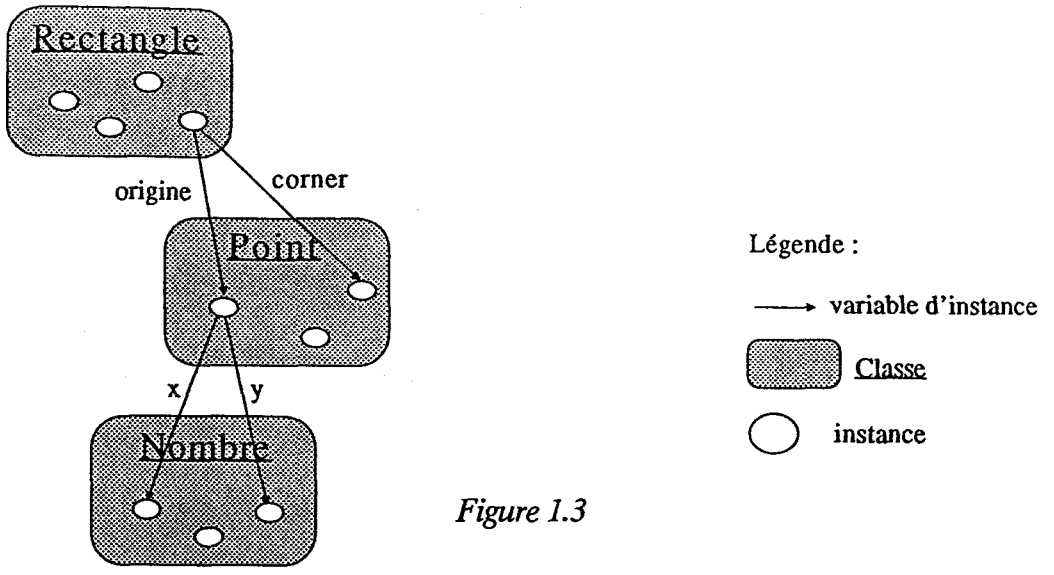


Figure 1.3

Les relations inter-objets, obtenues par la définition de variables d'instance, sont de deux sortes :

- Elles peuvent tout simplement définir l'objet. Dans l'exemple de la figure 3, un rectangle se définit par deux points, les sommets de deux angles opposés. Un point est lui-même défini par deux valeurs numériques fixant ses coordonnées.

- Elles peuvent plus généralement établir une relation statique entre deux objets, dans le sens "faire référence à". Dans l'exemple de la figure 4, représentant la structure de représentation graphique d'un commutateur, l'objet vue-d'un-choix-binaire fait référence à son modèle, l'objet choix-binaire (la représentation faisant ainsi référence à la donnée qu'elle représente).

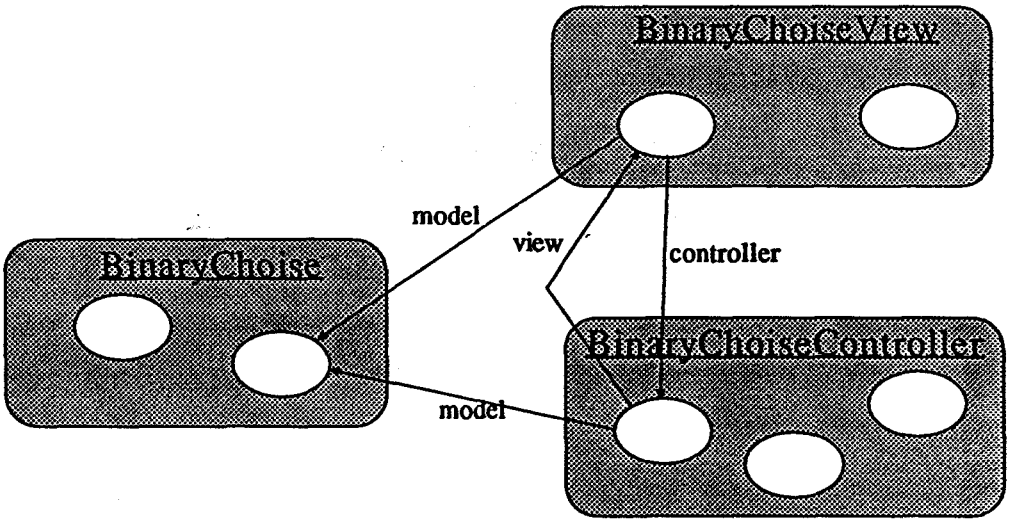


Figure 1.4

La définition de variables d'instance est un moyen simple pour établir des liens statiques entre objets. Cependant cette solution s'avère limitée lorsqu'il s'agit de modéliser des relations nombreuses ou complexes (définition d'un nombre incalculable de variables d'instance et de méthodes nécessaires à leur gestion). Une utilisation intempestive des variables d'instance, pour modéliser les relations statiques entre objets, risque de perdre l'utilisateur dans un nombre trop important d'informations, toutes données au même niveau : parmi une dizaine de variables d'instance, comment discerner les variables décrivant la structure de l'objet, des variables décrivant ses relations avec d'autres objets ? En cas d'évolution dans l'une de ces relations, comment retrouver exactement toutes les méthodes à modifier dans chacun des objets concernés par la relation ?

1.1.6) L'envoi de message

L'envoi de message est le seul moyen d'agir sur les objets. Il établit une relation dynamique entre objets, leur permettant de dialoguer. Un message implique toujours une réponse. Quand il n'est pas de structure trop complexe, il se compose schématiquement :

- d'un receveur, c'est l'objet auquel s'adresse le message,
- d'un sélecteur, nom de la méthode à appliquer au receveur,
- d'un ensemble d'arguments, paramètres de cette méthode.

Lorsqu'un message est envoyé, le système cherche la méthode correspondant au sélecteur dans la classe de l'objet receveur, ou dans sa hiérarchie. S'il trouve cette méthode, il l'applique au receveur, sinon il renvoie un message d'erreur.

L'envoi de message est donc un moyen simple pour mettre en relation différents objets. Etant donné son caractère dynamique, il convient tout particulièrement pour modéliser des processus de communication.

1.1.7) La relation de partie

La relation de partie est peu implantée dans les langages de programmation orientée objet [BLA] [WOL] [BOR]. Cependant certaines applications offrent à l'utilisateur la possibilité de définir les objets par composition.

1.1.7.1) **Le principe de base de la composition d'objets :**

Tout système, et par conséquent tout objet, peut être décomposé en sous-systèmes, eux même décomposables en sous-systèmes et ainsi de suite jusqu'à l'obtention de systèmes primitifs. Cette représentation d'un tout en parties de plus en plus fines est communément appelée hiérarchie de parties. La composition d'objets est en ce sens une technique de description des objets complexes : l'utilisateur définit des systèmes simples qu'il compose pour créer des ensembles de plus en plus compliqués.

Dans un objet composé, on retrouve deux types d'informations : les informations propres à chacune des parties et les informations propres au tout (qui résultent précisément de la composition de plusieurs objets au départ indépendants). Dans la hiérarchie de parties que constitue l'objet composé, ces informations doivent être stockées à des niveaux logiques donc à des niveaux différents. Toute information concernant le tout ne doit pas être stockée dans les parties, inversement, toute information sur une partie, qui n'est pas modifiée par le tout, doit rester stockée dans cette partie. De plus, dans l'idéal, le tout connaît les parties qui le composent mais ces parties n'ont pas connaissance du tout. Les relations de partie devraient donc être des relations de l'objet composé vers ses composantes, c'est à dire des relations de type "apourpartie" et non des relations de type "estpartie-de".

1.1.7.2) **Les problèmes liés à la composition d'objets**

Quand un objet est assemblé à partir de parties, ces parties ne sont plus des objets indépendants. L'utilisateur peut-il agir sur ces objets directement ou ne peut-il agir que par l'intermédiaire du tout ? L'objet composé est-il vu comme une boite noire (parties invisibles à l'utilisateur) ou comme un ensemble de composantes accessibles (parties visibles par l'utilisateur) ?

La réponse à ces questions dépend de ce que l'on entend par composition d'objet.

Si l'objet composé est une boite noire dont les parties sont inaccessibles, la composition d'objets s'apparente dans ce cas à la notion d'encapsulation. Il suffit de définir chaque partie comme variable d'instance d'un l'objet-tout et de définir l'ensemble du protocole relatif à l'objet composé à ce niveau. Chacune des parties n'est alors accessible que de l'objet-tout lui-même. L'inconvénient majeur de cette démarche est qu'elle oblige à définir une classe pour chaque composition d'objets. L'encapsulation écrase totalement la hiérarchie de partie.

Si l'objet composé est vu comme un ensemble de composantes accessibles, il devient nécessaire de contrôler l'accès à ces composantes. En effet, toute action sur une des parties doit être communiquée au tout pour qu'il répercute, si besoin est, cette action sur l'ensemble de ses composantes. Le problème se pose alors, lorsqu'on veut modifier un objet, de savoir s'il est composant d'un autre objet plus complexe. On voit ici que la relation inverse "est partie de" ne peut en pratique être ignorée.
Vue sous cet angle la composition d'objets est un problème difficile dans la modélisation des systèmes. Les relations de base de la programmation orientée objet que nous avons vus jusqu'ici ne permettent pas de modéliser les relations de composition. Les deux paragraphes suivants montrent une première approche des systèmes composés définis par la déclaration de dépendances (triade M.V.C du Smalltalk-80 et M.V.C étendu). Nous verrons par la suite comment aborder ce problème par l'utilisation de contraintes modélisant et gérant les interactions entre les différentes parties d'un tout.

1.2. LA RELATION DE DEPENDANCE

La notion de dépendance est une notion que l'on retrouve à différents niveaux en programmation orientée objet :

- une classe étant définie dans une hiérarchie de classes, elle est dépendante de ses super-classes ;
- si la définition d'une classe évolue, ses instances doivent être actualisées en conséquence ;
- ...

Ce type de dépendance est géré par les messages de création et modification d'une classe.

Il existe outre ces formes de dépendances implicites à la programmation orientée objet, une relation de dépendance explicite, introduite par Smalltalk-80 [MEV] [GOL]. Elle permet de déclarer que les propriétés ou actions d'un objet sont directement dépendantes des propriétés ou actions d'un autre objet, sans avoir déclaré le second comme variable d'instance du premier. Une dépendance est une relation, définie par l'utilisateur, d'un objet-maître vers un ou plusieurs autres objets qui deviennent, de ce fait, objets-dépendants. Lorsqu'un changement intervient sur l'objet-maître, il est automatiquement signalé à tous ses objets-dépendants afin qu'ils puissent en tenir compte.

Le mécanisme de dépendance

Une relation de dépendance se concrétise par un pointeur, de l'objet-maître vers l'ensemble de ses objets-dépendants. Cette relation est répertoriée dans un dictionnaire, le dictionnaire des dépendances, où sont enregistrées l'ensemble des dépendances définies sur un système. Lorsqu'une dépendance a été établie, un mécanisme interne répercute de façon systématique tout changement de l'objet-maître sur ses dépendants. Ce mécanisme est déclenché par l'objet-maître qui signale une modification, il envoie alors à tous ses objets-dépendants une demande de mise à jour.

Répercussion d'un changement de l'objet-maître vers ses objets-dépendants

Nous avons vu que le mécanisme de dépendance permettait de faire savoir à tous les objets-dépendants d'un objet-maître que celui-ci avait changé et qu'ils devaient, par conséquent, se mettre à jour. Les méthodes permettant un dialogue entre l'objet-maître et ses dépendants sont de 3 types :

- L'objet-maître doit changer, il demande à tous ses dépendants s'ils sont d'accord avec ce changement.
Le comportement par défaut d'un objet-dépendant est d'accepter tout changement de l'objet dont il dépend. Cette méthode peut être redéfinie dans une sous-classe si l'on désire changer ce comportement pour une catégorie d'objets.

- L'objet-maître annonce qu'il a subit un changement et demande à tous ses dépendants d'en tenir compte.
Les dépendants de l'objet-maître, informés d'un changement, en tiennent compte par une remise à jour. Par défaut, l'objet-dépendant ne répond pas à la demande mise à jour, ou plus exactement, il y répond en ne faisant rien. Les messages de mise à jour doivent être redéfinis par l'utilisateur dans les sous-classes concernées

- Certains messages permettent une communication entre l'objet-maître et ses dépendants, en dehors de la répercussion d'un changement.
Le mécanisme de dépendance permet donc de lier les comportements de deux objets. Si l'utilisateur veut définir un objet B dépendant d'un objet A, il lui faut:

1°) ajouter l'association, A -> B, dans le dictionnaire des dépendances ;

2°) pour chaque méthode définie dans la classe de l'objet A, si cette méthode

modifie l'état du receveur, y inclure le message indiquant qu'un changement est intervenu *(self changed: anAspectSymbol)* ;

3°) définir, dans la classe de l'objet B, les méthodes de mise à jour (*update:* , *update:with:* ,...).

Le principal défaut de cette relation de dépendance réside dans le deuxième point. L'utilisateur doit signaler tout changement de l'objet-maître dans les méthodes définies pour sa classe et ses super-classes. S'il réutilise pour son application des classes déjà existantes, l'utilisateur devra donc scruter tout le protocole de ces classes, trouver les méthodes modifiant le receveur et les compléter pour gérer les dépendances qu'il établira.

L'annexe_1 de ce document détaille l'implantation du mécanisme de dépendance dans le langage de programmation Smalltalk-80. Y figure également, un exemple d'utilisation simple, basé sur la modélisation d'un carrefour routier à feux tricolores.

1.2.1) Le système Modèle-Vue-Contrôleur

L'utilisation la plus significative du mécanisme de dépendances en Smalltalk-80 est la définition de la structure M.V.C. (Modèle Vue Contrôleur). Nous ne donnerons ici qu'un bref descriptif de cette structure. Pour plus de détails, le lecteur voudra bien se reporter aux ouvrages [PIN] [MEV] [GOL].

En Smalltalk-80, toutes les applications interactives sont construites sur le modèle M.V.C.. Cette structure repose sur trois entités:

- **le modèle:** c'est l'objet référençant la donnée sur laquelle on veut travailler. L'objet 'modèle' peut être instance de toute classe se prêtant à une représentation graphique ;

- **la vue:** c'est la représentation visuelle de l'objet référencé par le modèle. Son rôle est celui d'une interface de sortie ;

- **le contrôleur:** c'est l'objet fournissant le protocole d'interaction entre l'utilisateur, la vue et son modèle. Son rôle est celui d'une interface d'entrée.

Les relations entre un modèle, la vue et le contrôleur associés, sont représentées par la figure 1.5.

La vue doit être une représentation exacte de l'objet référencé par le modèle. Si ce dernier subit un changement, il faut donc que la vue en soit immédiatement informée pour pouvoir illustrer fidèlement le nouvel état de son modèle.

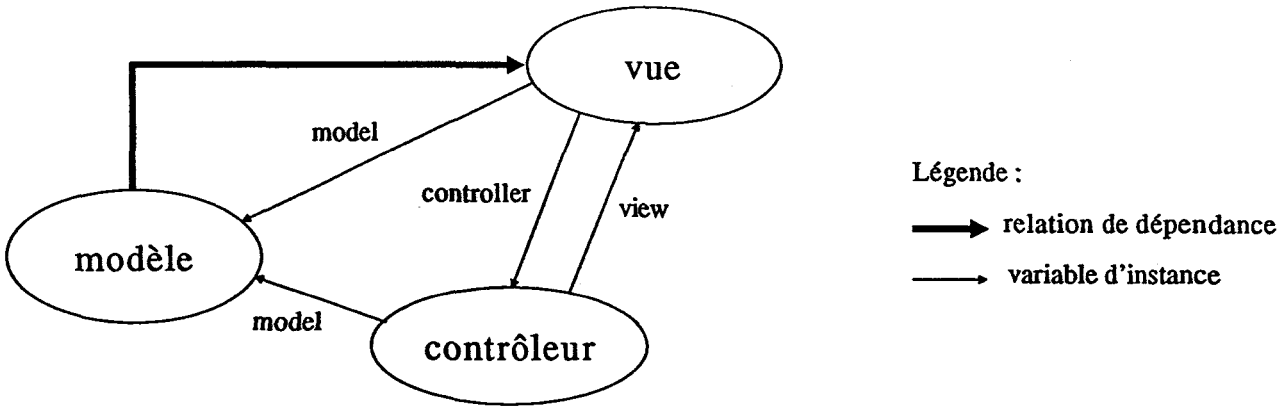


Figure 1.5

Gérer l'interaction modèle-vue au moyen de variables d'instance n'est pas envisageable pour plusieurs raisons :

- Cela implique que l'on connaisse, lors de la définition de la classe du modèle, le nombre exact de ses représentations quelle que soit l'application ; mieux encore, que ce nombre n'évolue pas d'une application à l'autre.

- Supposons que le modèle ait plusieurs représentations, chacune des méthodes modifiant le modèle devra inclure un message de mise à jour pour chacune des représentations. Se posera alors le problème de l'héritage des représentations ! Si un modèle hérite de variables d'instance associées à des représentations, l'utilisateur ne devra pas oublier leur mise à jour en cas de modification.

Pour ces diverses raisons, on préfère définir une relation de dépendance du modèle vers l'ensemble de ses représentations. D'abord, il est très facile d'ajouter ou d'enlever un élément de la liste des objets-dépendants d'un objet-maître et donc de faire évoluer le nombre des représentations d'un modèle (figure 1.6). Ensuite il suffira d'inclure, dans chaque méthode modifiant le modèle, le message d'appel au mécanisme de dépendance.

Lorsque le modèle est modifié, il reçoit le message *changed* (ou *changed:with:*) qui répercute le changement sur tous les objets-dépendants du modèle, donc sur toutes les vues, en leur envoyant le message *update* (ou *update:with:from:*), message de mise à jour.

De façon générale, lorsqu'il est envoyé à une instance de la classe Vue, ou d'une de ses sous classes, le message de mise à jour consiste à rafraîchir la représentation graphique, c'est à dire à redessiner la vue.

En plus de la dépendance, modèle-vue, la vue et le contrôleur doivent se connaître mutuellement, et connaître le modèle sur lequel ils travaillent.
En effet, on sait que l'utilisateur interagit, par le biais d'une interface d'entrée qu'est le contrôleur, avec l'objet ou sa représentation (le modèle ou la vue). Le contrôleur doit donc connaître le modèle et la vue qui lui sont associés.
Si l'utilisateur interagit avec la vue (par exemple, en modifiant une partie d'un texte édité), celle-ci doit connaître le modèle qui lui est associé pour que tout changement ait lieu sur ce modèle lui-même. Attention, la relation de dépendance est unidirectionnelle ! Définir la dépendance d'une vue vis à vis d'un modèle ne crée aucune dépendance du modèle envers la vue.

Pour maintenir ces relations, les classes "Views" (sous entendu la classe Vue et ses sous-classes) possèdent pour variables d'instance les variables **model** et **controller.** De même, les classes "Controllers" possèdent pour variables d'instance les variables **view** et **model.**

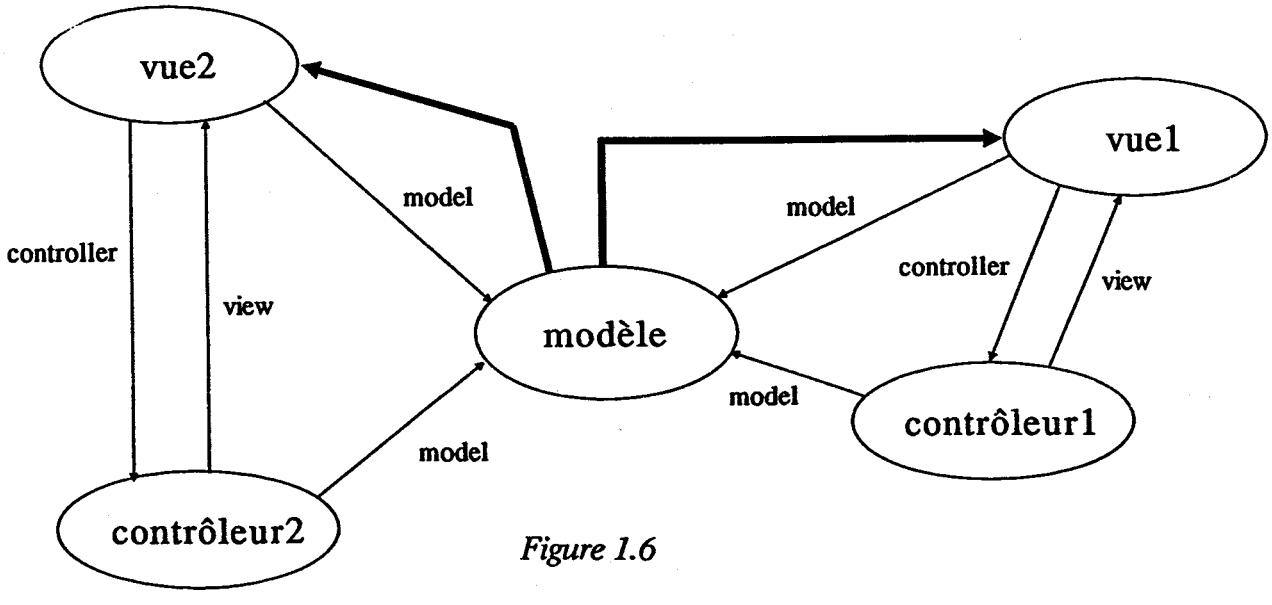


Figure 1.6

Par définition, la structure M.V.C établit donc deux types de dépendances sur les objets qui la composent :

- Une dépendance réelle entre le modèle et la vue, marquée par la relation, modèle -> vue, dans le dictionnaire des dépendances.

- Les dépendances introduites par la définition de variables d'instance.

Le réseau des variables d'instance permet la gestion des interactions de type contrôleur-modèle ou contrôleur-vue. Pour gérer ces interactions, il n'est pas utile de créer une relation de dépendance effective 'contrôleur -> modèle' ou 'contrôleur -> vue' :

* dans le premier cas, il est évident que le comportement du modèle n'est pas dépendant, au sens strict, de celui du contrôleur. Si l'objet référencé par le contrôleur change, le modèle lui, n'a que faire de ce changement, seul les messages qui lui seront envoyés risquent d'être modifiés, à lui d'y répondre ou de ne pas y répondre ;

* dans le second, on conçoit qu'une représentation visuelle d'un objet soit dépendante de celui-ci, on conçoit moins que l'objet-modèle soit dépendant au sens strict de sa représentation (si la vue change de format ou de dessin, l'objet qu'elle représente n'a aucun intérêt à connaître ces modifications).

1.3. LE MODELE M.V.C. ETENDU

Le modèle M.V.C. est en Smalltalk-80 la structure de définition de toute application interactive. C'est, par conséquent, la première structure vers laquelle on se tourne pour construire une animation ou une simulation interactive.

Pour bâtir une animation, il faut :

1°) définir les différents objets dont on veut animer le comportement (ensemble des objets {O}) ;
2°) définir leur(s) représentation(s) graphique(s) ;
3°) mettre en relation chaque objet de l'ensemble {O} avec sa représentation graphique de telle sorte que celle-ci reflète toujours, très exactement, l'état de l'objet auquel elle est associée ;.
4°) si on envisage une animation interactive, c'est à dire donnant à l'utilisateur la possibilité d'intervenir sur le comportement des objets représentés au moyen de la souris ou du clavier, il faut associer à la paire objet-représentation un troisième objet, instance de la classe Contrôleur, qui gérera toute interaction entre l'utilisateur et l'application.

Il semble donc, effectivement, que le système Modèle-Vue-Contrôleur soit un outil, adapté à la conception d'animations graphiques interactives.

Cependant, pour bâtir une animation sur le comportement d'un ensemble d'objets, il faut non seulement fournir une représentation graphique fidèle de ces objets, mais aussi associer un scénario d'événements graphiques aux actions pouvant être perpétrées sur ces objets (sélections, permutations...).

Par exemple, pour construire une animation sur le comportement d'une pile, il faudra fournir une représentation graphique de la pile et associer une séquence d'événements graphiques à l'arrivée d'une donnée dans la pile ou à son retrait.

Bâtir une animation sur le système M.V.C., présente alors un inconvénient majeur : les différentes séquences graphiques, illustrant les actions sur le modèle, sont introduites dans la définition même de ces actions.

Dans l'exemple cité ci-dessus, tout le code graphique relatif au placement d'une donnée dans la pile sera inséré dans la méthode *'empiler: uneDonnée'* définie pour la classe PILE.

En d'autres termes, l'application et l'animation sont indissociables : l'application ne peut pas tourner sans l'animation, l'animation ne peut pas être transposée sur une autre application.

Pour résoudre ce problème de "portabilité" des animations, Robert A. Duisberg et Ralph L. London ont étendu la structure M.V.C. en y ajoutant une quatrième entité, la routine-graphique [LON].

Les relations entre le modèle, la vue, le contrôleur et la routine-graphique associés, sont représentées par la figure 1.7 :

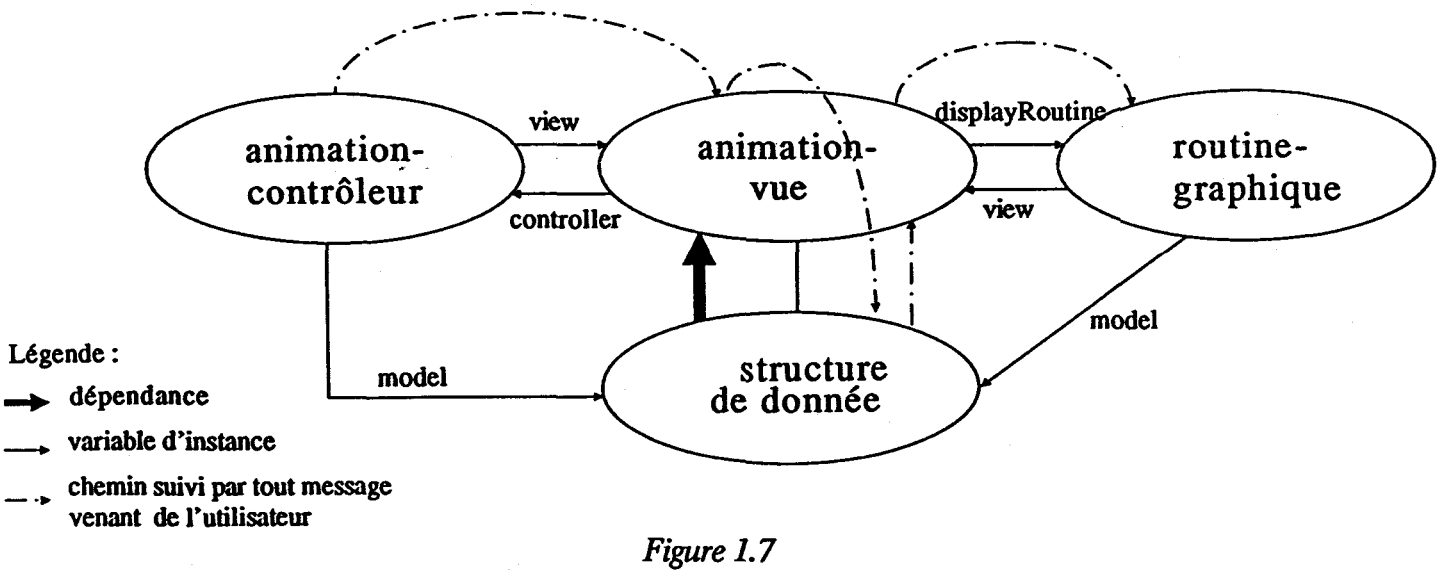


Figure 1.7

L'objet, routine-graphique, regroupe toutes les méthodes spécifiques à la création et à la gestion d'une animation reflétant l'état et le comportement d'un objet.

Dans cette nouvelle structure, désignée comme structure d'animation, R. London et R. Duisberg ont généralisé le rôle du contrôleur et de la vue en créant deux nouvelles classes, AnimationView et AnimationController, sous-classes respectives de StandardSystemView et MouseMenuController. Quelle soit la structure de donnée, c'est à dire quelque soit le modèle dont on désire animer le comportement, la vue et le contrôleur qui lui seront associés, resteront toujours des instances de AnimationView et de AnimationController. Le rôle de ces deux objets se borne, maintenant, à celui de relais : le contrôleur sert de relais entre l'utilisateur et le modèle (le receveur du menu-message n'est plus le contrôleur lui même mais son modèle), la vue sert de relais entre le modèle et la routine graphique associée (la demande de mise à jour transite bien, par la relation de dépendance, vers la vue associée au modèle mais elle est aussitôt transmise à la routine-graphique qui réalise la mise à jour effective). Le message de mise à jour, *update: anInterestingEvent*, a pour cette raison été défini dans la classe AnimationView par la méthode :

```
update: anInterestingEvent
        displayRoutine update: self with: anInterestingEvent
        "une instance de la classe InterestingEvent est qu'un regroupement de
        toutes les informations nécessaires à la routine graphique pour sa mise à
        jour".
```

Le principal intérêt de cette extension est qu'elle permet de dissocier l'application de son animation, de plus, si les objets DisplayRoutines sont suffisamment généraux, elle permet également de transposer la même animation sur plusieurs applications différentes.

Elle présente toutefois un inconvénient: cette structure devient vite lourde et complexe dés qu'il s'agit d'animer une application mettant en cause un certain nombre d'objets (entre autre quand il s'agit d'animer le comportement d'un objet composé).

Prenons, pour exemple, le cas de l'animation d'un système Producteur-Consommateur. Ce système met en relation quatre types d'objets: un producteur, un Consommateur, un magasin et un moniteur gérant les accès au magasin. Les relations liant ces différents objets peuvent être modélisées comme suit (figure 1.8) :

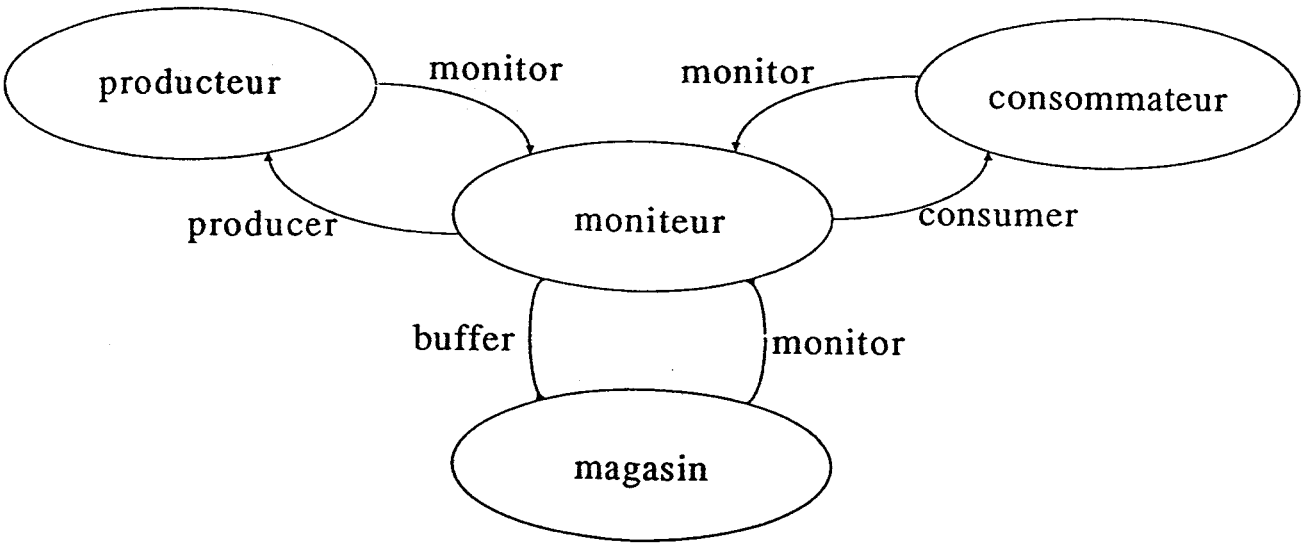


Figure 1.8

Dans cette application, l'utilisateur commande à son gré la production ou la consommation. Dans le cas d'une production, une donnée transite du producteur vers le magasin. Elle transitera du magasin vers le consommateur dans le cas d'une consommation. Si le magasin est vide, l'utilisateur ne peut consommer. Sa demande doit rester en attente jusqu'à ce qu'une donnée ait été produite. Inversement, si le magasin est plein, l'utilisateur ne peut plus produire. Sa demande doit rester en attente jusqu'à ce qu'une donnée ait été consommée.
Une représentation graphique de l'ensemble est donnée par la figure 1.9.

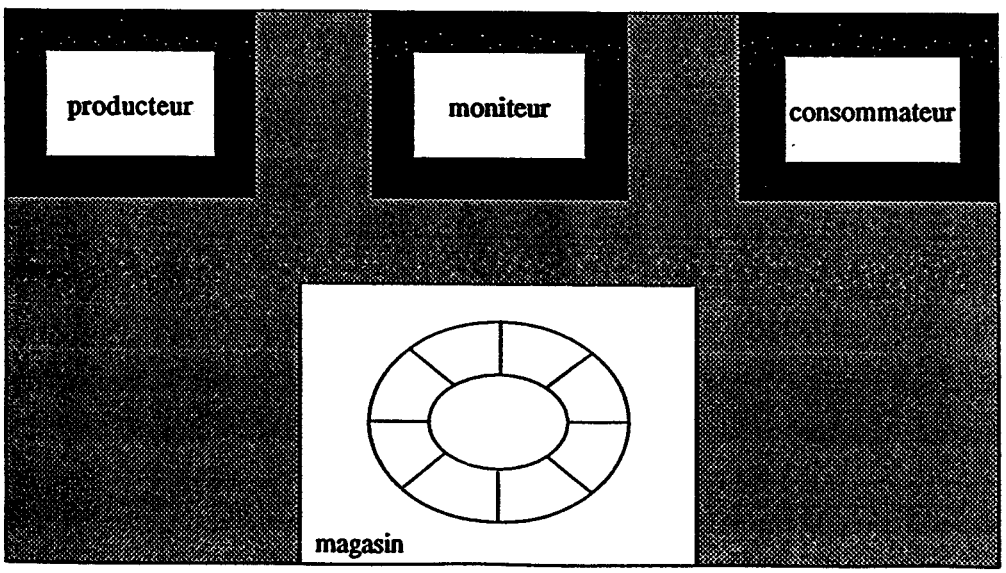


Figure 1.9

La structure d'animation associée à cette application est représentée par la figure 1.10 :

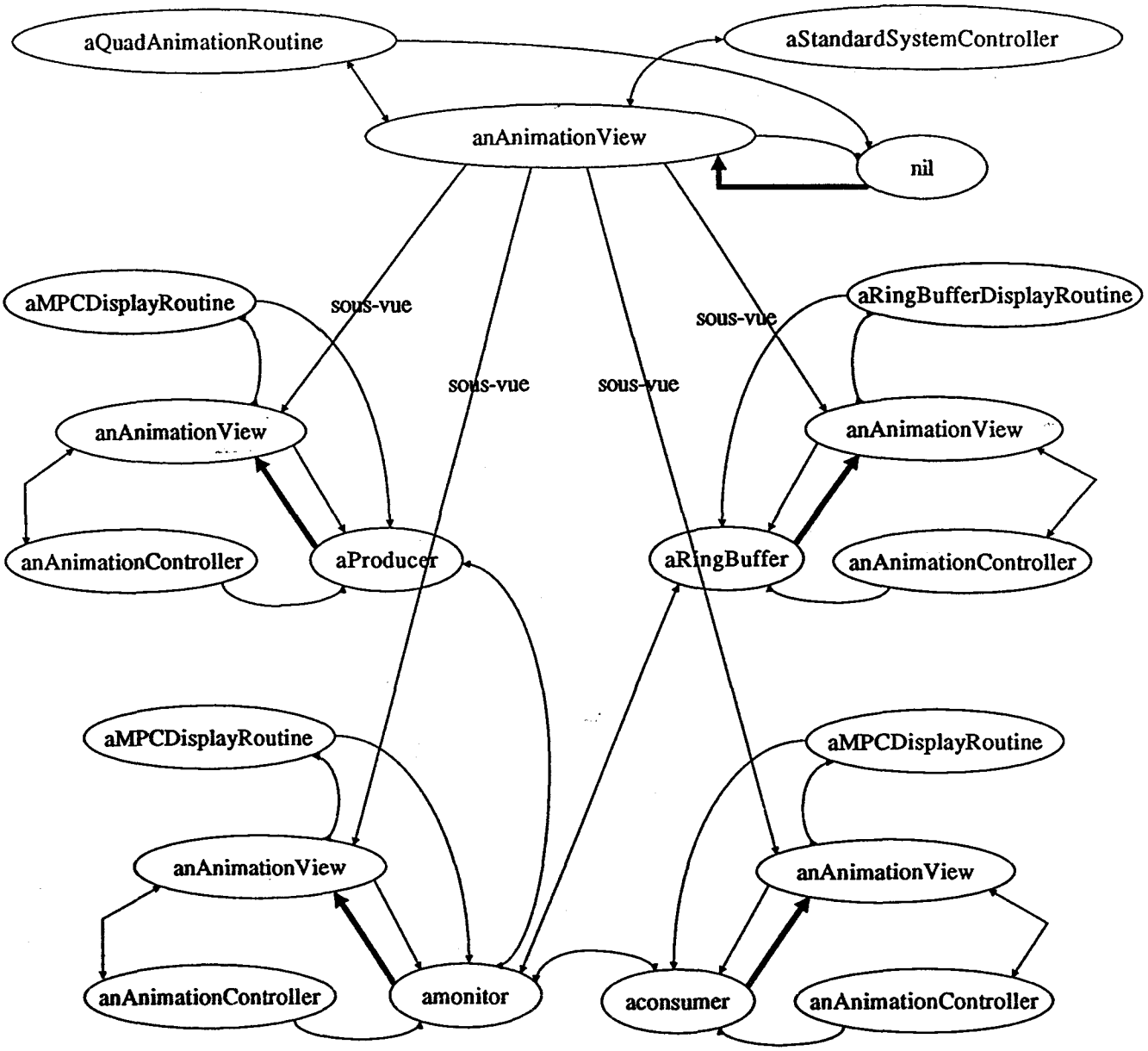


Figure 1.10

La complexité de la structure d'animation obtenue pour cet exemple, qui ne met en relation que quatre types d'objets principaux, laisse présager des difficultés non seulement de définition mais surtout de maintenance des applications bâties sur ce modèle.

CONCLUSION

Dans ce chapitre, nous avons dressé une liste des relations les plus couramment utilisées en programmation orientée objet, spécialement en Smalltalk-80. Cette liste, qui n'a pas la prétention d'être complète, souligne cependant l'absence de relations complexes capables de modéliser, complètement et simplement, les interactions entre objets.

Différents axes de recherche ont été ou sont encore explorés pour tenter de répondre à ce problème. Parmi ceux-ci, l'intégration de la notion de contrainte dans la programmation orientée objet, déjà abordée dans certains travaux (conception des systèmes ThingLab et Animus) semble être l'un des axes de recherche dont on puisse attendre le plus.

CHAPITRE 2 : CONTRAINTES ET OBJETS CONTRAINTS

CONTRAINTES ET OBJETS CONTRAINTS

L'énoncé d'un problème exprime souvent quantité de contraintes à résoudre sur un ensemble de données.

Les deux premières parties de ce chapitre définissent les notions de contrainte et d'objet contraint. Il existe de nombreuses façons de contraindre un objet, la contrainte pouvant décrire le comportement d'un objet ou sa structure de composition. Nous distinguerons quatre types d'objets contraints, l'objet simple, l'objet simple appartenant à une hiérarchie, l'objet composé et l'objet composé appartenant à une hiérarchie.

Les paragraphes suivants donnent une classification des contraintes selon qu'elles s'expriment sur tous les objets d'une classe ou sur une instance en particulier, qu'elles soient statiques ou dynamiques, permanentes ou à durée limitée.

Enfin, la dernière partie de ce chapitre introduit les principes de base du mécanisme de résolution des contraintes.

2.1. LA NOTION DE CONTRAINTE

La notion de contrainte est une notion si vaste qu'il est difficile de la définir très précisément. Le terme 'contrainte' s'emploie dans bon nombre de cas, quel que soit le problème posé, dans quelque domaine qu'il soit.

On parle

* de contraintes de calcul des quantités de produits à usiner compte tenu d'une production à atteindre,

* de contraintes de conception pour l'agencement d'un atelier sachant que tel type de pièces passera du poste A à un poste B et tel autre du poste A aux postes C, puis B ; que pour passer par le poste C, une pièce doit d'abord avoir été traitée sur un poste E ...,

* de contraintes de parallélisme pour les figures géométriques,

* de contraintes de niveau pour le remplissage d'une cuve ou de débit pour l'ouverture d'une vanne,

* de contraintes de fonctionnement pour les machines, ...

Dans tous les cas possibles et imaginables, la contrainte représente une condition à remplir pour atteindre un but, un impératif auquel on soumet un ou plusieurs objets.

Comment définir rigoureusement la notion de contrainte ?

Une contrainte est définie par un prédicat qui établit une relation sur un ensemble de variables. Nous parlerons ici de variables et non pas de données car la notion de contrainte suggère la recherche d'une solution à un problème posé, cette solution n'étant pas toujours unique.

La formulation de la contrainte doit donc rester déclarative, l'instanciation des variables se faisant dans une étape ultérieure à la définition de la contrainte, lors de l'initialisation ou de la résolution.

Par exemple, dans le domaine de la géométrie, on dira que le point milieu d'un segment de droite est soumis à une contrainte de prédicat *'coordonnées-du-point-milieu = 1/2 (coordonnées-de-la-1e-extrémité + coordonnées-de-la-2e-extrémité)'*. Cette contrainte est purement déclarative. Elle peut s'appliquer à tout segment de droite ayant un point milieu, quelque soit ses coordonnées. Seule l'instanciation des variables *coordonnées-de-la-1e-extrémité*, et *coordonnées-de-la-2e-extrémité* nous ramène à un segment précis.

La solution à une contrainte est une combinaison d'affectations qui vérifie son prédicat. Cette solution étant rarement unique, résoudre une contrainte, revient à donner une des combinaisons possibles.

On distingue différents cas de résolution d'une contrainte :

1°) La contrainte a été initialisée mais certaines de ses variables ne sont pas instanciées. Dans ce cas une solution à la contrainte est une combinaison des affectations déjà connues et d'affectations obtenues par résolution.

Soit, par exemple, la contrainte *'a + b = c'* définissant un additionneur à deux entrées. Cette contrainte est établie sur un ensemble de trois données ayant pour domaines respectifs les ensembles {1...4}, {1...6} et {1...12}. La variable *a* est affectée à la première donnée, la variable *b* à la seconde , *c* à la troisième. Si à l'initialisation, deux de ces variables sont instanciées, la résolution de la contrainte donnera une valeur à la troisième variable. Dans ce cas la solution est unique (si *a* vaut 3 et *c* vaut 7 alors le chiffre 4 sera affecté à *b*). Si une seule variable est instanciée, dans ce cas la résolution de la contrainte permet de retenir plusieurs solutions (si *b* vaut 6, *a* et *c* pourront être instanciées respectivement à (1,7), (2,8), (3,9) ou (4,10)).

2°) Toutes les variables de la contrainte sont instanciées mais la contrainte n'est pas satisfaite. Cette situation peut se produire si les variables sont mal initialisées ou si une nouvelle valeur a été affectée à l'une des variables (la combinaison des affectations alors obtenue n'est pas toujours solution de la contrainte).

Dans ce cas, la résolution de la contrainte va entraîner une modification des autres variables manipulées par celle-ci (de toutes ces variables ou seulement d'un certain nombre d'entre elles selon la méthode de résolution).

Dans l'exemple cité précédemment, supposons qu'à l'initialisation *a* vaut 3, *b* vaut 4 et *c* vaut 7. Puis en cours de processus, la variable *b* est affectée à 2. La contrainte *'a + b = c'* n'est plus satisfaite. Pour résoudre le problème on peut affecter une nouvelle valeur à la variable *a* ou à la variable *c*, affecter de nouvelles valeurs à la fois à *a* et à *c* ou modifier de nouveau la valeur affectée à *b*. Si tous ces moyens tendent à résoudre la contrainte, certains sont plus envisageables que d'autres, dans le cas présent la solution la plus naturelle étant de modifier soit la variable *a* soit la variable *c*.

Le choix de la méthode à employer pour résoudre une contrainte est fonction de la technique de résolution adoptée. Le chapitre 4 présentent plusieurs techniques de résolution des contraintes.

Remarque: Dans les deux cas envisagés, la contrainte peut ne pas être satisfaite, soit par manque d'informations, si le nombre de variables instanciées ne suffit pas à déterminer l'ensemble des variables non affectées , soit parce que la contrainte ne possède pas de solution. Supposons, dans notre exemple, que la variable *c* soit initialisée à la valeur 11, dans ce cas il n'existe pas de solution à la contrainte.

De façon plus générale et quel que soit le formalisme adopté, une contrainte définit une relation sur un ensemble de variables, relation que nous noterons P(a,...,x), où P est le prédicat de la contrainte et {a,...,x} l'ensemble des variables qu'elle met en cause. Cette relation peut être de tout ordre, pour peu qu'elle impose une action, une attitude ou un état à l'objet ou à l'ensemble des objets auxquels elle est attachée. Le terme "imposer" est important car il souligne l'aspect systématique de la résolution. Chaque fois qu'une nouvelle valeur est affectée à l'une des variables de la contrainte, il vaut vérifier si la contrainte est toujours satisfaite et, si elle ne l'est pas, il faut résoudre la contrainte (§ 2.7).

2.2. LA NOTION D'OBJET CONTRAINT

L'état d'un objet est donné par l'ensemble des valeurs affectées à ses variables (variables d'instance ou variables de classe). Or la contrainte étant précisément une relation sur un ensemble de variables, rien n'interdit de penser que les notions de contrainte et d'objet puissent être associées.

Jusqu'alors peu de langages de programmation orientée objet ont intégré la notion de contrainte. On remarque d'autre part que les langages existant sont fortement liés au domaine d'application pour lequel ils ont été développés. Ainsi par exemple, ThingLab [BOR] et Animus [DUI] sont des outils destinés à l'animation ou la simulation graphique de petits systèmes. Coral [SZE] ou Grow [BAR] sont des outils d'aide à la construction d'interfaces graphiques. D'autres outils encore sont plus spécialement destinés à la définition et à la gestion du multifenêtrage [EPS] ou au D.A.O. (dessin assisté par ordinateur) [DAH][FER].

Cette spécificité des outils n'est pas due au hasard. Nous verrons dans les chapitres suivants, que l'obstacle principal à l'utilisation des contraintes est l'explosion combinatoire qui rend difficile parfois impossible la résolution de l'ensemble des contraintes définies sur un système (on parlera de réseau de contraintes). Les systèmes

contraints utilisent de ce fait un domaine spécifique de la connaissance pour réduire la durée des temps de calcul nécessaire à la résolution.

Cependant la notion d'objet contraint peut être généralisée et s'appliquée à tout objet, pour décrire son comportement en tant qu'individu ou en tant que partie d'un tout. Le moyen le plus simple pour définir les différents types d'objets contraints est de suivre la classification des objets dits classiques. On obtient alors quatre types d'objets contraints :

- l'objet simple soumis à une contrainte intrinsèque statique ou dynamique. Dans ce cas, la contrainte est définie sur les variables d'état de l'objet contraint soit à sa définition (contrainte statique), soit dans le déroulement de l'application (contrainte dynamique) ;

- l'objet simple appartenant à une hiérarchie d'objets contraints ;

- l'objet composé soumis une contrainte pouvant être définie à différents niveaux : on parlera de contrainte intrinsèque si elle est définie sur les variables d'état de l'objet tout et de contrainte extrinsèque si elle est définie entre plusieurs composantes de l'objet tout ;

- l'objet composé appartenant à une hiérarchie d'objets.

2.2.1) L'objet contraint simple

L'objet contraint simple est un objet sans composante sur lequel on définit une ou plusieurs contraintes intrinsèques statiques décrivant son comportement.

Par objet sans composante, on désigne les objets dont les variables d'instance réfèrent toujours à des données primitives (une valeur numérique ou un symbole). C'est le cas simple de l'additionneur à deux entrées. Cet objet se définit par trois variables (**entrée1, entrée2, somme**) sur lesquelles est établie la contrainte de prédicat *'entrée1 + entrée2 = somme'*. Cela pourrait tout aussi bien être une contrainte définie sur une vanne, objet dont le débit est fonction du degré d'ouverture, ou sur une résistance électrique, objet sur lequel est définie la loi d'Ohm.

Il est relativement facile de gérer la contrainte définie sur un objet simple puisqu'il en est directement "propriétaire". Lorsque l'une des variables de l'objet simple est

modifiée, il faut vérifier s'il répond toujours à sa contrainte (éventuellement à ses contraintes). Si la réponse est négative, il suffit de corriger les autres variables de l'objet simple de sorte qu'il respecte de nouveau la contrainte.

2.2.2) Cas de l'objet simple appartenant à une hiérarchie

Le principe de l'héritage a déjà été exposé dans le premier chapitre : un objet appartenant à une hiérarchie d'objets hérite de toutes les variables et de toutes les méthodes définies dans cette hiérarchie. Il est donc logique de penser qu'il puisse hériter également des contraintes définies sur les variables dont il hérite.

Reprenons l'exemple de l'additionneur à deux entrées et supposons que l'on veuille définir un objet à double emploi, à la fois additionneur et multiplicateur. L'objet additionneur/multiplicateur hérite de l'objet additionneur (pour les trois variables d'instance **entrée1, entrée2 et somme**) et possède en plus la variable d'instance **produit.** Sur ce nouvel objet, est définie la contrainte *'entrée1 * entrée2 = produit'.*

Deux stratégies sont alors possibles :

Soit, par mesure de facilité, on décrète que la contrainte ne peut être héritée. Dans ce cas , il faudra établir de nouveau sur l'objet additionneur_multiplicateur la contrainte relative à l'addition, *'entrée1 + entrée2 = somme'*. Cette démarche n'est pas rare même si elle "renie" un des principes fondamentaux de la programmation orientée objet. En fait, elle permet de passer outre les problèmes de conflits entre contraintes héritées et contraintes nouvellement définies. S'il y a incompatibilité entre les différentes contraintes établies sur un objet, l'utilisateur seul pourra être mis en cause. Cette démarche permet également à un objet de ne pas hériter de certains comportements contraints imposés dans sa hiérarchie.

Soit on considère que la contrainte est une donnée dont on peut hériter, au même titre que les variables ou les méthodes définies dans la hiérarchie. Cette démarche, quoique plus naturelle n'est pas sans poser de problèmes. D'abord elle impose qu'un objet hérite obligatoirement des comportements contraints définis pour les objets de sa hiérarchie. C'est un fait à admettre et qui demande simplement à l'utilisateur une plus grande prudence dans la définition de ses classes d'objets contraints. Mais le principal problème réside dans la gestion des conflits entre contraintes. Ces conflits peuvent être de différentes natures. D'abord il peut advenir que certaines contraintes soient redondantes. Même s'il n'y a pas vraiment de conflit dans ce cas précis, la redondance

peut coûter cher en temps de calcul et nous verrons par la suite que ce temps est trop précieux pour être ainsi gaspillé. Ensuite, il peut y avoir conflit apparent entre deux contraintes. Si à un certain niveau de la hiérarchie une contrainte impose qu'un objet soit noir et que, à un niveau plus bas, une autre contrainte impose qu'il soit blanc, le problème est sans solution. Enfin, il peut arriver qu'un conflit ne soit pas apparent (qu'une contrainte ne soit pas la négation d'une autre) mais que l'ensemble des contraintes définies sur un objet ne permette pas d'aboutir à une solution.

Ces problèmes de conflits entre contraintes ne sont pas simples à résoudre. Si on peut reconnaître facilement deux contraintes identiques (de même prédicat) ou deux contraintes opposées (affectant chacune une valeur différente à la même variable), il est quasi impossible de découvrir, à priori, les autres conflits. Seule une tentative de résolution des contraintes peut, dans ce cas, mettre à jour les conflits.

2.2.3) L'objet composé contraint

Un objet composé met en relation plusieurs objets qui peuvent être simples ou eux-mêmes composés.

On désignera, dans ce paragraphe, par objet composé tout objet dont les variables réfèrent à d'autres objets non primitifs. Le cas de la composition d'objets par définition de contraintes entre objets sera traité au paragraphe 2.3.3.

Par exemple, l'objet ligne_horizontale définie par deux variables d'instance **(extrémité1, extrémité2)** est un objet composé. Ces variables font référence chacune à un point, sur lesquels on impose la contrainte d'alignement *'extrémité1 y = extrémité2 y'*. A partir de cette ligne, il est possible de créer un autre objet composé, la ligne_horizontale_à_point_milieu constitué par deux composantes, une ligne_horizontale et un point, référencées respectivement par les variables d'instance **(lignehz, pointml)**. Sur ces variables, on définit la contrainte *'pointml = 1/2 (lignehz extrémité1 + lignehz extrémité2'*, plaçant le point à égale distance des deux extrémités de la ligne.

L'exemple cité montre qu'il faut, dans le cas d'objets composés, distinguer les contraintes uniquement relatives aux composantes, des contraintes relatives à l'objet composé lui-même. La contrainte d'alignement *'extrémité1 y = extrémité2 y'* est propre à la composante ligne_horizontale, alors que la contrainte *'pointml = 1/2 (lignehz extrémité1 + lignehz extrémité2'* est relative à l'objet composé.

2.2.3.1) Contrainte définie sur une composante

2.2.3.1.a) cas d'une contrainte non réactive

C'est le cas d'une contrainte affectant exclusivement la composante sur laquelle elle est définie et dont la résolution n'affecte pas les autres parties de l'objet composé.

Supposons que sur l'objet ligne_horizontale, composante de l' objet ligne_horizontale_à_point_milieu, soit établie une contrainte de couleur : en plus des variables **extrémité1 et extrémité2**, on a définit pour la ligne_horizontale, une troisième variable **couleur** sur laquelle est établie la contrainte *'couleur = rouge'*. La résolution de cette contrainte ne peut en aucun cas affecter les autres composantes (ici le point_milieu), ni l'objet composé en tant que tout.

Les contraintes non réactives sont très faciles à gérer, il suffit lorsqu'une composante d'un objet composé est modifiée de vérifier, et au besoin de satisfaire, les contraintes qui lui sont propres.

2.2.3.1.b) cas d'une contrainte réactive

C'est le cas d'une contrainte définie sur une composante et dont la résolution affecte d'autres parties de l'objet composé, voire l'objet composé dans sa globalité. Il faut alors généraliser la demande de satisfaction de la contrainte à la demande de satisfaction d'un réseau de contraintes, constitué des contraintes définies sur la partie modifiée et des contraintes définies sur les parties qui lui sont connectées.

Si l'une des extrémités de notre ligne_horizontale est déplacée verticalement. La contrainte d'horizontalité n'est plus respectée (figures 2.1.a, 2.1.b). Pour satisfaire de nouveau la contrainte, il faut déplacer de la même manière l'autre extrémité de la ligne (figure 2.1.c). Dans ce cas, les coordonnées du point milieu sont totalement fausses et doivent être recalculées à partir des nouvelles coordonnées de la ligne (figure 2.1.d). La contrainte d'horizontalité *'extrémité1 y = extrémité2 y'*, définie sur la composante ligne_horizontale, est donc une contrainte réactive, puisque sa résolution a affecté d'autres composantes que celle pour laquelle elle a été définie.

L'exemple donné, pourtant très simple, souligne deux difficultés relatives à l'introduction des contraintes dans la programmation orientée objet :

- la nécessité, dans le cas de contraintes réactives, d'étendre la résolution des contraintes d'une composante au reste de l'objet composé, ou du moins à une partie de celui-ci (démarche ascendante). ;

- la nécessité de savoir, lorsqu'un objet est modifié, s'il est composante d'un autre objet de façon à répercuter le changement sur celui-ci. Dans notre exemple (figures 2.1.b et 2.1.c), si on rétablit la contrainte définie sur la ligne_horizontale, sans savoir que cette ligne fait partie d'un objet composé, les coordonnées du point_milieu resteront inchangées (puisque la contrainte *'pointml = 1/2 (lignehz extrémité1 + lignehz extrémité2)'* n'appartient pas à la composante ligne_horizontale). Sans cette information, il n'est pas possible de propager la résolution des contraintes de la composante vers l'objet composé.

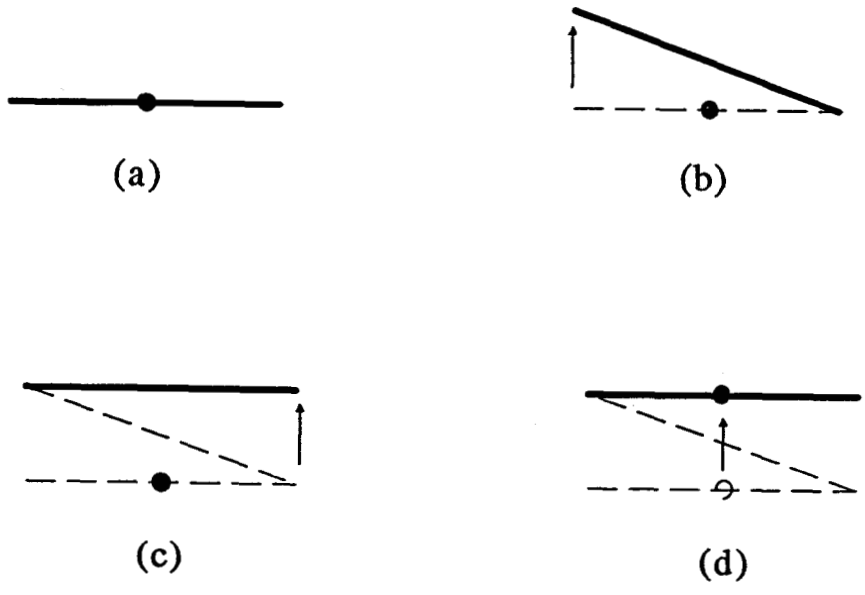


Figure 2.1

2.2.3.1.c) cas particulier de la contrainte d'ancrage

Une ancre est une contrainte qui interdit tout changement de la variable sur laquelle elle est définie. Supposons qu'une ancre soit définie sur un objet, si, pour répondre à une action sur l'objet contraint, les autres contraintes ne peuvent être satisfaites sans briser l'ancre, alors l'action est refusée.

2.2.3.2) contrainte définie sur l'objet composé

Il est possible de définir deux types de contraintes sur l'objet composé :

- Les contraintes établies sur l'objet composé et n'affectant pas ses composantes. Pour ce type de contraintes, on retombe sur le cas déjà étudié de la contrainte non réactive (§ 2.2.3.1.a), ou de la contrainte définie sur un objet simple (§ 2.2.1).

- Les contraintes relatives à la composition des différentes parties constituant l'objet composé.
Dans notre exemple, la contrainte *'pointm1 = 1/2 (lignehz extrémité1 + lignehz extrémité2)'* est une contrainte établie sur l'objet composé ligne_horizontale_à_point_milieu définissant la règle de composition entre la ligne et le point.

Pour satisfaire ce dernier type de contraintes, il faut avoir une démarche descendante. Lorsque l'on résout une contrainte définie sur un objet composé, si cette résolution modifie une des composantes, il faut alors vérifier et satisfaire les contraintes définies sur cette composante.

2.2.3.3) Exemple

L'exemple du quadrilatère inclut les différents cas de contraintes définies sur un objet composé.

On définit le quadrilatère comme un objet constitué de quatre composantes, chacune d'elles étant une ligne_à_point_milieu. Ces composantes sont référencées par les variables **lpm1, lpm2, lpm3 et lpm4.** La structure complète de l'objet quadrilatère est donnée par la figure 2.2 .

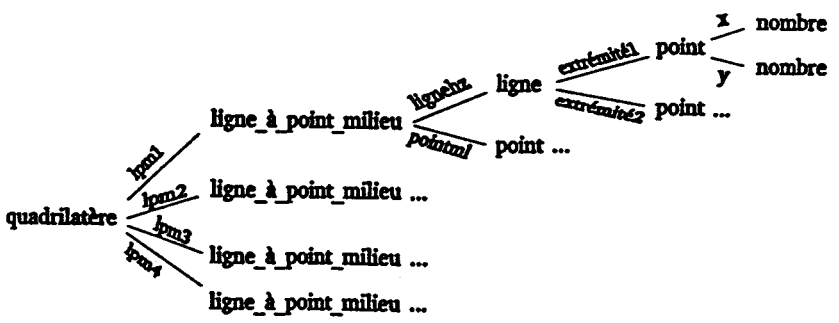


Figure 2.2

Sur cet objet, les contraintes sont définies à différents niveaux :

- Au niveau du quadrilatère, on définit la contrainte de composition des quatre cotés *'lpm1 extrémité1 = lpm2 extrémité2. lpm2 extrémité1 = lpm3 extrémité2. lpm3 extrémité1 = lpm4 extrémité2. lpm4 extrémité1 = lpm1 extrémité2.'*

- Sur l'objet ligne_à_point_milieu, on définit la contrainte *'pointml = 1/2 (lignehz extrémité1 + lignehz extrémité2)'.*

Vis à vis de l'objet quadrilatère, cette contrainte est définie sur une composante. Elle n'est pas réactive dans le sens où, pour résoudre la contrainte, on choisira toujours de recalculer les coordonnées du point milieu.
La ligne_à_point_milieu se composant elle-même de deux objets, c'est donc aussi une contrainte définie sur un objet composé.

Supposons le quadrilatère initialisé (figure 2.3.a). Si on déplace l'une des extrémités de la ligne **lpm4** (figure 2.3.b), elle n'obéit plus à sa contrainte. Les coordonnées de son point milieu sont donc recalculées (figure 2.3.c). La ligne_à_point_milieu étant un objet composé, il faut regarder, au niveau de ses composantes, si celles-ci répondent toujours à leurs propres contraintes (démarche descendante). Dans le cas présent, les composantes de la ligne_à_point_milieu n'ont pas de contrainte propre. D'autre part, la ligne **lpm4** étant aussi une composante de l'objet quadrilatère, il faut vérifier si l'objet composé lui-même obéit toujours à ses contraintes (démarche ascendante). Pour résoudre sa contrainte, le quadrilatère modifie les coordonnées de la ligne **lpm1** (figure 2.3.d). Le quadrilatère étant lui-même un objet composé, il faut de nouveau vérifier si ses composantes vérifient toujours leurs propres contraintes. La ligne **lpm1** n'obéissant plus à sa contrainte, les coordonnées de son point milieu sont recalculées (figure 2.3.e).

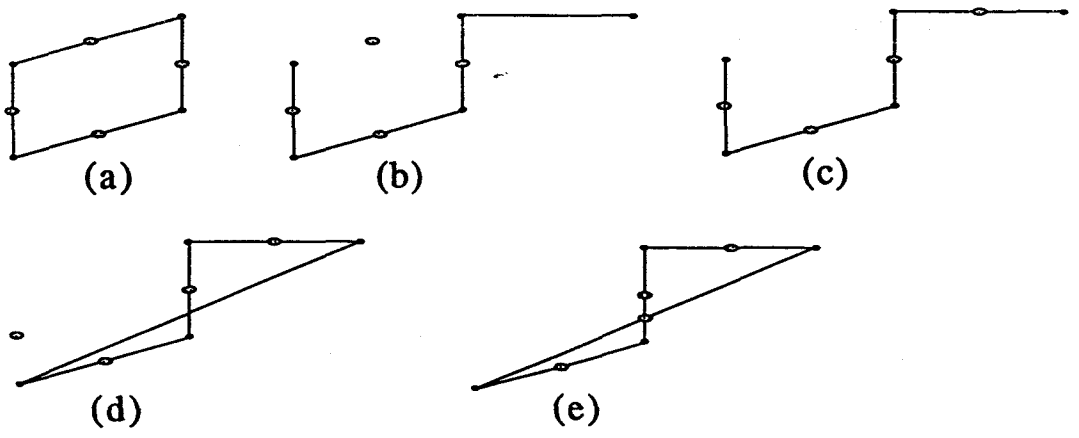


Figure 2.3

2.2.4) L'objet composé appartenant à une hiérarchie

Le paragraphe précédant ne considère que les objets composés dont la structure de composition est définie au moyen de variables d'instance. Dans ce cas, le problème de l'héritage des contraintes pour un objet composé appartenant à une hiérarchie est identique à celui formulé pour l'objet simple appartenant à une hiérarchie (§ 2.2.2). Puisqu'un objet hérite des variables d'instance définies dans sa hiérarchie, un objet composé héritera nécessairement des composantes constituant les objets de ses super-classes et des contraintes définies sur celles-ci.

Dans le chapitre I, nous avons vu que l'utilisation des variables d'instance pour la composition d'objets n'est envisageable que pour de petits systèmes (nombre de classes et de variables réduits). Dans le cas de systèmes complexes, il est préférable de définir l'objet composé comme un ensemble d'objets sur lesquels sont établies directement des relations de composition. Pour ce faire, il est nécessaire de pouvoir définir des contraintes entre instances.

2.3. CONTRAINTES DE CLASSE ET CONTRAINTES D'INSTANCE

2.3.1) Contraintes de classe

Une contrainte de classe est une contrainte qui s'applique à toutes les instances de la classe. Elle est déclarée, de façon unique, à la création de celle-ci et définit une relation sur les variables d'instance.

La contrainte de classe est une contrainte statique utilisée essentiellement pour définir le comportement intrinsèque de ses instances. C'est la seule contrainte qui peut être reçue par héritage.

Par exemple on définira pour la classe Résistance, sur les variables d'instance (u_1, u_2, r, i), la contrainte de classe '$u_2 - u_1 = r * i$', puisque la loi d'Ohm s'applique de la même façon à toutes résistances.

2.3.2) Contraintes d'instance

Une contrainte d'instance peut être établie et retirée, à tout moment, sur un l'objet ou sur un ensemble d'objets instanciés.

Elle doit pouvoir être définie sur un objet même s'il n'appartient pas à une classe contrainte. Si c'est le cas, cet objet aura un comportement identique à celui d'une instance de classe contrainte, à savoir qu'à chaque modification d'une de ses variables d'instance il cherchera à vérifier sa contrainte. Si une contrainte d'instance est définie sur un objet possédant déjà une contrainte de classe, l'ensemble des contraintes définies sur l'objet constituera un réseau de contraintes sur ses variables d'instance.

2.3.2.1) Contrainte définie sur une instance

Une contrainte d'instance, définie sur un objet, établit une relation entre ses variables d'instance , relation qui ne s'applique qu'à cet objet en particulier indépendamment des autres instances de sa classe.

Ce type de contraintes est utilisé pour donner un comportement spécifique à un objet, sans pour cela devoir définir une nouvelle classe. Elle ne décrit pas le comportement intrinsèque de l'objet (qui serait partagé par tout objet de même nature), mais son comportement particulier dans un contexte donné.

C'est le cas par exemple lorsque l'on impose une trajectoire à un point ou lorsque l'on ancre une variable d'instance (une ancre est une contrainte définie sur un objet qui interdit toute modification de la variable d'instance sur laquelle elle est établie).

2.3.2.2) Contrainte établie entre instances

Une contrainte, définie sur plusieurs objets, établit une relation entre leurs variables d'instance respectives. Ce type de contraintes est essentiellement utilisé pour faire de la composition d'objets sans avoir recourt à l'utilisation des variables d'instance.

Par exemple, dans la modélisation d'un système de production, pour associer un capteur de niveau à une cuve, il suffira d'établir entre ces deux objets la contrainte *'niveau = grandeur_captée'*, où **niveau** est une variable d'instance appartenant à l'objet cuve et **grandeur_captée** une variable d'instance de l'objet capteur.

2.4. CONTRAINTES STATIQUES ET CONTRAINTES DYNAMIQUES

Jusqu'alors dans nos exemples, nous n'avons cité que des cas de contraintes statiques qu'elles soient intrinsèques (définies sur les variables d'instance d'un objet) ou extrinsèques (définies sur les différentes parties d'un tout). Mais la contrainte peut tout aussi bien être établies dynamiquement. Dans ce cas cependant, il ne pourra s'agir que de contraintes d'instance.

Contraintes dynamiques établies en régime contrôlé

Reprenons l'exemple de la modélisation et de la simulation interactive d'un système de production. Une fois le système modélisé, l'utilisateur désire faire des essais de commande. Il initialise différents paramètres et pilote le processus en imposant des contraintes sur ses éléments, suivant les résultats qu'il obtient. Le but de sa démarche est de trouver, par essais successifs, les commandes optimales et l'ensemble des contraintes à imposer sur le processus pour effectuer une mission donnée.

Contraintes dynamiques établies en régime autonome

Pour assurer à l'installation toutes garanties de sécurité et à la simulation un maximum de réalisme, un mécanisme de simulation de défaillances est intégré à l'outil de simulation. La possibilité d'un défaut de fonctionnement sur un outil suit une loi de probabilité, élevée lorsque cet outil est neuf , faible lorsqu'il est dans l'intervalle de durée de vie fixé par le constructeur, puis élevée de nouveau lorsque l'outil est usagé. Se basant sur cette loi, le mécanisme de simulation des défaillances provoquera un défaut de fonctionnement sur tel ou tel élément, en lui imposant un comportement inhabituel (contrainte définie en régime autonome, selon une loi continue).

Supposons maintenant que le système soit au point (modélisation, simulation et commandes) et que l'on utilise cet outil pour l'apprentissage des pilotes. L'apprentissage est basé sur un scénario de pannes, imposant aux éléments de l'installation des contraintes de mauvais fonctionnement (contraintes définies en régime autonome, suivant un schéma contrôlé). Puis, suivant les réactions du pilote face aux défaillances, le système réagit en imposant de nouvelles contraintes sur le système (contraintes définies en régime autonome suivant une relation événementielle).

2.5. CONTRAINTES A DUREE DE VIE LIMITEE

Une contrainte peut avoir une durée de vie limitée, soit liée à un intervalle de temps, soit valable jusqu'à l'occurrence d'un événement.

Pour les contraintes dont la durée de vie est liée à un intervalle de temps, on peut citer toutes les contraintes liées à des comportements transitoires. Par exemple, les contraintes décrivant le comportement d'une machine au moment de sa mise en route.

Dans les contraintes à durée de vie limitée, on peut aussi prendre en compte les cas de commutation de contraintes. Pour les contraintes temporelles, on peut définir une contrainte sur chaque intervalle de temps (tracé d'une fonction par contrainte sur un point, pour les fonctions définies par parties). En utilisant le principe de commutation, il est également possible de définir des objets à comportement multiple, chaque comportement étant décrit par une contrainte (ou un ensemble de contraintes).

2.6. LES DIFFERENTS TYPES DE CONTRAINTES

Une contrainte est une relation qui peut être mathématique ou symbolique selon les données que l'on manipule. Elle peut donc, à priori, exprimer n'importe quel type de relation :

- égalitaire, comme la contrainte définie pour l'objet ligne_horizontale, '*extrémité1 y = extrémité2 y*' ;
- inégalitaire, comme la contrainte '|*extrémité1 x = extrémité2 x*|', définie pour les lignes dont la projection sur l'axe des x est inférieure à une limite ;
- ensembliste, comme la contrainte '|*extrémité1 x = extrémité2 x*| $\in [lim_{inf}\ lim_{sup}]$' définie sur une ligne ;
- fonction du temps, par exemple lorsque la position d'un mobile est régie par la contrainte '*x = sin(t)*' ;
- différentielle, lorsque le mouvement d'un mobile est contraint à une vitesse constante, '*dx/dt = cste*' ;
- complexe ou multiple.
- énumérative ...

l'absence de contrainte pouvant, dans cette classification, être considérée comme une relation neutre.

2.7. LA RESOLUTION DES CONTRAINTES

Le principal intérêt des contraintes est qu'elles expriment de façon déclarative toutes relations définies sur un objet ou entre objets. De façon déclarative, c'est à dire le plus simplement possible. Il ne serait donc pas logique de demander à l'utilisateur, après avoir définit les contraintes, d'en gérer la résolution. Cette tâche est celle d'un mécanisme interne, intégré dans le langage de programmation, appelé mécanisme de résolution des contraintes. Chaque fois qu'un objet contraint change d'état, ce mécanisme vérifie s'il obéit toujours à sa contrainte et, dans la négative, il résout la contrainte.

Ce paragraphe ne décrit pas les différentes techniques employées pour satisfaire les contraintes (Chapitre 4) mais donne un aperçu du fonctionnement du mécanisme de résolution.

2.7.1) Vérification des contraintes

Pour être dans un état stable, il faut toujours que l'ensemble des contraintes définies sur un objet soient satisfaites. Il faudra donc veiller

1°) à satisfaire ces contraintes lors de la création de l'objet contraint ou lors de son initialisation,

2°) à satisfaire ces contraintes chaque fois que l'état de l'objet contraint est modifié, donc, chaque fois qu'il y modification de la valeur d'une variable d'instance, si l'on se place au niveau de l'objet simple, ou chaque fois qu'il y a action sur une composante si l'on se place au niveau de l'objet composé.

Dans chacun de ces deux cas, il faut lancer la procédure de résolution des contraintes, procédure qui se déroule en deux phases:

- **La vérification.** Ce point consiste à regarder si le prédicat de chacune des contraintes définies sur l'objet est vrai. C'est à dire regarder si l'objet contraint obéit, ou non, à l'ensemble de ses contraintes.

- **La résolution.** Si la vérification renvoie une réponse négative, cette étape consiste à trouver une solution globale à toutes les contraintes définies sur l'objet.

2.7.2) Résolution des contraintes

Plusieurs cas seront à envisager:

- *cas d'un objet simple soumis à une seule contrainte*

C'est le cas le plus simple, si la contrainte n'est pas satisfaite il suffit de la résoudre en veillant seulement à respecter le principe de moindre étonnement et la volonté de l'utilisateur.

Exemple : Soit une ligne_à_point_milieu dont l'état initial est présenté figure 2.4.a. Si l'une des extrémités de la ligne est déplacée comme l'indique la figure 2.4.b, la contrainte n'est plus vérifiée. Pour satisfaire de nouveau la contrainte, il faut soit recalculer les coordonnées du point milieu (figure 2.4.c), soit effectuer sur l'autre extrémité un déplacement inverse (rotation de la ligne autour du point milieu, figure 2.4.d), soit revenir à l'état initial.

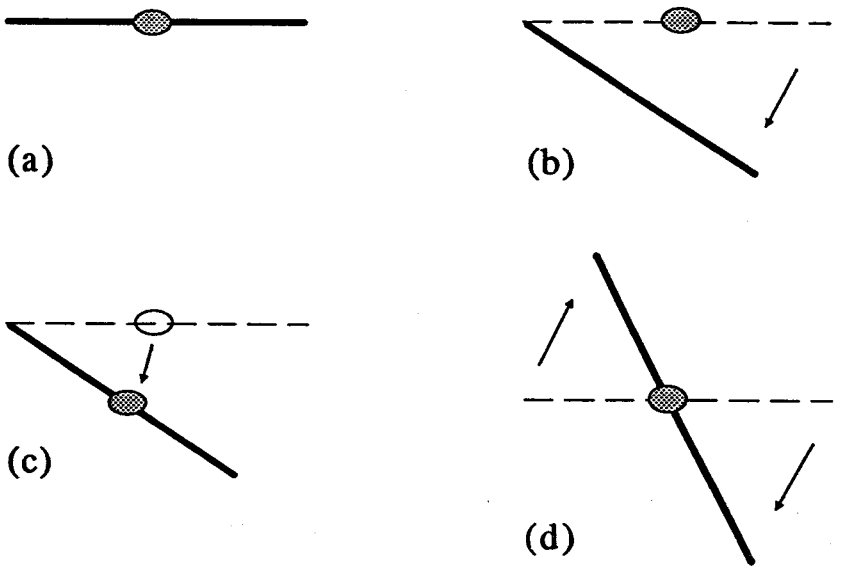


Figure 2.4

En déplaçant une extrémité de la ligne, on peut supposer que l'utilisateur s'attende à voir modifier, pour satisfaire la contrainte, la position du point milieu plutôt qu'à voir se déplacer l'autre extrémité (moindre étonnement) ou à revenir à l'état initial (respect de la volonté de l'utilisateur).

- Cas d'un objet simple soumis à plusieurs contraintes ou d'un objet composé

Dans ce cas, il ne faut plus prendre en compte une seule contrainte mais un réseau de contraintes.

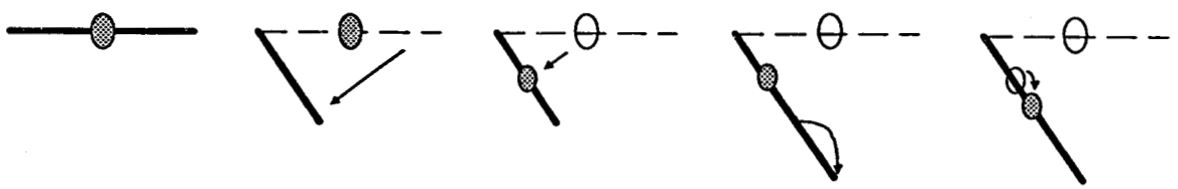
Plusieurs possibilités peuvent se présenter:

1°) les contraintes du réseau peuvent être satisfaites simultanément.

Le problème est alors de trouver une séquence optimale des différentes méthodes à appliquer pour résoudre toutes les contraintes.

Exemple : Soit le cas d'une_ligne_à_point_milieu de longueur constante dont on déplace une extrémité, l'autre extrémité étant ancrée. Deux séquences de résolution peuvent être choisies (figures 2.5.a et 2.5.b). La première séquence recalcule d'abord les coordonnées du point_milieu, rétablit la longueur de la ligne et enfin recalcule de nouveau les coordonnées du point_milieu. La seconde séquence rétablit d'abord la longueur de la ligne puis recalcule ensuite les coordonnées du point_milieu. Cette séquence est optimale, la première ne l'est pas.

1ère séquence



2ème séquence

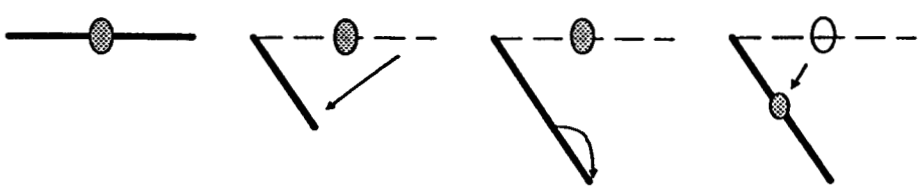


Figure 2.5

2°) l'objet est un objet **sur-contraint.**

C'est le cas lorsqu'il y a conflit entre deux contraintes, qui ne peuvent de ce fait être satisfaites simultanément.

L'intérêt pour l'utilisateur serait d'utiliser un système qui sache se sortir d'une telle situation sans faire systématiquement appel à lui. Plusieurs techniques permettent de se sortir des conflits entre contraintes, la plus utilisée étant la définition de priorités. Un ordre de priorité, défini sur les contraintes, donne la possibilité au système de laisser une contrainte insatisfaite au détriment d'une autre : lorsque deux contraintes sont en conflit, si l'une est prioritaire sur l'autre, le système cherchera à satisfaire celle-ci.

Exemple : Soit le cas d'une ligne_horizontale dont une extrémité est ancrée et dont l'utilisateur veut lier l'autre extrémité au déplacement de la souris.

L'objet ligne est soumis à trois contraintes:
- celle liant l'extrémité sélectionnée à la position de la souris,
- l'ancrage fixant l'autre extrémité,
- la contrainte d'horizontalité.

La contrainte d'horizontalité est une contrainte de forte priorité (elle décrit la nature même de l'objet) et ne peut rester insatisfaite. Le système doit donc décider s'il va renoncer à suivre la position de la souris et donc refuser l'action ou détruire l'ancrage sur l'autre extrémité. Si la contrainte d'ancrage est de priorité plus élevée que la contrainte de suivi de la souris, il optera pour la première solution, qui semble la moins étonnante.

S' il n'y a pas de possibilité de résolution, c'est à dire si les contraintes en conflit sont de même priorité, le système pourra faire appel à un algorithme de relaxation ou, en dernier recourt, à l'utilisateur. L'algorithme de relaxation se base sur une notion de distance [BOR] qui permet de déterminer le degré de satisfaction d'une contrainte (la distance est nulle si la contrainte est résolue, entière et plus ou moins élevée si non).

CONCLUSION

Ce chapitre donne les éléments de base pour l'introduction des contraintes dans la programmation orientée objet.

Il constitue une synthèse des recherches que nous avons menées sur les notions de contrainte et d'objet contraint.

Un langage à contraintes doit permettre à l'utilisateur de définir, de façon déclarative, certaines relations qu'un système doit satisfaire. L'utilisateur décrit comment doivent se comporter les différents objets du système, puis un mécanisme de résolution des contraintes prend le relais. C'est à lui que revient la tâche de satisfaire, en permanence, toutes les contraintes établies sur le système.

Nous nous appuierons sur les résultats obvenus dans ce chapitre pour définir l'expression des contraintes en programmation orientée objet (chapitre 3) et construire une méthode de propagation des contraintes adaptée au réseau d'objets contraints (chapitre 4).

CHAPITRE 3 : EXPRESSION DE LA CONTRAINTE

CHAPITRE 3 : EXPRESSION DE LA CONTRAINTE

Ce chapitre présente deux formalisations différentes des contraintes.

La première partie présente un formalisme développé dans le domaine des mathématiques appliquées. Il donne une définition formelle de la contrainte et des réseaux de contraintes.
Les contraintes décrites dans cette partie sont des contraintes non-constructives dont on peut seulement tester la validité. Elles sont définies sur un ensemble de variables ayant un domaine de définition fini et discret. Tester la validité d'une contrainte revient, dans ce cas précis, à vérifier si la contrainte est satisfaite pour une instanciation donnée de toutes ses variables.

La seconde partie présente un formalisme plus "descriptif" de la contrainte, développé en Intelligence Artificielle pour l'intégration des contraintes dans les langages de règles ou les langages à objets. Les contraintes décrites dans cette partie sont des contraintes constructives, définies de manière plus ou moins déclarative. Une contrainte constructive englobe deux types d'informations, le prédicat de la contrainte et un ensemble de moyens permettant de résoudre la contrainte si elle n'est pas satisfaite pour une instanciation donnée de ses variables.

3.1. LA THEORIE DES CONTRAINTES

Cette première partie présente une formalisation du concept de contrainte développée en mathématiques appliquées. Elle donne une définition formelle des contraintes et des réseaux de contraintes établis sur des variables ayant des domaines de définition finis et discrets.

Même si ce formalisme n'est pas directement transposable en programmation orientée objet, il est à la base de tous travaux sur les contraintes et nous a permis d'aborder, à travers les problèmes de cohérence des réseaux, les notions de propagation de contraintes.

3.1.1) Définitions et propriétés fondamentales [MON], [FRE]

Une contrainte est une relation définie sur un ensemble de variables :
- si elle est définie sur une unique variable, la contrainte est dite contrainte unitaire,
- si elle est définie sur deux variables, on parlera de contrainte binaire,
- si elle est définie sur un ensemble de n variables, de contrainte n-aire.

Plus précisément, soit deux variables x_1 et x_2 de domaines de définition respectifs X_1 et X_2.

$$x_1 \in X_1 = \{ x_{11}, x_{12}, x_{13}, ..., x_{1N1} \},$$
$$x_2 \in X_2 = \{ x_{21}, x_{22}, x_{23}, ..., x_{2N2} \}.$$

Si une contrainte existe entre les éléments de X_1 et X_2, toutes les paires (x_{1i}, x_{2j}) ne sont probablement pas permises. On appelle relation ou contrainte entre les ensembles X_1 et X_2, tout ensemble, noté P, de paires autorisées. On considérera de façon générale qu'une contrainte de l'ensemble X_1 vers l'ensemble X_2 n'est pas bijective :

$$P_{12} \neq P_{21}.$$

Toute contrainte P_{12}, de l'ensemble X_1 vers l'ensemble X_2, est un sous-ensemble de l'ensemble produit $X = X_1 * X_2$.
Si on affecte à chaque paire de l'ensemble produit X la valeur 1 ou 0 selon, respectivement, que cette paire appartienne ou non à une contrainte donnée, toutes les contraintes P_{12} possibles (soit $2^{N1.N2}$ relations) peuvent être caractérisées par un

nombre binaire de N1.N2 digits. Ces digits seront arrangés sous la forme d'une matrice N1×N2, $[P_{12,ij}]$, dont les colonnes correspondent aux éléments de X_2 et les lignes aux éléments de X_1 :

$$P_{12,ij} = 1 \text{ si et seulement si } (x_{1i}, x_{2j}) \in P_{12}.$$

Par exemple, soit $X_1 = \{ a, b \}$ et $X_2 = \{ c, d, e\}$ les domaines de définition de deux variables x_1 et x_2, et $P_{12} = \{ (a, c), (a, e), (b, d)\}$ une contrainte sur ces variables, la matrice binaire caractéristique de P_{12} est :

$$P_{12} = \begin{vmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{vmatrix} \quad <=> \quad \begin{cases} (a,c) \in P_{12}, & (a,d) \notin P_{12}, & (a,e) \in P_{12} \\ (b,c) \notin P_{12}, & (b,d) \in P_{12}, & (b,e) \notin P_{12} \end{cases}$$

On définit sur les contraintes les opérations suivantes :

- *l'inversion* : l'inverse de la contrainte P_{12}, P_{12}^{-1}, est donnée par la transposée de la matrice $[P_{12,ij}]$.

$$P_{12}^{-1} = [P_{12,ij}]^T = P_{21}$$

- *la négation*

$$\bar{P}_{12} = \neg P_{12} \text{ si et seulement si } \bar{P}_{12,ij} = \neg P_{12,ij} \ \forall i,j$$

- *l'union de deux contraintes*

$$P_{12} = P'_{12} \cup P''_{12} \text{ si et seulement si } P_{12,ij} = P'_{12,ij} \vee P''_{12,ij} \ \forall i,j$$

- *l'intersection de deux contraintes*

$$P_{12} = P'_{12} \cap P''_{12} \text{ si et seulement si } P_{12,ij} = P'_{12,ij} \wedge P''_{12,ij} \ \forall i,j$$

- *l'inclusion*

$$P_{12} \subseteq P'_{12} \text{ si et seulement si } P_{12,ij} \leq P'_{12,ij} \ \forall i,j$$

- *la contrainte nulle et la contrainte totale,*

$$\emptyset_{12} : \emptyset_{12,ij} = 0 \ \forall i,j \ ; \ U_{12} : U_{12,ij} = 1 \ \forall i,j$$

telles que, pour toute contrainte P_{12},

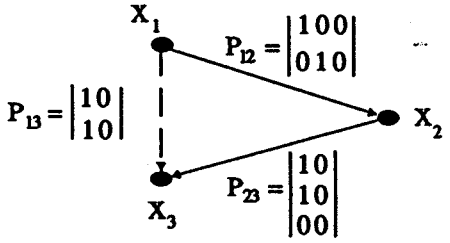
$$P_{12} \cup \emptyset_{12} = P_{12} \ ; \ P_{12} \cap U_{12} = P_{12} \ .$$

Supposons, maintenant, qu'il existe une contrainte P_{12} entre deux variables x_1 et x_2 et une contrainte P_{23} entre x_2 et une troisième variable x_3, existe-t-il une **contrainte induite** P_{13} entre les variables x_1 et x_3 ?
Si une telle contrainte existe, alors pour toute paire (x_{1i}, x_{3j}) de P_{13}, il existe au moins une valeur x_{2k} de X_2 telle que (x_{1i}, x_{2k}) et (x_{2k}, x_{3j}) appartiennent respectivement à P_{12} et P_{23}.

On définit ainsi une opération de *composition de contraintes* :

$$P_{13} = P_{12} \, . \, P_{23} \text{ ssi } P_{13,ij} = \bigvee_{k=1}^{N2} P_{12,ik} \wedge P_{23,kj} \quad \forall i,j$$

Par exemple, soit trois variables x_1, x_2 et x_3 de domaines de définition $X_1 = \{x_{11}, x_{12}\}$, $X_2 = \{x_{21}, x_{22}, x_{23}\}$ et $X_3 = \{x_{31}, x_{32}\}$. On définit sur ces variables les contraintes $P_{12} = \{(x_{11},x_{21}), (x_{12},x_{22})\}$ et $P_{23} = \{(x_{21},x_{31}), x_{22},x_{31})\}$, P_{12} est définie de X_1 sur X_2 et P_{23} est définie de X_2 sur X_3.
Par composition des deux contraintes P_{12} et P_{23}, on définit une contrainte induite P_{13}, de X_1 sur X_2, telle que $P_{13} = \{(x_{11},x_{31}), (x_{12},x_{31})\}$ comme illustré par la figure 3.1.



$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{vmatrix} * \begin{vmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 0 \end{vmatrix} = \begin{vmatrix} 1 & 0 \\ 1 & 0 \end{vmatrix}$$

$$P_{12} * P_{23} = P_{13}$$

Figure 3.1

L'opération de composition est associative et possède pour éléments neutres, les contraintes unitaires I_k :

$$I_{k,ij} = 1 \text{ si et seulement si } i = j \qquad \forall\, i,j \leq \text{dimension de } X_k$$

$$\text{telle que } P_{12} \,.\, I_2 = I_1 \,.\, P_{12} = P_{12} \,.$$

La composition de contrainte n'est pas distributive par rapport à l'intersection. En règle générale,

$$P_{12}.(P'_{23} \cap P''_{23}) \neq P_{12}.P'_{23} \cap P_{12}.P''_{23} \,.$$

Soit les contraintes P_{12}, P'_{23} et P''_{23} des contraintes définies sur les ensembles $X_1 = \{x_{11}, x_{12}\}$, $X_2 = \{x_{21}, x_{22}\}$ et $X_3 = \{x_{31}, x_{32}\}$.

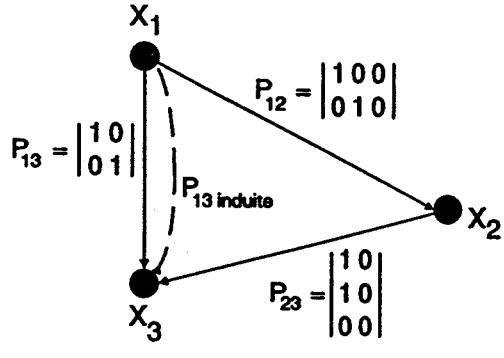
$$P_{12} = \begin{vmatrix} 1 & 1 \\ 0 & 0 \end{vmatrix} \; ; \; P'_{23} = \begin{vmatrix} 0 & 0 \\ 1 & 0 \end{vmatrix} \; ; \; P''_{23} = \begin{vmatrix} 1 & 0 \\ 0 & 0 \end{vmatrix}$$

$$P_{12}.(P'_{23} \cap P''_{23}) = \begin{vmatrix} 0 & 0 \\ 0 & 0 \end{vmatrix} \quad \text{et} \quad P_{12}.P'_{23} \cap P_{12}.P''_{23} = \begin{vmatrix} 1 & 0 \\ 0 & 0 \end{vmatrix}$$

D'autre part, s'il existe une contrainte directe P_{13} entre les variables x_1 et x_3 , on appellera **contrainte restreinte**, sur ces deux variables, la contrainte Ps_{13} telle que :

$$Ps_{13} = P_{13} \cap P_{12}.P_{23} \,.$$

Si on reprend l'exemple présenté figure 3.1 en supposant qu'il existe une contrainte directe $P_{13} = \{(x_{11}, x_{31}), (x_{12}, x_{32})\}$, dans ce cas la contrainte restreinte entre les ensembles X_1 et X_2 est donnée par $Ps_{13} = \{(x_{11}, x_{31})\}$ (figure 3.2).



$$\begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix} \wedge \begin{vmatrix} 1 & 0 \\ 1 & 0 \end{vmatrix} = \begin{vmatrix} 1 & 0 \\ 0 & 0 \end{vmatrix}$$

$$P_{13} \text{ et } P_{12}.P_{23} = Ps_{13}$$

Figure 3.2

De même qu'une contrainte binaire d'une variable x_1 vers une variable x_2 a été définie comme un sous-ensemble du produit des domaines de définition des deux variables, $X_1 * X_2$, on définira une contrainte n-aire , σ , comme l'ensemble des n-uplets satisfaisant une propriété donnée :

$$\sigma \subseteq X_1 * X_2 * X_3 *...* X_n \ .$$

Soit σ , une contrainte définie sur les ensembles $X_1 = \{x_{11}, x_{12}, x_{13}, x_{14}\}$, $X_2 = \{x_{21}, x_{22}, x_{23}, x_{24}\}$, $X_3 = \{x_{31}, x_{32}, x_{33}, x_{34}\}$, $X_4 = \{x_{41}, x_{42}, x_{43}, x_{44}\}$ et $X_5 = \{x_{51}, x_{52}, x_{53}, x_{54}\}$ telle que :

$$\sigma = \{(x_{11},x_{21},x_{34},x_{42},x_{53}), (x_{12},x_{24},x_{33},x_{41},x_{51}), (x_{13},x_{21},x_{32},x_{42},x_{51})\}$$

En écriture plus compacte, on notera :

$$\sigma = \begin{vmatrix} 1 & 2 & 3 \\ 1 & 4 & 1 \\ 4 & 3 & 2 \\ 2 & 1 & 2 \\ 3 & 1 & 1 \end{vmatrix} <=> \begin{vmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{24} & x_{21} \\ x_{34} & x_{33} & x_{32} \\ x_{42} & x_{41} & x_{42} \\ x_{53} & x_{51} & x_{51} \end{vmatrix}$$

Dans cette représentation, chaque colonne donne les indices des variables d' un n-uplet admis par la contrainte.

3.1.2) Les réseaux de contraintes

3.1.2.1) Définition générale

Le terme réseau de contraintes est employé pour désigner toute contrainte ou ensemble de contraintes définies sur plus de deux variables (n > 2).

Un réseau de contraintes pourrait donc, à priori, se formaliser de la même façon qu'un ensemble de contraintes n-aires, cependant la quantité d'informations mise en cause par ce formalisme, qui croît exponentiellement avec n, devient très vite inexploitable [MON] :

Si un réseau de contraintes englobe n variables notées x_i , avec $x_i \in X_i = \{x_{i1}, x_{i2},..., x_{iNi}\}$, le nombre maximum de contraintes n-aires définissables sur ce réseau est $2^{N1...Nn}$ contraintes. Chacune des contraintes n-aires sera alors

représentée par un mot de N1×...×Nn bits. Supposons un réseau définissant dix contraintes sur dix variables pouvant prendre chacune cent valeurs, 10^{21} bits seraient alors nécessaires à la représentation de ce réseau.

En conséquence, puisqu'il n'est pas concevable de travailler à ce niveau de représentation (donnant vue sur les n variables en même temps), un réseau de contraintes sera considéré, de façon plus locale, comme un ensemble de contraintes binaires (donnant vue sur les variables prises deux à deux).

3.1.2.2) Réseau binaire

Un réseau R de contraintes binaires est un ensemble d'ensembles $X = \{X_1, X_2, ..., X_n\}$ et de relation P_{ij}, de chacun des ensembles X_i vers tout autre ensemble X_j (i et j variant de 1 à n). Ainsi défini, un réseau binaire peut représenter une contrainte n-aire , σ, telle que :

$$\sigma = \{a \mid a \in X_1 * X_2 * ... * X_n \; ; (\forall i,j) \; S = X_i * X_j \, , a_s \in P_{ij} \} \, .$$

Un n-uplet est admis par la contrainte n-aire σ, si et seulement si, sa projection sur tout sous-espace, de dimension deux, de l'ensemble produit $X_1 * X_2 * ... * X_n$ est un couple admis par une contrainte binaire du réseau.

Soit trois contraintes P_{12}, P_{23} et P_{13} définies sur les ensembles $X_1 = \{x_{11}, x_{12}\}$, $X_2 = \{x_{21}, x_{22}, x_{23}\}$ et $X_3 = \{x_{31}, x_{32}\}$: $P_{12} = \{(x_{11}, x_{21}), (x_{12}, x_{22})\}$, $P_{23} = \{(x_{21}, x_{31}), (x_{21}, x_{32}), (x_{22}, x_{31}), (x_{23}, x_{31})\}$ et $P_{13} = \{(x_{11}, x_{31}), (x_{11}, x_{32}), (x_{12}, x_{32})\}$.
Ce réseau de contraintes binaires définit une relation n-aire σ, telle que $\sigma = \{(x_{11}, x_{21}, x_{31}), (x_{11}, x_{21}, x_{32})\}$ (figure 3.3).

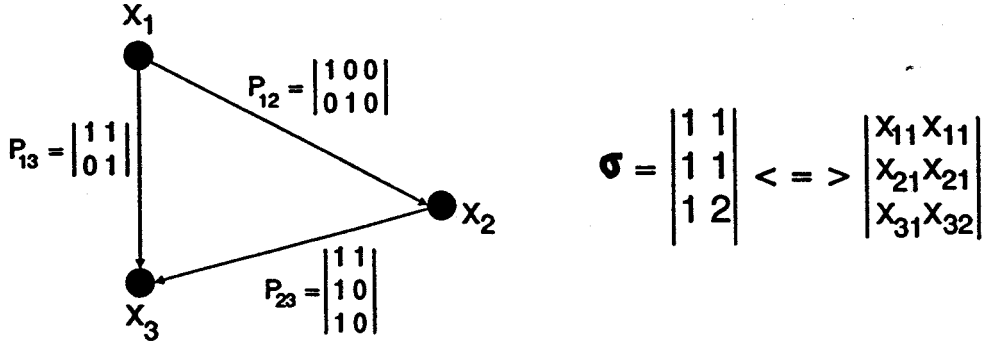


Figure 3.3

Inversement, étant donnée une contrainte n-aire, σ , il est toujours possible, par projections, de générer un réseau de contraintes binaires R' approchant la contrainte σ par excès [MON]. Les contraintes binaires du réseau sont obtenues par projection :

$$P'_{ij} = \{a_S \mid a \in \sigma \text{ et } S = X_i * X_j \}, \qquad (1)$$

Soit la contrainte n-aire σ établie sur les ensembles X_1, X_2 et X_3, définis dans l'exemple précédent : $\sigma = \{(x_{11}, x_{21}, x_{31}), (x_{11}, x_{22}, x_{32}), (x_{12}, x_{22}, x_{31})\}$.
Les contraintes binaires P'_{12} , P'_{23} , P'_{13} obtenues par projection respectives sur les sous-ensembles X_1*X_2, X_2*X_3 et X_1*X_3 sont :

$$P'_{12} = \begin{vmatrix} 1 & 1 \\ 0 & 1 \end{vmatrix} \; ; \; P'_{23} = \begin{vmatrix} 1 & 0 \\ 1 & 1 \end{vmatrix} \; ; \; P'_{13} = \begin{vmatrix} 1 & 1 \\ 1 & 0 \end{vmatrix}$$

Le réseau équivalent R' se construit en énumérant toutes les relations binaires possibles entre les n variables de la contrainte n-aire (à noter que $P'_{ij} = P'_{ji}$, le réseau déduit est symétrique). Ainsi obtenu, il définit une contrainte, σ', plus large que la contrainte d'origine σ ($\sigma \subseteq \sigma'$), et il n'existe pas d'autre réseau binaire R", associable à une contrainte n-aire σ", tel que $\sigma \subseteq \sigma" \subseteq \sigma'$

Dans l'exemple cité ci-dessus, le réseau binaire équivalent définit une contrainte n-aire σ' telle que $\sigma' = \sigma \cup \{(x_{11}, x_{22}, x_{31})\}$.

Remarque: Si la contrainte n-aire, σ , définit directement un ensemble de contraintes binaires sur n variables, alors le réseau binaire R', obtenu par projections, est exactement égal à la contrainte σ, ($\sigma' = \sigma$).

Si les domaines de définition des n variables du réseau sont tous de cardinal égal à N, l'espace mémoire occupé par la représentation approchée d'une contrainte n-aire sous la forme d'un réseau binaire est approximativement de n^2N^2 bits. Il est donc préférable de substituer, à un réseau de contraintes n-aires, le réseau de contraintes binaires équivalent obtenu par projections.

Dans la suite de ce paragraphe, lorsque nous faisons référence à un réseau de contraintes, il s'agit toujours du réseau binaire équivalent obtenu par projection.

3.1.2.3) Equivalence et relation d'ordre

L'opération d'inclusion, définie au paragraphe 1.1.1, permet d'établir une relation d'ordre partiel entre les contraintes :

> Une contrainte P_{ij} est dite plus fine qu'une contrainte P'_{ij} si, et seulement si, toute paire de valeurs solution de P_{ij} est aussi solution de P'_{ij} (on notera $P_{ij} \subseteq P'_{ij}$).

De ce fait, il est possible d'établir un ordre partiel entre des réseaux de contraintes définis sur le même ensemble de variables :

> Soient deux réseaux de contraintes, R et R', définis sur n variables, $v_1 ... v_n$,
>
> $R \subseteq R'$ si et seulement si $\forall i\ (1 \leq i \leq n), \forall j\ (1 \leq j \leq n), P_{ij} \subseteq P'_{ij}$).
>
> On dira que R est plus fin que R'.

Deux réseaux de contraintes définis sur un même ensemble de n variables sont **équivalents** s'ils représentent exactement la même contrainte n-aire, c'est à dire s'ils acceptent tous deux les mêmes n-uplets pour solutions.

3.1.2.4) Représentation graphique d'un réseau

On représente couramment les réseaux de contraintes comme des graphes dirigés et étiquetés (figure 3.4) :

> - chaque noeud du graphe correspond à une variable du réseau, il est associé à un ensemble représentant le domaine de définition de la variable,
>
> - les contraintes sont représentées par des arcs étiquetés et dirigés sur les noeuds (une contrainte unitaire se représentant par une simple boucle). Un arc entre deux noeuds représente en fait l'intersection des contraintes définies sur les variables associées à ces noeuds. Dans un "graphe" de contraintes, toutes les variables qui participent à la même contrainte sont donc adjacentes.

Un réseau de contrainte binaire étant symétrique, à chaque arc d'un noeud i vers un noeud j, symbolisant la contrainte P_{ij}, correspond un arc inverse symbolisant la contrainte identique P_{ji}.

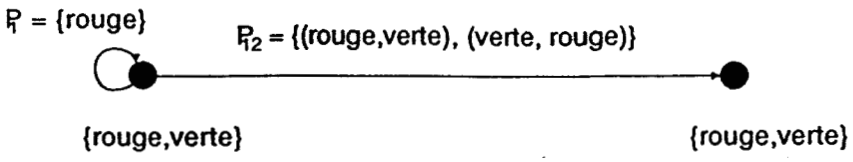


Figure 3.4

3.1.3) Cohérence d'un réseau de contraintes [MAC][DEC][FRE]

Le principe de retour arrière, ou "backtracking", est souvent utilisé par les mécanismes de satisfaction de contraintes (la satisfaction de contrainte est la recherche d'un ou de l'ensemble des n-uplets satisfaisant, de façon simultanée, toutes les contraintes définies sur n variables).

Soit, par exemple, le réseau de contraintes donné par la figure 3.5, défini sur les variables x_1, x_2 et x_3 de domaines respectifs $X_1 = \{a, b\}$, $X_2 = \{b, c, d, e\}$ et $X_3 = \{e, f\}$.

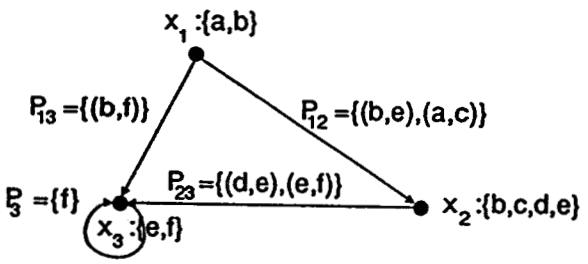


Figure 3.5

Si une procédure de satisfaction des contraintes basée sur le retour arrière instancie les variables du réseau dans l'ordre x_1, x_2 et x_3, elle affecte à x_1 la première valeur de X_1. Aucune contrainte unitaire n'étant établie sur la variable x_1, la première valeur de X_2 est affectée à la variable x_2. Si le couple de valeurs obtenues pour x_1 et x_2 ne vérifie pas la contrainte P_{12}, la dernière instanciation est remise en cause : la deuxième valeur de X_2 est affectée à la variable x_2 puis on regarde de nouveau si le couple de valeurs obtenues pour x_1 et x_2 vérifie la contrainte P_{12} (quand toutes les valeurs d'un domaine ont été prises en compte, on remet en cause la dernière instanciation de la variable précédente). Si le couple de valeurs obtenues pour x_1 et x_2 vérifie la contrainte, la première valeur de X_3 est affectée à la variable x_3 puis on

regarde si le couple de valeurs obtenues pour x_2 et x_3 vérifie la contrainte P_{23}. Si la contrainte n'est pas vérifiée, la dernière instanciation est remise en cause : la deuxième valeur de X_3 est affectée à la variable x_3 puis on regarde de nouveau si le couple de valeurs obtenues pour x_2 et x_3 vérifie la contrainte P_{23}. Si oui, on regarde si la valeur affectée à x_3 vérifie également la contrainte unitaire P_3 puis la contrainte P_{13}. Si oui, le triplet de valeurs obtenues pour x_1, x_2 et x_3 est solution du réseau de contraintes. Si non, on remet en cause la dernière instanciation...

La séquence totale d'instanciation des variables du réseau présenté figure 3.5, par cette procédure de satisfaction des contraintes, est représentée figure 3.6.

ligne 1	a	
ligne 2	ab	$(a, b) \notin P_{12}$
ligne 3	ac	la dernière instanciation de x_2 est remise en cause
ligne 4	ace	$(c, e) \notin P_{23}$
ligne 5	acf	$(c, f) \notin P_{23}$
ligne 6	ae	toutes les valeurs de X_3 ont été testées, la dernière instanciation de x_2 est remise en cause
ligne 7	ad	$(a, d) \notin P_{12}$
ligne 8	ae	$(a, e) \notin P_{12}$
ligne 9	a	toutes les valeurs de X_2 ont été testées, la dernière instanciation de x_1 est remise en cause
ligne 10	b	
ligne 11	bb	$(b, b) \notin P_{12}$
ligne 12	be	$(b, c) \notin P_{12}$
ligne 13	bd	$(b, d) \notin P_{12}$
ligne 14	be	
ligne 15	bee	$(e, e) \notin P_{23}$
ligne 16	bef	solution du réseau de contraintes
ligne 17	be	toutes les valeurs de X_3 ont été testées, la dernière instanciation de x_2 est remise en cause
ligne 18	b	toutes les valeurs de X_2 ont été testées, la dernière instanciation de x_1 est remise en cause.

Figure 3.6

Cette technique de recherche d'une solution par retour arrière, appliquée à la résolution des contraintes, présente certains défauts qui, s'ils ne nuisent pas à l'efficacité de l'algorithme de satisfaction, en diminuent la rapidité d'exécution :

- Soit, dans un réseau de contraintes, la contrainte unitaire P_i définie sur une variable x_i de domaine de définition X_i. S'il existe une valeur de X_i ne vérifiant pas P_i, lors de la recherche d'une solution globale pour le réseau, cette valeur entraînera des instanciations répétées aboutissant toutes à l'échec. En éliminant, à priori, toutes les valeurs de X_i ne vérifiant pas la contrainte unitaire P_i définie sur x_i, on évite lors de la satisfaction des contraintes cet aspect infructueux de la recherche.

Pour le réseau de contraintes présenté figure 3.5, la valeur *e* peut être enlevée de X_3 puisqu'elle ne vérifie pas la contrainte unitaire P_3. De ce fait, les lignes 4 et 5 de la séquence d'instanciation (figure 3.6) n'existeront plus.

- Soit une contrainte P_{ij} (j > i) établie sur deux variables x_i et x_j d'un réseau de contraintes défini sur n variables. Les variables sont instanciées dans l'ordre $x_1,x_2,...,x_i,...,x_j,...,x_n$. Si x_i a été instanciée à une valeur *a* de X_i et qu'aucune valeur de X_j ne permet d'obtenir un couple de valeurs solution de P_{ij}, alors au moment d'instancier x_j l'algorithme de recherche essayera toutes les valeurs de X_j, échouera, affectera une nouvelle valeur à x_{j-1}, réessayera toutes les valeurs de X_j, échouera de nouveau...pour conclure finalement à la non validité de la valeur *a* affectée à xi. Comme précédemment, il serait judicieux d'éliminer la valeur *a* du domaine X_i avant de chercher une quelconque solution répondant à l'ensemble des contraintes.

Dans le réseau de contraintes figure 3.5, on considère la contrainte P_{13}. Si la valeur *a* est affectée à x_1, aucune valeur dans X_3 ne permet de former avec *a* un couple de valeurs solution de la contrainte P_{13}. Si *a* est enlevée de X_1, les lignes 1 à 9 ne figureront plus dans la séquence totale des instanciations présentée figure 3.6.

- Supposons maintenant qu'il existe une valeur *a* de X_i et une valeur *b* de X_j, respectivement pour les variables x_i et x_j, telles que $(a) \in P_i$, $(b) \in P_j$, $(a,b) \in P_{ij}$, mais qu'il n'existe aucune valeur *c* de X_k, attribuable à x_k, telle que $(c) \in P_k$, $(a,c) \in P_{ik}$ et $(c,b) \in P_{kj}$ soient simultanément satisfaites. De ce fait, la paire de valeurs (a, b) ne pourra se retrouver dans aucun des n-uplets solutions du réseau de contraintes (satisfaisant simultanément toutes les contraintes du réseau). Ces deux valeurs peuvent donc être retirées de leur domaine de définition respectif.

Puisqu'il est possible, nous l'avons vu précédemment, d'établir différentes représentations d'un même ensemble de contraintes, on peut espérer, partant d'une représentation initiale d'un réseau de contraintes, trouver une représentation équivalente plus adaptée à la méthode du "backtracking". Si un n-uplet est solution d'un réseau, il vérifie globalement l'ensemble des contraintes définies sur le réseau, il vérifie aussi localement chacune d'entre elles. Inversement, si un l-uplet (l < n) ne vérifie pas localement une partie des contraintes du réseau, il ne pourra participer à aucun n-uplet solution de ce réseau.

Résoudre localement les contraintes d'un réseau, permettrait donc de définir un réseau équivalent, débarrassé de toute valeur ne participant pas à une solution globale : on appelle cette démarche, la vérification de cohérence d'un réseau de contraintes.

3.1.3.1) Cohérence totale d'un réseau de contraintes

Un réseau de contraintes est cohérent s'il est cohérent par rapport à chacun de ses noeuds, chacun de ses arcs et chacun des chemins définis par ses arcs.

- La cohérence/noeuds garantit que pour chaque noeud du réseau, toutes les valeurs de l'ensemble de définition associé sont solutions de la contrainte unitaire définie sur le noeud.

- La cohérence/arcs garantit que pour chaque noeud, origine d'un arc, toute valeur de l'ensemble de définition associé est élément d'au moins une paire, solution de la contrainte binaire directe partant du noeud.

- La cohérence/chemins garantit que toute paire, solution d'une contrainte directe P_{ij}, est élément d'au moins une solution pour toute contrainte induite du noeud i au noeud j.

3.1.3.2) Cohérence par rapport à un noeud

Un noeud est cohérent si et seulement si chaque valeur de son ensemble de définition vérifie la contrainte unitaire définie sur la variable associée à ce noeud :

$$\forall x \in X_i, x \in P_i.$$

Si on définit la fonction booléenne $P_i(x)$ par :

$P_i(x) : X_i \rightarrow \{vrai, faux\}$,
$P_i(x) = vrai$ ssi $x \in P_i$ et $P_i(x) = faux$ si $x \notin P_i$,

un noeud est cohérent si :

$\forall x \in X_i, P_i(x) = vrai.$

La cohérence d'un réseau par rapport à ses noeuds ne met en cause que les contraintes unitaires, aucune interaction entre les noeuds n'est à prendre en compte. Une simple procédure, associée à une boucle, permet donc de vérifier ce type de cohérence [MAC] :

- procédure de vérification de cohérence d'un noeud, i , associé à un ensemble de définition X_i :

procédure NC(i);
$X_i \leftarrow X_i \cap \{ x \mid P_i(x) = \text{vrai} \}$;

- boucle de vérification de cohérence/noeud d'un réseau de n noeuds :

for (i = 1) to (i = n) do NC(i);

Soit le réseau de contraintes présenté figure 3.7. La procédure de vérification de cohérence/noeud lancée sur ce réseau, éliminera la valeur *verte* du domaine de définition du noeud soumis à la contrainte unitaire P_1.

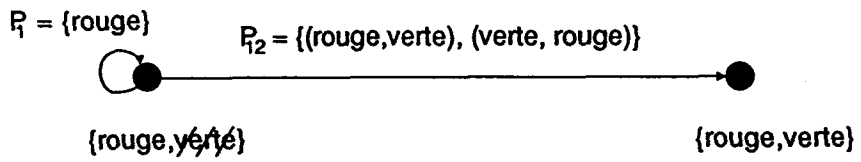


Figure 3.7

3.1.3.3) Cohérence par rapport à un arc

Un arc (i,j) est cohérent si et seulement si pour toute valeur de l'ensemble de définition du noeud i, vérifiant la contrainte unitaire sur x_i, il existe une valeur de l'ensemble de définition du noeud j, vérifiant la contrainte unitaire sur x_j et la contrainte binaire P_{ij}.

$$\forall x \in X_i / (x) \in P_i , \exists y \in X_j , (y) \in P_j \text{ et } (x,y) \in P_{ij} .$$

Si on définit la fonction booléenne $P_{ij}(x,y)$ par :

$P_{ij}(x,y) : X_i * Xj \rightarrow \{vrai, faux\}$,
$P_{ij}(x,y) = vrai$ ssi $(x,y) \in P_{ij}$ et $P_{ij}(x,y) = faux$ si $(x,y) \notin P_{ij}$,

un arc est cohérent si :

$$\forall x \in X_i / P_i(x) = vrai, \ \exists y \in X_j, P_j(y) = vrai \text{ et } P_{ij}(x,y) = vrai .$$

Afin d'établir ce type de cohérence, on se basera sur l'observation suivante [FIK]: étant donnés deux domaines discrets X_i et X_j, associés à deux variables cohérentes x_i et x_j, si pour une valeur x de X_i, il n'existe aucune valeur y de X_j vérifiant la contrainte binaire P_{ij} ($P_{ij}(x,y)$ = vrai), alors la valeur x peut être enlevée de X_i. Lorsque cela est fait pour toute valeur x de X_i, l'arc (i,j) est cohérent.

De cette observation découle la procédure de vérification de cohérence d'un arc [MAC] :

```
- procédure AC(i,j);
  begin
  Delete ← false;
  for each x ∈ Xi do
  if there is no (y ∈ Xj) such that Pij(x,y) then
        begin
        delete x from Xi;
        Delete ← true;
        end;
  return Delete;
  end
```

Sur le réseau présenté figure 3.8, la procédure de vérification de cohérence/arc enlève la valeur *rouge* du domaine de définition X_2.

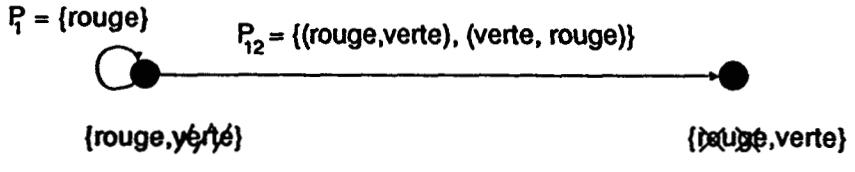


Figure 3.8

Toutefois, appliquer la procédure AC(_) à un arc (i,j) du réseau ne le rend pas définitivement cohérent. En effet, si un autre arc (j,k) du réseau est rendu ultérieurement cohérent par la même procédure, le domaine X_j s'en trouvant modifié, il est possible que l'arc (i,j), précédemment traité, ne soit plus cohérent pour ce nouvel ensemble de définition.
Il n'est donc pas possible de rendre un réseau cohérent par rapport à ses arcs en exécutant, une seule fois pour chacun d'eux, la procédure de vérification de cohérence d'un arc.

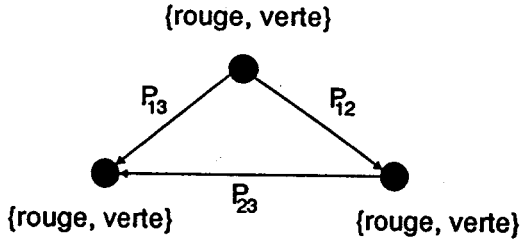
Deux stratégies sont alors envisageables :

- La première solution consiste à appliquer la procédure AC(_) à chacun des arcs du réseau et recommencer cette opération jusqu'à ce qu'elle n'entraîne plus aucune modification des ensembles de définition du réseau.
Cette stratégie, bien qu'étant la plus simple à mettre en oeuvre , est loin d'être optimale, puisqu'à chaque modification de l'ensemble de définition d'un noeud, elle appliquera la procédure AC(_) à tous les arcs du réseau qu'ils soient ou non affectés par ce changement.

- La seconde approche consiste à ranger dans une file d'attente tous les arcs du réseau et à exécuter séquentiellement la procédure AC(_) pour chaque élément retiré de la file jusqu'à épuisement de celle-ci. Si un arc (i,j) est rendu cohérent (si AC(i,j) renvoie '*true*'), seuls les arcs arrivants sur le noeud, i , peuvent en être affectés. Ils seront donc ajoutés à la file d'attente, s'ils n'y sont pas déjà, afin que leur cohérence soit de nouveau vérifiée.
Cette approche est la plus fréquemment citée dans la littérature [MON][MAC][WAL]. A titre d'exemple, nous pouvons citer l'algorithme de vérification de cohérence/arcs donné par [MAC] :

```
...
begin
for (i = 1) to (i = n) do NC(i);
Q ← {(i,j) | (i,j) ∈ arcs(ℜ), i≠j};
while Q not empty do
   begin
   select and delete any arc(k,m) from Q;
   if AC(k,m) then Q ← Q ∪ {(i,k) | (i,k) ∈ arcs(ℜ), i≠k , i≠m};
   end;
end
```

Remarque: Un réseau peut être cohérent par rapport à ses arcs sans pour autant avoir de solution. Prenons l'exemple d'un trièdre dont chaque face doit être de couleur différente. On dispose pour peindre chacune des faces de l'ensemble des couleurs {rouge, vert, bleu}. Le réseau de contraintes obtenu est présenté figure 3.9. Ce réseau est cohérent par rapport à ces arcs mais ne possède aucune solution.



$P_{12} = P_{23} = P_{13} = \{(rouge,verte),(verte,rouge)\}$

Figure 3.9

3.1.3.4) Cohérence par rapport à un chemin

Un chemin de longueur m le long des noeuds $(i_0,i_1,...,i_m)$ est cohérent si et seulement si pour toutes valeurs x et y, appartenant respectivement aux ensembles de définition des noeuds i_0 et i_m, telles que $(x) \in P_{i0}$, $(y) \in P_{im}$ et $(x,y) \in P_{i0im}$, il existe une séquence de valeurs $(z_1,...,z_{m-1})$ correspondant à chaque noeud du chemin telle que :

- $(z_1) \in P_{i1}$,..., $(z_{m-1}) \in P_{im-1}$,

- $(x,z_1) \in P_{i0i1}$, $(z_1,z_2) \in P_{i1i2}$,..., $(z_{m-1},y) \in P_{i(m-1)im}$.

En d'autres termes, la paire (x,y) est permise par un chemin de i_0 à i_m si, à chaque noeud intermédiaire, des valeurs peuvent être trouvées qui satisfassent les contraintes unitaires et binaires le long du chemin.

Cas le plus simple d'un chemin de longueur 2:

Un chemin de longueur 2, d'un noeud i vers un noeud j en passant par le noeud k , est cohérent si et seulement si :

$$\forall (x,y) \in X_i * X_j \mid (P_i(x) = \text{vrai et } Pij(x,y) = \text{vrai et } P_j(y) = \text{vrai })$$

$$\exists z \in X_k \mid (P_{ik}(x,z) = \text{vrai et } P_k(z) = \text{vrai et } P_{kj}(z,y) = \text{vrai }) .$$

En respectant la notation employée jusqu'ici, un chemin de longueur 2 d'un noeud i vers un noeud j, passant par le noeud k, est cohérent si et seulement si toute paire permise par la contrainte directe P_{ij} est aussi permise par la contrainte induite $P_{ik}.P_{kj}$. La relation P_{ij} doit donc vérifier l'égalité suivante :

$$P_{ij} = (P_{ij} \cap P_{ik}.P_{kk}.P_{kj}) .$$

Partant de cette propriété, la procédure de vérification de cohérence d'un chemin de longueur 2 peut être définie comme suit [MAC] :

```
- procédure PC(i,k,j);
begin
Z ← Pij ∩ Pik.Pkk.kPkj;        (* s'il n'existe pas de contrainte directe entre les noeuds
if Z = Pij                      vi et vj, alors Pij = Uij *)
   then return false
   else
      begin
      Pij ← Z;
      return true;
      end;
end
```

Un théorème, établit par U. Montanari [MON], montre qu'un réseau de contraintes est cohérent par rapport à ses chemins, si tout chemin de longueur 2 de ce réseau est cohérent.

Nous donnerons pour exemple un algorithme, proposé par [MAC], permettant d'établir la cohérence d'un réseau par rapport à ses chemins. Dans cet algorithme, chaque fois qu'un chemin de longueur 2 est rendu cohérent, la procédure de cohérence/chemin est relancée sur les chemins affectés par le changement intervenu :

- procédure RP(i,k,j);

(cet procédure renvoie l'ensemble des chemins de longueur 2 affectés par un changement sur le chemin (i→k→j) *)*

if i < j
  then
    return Sa = {(i,j,m) | (i≤m≤n) , (m≠j)} ∪ {(m,i,j) | (1≤m≤j) , (m≠i)}
                ∪ {(j,i,m) | (j<m≤n)} ∪ {(m,j,i) | (1≤m<i)};
      (cas où i≠j : Sa regroupe tous les chemins de longueur 2 incluant les arcs (i,j) ou (j,i) *)*
  else
    return Sb = {(p,i,m) |(1≤p≤m) , (1≤m≤n) , ⌉(p=i=m) , ⌉(p=m=k)};
      (cas où i=j : Sb regroupe tous les chemins de longueur 2 passant par le noeud i)*

- le corps du programme principal étant :

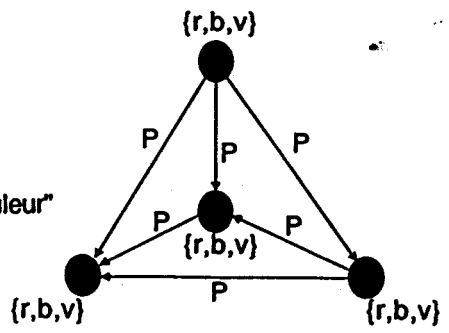
...
begin
    Q ← {(i,k,j) | (i≤j) , ⌉(i=k=j)};
    while (Q not empty) do
      begin
      select and delete a path (i,k,j) from Q;
      if PC((i,k,j)) then Q ← Q ∪ RP((i,k,j));
      end;
end

3.1.3.5) Cohérence d'un réseau de contraintes et satisfaction de contraintes

Bien qu'elle s'apparente à une technique de propagation de contraintes, dans bon nombre de cas, la vérification de cohérence d'un réseau ne permet pas à elle seule de résoudre les problèmes de satisfaction de contraintes (CSP). En effet, elle élimine, de façon définitive, toutes les valeurs localement incohérentes, qui ne peuvent participer à une solution finale, mais n'offre que rarement une solution à la contrainte globale.

Soit une pyramide dont on veut colorer chacune des faces d'une couleur appartenant à l'ensemble {rouge, vert, bleu}, de telle façon que deux faces adjacentes n'aient pas la même couleur. Le réseau de contraintes obtenu est présenté figure 3.10.

P : "n'a pas la même couleur"



Dans ce cas précis, le problème posé n'a pas de solution (solution globale = {}). Utilisée seule, la vérification de cohérence du réseau de contraintes ne suffit pas à établir cette réponse .

Figure 3.10

Les techniques de vérification de cohérence ne sont donc pas des techniques de satisfaction de contraintes mais plutôt des techniques de pré-calcul permettant de faciliter la recherche d'une ou des solutions globales au réseau de contraintes.

3.2. LA CONTRAINTE CONSTRUCTIVE

Dans le paragraphe précédent, les contraintes sont exprimées de façon non-constructive. Une contrainte non-constructive est une contrainte dont on peut seulement tester la validité (Pij(xi,xj) est vrai ssi (xi,xj) ∈ Pij) [GUS].

De nombreuses contraintes peuvent cependant s'exprimer de façon constructive . Dans ce cas, la contrainte est définie par la donnée d'un prédicat et d'un ensemble d'expressions fonctionnelles représentant ses variables.

Les contraintes constructives peuvent à la fois être testées puis validées.

3.2.1) Expression de la contrainte

Une contrainte constructive se définit par

- un prédicat, utilisé pour tester la contrainte et savoir si elle est ou n'est pas satisfaite ;

- un ensemble d'expressions ou de méthodes : chaque expression représente un moyen de résoudre la contrainte. **Lorsque l'une de ces expressions est appliquée, la contrainte est immédiatement résolue.**

Par exemple, la contrainte $a = b + c$ est décrite par son prédicat ('$a = b + c$') et par l'ensemble des trois expressions :

affecter à a la valeur (b + c),
affecter à b la valeur (a - c),
affecter à c la valeur (a - b).

Dans un langage procédural, ces contraintes peuvent être écrites à l'aide d'une primitive qui permet de définir un modèle de contrainte par la donnée d'un prédicat et d'un ensemble de méthodes. Par exemple, pour définir la contrainte même-hauteur qui permet de maintenir l'égalité entre les hauteurs de deux objets donnés de type Meuble, on pourrait écrire [BER]:

defcontrainte même-hauteur
soit m1 un meuble
m2 un meuble
prédicat hauteur de m1 = hauteur de m2
méthodes [hauteur de m1] ← hauteur de m2
[hauteur de m2] ← hauteur de m1
fin-contrainte

En programmation orientée objet, la contrainte peut se définir comme une relation, à l'instar de la relation de dépendance, ou comme un objet, auquel cas établir une contrainte revient à définir une association entre deux types d'objets, les objets contraints et les contraintes.

La relation de dépendance, étudiée au chapitre 1, peut être vue comme une considérée comme étant une contrainte simple, unidirectionnelle, d'un objet maître vers un ensemble d'objets dépendants. Dans ce cas précis, le prédicat de la contrainte est défini implicitement par la dépendance et les divers moyens de résoudre la contrainte sont donnés par les méthodes de mise à jour définies pour chacun des objets dépendants (*'update: anAspectSymbol'*).

Si on s'en tient à une telle structure de définition, il sera difficile pour l'utilisateur de décrire une contrainte de façon déclarative, puisque les informations concernant la contrainte sont dispersées sur l'ensemble du réseau constitué par l'objet maître et ses dépendants. D'autre pas si la contrainte n'est pas unidirectionnelle mais multidirectionnelle, comme la contrainte 'a = b + c' donnée précédemment, il devient vite impossible de gérer le réseau des dépendances tant sa structure est lourde et complexe.

Définir la contrainte comme un objet nous a permis au contraire de regrouper, dans une même structure, l'ensemble des informations relatives à la contrainte.

Le prédicat et l'ensemble des méthodes de résolution décrivent la structure même de la contrainte alors que les méthodes permettant de définir ou de modifier la contrainte, de vérifier son prédicat ou d'appliquer l'un de ses méthodes de résolution, donnent une description fonctionnelle de cet objet.

Pour décrire la structure et le comportement général communs à toutes contraintes, nous avons créé la classe CONTRAINTE qui possède, entre autres, les variables d'instance **règle** et **méthodes** faisant référence, pour la première au prédicat de la contrainte, pour la seconde à une collection de méthodes de résolution. Les méthodes d'instance écrites dans la classe CONTRAINTE permettent de manipuler l'objet contrainte

dans sa globalité, par exemple pour des opérations de comparaison ou de modification, ou de ne manipuler qu'une de ses composantes, par exemple pour vérifier le prédicat donné par la variable d'instance **règle** ou appliquer une des méthodes de résolutions définies dans la variable d'instance **méthodes.**

3.2.2) Comment établir les méthodes de résolution d'une contrainte ?

Puisque notre but est de simplifier au maximum la tâche de l'utilisateur, en lui permettant d'intégrer dans son application des objets contraints, l'idéal serait qu'il n'ait à définir que le prédicat de la contrainte, le mécanisme de résolution des contraintes se chargeant du reste. Dans ce cas, il serait alors possible de parler de contraintes purement déclaratives.

En pratique, il est très difficile de demander au mécanisme interne de satisfaction des contraintes d'établir, de lui-même, les différentes méthodes de résolution de chacune des contraintes définies par l'utilisateur. Nous avons vu au chapitre 2 qu'il existe une très grande variété de contraintes. Pour qu'un système interne puisse, à partir d'un prédicat quelconque, en déduire les différentes méthodes de résolution de la contrainte, il faudrait qu'il intègre des mécanismes complexes de résolution d'équations. Pour la contrainte 'a = b + c' citée précédemment, les méthodes de résolution se déduisent facilement, mais si on considère une contrainte 'a= b2 + 3*b + c', il faudrait au système interne un solveur d'équation pour déduire la méthode de résolution permettant d'affecter b lorsque les variables a ou c sont modifiées. De même si une contrainte 'x = sin (y)' était définie, le système devrait pouvoir inverser l'équation mathématique donnée par le prédicat de la contrainte pour en déduire la valeur de y en fonction de x.

Plusieurs solutions peuvent être adoptées :

1°) Si le domaine d'application est précis et que l'on se cantonne à ce domaine (par exemple la modélisation des circuits électriques ou la représentation de figures géométriques), alors la diversité des contraintes est réduite et il est possible d'intégrer au mécanisme interne de résolution des solveurs adaptés précisément aux contraintes employées.

2°) Si le domaine d'application n'est pas précisé, la solution la plus simple est de charger l'utilisateur de définir non seulement le prédicat de la contrainte mais aussi ses méthodes de résolution [BOR]. Cette solution a au moins un avantage, c'est que

l'utilisateur ne risque pas d'être surpris par l'application d'une méthode de résolution à laquelle il ne s'attend pas.

3°) En programmation orientée objet, il est possible de construire une bibliothèque de classes pour décrire les différents types de contraintes. On aura par exemple les classes CONTRAINTESEGALITAIRES, CONTRAINTESINEGALITAIRES, CONTRAINTESTEMPORELLES ... Dans chacune de ces classes, il est possible d'écrire des procédures de recherche des méthodes de résolution de la contrainte, ce qui revient à associer un solveur à chaque type de contraintes.

Notre objectif étant d'introduire les contraintes dans un outil de programmation orientée objet pour pouvoir utiliser les objets contraints dans toutes applications, nous avons retenu les deux dernières solutions proposées. L'utilisateur aura ainsi à sa disposition une hiérarchie de classes lui permettant de décrire certaines contraintes de façon purement déclarative. Pour des contraintes spécifiques à son application, il pourra utiliser la classe CONTRAINTE et définir à la fois le prédicat et les méthodes de résolution de ces contraintes quelles qu'elles soient. Libre à lui, s'il emploie fréquemment les mêmes types de contraintes spécifiques, d'enrichir la bibliothèque des classes de contraintes en créant ses propres classes.

3.2.3) Les priorités

3.2.3.1) Priorité définie sur la contrainte

L'utilisateur ayant la possibilité d'établir des contraintes à tous niveaux de représentation de son application, il est évident que certaines de ces contraintes seront plus impératives que d'autres. Pour faciliter la résolution de toutes les contraintes définies sur un système, il est possible d'établir un ordre de priorité sur les contraintes [BOR]. Plus la priorité est élevée, plus la contrainte est impérative et doit absolument être satisfaite.

Si les contraintes sont hiérarchisées selon un ordre de priorité, le mécanisme de résolution interne cherchera dans un premier temps à résoudre les contraintes de priorité la plus élevée, puis les contraintes de priorité juste inférieure et ainsi de suite jusqu'à ce qu'il ait scruté toutes les contraintes. Si lors de la résolution il y a conflit entre contraintes de priorités différentes (si les deux contraintes ne peuvent être simultanément résolues), le mécanisme choisira de résoudre la contrainte de priorité la

plus élevée au détriment de l'autre contrainte. Cette démarche ne pose aucun problème si la contrainte négligée est de très faible priorité. Mais si elle est elle-même d'une priorité élevée, l'ensemble risque d'aboutir à des résultats aberrants, ne répondant plus aux exigences formulées par l'utilisateur.

L'utilisation de priorités définies sur les contraintes est à double tranchant :

- d'une part, elle permet de simplifier le processus de résolution en laissant au mécanisme interne la possibilité d'abandonner certaines contraintes en cas de conflits. Ce qui permet d'avoir un système plus autonome (faisant moins souvent appel à l'utilisateur en cas de conflit) ;

- d'autre part, plus les degrés de priorité définies sur un réseau de contraintes sont nombreux, plus la quantité de contraintes négligées risque d'être importante. Dans ce cas, le système fait moins souvent appel à l'utilisateur mais le résultat obtenu est rarement celui escompté. C'est pourquoi nous avons choisi de ne pas retenir ce principe de priorités définies sur les contraintes. Si l'utilisateur décide d'en faire usage, il lui suffira d'ajouter une variable d'instance **priorité** à la classe CONTRAINTE et de modifier le processus de satisfaction des contraintes pour qu'il en tienne compte.

3.2.3.2) Priorité définie sur les méthodes de résolution

Le problème se pose souvent lorsque l'on doit résoudre une contrainte de savoir quelle méthode employée. Par exemple pour la contrainte 'a = b + c', si on modifie la valeur de a, comment rétablir la contrainte ? Doit-on recalculer la valeur de la variable b ou celle de la variable c ?

Ce choix est difficile mais peut être guidé par différents facteurs :

1°) Un ordre de priorité peut être donné sur les différentes méthodes de résolution d'une contrainte. Cette priorité respecte toutefois un principe fondamental, 'on ne modifie pas de nouveau, par l'application d'une méthode de résolution, une variable qui vient d'être modifiée par le système ou par l'utilisateur'.

Soit la contrainte 'point_milieu = 1/2(extrémité1 + extrémité2) établie sur une ligne ayant un point milieu dont les méthodes de résolution sont :

point_milieu ← 1/2(extrémité1 + extrémité2),
extrémité1 ← (2*point_milieu - extrémité2),
extrémité2 ← (2*point_milieu - extrémité1).

Si on modifie la position de la première extrémité, on a alors le choix entre bouger la seconde extrémité ou bouger le point milieu. La deuxième solution étant préférable à la première, si une priorité plus forte était donnée à la méthode de résolution [point_milieu ← 1/2(extrémité1 + extrémité2)], par rapport à la méthode [extrémité2 ← (2*point_milieu - extrémité1)], le mécanisme de résolution des contraintes optera pour cette méthode plutôt que pour l'autre.

Cet ordre de priorité peut être donné implicitement par la sémantique de la contrainte. Par exemple, les méthodes de résolution de la contrainte sont définies par ordre de préférence, la première méthode définie étant celle de priorité la plus élevée. Pour résoudre une contrainte, on appliquera la première méthode, dans l'ensemble des méthodes de résolution, qui ne remet pas en cause la dernière modification survenue. Dans notre système, nous avons retenu cette première solution car elle est applicable quelle que soit l'application.

2°) Il est possible de mémoriser, pour chaque variable, sa date de dernière modification. On choisit alors en priorité les méthodes de résolution affectant les variables les plus anciennement modifiées ou inversement les variables les récemment affectées. Cette solution ne peut toutefois pas être retenue dans le cas de certaines applications, comme par exemple la modélisation des circuits électriques.

3°) La dernière solution consiste à recourir à l'arbitrage de l'utilisateur [STEE] pour le choix de la variable à modifier. Cette démarche ne doit cependant être adoptée que pour des cas exceptionnels de conflits sinon le système devient rapidement insupportable.

3.2.4 Opérations sur les contraintes constructives

Dans la première partie consacrée aux contraintes non-constructives, nous avons vu qu'il est possible de réaliser sur ce type de contraintes différentes opérations telles que l'inversion, la négation, l'union ou la composition.

Il est difficile de définir de telles opérations sur les contraintes constructives. Comment établir l'inverse ou la négation d'un prédicat ? Comment définir l'intersection, l'union ou la composition de contraintes constructives ?

De façon générale, on pourra cependant convenir des lois suivantes :

- La négation d'une contrainte constructive est donnée par la négation de son prédicat. Pour que la contrainte non(P_{12}) soit satisfaite, il faudra que le test de vérification du prédicat de la contrainte P_{12} soit négatif.

- L'union de deux contraintes constructives donne une contrainte multiple composée de deux prédicats (respectivement, le prédicat de chacune des contraintes réunies). Cette contrainte est satisfaite si au moins un de ses prédicats est vérifié.

- L'intersection de deux contraintes constructives donne également une contrainte multiple composée de deux prédicats qui ne sera satisfaite que si ses deux prédicats sont simultanément vérifiés.

Pour la composition de contraintes, il n'est pas possible d'établir une loi précise mais, de façon assez élémentaire, il est souvent possible de ramener la composition de contraintes à l'intersection de contraintes.

3.2.5) Cohérence d'un réseau de contraintes constructives

Il n'est pas possible de vérifier la cohérence d'un réseau de contraintes constructives, comme il est possible de le faire pour un réseau de contraintes non-constructives (cohérence/$_{noeuds,arcs,chemins}$), puisqu'une contrainte constructive s'exprime sur un ensemble de variables et non pas par un ensemble de données.

$P_{constructive}$: 'a = b + c'
$P_{non\text{-}constructive}$: {(1,1,2), (1,2,3), (2,2,4}

La seule façon de vérifier la cohérence d'un réseau de contraintes constructives est d'essayer de le satisfaire pour voir s'il admet une solution. Pourtant on peut retenir certains enseignements des méthodes de vérification de cohérence proposées au paragraphe 3.1.3 de ce chapitre :

- Tout d'abord, elles introduisent la notion de propagation de contraintes. Quand on vérifie la cohérence d'un réseau par rapport à un arc donné (i,j), on répercute la vérification de cohérence sur tous les arcs possédant le noeud i ou le noeud j comme extrémité. On procède de même pour la vérification de cohérence par rapport aux chemins.

- Il est possible d'avoir une vue locale du réseau de contraintes. Les procédures de vérification de cohérence présentées analysent le réseau de contraintes en ne tenant en compte, à un instant donné, que de trois variables sur l'ensemble du réseau (trois étant un maximum, atteint pour la vérification de cohérence sur un chemin de longueur 2). Le réseau est cependant considéré dans sa globalité grâce à la relance de la vérification de cohérence sur tout arc (ou sur tout chemin) affecté par un changement.

On retrouvera ces différents points (propagation et vue locale d'un réseau) dans la plupart des méthodes de satisfaction des contraintes constructives, présentées au chapitre .

3.2.6 Implantation en Smalltalk-80

3.2.6.1 Définition de la contrainte

La contrainte est définie comme un objet, ainsi elle peut être manipulée facilement par l'utilisateur et par le mécanisme interne de satisfaction des contraintes.

L'utilisateur crée la contrainte, la modifie ou la détruit. Le mécanisme interne lance la procédure de satisfaction des contraintes sur le réseau d'objets contraints. Cette procédure scrute l'ensemble des contraintes du réseau, demandant à chacune d'elle de vérifier son prédicat et éventuellement d'appliquer l'une de ses méthodes de résolution.

3.2.6.2 La classe REGLE

Cette classe permet de définir par une expression quelconque le prédicat d'une contrainte. Elle possède deux variables d'instance, **calcul** qui référence le prédicat et **model** qui référence l'objet contraint.

Pour créer une instance de la classe REGLE, on envoie le message *Regle new: aString*, où l'argument *aString* représente l'expression du prédicat.

Par exemple, pour définir le prédicat de la contrainte *'a = b + c*2'*, on emploiera tout simplement le message :

*Regle new: 'a = b + c*2'.*

Une méthode d'instance qui permet de tester l'exactitude d'un prédicat a été définie dans le protocole 'vérification' de la classe REGLE. Elle évalue l'expression référencée par la variable d'instance **calcul**, appliquée aux valeurs particulières de l'objet contraint référencé par la variable d'instance **model**. Cette méthode s'écrit :

```
vérifiée
    | t terme |
    t <- ReadStream on: calcul.
    terme <- Compiler new
                evaluate: t
                in: nil
                to: model
                notifying: nil
                ifFail: nil.
    terme ifTrue: [^true].
    ^false
```

La classe REGLE permet donc de définir n'importe qu'elle type de contraintes puisque l'expression donnée en prédicat n'est en fait qu'une ou plusieurs méthodes à évaluer. A partir de cette classe, il est possible de définir des sous-classes pour certains types de prédicats en particulier. On définit par exemple les classes EGALITE, INEGALITE, FONCTIONTEMPORELLE pour lesquelles il sera alors possible de déduire automatiquement les méthodes de résolution correspondantes.

3.2.6.3 La classe METHODE

La classe METHODE permet de définir les méthodes de résolution de la contrainte. Elle possède pour variables d'instance :

- **résultat** qui fait référence à l'objet contraint, ou plus exactement, à la variable d'instance de l'objet contraint qui doit être instanciée par la méthode ;
- **calcul** qui donne l'expression à évaluer pour obtenir la valeur à assigner au résultat ;
- **model** qui référence l'objet contraint ;
- **méthodeCompilée** qui pointe sur une instance de la classe COMPILEDMETHOD.

Pour créer une instance de METHODE, on envoie le message :

Méthode result: aString
calcul: aString.

Par exemple, pour créer la méthode de résolution *'b ← a - c*2'*, on enverra le message:

Méthode result: 'b'
*calcul: 'a - c*2'*

Une méthode d'instance permettant d'appliquer la méthode de résolution a été définie dans le protocole 'application'. Elle évalue l'expression référencée par la variable d'instance **calcul** et assigne le résultat à la variable d'instance de l'objet contraint désignée, directement ou indirectement, par la variable d'instance **résultat.**

Afin d'accélérer l'évaluation d'une méthode de résolution au cours du processus de satisfaction des contraintes, une méthode d'instance permettant de compiler les méthodes de résolution d'une contrainte de classe a été définie dans le protocole 'compilation' de la classe CLASS. La méthode compilée est stockée dans la variable d'instance **méthodeCompilée** de la méthode de résolution. De la même façon, une méthode d'instance permettant de compiler les méthodes de résolution d'une contrainte d'instance a été définie dans le protocole 'compilation' de la classe OBJECT. Ainsi, chaque fois qu'une contrainte cherchera à appliquer une de ses méthodes de résolution, elle lancera directement la méthode compilée correspondante sans passer par l'étape intermédiaire d'interprétation de la méthode de résolution (interprétation de **calcul** et interprétation de **résultat**).

Remarque : Pour créer une méthode se composant de plusieurs messages, les variables d'instance **résultat** et **calcul** seront données sous la forme d'un tableau.
Par exemple, pour créer la méthode 'a ← b - c*4 . b ← a + d', on enverra le message:

Méthode result: #('a' 'b')
*calcul: #('b - c*4' 'a + d').*

3.2.6.4 La classe CONTRAINTE

La classe CONTRAINTE est super-classe de toutes les classes modélisant une contrainte. Dans cette classe sont décrits la structure et le comportement communs à toute contrainte.

CONTRAINTE est sous classe de la classe OBJECT et possède trois variables d'instance principales :

- **règle** qui référence le prédicat de la contrainte, instance de la classe REGLE ou d'une de ses sous classes,
- **méthodes**, qui référence une instance de la classe ORDEREDCOLLECTION, dont les éléments sont les méthodes de résolution de la contrainte,
- **modèle** qui référence l'objet contraint (ou les objets contraints) mis en cause par la contrainte.

Les protocoles de création, de comparaison, de vérification et de résolution d'une contrainte sont définis dans la classe CONTRAINTE.

Pour créer une contrainte, on envoie simplement le message :

Contrainte
règle: UneRègle
méthodes: uneOrderedCollectionDeMéthodes.

La contrainte définie sur la classe LIGNE_A_POINT_MILIEU est ainsi créée par le message :

Contrainte
règle: (Regle new: 'midPoint = (line beginPoint + line endPoint)/2')
méthodes: (OrderedCollection new
add: (Méthode result: 'midPoint' calcul: '(line beginPoint + line endPoint)/2);
*add: (Méthode result: 'line beginPoint' calcul: 'midPoint*2 - line endPoint');*
*add: (Méthode result: 'line endPoint' calcul: 'midPoint*2 - line beginPoint')).*

Les méthodes de comparaison des contraintes nous permettent seulement pour le moment de vérifier l'égalité de deux contraintes. Deux contraintes sont identiques si leurs règles sont semblables et si elles possèdent également les mêmes méthodes de résolution. Si deux contraintes ont une même règle mais n'ont pas le même ensemble de méthodes de résolution, elles sont considérées comme étant différentes. Dans ce cas précis, la composition des contraintes permet de déduire une contrainte résultante ayant

pour prédicat la règle commune aux deux contraintes et pour méthodes de résolution la réunion des ensembles de méthodes de résolution de chacune des contraintes.

Les méthodes permettant à une contrainte de vérifier si son prédicat est vrai et d'appliquer une de ses méthodes de résolution sont définies dans le protocole 'satisfaction' de la classe CONTRAINTE.

La méthode *'satisfaite'* permet de s'assurer que la règle définissant la contrainte est satisfaite :

```
satisfaite
    règle vérifiée ifTrue: [^true].
    ^false
```

La méthode *'satisfaire: aString'* permet d'appliquer une des méthodes de résolution de la contrainte. L'argument, aString, guide le choix de la méthode à appliquer parmi l'ensemble des méthodes définies : si aString est le nom d'une variable d'instance de l'objet-contraint ou le chemin menant à cette variable, la méthode choisie sera la première des méthodes ne modifiant pas cette variable (c'est à dire la première méthode ayant aString inclus comme sous-séquence de sa variable d'instance **calcul** et non inclus comme sous-séquence de sa variable d'instance **résultat**) :

```
satisfaire: aString

    "applique un des méthodes de résolution de la contrainte, ne remettant pas en
cause le changement indiquer par aString"

    | amethode |
    amethode <- méthodes detect: [ :methode |
            ('*',aString,'*' match: méthode calcul asString)
    and:[('*',aString,'*' match: méthode résultat asString)not ]]
                ifNone: [ méthodes at: 1].
    amethode appliquer
```

3.2.6.5 La classe MULTICONTRAINTES

Les instances de la classe MULTICONTRAINTES sont des contraintes multiples, c'est à dire des contraintes définies par plusieurs prédicats.

La classe MULTICONTRAINTES possède deux variables d'instance :

- **corps** qui référence une instance de la classe ORDEREDCOLLECTION dont les éléments sont des contraintes,
- **model** qui référence l'objet contraint.

Une contrainte multiple, instance de la classe MULTICONTRAINTES est créée par le message :

Multicontraintes sur: uneOrderedCollectiondeContraintes

Comme pour la classe CONTRAINTE, les protocoles de création, de vérification et de résolution d'une contrainte multiple sont définis dans la classe MULTICONTRAINTES. Une multicontrainte est satisfaite si chacune des règles des contraintes qui la composent est vérifiée. Satisfaire une contrainte multiple, c'est satisfaire, parmi les contraintes qui la composent, toutes les contraintes dont la règle n'est pas vérifiée.

3.3. LES CONTRAINTES EN PROGRAMMATION ORIENTEE OBJET : RESEAU D'OBJETS CONTRAINTS ET RESEAU DE CONTRAINTES

Le concept d'objet permet une description fonctionnelle des différents éléments constituant un système. Pour conserver le caractère descriptif de la représentation en termes d'objets, les contraintes que nous avons introduites dans notre système sont des contraintes constructives. Le prédicat de la contrainte décrit ainsi une relation sur des variables d'instance nommées définies pour un ou plusieurs objets. Dans l'exemple de la classe LIGNEAPOINTMILIEU, le prédicat de la contrainte s'écrit simplement *'pointml = 1/2(extrémité1 + extrémité2)'*, où **pointml**, extrémité1 et **extrémité2** sont des variables d'instance de la classe. De plus les contraintes constructives permettent l'utilisation d'un mécanisme interne de satisfaction des contraintes, capable de résoudre les contraintes établies sur un système en déclenchant automatiquement une séquence de méthodes de résolution, ce qui va simplifier considérablement la construction des applications.

3.3.1 L'objet contraint

Nous avons vu au chapitre 2 qu'il est possible en programmation orientée objet de définir deux types de contraintes, les contraintes de classe et les contraintes d'instance.

La notion de contrainte de classe a déjà été implantée dans certains langages à objets (Cf. annexe_2). Le principal défaut de ces systèmes est qu'ils ne permettent pas de définir une contrainte sur une classe d'objets, dite classique, et donc d'intégrer après coups les objets contraints dans une application déjà existante. D'autre part la notion de contrainte d'instance n'a, à notre connaissance, jamais été implantée dans un langage de

programmation orientée objet. Notre soucis a donc été d'établir une méthodologie permettant d'intégrer la notion de contrainte dans un langage orienté objet quelconque et offrant, à l'utilisateur, la possibilité de définir tous types de contraintes (de classe ou d'instance, statiques ou dynamiques), même sur les objets d'une application déjà existante.

3.3.1.1) La contrainte de classe

La contrainte de classe est une contrainte partagée par tous les objets de la classe. Sa définition s'apparente donc à la définition d'une variable de classe. Seule la procédure d'héritage est différente selon qu'il s'agisse d'une contrainte ou d'une variable de classe.

Une sous-classe hérite directement, sans contrôle, des variables de classe définies pour ses super-classes, alors que l'héritage d'une contrainte peut être source de conflits :

- la contrainte définie pour une classe peut contredire, en partie ou en totalité, une contrainte définie dans une de ses super-classes ;
- la contrainte définie pour une classe peut inclure une contrainte déjà définie dans une super-classe;
- en cas d'héritage multiple, les contraintes définies pour des super-classes de même niveau peuvent être en conflit.

Pour prendre en compte ces différentes possibilités de conflits ou de redondance, il suffit d'intégrer dans la procédure d'héritage d'une contrainte de classe des opérateurs de comparaison et de composition de contraintes.

Il apparaît donc nécessaire de modifier la structure même de la classe pour qu'elle permette la définition d'un nouveau type de variable : la contrainte de classe. D'autre part cette modification de structure doit pouvoir être acceptée par les classes déjà existantes dans le système.

3.3.1.2) La contrainte d'instance

Pour associer une contrainte de classe à un objet, il nous a suffit de mémoriser la contrainte dans une variable spécialement créée, définie dans la structure de sa classe. La contrainte d'instance étant spécifique à l'objet, elle ne peut être mémorisée dans une telle variable.

Dans beaucoup de langages orientés objet, s'il est possible de modifier la structure d'une classe, il est impossible de changer la structure d'un objet. On ne peut donc suivre la même démarche que précédemment pour associer une contrainte d'instance à un objet.

Le plus simple dans ce cas est de créer un dictionnaire des instances contraintes, déclaré en variable globale. Les données stockées dans ce dictionnaire sont des associations de type 'objet → {contraintes}'.

Les méthodes permettant de définir, de modifier ou de détruire une contrainte d'instance sur un objet doivent être définies dans la classe d'objet se trouvant à la racine de la hiérarchie des classes.

Remarque: la démarche décrite ici est identique à celle suivie pour définir une dépendance sur un objet : le dictionnaire des dépendances est défini en variable globale et les méthodes d'accès à ce dictionnaire sont définies dans la classe Object (Cf. chap. 1, § 1.2).

3.3.2 Implantation en Smalltalk-80

3.3.2.1) La contrainte de classe

La notion de contrainte définie pour une classe d'objet a été introduite par le biais d'un nouveau type de variables, la variable **classConstraintNetwork**, partagée par toutes les instances de la classe mais dont l'héritage suit certaines lois.

Le message de création d'une classe contrainte reste très proche du message classique de création des classes :

```
#NameOfSuperClass subclass: #NameOfClass
    instanceVariableNames: 'intsVarName1 instVarName2'
    classVariableNames: 'ClassVarName1 ClassVarName2'
    poolDictionaries: ''
    classConstraintNetwork: aConstraint
    category: 'cat'
```

Lors de la définition de la classe contrainte, la contrainte de classe donnée par l'argument *aConstraint* est prise en compte de la façon suivante :

Le système regarde si la classe créée hérite d'une super-classe contrainte
- si non, la contrainte *aConstraint*, quand elle existe, est affectée à la variable **classConstraintNetwork** de la classe créée ;
- si oui, le système fusionne dans une contrainte multiple les contraintes associées à la super-classe et la contrainte *aConstraint*, si celle-ci existe. La contrainte multiple est ensuite affectée à la variable **classConstraintNetwork** de la classe créée

Pour la fusion des contraintes nous considérons, pour le moment, que toutes les contraintes définies sur un objet s'additionnent. Plus exactement, si une classe contrainte hérite d'une super-classe contrainte, alors la contrainte résultante pour la classe créée est une multicontrainte constituée de toutes les contraintes définies dans la hiérarchie. Pour prendre en compte les différentes possibilités de conflits ou de redondance dans l'héritage des contraintes, il suffira d'intégrer dans la procédure d'héritage de la variable **classConstraintNetwork** (définie dans la classe CLASSBUILDER) des opérateurs de comparaison, d'intersection et de composition de contraintes.

Plusieurs méthodes permettant de modifier, d'ajouter ou de supprimer une contrainte à la contrainte de classe, ont été ajoutées au dictionnaire des méthodes de la classe CLASSE. Elles permettent entre autres de définir une contrainte de classe sur une classe existante, qu'elle soit déjà ou non une classe contrainte.

3.3.2.2) Le Browser 'System Constraint Browser'

La hiérarchie des classes contraintes forme une nouvelle hiérarchie de classes. Pour plus de clarté, nous avons créé une nouvelle fenêtre, de fonction identique à une fenêtre browser normale, mais exclusivement réservée à la représentation des classes contraintes. C'est dans cette fenêtre, instance de la classe BROWSERVIEW, de label "System Constraint Browser", que seront définies toutes les classes contraintes (figure 3.11).

Pour créer le browser de classes contraintes, nous avons ajouté, dans la classe BROWSER, la variable d'instance **sousorganization** qui référencera la variable globale **SousSystemOrganization**. **SousSystemOrganization**, est une instance de SYSTEMORGANIZER, tout comme la variable globale **SystemOrganization**, mais si cette dernière référence l'organisation de toutes les classes, contraintes ou non, la première ne référencera que l'organisation des classes contraintes. A l'affichage, toutes les

apparaîtront dans le browser 'System Constraint Browser' et toutes les catégories de classes, appartenant à la variable globale **SystemOrganization** et n'appartenant pas à **SousSystemOrganization**, apparaîtront dans le browser 'System Browser' (figure 3.12).

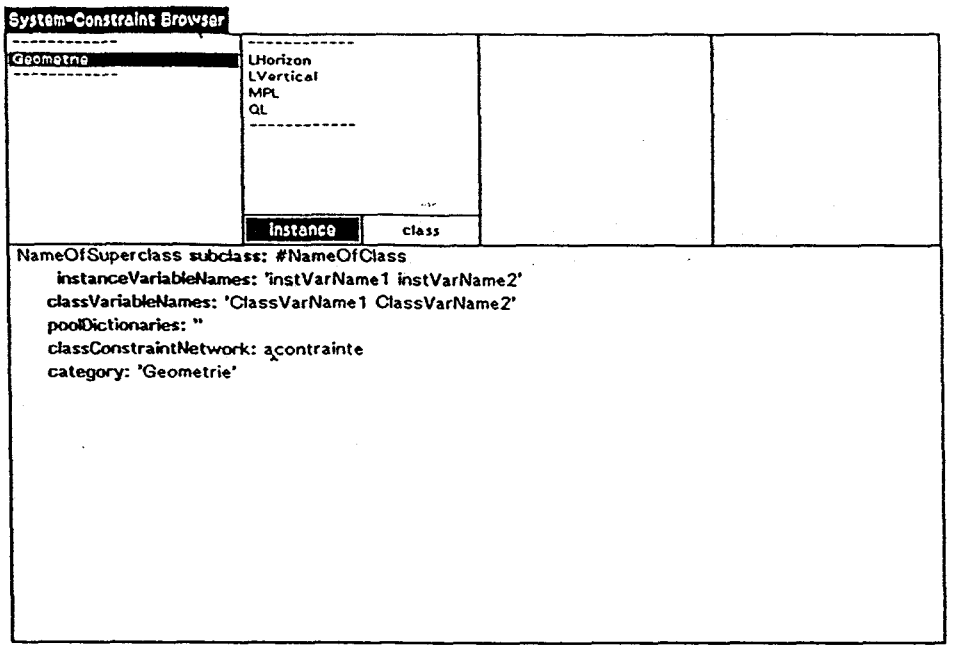


Figure 3.11

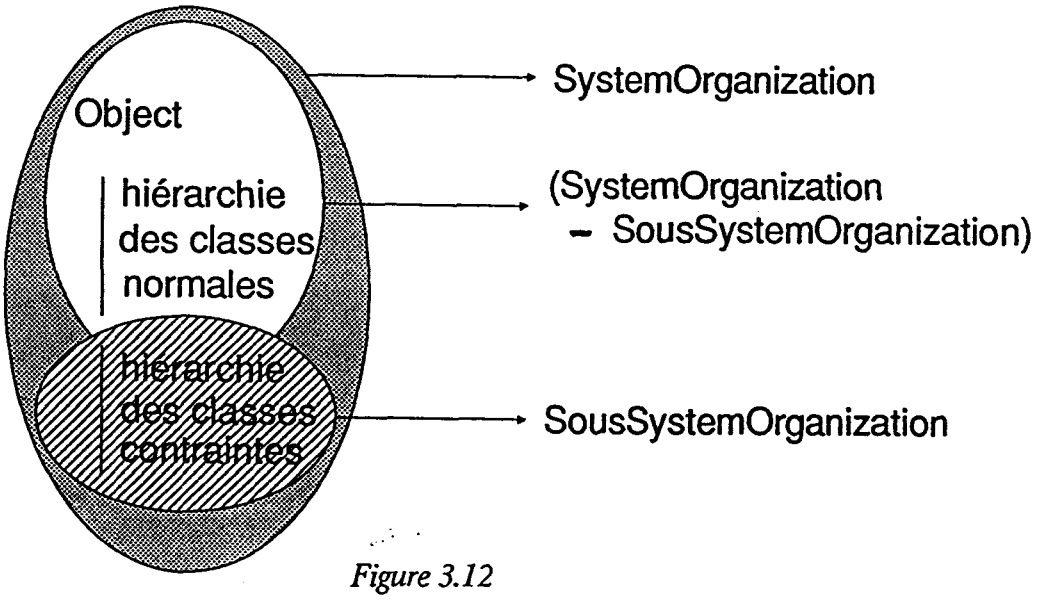


Figure 3.12

3.3.2.3) La contrainte d'instance

Pour créer une contrainte sur une instance, il suffit d'ajouter l'association (objet → contrainte) dans le dictionnaire ConstrainObjectDictionary défini en variable globale. Ce dictionnaire référence toutes les contraintes d'instance établies sur le système.

Les méthodes d'instance permettant de définir ou de détruire une contrainte sur un objet ont été ajoutées dans la classe OBJECT.
La méthode *'contraint: aConstraint'* permet de définir la contrainte, *aConstraint*, sur un objet sans contrainte d'instance.
Les méthodes *'addOwnConstraint: aConstraint'* et *'removeOwnConstraint: aConstraint'* ajoute ou enlève, à l'ensemble des contraintes d'instance définies sur l'objet contraint, la contrainte d'instance *aConstraint*.

Si une contrainte d'instance est définie entre plusieurs objets contraints, chacun de ces objets référence la contrainte dans le dictionnaire ConstrainObjectDictionary.

CONCLUSION

Ce chapitre décrit deux représentations totalement différentes des contraintes et des réseaux de contraintes.

Le formalisme mathématique, adopté pour décrire les contraintes non constructives, a été intégré dans de nombreux langages à base de contraintes. Il a été développé et étendu pour représenter également des contraintes établies entre variables ayant des domaines de définition non finis. Aujourd'hui ce formalisme est surtout utilisé dans les outils de programmation logique avec contraintes [DEC][JAF][HEI].

Le second formalisme présenté est celui nous avons utilisé pour intégrer les contraintes dans la programmation orientée objet. Il permet à l'utilisateur de définir les contraintes, si ce n'est de façon déclarative, au moins de façon descriptive. La définition des contraintes est la seule tâche qui incombe à l'utilisateur, la résolution de l'ensemble des contraintes établies sur un système est la tâche d'un mécanisme interne de satisfaction des contraintes dont le fonctionnement est décrit au chapitre suivant.

CHAPITRE 4 : LA SATISFACTION DES CONTRAINTES

CHAPITRE 4 : LA SATISFACTION DES CONTRAINTES

Le mécanisme interne de satisfaction des contraintes est un mécanisme, transparent à l'utilisateur, dont la seule tâche est de maintenir résolues en permanence toutes les contraintes définies sur le système : c'est le coeur de tout système à base de contraintes.

Ce chapitre décrit différentes méthodes de satisfaction des contraintes.

Les premières méthodes exposées traitent la résolution des contraintes non constructives. Ces méthodes suivent deux démarches opposées. Certaines guident l'instanciation des variables par l'évaluation des contraintes, d'autres guident l'évaluation des contraintes par l'instanciation des variables.

La seconde partie de ce chapitre définit le principe de propagation des contraintes appliqué à la satisfaction des contraintes constructives. Deux cas de satisfaction des contraintes sont répertoriés, la satisfaction par propagation initiale des contraintes et la satisfaction par propagation dynamique des contraintes.

Enfin, la dernière partie de ce chapitre expose la méthode de propagation des contraintes que nous avons développée pour la satisfaction des réseaux d'objets contraints. Cette méthode dirige l'évaluation des contraintes par l'instanciation des variables de l'objet contraint (ou des objets contraints).

4.1. METHODES DE SATISFACTION DES CONTRAINTES NON CONSTRUCTIVES [BER]

Satisfaire un réseau de contraintes non-constructives défini sur n variables de domaines de définition respectifs D_1, ..., D_n, revient à effectuer l'une des opérations suivantes :

- vérifier si un n-uplet (élément du produit cartésien D_1*...*D_n) est une solution globale possible pour le réseau ; en d'autres termes, vérifier si les valeurs qu'il attribue aux n variables du réseau sont telles que toutes les contraintes soient simultanément satisfaites ;

- trouver une solution au réseau de contraintes, c'est à dire, trouver un n-uplet, affectation des n variables du réseau, vérifiant simultanément toutes les contraintes ;

- trouver toutes les solutions possibles pour le réseau, soit l'ensemble des n-uplets, sous-ensemble du produit cartésien D_1*...*D_n, permis par toutes les contraintes du réseau.

Pour vérifier si un n-uplet satisfait globalement un réseau de contraintes, il suffit de tester successivement le prédicat de chacune des contraintes. Si un seul de ces tests s'avère négatif, le n-uplet n'est pas solution du réseau.

Les méthodes de scrutation d'une base de faits basées sur le retour arrière sont particulièrement bien adaptées à la recherche d'une solution dans le cas d'un réseau de contraintes non-constructives (ici, les faits considérés sont les contraintes). Contrairement à une méthode plus simple mais beaucoup plus coûteuse en temps de calcul, qui consiste à tester successivement tous les éléments du produit cartésien D_1*...*Dn jusqu'à trouver un n-uplet solution, elles combinent instanciation et évaluation progressives des contraintes du réseau.

Soit les évaluations sont guidées par les instanciations, dans ce cas on considère d'abord les contraintes établies sur les variables initialisées, ce qui permet d'instancier de nouvelles variables. Ces variables mettant en cause d'autres contraintes, celles-ci seront à leur tour considérées et ainsi de suite jusqu'à ce que toutes les variables du

réseau soient instanciées. Cette démarche se base sur l'adjacence des variables dans un réseau de contraintes (elle évalue de proche en proche les arcs du réseau de contraintes en partant d'un ensemble de noeuds initiaux).

Soit les instanciations sont guidées par les évaluations, ce qui consiste à traiter les contraintes selon un ordre établi et à instancier les variables au fur et à mesure de leur évaluation (Prolog).

Dans les deux cas, si une contradiction survient, c'est à dire si la contrainte évaluée ne peut être validée compte tenu des instanciations précédentes, la valeur attribuée à la dernière variable instanciée est remise en cause, puis le processus d'instanciation/évaluation est relancé à partir de ce point de retour.

De nombreuses variantes et améliorations peuvent être apportées au principe de retour arrière décrit précédemment. Elles portent en général sur l'ordre d'instanciation des variables, sur le choix de la valeur à attribuer ou sur la position du retour (possibilité de retour par saut).

Soit par exemple le réseau de contraintes présenté figure 4.1. A l'initialisation, seule la variable x_1 est instanciée (x_1 = a).

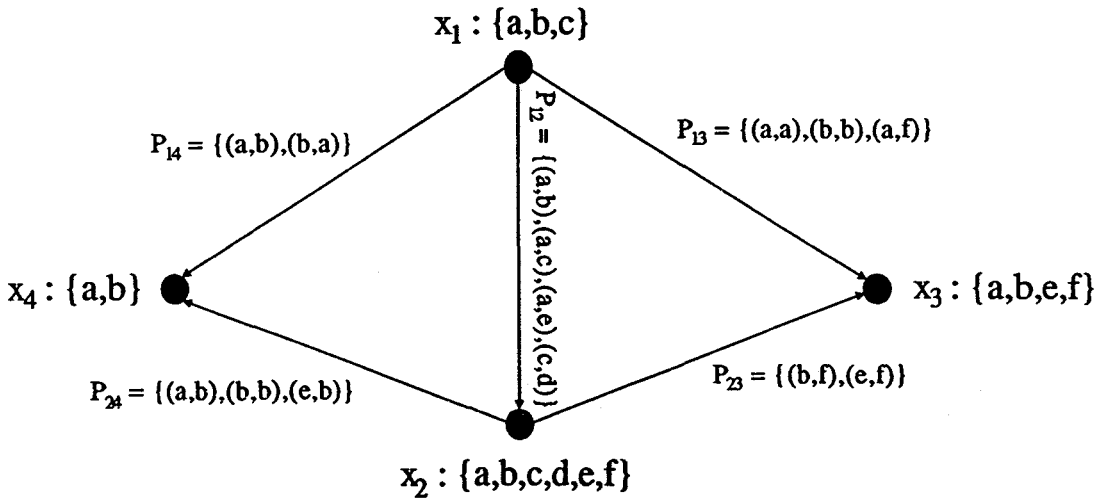


Figure 4.1

- Si l'évaluation des contraintes est guidée par les instanciations, la première contrainte évaluée est P_{12} (ou P_{13} ou P_{14} selon le choix imposé par l'algorithme). Cette contrainte, si x_1 = a, est satisfaite pour x_2 = b, x_2 = c ou x_2 = e. Si le choix se fait sur la première instanciation possible, on aura x_1 = a et x_2 = b. La dernière variable

instanciée étant x_2, la prochaine contrainte évaluée est P_{23} (ou P_{24}). P_{23} donne x_3 = f, ce qui vérifie P_{13}. On évalue alors P_{14} (puisque P_{12} et P_{13} ont déjà été vérifiées), ce qui donne x_4 = b et qui vérifie P_{24}.

- Si l'instanciation des variables est guidée par l'évaluation des contraintes, on décidera par exemple de scruter les contraintes dans l'ordre P_{12}, P_{13}, P_{14}, P_{23} et P_{24}. Si x_1 = a, P_{12} est satisfaite pour x_2 = b, x_2 = c ou x_2 = e. On choisit x_2 = b. P_{13} n'est satisfaite que pour x_3 = a ou x_3 = f. On choisit x_3 = a. P_{14} impose x_4 = b. P_{23}, si x_2 = b et x_3 = a, n'est pas satisfaite. On remet dons en cause la dernière instanciation, ici x_4 = b. Si x_4 est différent de b, P_{14} ne peut être vérifiée. On remet donc en cause l'instanciation précédente, soit x_3 = a, et on choisit x_3 = f. P_{14} impose de nouveau x_4 = b. P_{23}, si x_2 = b et x_3 = f, est satisfaite. Il en est de même pour P_{24} si x_2 = b et x_4 = b.

L'exemple présenté montre qu'il est souvent plus rapide de se laisser guider par l'instanciation que par l'évaluation. Cependant dans les deux cas, la vérification de la cohérence du réseau (présentée au chapitre 3, § 3.1.3) est une méthode de pré-calcul qui permet de gagner énormément de temps lors de la satisfaction du réseau de contraintes en limitant le nombre de choix possibles. Après vérification de sa cohérence, le réseau présenté figure 4.1 devient le réseau présenté figure 4.2 :

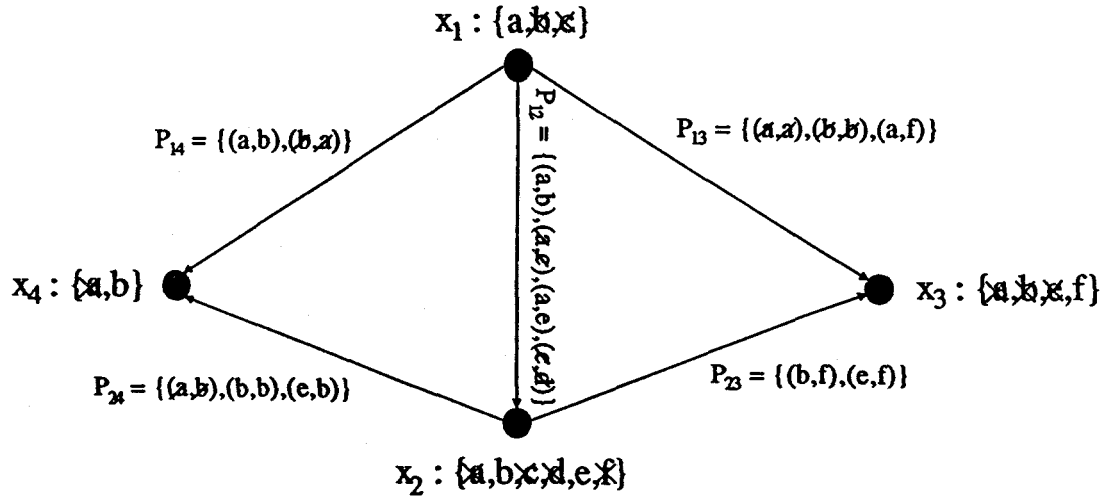


Figure 4.2

La procédure de vérification de cohérence a permis, ici, de limiter le choix pour la variable x_2 à deux valeurs {b,e} et de ne laisser aucun choix pour les autres variables x_1 = {a}, x_3 = {f} et x_4 = {b}.

Rina Dechter [DEC], décrit de façon assez complète différents algorithmes de satisfaction des contraintes utilisant le retour arrière. Elle y expose également une extension des méthodes de vérification de cohérence des réseaux, adaptées aux réseaux ordonnés ("d-arc-consistency" et "d-path-consistency"). Notre propos n'étant pas ici de développer davantage ces techniques, nous invitons le lecteur à se référer aux travaux précédemment cités.

Eugène C. Freuder [FRE] propose, quant à lui, une autre méthode de satisfaction des contraintes non-constructives qui n'est pas basée sur le retour arrière. Ayant constaté que le fait de rendre un réseau cohérent par rapport à ses noeuds, puis ses arcs et ses chemins, réduit progressivement l'espace de recherche mais ne suffit pas à synthétiser complètement la contrainte globale, E. C. Freuder généralise la notion de cohérence par rapport aux noeuds/ars/chemins à celle de k-cohérence d'un sous-réseau, établi sur k variables ($k \leq n$) du réseau de contraintes initial. Il est démontré que la vérification de k-cohérence d'un réseau, pour k variant successivement de 1 à n, permet alors d'isoler l'ensemble des n-uplets satisfaisant la contrainte globale.

4.2. METHODES DE SATISFACTION DES CONTRAINTES CONSTRUCTIVES

[BOR][BER][GÜS]

Satisfaire un réseau de contraintes, dans le cas de contraintes constructives, prend un autre sens. Pour un tel réseau, la donnée d'un certain nombre de variables permet d'instancier les autres variables du réseau, en utilisant les méthodes de résolution des contraintes. Dans l'exemple plusieurs fois cité de la contrainte 'a = b + c', il suffit que deux des variables soient instanciées pour que l'on puisse en déduire la valeur de la troisième en appliquant l'une des trois méthodes de résolution proposées.

Nous distinguerons deux cas de satisfaction de contraintes que nous appellerons la satisfaction initiale des contraintes et la satisfaction dynamique des contraintes.

Satisfaction initiale des contraintes :

Seules certaines variables du réseau sont déjà instanciées. Dans ce cas, satisfaire le réseau, c'est chercher à instancier les variables encore indéterminées, de façon à satisfaire toutes les contraintes du réseau.

Pour ce faire, il suffit d'examiner les contraintes agissant sur les variables connues. Certaines de ces contraintes vont permettre, par leurs méthodes de résolution, de déterminer de nouvelles variables jusque là indéterminées. Les instanciations de ces nouvelles variables mettront en cause d'autres contraintes, qui permettront à leur tour de déterminer d'autres variables et ainsi de suite jusqu'à la détermination de toutes les variables du réseau. Cette technique est appelée propagation des contraintes.

Par exemple, pour créer une ligne à point milieu, il suffit de donner les coordonnées de ses deux extrémités. Le point-milieu de la ligne n'étant pas instancié, le prédicat de la contrainte *'pointml = 1/2(extrémité1 + extrémité2)'* ne peut être vérifié. Le mécanisme de satisfaction des contraintes résout la contrainte du point-milieu en appliquant, parmi les méthodes de résolution de la contrainte, celle permettant d'instancier le point-milieu ('pointml ← 1/2(extrémité1 + extrémité2)').

Il se peut, dans certain cas, que le nombre de variables instanciées ne soit pas suffisant pour permettre de résoudre toutes les contraintes définies sur le réseau de variables. Dans ce cas la propagation initiale des contraintes échoue.

Satisfaction dynamique des contraintes :

Toutes les variables du réseau de contraintes sont instanciées et toutes les contraintes sont satisfaites (on dira que le réseau est stable pour ses contraintes) puis une des variables est modifiée. Dans ce second cas, satisfaire le réseau de contraintes, c'est modifier la valeur d'autres variables de façon à rétablir la stabilité du réseau, pour qu'il vérifie de nouveau toutes ses contraintes. Dans ce cas, comme dans le cas précédent, la valeur de la variable initialement modifiée va être propagée, le long des contraintes agissant sur cette variable, ce qui entraînera la modification de nouvelles variables qui seront elles aussi propagées...

L'exemple du quadrilatère présenté au chapitre 2 (§ 2.2.3.3) illustre un cas de satisfaction dynamique des contraintes.

Si toutes les contraintes définies sur le réseau ne peuvent être satisfaites, plusieurs démarches sont possibles :

- le changement effectué sur le réseau et ayant entraîné son instabilité est refusé.
Si l'utilisateur veut déplacer, à l'aide de la souris, une ligne dont les extrémités sont ancrées, son action sera refusée par le système ;

- certaines contraintes sont délaissées au profit de contraintes plus impératives (soit le mécanisme interne de satisfaction des contraintes a la possibilité de comparer le degré de priorité des contraintes [BOR], soit il fait appel à l'utilisateur pour le guider dans ce choix) ;

- le mécanisme de satisfaction des contraintes fait appel à un algorithme de relaxation (ou approximation numérique) qui cherchera à résoudre au mieux l'ensemble de contraintes définies sur le réseau. L'algorithme de relaxation instancie arbitrairement les variables du réseau et regarde dans quelle mesure les contraintes sont satisfaites. Pour ce faire, il utilise une fonction erreur, définie pour chaque contrainte, dont l'évaluation donne le degré de satisfaction de celle-ci (zéro si la contrainte est satisfaite, un nombre plus ou moins élevé selon que la contrainte est plus ou moins bien satisfaite). Par itération, l'algorithme de relaxation ajustera les valeurs affectées aux variables du réseau pour que les fonctions erreur des contraintes décroissent ;

- le mécanisme interne de satisfaction des contraintes envoie un message d'erreur signalant à l'utilisateur que la satisfaction des contraintes a échoué [SUS] ;

Lors de l'initialisation d'un réseau de contraintes, il est fréquent d'avoir à la fois une satisfaction dynamique et une satisfaction initiale des contraintes. C'est le cas, en particulier, lorsque les variables du réseau sont mal initialisées (réseau initialement incohérent).

Par exemple, pour créer une instance de la classe **Ligne_Horizontale_à_Point_Milieu**, il suffit de donner les coordonnées des deux points extrémités (figure 4.3.a). Sur cet objet partiellement initialisé sont définies deux contraintes, la contrainte d'horizontalité et la contrainte du point-milieu. L'objet créé ne répondant pas à sa première contrainte, le mécanisme de satisfaction des contraintes rétablit l'horizontalité de la ligne en appliquant l'une des méthodes de résolution de la contrainte, ici on applique la méthode

'extrémité1 y ← extrémité2 y' (satisfaction dynamique de la contrainte, figure 4.3.b). Le point-milieu de la ligne n'étant pas instancié, le prédicat de la contrainte *'pointml = 1/2(extrémité1 + extrémité2)'* ne peut être vérifié. La contrainte est résolue en appliquant une de ses méthodes de résolution qui permet d'affecter la variable *pointml* (satisfaction initiale de la contrainte, figure 4.3.c).

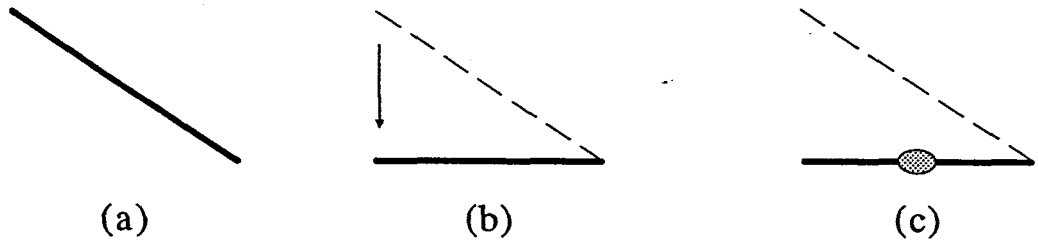


Figure 4.3

4.2.1) Propagation locale et propagation globale

La propagation locale désigne couramment la propagation d'une valeur le long d'une contrainte : la propagation locale de la valeur attribuée à un noeud du réseau se fait le long d'un des arcs partant de ce noeud. C'est une prise en compte, par une partie du réseau, du changement intervenu sur une variable.

La propagation globale désigne la propagation d'une valeur, attribuée à une variable, le long de toutes les contraintes agissant sur cette variable, puis la propagation globale de toutes les variables affectées par cette propagation : la propagation globale de la valeur attribuée à un noeud du réseau se fait le long de tous les arcs partant de ce noeud, puis le long de tous les arcs partant de chacun des noeuds atteints... C'est une prise en compte, par la totalité du réseau, du changement intervenu sur une variable.

4.2.2) Principe de propagation

Nous présentons ici le principe de propagation dans sa version la plus simple [BER], tout en sachant bien sûr que de nombreuses subtilités pourraient en accroître les performances. Cette version s'applique plus particulièrement à la satisfaction initiale des contraintes mais peu de modifications doivent y être apportées pour traiter le cas de la satisfaction dynamique.

On distingue sur le réseau l'ensemble des variables instanciées. Les contraintes agissant sur ces variables sont stockées dans un ensemble. Il s'agit de l'ensemble des contraintes déclenchables, c'est à dire l'ensemble des contraintes pouvant permettre de déterminer, à partir de variables instanciées, des variables encore indéterminées.
Pour satisfaire le réseau de contraintes, on exécute le cycle suivant :

Tant que l'ensemble des contraintes déclenchables n'est pas vide, on enlève la première contrainte de l'ensemble et on lui applique l'une des règles suivantes :

- si toutes les variables mises en cause par la contrainte sont déjà instanciées et si la contrainte est satisfaite, ne rien faire ;

- si toutes les variables mises en cause par la contrainte sont déjà instanciées et si la contrainte n'est pas satisfaite, stopper le processus de propagation et signalez une incohérence ;

- si la contrainte ne peut être résolue, par manque de données (c'est à dire s'il n'y a pas assez de variables connues pour appliquer l'une des méthodes de résolution de la contrainte), ne rien faire. La contrainte sera réexaminée lorsque de nouvelles variables, participant à cette même contrainte, auront été instanciées ;

- si la contrainte peut être résolue, appliquer une de ses méthodes de résolution et ajouter dans l'ensemble des contraintes déclenchables toutes les contraintes définies sur les variables nouvellement déterminées, sauf la contrainte en cours de résolution et les contraintes appartenant déjà à l'ensemble.

Si après exécution du cycle complet toutes les variables du réseau n'ont pu être instanciées, le problème ne peut être résolu par simple propagation en partant de l'état initial donné.

Pour démarrer la propagation des contraintes, il est nécessaire qu'un certain nombre de variables du réseau soient déjà instanciées. Cette instanciation initiale du réseau peut être imposée par l'utilisateur, auquel cas si la propagation échoue, la satisfaction des contraintes échoue également. L'instanciation initiale peut, à l'inverse, être proposée arbitrairement par le mécanisme interne de satisfaction des contraintes, dans ce cas, si la propagation aboutit à un échec, ce choix peut être remis en cause. La satisfaction des contraintes n'échouera que si la propagation échoue pour toutes les instanciations initiales proposées par le mécanisme.

Le principal avantage de la propagation des contraintes sur les méthodes utilisant le retour arrière est qu'elle permet de ne pas instancier, de façon hypothétique, certaines variables du réseau : **lorsque des variables sont instanciées, suite à l'application de l'une des méthodes de résolution d'une contrainte, elles le sont de manière certaine et la contrainte résolue est obligatoirement vérifiée.**

L'algorithme de propagation des contraintes présenté guide l'instanciation des variables par l'évaluation des contraintes du réseau. En effet, après l' initialisation de certaines variables, toutes les contraintes déclenchables sont placées dans un ensemble, pour être traitées une à une. Leur évaluation respective permettra d'instancier de nouvelles variables et les contraintes rendues déclenchables par ces instanciations seront ajoutées à l'ensemble pour être à leur tour évaluées.

La majorité des algorithmes de propagation sont basés sur ce principe de l'instanciation guidée par l'évaluation. Pourtant, si la technique inverse de l'évaluation guidée par l'instanciation est plus efficace pour satisfaire les réseaux de contraintes non-constructives, on peut supposer qu'il en est de même pour les réseaux de contraintes constructives.

La troisième partie de ce chapitre (§ 4.3) décrit la méthode de propagation employée par le mécanisme interne de satisfaction des contraintes de notre plate-forme de simulation. A l'inverse du processus de propagation décrit précédemment, elle guide l'évaluation et la résolution des contraintes par l'instanciation des variables.

4.2.3) Propagation et satisfaction dynamiques des contraintes

Le problème posé par la satisfaction dynamique des contraintes est de gérer la stabilité d'un réseau de contraintes à travers son évolution : un réseau de contraintes évolue si la valeur d'une de ses variables est modifiée, si on lui ajoute ou si on lui retire une contrainte, si on lui ajoute ou si on lui retire une variable. Il ne s'agit donc pas, dans ce cas, d'établir la valeur des variables indéterminées sur le réseau, puisque toutes les variables sont déjà instanciées, mais de vérifier si les valeurs attribuées à ces variables sont toujours correctes compte tenu de l'évolution du réseau et des contraintes qui les lient. Si tel n'est pas le cas, d'autres variables du réseau de contraintes devront à leur tour être modifiées pour rétablir sa stabilité.

Le principe de propagation décrit précédemment, appliqué à la satisfaction

dynamique des contraintes, reste inchangé [BER] :

Toutes les contraintes mises en cause par un changement survenu sur le réseau sont stockées dans un ensemble. On effectue alors le cycle suivant:

> Tant que cet ensemble de contraintes déclenchables n'est pas vide, on retire la première contrainte de l'ensemble et on lui applique une des règles suivantes :
>
> - *si la contrainte est satisfaite, ne rien faire ;*
>
> - *si la contrainte n'est pas satisfaite, appliquer l'une de ses méthodes de résolution et ajouter à l'ensemble des contraintes déclenchables toutes les contraintes définies sur les variables nouvellement modifiées, sauf la contrainte en cours de résolution et les contraintes appartenant déjà à l'ensemble.*

La principale difficulté est alors de choisir parmi les méthodes de résolution de la contrainte évaluée laquelle doit être appliquée pour satisfaire la contrainte. Ce choix est souvent très difficile à faire, il est fonction de la contrainte, du changement initialement intervenu sur le réseau, des changements qu'il va lui-même entraîner sur les variables du réseau, du domaine d'application...

Il est impossible de donner une règle universelle pour le choix d'une méthode de résolution dans la satisfaction d'une contrainte. En règle générale, ce choix s'appuie sur l'un ou l'autre des principes suivants :

- le changement intervenu initialement sur le réseau ne doit pas être remis en cause sauf s'il n'existe aucun autre moyen de rétablir la stabilité du réseau ;

- les contraintes étant maintenues en permanence par un mécanisme interne de résolution et non par l'utilisateur, il faut éviter de dérouter ce dernier. En d'autres termes, il faut que l'utilisateur obtienne ce à quoi il puisse logiquement s'attendre. On appelle cela le principe de moindre étonnement [BOR]. Souvent ce principe impose que le nombre de variables remises en cause par la satisfaction des contraintes soit minimal ;

- les variables peuvent être marquées de la date de leur dernière modification. En effet, dans certains cas, on peut choisir de modifier en premier lieu les variables les plus anciennes ou les plus récentes ;

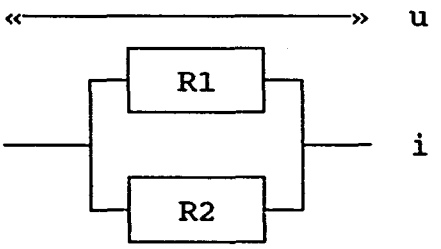
- la sémantique de la contrainte peut faire apparaître un lien de causalité entre les variables qu'elle met en cause. Ce ou ces liens permettent de définir des relations de cause à effet entre un changement intervenu sur une variable et la méthode de résolution à appliquer dans ce cas. Dans le même registre, il est possible de donner un ordre préférentiel dans la définition des méthodes de résolution de la contrainte ;

- en dernier recours et dans le cas d'un système interactif, il est toujours possible de faire appel à l'utilisateur. Cette solution doit cependant n'être envisagée qu'à titre exceptionnel [SUS] car elle risque fort de lasser l'utilisateur par des appels incessants.

4.2.4) Les problèmes de circularité

Les problèmes de circularité dans la propagation des contraintes sont liés à la présence de boucles dans le réseau.

Prenons un exemple simple, tiré de l'étude des circuits électriques, et envisageons différents moyens de résoudre le problème : on désire modéliser le circuit suivant (caractérisé par des tensions et des courants), constitué de deux résistances mises en parallèle.



On obtient le réseau de contraintes représenté ci-après par la figure 4.3.

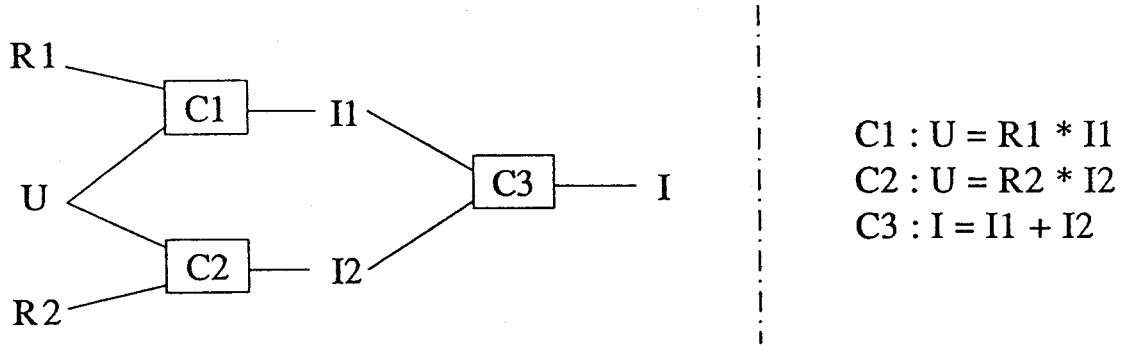


Figure 4.3

Initialement seules les valeurs de R_1, R_2 et i sont fixées. Toutes les variables indéterminées appartenant à la boucle, lors de la satisfaction initiale des contraintes, la propagation va échouer :

Au départ du cycle, les trois contraintes C_1, C_2, C_3 sont dans l'ensemble des contraintes à évaluer, elles seront examinées successivement mais ne pourront être résolues faute de données. A la fin du cycle, aucune variable supplémentaire n'aura été instanciée.

Dans le cas de contraintes algébriques, plusieurs solutions peuvent être proposées aux problèmes de circularité :

la propagation symbolique, utilisée par exemple dans Consat et ThingLab [GÜS][BOR]. Elle consiste à propager une expression symbolique, à travers le réseau, comme valeur provisoire d'une variable. Dans notre exemple, on affecte à la variable u la valeur symbolique *a*, ce qui permet de donner à i_1, i_2 et i, respectivement, les valeurs a/R_1, a/R_2 et $a/(R_1+R_2)$. La variable i étant instanciée, la valeur de *a* peut être calculée, les variables u ainsi que i_1 et i_2 pourront donc à leur tour être instanciées. La propagation n'échoue pas, toutes les variables du réseau sont affectées, le réseau est cohérent.

le principe des vues multiples, utilisé par exemple dans Constraints [STE]. Il consiste à décrire le sous réseau que forme la boucle de contraintes par une nouvelle contrainte, redondante vis à vis des contraintes formant la boucle. On obtient alors un réseau équivalent, pour lequel la satisfaction des contraintes ne pose pas de problème de circularité. Dans l'exemple cité, définir une nouvelle vue sur le réseau, revient à donner la contrainte de définition d'une résistance

équivalente à deux résistances mises en parallèle.
Pour obtenir cette nouvelle vue sur le réseau, il faudra soit faire appel à l'utilisateur, soit intégrer dans le mécanisme de résolution des contraintes, des lois de composition lui permettant définir une nouvelle contrainte à partir de contraintes existantes.

la propagation du prédicat, cette méthode découle des deux méthodes précédentes, à la différence prés qu'elle ne propage plus, comme la première méthode, une valeur symbolique, mais une expression. Ainsi, si on affecte à i_1 l'expression 'u*R_1' et à i_2 l'expression 'u*R_2', on obtient alors i= u*R_1 + u*R_2, ce qui nous ramène à la contrainte redondante proposée par la seconde méthode. On en déduit les valeurs de u, i_1 et i_2. Cette méthode revient à admettre la satisfaction différée d'une contrainte, qui se trouve propagée le long du réseau de contraintes.

la relaxation qui consiste à choisir des valeurs initiales pour les variables appartenant à la boucle de contraintes. Par itération, c'est valeurs sont ajustées de façon à satisfaire les contraintes de la boucle du mieux possible. La relaxation n'est pas une technique de résolue rapide, aussi, dans de nombreux systèmes, elle n'est utilisée qu'en dernier recours.

4.3 RESEAUX D'OBJETS CONTRAINTS ET RESEAUX DE CONTRAINTES

Un réseau d'objets contraints est un ensemble d'objets sur lesquels sont définies deux types de contraintes : les contraintes établissant une relation sur les variables d'instance d'un objet en particulier (contraintes propres à l'objet) et les contraintes établissant une relation sur des variables d'instances appartenant à différents objets (contraintes de composition d'objets).

Si on se rapporte à la définition d'un réseau de contraintes donnée au chapitre 3 (§ 3.1.2.4), un réseau d'objets contraints peut être représenté par un graphe :

- chaque noeud du graphe correspond à un objet contraint ;

- les contraintes propres à un objet sont représentées par une simple boucle sur le noeud correspondant à l'objet et les contraintes de composition par des arcs entre plusieurs noeuds.

La figure 4.5 illustre le réseau d'objets contraints créé par la construction d'une ligne horizontale à point milieu. Cet objet possède deux composantes, un point et une ligne_horizontale. La contrainte de point-milieu est définie entre les deux composantes alors que la contrainte d'horizontalité n'est définie que sur une seule composante.

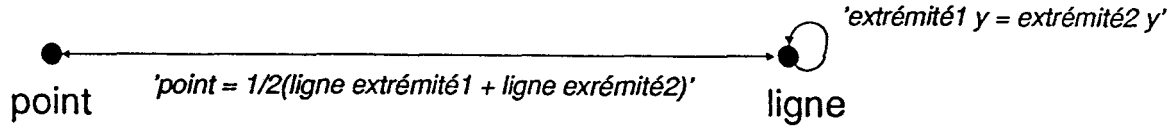


Figure 4.5

Un objet étant décrit par l'ensemble de ses variables d'instance, il existe donc implicitement, derrière le réseau d'objets contraints défini précédemment, un réseau de contraintes plus fin qui peut être représenté par un graphe étiqueté :

- chaque noeud du graphe correspond à une variable d'instance, il est étiqueté du nom donné à cette variable dans la structure de l'objet contraint ;

- les contraintes sont représentées par des arcs sur les noeuds, une contrainte définie sur une unique variable se représentant par une simple boucle.

Le réseau de contraintes, illustré figure 4.6, est une représentation plus fine du réseau d'objets contraints donné par la figure 4.5.

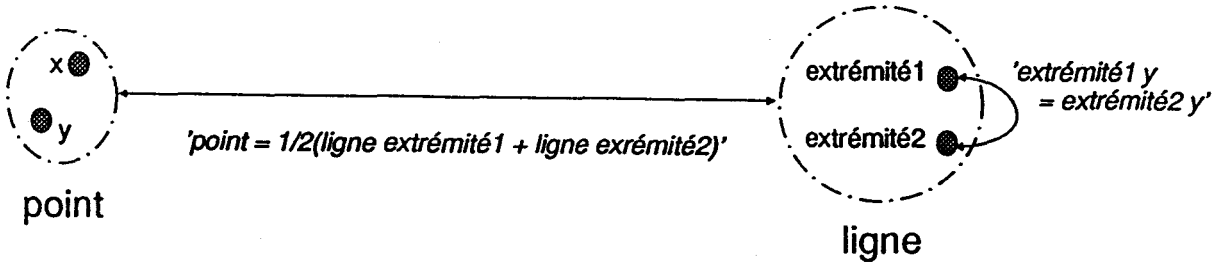


Figure 4.6

Si on considère qu'une variable d'instance pointe elle aussi sur un objet qui, s'il n'est pas primitif, possède également des variables d'instance, il est possible d'obtenir une représentation de plus en plus détaillée du réseau d'objets contraints (figure 4.7).

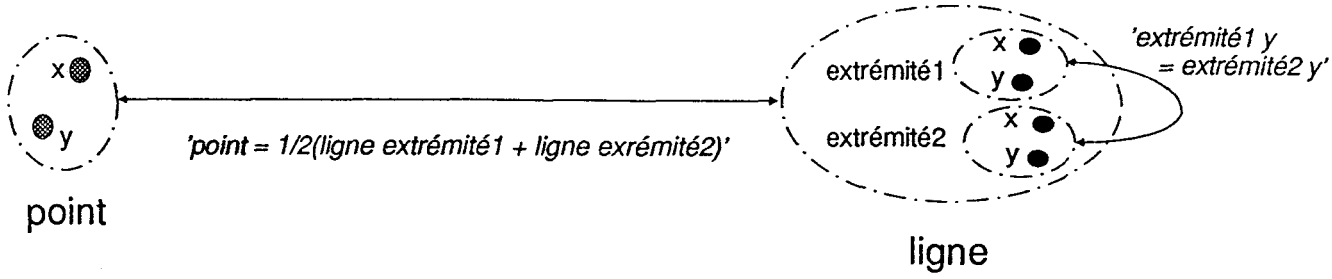


Figure 4.7

4.4. SATISFACTION DES CONTRAINTES DEFINIES SUR UN RESEAU D'OBJETS CONTRAINTS

4.4.1) Etat de l'Art

Peu de travaux abordent la résolution des contraintes pour un réseau d'objets contraints. Les résultats les plus significatifs en ce domaine concernent le langage de simulation ThingLab [BOR], dont le mécanisme est repris par de nombreux systèmes comme outil de définition et de résolution de contraintes.

Le mécanisme interne de satisfaction des contraintes implanté dans ThingLab est construit sur le principe de propagation présenté au paragraphe 2.2.3, guidant l'instanciation par l'évaluation, rendu plus performant par quelques améliorations :

- Les contraintes possèdent un degré de priorité en plus d'une règle et d'un ensemble de méthodes de résolution. Quand un objet contraint subit une modification, les contraintes mises en cause par ce changement sont ordonnées, suivant leur priorité, dans un ensemble. Les contraintes de plus forte priorité sont les premières évaluées, puis les contraintes de priorité inférieure, et ainsi de suite jusqu'à que l'ensemble des contraintes à évaluer soit vide. Lors du cycle de résolution, si une contrainte de faible priorité ne peut être satisfaite compte tenu des instanciations effectuées, elle sera délaissé pour ne pas remettre en cause les contraintes de priorités plus élevées déjà résolues ni faire appel à l'utilisateur.

- ThingLab est un outil particulièrement destiné à la gestion des interfaces graphiques homme-machine. Les contraintes permettront par exemple à la représentation graphique d'un objet de suivre le déplacement de la souris. Il est donc nécessaire que la résolution des contraintes soit la plus rapide possible. Pour ce faire, le processus de satisfaction des contraintes se déroule en deux phases distinctes, la planification et l'évaluation.

Dans la phase de planification, aucune instanciation n'est effectuée. Le mécanisme interne de satisfaction des contraintes scrute l'ensemble des contraintes à évaluer et recherche la séquence optimale des méthodes de résolution à appliquer pour satisfaire toutes les contraintes (ou s'il y a conflit le plus grand nombre d'entre elles). Quand il a trouver cette séquence, il la compile dans une méthode appelée plan de résolution.

Chaque fois qu'un même changement intervient sur le réseau d'objets contraints, le mécanisme interne fait appel au plan de résolution correspondant et instancie alors toutes les variables du réseau. C'est la phase d'évaluation.

La phase de planification peut être vue comme une phase d'apprentissage. Lorsqu'une nouvelle modification est faite sur les objets du réseau, il n'existe aucun plan de résolution correspondant à ce changement. Le système est alors obligé de planifier sa résolution, ce qui nécessite un temps de calcul assez important : plus les contraintes et les objets sont nombreux, plus la séquence de résolution sera longue et plus les combinaisons de méthodes résolution sont nombreuses (explosion combinatoire).

ThingLab ne permet que la définition de contraintes de classe, ce qui limite les possibilités de composition d'objets et donc rend difficile la modélisation de systèmes complexes tels que les processus industriels. De plus la simulation de tels processus nécessite de pouvoir créer de contraintes dynamiques sur les objets du système, ce qui n'est pas possible avec ce langage de simulation.

Une description plus détaillée de ThingLab est donnée en annexe_1.

4.4.2) Propagation des contraintes dans un réseau d'objets contraints

L'outil de définition et de satisfaction des contraintes que nous voulons mettre en place doit répondre aux impératifs suivants :

- (1) Il doit pouvoir être utilisé quelque soit le domaine d'application. Le mécanisme interne de satisfaction des contraintes doit donc être indépendant du type de contraintes utilisées.

- (2) Il doit pouvoir s'adapter sur une application déjà existante.

- (3) L'utilisateur doit pouvoir, indifféremment, définir les objets de son application en programmation orientée objet classique ou en programmation orientée objet avec contraintes. D'autre part, son application doit pouvoir inclure à la fois des objets contraints et des objets non contraints.

- (4) Le mécanisme de satisfaction des contraintes doit être transparent pour l'utilisateur.

- (5) Le mécanisme de satisfaction des contraintes doit être transparent pour tout autre mécanisme interne déjà implanté dans l'application.

La structure de définition des contraintes, décrite au chapitre 3 (§ 3.2), permet d'exprimer n'importe quel prédicat et la contrainte ainsi définie peut être associée soit à une classe soit à une instance (CF. Chap. 3, § 3.3). Si, pour sa part, le mécanisme interne de satisfaction des contraintes ne tient aucun compte de la nature des contraintes qu'il manipule, l'outil résultant sera alors indépendant du domaine d'application et du type de contraintes utilisées.

Une contrainte peut être établie sur une classe (respectivement sur une instance), nouvellement créée ou déjà définies dans le système. En effet, la modification apportée à la structure de définition d'une classe, pour inclure la contrainte de classe, peut être supportée par une classe déjà existante et il suffit, pour contraindre une instance, d'ajouter l'association 'objet → contrainte' dans le dictionnaire des instances contraintes, ce qui peut être fait pour tout objet. Il suffit donc, pour que l'outil créé puisse s'adapté sur une application déjà existante, que le mécanisme de résolution des contraintes soit indépendant des objets de l'application, en d'autres termes que le code relatif au processus de résolution des contraintes soit séparé du code de l'application.

4.4.2.1) Appel au processus de satisfaction des contraintes

Un réseau d'objets contraints est considéré comme stable si toutes les contraintes définies sur les objets du réseau sont satisfaites. Si un objet contraint est modifié, qu'il soit objet simple ou composante, une des contraintes du réseau peut ne plus être vérifiée, le réseau est donc rendu instable. Dans ce cas, il faut que le processus de

satisfaction des contraintes rétablisse, automatiquement et de façon transparente pour l'utilisateur, la stabilité du réseau de contraintes.

Une première solution consiste à glisser un message d'appel au mécanisme de satisfaction des contraintes dans chacune des méthodes comprises par l'objet contraint et modifiant celui-ci. C'est la technique dite du "trigger", utilisée par exemple pour la gestion des relations de dépendance : la mise à jour des objets dépendants est déclenchée par le message *'update: anAspectSymbol'*, ajouté au code de toutes les méthodes modifiant l'objet maître.

Cette solution ne permet malheureusement pas d'intégrer facilement les contraintes dans une application déjà existante. En effet, elle suppose un travail de mise à jour important, l'utilisateur devant retrouver toutes les méthodes susceptibles de modifier l'objet contraint.

Le problème posé est de lancer un appel au mécanisme de satisfaction des contraintes, chaque fois qu'un objet contraint est modifié, de façon systématique et, si possible, sans avoir recours à l'intervention de l'utilisateur.

Modifier un objet, c'est modifier la valeur d'une de ses variables d'instance, il suffit donc, chaque fois qu'une variable d'instance d'un objet contraint est affectée, de vérifier si toutes les contraintes définies, directement ou indirectement, sur cet objet sont toujours vérifiées. Si une seule de ces contraintes n'est pas satisfaite, la procédure de satisfaction des contraintes est déclenchée.

4.4.2.2) Déclenchement du test de vérification des contraintes par attachement procédural

L'attachement procédural nous permet d'imposer, à tout objet contraint, un comportement spécifique lors de l'affectation de ses variables d'instance.

Dans le cas d'un objet contraint, le message d'affectation d'une variable d'instance a été 'détourné', de sorte qu'à chaque affectation, un appel à la procédure de satisfaction des contraintes est lancé. Si une seule des contraintes définies sur l'objet contraint n'est pas vérifiée, la procédure de satisfaction des contraintes est appliquée.

Contrairement à la technique du "trigger", l'attachement procédural n'impose à l'utilisateur aucun travail de mise à jour des méthodes déjà définies dans son

application. Seul le message permettant l'affectation des variables d'instance est modifié puis l'application est immédiatement recompilée pour tenir compte de la modification intervenue.

Implantation en Smalltalk-80

En Smalltalk-80 le message d'affectation est défini soit par le symbole '←', soit par le symbole ':='. Lorsqu'à l'analyse syntaxique d'un message, le compilateur rencontre l'un de ces deux symboles, il le remplace par le message d'affectation.

Les méthodes d'analyse syntaxique définies dans la classe Parser ont donc été modifiées pour remplacer, dans le cas d'un objet contraint, l'affectation classique par le message *'aConstrainObject instanceVarAt:anInteger put:anObject'*, où *anInteger* donne l'indexe, dans la structure interne de l'objet contraint, de la variable d'instance à affecter et *anObject* donne la valeur affectée à la variable d'instance. En plus de l'affectation proprement dite, cette méthode envoie à l'objet contraint le message, *'contrainte: anAspectSymbol'*, lui demandant de vérifier s'il respecte toujours l'ensemble des contraintes l'affectant.

4.4.2.3) La procédure de satisfaction des contraintes

C'est à l'objet contraint, lorsqu'il a été modifié, de vérifier s'il répond toujours à l'ensemble des contraintes l'affectant. Si tel n'est pas le cas, c'est à lui de satisfaire l'ensemble des contraintes définies sur le réseau d'objets contraints dans lequel il s'insère. Par contre, c'est la contrainte qui vérifie si son prédicat est vrai et déclenche ses propres méthodes de résolution.

1) Pour vérifier s'il répond à toutes ses contraintes, l'objet contraint regarde d'abord s'il est soumis à une contrainte de classe. Si oui, il demande à cette contrainte de tester si son prédicat est vrai pour les valeurs attribuées à ses variables d'instance. Si la réponse est négative, il demande à la contrainte de se résoudre en appliquant l'une de ses méthodes de résolution. Puis l'objet contraint regarde s'il est soumis à une contrainte d'instance intrinsèque. Si oui, il fait les mêmes démarches que pour une contrainte de classe.

Si on se réfère à la représentation du réseau d'objets contraints sous la forme d'un graphe, cette première étape dans la satisfaction des contraintes consiste à résoudre, en

premier lieu, les contraintes représentées par une simple boucle sur un noeud. La procédure de satisfaction des contraintes regarde donc localement, au niveau de l'objet contraint, s'il répond au moins à ses propres contraintes.

Les représentations de plus en plus fines d'un réseau d'objets contraints sous la forme de réseaux de contraintes (Cf. § 4.3) montrent que la résolution des contraintes propres, définies sur l'objet contraint, peut s'avérer fort complexe. En effet, les variables d'instance de l'objet contraint peuvent pointer sur des objets qui eux aussi ont leurs propres contraintes. Résoudre une contrainte définie sur un objet contraint revient à modifier une ou plusieurs de ses variables d'instance (en appliquant une des méthodes de résolution de la contrainte) et donc, éventuellement, à modifier un nouvel objet contraint. Les contraintes définies sur cet objet devront à leur tour être vérifiées et au besoin satisfaites.

2) Puis l'objet contraint regarde s'il est composante d'un objet plus complexe. En fait, il regarde s'il partage une contrainte avec d'autres objets contraints. Si oui, il demande à chacun de ces objets de vérifier et au besoin de satisfaire à son tour toutes ses contraintes.

Si on se réfère de nouveau à la représentation sous la forme d'un graphe du réseau d'objets contraints, cette seconde phase propage la résolution des contraintes à tous les noeuds voisins du noeud représentant l'objet contraint (deux noeuds sont voisins s'ils sont reliés par une contrainte).

On retrouve donc dans la procédure de satisfaction des contraintes décrite ci-dessus les notions de propagation locale et de propagation globale définies au paragraphe 4.2.1 . Cette procédure guide l'évaluation des contraintes du réseau par l'instanciation des variables d'instance, contrairement aux procédures de satisfaction des contraintes constructives décrites aux paragraphes 2.2.2 et 2.2.3 .

Quand le réseau d'objets contraints est complexe, c'est à dire s'il existe de nombreuses contraintes de composition, la résolution globale du réseau est relativement lente car elle vérifie les contraintes de manière descendante, de l'objet composé vers ses autres composantes et de manière ascendante, de la composante vers l'objet composé, (Cf. Chap. 2, § 2.2.3). Toutefois pour améliorer les performances du processus de résolution, il est possible, comme dans ThingLab, d'établir des plans de résolution compilés. Chaque fois qu'une variable d'instance donnée sera affectée, la méthode compilée correspondante sera exécutée. Le processus de résolution des contraintes ne

sera alors lancé que dans trois cas : pour une première affectation de la variable d'instance, lorsqu'une contrainte est ajoutée ou retirée du réseau, lorsqu'un objet contraint est ajouté ou retiré du réseau.

Implantation en Smalltalk-80

Tout objet contraint comprend la méthode *'constraint: anAspectSymbol'* qui lance la procédure de résolution des contraintes établies sur l'objet (vérification et satisfaction). L'argument *anAspectSymbol* désigne le type de changement qu'a subi l'objet contraint.

```
constraint: anAspectSymbol
|père|
ResolutionContrainteEnCour := true.
self class contrainte isNil
    ifFalse: [self class contrainte model: self.
        self class contrainte satisfaite ifFalse: [self class contrainte satisfaire:
anAspectSymbol]].
self ownContraint isNil
    ifFalse: [self ownContraint model: self.
        self satisfaite ifFalse: [self ownContraint satisfaire: anAspectSymbol]].
père := self constraintOwner
père notNil ifTrue: [père do: [:anelem | anelem contrainte]
ResolutionContrainteEnCour := true.
í self
```

Les méthodes *'satisfaite'* et *'satisfaire: anAspectSymbol'* ont été définies dans la classe abstraite Contrainte, super-classe de toutes classes de contraintes (Cf Chap. 3, § 3.2.6.4). La première méthode teste si le prédicat de la contrainte est vrai pour les valeurs affectées aux variables d'instance de l'objet contraint. La seconde méthode résout la contrainte en appliquant l'une de ses méthodes de résolution. Le choix de cette méthode est fonction de l'argument *anAspectSymbol*, qui désigne la nature du changement subi

par l'objet contraint, et de la préférence définie sur telle ou telle autre méthode de résolution (Cf. Chap. 3, § 3.2.3.2).

CONCLUSION

Ce chapitre décrit différentes techniques de satisfaction des contraintes. Il montre la progression des recherches effectuées dans ce domaine, partant de la résolution des contraintes non-constructives jusqu'à la satisfaction dynamique des contraintes constructives.

Plusieurs applications, basées sur les diverses méthodologies exposées ici , ont abouti à des systèmes de résolution de contraintes tels Constraints [SUS][STE], Consat [GÜS], ou ThingLab [BOR]. Une synthèse bibliographique de tous ces travaux est donnée dans l'annexe_2.

Ce chapitre présente également une nouvelle approche pour la satisfaction des contraintes constructives, sur laquelle se base le mécanisme interne de satisfaction des contraintes implanté dans notre plate-forme de simulation. Cette approche, calquée sur une technique déjà existante de satisfaction des contraintes non_constructives, guide la résolution des contraintes par l'instanciation des variables de l'objet contraint. Elle utilise la propagation locale pour résoudre les contraintes intrinsèques définies sur le réseau d'objets contraints et la propagation globale pour résoudre les contraintes extrinsèques.

CHAPITRE 5 : DOMAINES D'APPLICATION

CHAPITRE 5 : DOMAINES D'APPLICATION

Ce chapitre décrit l'utilisation des objets contraints dans trois domaines d'application : la construction orientée objet des interfaces utilisateurs, la simulation interactive des systèmes de production industriels et l'animation d'algorithmes.

Dans la construction des interfaces homme-machine les contraintes peuvent être utilisées pour décrire la structure d'une fenêtre et l'agencement des différentes fenêtres dans l'interface. Elles peuvent aussi, et surtout, être utilisées pour maintenir, en permanence, une parfaite cohérence entre une représentation-écran et son modèle interne.

Dans la simulation interactive des systèmes de production, les contraintes sont utilisées pour décrire le comportement opératif des objets technologiques (contraintes intrinsèques établies sur l'objet) et la composition de ces objets dans le système de production (contraintes extrinsèques établies entre objets technologiques). La satisfaction de l'ensemble des contraintes établies sur le réseau permet alors de déduire, des comportements opératifs propres aux objets technologiques et de leur composition, le comportement opératif global du système de production.

Enfin, l'utilisation des contraintes pour l'animation des algorithmes permet essentiellement de maintenir la cohérence entre les données internes manipulées par l'algorithme et leurs représentations graphiques.

5.1. UTILISATION DES CONTRAINTES POUR LA CONSTRUCTION ORIENTEE OBJET DES INTERFACES UTILISATEURS

La construction d'interfaces interactives représente, très souvent, un travail de longue haleine. Elle requiert l'écriture d'une quantité imposante d'instructions relatives à la gestion des interfaces d'entrée (clavier, souris...), des fenêtres, des menus (affichage, prise en compte du choix de l'utilisateur...) ainsi que de codes relatifs au maintien d'une parfaite cohérence entre la représentation-écran et la donnée interne représentée.
D'autre part, pour certaines applications, si la possibilité d'intégrer de l'animation (voire une simulation) apporte un atout supplémentaire à l'outil utilisé, elle ajoute à la complexité de conception de l'interface.

L'utilisation d'un langage de programmation orientée objet permet tout d'abord une décomposition de l'interface en éléments simples. Tout objet se composant d'une structure de donnée et d'un ensemble de méthodes, chaque élément de l'interface trouvera donc, dans sa propre définition, toutes les informations nécessaires à sa représentation dans l'interface et à son comportement en tant que donnée interne.

De fait, une majeure partie des instructions entrant dans la création d'une interface est consacrée à la définition et au maintien d'une multitude de relations établies sur ces éléments, ou entre ces éléments. Pour cette raison, l'introduction des contraintes dans le langage de programmation va faciliter la construction de l'interface et la rendre plus accessible : une contrainte est une relation qui doit être maintenue mais maintenir cette relation est le rôle du système et non plus celui de l'utilisateur.

Les contraintes peuvent être employées à différents niveaux dans la construction des interfaces homme-machine, notamment :

- pour assurer une parfaite cohérence entre une donnée interne et ses représentations graphiques. Si le système utilise et modifie une donnée, toutes les représentations graphiques de cette donnée sont instantanément remises à jour. Inversement, si l'utilisateur agit sur une représentation graphique au moyen d'une des interfaces d'entrée, son modèle et les autres représentations graphiques du même modèle doivent tenir compte immédiatement du changement intervenu ;

- pour configurer l'interface : positionnement et agencement des fenêtres, contraintes de présence à l'écran et conditions de chevauchement ... ;

- pour la gestion des interfaces d'entrée (par exemple le suivi d'une trajectoire décrite à la souris).

En Smalltalk-80, l'interaction système-utilisateur est gérée par le mécanisme M.V.C, Modèle-Vue-Contrôleur (Cf. Chap 1, § 1.2.1). Toutes les fenêtres, lieux de l'écran où se déroule l'interaction, sont bâties autour de cette triade.

Le mécanisme M.V.C se base sur la relation de dépendance entre le modèle et la vue pour assurer une parfaite cohérence entre la représentation graphique et l'objet qu'elle représente : lorsque le modèle est modifié, il signale à tous ses dépendants, donc à toutes les vues qui lui sont associées, qu'il a changé en s'envoyant le message *changed* qui répercute le changement sur tous les dépendants en leur envoyant le message *update*.
Cette démarche est astreignante pour l'utilisateur qui ne sait pas toujours très exactement dans quelle méthode il doit signaler un changement sur le modèle ni quels paramètres passer pour caractériser ce changement.

Si une contrainte est utilisée pour définir la relation modèle-vue, le système se chargera, sans qu'il le lui soit explicitement demandé, de rafraîchir la vue si le modèle est modifié et inversement. En effet, la contrainte étant multidirectionnelle, elle permet une mise à jour dans les deux sens, de la vue par rapport au modèle et du modèle par rapport à la vue, ce que ne permet pas la relation de dépendance (à moins de créer une double dépendance).

En schématisant, l'utilisateur a besoin pour décrire l'interaction modèle-vue, de définir deux méthodes de mise à jour, l'une dans la classe du modèle, l'autre dans celle de la vue. Pour reprendre les termes employées dans la structure M.V.C., il définit la méthode *'update: anAspectSymbol'* à la fois pour le modèle et pour la vue. Il lui faut ensuite, établir une contrainte entre le modèle et chacune de ses vues. De façon générale, si on considère qu'une remise à jour est effectuée quelque soit la modification survenue sur le modèle ou sur une de ses représentations, la contrainte entre ces objets peut se définir de la sorte :

```
contrainte :
        règle : 'false'
        méthodes : 'model update: anAspectSymbol.
                        view update: anAspectSymbol'.
```

L'appel au mécanisme de résolution des contraintes est lancé chaque fois qu'un objet contraint est modifié.

Supposons, dans un premier temps, que ce soit le modèle qui ait changé. L'évaluation du prédicat de la contrainte renvoie 'false', ce qui indique que celle-ci n'est pas vérifiée. La contrainte doit donc être résolue en appliquant une de ses méthodes de résolution. Puisque le changement survenu concerne le modèle, c'est la deuxième méthode de résolution qui sera choisie, la vue sera donc bien remise à jour.

Supposons, maintenant, que ce soit la vue qui ait subit une modification. L'évaluation du prédicat de la contrainte renvoie 'false', ce qui montre que la contrainte n'est pas vérifiée. De nouveau, la contrainte est résolue en appliquant une de ses méthodes de résolution. Le changement concernant , cette fois-ci, la vue et non plus le modèle, c'est la première méthode de résolution qui sera choisie. Le modèle sera donc remis à jour.

Si l'utilisateur veut, pour son application, mettre une condition à la mise à jour du modèle ou de la vue, il suffit de changer le prédicat de la contrainte pour que celui-ci ne soit pas toujours faux.

Outre le fait de maintenir une parfaite cohérence entre le modèle et la vue, les contraintes peuvent également être utilisées pour définir la structure d'une fenêtre (contraintes intrinsèques établies sur la vue) ou l'agencement d'un ensemble de fenêtres (contraintes extrinsèques établies entre plusieurs vues) [EPS] [COH].

Les variables propres à la vue concernent en générale ses dimensions (hauteur et largeur), sa position (angle supérieur gauche ou inférieur droit), ses proportions, sa zone d'interactivité et son activité. Sur chacune de ses variables, il est possible de définir des contraintes, que ce soit pour :

- définir une taille minimale, '*hauteur* > *limite1 et largeur* > *limite2*';
- maintenir une proportion, '*hauteur* = f^{θ} *(largeur)*' ;
- réglementer le positionnement d'une fenêtre pour qu'elle apparaisse toujours dans une zone précise de l'écran ;
- définir une zone d'interactivité en dehors de laquelle l'intervention d'un utilisateur ne pourra être prise en compte ;
- imposer des conditions à l'ouverture ou à la fermeture de la fenêtre ;

...

Les contraintes extrinsèques établies sur plusieurs fenêtres sont essentiellement utilisées pour décrire leur agencement dans l'interface :

- définir une loi sur les proportions relatives de plusieurs fenêtres ;
- imposer une contrainte d'alignement ou d'adjacence sur plusieurs fenêtres ;
- définir une loi de "co-présence" de plusieurs fenêtres à l'écran ;
- ...

En Smalltalk-80 par exemple, les fenêtres ouvertes sur l'écran sont totalement indépendantes. Seule existe une notion de dépendance entre une super-vue et les sous-vues constituant une fenêtre. L'agencement des sous-vues dans la super-vue (proportions et positions) est fixé, de façon définitive, dans la méthode de création de la fenêtre. Une fois la fenêtre ouverte, l'utilisateur n'a la possibilité d'agir que sur la super-vue. S'il la déforme ou la déplace, les coordonnées et la position des sous-vues sont recalculées, puis les vues affichées. En aucun cas, l'utilisateur ne peut déformer ou déplacer une sous-vue à l'intérieur même de la super-vue.

Au contraire, si l'agencement des vues dans une même fenêtre est décrit par des contraintes extrinsèques sur les vues, cela supprime les notions de super-vue et de sous-vue. Toutes les vues d'une fenêtre sont définies au même niveau et peuvent donc être manipulées indépendamment par l'utilisateur, les contraintes d'agencement maintenant en permanence une cohérence au niveau de la fenêtre.

A l'heure actuelle, nous n'avons pas encore développé d'interface utilisant la notion de contrainte. Cependant, un des objectifs de notre équipe est de permettre une simulation "distribuée" sur plusieurs postes de travail, avec des autorisations d'accès à la plate-forme de simulation définies à différents niveaux. Par exemple dans le cas de l'apprentissage de la conduite des processus industriels par simulation, l'élève n'aura pas les mêmes droits d'accès, ni n'accédera aux même fenêtres, que l'instructeur. Dans ce cas, les contraintes vont nous permettre de définir les droits d'accès de chacun des utilisateurs aux différentes fenêtres de la simulation, elles vont également permettre de répercuter (éventuellement de ne pas répercuter) les opérations de mises à jour d'un modèle ou de ses représentations sur tous les postes de simulation.

5.2. Utilisation des contraintes dans la simulation interactive et orientée objet des systèmes de production

La programmation orientée objet pour la modélisation et la simulation des systèmes complexes, tels les processus industriels, présente plusieurs avantages :

- Elle fournit une structure représentation, précise et naturelle, du domaine exploité. Dans cette structure, les concepts de classe et d'héritage permettent de regrouper, dans une description hiérarchique, les objets à comportement identique ou voisin.
Par exemple, le domaine 'processus par lots' contiendra des vannes (manuelles, hydrauliques, pneumatiques, ...), des moteurs (pompes, malaxeurs, ventilateurs), des capteurs, des réservoirs ...; tandis que le domaine 'robot' contiendra des bras, des pinces, des capteurs, des moteurs (pas à pas, continu) ...

- Elle assure une réutilisabilité des données, de plus, le fait que les objets soient indépendants permet une programmation découpée (éventuellement confiée à plusieurs individus).

- Elle permet les évolutions et les modifications d'une application sans remettre en cause la validité du système global.

Partant de ce constat, notre équipe a développé, une première plate-forme de simulation orientée objet des processus par lots [CAN]. Cet outil est basé sur une décomposition en objets des systèmes de production dans le but de simuler leur comportement opératif global. Ces recherches ont souligné la difficulté que rencontre l'utilisateur à définir, précisément et simplement, les relations régissant les différents objets d'un système. Elles ont surtout mise en évidence la difficulté à déduire un comportement opératif global pour un système, partant des comportements opératifs individuels de chacun des objets qui le composent.

Ce paragraphe décrit dans quelles mesures les contraintes peuvent être intégrées à la plate-forme de simulation déjà existante.

5.2.1) Les contraintes dans la modélisation orientée objet des systèmes de production

Décrire un système de production revient à décrire la partie commande de ce système et son comportement opératif suite à l'exécution d'une commande. Dans la

modélisation orientée objet des systèmes de production décrite par [CAN], on retrouve trois types d'objets :

- les objets-technologiques qui modélisent les différents éléments constituant le système. Ces objets sont répertoriés selon deux catégories, les objets d'action et les objets de compte-rendu. Les objets d'action influencent les flux de matières et contribuent à la transformation des produits. On retrouve dans cette catégorie d'objets, les vannes, les malaxeurs, les pompes ... Les objets de compte-rendu caractérisent les grandeurs relatives à la matière d'oeuvre ou rendent compte tout simplement d'une action. On retrouve dans cette catégorie d'objets tous les capteurs de l'installation ;

- les objets-missions qui permettent l'exécution d'une tâche demandée par la partie commande ;

- les objets opératifs qui décrivent le comportement de la matière d'oeuvre, tenant compte pour cela, de l'état des objets technologiques.

Cette modélisation entraîne obligatoirement une dualité des décompositions en objets-missions et objets-opératifs. Prenons l'exemple très simple, d'un petit système constitué d'une vanne et de deux réservoirs (figure 5.1).

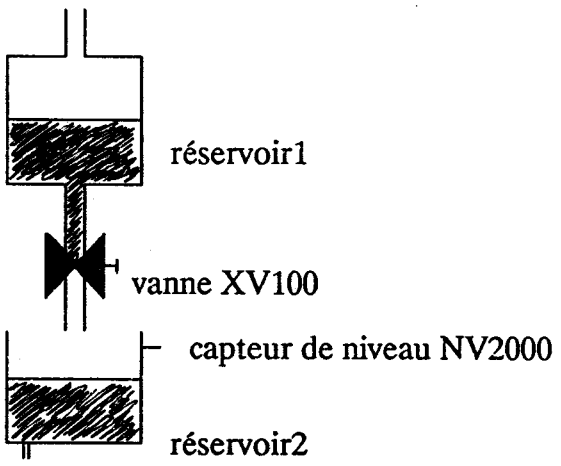


Figure 5.1

La modélisation orientée objet de ce système est donnée par le diagramme présenté figure 5.2.

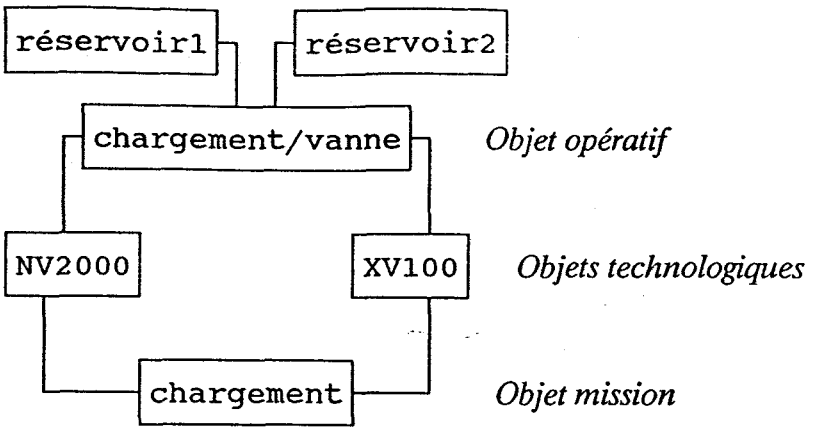


Figure 5.2

Ce diagramme montre bien la symétrie des décompositions en objets missions et en objets opératifs. Cette dualité entre les deux décompositions est essentiellement due à l'impossibilité de définir des relations entre objets technologiques. Le comportement opératif global du système ne peut donc se déduire directement du comportement opératif local propres à chacun des objets technologiques et de leur composition.

Nous avons vu dans les chapitres précédents qu'une contrainte peut être utilisée pour définir les lois régissant la composition d'objets ou, plus simplement, pour décrire le comportement individuel d'un objet. Définir des contraintes sur l'objet technologique va donc nous permettre, tout d'abord, de décrire de façon déclarative le fonctionnement opératif de cet objet. Ensuite, définir des contraintes entre objets technologiques va nous permettre de modéliser les interactions entre ces différents objets et suppléer ainsi au rôle de l'objet opératif.

5.2.2) Utilisation des contraintes pour décrire le comportement opératif d'un objet technologique

Le comportement opératif d'un objet technologique est souvent décrit par une loi physique définie sur ses variables d'état. Par exemple, le débit d'une vanne est fonction de son degré d'ouverture, le niveau de remplissage d'une cuve est fonction des débits d'entée et de sortie ...

Ces lois peuvent très facilement être décrites par des contraintes intrinsèques établies sur les variables d'instance de l'objet technologique. Ainsi, si la variable d'instance représentant le degré d'ouverture de l'objet vanne est modifiée, le mécanisme

de satisfaction des contraintes rétablira systématiquement la valeur de la variable débit.

5.2.3) Utilisation des contraintes pour modéliser les interactions entre objets technologiques

Comme nous le savons déjà, la connaissance de tous les objets technologiques n'est pas suffisante pour modéliser et simuler le comportement global d'un système de production. Encore faut-il modéliser la composition de ces objets et leurs interactions. Alors seulement, il est possible de déduire le comportement du système de production, à partir de l'état et du comportement de tous les objets technologiques le constituant.

En Smalltalk-80, comme dans d'autres langages de programmation, il existe déjà différents moyens pour associer des objets :

- les faire communiquer par l'envoi de messages ;
- définir une relation de dépendance entre ces objets ;
- affecter un objet à la variable d'instance d'un autre objet.

Il apparaît vite, très ardu de modéliser le comportement opératif global d'un système de production uniquement par l'envoi de messages, ou la déclaration de dépendances, entre objets technologiques. D'autre part, nous avons vu au chapitre 1 (§ 1.1.5) que l'utilisation des variables d'instance n'est pas à encourager lorsque l'on modélise un système complexe.

Par contre, l'utilisation des contraintes pour décrire la composition d'objets technologiques se fait simplement en déclarant une relation sur leurs variables d'instance respectives. Dans l'exemple illustré par la figure 5.1, il suffit de deux contraintes pour décrire la composition du système : une contrainte d'instance entre les objets technologiques réservoir1 et vanne-XV100 (liant le débit de sortie du réservoir au débit de la vanne) et une contrainte d'instance entre les objets technologiques vanne-XV100 et réservoir2 (liant le débit de la vanne au débit d'entrée du réservoir).

Le comportement opératif global du système de production se déduit alors, par résolution des contraintes, du comportement opératif de chacun des objets technologiques et de leurs interactions. Sur le système décrit par la figure 5.1, il sera défini, en tout et pour tout, six contraintes :

- deux contraintes extrinsèques, 'réservoir1 debitOut = vanne débit' et 'vanne débit = réservoir2 débitIn', pour définir la composition des objets

technologiques réservoir1, vanne et réservoir2 ;

- une contrainte extrinsèque 'réservoir2 niveau = capteur-NI2000 valeur' pour définir la composition de l'objet technologique réservoir2 et du capteur de niveau NI2000 ;

- la contrainte intrinsèque 'débit = $f\theta$(degré-d'ouverture)' définie sur l'objet vanne ;

- la contrainte intrinsèque 'niveau =fh(débitIn,débitOut)' définie pour chacun des réservoirs.

Quand la vanne est ouverte, le mécanisme de satisfaction des contraintes attribue une valeur à la variable d'instance **débit** de l'objet vanne. Cette variable d'instance étant soumise à d'autres contraintes, la valeur du débit sera propagée, puis attribuée à la variable d'instance **débitOut** de l'objet réservoir1 et à la variable d'instance **débitIn** de l'objet réservoir2. Pour satisfaire les contraintes intrinsèques définies sur ces deux objets, le mécanisme de satisfaction des contraintes recalculera, à chaque pas de simulation, la valeur à attribuer à leurs variables d'instance **niveau** respectives. Enfin, la contrainte de composition définie entre le second réservoir et le capteur de niveau entraînera une mise à jour systématique de la grandeur mesurée par le capteur.

5.2.4 Utilisation des contraintes pour construire l'interface de la simulation interactive

Dans un langage conventionnel, chaque fois qu'une donnée interne est modifiée, l'utilisateur est obligé de rappeler que sa ou ses représentations-graphiques doivent être rafraîchies. Inversement, si l'utilisateur modifie une des représentations-graphiques, il doit à chaque fois appeler une procédure de mise à jour de la donnée interne et de ses autres représentations. Ce qui, on le conçoit, augmente considérablement la quantité d'instructions relatives à la gestion des interfaces de simulation.

Dans un système axé sur les contraintes, l'utilisateur spécifie, de façon déclarative, la relation liant la donnée interne et chacune de ses représentations-graphiques, le système se chargeant, quant à lui, de maintenir ces relations : si la donnée est modifiée, ses représentations graphiques sont systématiquement mises à jour ; inversement, si l'une des représentations-graphiques est modifiée, le système répercute le changement sur la donnée elle même qui, se trouvant ainsi modifiée, entraîne à son tour la mise à jour de toutes ses autres représentations.

Dans la version actuelle de la plate-forme de simulation, les contraintes sont effectivement utilisées pour décrire le comportement individuel des objets technologiques. La composition d'objets est, pour l'instant, décrite par la définition de liens.

Un lien permet le partage d'une même valeur par des variables d'instance définies pour des objets technologiques différents. Par exemple, un lien établi entre une vanne et une cuve permettra à ces objets de partager la même valeur de débit, de même qu'un lien établi entre un réservoir et un capteur de niveau permettra à ces objets de partager la même grandeur. Lorsque la valeur d'un lien est modifiée, tous les objets partageant cette valeur sont appelés à vérifier leurs propres contraintes. Le lien représente donc une version simplifiée de la contrainte d'instance définie entre plusieurs objets. Cette version n'autorise que le partage d'une ou plusieurs valeurs par des variables d'instance définies pour des objets différents, elle ne permet pas de définir une relation plus complexes entre ces variables d'instance.

La résolution des contraintes, dans la plate-forme de simulation actuelle, se borne donc à la résolution des contraintes intrinsèques définies sur les objets technologiques et à la propagation des valeurs le long des liens établis entre plusieurs objets contraints. Toutefois, ces premiers essais nous ont permis de valider la méthodologie proposée pour l'introduction des contraintes dans la programmation orientée objet, ainsi que la méthode de satisfaction des contraintes décrite au chapitre 4 (§ 4.4.2).

5.3. UTILISATION DES CONTRAINTES POUR L'ANIMATION DES ALGORITHMES

L'utilisation des contraintes dans l'animation des algorithmes concerne, essentiellement, le maintien d'une parfaite cohérence entre la donnée interne manipulée par l'algorithme et ses représentations-graphiques.

Supposons que l'on ait à animer un algorithme de tri, sur les éléments entiers d'un tableau, décrit par la méthode suivante:

```
selectionSort
|t1 t2 t3 t4 t5 t6 t7|

1 to: self size - 1 do:
 [:t6|
 t1 := (self at: t6).
 t2 := t6.
 t6 + 1 to: self size do:
  [:t7|
  (self at: t7) < t1
   ifTrue:
   [t1 := (self at: t7).
   t2 := t7]].
 t6 = t2
  ifFalse:
  [t3 := (self at: 6).
  self at: t6 put: 0.
  self at: t6 put: (self at: t2).
  self at: t2 put: 0.
  self at: t2 put: t3.
  t3 := 0]]
```

Principe du tri :

Sélectionner le premier élément du tableau, le comparer à tous ses suivants dans le tableau de façon à trouver le minimum de tous les éléments qui lui sont inférieurs. S'il en existe un, le permuter avec le premier élément. Recommencer l'opération en sélectionnant le second élément, puis le troisième et ainsi de suite jusqu'à avoir ordonné la totalité du tableau.

Pour l'animation de cet algorithme de tri, on utilise deux représentations graphiques, une pour le tableau d'entiers et une pour la corbeille (figure 5.3) :

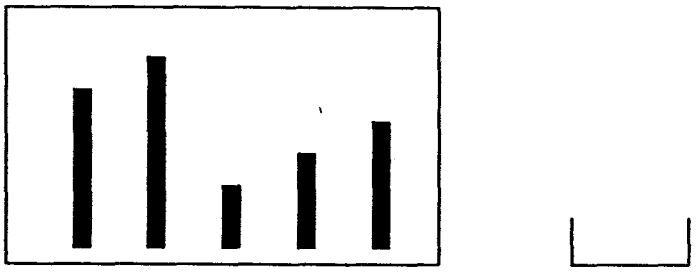


Figure 5.3

Une contrainte, définie entre le tableau et sa représentation graphique, assure que chacune des stries de la représentation est proportionnelle à l'entier qui lui correspond dans le tableau ; il en va de même pour la corbeille qui peut être considérée comme un tableau, à une dimension, initialement vide. Si un des éléments du tableau est modifié, la hauteur de la strie correspondante, dans la représentation graphique du tableau, est immédiatement réajustée ; inversement, si l'utilisateur modifie la hauteur d'une des stries, l'entier qui lui correspond dans le tableau est immédiatement recalculé.

Ainsi dans l'algorithme de tri, à chaque permutation de deux entiers dans le tableau, (le premier entier est mis dans la corbeille, le second prend la position du premier dans le tableau puis le contenu de la corbeille est mis à la position restée

vacante), les représentations graphiques du tableau et de la corbeille sont mises à jour plusieurs fois, de façon à toujours représenter l'état de leur modèle :

* *corbeille := self at: t6*
  dans la représentation graphique de la corbeille apparaît une strie dont la hauteur est proportionnelle à l'entier maintenant contenu dans la corbeille.

* *self at: t6 put: 0*
  la strie de rang t6 est effacée (remplacée par une strie de hauteur nulle) puisque l'élément correspondant dans le tableau est maintenant l'entier 0.

* *self at: t6 put: (self at: t2)*
  une strie, identique à la strie de rang t2, apparaît au rang t6 dans la représentation graphique du tableau

* *self at: t2 put: 0*
  la strie de rang t2 est effacée (remplacée par une strie de hauteur nulle) puisque l'élément correspondant dans le tableau est maintenant l'entier 0

* *self at: t2 put: corbeille*
  une strie, identique à celle contenue dans la représentation graphique de la corbeille, est dessinée au rang t2 dans la représentation graphique du tableau.

* *corbeille := 0*
  la corbeille est vide , sa représentation contient une strie de hauteur nulle.

Remarque : si l'utilisateur, ou l'"animateur", le désire, il peut également, comme dans Animus [DUI], piéger certains sélecteurs pour qu'ils déclenchent une série d'instructions graphiques simulant par exemple la trajectoire d'une strie vers une nouvelle position dans la représentation graphique du tableau ou vers la corbeille.

CONCLUSION

Ce chapitre expose les différentes possibilités d'utilisation des objets contraints dans trois domaines d'application, la construction des interfaces utilisateurs, la simulation interactive des systèmes de production industriels et l'animation des alogorithmes.

Cette liste n'est évidemment pas exhaustive, nous avons pour notre part envisager l'utilisation des contraintes dans ces trois domaines d'application car ils représentent les domaines d'activité et de recherches explorés par notre équipe.

CONCLUSION

CONCLUSION GENERALE

Le travail présenté dans ce mémoire souligne l'intérêt d'introduire les contraintes dans la programmation orientée objet.

L'utilisation des contraintes a essentiellement pour but de simplifier la tâche de l'utilisateur en lui permettant de décrire, de façon déclarative, une partie des relations définissant son application. Satisfaire toutes les contraintes établies sur un ensemble d'objets reste exclusivement la tâche d'un mécanisme interne de satisfaction des contraintes.

La première partie de ce travail présente les notions de contraintes et d'objets contraints.
Une contrainte, établie sur des objets, définit une relation sur leurs variables d'instance. On distingue deux types de contraintes : les contraintes intrinsèques définies sur les variables d'instance d'un même objet et les contraintes extrinsèques définies sur les variables d'instance de plusieurs objets.
Dans la représentation d'un réseau d'objets contraints, les contraintes intrinsèques sont décrites par des boucles sur le noeud représentant l'objet contraint et les contraintes extrinsèques sont décrites par des arcs entre plusieurs noeuds.

La deuxième partie de ce travail définit une méthodologie permettant d'introduire les contraintes dans les langages de programmation orientée objet, avec une implantation particulière en Smalltalk-80. La démarche que nous avons suivie permet d'introduire les contraintes dans une application orientée objet déjà existante. L'outil créé ne constitue donc pas, en lui-même, un langage de programmation, mais plutôt une extension d'un langage de programmation orientée objet qui permet indifféremment la définition et l'utilisation d'objets contraints et d'objets non contraints.

La méthode de satisfaction des contraintes, utilisée par le mécanisme interne de satisfaction des contraintes, est une nouvelle méthode, inspirée des techniques de propagation des contraintes non_constructives, qui guide la résolution des contraintes par l'évaluation des variables. Elle s'appuie sur une représentation plus fine du réseau d'objets contraints en réseaux de contraintes. La propagation des contraintes à travers

ce réseau d'objets contraints est à la fois une propagation descendante, le l'objet composé contraint vers ses composantes, et une propagation ascendante, de l'objet composante vers l'objet composé.

Enfin, la troisième partie traite de l'utilisation des objets contraints dans trois domaines d'application : la construction orientée objet des interfaces utilisateurs, la simulation interactive des processus industriels et l'animation d'algorithmes.

Les problèmes posés par l'introduction des contraintes dans la programmation orientée objet n'ont pas tous été abordés ici. Il reste à considérer la définition de contraintes temporelles et la résolution de ce type de contraintes.
D'autre part, certaines utilisations des contraintes dans la simulation des systèmes de production reste à valider, notre objectif étant de pouvoir, à plus ou moins long terme, utiliser les contraintes à différents niveaux dans la plate-forme de simulation :
- Pour décrire le comportement opératif propre à chacun des objets technologiques constituant le système.
- Pour décrire la composition des objets technologiques dans le système.
- Pour maintenir une parfaite cohérence entre l'objet technologique et ses représentations graphiques dans le synoptique de l'application.
- Pour gérer les droits d'accus aux différentes fenêtres de la simulation dans le cas d'une application distribuée sur plusieurs postes.
- Dans le cas d'une simulation distribuée, pour maintenir la cohérence entre les représentations d'un objet technologique dispersées sur les différents postes de la simulation. et l'objet lui-même.

Enfin, pour être complet, notre système devra inclure des mécanismes permettant d'accélérer le processus satisfaction des contraintes et de résoudre les cas de circularité par l'appel à un algorithme de relaxation.
La lenteur des mécanismes de satisfaction des contraintes est un problème délicat qui ne peut, apparemment, être résolue qu'en décomposant la satisfaction des contraintes en deux phases : une phase de planification (ou d'apprentissage) et une phase d'évaluation. Cette solution est toutefois peu envisageable dans le cas d'une simulation interactive des systèmes de production. En effet, dans la plate-forme de simulation décrite, la majorité des contraintes sont définies de façon dynamique, il n'y a donc pas, à proprement parler, de phase d'apprentissage. Il nous faudra donc trouver une autre solution à ce problème de rapidité dans la résolution des contraintes.

BIBLIOGRAPHIE

[ALM] J. Almarode

Rule-Based Delegation for Prototypes

OOPSLA'89 Proceedings, October 1989

[BAR] P.S. Barth

An object oriented approach to graphical interfaces

ACM Transactions on Graphics, V.5, N°2, April 1986.

[BER] P. Berlandier

Intégration d'outils pour l'expression et la satisfaction des contraintes dans un générateur de systèmes experts

INRIA, Rapport de Recherche n°924, Novembre 1988

[BLA] E. Blake, S. Cook

On including part hierarchies in object oriented languages with an implementation in Smalltalk

ECOOP'87, Paris, Juin 1987.

[BOO] G. Booch

Object Oriented Development

IEEE Transactions on Software Engineering, February 1986

[BOR] A.H. Borning

Classes versus prototypes in object oriented languages

ACM/IEEE Fall Joint Computer Conference, pp 36-40, Dallas, November 1986

A. Borning

The programming language aspect of ThingLab, a constraint oriented simulation laboratory

ACM Transactions on Programming languages and systems, V.3, N°4, October 1981.

A. Borning, R. Duisberg, B. Freeman-Benson, A. Kramer, M. Woolf
Constraint Hierarchies
OOPSLA '87, October 4-8, 1987

A.H. Borning
Constraints and functional programming
Sixth Annual International Phoenix Conference on Computers and Communications,pp 300-306, Scottsdale, AZ, February, 1987.

[CAN] E.Cantegrit
Modélisation et Simulation Orientées Objets des processus par lots
Th. de Doctorat, Université de LILLE 1, Fevrier 1988

[CAR] L. Cardelli
A Semantic of multiple inheritance
Lecture Notes in Computer Science, Springer-Verlag, New-York 1984

[CHI] MH. Chiloup, B. Cantegrit
Cnstraints in interactive Object Oriented Simulation
IMACS Annals on Computing and Applied Mathematics Proceedings MIM-S»'90, September 1990

Using Constraints for the construction of Object Oriented Interactive User Interfaces
Visualization in Engineering and Science, Linköping, August 1991

[DAH] K. El Dahshan
Constraint propagation in an object oriented environment, a mechanical CAD experience
Tools'90, Paris, June, 1990

[DEC] R. Dechter
Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition
Artificial Intelligence 29, 1990.

R. Dechter, J. Pearl
The anatomy of easy problems: a constraint satisfaction formulation
Proceedings of the IJCAI-85, Los Angeles, CA, 1985, pp 1066-1072.

R. Dechter, J. Pearl
Network-based heuristics for constraint-satisfaction problems
Artificial intelligence 34, pp 1-38, 1988.

R. Dechter, I. Meiri, J. Pearl
Temporal constraint networks
Technical report, R-113-L, october 1989

[DUC] R. Ducournau, M. Habib
On some algorithms for multiple inheritance in object oriented programming
ECOOP'87, Paris, Juin 1987

[DUI] R. A. Duisberg
Constraint-based animation: temporal constraints in the animus system
University of Washington, Ph. D. 1986

R.A. Duisberg
Animated graphical interfaces, using temporal constraints
CHI'86 Proceedings, April 1986

[EPS] D. Epstein, W.R. LaLonde
A Smalltalk Window system based on Constraints
OOPSLA'88, pp 83-94, September 25-30, 1988.

[FER] G. Fertey, B. Peroche
Creating 3D scenes with constraints
Tools'90, Paris, June, 1990

[FRE] E.C. Freuder
Synthesizing constraint expressions
ACM Communications, V.21, N°11, November, 1978.

[GOL] A. Goldberg, D. Robson
Smalltalk-80, The language and its implementation
Addison-Wesley Publishing Compagny, 1983

[GUS] H.W. Güsgen
CONSAT: a system for constraint propagation
Morgan Kaufmann publishers, Inc., San Mateo, California

H.W. Güsgen, J. Hertzberg
Some fundamental properties of local constraint propagation
Artificial intelligence 36, pp 237-247, 1988.

[HAI] B. Hailpern, V. Nguyen
A Generalized Object Model
Research Directions in OOP, Eds Shriver and Wegner, MIT Press, 1987

[HEI] N. Heintze, S Michaylov, P. Stuckey
Constraint logic programming and some electrical engineering problems
Technical Report N°84, March, 1987.

[JAF] J. Jaffar, J.L. Lassez
Constraint Logic Programming
Proceedings Symposium on Principles of programming Languages, Munich, 1987.

[LIE] H. Lieberman
Using Prototypical objects to Implement Shared Behavior in Object Oriented Systems
OOPSLA'86, Portland, Oregon, September 1986, pp 214-223.

[LON] R.L. London, R.A. Duisberg
Animating Programs using Smalltalk
IEEE Computer, pp 61-71, August 1985.

[MAC] A.K. Mackworth
Consistency in networks of relations
Artificial Intelligence 8, pp 99-118, 1977.

The complexity of some polynomial network consistency algorithms for constraint satisfaction problems
Artificial Intelligence 25, pp 65-74, 1985.

[MEV] A. Mevel, T. Guégen
Smalltalk-80
Eyrolles Edition, 1987

[MON] U. Montanari
Networks of constraints: fundamental properties and applications to picture processing
Information Sciences 7, pp 95-132, 1974.

U. Montanari, F. Rossi
Exact solution in linear time of network of constraints using perfect relaxation
Proceedings of the 1st international conference on principles of knowledge représentation and reasoning, pp 394-399, 1989.

[PIN] L.J. Pinson, R.S.Wiener
An Introduction to Object Oriented Programming and Smalltalk
Addison-Wesley Publishing Compagny

[SNY] A. Snyder
Encapsulation and Inheritance in Object Oriented Programming Languages
OOPSLA'86, Oregon, September 1986

[STE] M. Stefik, D.G. Bobrow
Object Oriented Programming : themes and variations
AI magazine 6, pp 40-62, 1984

[STEE] G.L Steele Jr
The definition and implementation of a computer programming language based on constraints
Technical Report,MIT. AI-TR-595, August 1980

[STEI] L.A. Stein
Delegation is inheritance
OOPSLA '87, October 4-8, 1987

[STR] B. Stroustrup
What is Object Oriented Programming ?
ECOOP'87 Proceedings, Paris, June 1987

[SUS] G.J. Sussman, G.L. Steele Jr
CONSTRAINTS - a language for expressing almost-hierarchical descriptions
Artificial intelligence 14 , pp 1-39, 1980

[SZE] P.A. Szekely, B.A. Myers
A user interface Tollkit based on graphical objects and constraints
OOPSLA'88, September 25-30, 1988.

[WAL] D.L. Waltz
Generatic Semantic descriptions from drawings of scenes with shadows
AI-TR-271, AI. Lab. , MIT, Cambridge, Mass., 1972

[WEG] P. Wegner
Dimensions of object oriented language design
OOPSLA '87, October 4-8, 1987

[WOL] F. Wolinski
Gestion des contraintes dans la structuration des objets en sous-objets
RFIA, Paris, 1989.

[ZDO] S. Zdonik, P. Wegner
Language and Methodology for Object Oriented Databases
Hawaï Conference on System Sciences, January 1986

ANNEXE_1

ANNEXE_1 : IMPLANTATION EN SMALLTALK-80 DU MECANISME DE DEPENDANCE

La définition d'une relation de dépendance se déroule en deux phases :

- la définition même de la relation ;
- la gestion de la dépendance.

Les mécanismes de base nécessaires à la définition et à la gestion des dépendances tiennent en peu de méthodes, toutes définies au niveau de la classe OBJECT. Ces méthodes décrivent le comportement par défaut de tout objet-maître et de tout objet-dépendant. Il revient à l'utilisateur de les redéfinir pour les classes particulières décrivant son application.

1. DEFINITION DE LA RELATION DE DEPENDANCE

1.1 La variable globale DependentsFields

DependentsFields est un dictionnaire déclaré comme variable globale. En tant que tel, il sera donc accessible de toutes les classes du système.
Dans ce dictionnaire sont stockées toutes les relations de dépendance définies sur l'image. Il est, comme tout dictionnaire, constitué d'une suite d'associations, 'clé → valeur', où chaque clé référence un objet-maître et la valeur associée, une collection des objets-dépendants de l'objet-maître.

1.2) Méthodes de gestion du dictionnaire des dépendances

Les méthodes générales d'accès au dictionnaire des dépendances doivent être comprise par tout objet, puisqu'à priori tout objet peut avoir de dépendants, elles ont donc été écrites dans la classe OBJECT. Ces méthodes permettent d'ajouter ou d'enlever une dépendance sur un objet-maître, de consulter ou de détruire tous les liens de dépendance définis sur un objet

1.2.1) Pour définir une dépendance

Dans toutes les méthodes qui vont suivre, dans ce paragraphe comme pour les suivants, l'objet receveur du message est l'objet défini comme objet-maître.

Object methodsFor: 'dépendent access'
** addDependent: anObject*
^(DependentsFields at: self ifAbsent: [^ DependentsCollection new]) add: anObject

DependentsFields at: self ifAbsent: [...], renvoie la collection des objets-dépendants stockée dans le dictionnaire des dépendances à la clé "self" qui référence le receveur. Si le receveur n'a pas encore de dépendant, ce message crée une collection.
add: anObject, ajoute à la collection renvoyée précédemment l'objet-dépendant, anObject.

1.2.2) Pour détruire une dépendance

Object methodsFor: 'dépendent access'
** removeDependent: anObject*
^(DependentsFields at: self) remove: anObject

remove: anObject, retire l'objet, anObject, de la collection des objets-dépendants du receveur.

1.2.3) Pour accéder à tous les dépendants d'un objet

Object methodsFor: 'dépendent access'
** myDependents*
^DependentsFields collectionAt: self ifAbsent: [nil]

renvoie la collection des objets-dépendants associée au receveur dans le dictionnaire des dépendances.

1.2.4) Pour détruire tous les liens de dépendance définis sur un objet

Object methodsFor: 'release'

* *breakDependents*

DependentsFields removeKey: self ifAbsent: []

retire la clé "self", qui référence le receveur, du dictionnaire des dépendances. Tout lien de dépendance entre le receveur (comme objet-maître) et tout autre objet est détruit.

* *release*

self breakDependents

Remarque

Il est possible de redéfinir ces différentes méthodes dans une sous classe d'OBJECT, si cela s'avère nécessaire. On verra par exemple, à la fin de ce chapitre, comment le message *release* a été redéfini dans la classe TRAFFICLIGHT.

2. REPERCUSSION D'UN CHANGEMENT DE L'OBJET-MAITRE VERS SES OBJETS-DEPENDANTS

Nous savons que le mécanisme de dépendance permet de faire savoir à tous les objets-dépendants d'un objet-maître que celui-ci a changé et qu'ils doivent, par conséquent, se mettre à jour.
Les méthodes permettant de répercuter de cette façon un changement de l'objet-maître vers ses dépendants ont été implantées dans la classe OBJECT.

2.1) L'objet-maître veut changer, il demande à tous ses dépendants s'ils sont d'accord avec ce changement

Object methodsFor: 'changing'

** changeRequest*

^self myDependents updateRequest

cette méthode envoie à la collection des objets-dépendants du receveur le message updateRequest.

DependentsCollection methodsFor: 'updating'

** updateRequest*

1 to: self size do: [:i | (self at: i) updateRequest ifFalse: [^false]]. ^true.

quand elle reçoit ce message, la collection des objet-dépendants envoie, à chacun de ses objets, le message updateRequest.

Object methodsFor: 'updating'

** updateRequest*

^true

le comportement par défaut d'un objet-dépendant est d'accepter tout changement de l'objet dont il dépend. Cette méthode doit être redéfinie dans une sous classe d'OBJECT si l'on désire changer ce comportement pour une catégorie d'objets.

Object methodsFor: 'changing'

** changeRequest: anAspectSymbol*

^self myDependents updateRequest: anAspectSymbol

cette méthode est identique à la précédente, mais précise la nature du changement.

DependentsCollection methodsFor: 'updating'

** updateRequest: anAspectSymbol*

1 to: self size do: [:i | ((self at: i) updateRequest: anAspectSymbol) ifFalse: [^false]].

^true.

Object methodsFor: 'updating'

** updateRequest: anAspectSymbol*

^self updateRequest

par défaut, l'objet-dépendant accepte la demande quelque soit la nature du changement

2.2) L'objet-maître annonce qu'il a subit un changement et demande à tous ses dépendants d'en tenir compte

Object methodsFor: 'changing'

** changed*

self changed: nil

** changed: anAspectSymbol*

self changed: anAspectSymbol with: nil

** changed: anAspectSymbol with: aParameter*

self myDependents

update: anAspectSymbol

with: aParameter

from: self

ce message envoie à la collection des objets-dépendants du receveur le message *update:with:from:*.

DependentsCollection methodsFor: 'updating'

** update: anAspect with: aParameter from: anObject*

1 to: self size do: [:i | (self at: i) update: anAspect with: aParameter from: anObject].

le message *update:with:from:* est transmis à tous les objet-dépendants de l'objet-maître.

Les dépendants de l'objet-maître, informés du changement de celui-ci, en tiennent compte par une remise à jour:

Object methodsFor: 'updating'

** update: anAspectSymbol*

^self

** update: anAspectSymbol with: aParameter*
^self update: anAspectSymbol

** update: anAspectSymbol with: aParameter from: anObject*
^self update: anAspectSymbol with: aParameter

par défaut l'objet-dépendant ne répond pas à ces messages de mise à jour, ou plus exactement, il y répond en ne faisant aucune action. Ces messages doivent être redéfinis dans les sous classes concernées, par exemple dans la classe VIEW, le message *update* a été redéfini par:

update
self display

2.3) Les messages, broadcast et perform, définis dans Object

Ces messages permettent à l'objet maître de dialoguer avec ses dépendants, en dehors de la simple répercussion d'un changement.

- la demande :

Object methodsFor: 'message passing'
** broadcast: aSymbol*
self myDependents performUpdate: aSymbol

** broadcast: aSymbol with: anObject*
self myDependents performUpdate: aSymbol with: anObject

le premier message envoie à tous les objets-dépendants du receveur le message de sélecteur aSymbol. Le second envoie le message de sélecteur aSymbol avec l'argument, anObject.

- l'action :

DependentsCollection methodsFor: 'updating'
** performUpdate: aSymbol*
1 to: self size do: [:i | (self at: i) perform: aSymbol]

** performUpdate: aSymbol with: anObject*
1 to: self size do: [:i | (self at: i) perform: aSymbol with: anObject

envoie au receveur le message, de sélecteur asymbol, avec l'argument anObject s'il existe.

3. EXEMPLE D'UTILISATION DU MECANISME DE DEPENDANCE [MEV]

On désire modéliser le fonctionnement d'un feu tricolore. L'objet, feu, est constitué de trois lampes de couleur différente. Seule l'une d'entre elles peut être allumée (état On) à un moment donné. De ce fait, l'état On-Off de chacune des trois lampes est dépendant de l'état des deux autres.

On crée la classe LAMPE comme suit:

```
Object  subclass: #Light
        instanceVariableNames: 'state'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Exemple'
```

Light methods for: 'initialization'

"L'état de la lampe est modélisé par une variable booléenne. Si la lampe est allumée cette variable est à vrai, sinon elle est à faux."

```
* setOff
        state := false
* setOn
        state := true
```

Light methods for: 'test'

```
* isOff
        ^ state not
* isOn
        ^ state
```

Light methods for: 'operate'

```
* turnOff
"Si la lampe est allumée, le fait de l'éteindre oblige à signaler ce changement (toOff)"
        self isOn ifTrue:
        [ state := false. self changed: #toOff with: self ]
```

** turnOn*

"Si la lampe est éteinte, le fait de l'allumer oblige à signaler ce changement (toOn)"

self isOff ifTrue:

[state := true. self changed: #toOn with: self]

Light methods for: 'updating'

** update: statusChange with: aLight*
"le receveur est dépendant de l'objet, aLight, et l'état de celui-ci vient de changer, ce changement étant référencé par le paramètre statusChange. Si statusChange est une mise-à-On et que le receveur est allumé, alors il est aussitôt mis-à-Off"
aLight = = self ifTrue: [^self]
(statusChange = #toOn & isOn) ifTrue:
[self turnOff]

La classe TRAFFICLIGHT est définie pour représenter un ensemble de lampes coordonnées (les lampes étant dépendantes les unes des autres).

Object subclass: #TrafficLight
instanceVariableNames: 'lights'
classVariableNames: ''
poolDictionaries: ''
category: 'Eemple-Lights'

TrafficLight methods for: 'release'
** release*
"cette méthode détruit toutes les dépendances définies sur le receveur. Elle détruit de même toutes les dépendances définies sur les lampes constituant le receveur."
super release.
lights do: [:alight | alight release].
lights := nil

TrafficLight methods for: 'initialization'

```
* with: anInteger
        lights := Array new: (anInteger max: 1).
        1 to: anInteger do:
            [:index | lights at:index put: Light new setOff].
        lights do:[:eachlight | lights do:[:dependentlight |
            eachlight ~~ dependentlight ifTrue:
            [eachlight addDependent: dependentlight]]]
```

Une instance de la classe TRAFFICLIGHT est créée par le message, *'TrafficLight new with: aNumber'*, où l'argument *aNumber* donne le nombre de lampes gérées par le feu. Chaque lampe, créée à l'état Off, est connectée à toutes les autres à travers un réseau de dépendances (message *addDependent:*). L'ensemble des lampes constituant le feu est référencé par la variable d'instance **lights.**

TrafficLight methods for: 'operate'

```
    * turnOn: alightNumber
        alightNumber > 0 & (alightNumber < = lights size)ifTrue:
        [(lights at: alightNumber) turnOn. self changed]
```

Pour agir sur une instance de la classe TRAFFICLIGHT, on lui envoie le message *'turnOn: alightNumber'*, où *alightNumber* désigne la lampe que l'on veut allumer. Cette lampe reçoit, alors, le message *'turnOn'*, qui la met à l'état "On". Tous ses dépendants, c'est à dire les autres lampes gérées par le feu, sont mises à l'état "Off". En effet, la méthode *'turnOn'* définie dans la classe LIGHT inclue le message, *'self changed: #toOn with:self'*, où self désigne la lampe mise à "On". Ce message, nous l'avons vu, répercute le changement sur tous les objet-dépendants du receveur, en particulier sur les autres lampes constituant le feu.

Remarque: Lorsqu'une instance de la classe TRAFFICLIGHT reçoit le message *'turnOn:alightNumber'*, elle s'envoie le message *'changed'*, qui signale à tous ses dépendants qu'elle a subit un changement et leur demande de se mettre à jour. Ce message sera utile, par exemple, si on modélise un carrefour routier régi par des feux tricolores.

Utilisation d'un feu

Soit un feu tricolore dont les lampes sont notées R, O, V.

A la création de cet objet-feu, les variables d'instance **status** de chacune des lampes (respectivement $\mathbf{status_R}$, $\mathbf{status_O}$, $\mathbf{status_V}$) sont instanciées à la valeur booléenne 'false'. A l'initialisation, toutes les lampes du feu tricolore sont donc éteintes. Puis trois associations sont ajoutées au dictionnaire des dépendances DependsFields :

' R -> (V,O) ',
' O -> (V,R) ',
' V -> (O,R) '.

Pour allumer la lampe R ont envoie le $message_{(1)}$ *'feu turnOn:1'*. Ce message déclenche la séquence de messages suivante :

- $message_{(2)}$ *'R turnON'*,
qui instancie la variable d'instance $\mathbf{status_R}$ à la valeur booléenne 'true' et envoie le $message_{(3)}$ *'R changed: #toON with: R'*. Ce message envoie, à tous les objets dépendants de la lampe R, le $message_{(4)}$ *'update: #toOn with: R'* qui n'à aucune incidence sur les lampes O et V car elles sont déjà éteintes.

- $message_{(5)}$ *'feu changed'*,
qui n'a aucun effet puisque le feu tricolore n'a pas d'objet dépendant.

Partant de ce nouvel état, on envoie le $message_{(6)}$, *'feu turn:3'*, qui déclenche la séquence de messages suivante :

- $message_{(3)}$ *'V turnON'*,
qui instancie la variable d'instance $\mathbf{status_V}$ à la valeur booléenne 'true' et envoie le $message_{(7)}$ *'V changed: #toON with: R'*. Ce dernier message envoie, à tous les objets dépendants de la lampe V, le $message_{(8)}$ *'update: #toOn with: V'*.

- *'R update: #toOn with: V'* affecte à $\mathbf{status_R}$ la valeur booléenne 'false' (la lampe R est éteinte), en lui envoyant le message *'turnOff'*. Les objets dépendants de R, c'est à dire les lampes O et V, reçoivent le message *'update: #toOff with: V'* qui n'à aucune incidence sur ces lampes car elles sont déjà éteintes.

- *'O update: #toOff with: V'* n'a aucun effet sur la lampe O qui est éteinte.

- message$_{(9)}$ *'feu changed'*,
qui n'a aucun effet puisque le feu tricolore n'a pas d'objet dépendant.

Les relations de dépendance établies entre les lampes du feu tricolore ont donc permis, lorsqu'une lampe est allumée, d'éteindre toutes les autres.

ANNEXE_2

ANNEXE_2 : SYNTHESE BIBLIOGRAPHIQUE

Parmi les premiers travaux introduisant la notion de contrainte et mettant en évidence l'intérêt d'un mécanisme de satisfaction automatique des contraintes, on peut citer les systèmes Sketchpad d'Ivan Sutherland (1963) et Constraints de Guy L. Steele et Gerald J. Sussman (1978).

- **Sketchpad** est un système interactif de conception et d'édition graphique. Dans ce système, la notion de contrainte est étroitement liée à celle de composition d'objets. Les chemins permettant de remonter de "pères en pères" sont précompilés puis seront utilisés pour propager, compte tenu des contraintes, le long de la structure de l'objet composé, les différentes valeurs instanciées ("one pass method").
Quand la propagation, exclusivement locale, échoue, le mécanisme de satisfaction fait appel à la relaxation: chaque contrainte est alors représentée par un sous programme simple qui calcule et cherche à minimiser une valeur d'erreur (degré de satisfaction de la contrainte pour les valeurs actuelles).
Sketchpad utilise les contraintes pour modéliser les lois de la géométrie, toutes sont des égalités définissant des relations linéaires sur les variables.

- **Constraints** est un système destiné à la conception de circuits électriques. Il s'appuie sur un ensemble de contraintes primitives (additionneur, multiplieur...) permettant de modéliser les circuits.
Le mécanisme de satisfaction des contraintes, employé ici, est également basé sur la propagation locale des valeurs, mais contrairement au système précédent, il n'intègre pas d'algorithme de relaxation. Lorsque la propagation des valeurs échoue (essentiellement lorsque les équations, définies par les contraintes, ne sont plus linéaires), le système cherche une autre "vue" de la partie du circuit lui posant problème: le mécanisme de satisfaction est alors relancé sur le nouvel ensemble de contraintes obtenu ("multiple redundant views method").
La satisfaction des contraintes étant basée sur un algorithme assez simple, il est fréquent que celle-ci échoue: le système fait, dans ce cas, appel à l'utilisateur. Pour l'aider dans ses décisions, Constraints enregistre en permanence la trace du plan de satisfaction des contraintes retenu et offre, ainsi, la possibilité d'un retour vers une étape intermédiaire ("dependency-directed backtracking method").

Nous ne décrirons pas ces deux systèmes plus en détail, nos propos étant simplement, ici, de mettre en évidence les "concepts clés" de ces travaux, concepts repris, nous le verrons, par la majorité des systèmes basés sur les contraintes. Le plus cité d'entre tous est, sans nul doute, ThingLab conçu par Alan Borning. Certains articles décrivent déjà ce langage dés 1979, toutefois les travaux le concernant n'ont réellement abouti qu'aux alentours de 1986.

Avant d'entrer dans la description de Thinglab et d'autres systèmes, faisons un bref rappel sur la théorie des contraintes, des réseaux de contraintes et de leur satisfaction. Sans remonter au "moyen âge", la figure 1 montre la chronologie des différents travaux publiés dans ce domaine, tout en séparant la théorie des applications (figure 1.a et figure 1.b respectivement).

1974 **Montanari,** 'Networks of constraints: fundamental properties'

1977 **Mackworth,** 'Consistency in networks of relations'
1978 **Freuder,** 'Synthesizing constraint expressions'

1981 **Stefik,** 'Molgen, planning with constraints'
1982 **Fox, Allen, Strohm,** 'Investigation in constraint-direct reasoning'

1985 **Dechter, Pearl,** 'A constraint satisfaction formulation'
Descotte, Latombe, 'Compromises among antagonist constraints'
Mackworth, Freuder, 'Complexity of some polynomial network consistency algorithms'

1987 **Davis,** 'Constraint propagation with interval labels'
Geffner, Pearl, 'An improved constraint propagation algorithm for diagnosis'
Heintze, Michaylov, Stuckey, 'Constraint Logic Programming'
Jaffar, Lassez, 'Constraint Logic Programming'
1988 **Dechter, Pearl,** 'Network-based heuristics for constraint satisfaction problems'
Güsgen, Hertzberg, 'Fundamental properties of local constraint propagation'
'Relaxing constraint network to resolve inconsistencies'
1989 **Dechter, Meiri, Pearl,** 'Temporal constraint network'
Fidelak, Güsgen, Voss, 'Temporal aspect in constraint based reasoning'
1990 **Dechter,** 'Enhancement schemes for constraint processing: backjumping, learning,...'

Figure 1.a

Nous nous bornerons, en ce qui concerne la théorie, à ne rappeler que quelques définitions de manière à mettre davantage l'accent sur diverses applications:

contrainte: une contrainte est une relation sur un ensemble de variables,

réseau de contraintes: lorsqu'il existe plusieurs contraintes sur un même ensemble de variables, on parlera de réseau de contraintes. Dans un tel réseau, les noeuds représentent les variables (plus généralement, l'ensemble des valeurs pouvant être prises par une variable), chaque arc représentant une contrainte sur deux de ces variables (relation liant deux noeuds entre eux),

'node-consistency': un noeud est dit consistant si pour chacune de ses valeurs, il répond aux contraintes qui lui sont propres,

'arc-consistency': un arc est dit consistant si chaque noeud qu'il relie est consistant et s'il existe, pour chacune des valeurs du noeud de départ, une valeur du noeud d'arrivée qui permette à la contrainte définie entre ces deux noeuds d'être vérifiée,

'path-consistency': un chemin de longueur m, entre deux noeuds quelconques d'un réseau, est dit consistant, si pour chaque couple de valeurs, composé d'une valeur du noeud de départ et d'une valeur du noeud d'arrivée, tel que la contrainte entre ces deux noeuds soit vérifiée, il existe une séquence de m valeurs vérifiant les principes de 'node consistency' et d'arc consistency',

consistance d'un réseau: un réseau est consistant s'il répond aux trois règles précédemment citées,

'consistency algorithmes': différents algorithmes permettant de rendre un réseau consistant, par rapport à ses noeuds, ses arcs ou ses chemins. Ces algorithmes ont été ébauchés par Ugo Montanary puis développés par Alan Mackworth. Rendre un réseau consistant revient à le rendre minimal (pour chaque noeud, seuls les domaines de valeurs probables sont maintenus),

'CLP, constraint logic programming': théorie de la programmation logique avec contraintes.
Les CLP programmes, basés sur un travail d'unification de sémantique formelle, sont hautement déclaratifs.
L'unification étant un cas particulier de la résolution des contraintes, les schémas classiques de la programmation logique (utilisés par exemple dans Prolog) ont été étendus, le concept de résolution des contraintes étant employé à la place du concept, plus restrictif, d'unification.
On peut ainsi définir des règles plus complexes du type:

```
complexmult (c(R1,I1),c(R2,I2),c(R3,I3)); -

        R3 = R1*R2 - I1*I2,
        I3 = R1*I2 + R2*I1.
```

Pour résoudre les contraintes, les CLP programmes utilisent des "solveurs de contraintes" adaptés à la technique de réduction des buts: durant l'exécution (recherche du but), la technique de réduction des buts permet de regrouper toutes les contraintes, puis les solveurs sont utilisés pour déterminer la valeur des variables encore indéterminées.

Pour plus d'informations concernant la théorie des contraintes, l'utilisateur pourra se reporter aux publications citées en bibliographie.

APPLICATIONS

1978 **Constraint** (Steele, Sussman)

1981 **ThingLab**, premiers travaux (Borning)

1985 **Consat**, premiers travaux (Güsgen)
Juno (Nelson)
1986 **Grow** (Barth)
ThingLab (Borning),**Animus** (Duisberg)
MvS|C (Lischka, Güsgen)
CRTL (Cohen, Smith, Iverson)

1988 **CWS** (Epstein, LaLonde)
Consat (Güsgen)
Coral (Szekely, Myers)

1990 **Cool** (Avesani, Perini, Ricci)
Opal (El Dahshan)

Figure 1.b

La figure 1.b donne l'ordre chronologique d'apparition de différents systèmes utilisant les contraintes. Cette liste n'est évidemment pas exhaustive; toutefois elle reflète, par sa diversité, les différents domaines d'applications justifiant l'usage des contraintes.

Le système **Constraints**, créé par Gerald J. Sussman et Guy L. Steele, à la fin des années 70, est un système destiné, nous l'avons vu, à la conception et à l'étude des circuits électriques. Sans revenir sur la description de ce système, on peut remarquer que G.L. Steele définit un programme, à base de contraintes, comme un réseau d'éléments connectés par des relations, tout comme les composants d'un circuit électrique sont connectés par des fils conducteurs. Chaque élément du réseau calcule et place, utilisant les informations disponibles localement, les nouvelles valeurs, dérivées d'autres valeurs attachées aux contraintes.

Dans leurs travaux, les auteurs ont mis en évidence, l'utilité des vues multiples d'un même système et l'intérêt d'enregistrer la justification de toute décision prise par le mécanisme de satisfaction des contraintes si une aide doit être demandée à l'utilisateur.

Les principaux reproches, pouvant être faits à Constraints, sont:

- les seules contraintes pouvant être représentées sont des contraintes algébriques simples,
- le mécanisme de satisfaction des contraintes fait fréquemment appel à l'utilisateur pour résoudre les conflits,
- le système n'offre aucune possibilité graphique

ThingLab est un outil de simulation orienté objet et contrainte, créé par Alan Borning, puis développé ensuite par une équipe plus large, composée entre autres de R. Duisberg, B. Freeman-Benson et J. Maloney.
ThingLab est un système en perpétuel développement, l'utilisateur trouvera en annexe une description détaillée de la première version du système, ThingLab I (qui regroupe les principaux concepts utilisés), et de l'une de ses premières extensions créée par R. Duisberg, le système **Animus**.
Alan Borning a été l'un des premiers à avoir introduit la notion de contrainte dans le domaine de la programmation orientée objet. Son système est, à l'heure actuelle, l'un des plus avancés pour ce qui est de l'intégration des différentes techniques de résolution des contraintes dans le mécanisme interne de résolution. Ce mécanisme utilise, dans un premier temps, les techniques de propagation des états connus et des degrés de liberté afin de trouver un plan de satisfaction des contraintes (one-pass method). S'il échoue dans cette tentative, le mécanisme compile une méthode de relaxation et en informe l'utilisateur, pour que celui-ci puisse, s'il le désire, fournir une nouvelle contrainte au système qui permettrait à la propagation d'aboutir (toutes les relations représentées par les contraintes sont, ici, exclusivement des égalités).
ThingLab a été conçu comme une extension de Smalltalk, ce qui a permis à ce système d'utiliser directement les notions d'objet, de classe, d'instance et de message. Cependant c'est un outil indépendant, il n'intègre donc pas l'utilisation des contraintes dans le langage de programmation Smalltalk.

Le principal domaine d'utilisation de ThingLab est la création et la gestion d'interfaces graphiques, utilisateur-machine, hautement conviviales. Les contraintes y sont essentiellement utilisées à deux fins:

- Maintenir une parfaite cohérence entre une donnée interne et sa représentation graphique: si le programme modifie la donnée interne, sa représentation est immédiatement mise à jour et inversement si l'utilisateur agit, par le biais de l'interface, sur la représentation graphique d'une donnée interne, celle-ci est immédiatement modifiée.
- Permettre la composition d'objets, par les contraintes de fusion (merges), et faciliter la description des relations existantes entre les différentes parties d'un objet composé.

Principaux inconvénients du système:

- il n'introduit que des contraintes statiques égalitaires,
- tout objet est défini comme un objet composé, sauf les objets dits primitifs (les entiers...),
- les objets sont typés,
- le principe d'héritage n'est pas retenu,

chaque classe possède un prototype. C'est une instance particulière de la classe ne répondant pas aux mêmes messages que les instances dites classiques. Le prototype

Consat est un système pour la satisfaction des contraintes créé par Hans Güsgen. Ce système utilise la stratégie standard de propagation locale (Waltz filtering) étendue de différentes façons pour calculer, soit localement, soit globalement, des solutions cohérentes.
Etant donnés des valeurs ou des ensembles de valeurs possibles pour un ensemble de variables, les réseaux de contraintes peuvent être utilisés:

- pour calculer les valeurs des variables indéterminées, sur l'ensemble des variables données,
- pour tester la consistance des valeurs,
- pour filtrer les ensembles de valeurs possibles, c'est à dire, pour éliminer les valeurs incohérentes de ces ensembles.

Consat peut, en théorie, manipuler

- des domaines de valeurs arbitraires,
- des valeurs multiples,
- des relations finies et infinies,
- des hiérarchies de contraintes.

Juno, est un système créé par Greg Nelson, qui "intègre harmonieusement un langage de description graphique à un éditeur d'image du type, *"ce-que-vous-voyez-est-ce-que-vous-obtenez"*. Les contraintes employées dans ce système sont exclusivement des contraintes géométriques fixant des positions dans la fenêtre d'édition.
Le mécanisme de satisfaction des contraintes transforme les contraintes géométriques (colinéarité, équidistance...) en contraintes numériques sur les coordonnées des points.
Dans le domaine de la géométrie, la non-linéarité étant fréquente (parallélisme, congruence...), Juno utilise un solveur de contraintes basé sur la méthode d'itération de Nelson-Raphson, cette méthode étant plus rapide que la relaxation classique.
Dans ce langage, une "commande contrainte" s'écrit de la forme:

LET Variables | Constraints IN Command END

- Variables est une liste des variables locales,
- Constraints est une conjonction de contraintes sur les valeurs des variables
- Command indique l'opération qui doit être faite sur les variables

Si aucune valeur ne permet à la liste des variables de répondre à l'ensemble des contraintes, la commande échoue.
Si les contraintes ne déterminent pas les valeurs des variables de manière unique, la commande est indéterministe.
Quatre types de contraintes sont permises:

- l'équidistance (x,y) cong (u,v),
- le parallélisme (x,y) para (u,v),
- l'horizontalité hor(x,y),
- la verticalité ver(x,y).

Grow (de Paul Barth) est un système orienté objet destiné à la construction d'interfaces graphiques.
Ce système se base sur:

- l'utilisation d'objets graphiques,
- la définition de relations sur ces objets
  * relations de composition,
  * relations graphiques (dépendance de données entre attributs d'objets graphiques).
- la séparation de l'application et de son interface

Les dépendances sont représentées de façon déclarative au niveau même des objets, ce qui permet l'utilisation des techniques classiques appliquées à la propagation des contraintes.
En fait, le système instauré par Grow travaille sur deux types de structures:

- la structure de composition, arbre des objets composant un objet composite. Cet arbre peut être d'une profondeur arbitraire, mais il reste une structure simple et linéaire.
- le treillis de dépendances.

Une dépendance est une relation spécifiant une dépendance de données entre attributs d'objets (notions de dépendants et de gardiens). Un attribut peut avoir plusieurs gardiens et plusieurs dépendants. Les dépendances étant spécifiées au niveau des attributs d'objets, à chaque attribut est annotée une liste de ses gardiens (attributs desquels il dépend).
Grow construit automatiquement et maintient le treillis de dépendances à partir des annotations des gardiens. Bien que non linéaire, le treillis ne doit, cependant, pas être cyclique d'où l'utilisation d'un algorithme simple de lecture et de propagation.

MvS|C, conçu par Christoph Lischka et Hans Güsgen, n'est pas un système mais une application utilisant un système à base de contraintes (Consat).
Cette étude montre l'intérêt d'un système de satisfaction des contraintes face à un problème spécifique à la musique: l'interprétation harmonique d'une mélodie chorale.

Les problèmes traités sont:

- la représentation de la connaissance musicale (le langage utilisé pour la représentation des données est Babylon),
- la modélisation de la "pensée musicale" et des interactions entre cette pensée et d'autres processus cognitifs (par l'utilisation du formalisme des contraintes).

Etant donnée une "phrase chorale" et un dictionnaire des figures mélodiques (association de notes et de contraintes d'harmonie), le système dresse une "carte" des figures reconnues dans la phrase. Cette carte est, en fait, un réseau de contraintes. De ce réseau, le mécanisme de satisfaction des contraintes déduit, par une technique de propagation locale, les ensembles de fonctions harmoniques correspondants aux notes.

CRTL, 'Constraint Rectangular Tiled Layout', est un système utilisant les contraintes pour gérer l'agencement et le maintient des fenêtres à l'écran (auteurs: Ellis Lohen, Edward Smith, Lee Iverson).
Différents types de contraintes sont référencées:

- contraintes de présence d'une fenêtre à l'écran,
- contraintes de taille et de position de fenêtre,
- contraintes d'adjacence et d'alignement,
- limites dans lesquelles le système doit automatiquement recadrer,
- contraintes d'organisation.

En cas de conflit entre contraintes, le CRTL utilise un principe déjà introduit dans d'autres systèmes (ex: ThingLab):
les contraintes sont organisées selon un ordre hiérarchique. Une contrainte de basse priorité peut alors être cassée si cela permet à une contrainte plus forte d'être maintenue.

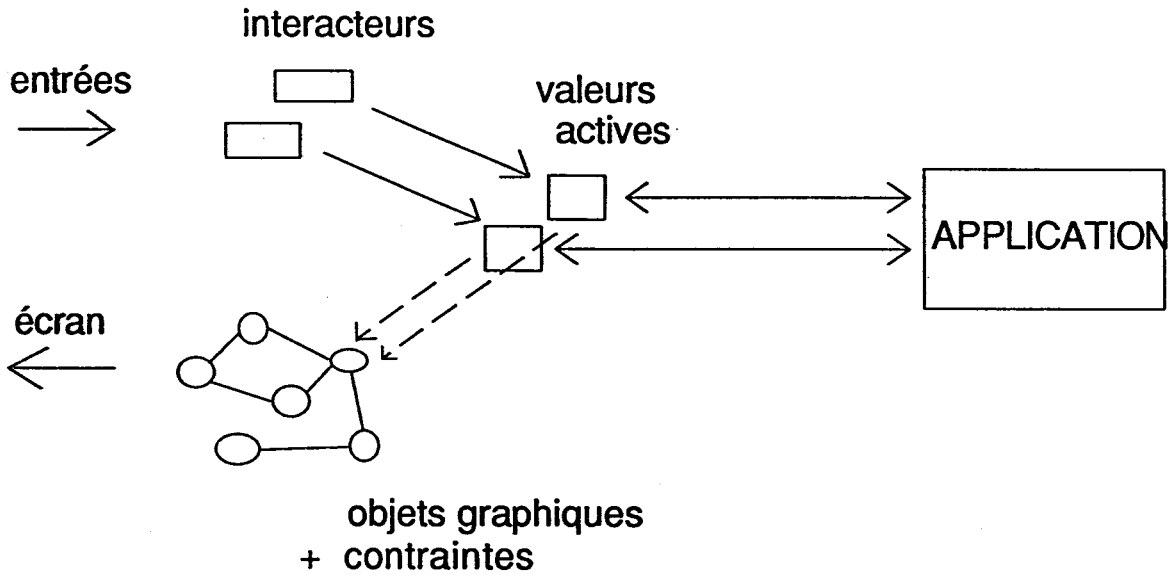
CWS ('Constraint Window system)', créé par Danny Epstein et Wilf LaLonde, utilise les contraintes pour spécifier les attributs des fenêtres et les relations entre elles.
Ce système est une extension de Smalltalk. L'utilisateur décrit la configuration d'une fenêtre en termes de contraintes prioritaires, c'est à dire, par des relations booléennes qui doivent être satisfaites suivant un schéma de priorités.
CWS permet de contraindre la taille et le système de coordonnées de la fenêtre. Des contraintes initiales peuvent être utilisées pour spécifier le dessin initial d'une configuration de fenêtres.
Deux types de contraintes peuvent être utilisés par l'utilisateur, les ancres et les contraintes égalitaires. Ces contraintes sont des relations entre variables contraintes (number-variable, point-variable, rectangle-variable, transformation-variable).
Comme le système d'Alan Borning, CWS compile les différents plans de résolution des contraintes. Compiler un plan est relativement lent mais la solution compilée est efficace, en fait optimale, pour un bon nombre de modifications.

Coral, créé par Petro Szekely et Brad Myers, est un outil d'aide à la conception d'interfaces graphiques. Le problème est donc, ici, de gérer les relations entre objets graphiques et les données internes qu'ils représentent.
Coral possède deux interfaces:

- une interface déclarative, pour spécifier les contraintes. L'utilisateur peut y définir des contraintes, de façon abstraite, pour les voir appliquées à différentes instances.
- une interface procédurale, elle permet à un système de gestion de l'interface utilisateur de créer, automatiquement, des appels à Coral.

Ce système introduit les notions:

- d'objets graphiques,
- de contraintes graphiques,
- de valeurs actives,
- d'interacteurs.



Les valeurs des attributs des objets graphiques (lignes, rectangles, arcs...), peuvent être définies en fonction de valeurs des attributs d'autres objets graphiques en utilisant les contraintes.
Seules les contraintes unidirectionnelles sont permises, ce qui enlève une grande part d'interactivité au système (si une donnée interne est modifiée, sa représentation graphique est immédiatement mise à jour, l'inverse n'étant pas vérifié).
Il conviendrait donc, pour ce système, de parler de relation de dépendance (unidirectionnelle) plutôt que de contrainte (multidirectionnelle).

COOL (Paolo Avesani, Anna Perini, Francesco Ricci) et **OPAL** (Kamal El Dahshan) sont deux systèmes encore en phase de développement.

* **COOL** est un système de satisfaction de contraintes construit sur un canevas similaire à celui de Consat. Il met en pratique les résultats tirés de l'étude des CSP, 'Constraint Satisfaction Problem' (cf. Montanari, Mackworth):
  - définition d'un réseau de contraintes,
  - "node/arc et path consistance",
  - méthode de résolution : "backtracking algorithme".

* **OPAL** fait partie de la famille des systèmes ICAD (Intelligent Computer Aided Design). Deux types de contraintes y sont définies:
  - des contraintes topologiques,
  - des contraintes géométriques ou contraintes de dimension.

La technique de résolution implantée dans ce système est la propagation locale.