Travail d'Etude et de Recherche

REALISATION OBJET D'UN MINI-SOLVEUR CSP

—

Sébastien RAMON
Master 1 Mathématiques Informatique (Informatique)
Faculté des sciences Jean PERRIN de Lens

Travail d'Etude et de Recherche encadré par
Stéphane CARDON et Christophe LECOUTRE
Centre de Recherche en Informatique de Lens (CNRS)

17 janvier 2007 - 11 mai 2007

"Les machines un jour pourront résoudre tous les problèmes,
mais jamais aucune d'entre-elles ne pourra en poser un"
Albert EINSTEIN, physicien allemand (1879-1955)

Table des matières

Chapitre 1

Introduction

1.1 Remerciements

Je remercie Monsieur GREGOIRE et l'ensemble du laboratoire de m'avoir accueillit au sein du CRIL et de m'avoir permis de réaliser ce projet.

Aussi je tiens à remercier tout particulièrement Stéphane CARDON et Christophe LECOUTRE pour leur encadrement irréprochable et pour le temps précieux qu'ils m'ont accordé durant cette période.

1.2 Description du sujet

Le but de ce projet est de réaliser un solveur de réseaux de contraintes binaires en C++ avec dossier d'analyse en UML afin de maintenir l'évolution du logiciel.

Un réseau de contraintes binaires est un graphe de n variables et e contraintes. Une contrainte est une relation binaire entre deux variables indiquant quel couple de valeurs est autorisé (ou interdit). La résolution d'un tel réseau consiste à trouver une affectation de chaque variable telle que toutes les contraintes soient vérifiées.

Dans sa première forme, le solveur devra :
- lire et charger des réseaux de contraintes au format XML,
- implanter la technique de filtrage AC la plus efficace à l'heure actuelle,
- implanter l'algorithme de résolution MAC effectuant une recherche en profondeur d'abord en maintenant une technique de filtrage.

Si le temps le permet, d'autres techniques de filtrage pourront être implantées (PC, SAC, etc...). D'ailleurs, le dossier d'analyse devra prendre en compte les eventuelles améliorations, tant au niveau filtrage qu'au niveau algorithmes de résolution. Finalement, il faudra garder à l'esprit que des modules pourront venir se greffer à cette base (transformation d'un CSP non binaire en un CSP binaire, etc...).

1.3 Plan du rapport

Afin de répondre aux lignes principales du projet dont description est faite ci-dessus, nous allons scinder celui-ci en plusieurs parties.

Dans un premier temps, nous allons nous atteler à la compréhension du thème du sujet. Effectivement, dans une partie appelée **Etude**, nous allons définir les *réseaux de contraintes, algorithmes de résolution* et autres *techniques de filtrage*, afin de lever toute ambiguité.

Ensuite, dans un souci de respect du cahier des charges, nous allons poursuivre au sein d'une partie appelée **Modélisation**, pour détailler une analyse du sujet au coeur de différents *diagrammes*

préalablement décrits.

Pour finir, nous présenterons notre implémentation du projet dans une partie appelée **Développement**, que nous aurons eu soin d'introduire en présentant les *outils* nécessaires à notre travail.

Enfin, nous *conclurons* afin de mettre en avant le bilan du projet, à travers l'évolution et les recherches que peut amener celui-ci

Première partie

Etude

Chapitre 2

Problèmes de satisfaction de contraintes

Nombre de problèmes en *Intelligence Artificielle* et en *Recherche Opérationnelle* peuvent être vus comme des problèmes de satisfaction de contraintes (Constraint Satisfaction Problems). De nombreux exemples existent en ordonnancement, planification, graphe, génétique, conception de circuits, ... Par ailleurs, les CSPs constituent en eux-mêmes un thème de recherche privilégié en IA depuis 1975, dont l'instance la plus caractéristique est le problème de satisfiabilité d'un ensemble de clauses de la logique propositionelle (SAT).

Un CSP est une **représentation d'un problème par ses contraintes**. De façon trés générale, une *contrainte* correspond à l'énoncé d'une propriété relative à différents objets. Ainsi, l'équation "$x.y + z = y$" met en relation les *variables* x, y et z en limitant les *valeurs* que peuvent prendre simultanément ces objets.

Résoudre un CSP consite à trouver un ensemble d'*instanciations* (variable = valeur) qui satisfasse l'ensemble des contraintes. Ce problème de décision est ***NP-Complet*** en consistance globale (le problème étant défini sur un produit d'ensembles, de cardinal fini, il peut se résoudre par une énumération complète des valeurs des variables). C'est pourquoi différents algorithmes de *consistance locale* (réduction du domaine d'application), et de recherche ont été proposés pour réduire la compléxité du problème (cf. *PPC*).

Exemple : *(firme automobile)*
Une firme automobile délocalise dans toute l'Europe la construction de l'un de ses modèles :

- Les portières et le capot sont fabriqués à LILLE où le constructeur ne dispose que de peinture rose, rouge et noire,
- La carroserie est fabriquée à HAMBOURG où le constructeur dispose de peinture blanche, rose, rouge et noire,
- Les pares-chocs, fabriqués à PALERME, sont toujours blancs,
- La bâche du toit-ouvrant est fabriquée à MADRID et ne peut être que de couleur rouge,
- Les enjoliveurs sont fabriqués à ATHENES où l'on ne possède que de la peinture rose et rouge.

Le concepteur de la voiture impose quelques-uns de ses désirs quant à l'agencement des couleurs :

- La carroserie, les portières et le capot doivent être de la même couleur,
- Les enjoliveurs, les pare-chocs et le toit-ouvrant doivent être plus clairs que la carosserie.

Comment répondre aux désirs du concepteur ?

2.1 Définition

Un *problème de satisfaction de contraintes* peut être représenté par un triplet (X, D, C), où :

- $X = \{v_1, v_2, ..., v_n\}$, est un ensemble de n variables,
- $D = \{d_1, d_2, ..., d_n\}$, est l'ensemble des n domaines finis associés aux variables ; chaque d_i est de cardinalité finie, soit $(d = max|d_i|)$,

- $C = \{c_1, c_2, ..., c_e\}$, est l'ensemble des e contraintes, chaque contrainte c_i est définie par un couple (v_i, r_i) :
  - v_i est une séquence de variable $(x_{i_1}, ..., x_{i_{n_i}}) \in X$ sur lesquelles porte la contrainte c_i. On appelle arité de c_i la longueur de v_i,
  - r_i est une relation définie par un sous-ensemble du produit cartésien $d_{i_1} \times ... \times d_{i_{n_i}}$ des domaines associés aux variables de v_i. Il représente les n-uplets de valeurs autorisées pour ces variables.

Le triplet (n, e, d) caractérise la *taille* d'un CSP. Un CSP *binaire* est un CSP dont toutes les contraintes ($c_i \in C$) ont une arité égale à 2.

La nature d'une contrainte n'est pas directement liée à son mode d'expression. En particulier, et dans le cas de domaine finis, il est toujours possible de ramener une contrainte exprimée en intention à une contrainte exprimée en extension, via l'ensemble de n-uplets de valeurs qu'elle autorise ou qu'elle interdit.

Exemple : *(firme automobile)*
Dans l'exemple de la *firme automobile*, on peut *définir* les différents points de la manière suivante. Les *variables* sont les différentes parties du véhicule, les *domaines* sont les ensembles de couleurs et les *contraintes* sont les désirs du concepteur, soit en intention (puis en extension, Tab. 2.1 et Tab. 2.2) :

- On définit la relation $<$ entre deux variables X_1 et X_2 comme étant $X_1 < X_2$ ssi X_1 est de couleur plus claire que X_2. On définit de la même manière la relation $=$ entre deux variables X_1 et X_2 comme étant $X_1 = X_2$ ssi X_1 et X_2 sont de la même couleur.
- $X = \{POR, CAP, CAR, PAR, TOI, ENJ\}$
- $D = \{(rs, rg, no), (rs, rg, no), (bl, rs, rg, no), (bl), (rg), (rs, rg)\}$
- $C = \{(POR = CAP), (POR = CAR), (CAP = CAR),$
- $(PAR < CAR), (TOI < CAR), (ENJ < CAR)\}$

Variables	Domaines
POR	$\{rs, rg, no\}$
CAP	$\{rs, rg, no\}$
CAR	$\{bl, rs, rg, no\}$
TOI	$\{rg\}$
PAR	$\{bl\}$
ENJ	$\{rs, rg\}$

Tab. 2.1: Variables du problème "firme automobile"

Contraintes	Scope	Relations
$C_1 = (v_1, r_1)$	$v_1 = (POR, CAP)$	$r_1 = \{(rs, rs), (rg, rg), (no, no)\}$
$C_2 = (v_2, r_1)$	$v_2 = (POR, CAR)$	$r_1 = \{(rs, rs), (rg, rg), (no, no)\}$
$C_3 = (v_3, r_1)$	$v_3 = (CAP, CAR)$	$r_1 = \{(rs, rs), (rg, rg), (no, no)\}$
$C_4 = (v_4, r_2)$	$v_4 = (PAR, CAR)$	$r_2 = \{(bl, rs), (bl, rg), (bl, no)\}$
$C_5 = (v_5, r_3)$	$v_5 = (TOI, CAR)$	$r_3 = \{(rg, no)\}$
$C_6 = (v_6, r_4)$	$v_6 = (ENJ, CAR)$	$r_4 = \{(rs, rg), (rs, no), (rg, no)\}$

Tab. 2.2: Contraintes et relations du problème "firme automobile"

2.2 Sémantique

Tout comme dans le cadre de la *logique*, il est necessaire de donner un *sens précis* à un CSP tel qu'il a été définit précédemment.

Etant donné un CSP $P = (X, D, C)$, on appelle **instanciation** $\mathcal{A}$ de $Y = \{x_{y_1}, ..., x_{y_{|Y|}}\} \subset X$ une application qui associe à chaque variable $x_{y_i} \in Y$ une valeur $\mathcal{A}(x_{y_i}) \in d_{y_i}$ (le domaine de x_{y_i}). Etant donné un CSP $P = (X, D, C)$, une instanciation $\mathcal{A}$ de Y **satisfait la contrainte** $c_i = (v_i, r_i)$ de C (noté $\mathcal{A} \models c_i$) ssi $v_i \subset Y$ et $\mathcal{A}(v_i) \in r_i$. A l'opposé, on dira qu'une instanciation $\mathcal{A}$ de Y **viole la contrainte** c_i ssi $v_i \subset Y$ et $\mathcal{A}(v_i) \notin r_i$.

Une **instanciation** est **consistante** si elle ne viole aucune contrainte. De manière plus formelle, étant donné un CSP $P = (X, D, C)$, une instanciation $\mathcal{A}$ des variables de $Y \subset X$ est dite consistante ssi $\forall c_i = (v_i, r_i) \in C, v_i \subset Y$, on a $\mathcal{A} \models c_i$. Une **solution d'un CSP** $\mathcal{S}$ de $P = (X, D, C)$ est une *instanciation consistante* des variables de X. On dit alors que l'instanciation $\mathcal{S}$ satisfait P (noté $\mathcal{S} \models P$). L'ensemble des solutions de P sera noté S_p. Un **CSP** $P = (X, D, C)$ est dit **consistant** si au moins une solution $S_p \neq \emptyset$ existe (*inconsistant* sinon).

Un **instanciation** est **globalement consistante** si elle fait partie d'une solution. Etant donné un CSP $P = (X, D, C)$, une instanciation $\mathcal{A}$ de $Y \subset X$ est dite globalement consistante ssi $\exists \mathcal{S} \in S_p$ telle que $\mathcal{A} \subset \mathcal{S}$. Un CSP est **globalement consistant** si toute instanciation consistante est globalement consistante. L'obtention de la consistance globale implique la résolution du problème.

Les CSPs $P = (X, D, C)$ et $P' = (X, D', C')$ sont dits **équivalents** (noté $P \equiv P'$) ssi $S_p = S_{p'}$. Si nous disposons de deux CSP équivalents, nous pouvons décider de résoudre l'un à la place de l'autre: ils ont le même ensemble de solutions. Ainsi, considérons le CSP P de l'exemple précédent et le CSP P' obtenu à partir de P, en supprimant de d_3 (le domaine de la variable CAR) les valeurs rs et rg. Ces deux CSP ont le même ensemble de solutions et sont donc équivalents, mais P' est intuitivement plus simple que P, le nombre d'instanciations complètes possibles étant plus faible.

Exemple (*firme automobile*) : Considérons le CSP défini précédemment.

$A1 = \{CAR = no, POR = rg, CAP = rg\}$, $\mathcal{A}2 = \{CAR = no, POR = no, CAP = no, TOI = rg, PAR = bl, ENJ = rs\}$ et $A3 = \{CAR = no, POR = no, CAP = no\}$ sont trois *instanciations* du CSP "firme automobile".

$\mathcal{A}1$ *satisfait* la contrainte C_1, mais *viole* les contraintes C_2 et C_3 impliquant les variables de l'instanciation. $\mathcal{A}2$ et $\mathcal{A}3$ satisfont les contraintes impliquées dans leur instanciation.

$\mathcal{A}1$ n'est pas une instanciation consistante, par contre $\mathcal{A}2$ et $\mathcal{A}3$ sont des *instanciations consistantes.*

$\mathcal{A}2$ est une *solution du CSP* "firme automobile", étant une instanciation consistante des variables de X.

Le *CSP* "firme automobile" est donc *consistant* puisqu'au moins une solution existe.

$\mathcal{A}3$ est une instanciation globalement consistante puisqu'elle fait partie d'une solution ($\mathcal{A}2$).

2.3 Représentation sous forme de graphes

Dans un CSP, le problème est représenté par ses contraintes. Une *contrainte* peut être représentée en *intention* ($x \neq y$), en *extension* (liste des couples autorisés ou interdits) ou sous forme de *graphe* (ainsi l'ensemble du problème peut être représenté par un réseau de contraintes).

On peut donner une représentation d'un CSP sous la forme d'un *hyper-graphe* dont les **sommets** seront les variables et dont les **hyper-arêtes** seront les contraintes. Dans le cas de CSP binaires, on pourra se contenter d'un *graphe*. Il s'agit souvent d'une représentation très utilisée, ce qui fait que l'on identifie souvent CSP, hyper-graphe de contraintes et graphe de contraintes (contraintes binaires seulement) à un réseau de contraintes.

2.3.1 Graphe de contrainte

A tout CSP (X, D, C) est associé un **graphe de contrainte** noté $G = (X, C)$ suivant les règles suivantes : (Cf. Fig. 2.1)

- Les sommets de G sont les variables,
- Les arêtes de G sont les contraintes (les relations sont notées en intention), on notera que la contrainte entre deux variables connectées peut être interpréter comme un ordre lexicographique strict, dans le sens des flèches.
- Ce graphe possède n sommets et e arcs.

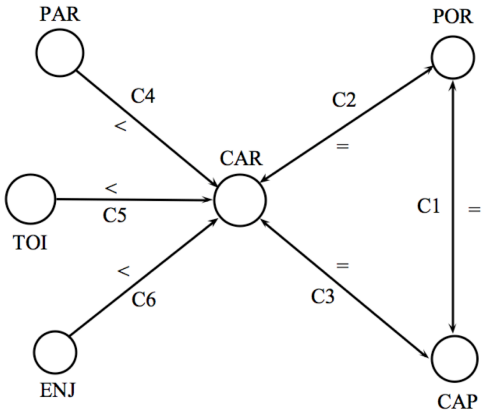


Fig. 2.1: Graphe de contrainte du problème "firme automobile"

2.3.2 Graphe de consistance

A tout CSP (X, D, C) est associé un **graphe de consistance** représentant les relations du CSP suivant les règles suivantes : (Cf. Fig. 2.2)

- Un sommet est associé à chaque valeur a tel que $a \in d_i$.
- Deux valeurs a et b tel que $a \in d_i$ et $b \in d_j$ sont reliées entre elles *ssi* elles vérifient la contrainte $C_{i,j}$,
- Ce graphe possède au plus $(n \times d)$ sommets et $(e \times d)$ arcs.

2.4 Consistance locale (filtrage)

La *consistance* est la propriété liée à la compatibilité entre *valeurs* de *domaine* et *contraintes*. Prouver qu'un CSP est consistant, c'est à dire globalement consistant, est un problème **NP-complet**. Ce problème nous amène à définir des propriétés "*moins fortes*" que la consistance, c'est la *consistance locale*. L'idée est de limiter la propriété de consistance à des *"sous-CSP"* de taille k (variables).

On peut ainsi poser le problème d'établissement de consistance locale ou filtrage en termes plus précis : étant donné un CSP $P = (X, D, C)$, construire un CSP $P' = (X, D', C')$ qui soit équivalent à P et qui soit *k-consistant*. Le CSP P' présente alors l'avantage d'être bien souvent plus rapide à résoudre. De nombreuses propriétés de consistance locale ont été ainsi définies comme la *consistance de noeud*, la *consistance d'arc* ou la *k-consistance*.

2.4.1 Consistance de noeud

Pour chaque contrainte *unaire* C_i (ne concernant qu'une seule variable) on retire du domaine de v_i toutes les valeurs qui ne satisfont pas C_i. On emploie souvent la consistance de noeud pour

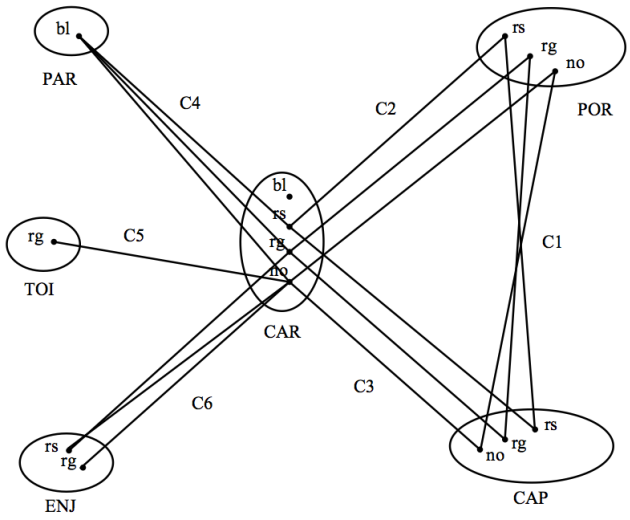


FIG. 2.2: Graphe de consistance du problème "firme automobile"

tester l'*existence*.

Exemple Fig. 2.3 :
Soit V_1, une variable de domaine d_1, tel que $d_1 = \{1, 2, 3\}$ liée à une contrainte C_1, tel que $C_1 = V_1 > 1$. La valeur $1 \in d_1$ ne satisfait pas la contrainte donc on la retire du domaine.

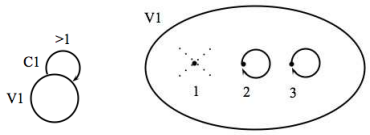


FIG. 2.3: Consistance de noeud d'une variable

2.4.2 Consistance d'arc (AC)

Il s'agit de la plus ancienne et de la plus utilisée des consistances locales. L'idée est simple : si une valeur v du domaine de x_i n'apparait dans aucun des n-uplets d'une contrainte portant sur x_i, il est certain que v ne participera pas à une solution. Soient v_i et v_j deux variables de domaines respectifs d_i et d_j, et liées par une contrainte C_{ij}, on note :

- la valeur b, tel que $b \in d_j$, **support** de la valeur a ,tel que $a \in d_i$, pour la contrainte C_{ij} *ssi* (a, b) vérifie C_{ij} (ou encore $(a, b) \in R_{ij}$).
- la valeur a, tel que $a \in d_i$, **viable** si elle possède au moins un support pour chaque contrainte $C_{ij} \in C$.
- un CSP est **arc consistant** *ssi* aucun domaine $d_i \in D$ n'est vide et si toutes les valeurs de tous les domaines sont *viables*.

Le premier algorithme de filtrage par AC (AC-1) est dû aux recherches de WALTZ an 1972, puis les algorithmes AC-3, AC-4, AC-5, AC-6, AC-7 firent leur apparition.

Exemple Fig. 2.4 : *(firme automobile)*
Dans cet exemple, on réduit le domaine des variables par consistance d'arc de la manière suivante : Tout d'abord, la valeur *bl* de la variable CAR n'a pas de support pour les contraintes affectant la variable donc, on la supprime du domaine de la variable CAR. Ensuite, les valeurs rs et rg du domaine de la variable CAR n'ont pas de support pour la contrainte C_5, elles ne sont pas viables,

donc on les supprime du domaine de la variable CAR. Ces suppressions impliquent que les valeurs rs et rg des domaines des variables POR et CAP n'ont plus de support pour leur contrainte avec la variable CAR, donc on les supprime de leur domaine respectif.

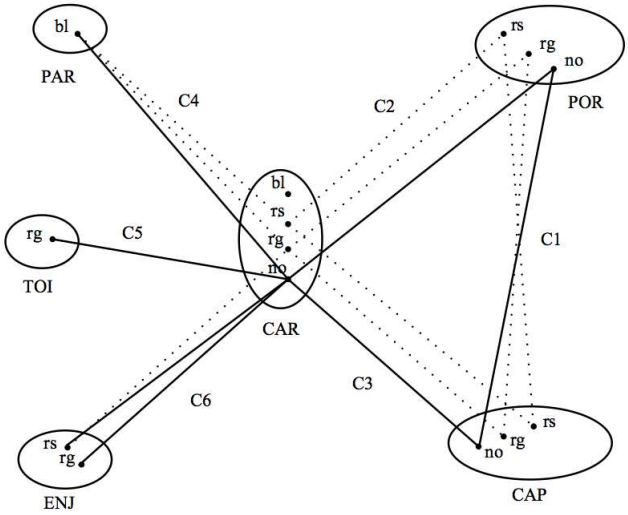


Fig. 2.4: Consistance d'arc du problème "firme automobile"

Une *valeur non-viable* ne peut participer à une quelconque solution. La suppresion des valeurs non-viables ne modifie jamais l'ensemble des solutions, mais la disparition d'une valeur non-viable peut rendre non-viable une valeur qui jusque-là était viable.

L'*arc-consistance* n'assure pas la consistance d'un problème. Dans certains cas, on obtient une solution après l'application de l'AC :

- Si on obtient un domaine vide par AC, alors le CSP est inconsistant,
- Si tous les domaines ont été réduit au singleton alors on obtient une solution du CSP.

Exemple Fig. 2.5 : *(CSP arc-consistant et inconsistant)*
Soit V_1, V_2 et V_3, des variables de domaine d_1, tel que $d_1 = \{1, 2\}$ et liées aux contraintes C_1, C_2 et C_3, tel que $C_1 = V_1 \neq V_2$, $C_2 = V_2 \neq V_3$ et $C_3 = V_3 \neq V_1$. Dans cette exemple, il n'y a pas de solution puisque si V_1 prend la valeur 1, V_2 doit prendre la valeur 2, V_3 doit prendre la valeur 1 et la contrainte C_3 n'est pas satisfaite. Pourtant le problème est arc-consistant (toutes les valeurs des variables sont viables).

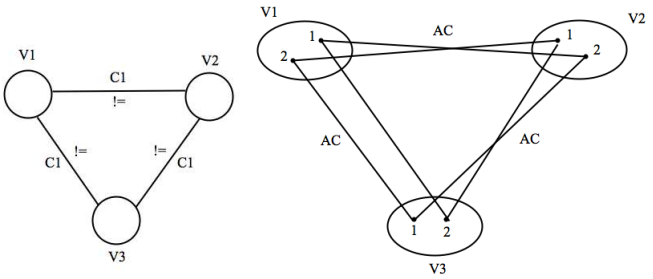


Fig. 2.5: Consistance d'arc d'un CSP arc-consistant et inconsistant

Dans le cas général, l'*arc-consistance* supprime des valeurs, et donc réduit l'espace de recherche (but recherché du filtrage), c'est pourquoi AC est l'une des techniques de filtrage la plus utilisée. Dans le chapître suivant, une étude des algorithmes de type AC est détaillée.

2.4.3 Consistance de chemin (PC)

Un CSP vérifie la consistance de chemin, ssi, pour tout couple de variables (X, Y) et pour toute troisième variable Z, si (x, y) est une instanciation consistante de X et de Y, alors il existe z dans D_z tel que (x, z) et (z, y) soient consistants.

Les premiers algorithmes de consistance de chemin ont été introduits par MONTANARI en 1974 et MACKWORTH en 1977. MOHR et HENDERSON ont proposé PC-3 en 1986. Enfin, HAN et LEE ont proposé PC-4 en 1988. La complexité temporelle est en $\mathbf{\Theta(n^3 \times d^3)}$.

Un arc (i, j) est consistant si le chemin (i, j, i) est *chemin-consistant*. Il existe donc un CSP *arc-consistant* mais non *chemin-consistant* pour l'exemple précédent. De la même manière que l'*arc-consistance* ne suffit pas à résoudre un CSP, la *chemin-consistance* à ses limites.

Exemple Fig. 2.6 : *(CSP chemin-consistant et inconsistant)*
Soient V_1, V_2, V_3 et V_4 des variables de domaine d_1, tel que $d_1 = \{1, 2, 3\}$ et liées aux contraintes C_1, C_2, C_3, C_4, C_5 et C_6 tel que $C_1 = \{V_1 \neq V_2\}$, $C_2 = \{V_2 \neq V_3\}$, $C_3 = \{V_3 \neq V_4\}$, $C_4 = \{V_4 \neq V_1\}$, $C_5 = \{V_1 \neq V_3\}$ et $C_6 = \{V_2 \neq V_4\}$. Dans cette exemple les variables doivent avoir une valeur différente des autres variables en relation par la contrainte. Donc il n'y a pas de solution puisque si V_1 prend la valeur 1, V_2 doit prendre la valeur 2, V_3 doit prendre la valeur 3, et il est impossible de trouver une valeur pour V_4. Pourtant le problème est chemin-consistant.

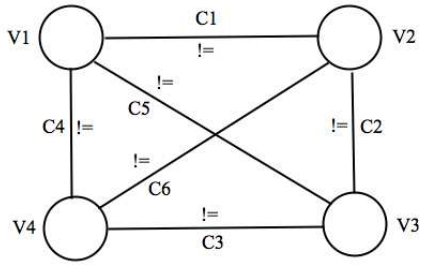


FIG. 2.6: Chemin-consistance d'un CSP chemin-consistant et inconsistant

2.4.4 K-consistance

Un CSP est dit *k-consistant* si pour tout n-uplet de k variables $(x_1, ..., x_k)$, pour toute instanciation $\mathcal{A}$ consistante des $(k-1)$ variables $(x_1, ..., x_{k-1})$, il existe une valeur $v \in d_k$ telle que l'instanciation $\mathcal{A} \cup \{x_k = v\}$ soit consistante.

Un CSP binaire P est k-consistant ssi, pour toute instanciation consistante de $k-1$ variables, il existe, dans le domaine de toute variable non instanciée, une valeur prolongeant cette instanciation en une instanciation consistante de k variables. La consistance de noeud correspond à la 1-consistance. La consistance d'arc, pour les CSP binaires, équivaut à la 2-consistance. La 3-consistance équivaut à la consistance de chemin.

Un algorithme permettant de déterminer la k-consistance en $\mathbf{\Theta(n^k \times d^k)}$ a été proposé par COOPER en 1989.

Chapitre 3

Programmation Par Contraintes

La *Programmation Par Contraintes* fournit le cadre et les méthodes pour la résolution de problèmes définis sur des domaines discrets (entiers, intervalles, ensembles) ou continus. Elle s'applique essentiellement au *problème de satisfaction de contraintes* (CSP) et reçoit la même définition.

Ici le *domaine* de définition des variables est l'ensemble des entiers noté $\mathbb{N}$. Les *contraintes* sur les variables sont binaires et la définition des valeurs des variables se fait en *extension*.

On rappelle que trouver une solution à un problème en PPC consiste à affecter une valeur à chaque variable, de telle sorte que la totalité des contraintes soient satisfaites $(v_1 C_1 v_2) \wedge (v_2 C_2 v_3) \wedge ...$

3.1 Méthode de résolution standard

Pour construire une solution, on considère un espace où chaque état est une instanciation $\mathcal{A}$, les règles de production permettant d'étendre l'instanciation $\mathcal{A}$ sur une variable avec toutes les valeurs possibles de son domaine. On construit un *arbre de recherche* où :

- Les noeuds sont les variables,
- Les branches sont des réductions de domaine.

Les feuilles de l'arbre ainsi exploré sont des instanciations complètes dont on pourra aisément vérifier si elles sont, ou non, des solutions du CSP traité. Le parcours se réalise toujours en profondeur d'abord : un parcours en largeur d'abord est inintéressant puisque les solutions sont à une profondeur connue. Tous les éléments du produit cartésien sont générés. Ne sont retenus que ceux qui satisfont l'ensemble des contraintes. Ainsi la totalité des solutions sont entièrement *construites* avant d'être *testée*s.

Exemple Fig. 3.1 : *(CSP 2 variables, 2 valeurs, 1 contrainte)*

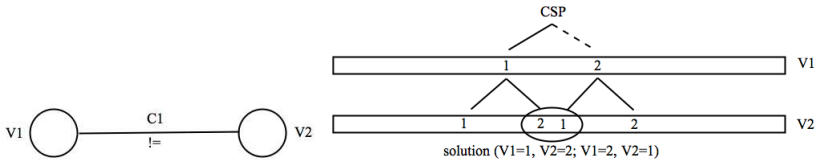


FIG. 3.1: Arbre appliqué au "generate-and-test"

Cette méthode standard de résolution dite "generate and test" est fortement combinatoire $\mathbf{\Theta(e \times d^n)}$. Il est donc nécessaire d'améliorer cette résolution (s'approcher d'une complexité polynomiale) à l'aide d'un certain nombre de techniques :

- Avant la recherche :
  - appliquer des *techniques de filtrage* (consistance d'arc, de chemin...)
  - guider la recherche par des *heuristiques d'ordonnancement* (ordre de choix des variables, des valeurs et des tests des contraintes)
- Pendant la recherche :
  - appliquer des *stratégies de recherche optimisées* (Backtrack, Forward Checking, MAC...)

3.2 Méthodes de filtrage

Le *filtrage* consiste en l'élimination d'éléments dont on est assuré qu'ils ne peuvent figurer dans une quelconque solution (valeurs ou n-uplets de relation). L'utilisation d'une technique de filtrage pour l'obtention de la *consistance* fait appel à la propagation de contraintes à l'aide d'*algorithmes de point fixe* (itérer jusqu'à ce que la propriété désirée soit établie).

Un filtrage tend vers la simplification du problème par réduction de l'espace de recherche et dans certains cas, il sert à détecter l'*inconsistance*.

3.2.1 Propagation de contraintes

La propagation de contraintes consiste à réduire le domaine des variables impliquées dans une contrainte, dans le but de *rétablir la consistance*. La représentation d'un problème CSP à l'aide d'un graphe de consistance nous donne ainsi un support de travail.

Un *réseau de variables* communiquent entre-elles par le seul intermédiaire des contraintes :
- Lorsque le domaine d'une variable est réduit (*consistance*), on *reveille* les contraintes qui impliquent cette variable,
- Lorsqu'une contrainte est réveillée, on examine s'il est possible de faire des *déductions* sur le domaine des autres variables de la contrainte.

La propagation de contraintes converge toujours vers soit :
- Une *contradiction* : une variable a un domaine vide, ce qui signifie que le problème n'a pas de solution,
- Un *point fixe* : toutes les déductions possibles ayant étés faites, il n'y a plus lieu de réduire les domaines des variables.

A l'issue de ce processus, trois cas de figure se présentent :
- Une contradiction à été identifiée, il faut remonter d'un niveau dans l'arbre de recherche (*backtracking*),
- Le domaine de chaque variable est un singleton, une *solution* a été trouvée,
- Le domaine d'une variable comporte plusieurs valeurs, il faut poursuivre l'énumération par une descente dans l'*arbre de recherche*.

3.2.2 Revise (consistance d'arc)

De nombreuses procédures d'établissement de la consistance d'arc ont été exhibées. On se limite ici au calcul de contraintes *unaires*, induites par des "sous-CSP" définis par des couples de variables. Les algorithmes les plus anciens s'appuient sur une procédure très simple, appelée **Revise.** $Revise(C_{ij}, d_i, d_j)$ retire du domaine de V_i toutes les valeurs sans support pour la contrainte C_{ij} et retourne un booleen indiquant si le domaine de x_i a été modifié.

Cette procédure est appliquée sur tous les couples de variables. Le problème essentiel est que la réduction du domaine d_i peut entraîner l'apparition de nouvelles valeurs arc-consistantes sur d'autres variables reliées par une contrainte à x_i , et qu'une seule irération ne suffit généralement pas. (Cf. Algorithme 1)

La complexité dans le pire des cas est $\mathbf{\Theta(d^2)}$.

Algorithme 1 : Revise ($C_{i,j}$: Contrainte; d_i,d_j : Domaine)

```
1 modification ← false;
2 foreach v ∈ d_i do
3 |   if ∄v' ∈ d_j|(v, v') ∈ rel(C_{i,j}) then
4 |   |   d_i ← (d_i\{v});
5 |   |   modification ← true;
6 return modification;
```

3.2.3 AC-1

Le filtrage par consistance d'arc AC-1 consiste à appliquer la consistance d'arc $Revise(C_{ij}, d_i, d_j)$ sur toutes les contraintes (*propagation de contraintes*) jusqu'à ce que plus aucune réduction ne s'effectue. De cette manière on palie au problème de la simple application de **Revise** sur tous les couples de variables. (Cf. Algorithme 2)

Algorithme 2 : AC1(Q : ens. Contraintes)

```
1 change ← true;
2 while change do
3 |   change ← false;
4 |   foreach C_{ij} ∈ Q do
5 |   |   change ← change ∪ Revise(x_i, x_j);
```

La compléxité d'AC-1 est en $\mathbf{\Theta(n \times e \times d^3)}$. Si une réduction est faite, on examine toutes les contraintes, au lieu de n'examiner que les variables directement concernées.

3.2.4 AC-3

Si une valeur a de D_i n'a pas de support dans D_j pour une contrainte C_{ij}, alors a est retirée de d_i. AC-3 examine les répercutions de ce retrait (v_i, a) sur les domaines de tous les voisins v_j de v_i $(j \neq i)$ (*propagation de contraintes*).

Soient les variables V_i et V_j de domaines respectifs d_i et d_j, la contrainte C_{ij}, les valeurs $x \in d_i$ et $y \in d_j$, telles que :
- x parcourt toutes les valeurs possibles de la variable V_i,
- y parcourt toutes les valeurs possibles de la variable V_j,
- après parcours, si (x, y) viole la contrainte, alors on enleve y du domaine de la variable V_j,
- On répète ces vérifications pour toutes les contraintes impliquant la variable V_j.

Il y a nécéssité de gérer une file comprenant les variables affectées par la réduction du domaine, pour eviter l'application inutilement répétée de **Revise**. On peut montrer que si d est la taille maximale des domaines des variables et e le nombre de contraintes, alors :
- Une contrainte C_{ij} est examinée au plus d fois (car une contrainte est examinée à cause de la suppression d'une valeur dans le domaine de sa seconde variable, et qu'on ne peut pas supprimer plus de d valeurs du domaine)
- Il y a donc au plus $(d \times e)$ examens de contraintes
- Chaque examen de contrainte necessite d^2 verifications

– L'algorithme a donc une complexité en $\mathbf{\Theta(e \times d^3)}$.

AC-3 est initialement dû aux recherches de MACKWORTH en 1977, puis revu par MACKWORTH et FREUDER en 1985. (Cf. Algorithme 3)

Algorithme 3 : AC3(Q : ens. Contraintes)

```
1 L ← liste des paires (x_i, x_y)|∃ une contrainte entre x_iet x_j;
2 while L est non vide do
3 |   choisir et supprimer dans L une paire (x_i, x_j);
4 |   if revise(x_i, x_j) = true then
5 |   |   L ← L ∪ {(x_k, x_i)|∃ une contrainte entre x_k et x_i};
```

3.2.5 AC-3r(m)

Pour un algorithme établissant la consistance d'arc (AC), un **support résiduel**, ou résidu, est un support qui a été trouvé et enregistré lors d'une exécution de la procédure qui détermine si une valeur est supportée par une contrainte. (ici on parlera de **revise3rm**)

L'algorithme élémentaire AC3 peur être affiné en exploitant les résidus comme suit : avant de rechercher un support pour une valeur en partant de zéro, la validité du résidu associé à la valeur est testée. On obtient alors un algorithme appelé AC-3r, et lorsqu la multi-directionnalité est exploitée, un algorithme appelé AC-3r(m).

L'idée est que lorsqu'un support t est trouvé, il peut être **enregistré** pour toutes valeurs apparaissant dans t. Par exemple, considérons une contrainte binaire C telle que $vars(C) = \{X, Y\}$. Si (a, b) est trouvé dans C lorsqu'on recherche un support de (X, a) dans (Y, b), dans les deux cas, il peut être enregistré comme étant le dernier de (X, a) dans C et le dernier support trouvé de (Y, b) dans C. Lorsque le dernier support n'est plus valide, il est nécessaire de chercher un nouveau support à partir de zéro.

On garde ces informations dans $Supp[C, Y, t[Y]]]$. $C(t)$ est un test de consistance ou $C(\perp)$ retourne $faux$ et $setNextTuple(C, X, a, t)$ retourne le plus petit tuple t' construit à partir de C tel que $t < t'$ et $t'[X] = a$. (Cf. Algorithme 4, Algorithme 5))

AC-3r(m) a une complexité spatiale en $\mathbf{\Theta(e \times d)}$ et une complexité temporelle, dans le pire des cas, en $\mathbf{\Theta(e \times d^3)}$.

Algorithme 4 : revise3rm(C : ens. Contraintes, X : ens. Variables)

```
1  nbElements ← |dom(X)|;
2  foreach a ∈ dom(X) do
3  |   if supp[C, X, a] est valide then
4  |   |   continue;
5  |   t ← seekSupport3(C, X, a);
6  |   if t = ⊤ then
7  |   |   éliminer a de dom(X);
8  |   else
9  |   |   foreach Y ∈ vars(C) do
10 |   |   |   supp[C, Y, t[Y]] ← t
11 |   return (nbElements ≠ |dom(X)|);
```

Algorithme 5 : seekSupport3(C : ens. Contraintes, X : ens. Variables, a : valeur)

```
1 t ←⊥;
2 while t ≠ ⊤ do
3 |   if C(t) then
4 |   |   return t;
5 |   t ← setNextTuple(C, X, a, t);
6 |   return t;
```

3.2.6 Applications restreintes à Revise

La figure 2.4 montre le CSP obtenu après application d'une procédure d'établissement de l'arc-consistance au CSP "firme automobile". Dans ce cas précis, le CSP obtenu est *globalement consistant*. La recherche d'une solution via l'algorithme *Backtrack* est alors gloutonne : elle s'effectue sans que l'on ait jamais à effectuer de retour-arrière. Rappelons qu'il ne s'agit pas d'un cas général : la figure 2.5 montre un CSP arc-consistant mais inconsistant.

Malgré le faible coût d'établissement de l'arc-consistance, on a défini un certain nombre d'algorithmes qui établissent un sous-ensemble de l'arc-consistance. Comme nous le verrons, ils sont surtout utilisés en collaboration avec l'algorithme *Backtrack*.

Si l'on considère que les n variables d'un CSP sont ordonnées, on peut définir des applications restreintes de la procédure *Revise* à partir d'une variable x_i donnée : (illustration des restrictions Fig. 3.2)

- **Check backward** : on applique la procédure *Revise* à tous les couples (x_i, x_k) tels que $1 \leq k \leq i$.
- **Check forward** : on applique la procédure *Revise* à tous les couples (x_k, x_i) tels que $i \leq k \leq n$.
- **Partial look future** : on applique la procédure *Revise* à tous les couples (x_j, x_k) tels que $i \leq j < k \leq n$.
- **Full look future** : on applique la procédure *Revise* à tous les couples (x_i, x_k) tels que $1 \leq j \neq k \leq n$.

3.3 Heuristiques d'ordonnancement

Les heuristiques d'ordonnancement permettent de *préparer le parcours de l'arbre de recherche* et sont classées selon différents ordres influant sur le parcours :

- **Ordre d'instanciation des variables** (ou *ordre vertical*), cet ordre peut avoir un impact très important sur l'efficacité de l'algorithme puisqu'il décide du moment à partir duquel la satisfaction d'une contrainte peut être vérifiée et influe donc sur la taille de l'espace exploré.
- **Ordre d'instanciation des valeurs** pour chaque variable (ou *ordre horizontal*), une telle heuristique n'aura aucun effet sur l'éfficacité de la recherche si le CSP est inconsistant ou si l'on cherche toutes les solutions puisqu'on devra alors explorer toute la largeur de l'arbre. Les heuristiques d'ordre horizontal les plus efficaces sont dépendantes du domaine.
- **Ordre de vérification des contraintes** à chaque noeud, il est possible après l'instanciation d'une nouvelle variable, de vérifier la satisfaction de plusieurs contraintes. Il n'influence pas la taille de l'espace exploré, mais permet de diminuer le nombre de test de satisfaction de contraintes (ou *constency checks*). Le *first fail principle* conduit à vérifie en premier les contraintes les moins satisfiables.

3.3.1 First fail principle

Cette heuristique est souvent introduite pour choisir un **ordre vertical** "pertinent". Elle est fondée sur le principe de l'*échec d'abord* qui cherche à faire apparaître le plus rapidement possible des violations de contraintes. Elle peut-être utilisée *statiquement* (avant l'éxecution) pour construire

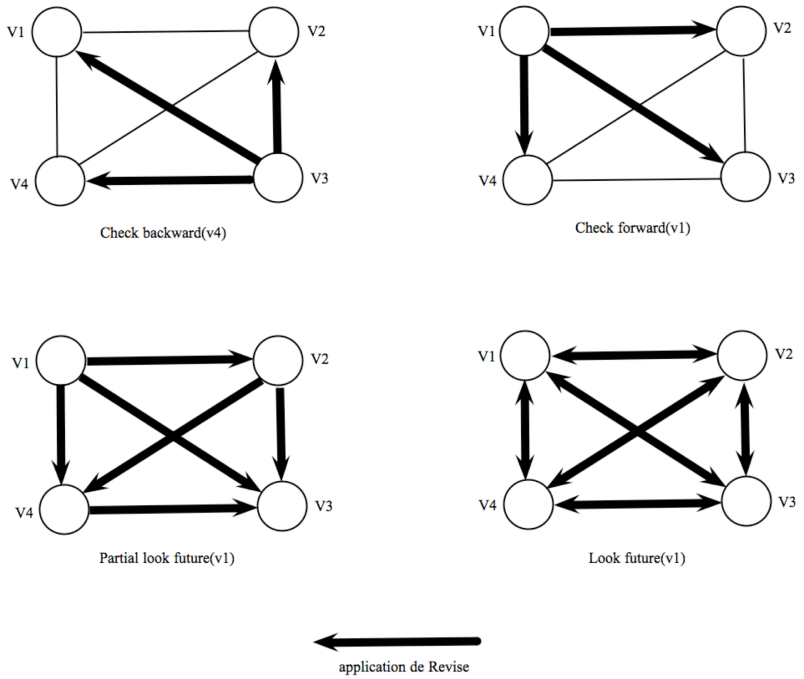


Fig. 3.2: Arc-consistances dégradées

un ordre qui sera utilisé de la même façon dans toutes les branches ou *dynamiquement,* en calculant pendant l'exécution de l'algorithme, et à chaque noeud, la prochaine variable qui sera instanciée :

- **Statiquement** :
  - **Cardinalité maximum** : on choisit une première variable aléatoirement (ou celle de degré maximum), la prochaine variable liée sera celle reliée (par des contraintes) au plus grand nombre de variables déjà définies dans l'instanciation courante.
  - *Degré de la variable* vis à vis des variables non instanciées.
  - *Variable participant au plus grand nombre de contraintes.*
  - *Variable impliquée dans la contrainte la moins satisfiable.*
  - *Ordre de largeur minimal.*
- **Dynamiquement** :
  - **Réarrangement dynamique** : on choisit la variable ayant le nombre minimum de valeurs vérifiant une propriété de consistance locale donnée.

3.4 Stratégies de recherche

Le but de cette étape est d'améliorer la méthode de résolution standard "*generate and test*". Il existe plusieurs niveaux de filtrages en cours de résolution :

- Backtrack, amélioration simpliste du *"generate and test"*
- Forward Checking (FC), valeurs directement inconsistances,
- Real Full Lookhead (RFL MAC),
  - $FC+AC$ (AC est appliqué avant la recherche et à chaque étape).

3.4.1 Backtrack

Le *retour en arrière chronologique*, dit Backtrack est une amélioration triviale du *generate and test* (vu en début de chapître) consistant à vérifier la satisfaction de chaque contrainte dès que possible, nous conduisant donc au *test and generate*. Il *construit* chaque instanciation en maintenant l'invariant : toutes les variables instanciées sont compatibles entre-elles.

Il suffit en fait qu'une contrainte soit violée pour qu'aucune des instanciations filles d'un noeud ne puisse être consistante. Or, il est possible de vérifier qu'une contrainte est violée dès que toutes les variables sur lesquelles elle porte ont été instanciées : Toute affectation non consistante est non globalement consistante

Principe : à chaque étape $(1, ..., k, ..., n)$:
- On instancie une nouvelle variable X_k,
- On élimine les valeurs devenues impossibles des domaines D_i pour $i > k$,
- Si un domaine D_i devient vide, il y a alors retour arrière sur la variable précédente.

Cette approche tout comme la *méthode de résolution standard* est fortement combinatoire $\mathbf{\Theta(d^n)}$.

Exemple Fig. 3.3 : *(CSP 2 variables, 2 valeurs, 1 contrainte, ordre d'instanciation : V1-V2)*

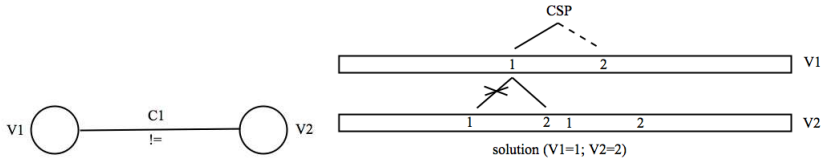


FIG. 3.3: Arbre appliqué au "backtrack"

3.4.2 Sophistication de l'algorithme Backtrack

L'algorithme Backtrack souffre de nombreux défauts : parcours aveugle de l'espace sans prise en compte d'informations "évidentes" sur la non consistance globale d'instanciations partielles, découverte répétée des mêmes inconsistances locales...

Pour sophistiquer l'algorithme Backtrack, il parait évident de modifier l'algorithme lui-même en tentant de détecter, à moindre coût et le plus rapidement possible, la non consistance globale de l'instanciation courante. On sait alors qu'il est possible d'aboutir à une solution dans cette branche de l'arbre et on peut continuer l'exploration dans une autre branche. Les différentes stratégies existantes peuvent être séparées en :

1. **Schémas rétrospectifs** : (ou *look-back schemes*) qui profitent de l'exploration arborescente effectuée durant l'exécution de l'algorithme, et en particulier des violations de contraintes, pour démontrer, à moindre coût, qu'une *instanciation strictement contenue* dans l'instanciation courante, et la plus petite possible, est non globalement consistante. Deux catégories d'algorithmes, finalement assez proches, rentrent dans ce cadre général :

   a) Le *Backtrack intelligent* utilise immédiatement la preuve qu'une instanciation $\mathcal{A}$, incluse dans l'instanciation courante, est non globalement consistante en effectuant un *retour-arrière non chronologique* qui supprime de l'espace exploré, simplement (et sans consommation mémoire) une partie des instanciations qui contiennent $\mathcal{A}$;

   b) La *mémorisation de contraintes* (aussi appelée *apprentissage*) va plus loin en mémorisant dans e CSP le fait que $\mathcal{A}$ est non globalement consistante sous la forme d'une contrainte interdisant $\mathcal{A}$ (qui est *induite* par le CSP). Cette mémorisation s'ajoute au retour-arrière intelligent. Il faut alors faire un judicieux compromis entre la mémoire consommée et les effets bénéfiques de la mémorisation ;

2. **Schémas prospectifs** : (ou *look-ahead schemes*) qui modifient le mécanisme d'exploration habituel de l'algorithme *Backtrack* afin de détecter *a priori* les instanciations non globalement consistantes. On va donc déclencher une exploration "annexe" après chaque instanciation. Là encore, un judicieux compromis doit être effectué entre le coût d'une exploration plus large et les effets bénéfiques de celle-ci.

Les *méthodes prospectives* sont, en général, *les plus efficaces*. La majorité des langages et systèmes logiciels s'appuyant sur le cadre CSP (en particulier les langages de programmation par contraintes) utilisent ces techniques pour résoudre le problème. C'est pourquoi nous ne développerons pas les schémas rétrospectifs dans les sections suivantes.

L'algogorithme *Backtrack* est simplement modifié en remplaçant la vérification de la consistance de l'instanciation partielle courante par l'établissement d'une *propriété de consistance locale*. Il y a retour-arrière dès que la propriété de consistance locale n'est plus vérifiée. Il est alors nécessaire de "défaire" les effets du filtrage effectué précédemment.

L'essentiel des algorithmes utilisés s'appuient sur les versions "dégradées" de la consistance d'arc que nous avons présentées précédemment. Chaque instanciation peut donc supprimer des valeurs des domaines d'autres variables. Il y a retour-arrière dès qu'un domaine est vide. Selon le niveau d'arc-consistance utilisé, on aboutira aux algorithmes :

- **Backtrack** ou *test and generate* : emploi d'un *check backward* après chaque liaison ;
- **Forward-checking** : emploi d'un *check forward* après chaque liaison ;
- **Partial look-ahead** : emploi d'un *partial look future* après chaque liaison ;
- **Full look-ahead** : emploi d'un *look future* après chaque liaison ;
- **Real full look-ahead** : établissement de la *consistance d'arc complète* après chaque instanciation

Dans les quatre derniers cas, on parle alors d'algorithmes de type *choix et propagation*, qui permettent d'obtenir une efficacité très supérieure à celle du classique *Backtrack*. Selon les tests très limités, le *forward-checking* semble être le compromis le plus interessant. Cependant, la majorité des langages de programmation avec contraintes ont fait le choix d'établir des consistances moins dégradées (généralement proche de l'arc-consistance totale), pouvant éventuellement s'ajuster suivant la nature des contraintes. (Cf. Fig. 3.4)

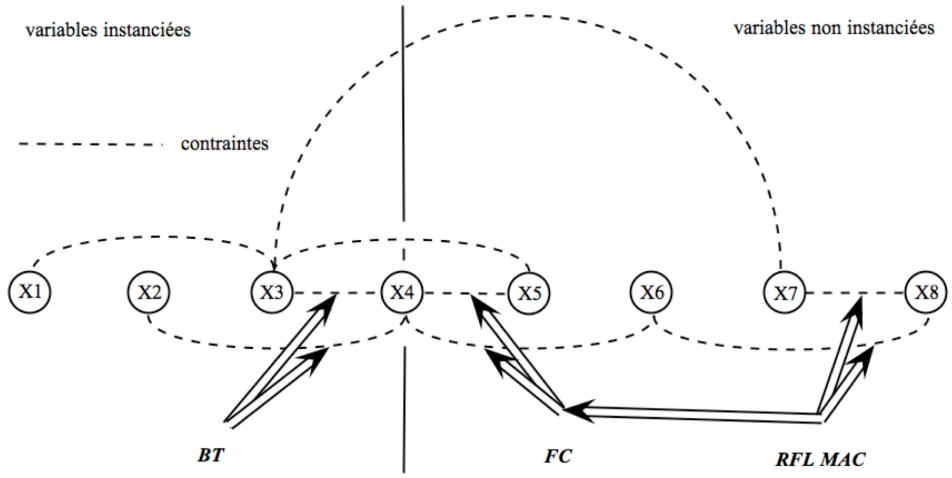


Fig. 3.4: Comparaison des stratégies de recherche

3.4.3 Forward Checking (FC)

Le *Forward checking* construit chaque instanciation en maintenant l'invariant :

- Toutes les variables instanciées sont compatibles entre-elles,
- Pour toute variable non instanciée, il existe au moins une valeur consistante avec les variables déjà instanciées.

Cet algorithme reste le plus simple à réaliser. L'algorithme est incarné dans la procédure récursive **FC** qui prend en argument une séquence de variable V à instancier (initialement X en entier) et une instanciation $\mathcal{A}$ (initialement vide). Elle utilise une procédure auxiliaire appelée *check-forward*. Nous supposerons que la procédure *Push*, qui prend en argument une séquence de variables, empile les domaines des variables de la séquence sur une pile et que la procédure *Pop* les dépile. (Cf. Algorithme 6 et 7)

Le *Forward-Checking* est dû aux recherches de HARALICK en 1980.

Algorithme 6 : FC(V : ens. Variables, $\mathcal{A}$: une instanciation)

```
1 if V = 0 then
2 |  A est une instanciation;
3 else
4 |  x_i in V;
5 |  foreach v in d_i do
6 |  |  Push(V\{x_i});
7 |  |  if checkForward(x_i, v, V) then
8 |  |  |  F(V\{x_i}, A ∪ {x_i ← v});
9 |  |  Pop(V\{x_i});
```

Algorithme 7 : checkForward(x_i : Variable, v : une valeur de x_i, V : ens. Variables)

```
1 consistant = true;
2 foreach x_j in V\{x_i} do
3 |  while consistant do
4 |  |  foreach v' in d_j do
5 |  |  |  if {x_i → v, x_j → v'} est non consistante then
6 |  |  |  |  d_j ← d_j\{v'};
7 |  |  if d_j = ∅ then
8 |  |  |  consistant ← false;
9 return consistant;
```

3.4.4 Real Full Look-ahead (RFL MAC)

L'algorithme MAC a pour objectif de résoudre des instances de CSP et réalise une *recherche en profondeur d'abord avec retours-arrières*. A chaque étape de la recherche, une assignation de variable est effectuée suivie par un processus de filtrage qui correspond à établir la consistance d'arc.

l'étape de filtrage, quand on a instancié la k-ième variable, correspond à la suppression des valeurs des domaines qui ne vérifient pas la **consistance d'arc**. On appele aussi cette méthode MAC (Maintaining Arc Consistency) pour *maintenance de la consistance d'arc*.

Deuxième partie

Modélisation

Chapitre 4

Unified Modeling Language

Aprés avoir défini précisement les *problèmes de satisfaction de contrainte* et ses composantes (liées au sujet du Travail d'Etude et de Recherche) dans la partie précédente, nous allons maintenant les *modéliser*. Cette phase très importante va nous permettre de définir les structures et le *modèle* du logiciel à implémenter pour permettre une conception et une évolution du logiciel plus aisée.

En informatique, un modèle a pour objectif de **structurer les données, les traitements, et les flux d'informations entre entités**. Les modèles informatiques développés dans les années 1970 étaient tous du type entité-relation. On peut citer SSADM (Structured Systems Analysis and Design Method) et MERISE. Ces modèles comportaient en général trois composantes principales :

- Données : modèle de données
- Traitements : modèles de traitements
- Flux (réseau) : diagrammes ou graphes de flux (voir diagramme de flux de données).

On a traditionnellement distingué trois niveaux de préoccupation : *conceptuel*, *logique* (ou organisationnel) et *physique*. Ces modèles ont généralement été appliqués à l'échelle des applications, voire des domaines, mais rarememnt à l'échelle des entreprises, de sorte que l'on trouvait des incohérences d'un domaine à l'autre dans une même entreprise. D'où des interfaces difficiles à établir lorsque les données n'étaient pas définies de le même façon d'un domaine à l'autre.

Dans les années 1990, la nécessité de remplacer ou de rénover les applications affectées par le problème de datation (voir passage informatique à l'an 2000) a entraîné la mise en oeuvre de progiciels de gestion intégrés à l'échelle des entreprises. Les données et applications ont été mises en cohérence souvent en fonction de la structure de ces progiciels, qui en général ont été conçus dans l'esprit des modèles *entité-relation*.

Le **Unified Modeling Language** (UML) a permis de définir un *langage commun* pour que ces projets soient menés à bien de la façon la plus cohérente possible entre toutes les méthodes qui avaient été employées.

4.1 Notion d'objet

Un **objet** modélise toute entité identifiable, concrète ou abstraite, manipulé par l'application. Il réagit à certains messages de l'extérieur ce qui détermine son *comportement*. Son comportement diffère selon son état interne.

Une **classe** est une structure logique et informatique représentant un ou plusieurs objets. Un objet est une *instance* d'une classe. Un *attribut* représente l'état interne d'un objet. Les méthodes sont les fonctionnalités proposées par l'objet.

L'**encapsulation** est le regroupement des données au sein d'une même structure. L'**héritage** définit une classe à partir d'une autre. Le **polymorphisme** permet à un objet de prendre plusieurs

formes. La **surcharge** permet à une même méthode d'avoir plusieurs signatures. La **redéfinition** est une méthode définie dans des classes différentes et ayant la même signature.

4.2 Définition

UML (en anglais Unified Modeling Language, « langage de modélisation unifié ») est un **langage graphique** de modélisation des données et des traitements. C'est une formalisation très aboutie et non-propriétaire de la modélisation objet utilisée en *génie logiciel*.

Il est l'accomplissement de la fusion des précédents langages de modélisation objet : Booch, OMT (Object Modeling Technique) et OOSE (Object Oriented Software Engineering). Il fût principalement issu des travaux de Grady Booch, James Rumbaugh et Ivar Jacobson. UML est un standard défini par l'OMG (Object Management Group). L'OMG travaille actuellement sur la version UML 2.1.

Le *formalisme* UML est composé de 13 types de diagrammes (9 en UML 1.3). UML n'étant pas une méthode, leur utilisation est laissée à l'appréciation de chacun, même si le diagramme de classes est généralement considéré comme l'élément central d'UML. De même, on peut se contenter de modéliser seulement partiellement un système, par exemple certaines parties critiques. Les 13 diagrammes UML sont dépendants hiérarchiquement et se complètent (Cf. Fig. 4.1) :

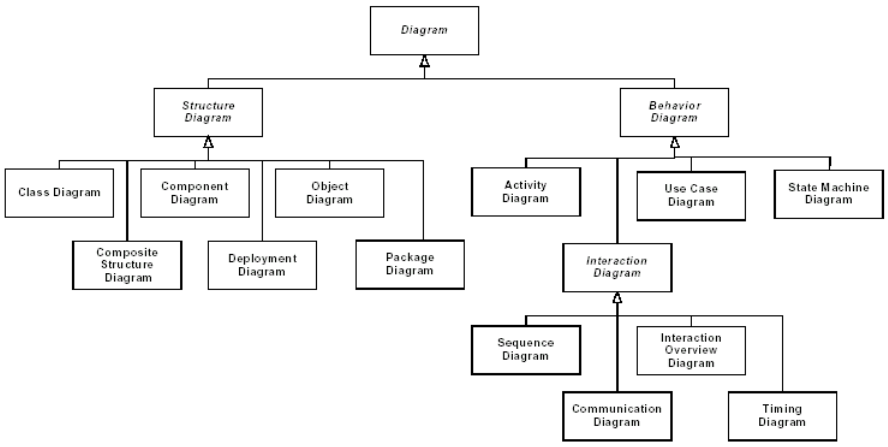


Fig. 4.1: Hiérarchie des diagrammes UML

4.3 Diagrammes

Les diagrammes étant une notion centrale de l'UML, on peut exhiber trois ensembles les regroupant : les diagrammes *structurels*, les diagrammes *comportementaux* et les diagrammes d'*interaction*. Aussi dans le cadre de notre sujet, seuls les diagrammes nous intéressant seront présenter plus en détails.

Les **diagrammes structurels** ou *diagrammes statiques* (cf. Structure Diagram) comportent différents types de diagramme :

- *Diagramme de classes* (cf. Class diagram) : il représente les classes intervenant dans le système.
- *Diagramme d'objets* (cf. Object diagram) : il sert à représenter les instances de classes (objets) utilisées dans le système.

- *Diagramme de composants* (cf. Component diagram) : il permet de montrer les composants du système d'un point de vue physique, tels qu'ils sont mis en oeuvre (fichiers, bibliothèques, bases de données...)
- *Diagramme de déploiement* (cf. Deployment diagram) : il sert à représenter les éléments matériels (ordinateurs, périphériques, réseaux, systèmes de stockage...) et la manière dont les composants du système sont répartis sur ces éléments matériels et interagissent avec eux.
- *Diagramme des paquetages* (cf. Package Diagram) : Un paquetage étant un conteneur logique permettant de regrouper et d'organiser les éléments dans le modèle UML, le Diagramme de paquetage sert à représenter les dépendances entre paquetages, c'est-à-dire les dépendances entre ensembles de définitions.
- *Diagramme de structure composite* (cf. Composite Structure Diagram) : permet de décrire sous forme de boîte blanche les relations entre composants d'une classe.

Les **Diagrammes Comportementaux** ou *Diagrammes dynamiques* (cf. Behavior Diagram) comportent différents types de diagramme :

- *Diagramme des cas d'utilisation* (use-cases) (cf. Use case diagram) : il permet d'identifier les possibilités d'interaction entre le système et les acteurs (intervenants extérieurs au système), c'est-à-dire toutes les fonctionnalités que doit fournir le système.
- *Diagramme états-transitions* (cf. State Machine Diagram) : permet de décrire sous forme de machine à états finis le comportement du système ou de ses composants.
- *Diagramme d'activité* (cf. Activity Diagram) : permet de décrire sous forme de flux ou d'enchaînement d'activités le comportement du système ou de ses composants.

Les **diagramme d'interactions** (cf. Interaction Diagram) comportent différents types de diagramme :

- *Diagramme de séquence* (cf. Sequence Diagram) : représentation séquentielle du déroulement des traitements et des interactions entre les éléments du système et/ou de ses acteurs.
- *Diagramme de communication* (cf. Communication Diagram) : représentation simplifiée d'un diagramme de séquence se concentrant sur les échanges de messages entre les objets.
- *Diagramme global d'interaction* (cf. Interaction Overview Diagram) : permet de décrire les enchaînements possibles entre les scénarios préalablement identifiés sous forme de diagrammes de séquences.
- *Diagramme de temps* (cf. Timing Diagram) : permet de décrire les variations d'une donnée au cours du temps.

4.3.1 Diagramme des cas d'utilisation

Le diagramme des cas d'utilisation représente le **comportement d'un système** (sous-système, classe, composant) *tel qu'un utilisateur extérieur le voit.* (Cf. Tab. 4.1, Fig. 4.2)

Concept	Fonction
Acteur	Rôle joué par une personne physique, un processus ou une chose qui interagit avec le système
Cas d'utilisation	Fonctionnalité visible depuis l'extérieur du système
Interaction	Chemin de communication entre un acteur et un cas d'utilisation
Relation d'inclusion	Le cas d'utilisation inclus va se produire à un moment ou un autre dans le cas d'utilisation de «départ»
Relation d'extension	Le cas d'utilisation étendu peut se produire en fonction du choix de l'acteur
Relation de généralisation	Un cas d'utilisation spécialisé représente une forme particulière du cas d'utilisation dont il «hérite».

TAB. 4.1: Concepts du diagramme des cas d'utilisation

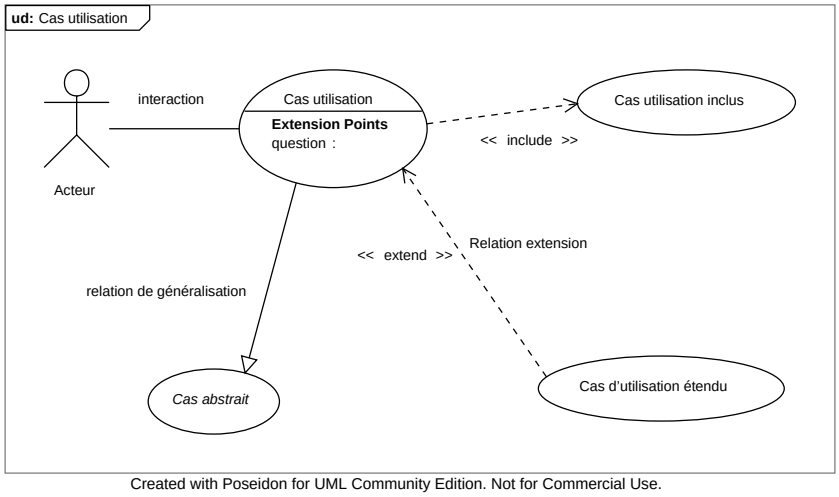


Fig. 4.2: Formalisme des concepts du DSU

4.3.2 Diagramme de séquence

Le diagramme de séquence montre les **intéractions entre objets** correspondant à un cas d'utilisation, à une opération ou à tout autre entité comportementale. (Cf. Tab. 4.2, Fig. 4.3)

Concept	Fonction
Ligne de vie	Durée de vie d'un objet. La création d'un objet peut apparaître ou avoir été effectuée dans un autre diagramme où au chargement du logiciel. La destruction d'un objet, symbolisée par une croix terminant la ligne de vie, peut apparaître ou non en cas d'objet persistant
Activation	Durée pendant laquelle l'objet est en cours d'utilisation (par une de ses méthodes) et/ou en attente du résultat de l'appel d'une méthode d'un autre objet
Appel synchrone	Utilisation d'une méthode d'un objet pour lequel l'objet/l'acteur appelant attend une réponse ou la fin de l'exécution
Appel asynchrone	Exécution d'une méthode de l'objet appelé sans attendre la réponse et l'objet/l'acteur appelant continue son «travail»
Fragment	Partie du diagramme de séquence ayant un rôle particulier et pouvant être constitué de sous-fragments

Fragment	Fonction
conditionnel (alt)	Fragment composé de deux sous-fragments ou plus possédant chacun une condition de garde indiquant quel fragment sera exécuté
optionnel (opt)	Cas particulier du fragment conditionnel dans lequel il n'y a qu'un seul sous-fragment, celui qui sera exécuté si la condition est vraie ou omis sinon
parallèle (par)	il est composé de deux sous-fragments ou plus qui sont tous exécutés simultanément
boucle (loop)	il est composé d'un sous-fragment qui sera exécuté jusqu'à ce que la condition de garde soit vraie

Tab. 4.2: Concepts du diagramme de séquence

4.3.3 Diagramme d'activité

Un diagramme d'activité est un graphe de noeuds et de transitions décrivant une **action complexe** dont les activités la composant peuvent être séquentielles et/ou parallèlles. (Cf. Tab. 4.3, Fig. 4.4)

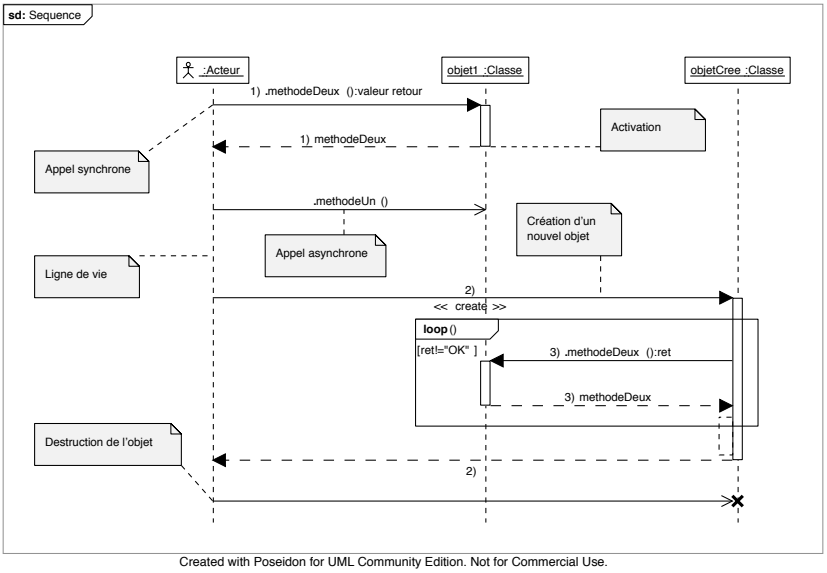


FIG. 4.3: Formalisme des concepts du DSE

Concept	Fonction
Activité	Spécification d'un comportement exécutable
Flot d'activités	«Exécution séquentielle» de plusieurs activités
Décision	Point de choix entre plusieurs activités qui donne lieu à un ensemble de flots - flots optionnels - dont un seul sera effectif
Fusion	Point de regroupement de plusieurs flots optionnels en un unique flot
Partitions	Ensemble de sous-ensembles d'activités gérés par des services différents
Synchronisation	Point de passage obligatoire et synchronisé des flots qui s'exécutent en parallèles
Parallélisation	Point de création de plusieurs flots, qui s'exécuteront en parallèles, à partir d'un flot d'activités
Nœud objet	Représente le changement de l'état interne d'un objet en fonction du flot courant. Il est considéré comme une activité à part entière et fait donc partie intégrante du flot d'activités

TAB. 4.3: Concepts du diagramme de d'activité

4.3.4 Diagramme de classes

Le diagramme de classe décrit, du point de vue de l'analyse, la **structure des entités manipulées** par les utilisateurs. En conception, il représente la structure d'un code orienté objet ou, à un niveau de détail plus important, les modules du langage de développement. (Cf. Tab. 4.4, Fig. 4.5)

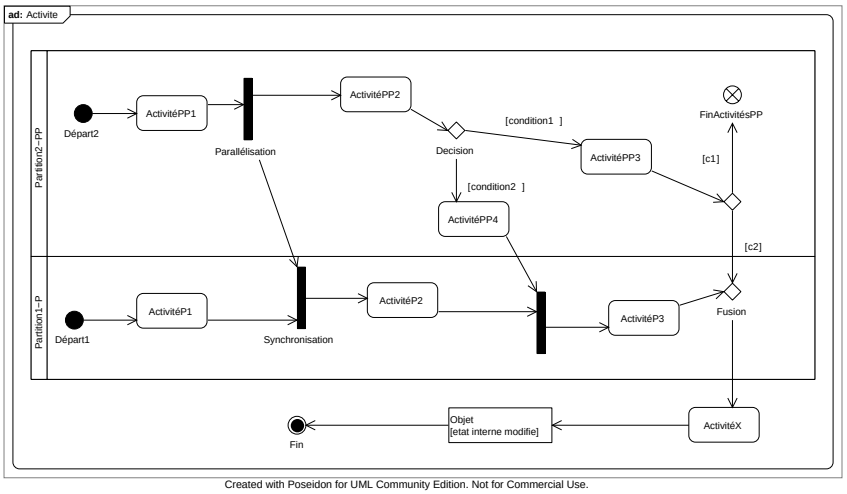


Fig. 4.4: Formalisme des concepts du DAC

Concept	**Fonction**
Classe	Structure des entités «de base» composant le logiciel
Interface	Jeu nommé d'opérations qui caractérise le comportement
Package	Sous-diagramme de classes dont les éléments partagent un thème commun
Association	Description d'une connexion entre les instances d'une classe
Agrégation	Forme d'association qui spécifie une relation d'inclusion entre un agrégat et une de ses parties constituantes
Composition	Forme forte d'agrégation avec des notions d'appartenance et de ligne de vie coïncidentes
Dépendance	Relation entre deux éléments du modèle
Généralisation/Spécialisation (ou héritage)	Relation entre une description plus spécifique et plus générale utilisée pour l'héritage et pour les déclarations polymorphes
Réalisation	Relation entre une spécification et son implémentation

Tab. 4.4: Concepts du diagramme de classes

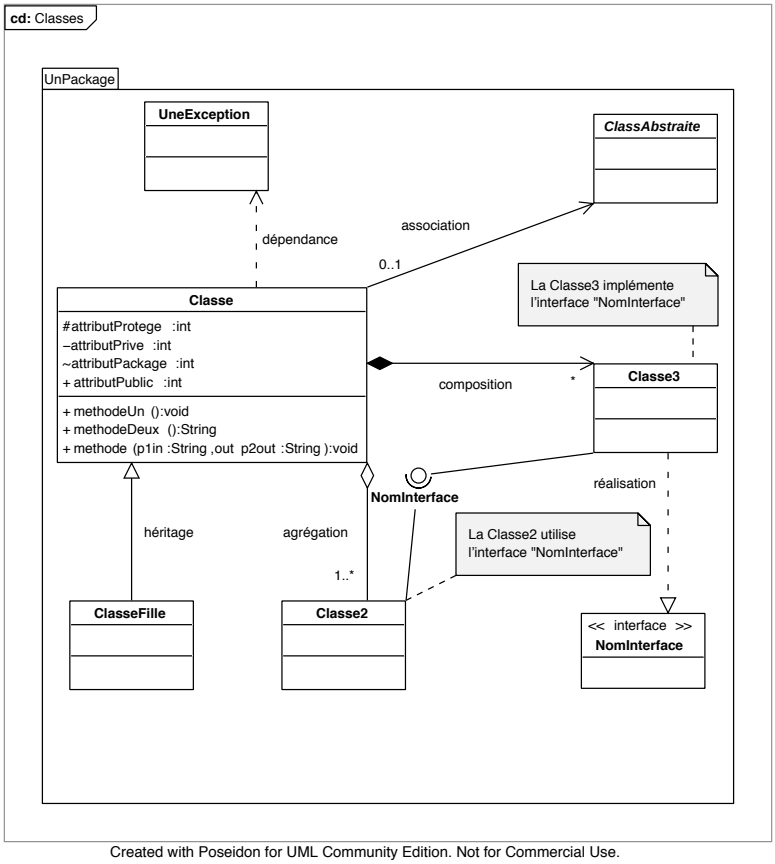


FIG. 4.5: Formalisme des concepts du DAC

Chapitre 5

Analyse du problème

Dans le chapitre précédent nous avons présenté le langage de modélisation UML avec lequel nous allons modéliser notre projet. Son utilisation nous permettra de développer une application objet répondant aux critères du sujet.

Avant d'illustrer les diagrammes UML nécessaires à la conception, le *recueil des besoins* introduit une vision utilisateur du système permettant de mieux comprendre son fonctionnement.

5.1 Recueil des besoins

Lors du recueil des besoins, on distingue deux utilisations du système : Le **Préprocessing** (application d'un filtrage uniquement) et la **Résolution** (résolution d'un CSP donné). On sait que l'on peut appliquer le *Préprocessing* avant la *Résolution*, et que l'un et l'autre doivent **Importer** un CSP et **Vérifier** sa consistance.

Le *Préprocessing* et la *Résolution* doivent respectivement **Filtrer** et **Résoudre** un CSP. Le *Filtrage* étant soit de type **Filtrage_AC** ou de type **Filtrage_PC** et la *Résolution* étant soit de type **Résolution_statique** (ordonnancement avant la résolution) ou soit de type **Résolution_dynamique** (ordonnancement pendant la résolution). La *Résolution* doit *Filtrer* et **Ordonnancer** . L'*ordonnancement* est soit de type **Ordonnancement_variable** ou de type **Ordonnancement_valeur**.

Un *utilisateur* du système peut appliquer le *Préprocessing* ou la *Résolution* à un CSP donné en ayant auparavant *charger* le solveur. C'est à dire, au moment du chargement du solveur, l'*utilisateur* fournit un ou plusieurs CSP dans un format quelconque (sous forme de fichier(s)) et les informations concernant le traitement à effectuer. (filtrage, résolution...)

5.2 Diagramme des cas d'utilisation

La conception d'un diagramme des cas d'utilisation est la *première étape* de notre modélisation. Ce diagramme sera le **squelette de notre application** en nous permettant de recueillir les besoins du système et en organisant la suite du développement. (Cf. Fig. 5.1)

En concevant le systéme de cette manière, on remarque que l'*évolution* de l'application est garantie. La création de cas d'utilisation *abstrait* permet ainsi d'effectuer facilement de l'*héritage* sur les fonctionnalités principales (résoudre, filtrer, ordonnancer).

5.3 Diagrammes de séquences

Après avoir réaliser le diagramme des cas d'utilisation du problème, il faut pouvoir représenter les *scénaris* associés. Les diagrammes de séquences vont nous permettre de faire apparaître la chronologie des échanges entre objets. Seuls les cas d'utilisation nous intérressant seront dévelopés.

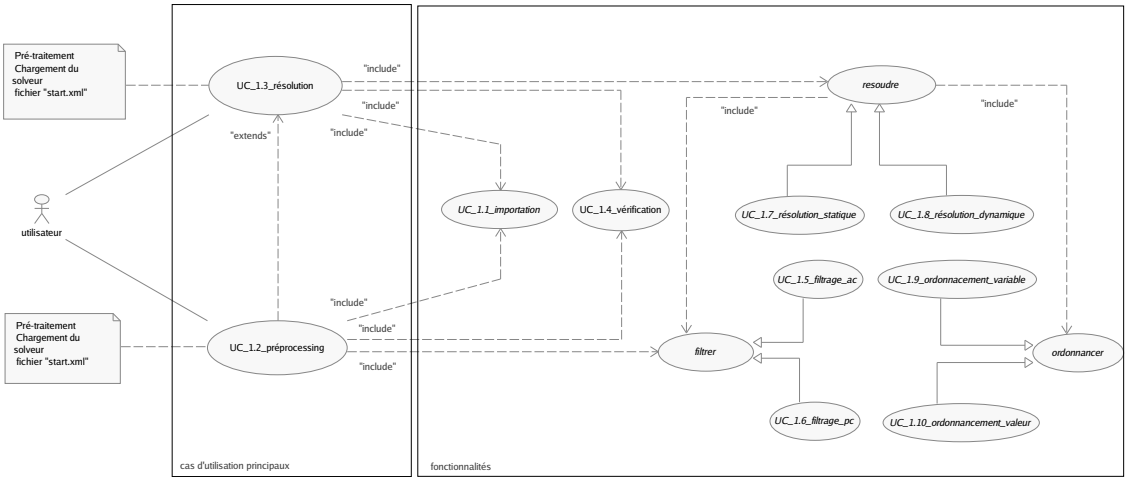


Fig. 5.1: Diagramme des cas d'utilisation

5.3.1 Importation

Chronologiquement l'*importation* intervient en tête, elle joue le rôle de l'*initialisation*. Le chargement du solveur ayant été validé, le solveur fait appel à l'importation pour créer le CSP à l'aide d'un **Parser** de fichier.

Tant que le *Parser* retourne des *Token* valide (domaine, variable...) il crée les instances, sinon il arrete l'importation et termine le programme. Enfin, il crée le CSP avec les instances parsées. (Cf. Fig. 5.2)

5.3.2 Préprocessing

Après avoir importer le CSP, le solveur lui applique le *filtrage* que l'utilisateur a mentionné. Si le filtrage s'est déroulé correctement et q'un *tuple* solution a été retourné, alors on le *vérifie*. Ensuite on sauvegarde les modifications que le filtrage a pu engendré dans le CSP. (Cf. Fig. 5.3)

5.3.3 Résolution

De la même manière que pour le *préprocessing*, après avoir importer le CSP (et eventuellement appliqué un *préprocessing*), le solveur lui applique la *résolution* que l'utilisateur a mentionné. Si la résolution s'est déroulé correctement et qu'un *tuple* solution a été retourné, alors on le *vérifie*. Ensuite on sauvegarde les modifications que la résolution a pu engendrer dans le CSP. (Cf. Fig. 5.4)

5.3.4 Vérification

En fin de programme, le solveur *vérifie* si le CSP est consistant ou non. Pour ce faire, pour chaque contrainte du CSP on demande à la relation concernée si le *tuple* concerné est accepté. Si ce n'est pas le cas, alors on retourne *faux* et l'on termine le programme. (Cf. Fig. 5.5)

5.3.5 Filtrage AC

Ce diagramme de séquence correspond à l'algorithme vu dans la partie précédente au Chapître *Programmation par contraintes, Méthodes de filtrage*. Son fonctionnement a donc dèja était détaillé. (Cf. Fig. 5.6)

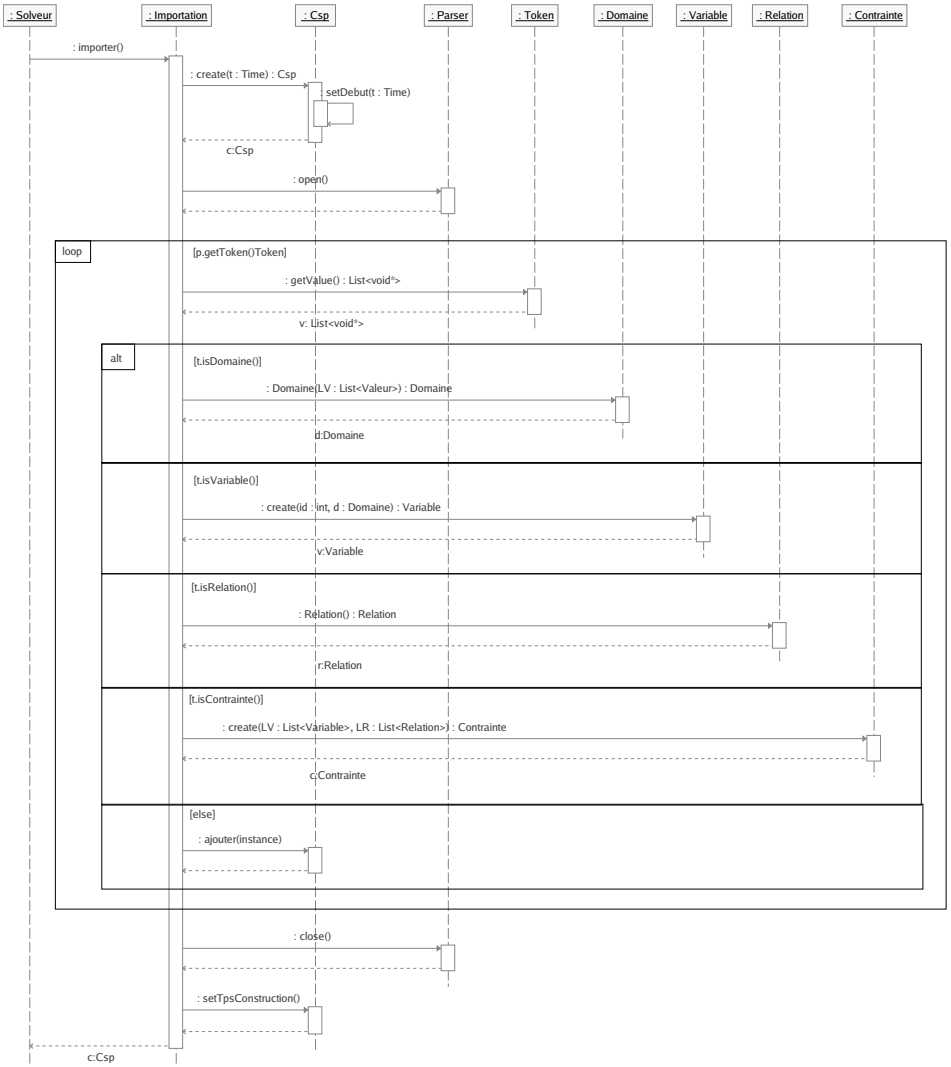


FIG. 5.2: Diagramme de séquence de l'importation

5.3.6 Résolution statique

Ce diagramme de séquence détaille le fonctionnement de la résolution statique. Autrement dit, une résolution *n'ordonnancant* les variables ou les valeurs *qu'une seule fois* (avant la résolution). Après cet ordonnancement, la résolution applique un *filtrage* à toutes les valeurs du domaine de chaque variable du CSP. Si après le *filtrage*, on n'a toujours pas de solution alors on rajoute la variable à la liste. (Cf. Fig. 5.7).

5.4 Diagramme d'activité

5.4.1 Revise AC-3r(m)

Les cas d'utliisations ayant étant définis, ce diagramme d'activité viens décrire l'**action complexe** qu'est la procédure *Revise* de l'agorithme *AC3r(m)* vue dans la partie précédente au Chapître *Programmation par contraintes, Méthodes de filtrage*. Son fonctionnement a donc dèja était détaillé. (Cf. Fig. 5.8)

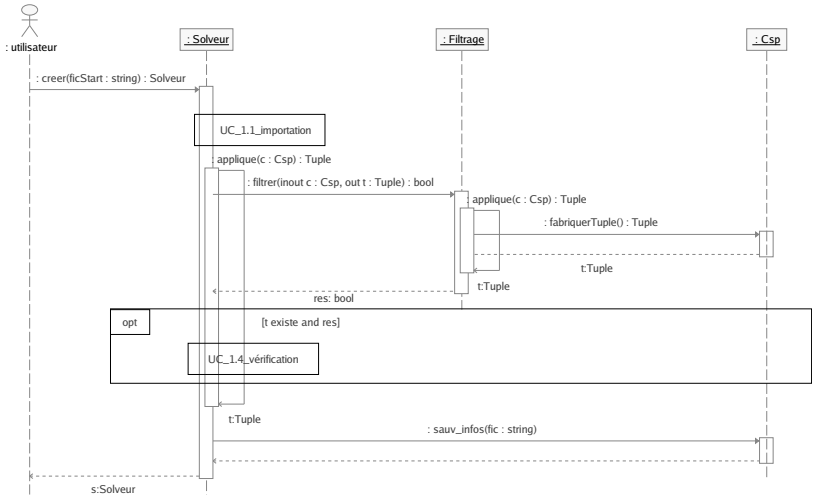


Fig. 5.3: Diagramme de séquence du préprocessing

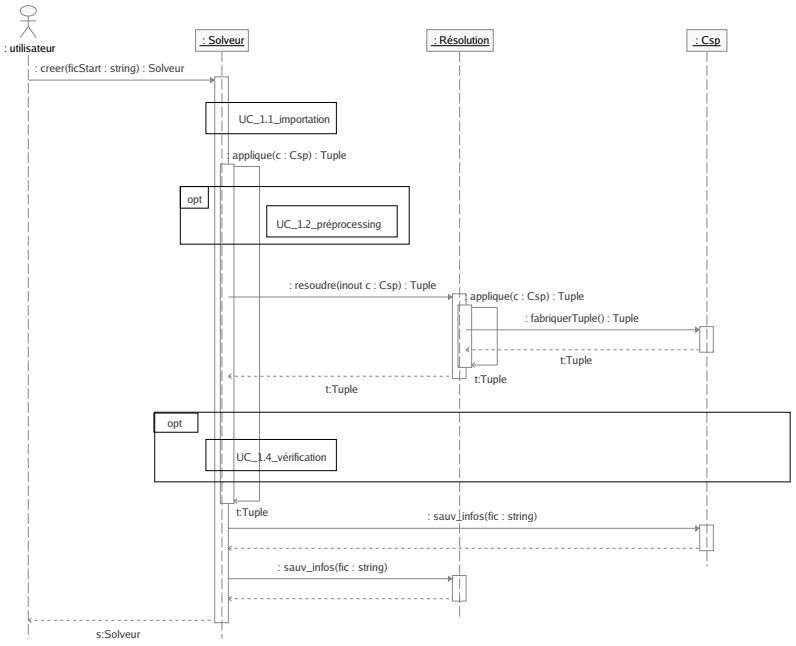


Fig. 5.4: Diagramme de séquence de la résolution

5.5 Diagrammes de classe

Les diagrammes de classes suivants représente la *structure du code orienté objet* que nous allons voir dans la partie suivante. Leur réalisation s'est faite en parrallèle des autres diagrammes et leur contenu a bien sûr était modifié au cours du développement.

5.5.1 Structure CSP

Ce diagramme contient les classes liées à la structure d'un CSP et met en avant les intéractions entre différentes classes. (Cf. Fig. 5.9)

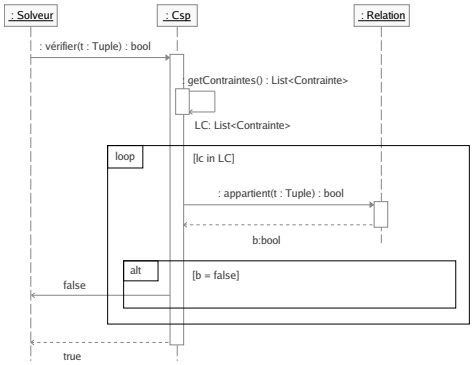


FIG. 5.5: Diagramme de séquence de la vérification

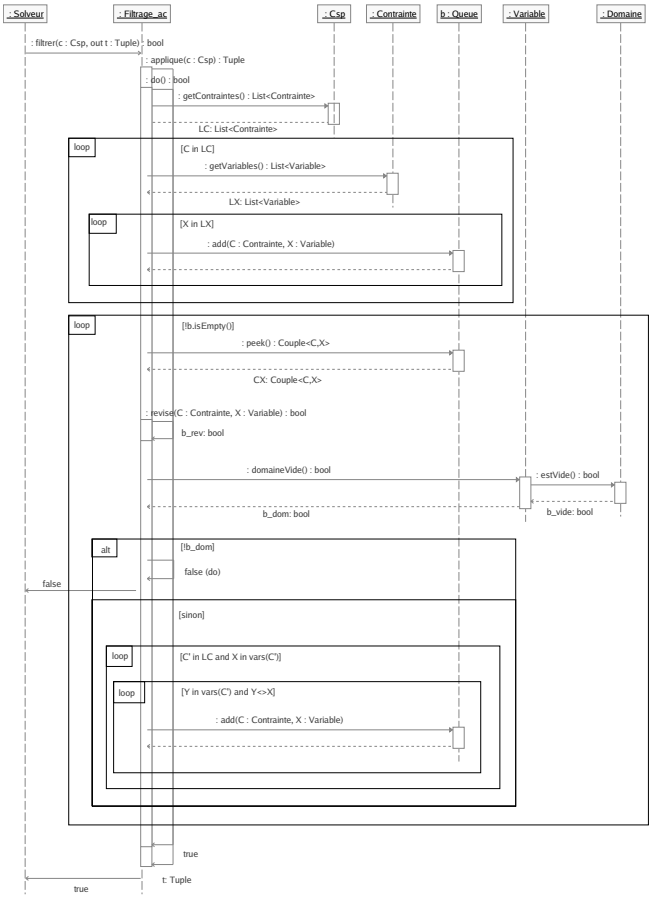


FIG. 5.6: Diagramme de séquence du filtrage AC

5.5.2 Solveur CSP

Ce diagramme contient les classes liées au fonctionnement algorithmique du projet (importation, filtrage, résolution...). (Cf. Fig. 5.10)

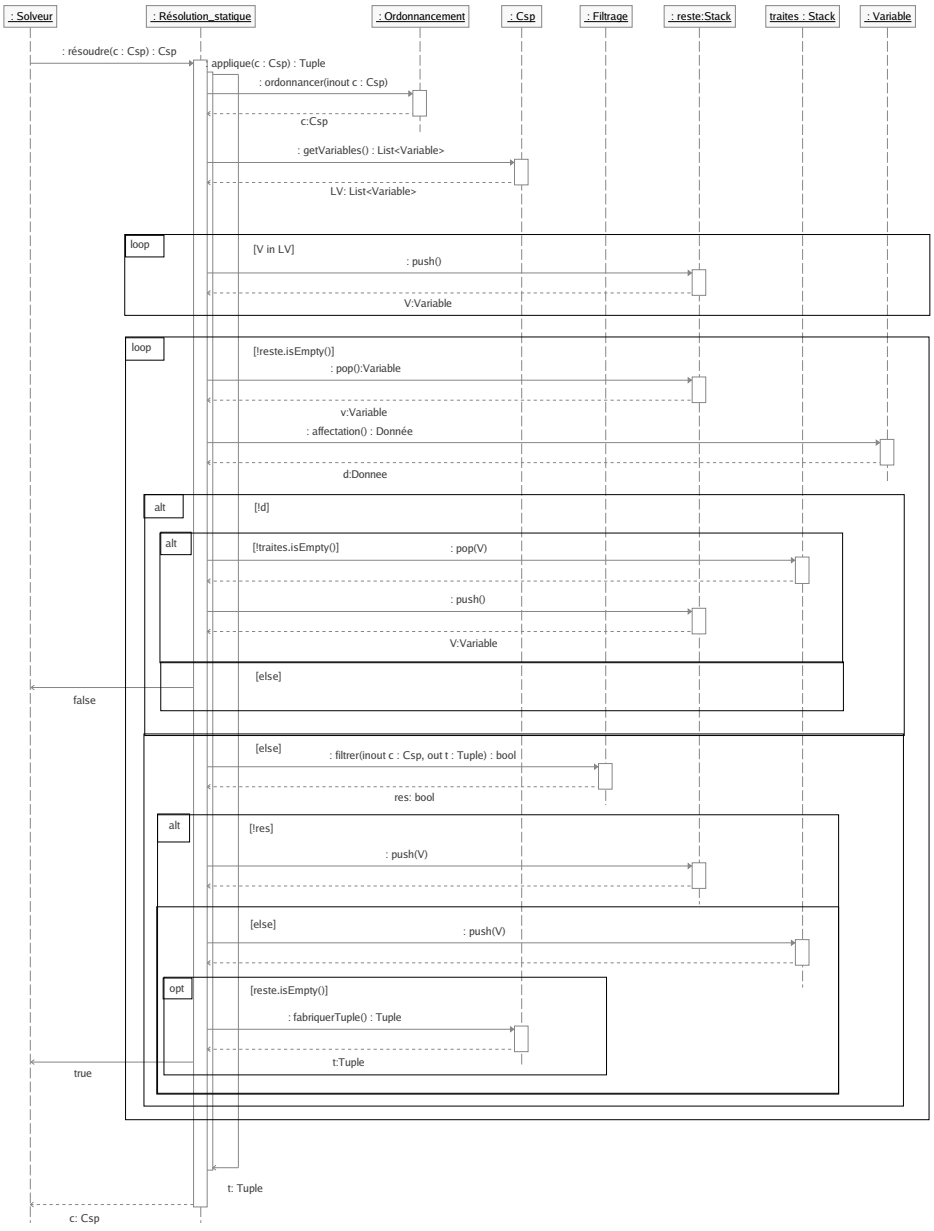


Fig. 5.7: Diagramme de séquence de la résolution statique

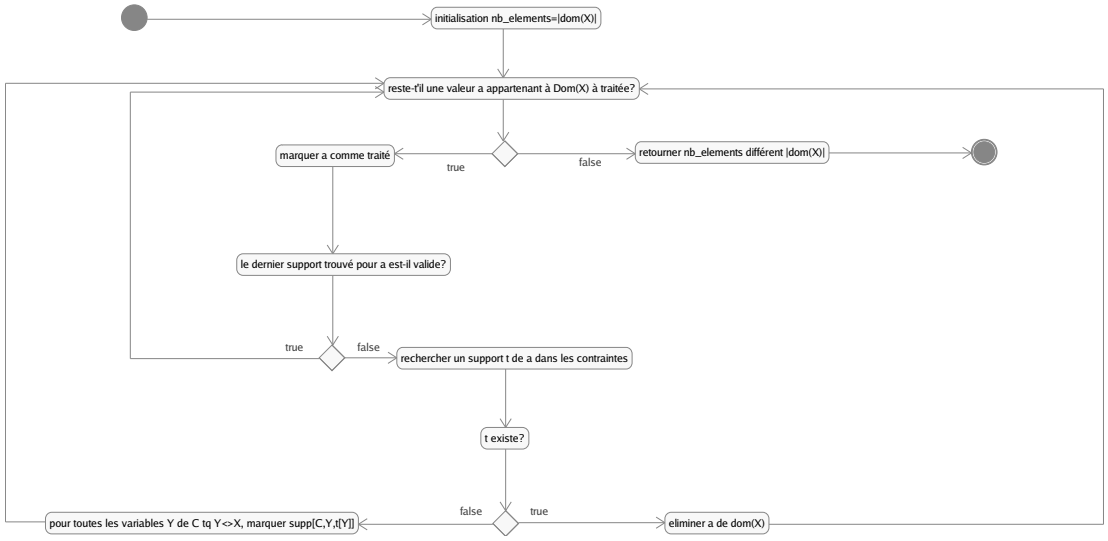


FIG. 5.8: Diagramme d'activité de Revise3r(m)

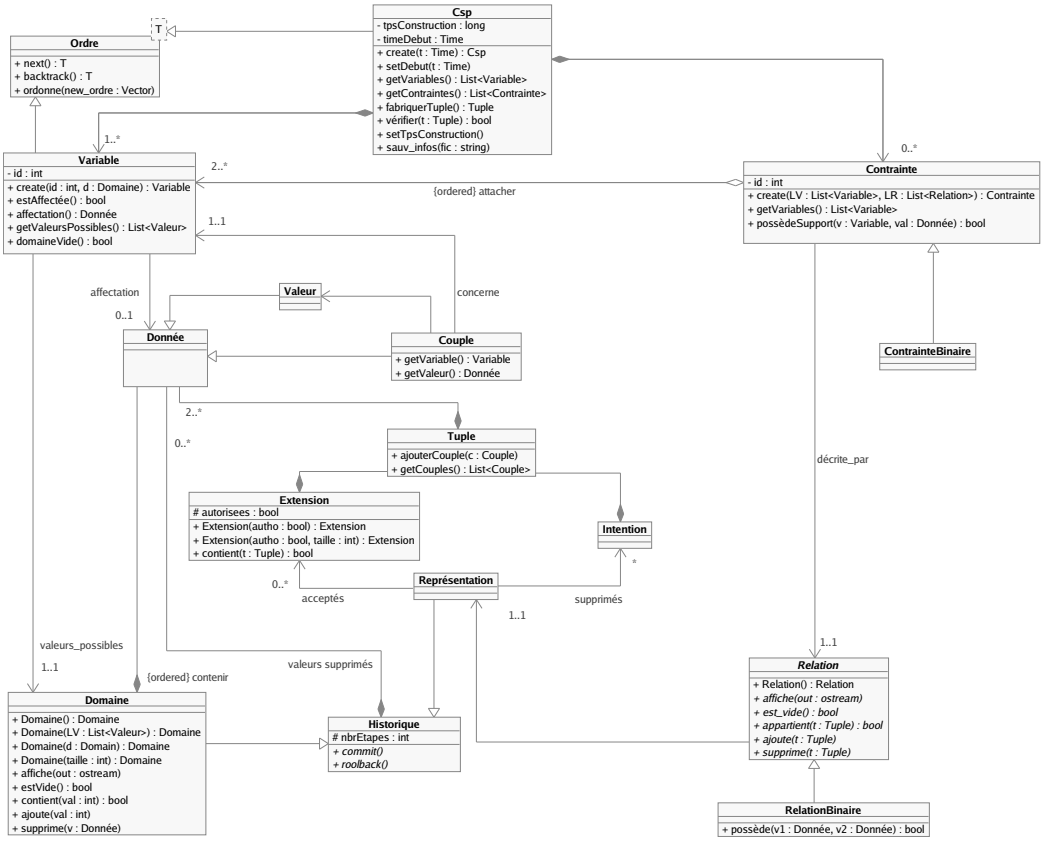


FIG. 5.9: Diagramme de classe du CSP

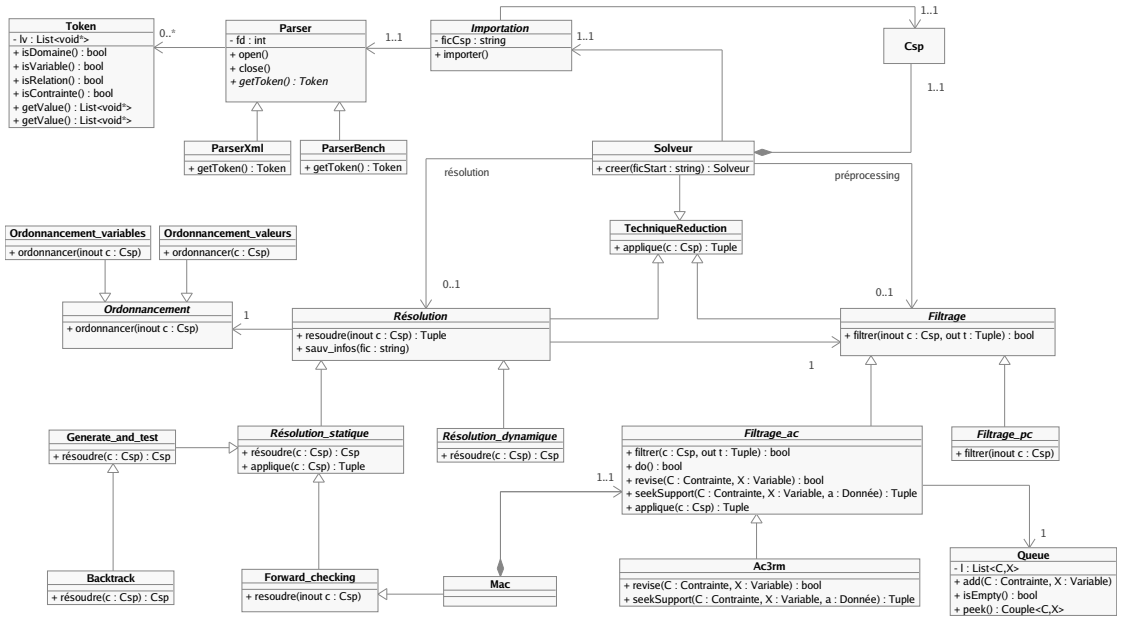


Fig. 5.10: Diagramme de classe du solveur

Troisième partie

Développement

Chapitre 6

Outils

Dans ce chapître, partant de la représentation XML des réseaux de contraintes et en passant par le choix du langage de programmation jusqu'à la définition des outils de notre structure, nous allons présenter les éléments nécessaires au développement de l'application.

6.1 Représentation normalisée des CSP

Comme nous l'avons vu dans la partie précédente, un réseau de contraintes peut être représenté par un *graphe de contrainte*. Cependant, cette représentation ne peux s'appliquer dans des problèmes de grande ampleur, le nombre de *variables* et de *contraintes* étant si grand que les *relations* ne sont plus visible.

Bien sûr, cette représentation *visuelle* ne peut être que difficilement comprise par un programme en son entrée. C'est pourquoi une **représentation normalisée** sous forme de fichier est nécéssaire.

Le **Parser de fichier** suivant la représentation présentée ci-après est le fruit du travail de Stéphane CARDON et ne fera pas l'objet d'une description approfondie dans les sections suivantes.

6.1.1 eXtensible Markup Language

XML (eXtensible Markup Language, langage de balisage extensible) est un **langage informatique de balisage générique**. Son objectif initial est de faciliter l'échange automatisé de contenus entre systèmes d'informations hétérogènes (interopérabilité), notamment sur Internet. XML est un sous-ensemble de SGML (Standard Generalized Markup Language) dont il retient plusieurs principes comme : la structure d'un document XML est définissable et validable par un schéma, un document XML est entièrement transformable dans un autre document XML. Cette syntaxe est reconnaissable par son usage des chevrons (< >) et s'applique à de plus en plus de contenus.

Un document au format XML a pour but de contenir des données **hiérarchisées**. En effet, ces dernières sont fournies sous forme d'une *arborescence d'eléments XML*. De ce fait, chaque fichier XML ne doit contenir qu'un et un seul élément dit élément racine. Un élément se compose d'un nom, d'une suite d'attributs avec leurs valeurs, éventuellement vide, et d'un contenu. La première ligne d'un document XML peut-être l'élément spécial '?xml'. Il permet de définir la version du document et son encodage. Ci-dessous un exemple d'élément XML : (avec et sans contenu, * signifiant 0 ou plusieurs)

```
<nom_element (nom_attribut=``valeur'')*> contenu </nom_element>
<nom_element (nom_attribut=``valeur'')*\>
```

6.1.2 Représentation XML des réseaux de contraintes

Pour représenter nos réseaux de contraintes nous allons utiliser un format de représentation utilisant le langage XML définit au point précédent. Cette représentation est due aux travaux de recherche de Christophe LECOUTRE pour le *comité d'organisation de la seconde compétition internationale de solveurs de CSP*, une description complète de celle-ci est disponible dans le document *'XML Representation of Constraint Networks Version 2.0'*.

Ce format permet de représenter des réseaux de contraintes sur des *domaines finis* avec des *contraintes* définies en *extension* ou en *intention*. Cela signifit que dans chaque *réseau*, les *domaines* (associés aux *variables*) correspondent à des ensembles finis de valeurs, et que les contraintes sont toutes définies explicitement par des ensembles de tuples, ou implicitement par des prédicats.

Quant une contrainte est donnée en intention, un prédicat doit être introduit et quant une contrainte est donnée en extension, un ensemble de tuples doit être introduit. Plus précisement, les contraintes peuvent être représentée en donnant :
- Un prédicat qui détermine quant un tuple donné est accepté ou refusé,
- Un ensemble de tuple accepté, appelé *'support'*,
- Un ensemble de tuple refusé, appelé *'conflicts'*.

Chaque réseau de contraintes est défini dans un *élément racine* appelé <instance>. Cet élément racine peut contenir les élements basique suivants : <presentation>, <domain>, <variable>, <relation>, <predicate> et <constraint>. Ces éléments basiques hormis <presentation> quant ils sont plusieurs dans l'instance doivent être placés dans des éléments de regroupage portant le nom de l'instance au pluriel. (<constraints>, <variables>...) Ces éléments ainsi que leurs attributs sont présentés ci-aprés sous forme d'un exemple.

```
<?xml version="1.0" encoding="UTF-8"?>
<instance>
<presentation name="firme_automobile" maxConstraintArity="2"
        nbSolutions="2" solution="POR=3 CAP=3 CAR=3 TOI=2 PAR=0 ENJ=1"
        maxSatisfiableConstraints="6" format="XCSP 2.0">
        This is the "firme automobile" instance represented in extension.
</presentation>

<domains nbDomains="5">
        <domain name="D0" nbValues="3">1..3</domain>
        <domain name="D1" nbValues="4">0..3</domain>
        <domain name="D2" nbValues="1">2</domain>
        <domain name="D3" nbValues="1">0</domain>
        <domain name="D4" nbValues="2">1..2</domain>
</domains>

<variables nbVariables="6">
        <variable name="POR" domain="D0"/>
        <variable name="CAP" domain="D0"/>
        <variable name="CAR" domain="D1"/>
        <variable name="TOI" domain="D2"/>
        <variable name="PAR" domain="D3"/>
        <variable name="ENJ" domain="D4"/>
</variables>

<relations nbRelations="4">
        <relation name="R0" arity="2" nbTuples="3" semantics="supports">
                1 1|2 2|3 3</relation>
        <relation name="R1" arity="2" nbTuples="3" semantics="supports">
```

```
                0 1|0 2|0 3</relation>
        <relation name="R2" arity="2" nbTuples="1" semantics="supports">
                2 3</relation>
        <relation name="R3" arity="2" nbTuples="2" semantics="supports">
                1 2|1 3|2 3</relation>
</relations>

<constraints nbConstraints="6">
        <constraint name="C0" arity="2" scope="POR_CAP" reference="R0"/>
        <constraint name="C1" arity="2" scope="POR_CAR" reference="R0"/>
        <constraint name="C2" arity="2" scope="CAP_CAR" reference="R0"/>
        <constraint name="C3" arity="2" scope="PAR_CAR" reference="R1"/>
        <constraint name="C4" arity="2" scope="TOI_CAR" reference="R2"/>
        <constraint name="C5" arity="2" scope="ENJ_CAR" reference="R3"/>
</constraints>
</instance>
```

6.2 Programmation Objet

Afin de pouvoir implémenter le problème en suivant notre modélisation, il nous faut pour cela utiliser un **langage objet**. Dans cette section, nous allons présenter notre choix de langage suivi de sa description et des outils de stockage de nos informations.

6.2.1 Choix du langage

Après avoir modéliser notre problème de manière objet, plusieurs langage de programmation de type C furent analysés :

- Le **C** : langage procédural donc moins adapté pour retranscrire notre modélisation, mais certainement plus rapide,
- Le **C static** : identique au C avec variables globales,
- Le **C objet** : langage simulant le mécanisme de classe,
- Le **C++** : langage objet donc parfait pour retranscrire notre modélisation, mais certainement moins rapide à cause des principes objets.

Afin de choisir, nous avons réaliser un *test simple* tel que dans chacun des langages le permettant (Cf. *Rapport Annexe, choix du langage*) nous avons implémenter un **tri à bulles** où :

- une classe A implémente la *création* du tableau contenant des entiers non ordonnés et la méthode *remonte* du tri effectuant la comparaison
- et une classe B héritant de la classe A et effectuant le *tri* pour chaque élément.

La complexité du tri est en $(n(n-1)/2)$ soit $300000+299999+...+2 = (300000*(299999))/2-1$. On exécute alors les test avec 300000 entiers pour chaque langage. (Cf. Fig. 6.1)

Langage	Real	User	Sys
C	108.26	108.01	0.02
C statique	113.93	113.68	0.01
C objet	109.83	109.52	0.04
C++	108.80	108.65	0.01

TAB. 6.1: Résultats des test du "tri à bulle" selon les langages

On s'apercoit alors que le fait d'utiliser le C++ par rapport à du C ne nous fait perdre que 0.6s, que le C++ est plus rapide que le C objet et que le C static respectivement de plus de 1.0s

et de plus de 5.0s. Sachant que l'on gagne en lisibilité en utilisant le C++, c'est avec ce langage que nous choisissons de programmer.

6.2.2 Langage C++

Le langage C++ est une "amélioration" du *langage C* (le langage C a été mis au point par *M.Ritchie* et *B.W.Kernighan* au début des années 70). **Bjarne Stroustrup**, un ingénieur considéré comme l'inventeur du C++, a en effet décidé d'ajouter au langage C les propriétés de l'**approche orientée objet**. Ainsi, vers la fin des années 80 un nouveau langage, baptisé C with classes (C avec des classes), apparait. Celui-ci a ensuite été renommé en C++, clin d'oeil au symbole d'incrémentation ++ du langage C, afin de signaler qu'il s'agit d'un langage C amélioré (langage C+1).

Le C++ reprend la quasi-intégralité des concepts présents dans le langage C, si bien que les programmes écrits en langage C fonctionnent avec un compilateur C++. En réalité le langage C++ est un surensemble du C, il y ajoute, entre autres, des fonctionnalités objet :
- **Encapsulation,**
- **Héritage,** (simple et multiple)
- **Polymorphisme.**

Grâce aux concepts objets (définis au chapître précédent) que le langage C++ englobe, nous allons pouvoir suivre de manière logique notre modélisation et ainsi implémenter de manière efficace notre problème. Il reste pourtant à déterminer comment nous allons représenter nos données physiquement.

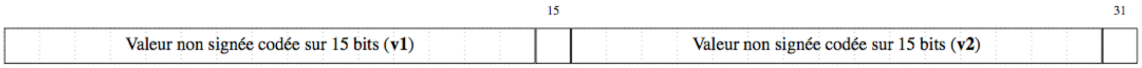
6.2.3 Classe Int : Champ de bits

Afin de gagner de l'espace et du temps lors de l'accés à nos valeurs, nous allons, dans une structure appelée "Int", les décrire en terme de bits. Il existe dans le langage C un moyen de réaliser de telles descriptions, à l'aide du concept de structure.

En effet, dans une déclaration de structure, il est possible de faire suivre la dèfinition d'un membre par une indication du nombre de bits que doit avoir ce membre. Dans ce cas, le langage C appelle cela un champ de bits.

Pour notre problème nous allons définir quatre champs : (Cf. Fig. 6.1)
- **v1** (15 bits) : permet de stocker une valeur signée de -16384 à 16384 (valeur réelle) ou une valeur non signée de 0 à 32768, (indice)
- **b1** (1 bit) : permet de définir un état, (de v1 généralement)
- **v2** (15 bits) : permet de stocker une valeur signée de -16384 à 16384 (valeur réelle) ou une valeur non signée de 0 à 32768, (indice)
- **b2** (1 bit) : permet de définir un état, (de v2 généralement)

Aussi nous disposons de méthodes nous permettant d'**accéder** (set) et de **modifier** (get) les champs *v1* et *v2* de notre champ de bits. C'est grâce à celles-ci que les classes représentant la structure au chapître suivant pourront utiliser efficacement notre champ de bit.



6.2.4 Classe Queue : File

Lors de l'étude des algorithmes de résolution vus au chapître précédent, nous avons pu nous apercevoir que certains d'entre-eux nécéssite d'**enfiler** et de **dépiler** à plusieurs reprise des *couples*. (du type Variable, Contrainte...) Ces opèration ne sont pas coûteuses en terme d'espace ni de temps

```
struct Int {
  unsigned int v1 : NB_BITS_VAL;
  unsigned int b1 : 1;
  unsigned int v2 : NB_BITS_VAL;
  unsigned int b2 : 1;
};

void setV1 (Int& i,int v);
void setV2 (Int& i,int v);
int getV1 (Int& i);
int getV2 (Int& i);
```

FIG. 6.1: structure objet de la classe Int

d'exécution donc on pourrait utiliser une *file* de Int pour les stocker.

Cependant l'opération d'**appartenance**, quant-à elle peut s'avérer très couteuse. En effet, dans notre cas elle ne peut se faire en *temps constant*. C'est donc cette opération qui nous pousse à implémenter notre propre structure afin de la faire fonctionner en *temps constant*. (Exemple d'utilisation en *Rapport Annexe, Tests unitaires, Queue*)

Représentation

Pour cela on va effectuer une **bijection** du couple dans l'ensemble des entiers naturels $\mathbb{N}$ et stocker cette valeur au lieu du couple. Pour savoir si le couple existe (**contains**), il suffira d'effectuer la **bijection inverse** pour savoir si il est présent ou non, ce qui donne l'application suivante :

- $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$
- $(c, v) \rightarrow c \times (nbVar \times 2) + v$
- $f^{-1} : (\alpha/(nbVar \times 2), \alpha\%(nbVar \times 2)) \leftarrow \alpha$

La *Queue* a pour attribut un **vecteur** contenant $n \times c$ *entiers*. Si une case de ce vecteur contient -1 alors le couple n n'est pas présent, sinon la case contient le *successeur* du couple dans le vecteur. Aussi, la *Queue* a comme attribut le **nombre de variables** et l'indice de **début** et de **fin** de la file. Bien sûr notre structure propose les mêmes fonctionnalités que toute structure de File, à savoir **empty, push, pop**. (Cf. Fig. 6.2)

```
class Queue {
protected:
  vector<int> q;
  unsigned int nbVar;
  int debut, fin;

public:
  Queue (unsigned int e, unsigned int n);

  void affiche (ostream& out);
  unsigned int bijection (unsigned int c, unsigned int v);
  void bijectionInv (unsigned int bij, unsigned int& c, unsigned int& v);

  bool empty ();
  void push (unsigned int ind_c, unsigned int ind_v);
  void pop (unsigned int& ind_c, unsigned int& ind_v);
  bool contains (unsigned int ind_c, unsigned int ind_v);
};
```

Fig. 6.2: structure objet de la classe Queue

Chapitre 7

Structure CSP

Dans ce chapître, nous allons présenter l'implémentation de la structure du CSP. Cette partie du programme doit *fournir* les méthodes permettant de réaliser les fonctionnalités principales du solveur (filtrage, résolution...) dans le chapître suivant *Solveur CSP*.

En effet, nous allons développer les *intéractions* entre les différentes classes de la structure du CSP. Aussi nous décrirons l'utilisation des *outils de la programmation objet* présentés dans le chapître précédent, pour chaque classe liée.

7.1 Modélisation hiérarchisée

Avant de développer l'implémentation de cette partie du programme, nous allons réorganiser le diagramme de classes de la structure CSP (chapître *Analyse du problème*, Fig. 5.9) selon la hiérarchie suivante :

- *Stockage des informations* : Cette partie représente les classes physiques déja implémentées (*Vecteurs...*) et les outils présentés au chapître précédent. (*champ de bits Int...*)
- *Représentation en extension des informations*
- *Représentation en intention des informations*
- *Entités logiques*
- *Description du problème*

Le nouveaux diagramme de classes de notre structure va nous permettre une implémentation plus simple du problème en partant des classes les moins abstraites jusqu'aux classes les plus abstraites. (Cf. Fig. 7.1)

7.2 Représentation en extension des informations

Cette partie, fournit les classes nécessaires à la représentation des informations en extension. Elle fait bien sûr appel aux classes moins abstraites du *stockage des informations*. La classe *Tuple* étant associée aux *Parsers* de fichier, ne fera pas l'objet d'une description plus approfondie.

7.2.1 Classe Historique

L'objectif de cette classe est de pouvoir **stocker** des valeurs supprimées de classes lui *héritant*. Elle met à disposition deux méthodes à redéfinir permettant de **valider** (commit) et d'**annuler** (rollback) ces suppressions.

Représentation

Elle possède un *int* (nbrEtapes) et un *vector<Int>* (valeursSupprimees). A l'initialisation *nbrEtapes* est nul, et lors d'un *Commit* cette valeur est incrémentée. (Cf. Fig 7.2)

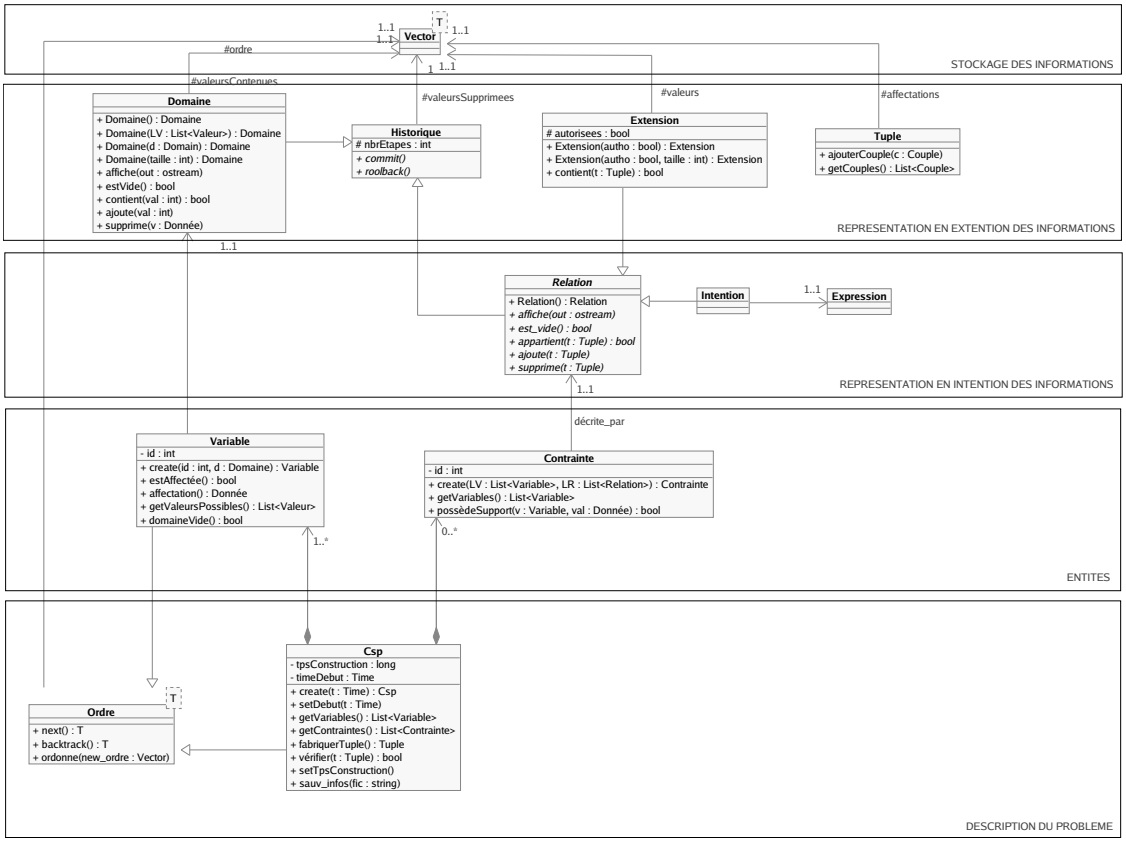


Fig. 7.1: Diagramme de classes hiérarchisée de la structure CSP

```
class Historique {
protected:
  unsigned int nbrEtapes;
  vector<Int> valeursSupprimees;

public:
  Historique ();

  virtual void commit ();
  virtual void rollback ()=0;
};
```

Fig. 7.2: Structure objet de la classe Historique

7.2.2 Classe Domaine

L'objectif de cette classe est de pouvoir **ajouter** et **supprimer** des valeurs à un *Domaine*. Ces modifications doivent pouvoir être **valider** (commit) et **annuler** (rollback) à tout moment. C'est pourquoi la classe *Domaine* hérite de la classe *Historique* dont elle utilise les fonctionnalités. (Exemples d'utilisation en *Rapport Annexe, Tests unitaires, Domaine*)

Des méthodes simples permettent de connaître la *taille* du Domaine, de savoir si un Domaine *contient* une valeur indicée. Mais aussi deux méthodes permettant de renvoyer la *valeur* réelle d'un indice dans le domaine et l'*indice* d'une valeur réelle.

Représentation

Un domaine représente l'ensemble des valeurs que peux prendre une Variable. En extension, cet ensemble doit être totalement listé. (Exemple : $D_1 = \{0, 1, 2\}$) Une valeur d'un domaine est représentée par un *Int* dans lequel certains sous ensembles de bits sont utilisés pour coder ces différents états. (Cf Fig. 7.3)

Utilisation des différents champs :

- **v1** : Valeur réelle. (de -16384 à 16384)
- **v2** : Indice de la prochaine valeur supprimée.
- **b1** (bit 15) : Indique si la valeur appartient au domaine, ou si elle est supprimée.
- **b2** (bit 31) : Indique si cette valeur (supprimée) est en fin de liste de suppression.

```
class Domaine : public Historique {
protected:
  vector<Int> valeursContenues;
  unsigned int nbValeurs;

public:
  Domaine () {};
  Domaine (int taille);
  Domaine (Domaine* dom);

  void affiche (ostream& out);
  bool estVide ();
  bool contient (unsigned int indice);
  unsigned int getTaille ();

  unsigned int idValeur (int valeur);
  int getValeur (unsigned int indice);

  void ajoute (int valeur);
  void supprime (unsigned int indice);
  virtual void commit ();
  virtual void rollback ();
};
```

FIG. 7.3: Structure objet de la classe Domaine

Supression d'une valeur

Lorsqu'il y a *suppression* d'une valeur, en connaissant son indice, on doit savoir si, à cette étape courante, il y a déjà eu une suppression : (Cf. Fig. 7.4)

- Si oui : on relie cette nouvelle valeur supprimé à l'ancienne et elle devient la dernière valeur supprimée,
- Si non : on marque cette valeur comme étant supprimée et comme étant en fin de liste de suppression. Finalement, elle devient la dernière supprimée à cette étape.

Annulation des suppressions : rollback

Lorsqu'on annule les suppressions effectuées à une étape donnée, on parcours la liste des valeurs supprimées (à l'aide de leur indice) en mettant le bit supprimé à 0. (Cf. Fig. 7.5)

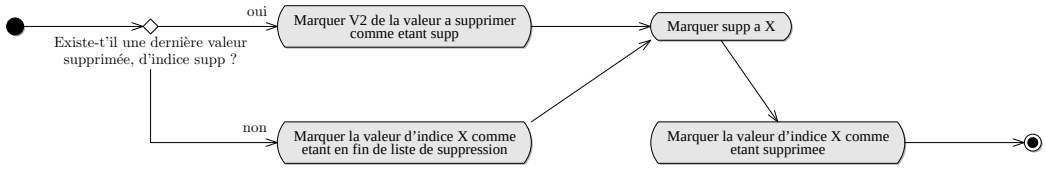


Fig. 7.4: DAC de la suppression d'une valeur d'un domaine

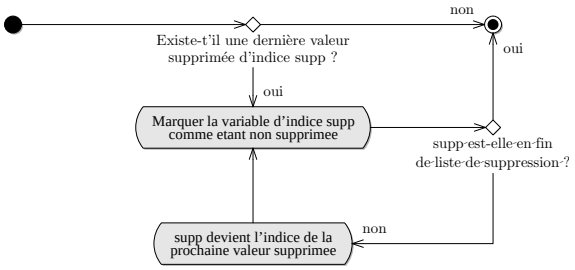


Fig. 7.5: DAC du rollback d'une valeur d'un domaine

7.2.3 Classe Extension

Une relation en extension d'une *Contrainte* donne les valeurs *autorisés* ou *refusées* (*Tuples*) entre deux Variables. Elle hérite de la classe *Relation* dont elle implémente toutes les méthodes. Elle possède un champ *autorisées* notant si les tuples représentés sont autorisés ou refusés. (Cf. *Rapport Annexe, Test unitaires, Extension*)

Des méthodes simples permettent de savoir si une relation en extension est *vide*, si un tuple (indice de valeurs) lui *appartient* et si une valeur indicée *possède* un *support*.

Représentation

Elle est représentée comme un graphe dirigé entre indices de variables. Cette représentation est gérée à l'aide de la classe *Int*. (Cf. Fig. 7.6)

Utilisation des différents champs :

- **v1** : Indice de la valeur d'une variable
- **v2** : Indice de la valeur d'une variable
- **b1** (bit 15) : Indique si l'arc $v1 \rightarrow v2$ existe
- **b2** (bit 31) : Indique si l'arc $v2 \rightarrow v1$ existe

Plus exactement, une contrainte est un **vector<Int>**, trié dans l'ordre croissant. L'ordre entre deux tuples binaires c_1 et c_2 est celui-ci :

$$c_1 < c_2 \text{ ssi } c_1.v1 < c_2.v1 \vee c_1.v2 < c_2.v2$$

Ajout d'un Tuple

Avec cette représentation, lors de l'ajout d'une contrainte entre les valeurs des variables a et b, si $i_{v_a} > i_{v_b}$, on regarde si le tuple i_{v_b}, i_{v_a} existe. Si oui, on met le bit **31** à 1, sinon on crée ce tuple, on met le bit **31** à 1 et le bit **15** à 0.

La taille maximale de cette contrainte est : $d+(d-1)+(d-2)+...+1$, donc $\frac{d(d-1)}{2}$ au lieu de d^2. Par contre, cela engendre un surcoût de recherche en d^2 à la création. Ce surcoût peut être réduit à $\log_2 \frac{d(d-1)}{2}$ si le vecteur est tout le temps trié mais de ce fait un ajout entraînera un décalage des tuples dans le vecteur.

```
class Extension : public Relation,public BitMatrice {
private:
  void setDimensions (unsigned int col,unsigned int lig);

protected:
  bool autorisees;
  unsigned long nbTuples;

public:
  Extension (bool autorisation);
  Extension (bool autorisation,unsigned int col,unsigned int lig);
  Extension (Extension* ext);

  virtual void affiche (ostream &out);
  virtual bool estVide ();
  virtual bool appartient (unsigned ind_v1, unsigned ind_v2);
  virtual bool support (unsigned int ind_var, unsigned int ind_val);

  void ajouteBrut (int v1,int v2);
  void setDomaines (Domaine* d1,Domaine* d2);

  virtual void ajoute (unsigned int ind_v1, unsigned int ind_v2);
  virtual void supprime (unsigned int ind_v1, unsigned int ind_v2);
  virtual void commit ();
  virtual void rollback ();
};
```

FIG. 7.6: Structure objet de la classe Extension

Suppression d'un Tuple

En premier lieu, il n'y a pas suppression effective dans la contrainte (comme pour le domaine). Seul le bit correspondant au bon arc est mis à 0. Par contre, le vecteur historique contiendra plus d'informations. Pour l'étape précédente de l'étape courante, l'historique contiendra tous les arcs supprimés jusqu'à un MARKER. Il correspond à 0 pour tous les champs. En feffet, tout élément de cet historique possèdera son bit **15** ou **31** à 1. (Cf. Fig. 7.7)

Annulation des suppressions : rollback

Il s'agit juste de decrémenter la taille de l'historique des suppressions jusqu'à obtenir le MARKER ou que l'historique soit vide. Pour chaque entrée de l'historique, on modifie le(s) bit(s) **15** et/ou **31** de l'entrée e correspondante dans la liste des contraintes à 0 s'ils sont à 1 dans l'entrée de l'historique. (Cf. Fig. 7.8)

7.3 La représentation en intention des informations

Cette partie, fournit les classes nécessaires à la représentation des informations en intention. Elle fait bien sûr appel aux classes moins abstraites de la *représentation en extension des informations* comme la classe *Historique*. Les classe *Intention* et *Expression* étant associées aux *Parsers* de fichier, ne feront pas l'objet d'une description plus approfondie.

Aussi cette classe comprend la classe *Relation* dont hérite les classes *Intention* et *Extension*. Autrement dit, une *Relation* est soit en *Intention* soit en *Extension*.

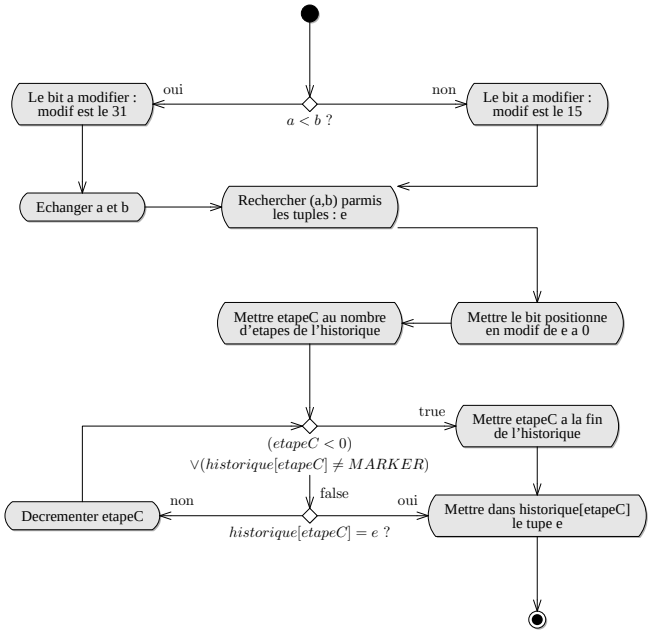


Fig. 7.7: DAC de la suppression d'un Tuple

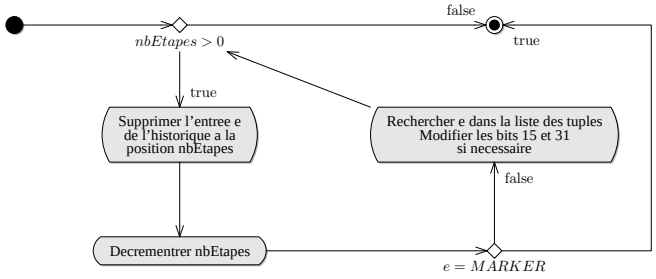


Fig. 7.8: DAC du rollback d'un Tuple

7.3.1 Classe Relation

Cette classe *abstraite* propose des méthodes pour les classes héritantes, afin de remplir un but simple : la représentation des relations. Cette classe permettra d'utiliser les relations en *intention* et en *extension* de manière totalement invisible puisque la *contrainte* possèdera une *relation* abstraite. (Cf. *Rapport Annexe, Test unitaires, Relation*)

Représentation

Elle hérite de la classe *Historique* afin de pouvoir stocker les suppressions de *Relation*. Aucune méthode n'est implémentée, seules les classes héritantes peuvent le faire. (Cf. Fig. 7.9)

7.4 Les entités logiques

Cette partie regroupe les classes centrales de notre problème, grâce à elles, le CSP peut effectuer les opérations le liant au *solveur*. Les *Contraintes* et les *Variables* sont des classes ayant comme instances des objets issus des classes les moins abstraites. (*représentation en extension...*)

```
class Relation : public Historique {
protected:
  Relation() : Historique () {};

public:
  virtual void affiche (ostream &out)=0;
  virtual bool estVide ()=0;
  virtual bool appartient (unsigned int ind_v1, unsigned int ind_v2)=0;
  virtual bool support (unsigned int ind_var, unsigned int ind_val)=0;

  virtual void ajoute (unsigned int ind_v1, unsigned int ind_v2)=0;
  virtual void supprime (unsigned int ind_v1, unsigned int ind_v2)=0;
};
```

FIG. 7.9: Structure objet de la classe Relation

7.4.1 Classe Contrainte

Une contrainte définie les relations autorisés (ou interdites) entre un *Tuple* de *Variables*. Elle peut vérifier qu'une valeur de l'une des variables **possède** un **support**. (Cf. *Rapport Annexe, Test unitaires, Contrainte*)

Représentation

Une contrainte possède deux indices de *variables* et une *Relation* entre ces deux *variables*. (tuples acceptés de la relation) Elle peut retourner la valeur de la variable associée à une variable indicée. (Cf. Fig 7.10)

```
class Contrainte {
protected:
  unsigned int ind_v1, ind_v2;
  Relation* rel;

public:
  Contrainte (unsigned int i_v1, unsigned int i_v2, Relation* r);

  unsigned int getV1 ();
  unsigned int getV2 ();
  Relation* getRelation ();
  int estDansVars (unsigned ind_var);
  void affiche (ostream &out);
  bool possedeSupport (unsigned int ind_var, unsigned int ind_val);
};
```

FIG. 7.10: Structure objet de la classe Contrainte

7.4.2 Classe Variable

Une variable possède un *Domaine* ordonné (héritage de *Ordre*). Elle peut savoir si elle est *affectée* et demander sa *prochaine affectation* sous forme de valeur ou sous forme d'indice. (Cf *Rapport Annexe, Test unitaires, Variable* et Fig. 7.11)

```
class Variable : public Ordre {
protected:
  Domaine* d;

public:
  Variable (Domaine* d);

  void affiche (ostream &out);
  Domaine* getDomaine ();
  bool estAffectee ();
  unsigned int affectation ();
  unsigned int affectInd ();
};
```

Fig. 7.11: Structure objet de la classe Variable

7.5 La description du problème

Cette partie représente la donnée essentielle sur laquelle le solveur va travailler. Elle fait appel aux entités logiques qu'elle possède comme attribut. (Cf. *Rapport Annexe, Tests unitaires, Csp*)

7.5.1 Classe Ordre

Cette classe va nous permettre d'ordonner différents objets. Elle ordonne entre-autre les *Variables* dans le CSP et les valeurs dans les *Variables*.

Représentation

Un ordre possède un vecteur d'*Int* stockant les indices des instances ordonnés et un entier non signé pour connaître l'indice courant. Les méthodes **next** et **backtrack** retourne respectivement l'indice ordonné suivant et pécédent. La méthode **ordonne** réordonne la liste des indices à partir de l'indice courant. (Cf. Fig. 7.12)

```
class Ordre {
protected:
  vector<Int> ordre;
  unsigned int ind_next;

public:
  Ordre (unsigned int taille);

  unsigned int next ();
  unsigned int backtrack ();
  void ordonne (const vector<unsigned int>& newOrdre);
};
```

Fig. 7.12: Structure objet de la classe Ordre

7.5.2 Classe CSP

Un CSP possède des *Contraintes* et des *Variables ordonnées*. On peut lui **ajouter** des *Variables* et des *Contraintes*. (Cf. Fig. 7.13)

Représentation

On peut obtenir une *Variable* ou une *Contrainte* en donnant son indice grâce aux méthodes **get**. Aussi lors de l'ajout d'une *Variable* ou d'une *Contrainte*, les méthodes **add** retournent l'indice de l'instance ajoutée.

```
class Csp {
protected:
  vector<Variable*> lv;
  vector<Contrainte*> lc;

public:
  Csp ();
  Csp (unsigned int v_size, unsigned int c_size);
  Csp (Csp* csp);

  void affiche (ostream &out);
  unsigned int sizeContrainte ();
  unsigned int sizeVariable ();
  void setNombreVariables (unsigned int v_size);
  void setNombreContraintes (unsigned int c_size);

  unsigned int addVariable (Variable* v);
  unsigned int addContrainte (Contrainte* c);
  Variable* getVariable (unsigned int indice);
  Contrainte* getContrainte (unsigned int indice);
};
```

FIG. 7.13: Structure objet de la classe Csp

Chapitre 8

Solveur CSP

Dans cette partie, nous allons présenter l'implémentation du solveur de CSP. Celle-ci suit entièrement la modélisation la concernant vue dans la partie précédente. Aussi les algorithmes du programme (AC-3r(m), MAC...) seront construits à partir de la structure du CSP vue au chapître précédent.

En effet, ici nous nous attacherons dans un premier temps à suivre la modélisation pour la repésentation des classes *abstraites*, puis nous nous appuierons sur la structure du CSP pour implémenter les algorithmes de *filtrage* et de *résolution.* héritant de ces classes. L'*importation* étant liée au *Parser de fichier*, elle ne fera pas l'objet d'une étude approfondie.

8.1 Filtrage

Cette section présente la partie *filtrage* du solveur. Bien entendu, elle se basera sur la modélisation pour l'implémentation des classes abstraites et sur la structure du CSP pour l'algorithme de filtrage implémenter dans notre programme.

8.1.1 Classe Filtrage

Cette classe *abstraite* définie la méthode **filtrer** qui renvoie *vrai* si le filtrage s'est bien effectué, *faux* sinon. Toute classe héritant de *Filtrage* devra donc redéfinir cette méthode. (Cf. Fig. 8.1)

```
class Filtrage {
protected:
  Filtrage () {};

public:
  virtual bool filtrer ()=0;
};
```

FIG. 8.1: Structure objet de la classe Filtrage

8.1.2 Classe Filtrage AC

Cette classe, elle-aussi *abstraite*, définie le *filtrage* par *consistance d'arc.* Elle hérite de la classe *filtrage* et doit donc redéfinir la méthode **filtrer**. Aussi elle propose les méthodes **revise** et **seek support** (vues dans la partie *Etude*) permettant de faire fonctionner un filtrage par consistance d'arc. Les classes héritant de *Filtrage AC* devront donc redéfinir ces trois méthodes. (Cf. Fig. 8.2)

```
class Filtrage_ac : public Filtrage {
protected:
  Filtrage_ac () : Filtrage () {};

public:
  virtual bool revise (unsigned int indC, unsigned int indV)=0;
  virtual int seekSupport (unsigned int indC, unsigned int indV,
        unsigned int indA,Domaine* dom,bool first ,unsigned int depart)=0;
};
```

Fig. 8.2: Structure objet de la classe Filtrage AC

8.1.3 Classe AC-3r(m)

Cette classe a pour but d'effectuer le filtrage de type AC-3r(m) vu dans la partie *Etude*. Elle hérite de la classe *Filtrage AC* et doit donc redéfinir les méthodes **filtrer**, **revise** et **seekSupport** pour pouvoir fonctionner. (Cf. *Rapport Annexe, Tests unitaires, AC-3r(m)*)

Représentation

La classe AC3-r(m) doit être **créer** avec un CSP. Aussi, elle possède comme attribut un *vecteur de vecteur de Int* pour stocker les supports (ou résidus) vus lors de l'étude. Les différents algorithmes (revise, seekSupport...) ne feront pas l'objet d'une description approfondie, celle-ci ayant dèja était faite au coeur des différents diagrammes de la modélisation. (Cf. Fig. 8.3)

```
class Ac3rm : public Filtrage_ac {
protected:
  Csp *csp;
  vector<vector<Int> > supp;

public:
  Ac3rm (Csp* c);

  virtual bool filtrer ();
  virtual bool revise (unsigned int indC, unsigned int indV);
  virtual int seekSupport (unsigned int indC, unsigned int indV,
        unsigned int indA, Domaine* dom,bool first ,unsigned int depart);
};
```

Fig. 8.3: Structure objet de la classe AC-3r(m)

8.2 Résolution

Cette section présente la partie *résolution* du solveur. Bien entendu, elle se basera sur la modélisation pour l'implémentation des classes abstraites. Cette partie et la partie ordonnancement n'ont pas fait l'objet d'une implémentation, cependant une description des classes abstraites de la résolution peut être faite. (Voir Fig. 8.8)

8.2.1 Classe Résolution

Cette classe *abstraite*, dont un diagramme UML fait la description, définie la méthode **résoudre** qui renvoie *vrai* si la résolution s'est bien effectuée, *faux* sinon. Toute classe héritant de *Résolution* devra donc redéfinir cette méthode. (Cf. Fig. 8.4)

```
class Resolution {
protected:
  Resolution () {};

public:
  virtual bool resoudre ()=0;
};
```

FIG. 8.4: Structure objet de la classe Résolution

8.2.2 Classe Résolution statique

Cette classe, elle-aussi *abstraite* et dont un diagramme UML fait aussi la description, définie le *résolution statique*. Elle hérite de la classe *résolution* et doit donc redéfinir la méthode **résoudre**. Les classes héritant de *Résolution statique* devront donc redéfinir cette méthode et les méthodes dont elle a besoin pour fonctionner. (Cf. Fig. 8.5)

```
class Resolution_statique : public Resolution {
protected:
  Resolution_statique () : Resolution () {};

public:
  // methodes abstraites dont la resolution statique
  // a besoin pour fonctionner
};
```

FIG. 8.5: Structure objet de la classe Résolution statique

8.2.3 Classe Forward checking

Cette classe, hérite de la *résolution statique* et doit donc redéfinir, la méthode **résoudre** (et éventuellement les méthodes dont la résolution statique a besoin pour fonctionner). Elle a besoin aussi pour être **créer** d'une classe auxiliaire vérifiant la consistance (de base **check forward**, Cf. Fig. 8.6).

```
class Fc : public Resolution_statique {
protected:
  Csp *csp;
  Filtrage *f;

public:
  Fc (Csp* c, Filtrage* f);
  Fc (Csp* c); {
        f = new CheckForward (csp *c);
  }

  virtual bool resoudre ();
  // methodes abstraites dont la resolution statique
  // a besoin pour fonctionner
};
```

FIG. 8.6: Structure objet de la classe Fc

8.2.4 Classe MAC

Cette classe dont la description a été faite dans la partie *Etude* hérite de la classe *Résolution statique* et reprend le **forward checking** dont la méthode auxiliaire n'est autre que **AC-3r(m)**. (Cf. Fig. 8.7)

```
class Mac3rm : Resolution_statique {
protected:
  Csp *csp;
  Filtrage *f;
  Resolution *r;

public:
  Mac3rm (Csp* c) {
        f = new Ac3rm (c);
        r = new FC (c, f);
  }

  virtual bool resoudre () {
        return r->resoudre ();
  }
};
```

Fig. 8.7: Structure objet de la classe Mac3r(m)

```
Csp *c;                                    // deja parse
Filtrage *f = new Ac3rm (c);               // preprocessing
f->filtrer ();

Resolution *f = new Mac3rm (c);
r->resoudre ();
```

Fig. 8.8: Tests unitaires d'une Resolution

Chapitre 9

Conclusion

9.1 Evolution

La partie *Résolution* (FC, MAC-3r(m)), dont l'étude et la modélisation ont été présentées dans les chapîtres correspondants, n'a pu être implémentée. Le programme, il est vrai, souffre de quelques latences dans son utilisation finale en comparaison au solveur *Abscon* (développé par l'équipe du CRIL), sur une phase de *préprocesing*. Cela se justifie sans doute par le manque d'heuristiques et de techniques de propagation n'ayant pas fait l'objet de notre implémentation. Le dossier d'analyse devrait permettre de résoudre ces problèmes de manière rapide dans un futur proche.

9.2 Recherche

En étudiant les CSPs et les diverses techniques de filtrage et de résolution proposées jusqu'à présent, on s'apercoit que la **complexité** n'a pu être réduite, pour l'instant, à une compléxité polynomiale. C'est pourquoi divers courants de pensées aimeraient implanter aux algorithmes de résolution un tout autre comportement.

Bien-sûr, pouvoir transmettre le comportement du **cerveau humain** aux algorithmes, semble être une idée intéressante. Grossissant le trait (sans prendre en compte l'inférence, l'instinct...), en déroulant notre comportement sur la résolution de problèmes simples, on s'apercoit que celui-ci pourrait s'apparenter à des techniques de résolution déjà en place au détail prés du stockage des informations dont nous abusons et qui rend la compléxité aussi importante.
Autrement dit, seules les parties du cerveau dont nous connaissons peu de chose seraient efficaces en terme algorithmique combinées à la rapidité d'exécution de nos ordinateurs.

La connaissance réduite de cette particularité de l'humain étant telle, nous pouvons penser que le comportement d'individus moins complexe pourrait être à son tour implémenté. De part leur résistance à l'érosion du temps ou encore à leur présence en nombre sur l'intégralité du globe, les **colonies de fourmis** seraient de bons candidats à cette expérimentation. En s'appuyant sur le *problème du voyageur de commerce*, on met en avant le lien de cette méthode avec les **systèmes multi-agents** ou encore les **systèmes neuronaux**.

Une dernière étude introduit un vision détournée de la résolution des problèmes. La théorie de l'évolution de DARWIN montre à quel point notre espèce à su résoudre les problèmes, ou plutôt s'y adapter, afin de survivre. C'est ainsi que les **algorithmes génétiques** sembleraient être potentiellement viable combinés à la vitesse d'éxécution de nos ordinateurs.

Bien évidemment ces hypothèses, ne faisant pas partie du sujet, ne sont en aucun cas vérifiées. Elles ne font donc pas l'objet d'une étude approfondie prouvant si elles pourraient l'être, mais mériteraient semble t-il une attention particulière, et pourraient être développées, sans aucunes prétentions, dans de futures recherches.

Liste des Algorithmes

Table des figures

Bibliographie

[IAIT] Jean-Marc. ALLIOT & Thomas SCHIEX, *Intelligence Artificielle & Informatique Théorique, Cépaduès Editions, 1994*

[C++] Bjarne STROUSTRUP, *Le langage C++, Pearson Education, 2003*

[AC3RM] Christophe LECOUTRE & Fred HEMERY, *Une étude des supports résiduels pour la consistance d'arc, Actes JFPC, 2006*

[SAC] Stéphane CARDON &Christophe LECOUTRE, *Une approche gloutone pour établir la singleton consistance d'arc, Actes JFPC, 2005*

[HEUR] Frédéric BOUSSEMART & Fred HEMERY & Christophe LECOUTRE & Lakhdar SAIS, *Heuristiques de choix de variables dirigées par les conflits, Actes JNPC, 2004*

[CSPGEN] Nicolas BARNIER & Pascal BRISSET, *Optimisation par hybridation d'un CSP avec un algorithme génétique, JFPLC, 1997*

[CSP] Patrice BOIZUMAULT, *Constraint Satisfaction Problems M1-Info UE13, Université de CAEN, 2007*

[SOC] Lakhdar SAIS, *Satisfaction et Optimisation de Contraintes M2-Info SOC,Université d'Artois, 2007*

[UML] Christophe CARDON, *Génie Logiciel M1-Info GL UML, Université d'Artois, 2007*

[PPC1] Wikipédia, *Survol de la Programmation par Contraintes, http ://fr.wikipedia.org/wiki/ Survol_ de_ la_ Programmation_ par_ Contraintes, 2007*

[PPC2] Wikipédia, *Programmation par contraintes, http ://fr.wikipedia.org/wiki/ Programmation_ par_ contraintes, 2007*

[C++REF] The C++ Ressources Network, *C++ Library Reference, http ://www.cplusplus.com/ reference, 2007*

[ALGGEN] Labo Algo, *Algorithme génétique, http ://labo.algo.free.fr/pvc/ algorithme_ genetique.html, 2007*

[ALGFOURM] Labo Algo, *Algorithme de la colonie de fourmis, http ://labo.algo.free.fr/pvc/ algorithme_ colonie_ de_ fourmis.html, 2007*

[FOURM] D-MOULI, *La vie des fourmis, http ://membres.lycos.fr/dmouli/, 2007*

"Il n'existe pas de problème dans la nature, mais seulement des solutions car l'état naturel est un état adaptatif donnant naissance à un système cohérent"
René DUBOS, biochimiste franco-américain (1901-1982)


UNIVERSITÉ D'ARTOIS
cril
centre de recherche en informatique de lens
CNRS
CENTRE NATIONAL
DE LA RECHERCHE
SCIENTIFIQUE