N° d'ordre : 2397 2006

Thèse présentée

en vue de l'obtention du grade de

docteur de

L'INSTITUT NATIONAL POLYTECHNIQUE DE TOULOUSE

École Doctorale : Systèmes

Spécialité : Systèmes Industriels

par

Thomas VAN OUDENHOVE DE SAINT GÉRY

CONTRIBUTION À L'ÉLABORATION D'UN FORMALISME GÉRANT LA PERTINENCE POUR LES PROBLÈMES D'AIDE À LA CONCEPTION À BASE DE CONTRAINTES

Soutenue le 15 novembre 2006, devant le jury composé de :

Jean-Pierre NADEAU	Examinateur (Président du jury)
Laurent GRANVILLIERS	Rapporteur
Philippe LUTZ	Rapporteur
Michel ALDANONDO	Examinateur (Directeur de thèse)
Paul GABORIT	Examinateur (Encadrant)
Élise VAREILLES	Examinateur (Encadrant)
Matthieu VERON	Examinateur (Invité)

Thèse préparée au Centre Génie Industriel de l'École des Mines d'Albi-Carmaux

Remerciements

En premier lieu, je voudrais remercier mes encadrants de thèse ; ce manuscrit ne serait pas aussi abouti sans leurs conseils et leur disponibilité : Michel ALDANONDO, directeur de thèse et Paul GABORIT, encadrant, sans oublier ÉliseVAREILLES, qui fût ma collègue pendant deux ans et a rejoint l'équipe d'encadrement pour la dernière année. Merci à tous les trois de m'avoir accompagné tout au long de ces trois années.

J'adresse également mes remerciements aux autres membres de mon jury, pour avoir accepté de juger nos travaux :

- Laurent GRANVILLIERS, rapporteur et Professeur à la Faculté des Sciences de Nantes (Laboratoire d'Informatique Nantes Atlantique),
- Philippe LUTZ, rapporteur et Professeur au Laboratoire d'Automatique de Besançon,
- NADEAU, président du jury et Professeur à l'École Nationale Supérieure des Arts & Métiers de Bordeaux,
- Mathieu VERON, examinateur, cofondateur et codirigeant de la société ADELYA.

Merci à tous ceux qui font ou ont fait partie du Centre Génie Industriel au cours de mon doctorat (Lionel, Matthieu, Jacques, Franck, Didier, Elyes, Hervé, Jacqueline(s), Fred, Marco et les docteurs ou doctorants : David, Franck, Mathieu, Jihed, Jaouher, Vérane, Amadou, Romain, Yosra, Samieh, Carmen, Naly et tous ceux que j'oublie) et plus particulièrement Isabelle. Merci aussi à Monsieur le *sysadmin* Emmanuel OTTON et Madame la *sysadmin* Cathy ORTEU.

Je remercie aussi tous les doctorants (et les autres) avec qui nous avons arraché la salarisation des doctorants, tout particulièrement ceux qui y ont joué un rôle actif, ainsi que les camarades Denis, Serge, Jean-Claude, Jean-Paul, Guy, Jean-Michel, Élisabeth, Rachel, Bruno,... J'adresse aussi des pensées particulières à Romain et Steph, El Patou, Seb, James, Gillou, Fabien, Petit Ohu, Clémence, Jeff et Anne-Cécile, Marilyn, Ana et Max, Emeline,... mais aussi à Bonrep's et Anne, Hélène et Grego, Branlo et Marie, Bi, Rem, le camarade Moulinovitch et les membres de « l'orchestre » *Blooming Traczir*.

Je tiens aussi à remercier toute ma famille, qui m'a soutenu pendant tout ce temps. Un immense merci à Cédric et Julien pour notre cohabitation dans la maison des trois petits cochons... Enfin, merci à tous ceux qui auraient une place ici et que j'ai oublié...

Pour finir, une petite pensée pour Donald E. KNUTH et Leslie LAMPORT, sans qui ce manuscrit ne serait pas ce qu'il est.

« L'histoire de l'humanité est une statistique de la contrainte. »

Léo FERRÉ (1916 – 1993)

Table des matières

Remerciements iii

Table des matières vii

Table des figures xi

Liste des tableaux xiii

Liste des algorithmes xv

Liste des exemples xvii

Introduction générale 1
- Cadre général . . . 1
- Problématique . . . 1
- Plan de lecture . . . 2

I Conception, configuration et approches à base de contraintes : état de l'art et problématique 5

1 Conception et configuration : présentation et besoins 7
- 1.1 Les différents types de conception . . . 8
  - 1.1.1 Conception créative . . . 8
  - 1.1.2 Conception innovante . . . 8
  - 1.1.3 Conception routinière . . . 8
  - 1.1.4 Conclusion . . . 9
- 1.2 Exploitation des connaissances . . . 9
  - 1.2.1 Exploitation de connaissances implicites . . . 10
  - 1.2.2 Exploitation de connaissances explicites . . . 10

1.2.3 Discussion et raisonnement retenu . . . 11
1.3 Aide à la conception et configuration . . . 12
1.4 Besoins en configuration et conception . . . 15
1.4.1 Des éléments de différentes natures . . . 16
1.4.2 Utilisation d'un modèle arborescent . . . 16
1.4.3 Prise en compte d'élements optionnels et notion de pertinence . . . 17
1.5 Conclusion . . . 19

2 Approches par contraintes : les CSP discrets et continus 21
2.1 Modélisation et approches par contraintes . . . 21
2.2 Recherche de solutions . . . 25
2.2.1 *Generate & Test* . . . 25
2.2.2 *BackTracking* . . . 25
2.2.3 *BackJumping* . . . 26
2.3 Filtrage sur des domaines discrets . . . 27
2.3.1 Classification générale . . . 27
2.3.2 La cohérence d'arc . . . 29
2.4 Filtrage sur des domaines continus . . . 30
2.4.1 Arithmétique des intervalles . . . 30
2.4.2 2B-cohérence . . . 32
2.4.3 *Box*-cohérence . . . 32
2.4.4 Discussion . . . 32
2.5 Diversité compilée . . . 33
2.6 Résolution et filtrage . . . 34
2.6.1 *Forward-Checking* . . . 34
2.6.2 *Look-Ahead* . . . 34
2.6.3 Discussion . . . 35
2.7 Conclusion . . . 35

3 Éléments optionnels et hiérarchie dans les CSP 37
3.1 Les DCSP . . . 38
3.2 Les CCSP . . . 40
3.3 L'ajout d'une valeur au domaine des variables . . . 40
3.4 Les CSPe . . . 41
3.5 Les ACSP . . . 42
3.6 Conclusion . . . 43

Transition 45

II Propositions de modélisation et implémentation pour la résolution de problèmes d'aide à la conception ou à la configuration 47

4 Intégration et principes de modélisation : les RCSP 49
4.1 Modéliser un composant standard optionnel . . . 49
4.2 Modélisation de contraintes conditionnelles . . . 50
4.3 Modéliser un composant paramétrable optionnel . . . 51
4.3.1 Préliminaires . . . 51

4.3.2 Modélisation d'un composant paramétrable optionnel par l'ajout d'une valeur au domaine . . . 53
4.3.3 Modélisation par l'ajout d'un attribut de pertinence . . . 54
4.3.4 Conclusion sur la pertinence des composants . . . 56
4.4 Modélisation de sous-ensembles optionnels . . . 56
4.4.1 Factorisation de l'attribut de pertinence . . . 57
4.4.2 Ajout d'éléments . . . 59
4.4.3 Conclusion sur la modélisation de sous-ensembles optionnels . . . 63
4.5 Les RCSP . . . 64
4.5.1 Définitions . . . 64
4.5.2 Agrégation des solutions . . . 64
4.5.3 RCSP— synthèse . . . 66
4.6 Conclusion . . . 66

5 Implémentation des CSP sous forme d'arbres syntaxiques 69
5.1 Le langage de description . . . 70
5.1.1 Les domaines . . . 70
5.1.2 Les variables . . . 71
5.1.3 Les contraintes . . . 71
5.1.4 Conclusion . . . 72
5.2 L'analyseur . . . 72
5.2.1 Opérateurs logiques . . . 72
5.2.2 Feuilles de comparaison symbolique . . . 73
5.2.3 Feuilles de comparaison numérique . . . 73
5.2.4 Domaines dans l'arbre syntaxique . . . 74
5.2.5 Bilan . . . 74
5.3 Le moteur de résolution . . . 75
5.3.1 Évaluation d'un nœud . . . 75
5.3.2 Mécanisme . . . 76
5.3.3 Ordre d'instanciation des variables . . . 77
5.3.4 Conclusion . . . 77
5.4 Le moteur de filtrage . . . 77
5.4.1 Introduction . . . 77
5.4.2 Logique modale . . . 78
5.4.3 Calcul des espaces pour les feuilles de comparaison symbolique . . . 79
5.4.4 Calcul des espaces pour les feuilles de comparaison numérique . . . 79
5.4.5 Calcul des espaces pour les nœuds logiques . . . 81
5.4.6 Filtrage . . . 88
5.4.7 Un exemple de filtrage . . . 89
5.4.8 Élagage de l'arbre au cours du filtrage . . . 91
5.4.9 Problèmes de stabilité numérique (convergence) . . . 96
5.5 Discussion et conclusion . . . 96

6 Enrichissements de l'implémentation 97
6.1 Enrichissement du langage . . . 97
6.1.1 Les attributs de variables . . . 97
6.1.2 L'aspect hiérarchique . . . 98

6.1.3 Conclusion . . . 101
6.2 Amélioration de la résolution . . . 101
6.2.1 Critères d'évaluation et problèmes considérés . . . 102
6.2.2 Adaptation des heuristiques de choix de variables . . . 102
6.2.3 Adaptation du *BackTrack* aux RCSP . . . 104
6.2.4 Conclusion . . . 105
6.3 Filtrage de RCSP— résultats et adaptation . . . 107
6.3.1 Déduction de la non-pertinence de variables . . . 107
6.3.2 Méta-informations . . . 112
6.4 Conclusion . . . 117

Conclusion générale **119**
Synthèse des travaux . . . 119
Perspectives . . . 120

Bibliographie **123**

III Annexes **133**

A Algorithmes **135**
A.1 Algorithmes pour CSP « classiques » . . . 135
A.2 Algorithme DCSP . . . 140
A.3 Algorithmes RCSP — filtrage des nœuds de composition logique . . . 141

B Autres exemples **147**
B.1 La voiture de MITTAL et FALKENHAINER . . . 147

Colophon **149**

Table des figures

1 Plan de lecture du mémoire 3

1.1 Extrait du processus de conception (PAHL & BEITZ, 1996) 7
1.2 Les différentes vues d'un produit 9
1.3 Connaissances et degrés de liberté selon le domaine 13
1.4 Processus d'aide à la conception ou de configuration 14
1.5 Paramètres et composants selon le domaine 15

2.3 Classification des variables selon leur type (VAREILLES, 2005) 23
2.4 Illustration du processus de *Generate & Test* 25
2.5 Illustration du processus de *BackTracking* 26
2.6 CSP arc-cohérent et non chemin-cohérent 28
2.7 Comparaison des puissances de filtrage 29
2.11 Position des variables instanciées sur plusieurs algorithmes (BARTÁK, 1998) . 35

4.4 Configuration de voiture — modélisation par RCSP et ★ 53
4.8 Configuration d'une voiture — factorisation des attributs de pertinence . . . 58
4.9 Configuration d'une voiture — modélisation grâce à des méta-variables . . . 60
4.10 Fonctionnement de l'ajout d'éléments par méta-variable (CCSP) ; X_3 est la méta-variable 61
4.11 Fonctionnement du méta-opérateur *Sure* 62
4.12 Agrégation des solutions 65

5.1 Racine et niveau logique de l'arbre syntaxique : conjonction des contraintes . 72
5.2 Feuille de comparaison numérique 74
5.3 Analyse des déclarations de domaine : feuilles résultantes 74
5.4 Arbre complet de l'exemple « voiture simple » 75
5.7 Espaces possiblement vrai et possiblement faux pour une feuille symbolique 79
5.8 Forme générale d'un arbre numérique représentant une feuille de comparaison numérique directe 80

5.9 Feuille de comparaison numérique directe . . . 80
5.10 Forme générale d'un arbre numérique représentant une feuille de comparaison numérique indirecte . . . 81
5.11 Feuille de comparaison numérique indirecte . . . 81
5.12 Utilisation d'espaces possiblement vrais virtuels . . . 82
5.13 Utilisation d'espaces possiblement vrais virtuels — le problème est localement cohérent . . . 83
5.14 Utilisation d'espaces possiblement vrais virtuels — le problème est localement incohérent (branche à gauche) . . . 83
5.15 Espaces possiblement vrai et possiblement faux pour un nœud not . . . 84
5.16 Espaces possiblement vrais et faux pour un nœud and à une seule variable . . . 85
5.17 Espaces possiblement vrais et faux pour un nœud and à deux variables . . . 85
5.18 Espaces possiblement vrais et faux pour un nœud or . . . 86
5.19 Espaces possiblement vrais et faux pour un nœud => . . . 87
5.20 Espaces possiblement vrais et faux pour un nœud <=> . . . 88
5.21 Réinjection des domaines de l'espace $\diamond V^{n-1}$ pour le filtrage de la phase n . . . 89
5.22 Exemple simple pour l'illustration du filtrage . . . 89
5.23 Arbre syntaxique et espaces possiblement vrais pour un exemple simple — le domaine de V_3 est réduit à cuir . . . 90
5.24 Arbre syntaxique et espaces possiblement vrais pour un exemple simple — le domaine de V_4 est réduit à automatique . . . 91
5.25 Arbre syntaxique et espaces possiblement vrais pour un exemple simple — les domaines D_3 et D_4 sont réduits . . . 92

6.1 Place d'un groupe dans l'arbre syntaxique . . . 99
6.7 Configuration de voiture — modélisation par RCSP et valeur ★ . . . 107
6.8 Arbre syntaxique — modélisation par RCSP et valeur ★ . . . 108
6.9 Configuration de voiture — modélisation par RCSP et attribut de pertinence . . . 108
6.10 Arbre syntaxique — modélisation par RCSP et attribut de pertinence . . . 109
6.11 Arbre syntaxique — modélisation par RCSP et attribut de pertinence — traduction évoluée . . . 110
6.12 Configuration de meuble — modélisation par RCSP et attribut de pertinence . . . 111
6.14 Extrait d'arbre syntaxique — nœud *Sure* . . . 113
6.15 Formes filtrées du nœud *Sure* avec élagage . . . 114

B.1 Schéma de la voiture de MITTAL & FALKENHAINER (1990), en utilisant la valeur ★ pour la non-pertinence . . . 148

Liste des tableaux

2.2 Ensemble des solutions de l'exemple 2.1 23

3.1 Résumé des possibilités des différents formalismes CSP étudiés 44

4.5 Solutions au problème de conception de l'exemple 1.7 page 18 54
4.13 Synthèse des agrégations de solutions pour les problèmes de conception . . . 66

6.3 Résultats en résolution (BT) — modélisation par ★ 104
6.4 Résultats en résolution (BT) — modélisation par attribut de pertinence 105
6.6 Résultats en résolution (BT-RCSP) — modélisation par attribut de pertinence 106
6.17 Résultats en filtrage numérique selon la formulation 116

Liste des algorithmes

5.5 BACKTRACKING (V_v, V_n) 76

6.5 RCSP-BT (V_v, V_n) 106
6.13 FILTRAGE DU NŒUD *Sure* 113

A.1 BACKTRACKING (V_v, V_n) 135
A.2 BACKJUMPING ($x_1, \ldots, x_i, X_i$) 136
A.3 REVISE($X_i \rightarrow X_j$) 136
A.4 AC-1 137
A.5 AC-3 137
A.6 INITIALISATION AC-6 137
A.7 SUPPORTSUIVANT(X_i, X_j, x, y, *supportvide*) AC-6 138
A.8 PROPAGATION AC-6 138
A.9 AC-3 – FORWARD-CHECKING(cv) 138
A.10 AC-3 – LOOK-AHEAD(cv) 139
A.11 DCSP (V_I) 140
A.12 CALC_SPACE_NOT ($modif_D$) 141
A.13 CALC_SPACE_AND ($modif_D$) 142
A.14 CALC_SPACE_OR ($modif_D$) 143
A.15 CALC_SPACE_EQUIVALENCE ($modif_D$) 144
A.16 CALC_SPACE_IMPLICATION ($modif_D$) 145

Liste des exemples

1.6 Moteur et roues d'une voiture — dérivé de SABIN & FREUDER (1996) 17
1.7 Voiture simple — dérivé de MITTAL & FALKENHAINER (1990) 18
1.8 Réacteur chimique — dérivé de GELLE (1998) 19

2.1 Modèle de configuration très simple 22
2.8 Extension d'une contrainte aux intervalles 31
2.9 Filtrage direct par arithmétique des intervalles 31
2.10 Contrainte 2B-cohérente 33

4.1 Modélisation de composants standard optionnels 50
4.2 Modélisation d'une contrainte conditionnelle 51
4.3 Illustration du filtrage sur des variables à pertinence non-déterminée 52
4.6 Configuration de meuble 55
4.7 CSP hiérarchique, dérivé de SABIN & FREUDER (1996) 57

5.6 Calcul des espaces — un exemple très simple 79

6.2 Groupes paramétriques — placement et configuration de bibliothèques 100
6.16 Exemple du réacteur chimique 114

Introduction générale

Cadre général

Les travaux présentés dans ce mémoire ont pour but d'aider à la conception de produits, services ou procédés. Nous distinguons plusieurs types de conception selon le degré de nouveauté du produit à concevoir. L'activité de conception que nous nous proposons d'assister est la conception routinière.

Du fait que la conception est routinière, nous disposons de connaissances sur le produit, le domaine et la démarche. Ces connaissances peuvent être des lois prouvées ou admises ou les retours d'expériences antérieures. Nous exploitons ces connaissances lors de la conception afin d'éviter des erreurs.

Il existe plusieurs méthodes de gestion de ces connaissances suivant qu'elles sont implicites ou explicites. Nous utilisons les approches par contraintes, qui permettent d'assister des tâches de conception autonome et interactive, reposant sur une gestion de connaissances explicites regroupées dans un modèle de connaissances.

Problématique

La plupart des travaux relatifs à l'aide à la conception à base de contraintes peuvent être classés selon deux aspects : l'aide à la conception (les paramètres sont généralement continus) et l'aide à la configuration (les paramètres sont généralement discrets). La configuration est un cas extrême de conception routinière, pour laquelle le modèle générique du produit est connu. La réalité des problèmes se situe souvent entre ces deux extrêmes. Nous nous situons sur ces deux domaines.

En conception comme en configuration, nous avons besoin de déterminer la présence des différents éléments du modèle (paramètres, fonctions, composants ou sous-ensembles) et de caractériser leur hiérarchie. Dans la majorité des travaux traitant des approches par contraintes, tous les éléments sont systématiquement présents dans le problème. Néanmoins, quelques travaux proposent des pistes pour répondre à certaines de ces problématiques.

Nos travaux visent à fournir un formalisme assez général pour couvrir le spectre des problèmes allant de la conception à la configuration.

Plan de lecture

Ce mémoire synthétise nos travaux sur l'intégration des différentes approches à base de contraintes utilisées pour l'aide à la conception ou à la configuration. Il se décompose en deux parties comprenant chacune trois chapitres (la figure 1 page suivante illustre le plan de lecture) :

- Partie I : état de l'art et problématique : conception, configuration et approches à base de contraintes
  - le premier chapitre introduit les notions de conception et de configuration, et identifie les besoins pour fournir des outils d'aide à la conception et à la configuration : paramètres discrets et continus, organisation hiérarchique et éléments optionnels ;
  - le deuxième chapitre aborde les approches par contraintes répondant au premier besoin : les CSP discrets et continus ;
  - le troisième chapitre présente une étude bibliographique sur la gestion d'éléments optionnels à base de contraintes.
- Partie II : propositions de modélisation et implémentation pour la résolution de problèmes d'aide à la conception ou à la configuration
  - le quatrième chapitre s'attache à intégrer les différentes variantes de CSP étudiées dans la première partie au sein d'un formalisme que nous proposons : les RCSP. Ce formalisme regroupe les CSP mixtes et les approches permettant de gérer des éléments optionnels. Nous dégageons ensuite des préconisations de modélisation pour les différents éléments d'un problème de conception ou de configuration ;
  - le cinquième chapitre présente l'implémentation que nous avons développée pour traiter des CSP mixtes — cette implémentation permet de considérer des contraintes très diverses représentées sous la forme d'arbres syntaxiques ;
  - le dernier chapitre propose des compléments à l'implémentation pour exploiter pleinement le formalisme RCSP modélisant des problèmes de conception ; des évaluations de cette implémentation sur quelques exemples simples complétent ce chapitre.

Dans ce mémoire, nous utiliserons l'acronyme anglophone CSP pour « Problème de Satisfaction de Contraintes », que ce soit au singulier ou au pluriel.

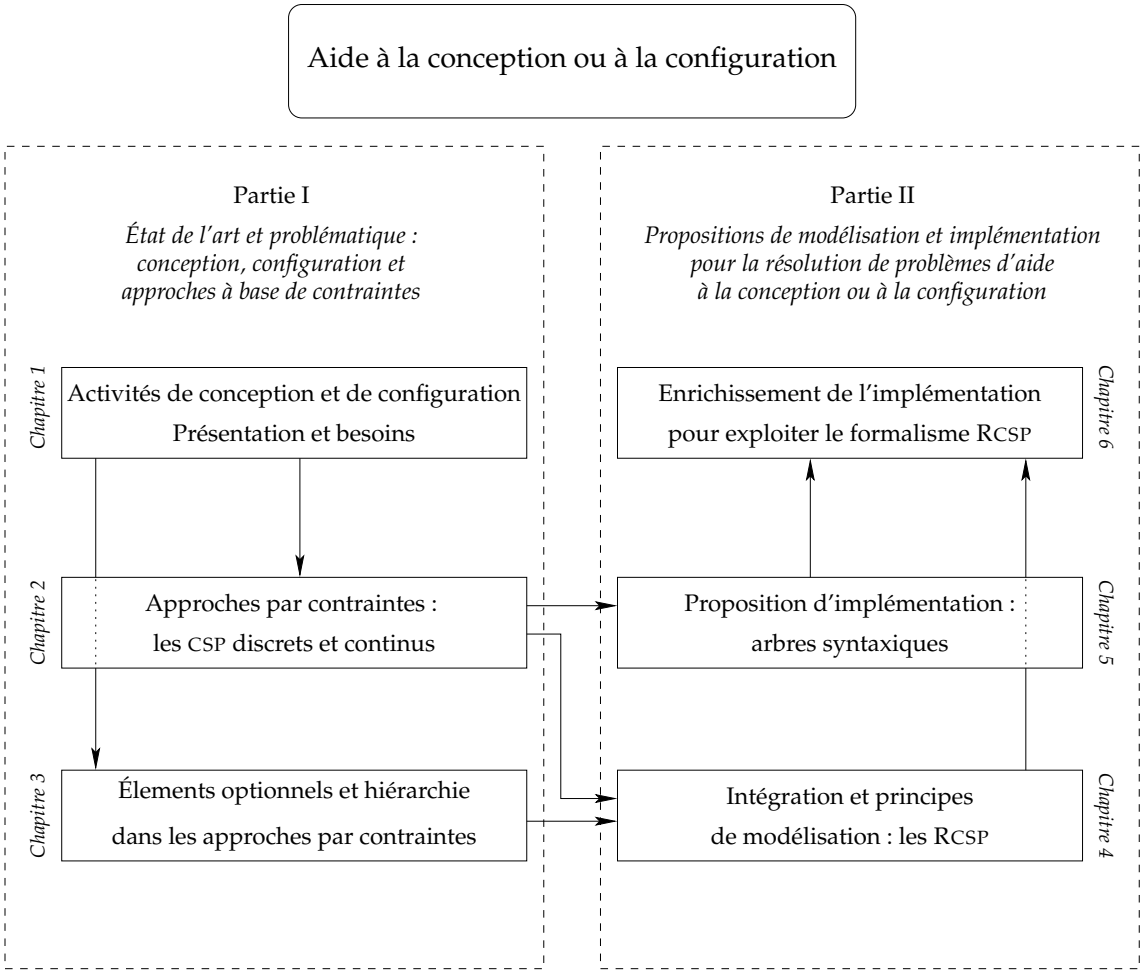
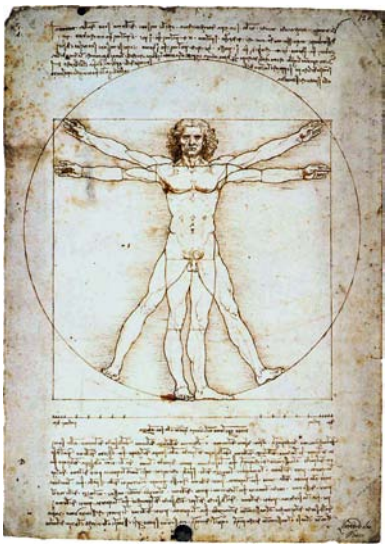


FIG. 1 — Plan de lecture du mémoire

Première partie

Conception, configuration et approches à base de contraintes : état de l'art et problématique



Chapitre 1

Activités de conception et de configuration : présentation et besoins

L'activité de conception comporte deux phases : la définition puis le développement d'un produit. Selon Pahl & Beitz (1996), le produit (ou service, procédé...) doit répondre à un ensemble de spécifications décrites dans un cahier des charges : spécifications techniques et spécifications fonctionnelles. La rédaction d'un cahier des charges est une tâche complexe ; elle consiste à recueillir les besoins des futurs utilisateurs du produit (clients) et la façon dont l'entreprise peut y répondre. Ainsi, la conception est fortement reliée aux problématiques organisationnelles et stratégiques d'une entreprise, ainsi qu'à ses savoir-faire. La figure 1.1 illustre le processus de conception simplifié par Hadj-Hamou (2002) issu du processus identifié par Pahl & Beitz (1996) : une première étape consiste à dégager du cahier des charges les fonctionnalités attendues du produit (objectif). Il faut ensuite identifier des solutions technologiques pour remplir cet objectif (domaine). La dernière étape consiste à évaluer les différentes solutions, puis à en sélectionner une. Choisir une de ces solutions fait appel aux connaissances particulières de l'entreprise et à ses savoir-faire (démarche).

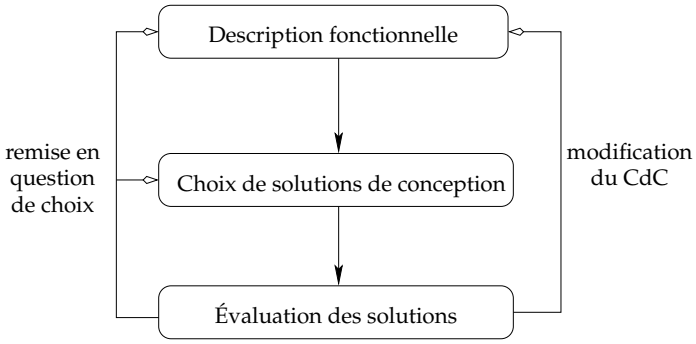


Fig. 1.1 — Extrait du processus de conception (Pahl & Beitz, 1996)

Dans ce cadre, l'aide à la conception consiste à fournir à un utilisateur des outils permettant d'assister tout ou partie du processus de conception.

Au cours de ce chapitre, nous présenterons différents types de conception. Nous en viendrons aux différentes techniques d'utilisation des connaissances pour l'aide à la conception : le raisonnement à base de cas (connaissances implicites), les systèmes experts et les raisonnements à base de contraintes (connaissances explicites). Nous introduirons ensuite un type de conception particulier : la configuration. Nous conclurons ce chapitre sur une identification des besoins pour l'aide à la conception.

1.1 Les différents types de conception

CHANDRASEKARAN (1990) a identifié trois types de conception, différenciés par la connaissance initiale que l'on a de l'objectif, du domaine et de la démarche : les conceptions routinière, innovante ou créative.

L'objectif est composé des fonctionnalités attendues du produit et le domaine des solutions techniques ou technologiques pouvant permettre de réaliser le produit. La démarche correspond aux différentes compétences et savoir-faire de l'entreprise permettant d'exploiter la connaissance du domaine pour répondre à l'objectif.

1.1.1 Conception créative

En conception créative, le concepteur ne dispose pas de connaissances sur le produit à concevoir. Le produit n'existe pas et n'a pas d'équivalent sur le marché, seul l'objectif est connu. Les technologies à utiliser pour que le produit puisse réaliser les fonctions pour lesquelles il est conçu sont à identifier et le concepteur doit s'approprier ces technologies.

Ce type de conception est le processus à mettre en œuvre pour un tout nouveau produit. Pour ce type de conception, les concepteurs procèdent généralement par essais successifs (processus itératif).

1.1.2 Conception innovante

En conception innovante, nous disposons de plus de connaissances sur le futur produit : son objectif ainsi que les technologies qui vont permettre de réaliser le produit sont connus, mais les stratégies de conception restent à définir. Lors d'un processus de conception innovante, il est possible d'utiliser des méthodes issues de la théorie de la résolution des problèmes d'invention (TRIZ, cf. MOEHRLE (2005)).

Ce type de conception est le processus à mettre en œuvre pour la mise sur le marché d'un produit correspondant à un besoin exprimé par les clients, mais non encore satisfait. Pour ce type de conception, les entreprises ont généralement peu d'expérience sur le produit à concevoir.

1.1.3 Conception routinière

En conception routinière, l'objectif du produit est parfaitement connu, ainsi que les technologies permettant de le réaliser. Nous parlons de conception routinière lorsque les concepteurs ont une certaine habitude et des savoir-faire sur la démarche de conception d'une même famille de produits.

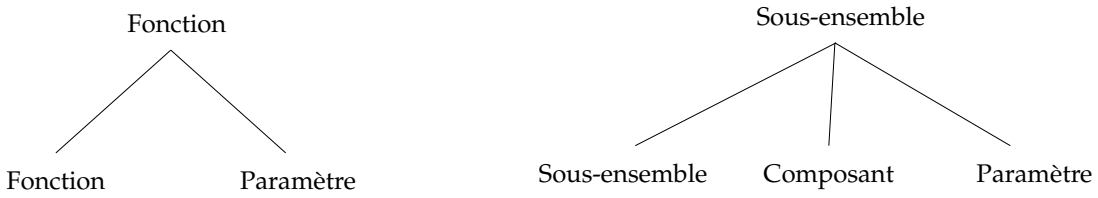
Ce type de conception concerne généralement un produit de complément de gamme, de remplacement ou d'améliorations incrémentales.

1.1.4 Conclusion

Pour la suite de ce mémoire, nous nous limiterons au domaine de la conception routinière. C'est en effet pour ce type de conception que les connaissances dont nous disposons sont les plus importantes. Ces connaissances peuvent être capitalisées puis exploitées pour aider à la conception.

En conception, un produit peut être vu de différentes façons, en particulier :

- selon une vue fonctionnelle : le produit est décrit par un ensemble de fonctions, qui sont elles-mêmes un ensemble de sous-fonctions ; les fonctions et sous-fonctions sont caractérisées par des paramètres (décrits par un domaine de définition ou une liste de valeurs) ;
- selon une vue physique : le produit est décrit par un ensemble de sous-ensembles, qui sont eux-mêmes un ensemble de sous-ensembles, de composants (un composant est une partie non-décomposable du produit) et de paramètres.



(a) Vue fonctionnelle (b) Vue physique

FIG. 1.2 — Les différentes vues d'un produit

Ces deux vues sont définies de façon hiérarchique et récursive (cf. figure 1.2). Un produit est donc caractérisé par un ensemble de paramètres (vue fonctionnelle) et/ou décrit par un ensemble de composants et paramètres ou nomenclature (vue physique). Des contraintes permettent de relier les différents éléments (fonctions, paramètres, composants, sous-ensembles) entre eux. Ces connaissances sont assemblées dans un « modèle de connaissances ».

La conception consiste à valuer des paramètres ou choisir des composants décrivant le produit, tout en respectant les contraintes. Aider à la conception revient donc à aider un utilisateur à valuer les paramètres ou choisir les composants, en ne lui proposant que les choix respectant les contraintes.

1.2 Exploitation des connaissances et aide à la conception routinière

Nous pouvons identifier deux techniques d'exploitation de connaissances pour aider à la conception suivant la nature de ces connaissances : implicites (qui concernent les savoir-faire non écrits) ou explicites (connaissances exprimées formellement et conservées). Pour exploiter des connaissances, il faut pouvoir les modéliser. Nous présentons trois techniques

majeures, issues de travaux en intelligence artificielle : le raisonnement à base de cas, qui exploite des connaissances implicites et explicites, les systèmes experts et les approches par contraintes, qui exploitent des connaissances explicites.

1.2.1 Exploitation de connaissances implicites

La principale méthode de gestion de connaissances implicites en aide à la conception est le raisonnement à base de cas (*Case-Based Reasoning*, cf. KOLODNER, 1993; WATSON & MARIR, 1994; AAMODT & PLAZA, 1994).

Raisonnement à base de cas

Pour ce type d'approche, il faut constituer une base de données correspondant à des instances réelles de solutions appelées « cas » ; un cas est décrit par un ensemble de descripteurs (qui correspondent à des paramètres ou composants). Lors de la conception d'un nouveau produit, certains descripteurs connus du nouveau produit sont comparés avec les différents descripteurs des produits contenus dans la base de cas. Il faut ensuite disposer d'une fonction d'évaluation ou de mesure de la similitude du nouveau produit avec ceux contenus dans la base de cas. Seuls les cas les plus proches du produit à concevoir sont identifiés et retournés à l'utilisateur. L'étape suivante consiste à sélectionner un ou des cas proches du nouveau produit et à les adapter aux descripteurs du nouveau produit à concevoir.

Enfin, la dernière étape consiste à enrichir la base de cas avec le produit conçu.

1.2.2 Exploitation de connaissances explicites

Extraction des connaissances

La gestion de connaissances explicites implique un premier travail d'extraction et d'explicitation de la connaissance. Cette étape s'effectue généralement par la recherche de connaissances dans la littérature ou par le biais d'entretiens avec les experts du domaine. Il existe donc deux catégories de connaissances :

- les lois prouvées ou admises (principe fondamental de la dynamique — deuxième loi de NEWTON, théorème de PYTHAGORE, axiomes d' EUCLIDE...)
- les règles issues de l'expérience (abaques...), des savoir-faire (de l'entreprise, des experts...), ou de choix des concepteurs (règles technologiques ou marketing).

Les lois relevant du domaine technologique dans lequel se situe le futur produit sont généralement bien connues. En revanche, extraire les connaissances auprès des experts est un travail long et fastidieux. Les modeleurs du modèle de connaissance (cogniticiens) doivent interviewer les experts pour leur « arracher » leurs connaissances et savoir-faire. Une fois ces connaissances extraites, il faut alors les valider et les modéliser.

Raisonnement à base de règles

Dans les systèmes experts, les connaissances extraites sont mises sous forme de faits et règles.

Les règles rassemblent la connaissance et le savoir-faire des experts. Ce sont des formules logiques simples, exprimées sous la forme d'une implication : prémisse $\Rightarrow$ conclusion.

Les faits représentent le domaine d'application : ce sont des axiomes logiques modélisant des faits connus ou admis, des faits déduits d'autres faits par les règles ou des faits fournis par l'utilisateur (choix de l'utilisateur).

Un moteur d'inférence permet d'enchaîner les faits et les règles pour aboutir à une conclusion (appelée but), dans notre cas une décision de conception. Le moteur d'inférence peut partir des faits existants et fournis par l'utilisateur pour conclure de nouveaux faits et atteindre un but ; nous parlons alors de chaînage avant. Il est aussi possible de partir du but et remonter aux faits initiaux ; nous parlons alors de chaînage arrière. Sur des systèmes simples (logique d'ordre 0), un des deux types de chaînage suffit à déduire le bon raisonnement des faits aux buts.

L'application à la conception des raisonnements à base de règles (ou systèmes experts) a été détaillée dans BROWN & CHANDRASEKARAN (1989).

Raisonnement à base de contraintes

Dans les raisonnements à base de contraintes, les connaissances extraites sont exprimées sous la forme de contraintes qui peuvent être des formules logiques, des expressions mathématiques ou des listes de tuples acceptables ou refusés.

Un réseau de contraintes, représentant à la fois la connaissance du domaine et le savoir-faire, va regouper des variables pourvues d'un domaine de définition et des contraintes. Ce réseau va ainsi caractériser un ensemble de solutions.

Un moteur raisonne alors sur ce réseau de contraintes pour aboutir à une ou plusieurs solutions respectant l'ensemble des contraintes. Il existe plusieurs types de moteurs :

- les moteurs autonomes : ceux-ci peuvent trouver une ou toutes les solutions du problème de conception, tout en respectant les contraintes ;
- les moteurs interactifs : ceux-ci aident l'utilisateur à faire ses choix interactivement — après chaque choix de l'utilisateur, le moteur utilise les contraintes pour ne présenter à l'utilisateur que les choix qui restent cohérents avec le système de contraintes.

Ainsi, le fonctionnement est similaire à celui des systèmes experts, mais une certaine cohérence des choix de l'utilisateur est assurée.

1.2.3 Discussion et raisonnement retenu

Le principal avantage du raisonnement à base de cas est le fait que les connaissances implicites peuvent être traitées, et n'ont pas besoin d'être formalisées pour être utilisées. La mise en place de l'outil en lui-même est donc assez rapide et la base de cas se maintient ensuite facilement par l'ajout des nouveaux produits (ou la suppression de cas devenus obsolètes). Cependant, la base de cas, pour être efficace, doit être relativement complète. Dans l'idéal, il faut donc disposer d'un grand nombre de cas : ceux-ci doivent couvrir de manière satisfaisante l'espace des solutions. De plus, la fonction d'évaluation de la distance entre un cas et le nouveau produit est assez complexe, et on ne peut aucunement garantir son optimalité, c'est-à-dire que le cas identifié est bien le cas le plus proche (en effet, il est possible qu'elle « oublie » certains cas assez proches). Enfin, une fois que l'on dispose de cas existants proches du nouveau produit, leur adaptation peut ne pas être triviale et nécessite un certain savoir-faire.

Les systèmes experts et les approches à base de contraintes nécessitent un effort d'extraction des connaissances important. Cette étape d'extraction des connaissances, en particulier par le biais d'entretiens avec des experts (scientifiques, techniques, mercatiques...) demande

un gros effort. Cet effort d'abstraction est très important, ce qui fait que de tels outils d'aide à la conception sont souvent complexes à mettre en place. Nous pouvons souligner que :

- les régles ou les contraintes mélangent des connaissances du domaine et des stratégies de résolution ;
- la connaissance sur un composant peut être décomposée en plusieurs régles ou contraintes.

Cependant, ils ont l'avantage de permettre un « dialogue » entre le moteur d'inférence ou de propagation et l'utilisateur pour aider ce dernier à faire ses choix (mode interactif). Les systèmes experts ont plusieurs inconvénients :

- un jeu de règles peut être incohérent (MAILHARRO, 1998) ;
- sur des systèmes complexes, même l'utilisation combinée des deux chaînages ne parvient pas toujours à une solution ;
- ces systèmes logiques ne sont pas à même de traiter des problèmes numériques ou quantifiés : le pouvoir expressif des systèmes experts se limite à des faits et règles discrètes ;

inconvénients que les raisonnements à base de contraintes permettent de contre-balancer par trois avantages majeurs :

- le maintien de la cohérence[1] pour toutes les nouvelles solutions obtenues ;
- sous certaines conditions, il est possible d'atteindre toutes les solutions d'un système, ou de prouver que ce système n'a plus de solutions.
- les contraintes peuvent prendre des formes très diverses : formules logiques, expressions mathématiques, listes de tuples...

Comme dans un certain nombre de cas (MULYANTO (2002) pour la conception d'avions et LOTTAZ *et al.* (1999) pour la conception en génie civil, par exemple, ou VERNAT (2004) pour une démarche plus générale) le domaine de conception est connu et les connaissances sont déjà exprimées sous une forme utilisable pour une modélisation par contraintes, nous ferons abstraction dans ce mémoire de l'étape d'extraction et de validation des connaissances.

La suite de ce mémoire se concentre donc sur les outils d'aide à la conception à base de contraintes pour l'aide à la conception.

1.3 Aide à la conception et aide à la configuration

Nous allons maintenant introduire un type particulier de conception routinière : la configuration. La configuration concerne un produit déjà conçu pour lequel il existe plusieurs variantes ; elle fait donc suite à un processus de conception.

Pour la conception, l'espace des solutions est *a priori* infini, puisque le produit à concevoir n'existe pas. Le travail de conception peut être vu comme un processus de restricion de cet espace de recherche. Pour la configuration, la diversité produit est connue. L'espace des solutions est donc fini et borné et correspond à un ensemble de composants, de paramètres et de sous-ensembles. Lors d'un processus de configuration, l'utilisateur saisit ses souhaits pour obtenir la définition d'un produit fini sous forme de nomenclature ; celle-ci doit être cohérente avec les contraintes du problème.

[1] Ici, la cohérence consiste à respecter les règles ou contraintes de faisabilité du produit.

La configuration est donc un cas extrême de la conception routinière, pour lequel les connaissances décrivant l'objectif, le domaine et la démarche sont formalisées dans un modèle produit. La littérature sur les raisonnements à base de contraintes en conception se base sur ces deux domaines d'application proches :

- l'aide à la conception — la vue fonctionnelle est alors préférée (en début de conception ; la vue physique doit aussi être prise en compte pour le choix des composants) : un produit est un ensemble de paramètres (souvent continus) et de fonctions ;
- l'aide à la configuration — dans ce cas, la vue physique est plus utilisée : un produit est un ensemble de sous-ensembles et de composants (souvent choisis dans une liste).

Nous nous basons sur les définitions suivantes pour effectuer une comparaison entre conception et configuration.

Définition 1.1 : *Conception*

Un problème de conception est spécifié par

(i) un ensemble de fonctions (exigées par le client et le consommateur du produit ou définies implicitement par le domaine) que doit remplir le produit et un ensemble de contraintes à satisfaire,

(ii) un ensemble de composants.

*Les contraintes peuvent porter sur les paramètres de définition du produit, sur le processus de réalisation du produit ou sur le processus de conception. La solution d'un problème de conception est définie par une spécification complète d'un ensemble de composants et de leurs relations (*CHANDRASEKARAN, *1990).*

Définition 1.2 : *Configuration*

*Étant donné un modèle représentant le produit générique, la configuration consiste à capturer, de manière cohérente vis-à-vis du modèle, les souhaits de l'utilisateur pour aboutir à la définition d'un produit réalisable en termes d'une nomenclature de fonctions et/ou de produits (*VERON, *2001).*

De ces définitions, nous pouvons proposer une première frontière entre conception et configuration.

Comme l'illustre la figure 1.3, la différence entre conception et configuration peut se mesurer en termes de degrés de liberté et de connaissances. Ainsi, lors de la conception d'un produit, nous disposons de beaucoup de libertés, mais de peu de connaissances sur le produit lui-même. En revanche, en situation de configuration, le produit est bien connu et l'utilisateur ne dispose que de peu de libertés de choix. En conception et en configuration, nous pouvons cependant identifier les mêmes acteurs, bien que leurs rôles diffèrent un peu :

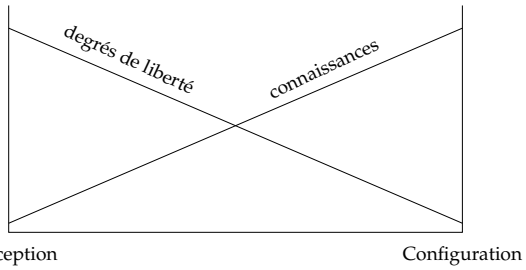


FIG. 1.3 — Connaissances et degrés de liberté selon le domaine

- le modeleur, chargé de construire le modèle de connaissances pouvant être utilisé ensuite pour l'assistance ;

- les experts devant communiquer leurs savoirs au modeleur pour qu'il puisse extraire et modéliser leurs connaissances ;
- l'utilisateur :
  - dans le cas de l'aide à la conception, le groupe de conception va utiliser les connaissances modélisées ainsi que ses propres connaissances pour concevoir son modèle de produit,
  - dans le cas de l'aide à la configuration, celui-ci va communiquer ses souhaits pour obtenir un produit configuré correspondant à ses attentes à partir du modèle de configuration.

Nous pouvons donc modéliser le processus d'aide à la conception ou d'aide à la configuration d'une façon similaire, illustrée par la figure 1.4.

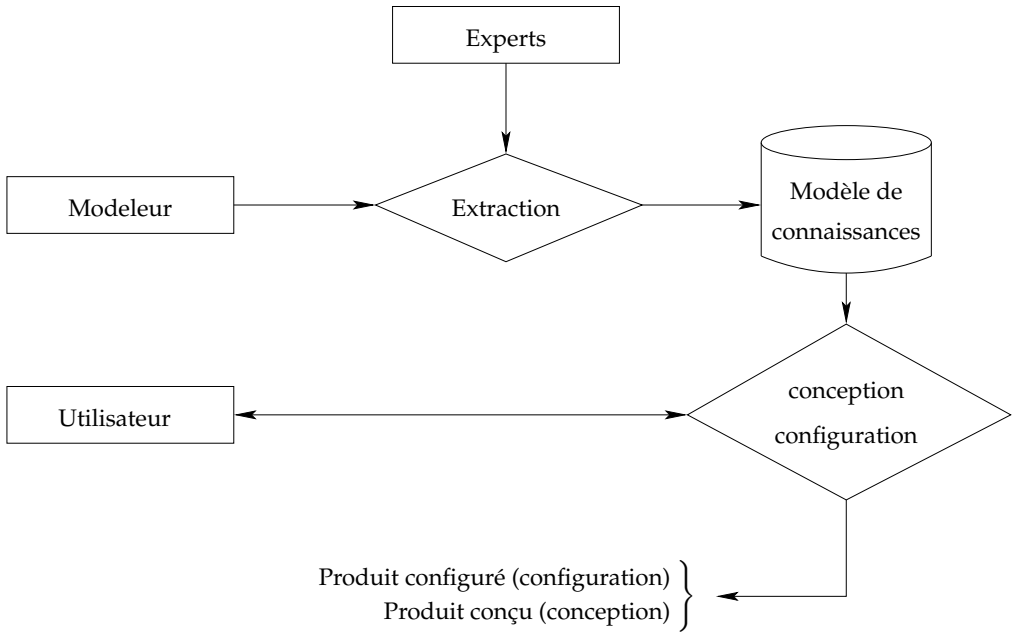


FIG. 1.4 — Processus d'aide à la conception ou de configuration

L'objectif d'un tel processus est d'arriver à une description précise du produit. Cette description comporte une liste des différents composants et paramètres valués que l'on peut assembler pour réaliser le produit final. Ces composants ou paramètres valués sont identifiés à partir du modèle de connaissances, qui peut donc être considéré comme un modèle générique du produit (à concevoir ou à configurer). Ce modèle générique représente donc les connaissances nécessaires aux processus de conception ou de configuration.

Définition 1.3 : ***Modèle générique***

Ensemble cohérent décrivant les propriétés et comportements du produit selon un point de vue (fonctionnel, structurel, technologique) TOLLENAERE *(1994).*

BERNARD (2000) propose une synthèse des différentes orientations de modèles de produits utilisables. De même, il existe plusieurs méthodes de modélisation d'un produit, parmi lesquelles nous pouvons citer MERISE, SADT, FAST, UML...

Du point de vue de la configuration, la phase de conception s'apparente à l'identification d'un modèle générique de produit, modélisé dans une des méthodes précédemment citées.

Ainsi, la configuration est toujours précédée d'une phase de conception, au cours de laquelle nous identifions le modèle générique du produit. La phase de configuration proprement dite consiste, à partir du modèle générique modélisé par un système de contraintes, à obtenir un produit fini (une des variantes du produit générique) correspondant aux souhaits de l'utilisateur.

La figure 1.3 page 13 illustre la différence entre conception et configuration du point de vue « degrés de liberté » et « connaissances ». Nous pouvons établir une deuxième frontière en termes de composants et de paramètres. Les paramètres constituent des éléments de modélisation du domaine ou de caractéristiques du produit.

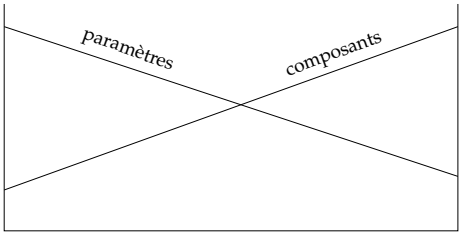


FIG. 1.5 — Paramètres et composants selon le domaine

Les composants représentent une modélisation de la diversité produit par des sous-ensembles ; ces sous-ensembles sont assemblés pour obtenir le produit. La figure 1.5 montre une autre différence entre conception et configuration : les paramètres, représentant une partie du produit, sont plus nombreux en conception, alors que les composants, représentant des sous-ensembles du produit, apparaissent plus en configuration. Les paramètres permettent de décrire le produit d'un point de vue fonctionnel, mais aussi chacun de ses composants, qui participent à la description physique du produit.

Nous pouvons donc considérer le modèle générique comme un ensemble de paramètres et de composants. Les processus de conception et de configuration consistent à déterminer lesquels utiliser.

Nous avons vu dans cette section que les processus d'aide à la conception et d'aide à la configuration pouvaient être considérés de la même manière. La principale différence se situe au niveau des connaissances à formaliser : celles-ci concernent plus le domaine technologique pour l'aide à la conception (CHANDRASEKARAN, 1990), et la diversité produit pour la configuration. Elles sont majoritairement exprimées en termes de paramètres lors de l'aide à la conception (qui nécessite aussi les connaissances du concepteur) et en termes de composants pour la configuration.

1.4 Besoins pour l'aide à la configuration et l'aide à la conception

Notre but est d'aider à la conception et à la configuration de produits, services ou procédés. Pour cela, nous utilisons des approches à base de contraintes, qui nécessitent l'extraction et la modélisation de la connaissance du produit sous la forme d'un modèle générique. Pour la construction et l'interprétation de ce modèle générique, nous avons besoin de manipuler différents types d'éléments.

Nous avons vu précédemment que le modèle générique pouvait être considéré comme un ensemble de paramètres (décrivant des fonctions du produit), de composants, mais aussi de sous-ensembles (de composants et/ou paramètres).

Les paramètres constituent les caractéristiques du produit (ou de composants ou sous-ensembles d'un produit) et sont généralement décrits par des domaines de définition (par exemple, la vitesse de rotation d'un moteur) ou une liste de valeurs (par exemple, la couleur d'une voiture).

Les composants sont des parties non décomposables du produit ; ils sont généralement achetés tels quels par le fabricant ; celui-ci assemble alors les différents composants pour obtenir un produit (fini ou semi-fini). Il existe trois types de composants :

standard : les composants standard sont à choisir dans une liste (par exemple, un microprocesseur de station de travail) ou peuvent être exprimés en termes de présence (par exemple, le vélo a-t-il un garde-boue ?) — les paramètres d'un composant standard sont tous figés ;

paramétrables : nous distinguons deux types de composants paramétrables : les composants à un paramètre (généralement continu, par exemple un type de profilé d'aluminium avec sa longueur) et les composants à plusieurs paramètres — un composant paramétrable a au moins un paramètre non figé ;

optionnels : les composants optionnels peuvent ne pas faire partie du produit fini ; leur participation à la solution du problème de conception est conditionnée.

Les sous-ensembles sont issus de la vue hiérarchique du modèle du produit ; ils regroupent plusieurs paramètres ou composants (par exemple, un sous-ensemble « roue de vélo » regroupe trois composants : la jante, la chambre à air et le pneu ; et deux paramètres : diamètre et largeur).

Pour réaliser un modèle générique, nous avons donc besoin de pouvoir modéliser :
- des éléments de différentes natures ;
- des sous-ensembles d'éléments ;
- la participation à la solution d'éléments optionnels.

Nous entendons par élément d'un problème de conception un composant, un paramètre, un sous-ensemble ou une contrainte liant des composants et/ou paramètres.

1.4.1 Des éléments de différentes natures

Le premier besoin identifié est de pouvoir traiter indifféremment plusieurs types d'éléments : des nombres (entiers, réels), symboles, booléens (particulièrement en configuration)... Nous pouvons ainsi classifier ces types, qui seront détaillés dans la section 2.1 page 21 :
- booléens ;
- symboliques ;
- numériques.

Il est possible d'avoir des paramètres numériques dont les valeurs sont dénombrables ou non-dénombrables. Pouvoir traiter ces types de paramètres implique de trouver des moyens d'exprimer les différentes solutions envisageables, ce qui peut poser problème lorsqu'ils ne sont pas dénombrables (cas des paramètres numériques continus).

Enfin, il doit être possible de pouvoir exprimer des contraintes entre tous ces types d'éléments. Ces contraintes doivent donc pouvoir être mixtes : des paramètres symboliques doivent pouvoir contraindre des paramètres numériques et vice versa (par exemple, le produit du diamètre par la largeur d'une roue peut contraindre le type de jante).

1.4.2 Utilisation d'un modèle arborescent

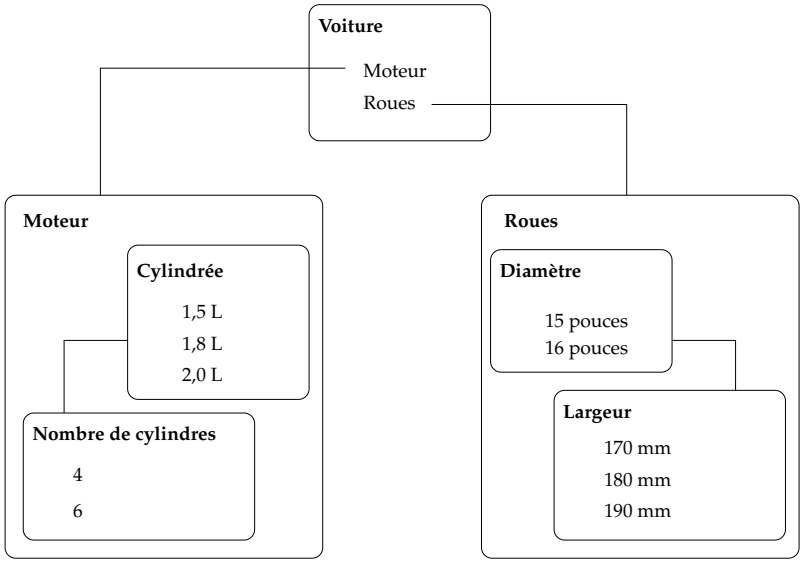
La hiérarchisation des modèles de connaissances doit nous permettre de gérer des composants ou paramètres par sous-ensembles, ce qui permet d'affiner la vision du produit au fur et à mesure du processus de conception. Ainsi, pour un certain sous-ensemble, l'utilisateur pourra choisir en premier lieu les paramètres influant les autres sous-ensembles (par

exemple, sur une roue, le diamètre), puis revenir plus tard sur le choix des autres paramètres ou composants de la roue (largeur, jante...).

Cet aspect est important en conception ou configuration, dans la mesure où les modèles produits sont généralement exprimés en termes hiérarchiques (paramètres formant des sous-ensembles du produit). L'exemple 1.6 illustre une partie d'un modèle de voiture composé de différents sous-ensembles.

EX. 1.6 — Moteur et roues d'une voiture — dérivé de SABIN & FREUDER (1996)

Une voiture est composée de roues et d'un moteur. Ces différents sous-ensembles comprennent à leur tour différents éléments paramétrables : la taille des roues (diamètre, largeur), la cylindrée et le nombre de cylindres du moteur.



Cet aspect peut avoir une influence sur les contraintes. En effet, certaines contraintes peuvent être attachées à un sous-ensemble du produit, et doivent donc être positionnées au niveau adéquat, de la même façon que les paramètres ou composants.

1.4.3 Prise en compte d'élements optionnels et notion de pertinence

Un produit générique possède souvent plusieurs options que l'utilisateur peut choisir d'intégrer ou non à son produit ; nous parlons alors de produit configurable. Les produits configurables comprennent des éléments optionnels, dont l'existence sera définie au cours du processus de configuration.

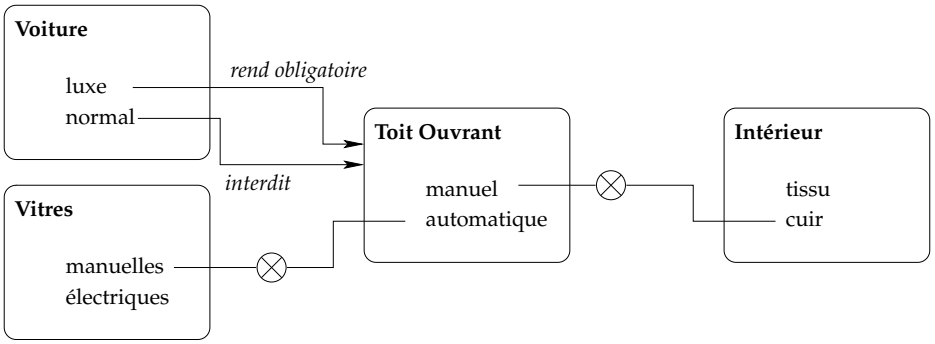
Prenons l'exemple de la voiture (cf. exemple 1.7 page suivante) dérivé de MITTAL & FALKENHAINER (1990). Le toit ouvrant est un composant optionnel, dont l'existence peut être contrainte par certains choix (par exemple, si la voiture est décapotable, il n'est pas possible d'avoir un toit ouvrant).

Dans certains modèles génériques, il peut être indispensable de modéliser l'existence ou non de contraintes entre éléments, sous certaines conditions. Prenons l'exemple du réacteur chimique (cf. exemple 1.8 page 19) inspiré de VAN VELZEN (1993) et présenté par GELLE (1998). Le calcul du volume de la cuve du réacteur est conditionné par la forme de cette

cuve. Étant donné que trois géométries de la cuve sont possibles, trois contraintes de calcul du volume co-existent dans le modèle. Une seule contrainte de calcul aura du sens lorsque la géométrie de la cuve aura été déterminée ; les deux autres ne seront alors pas prises en compte dans la solution de conception.

EX. 1.7 — Voiture simple — dérivé de MITTAL & FALKENHAINER (1990)

Considérons une voiture (luxe ou normale) équipée de vitres (manuelles ou électriques), d'un intérieur (cuir ou tissu) et d'un toit ouvrant (manuel ou automatique) disponible seulement sur la version « luxe ». Des vitres manuelles sont incompatibles avec un toit ouvrant électrique et un intérieur cuir est incompatible avec un toit ouvrant manuel. Ces deux dernières contraintes n'ont de sens que si le toit ouvrant est présent.



La flèche représente la pertinence conditionnée du composant *Toit Ouvrant* : le composant *Toit Ouvrant* est pertinent si et seulement si le composant *Voiture* est valué à « luxe ». Les relations représentées par ce symbole : ⊗ sont des combinaisons de valeurs interdites.

Ainsi, nous avons besoin de manipuler des éléments qui n'existent pas toujours dans la solution de conception. Nous devons aussi pouvoir interdire leur appartenance à la solution ou rendre leur présence obligatoire par le biais de contraintes sur d'autres éléments.

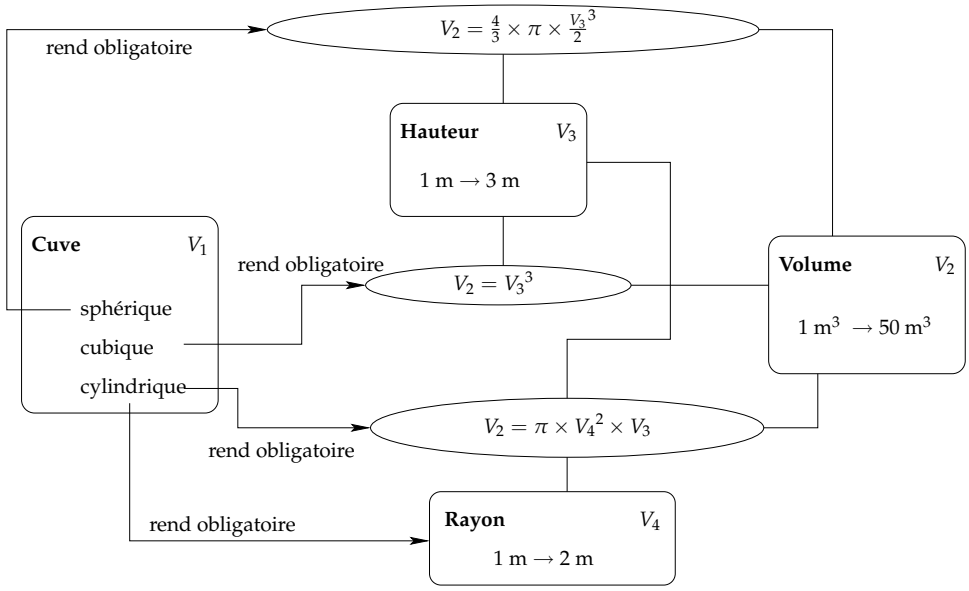
La notion de pertinence qui apparaît dans les exemples 1.7 et 1.8 page ci-contre peut se définir de la manière suivante :

Pertinence — définition fonctionnelle La pertinence d'un élément doit être vue du point de vue de l'utilisateur. Ainsi, certains éléments peuvent ne participer à aucune solution du problème. Ces éléments sont considérés comme non-pertinents. La gestion d'éléments optionnels consiste donc à déterminer leur pertinence ou leur non-pertinence au cours de la résolution d'un problème de conception.

Ce besoin d'éléments dont la pertinence est à déterminer se transpose sur les sous-ensembles d'éléments. Dans ce cas, il s'agit de rendre pertinents des sous-ensembles du produit pour ensuite choisir des composants ou valuer des paramètres tout en respectant les nouvelles contraintes introduites avec ce sous-ensemble.

EX. 1.8 — Réacteur chimique — dérivé de GELLE (1998)

Un réacteur chimique se compose d'une cuve, qui peut être cubique, sphérique ou cylindrique. Une contrainte reliée à chacune de ces valeurs permet de rendre pertinente la formule de calcul adaptée pour le volume de la cuve.



1.5 Conclusion

Dans ce chapitre, nous avons présenté les différents types de conception. Pour la conception routinière, nous avons constaté qu'il était possible de fournir des outils d'aide ou d'assistance basés sur un modèle de connaissances (raisonnement à base de cas, à base de règles ou à base de contraintes). Ces outils permettent d'exploiter la connaissance liée au produit de manière implicite ou explicite. Nos travaux exploitent les raisonnements à base de contraintes, qui utilisent des connaissances explicites sur l'objet à concevoir, pour fournir des outils d'aide à la conception de produits, services ou procédés.

Dans un deuxième temps, nous avons montré que la configuration est un type particulier de conception routinière. En effet, les processus de conception et de configuration sont similaires et basés sur un modèle générique de composants, paramètres et/ou sous-ensembles. Les mêmes outils d'assistance peuvent donc être utilisés pour ces deux domaines.

Nos travaux se positionnent donc sur l'aide à la conception routinière ou à la configuration, en utilisant des approches à base de contraintes.

Nous avons enfin identifié les besoins nécessaires à la réalisation et l'expression d'un modèle générique :

- le traitement d'éléments de différentes natures (paramètres discrets ou continus, symboliques ou numériques, composants standard ou paramétrables, sous-ensembles et contraintes) ;
- l'organisation du modèle, généralement exprimé en termes d'ensembles et de sous-ensembles de façon hiérarchique ;
- l'existence d'éléments optionnels, dont la pertinence est indéterminée au début du problème.

C'est la présence simultanée de ces trois aspects qui rend l'aide à la conception à base de contraintes délicate à mettre en œuvre.

Les deux chapitres suivants se focalisent sur les approches à base de contraintes permettant de construire le modèle générique du produit. Dans la suite de ce mémoire, nous emploierons indifféremment le terme de conception (plus générique) pour « conception », « aide à la conception », « configuration » et « aide à la configuration », de façon à éclaircir le discours.

Chapitre 2

Approches par contraintes : les CSP discrets et continus

Dans ce chapitre, nous allons présenter les approches à base de contraintes appellées CSP — *Constraint Satisfaction Problem*, Problème de Satisfaction de Contraintes. Les CSP permettent d'exprimer un modèle de connaissances à base de contraintes et de raisonner sur ces contraintes pour aider à la conception.

Dans un premier temps, nous montrerons comment modéliser un problème de conception grâce à un CSP. Nous aborderons ensuite différentes techniques de résolution de CSP discrets, puis différentes techniques de filtrage de CSP discrets, puis continus. Nous passerons rapidement sur des approches à base d'automates à états finis, pour en arriver aux techniques de résolution faisant appel au filtrage.

2.1 Modélisation et approches par contraintes

Mittal & Frayman (1989) ont montré qu'un CSP était un bon moyen de gérer les problèmes de configuration ; nous étendons cette notion à la conception, étant donné les similitudes entre ces deux domaines, vues au chapitre 1. Les CSP ont été introduits par Montanari (1974), puis repris par d'autres auteurs (Waltz, 1975; Mackworth, 1977; Mackworth & Freuder, 1985; Freuder, 1985).

Définition 2.1 : CSP

Un CSP *est modélisé comme un triplet* $(\boldsymbol{X}, \boldsymbol{D}, \boldsymbol{C})$ *tel que :*

- $\boldsymbol{X}$ *est un ensemble de variables ;*
- $\boldsymbol{D}$ *est l'ensemble des domaines des variables de* $\boldsymbol{X}$ *;*
- $\boldsymbol{C}$ *est un ensemble de contraintes sur les variables de* $\boldsymbol{X}$.

Pour les contraintes, il existe deux définitions dans la littérature : pour les CSP discrets, elles sont généralement considérées comme une liste de n-uplets (assignations variable–valeur) acceptables — on parle alors de table de compatibilité (disjonction de conjonctions)

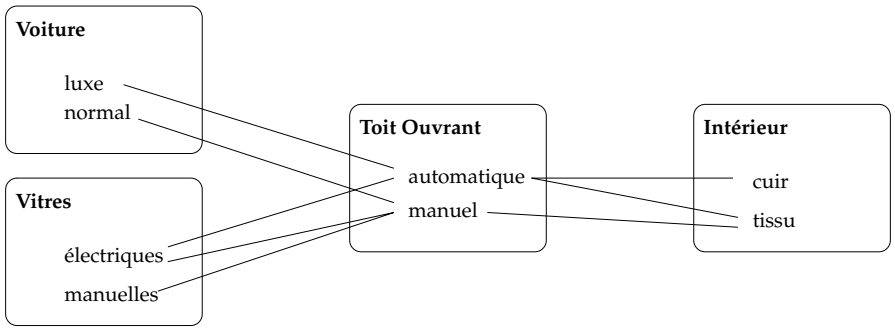
— ou une liste de n-uplets refusés (TSANG, 1993). Pour des CSP numériques, les contraintes sont généralement exprimées sous forme de formules mathématiques.

Nous étendons ces deux définitions pour proposer la définition 2.2, élaborée à partir de différents travaux portant sur la configuration : FARGIER & HENOCQUE (2002); ESTRATAT & HENOCQUE (2004).

Définition 2.2 : *Contrainte*

Une contrainte est une condition logique sur un ensemble de variables.

EX. 2.1 — Modèle de configuration très simple



La figure ci-dessus reprend le problème de configuration de voiture (exemple 1.7 page 18) dans lequel le toit ouvrant serait obligatoirement présent. La voiture peut être de type luxe ou normal. Le type du toit ouvrant (manuel ou automatique) dépend du type de la voiture. Le type du toit ouvrant interdit certains types de vitres ou d'intérieur : il n'existe pas de voiture ayant un toit ouvrant automatique sans vitres électriques et il n'existe pas de voiture ayant un intérieur cuir avec un toit ouvrant manuel. Les liaisons représentent les combinaisons autorisées entre deux variables.
À partir de la description du problème, nous pouvons identifier les différents éléments du CSP : les variables, les domaines des variables (ensemble des valeurs initialement admissibles pour chacune des variables) et les contraintes entre les variables :

$$
\begin{array}{l}
\text{L'ensemble de variables :} \\
\boldsymbol{X} \quad = \quad \{\text{Voiture, Vitres, Intérieur, Toit ouvrant}\} \\
\text{L'ensemble des domaines :} \\
\boldsymbol{D} \quad = \quad \{\{\text{normal, luxe}\}, \{\text{manuelles, électriques}\}, \\
\qquad\quad \{\text{cuir, tissu}\}, \{\text{manuel, automatique}\}\} \\
\text{L'ensemble } \boldsymbol{C} \text{ des contraintes :} \\
\text{Le type de voiture contraint le type de toit ouvrant :} \\
C_1 \quad : \quad \text{Voiture} = \text{luxe} \Leftrightarrow \text{Toit ouvrant} = \text{automatique} \\
\text{Le type du toit ouvrant contraint le type des vitres :} \\
C_2 \quad : \quad \text{Toit ouvrant} = \text{automatique} \Rightarrow \text{Vitres} = \text{électriques} \\
\text{Le type d'intérieur contraint le type de toit ouvrant :} \\
C_3 \quad : \quad \text{Intérieur} = \text{cuir} \Rightarrow \text{Toit ouvrant} = \text{automatique}
\end{array}
\tag{2.1}
$$

La phase de modélisation consiste à identifier les variables et leurs domaines de définition représentant les composants, paramètres, fonctions et sous-ensembles du problème,

ainsi que l'ensemble des contraintes portant sur ces variables. L'exemple 2.1 présente un modèle de configuration adapté de la voiture de MITTAL & FALKENHAINER (1990).

Définition 2.3 : *Solution d'un* CSP

Une solution d'un CSP *est une instanciation de toutes les variables telle que toutes les contraintes sont satisfaites.*

Sur l'exemple 2.1, l'ensemble des solutions peut être représenté par le tableau 2.2.

TAB. 2.2 — Ensemble des solutions de l'exemple 2.1

Variables / Solutions	Voiture	Vitres	Intérieur	Toit Ouvrant
Solution n° 1	normale	électriques	tissu	manuel
Solution n° 2	normale	manuelles	tissu	manuel
Solution n° 3	luxe	électriques	tissu	automatique
Solution n° 4	luxe	électriques	cuir	automatique

Une variable est caractérisée par son domaine de définition. Les techniques de traitement vont varier selon le type de ces variables : discrètes, continues, symboliques, numériques. La figure 2.3 illustre une cartographie des différents types de variables (VAREILLES, 2005).

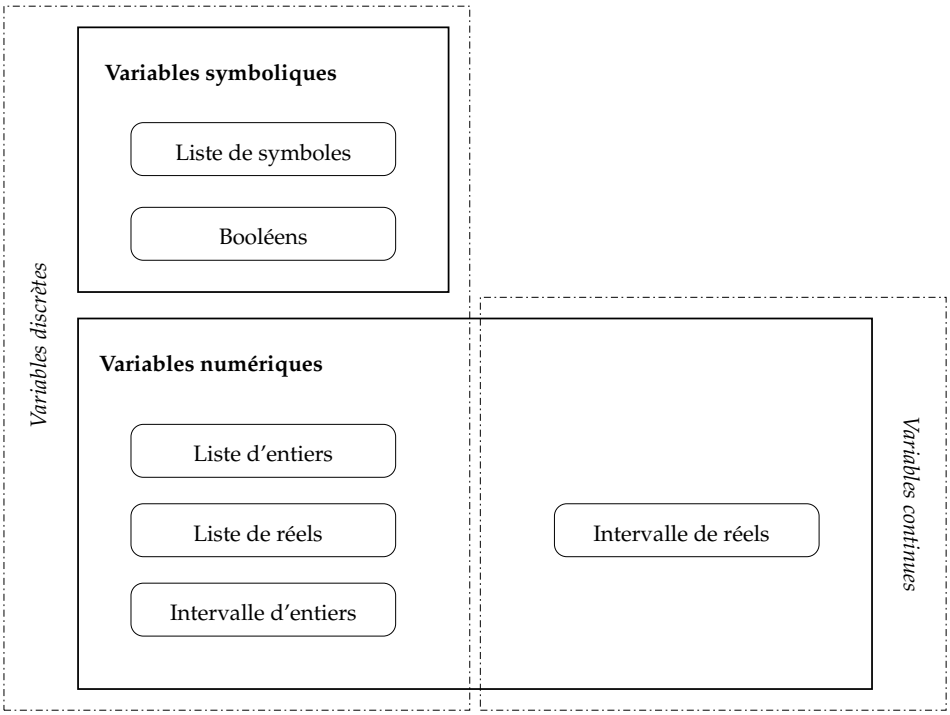


FIG. 2.3 — Classification des variables selon leur type (VAREILLES, 2005)

Nous retrouvons sur la figure 2.3 les trois types de CSP identifiés par GELLE (1998) :
– les CSP discrets, dont toutes les variables ont des domaines symboliques (par exemple, la couleur : rouge, vert ou bleu) ou numériques entiers (par exemple, le nombre de

tiroirs dans un meuble : 1, 2, 3 ou 4) ou réels (différentes longueurs : 1, 1 m, 1, 2 m ou 1, 3 m) ;
- les CSP continus, dont toutes les variables ont des domaines infinis (par exemple, une longueur entre 1 m et 2 m) ;
- les CSP mixtes, dont certaines variables sont discrètes et d'autres continues.

Deux questions émergent de cette classification :
- doit-on considérer les variables numériques entières comme des variables numériques réelles, de façon à pouvoir effectuer des calculs ?
- comment considérer les variables dont les domaines sont discrets et infinis ?

Dans la suite de ce mémoire, nous traiterons les variables numériques entières comme les réelles, ce qui nous autorisera les calculs sur ces variables. En revanche, nous ne trancherons pas nettement la deuxième question, à savoir comment traiter des variables discrètes à domaines infinis. Pour pallier ce point, nous proposons d'utiliser les complémentaires de domaines finis.

Types d'assistance

Nous identifions deux types d'assistance pour l'aide à la conception : autonome ou interactive.

La conception autonome consiste, à partir du modèle générique (éventuellement augmenté de certains choix de l'utilisateur ; des valuations de variables, par exemple) à trouver une ou toutes les solutions acceptables.

La conception interactive consiste à retirer du domaine des variables les valeurs qui ne sont plus cohérentes avec les choix de l'utilisateur. Seules les valeurs encore cohérentes sont présentées à l'utilisateur, qui peut alors soit exprimer de nouveaux choix, soit lancer une conception de type autonome lorsqu'il juge que les choix restants ne sont pas importants.

Les deux types de conception (autonome et interactive) font respectivement appel à deux techniques de traitement des CSP différentes : la résolution et le filtrage.

2.4 : *Résolution d'un* CSP

Résoudre un CSP *consiste à trouver une ou toutes les solutions possibles (autorisées par les contraintes).*

2.5 : *Filtrage d'un CSP*

Le filtrage consiste à réduire les domaines de définition des variables en tentant d'y supprimer les valeurs qui n'interviennent dans aucune solution.

Nous pouvons remarquer que la recherche de toutes les solutions sur des CSP contenant des variables continues pose des difficultés (mathématiques et de temps de calcul) : il faudrait identifier tous les éléments d'un ensemble infini... La recherche de solutions est complète et exacte pour les CSP discrets (problème NP-complet) alors qu'elle est approchée pour les CSP continus. En effet, les techniques permettant de résoudre des CSP continus font généralement appel à la discrétisation des domaines.

Nous allons maintenant présenter plusieurs algorithmes pour la résolution et le filtrage de CSP discrets et quelques méthodes de filtrage de CSP numériques.

2.2 Recherche de solutions

2.2.1 *Generate & Test*

Cet algorithme vérifie la validité d'une solution une fois que toutes les variables sont instanciées : toute la combinatoire est générée et testée. Ce mode de fonctionnement peut constituer une référence par rapport à d'autres algorithmes (SAGAUT, 1996) que nous verrons plus tard, mais il n'est en aucune façon satisfaisant : un tel processus de résolution est nécessairement exponentiel.

La figure 2.4 montre le processus utilisé par l'algorithme *Generate & Test* sur l'exemple 2.1 page 22. Toutes les solutions sont générées puis testées par rapport aux contraintes du CSP. On élimine alors les combinaisons violant les contraintes. Nous retrouvons bien l'ensemble des solutions présenté dans le tableau 2.2 page 23.

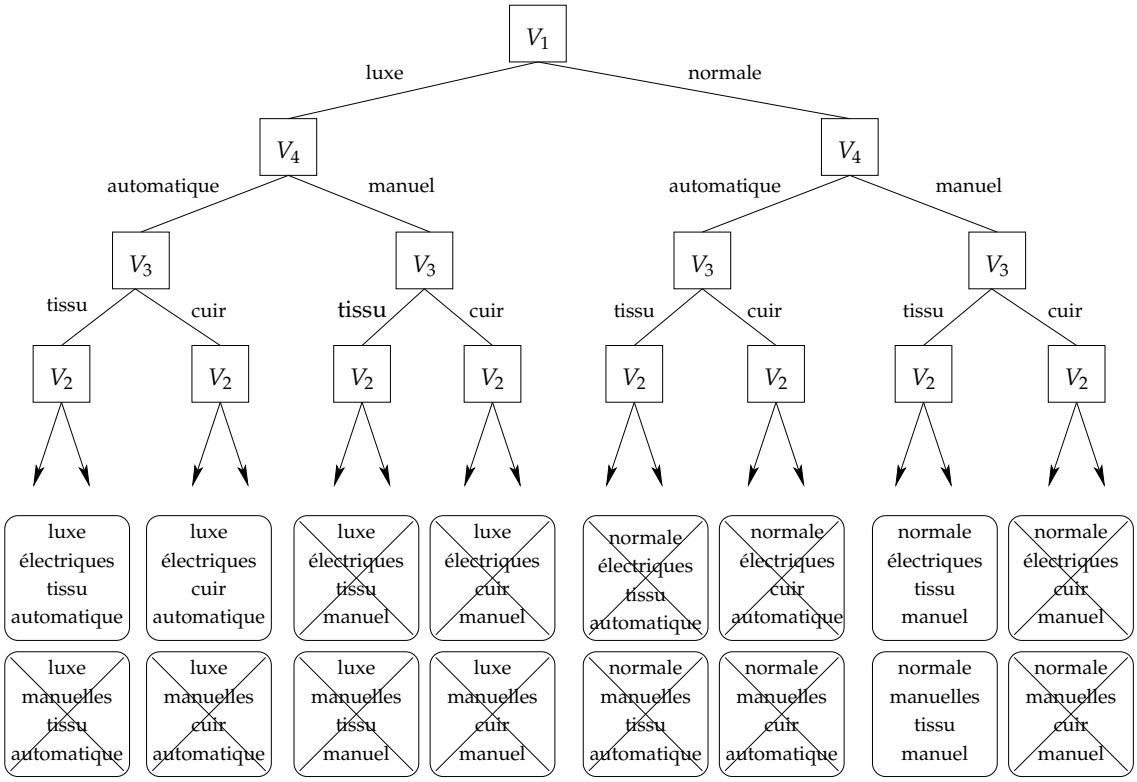


FIG. 2.4 — Illustration du processus de *Generate & Test* — ordre de valuation : V_1, V_4, V_3, V_2

2.2.2 *BackTracking*

La technique de retour-arrière consiste à valuer les variables les unes après les autres en vérifiant à chaque nouvelle valuation les contraintes dont toutes les variables ont été valuées. Lorsqu'une contrainte est violée, on remonte dans l'arbre de valuation des variables et on change la valeur de la dernière variable instanciée (GOLOMB & BAUMERT, 1965) (cf. algorithme A.1 page 135).

Il existe deux types d'heuristiques permettant d'améliorer cette technique : des heuristiques pour trouver une seule solution et des heuristiques permettant de trouver toutes les

solutions. Lorsque l'on cherche une solution, une fois que la valuation de toutes les variables est un succès, on stoppe. Si l'on cherche toutes les solutions, on retourne en arrière pour tester les autres valeurs des variables.

Le *BackTracking* est ainsi beaucoup plus efficace que le *Generate & Test* puisqu'il permet de couper certaines branches et ainsi d'éviter leur exploration. Pour en montrer l'interêt par rapport au *Generate & Test*, on se reportera à la figure 2.5 pour remarquer que seules certaines branches de l'arbre de valuation ont été explorées en totalité.

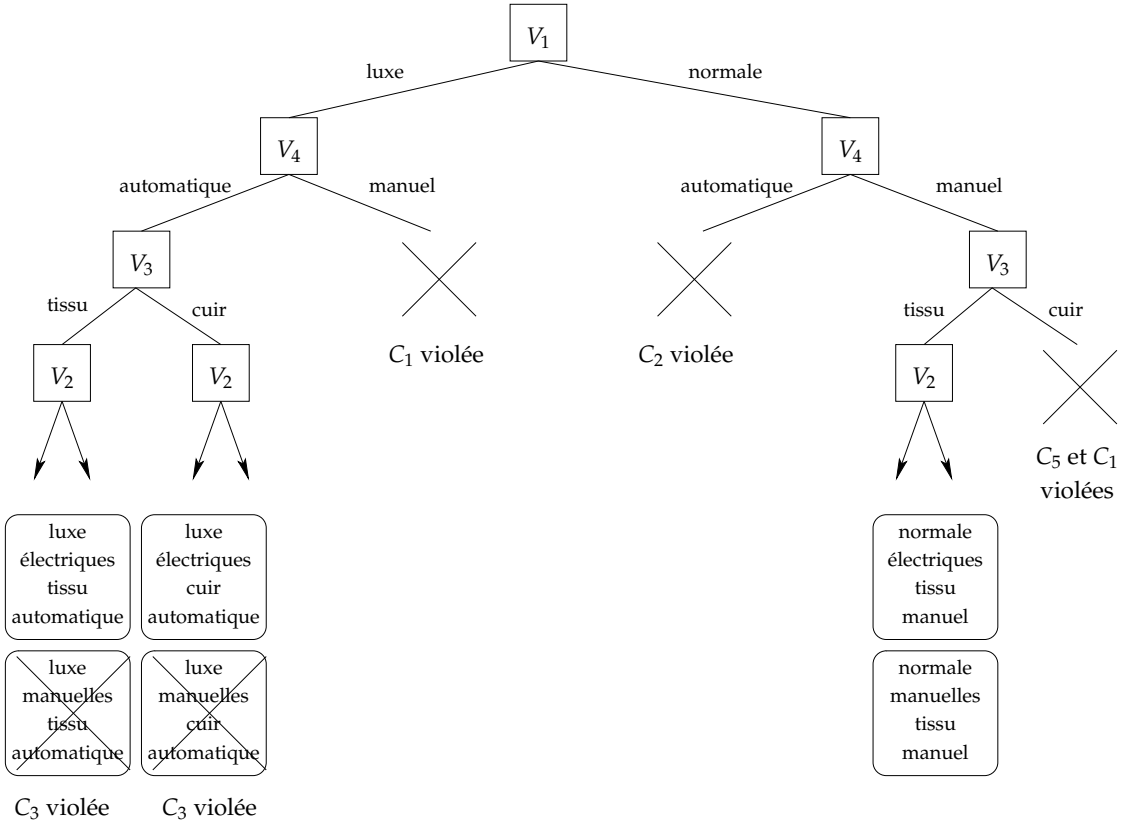


FIG. 2.5 — Illustration du processus de *BackTracking* — ordre de valuation : V_1, V_4, V_3, V_2

2.2.3 *BackJumping*

Il est possible de considérer cet algorithme commme un *BackTracking* intelligent (dirigé par les dépendances entre variables, cf. STALLMAN & SUSSMAN (1977)). En effet, lors d'un échec dans l'instanciation d'une variable, cet algorithme va essayer de retrouver la variable « fautive » (par exemple, celle qui est liée à la dernière instanciation par une contrainte). Pour cela, nous disposons de plusieurs heuristiques et donc de plusieurs algorithmes de *BackJumping* (l'algorithme A.2 page 136 montre un exemple de *BackJumping*) :

– de Gaschnig (GASCHNIG, 1979) ; cet algorithme permet d'effectuer des sauts en arrière lors de l'instanciation des variables « feuilles » [1]. Pour cela, il marque la dernière variable liée par contrainte à la variable feuille dont le domaine est vide, et peut ainsi remonter jusqu'à changer l'instanciation de cette variable fautive. Elle a été améliorée

[1] Variable feuille : dernière variable à être instanciée.

par la suite ; CLARK & HOLTE (1992) proposent une modification permettant de récupérer l'ensemble de variables (et non plus une seule) responsable de l'incohérence.
- basé sur les graphes (*Graph-based BackJumping*, cf. DECHTER (1990)) ; cet algorithme permet d'effectuer des sauts en arrière lors de l'instanciation. Lorsque l'on s'aperçoit que le domaine d'une variable X_i ne contient plus de valeurs cohérentes, on remonte à la dernière variable instanciée liée par contrainte avec X_i : X_j. Si cette variable ne peut pas prendre de valeur cohérente, on remonte alors à la dernière variable instanciée liée par contrainte avec X_i *ou* avec X_j, et ainsi de suite.
- dirigé par les conflits (*Conflict-directed BackJumping*, cf. PROSSER (1994)) ; cet algorithme utilise les principes des deux algorithmes précédents. Un ensemble de variables (*jump-back set*) contient les variables sur lesquelles on retournera en arrière s'il y a échec. Cet ensemble est maintenu pour chacune des variables instanciées. Cet algorithme de *BackJumping* est optimal.

Ces trois algorithmes de *BackJumping* ont été conçus pour des CSP binaires. En effet, le problème se pose du choix de la variable sur laquelle remonter en cas de contrainte d'arité[2] supérieure à 2. Cependant, BARTÁK (1998) recense plusieurs techniques de binarisation de CSP.

Bien que le *BackJumping* puisse être plus efficace que le *BackTracking*, nous exploiterons ce dernier pour la résolution de CSP, du fait de sa facilité d'implémentation, en gardant à l'esprit qu'il sera ensuite possible d'améliorer les performances avec *BackJumping*.

2.3 Filtrage sur des domaines discrets

Le filtrage est quasiment incontournable en conception interactive. En effet, il permet de réduire progressivement les domaines des variables lorsque des domaines ont été réduits (par propagation ou par l'utilisateur).

2.3.1 Classification générale

Il existe beaucoup d'algorithmes de filtrage. DEBRUYNE & BESSIÈRE (1997b) en ont établi une classification selon leur force de filtrage. Un algorithme plus filtrant demande plus de calculs, ce qui amène à se poser la question suivante : privilégie-t-on la puissance de filtrage ou la rapidité des calculs, quitte à garder des valeurs pouvant mener à des incohérences lors de la propagation ?

Un filtrage parfait a pour résultat les domaines parfaitement filtrés, desquels toutes les valeurs incohérentes ont été ôtées. Dans ce cas, nous parlons de filtrage global. Les algorithmes présentés ici permettent seulement du filtrage local : ils font l'objet d'un compromis entre rapidité d'exécution et complétude du filtrage.

Dans le cas général, tous les algorithmes suivants s'appliquent à des contraintes binaires. Seule la cohérence d'arc (AC) a fait l'objet d'extensions aux contraintes n-aires (BESSIÈRE *et al.*, 1999).

[2] L'arité est le nombre maximal de variables par contraintes d'un CSP.

Arc-Consistency (AC)

L'*Arc-Consistency* fait l'objet d'une section particulière (cf. section 2.3.2 page suivante). La base est une notion de support ; une valeur est cohérente tant qu'elle a au moins une valeur compatible dans chacun des domaines des variables « voisines » (reliées par contrainte). De façon très générale, les algorithmes AC enlèvent les valeurs qui n'ont plus de support dans les autres domaines.

k-*Consistency*

Si $k = 2$, la k-cohérence équivaut à l'*Arc-Consistency* ; lorsque $k = 3$, on parle de chemin-cohérence (PC : *Path-Consistency*). k est donc le nombre de variables prises en compte pour vérifier la cohérence des contraintes. Pour k supérieur à 2, une instantiation est k-cohérente si elle est $k-1$-cohérente et que l'on peut trouver une valuation d'une k^{e} variable telle que les contraintes soient respectées. À partir de $k = 3$, la complexité en temps et en espace de cet algorithme augmente grandement (BESSIÈRE *et al.*, 1995).

La figure 2.6 illustre le concept de k-cohérence sur un CSP qui est arc-cohérent. En effet, chacune des valeurs de chaque variable a des supports (des valeurs compatibles) dans chacun des domaines des variables qui lui sont reliées par contrainte.

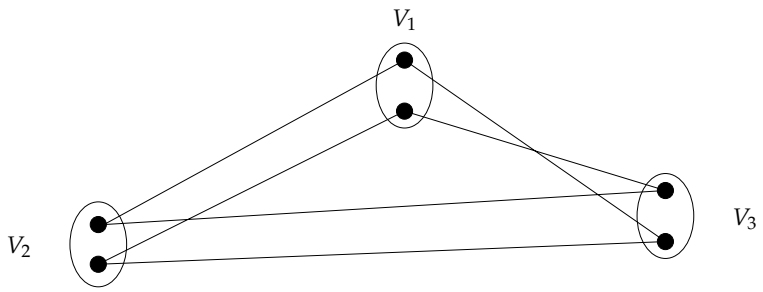


FIG. 2.6 — CSP arc-cohérent et non chemin-cohérent

En revanche, ce CSP n'est pas 3-cohérent. En effet, pour un couple de variables (par exemple, V_1 et V_2), il est impossible de trouver une valuation cohérente d'une troisième variable (ici, V_3). Ce CSP est donc arc-cohérent, mais n'est pas 3-cohérent (chemin-cohérent).

Cohérences plus fortes

L'algorithme *Restricted Path Consistency* (RPC) a été proposé dans le but d'enlever plus de valeurs incohérentes que l'*Arc-Consistency*, sans atteindre la complexité de *Path-Consistency*. Alors que les algorithmes PC les plus performants essayent d'étendre toutes les paires de valeurs, *Restricted Path Consistency* évite ce travail fastidieux et ne vérifie que certaines paires (BERLANDIER, 1995).

Max Restricted Path Consistency (Max-RPC) est la limite supérieure de la k-RPC, en présentant l'avantange d'être moins coûteuse que celle-ci si k devient élevé (DEBRUYNE & BESSIÈRE, 1997a).

L'algorithme *Neighborhood Inverse Consistency* (NIC) est basé sur l'étude du graphe complémentaire du CSP, où les variables n'étant pas reliées par des contraintes voient toutes leurs valeurs explicitement autorisées (FREUDER & ELFE, 1996).

Discussion sur les différents algorithmes

La figure 2.7, établie par DEBRUYNE & BESSIÈRE (1997b), montre les comparaisons sur les forces de filtrage que l'on peut effectuer entre ces différents algorithmes.

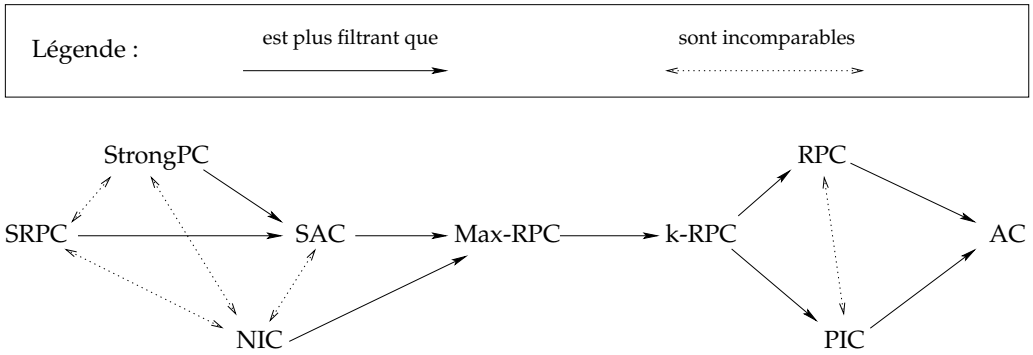


FIG. 2.7 — Comparaison de la puissance de filtrage de différents algorithmes (DEBRUYNE & BESSIÈRE, 1997b)

Nous n'allons pas détailler le fonctionnement de chacun de ces algorithmes. Il faut cependant retenir que plus un algorithme a une puissance de filtrage élevée, plus le filtrage est coûteux.

2.3.2 La cohérence d'arc

Pour le filtrage, nous avons choisi de nous concentrer sur l'*Arc-Consistency* car nous privilégions ce compromis entre qualité de filtrage et interactivité (les autres méthodes de filtrage perdent en temps ce qu'elles gagnent en qualité de filtrage).

Son nom provient du fait qu'elle vérifie les contraintes binaires dans le graphe correspondant au CSP, qui sont donc des arcs. Elle a été introduite par WALTZ (1975) et MACKWORTH (1977).

La contrainte reliant deux variables $(X_i \rightarrow X_j)$ est arc-cohérente si, pour chaque valeur x de la variable X_i, il existe une valeur y de D_j telle que la contrainte entre X_i et X_j permette cette instanciation (cf. équation 2.2).

$$(X_i \rightarrow X_j) \text{ arc-cohérent} \Leftrightarrow \forall x \in D_i, \exists y \in D_j \,/\, \{X_i = x, X_j = y\} \text{ cohérent} \qquad (2.2)$$

Pour rendre un arc $(X_i \rightarrow X_j)$ cohérent (par rapport à l'*Arc-Consistency*), il suffit de retirer du domaine de X_i les valeurs qui n'ont pas de support dans le domaine de X_j. Il faut noter que le fait d'enlever ces valeurs ne doit pas changer l'espace des solutions du CSP original. L'algorithme REVISE (cf. algorithme A.3 page 136) permet de retirer les valeurs du domaine de X_i qui n'ont pas (ou plus) de support dans le domaine de X_j et indique si le domaine de X_i a été réduit.

Pour que chaque arc du graphe de contraintes soit cohérent, il ne suffit pas d'appliquer l'algorithme REVISE une fois à chaque arc. En effet, lorsqu'un domaine X_i est réduit par propagation d'une contrainte $(X_i \rightarrow X_j)$, le domaine de chaque variable liée à X_i par une contrainte $(X_k \rightarrow X_i)$ doit être revérifié. En effet, certaines valeurs présentes dans D_k peuvent avoir perdu leur valeur support dans D_i lors de l'appel à REVISE sur l'arc $(X_i \rightarrow X_j)$. L'algorithme A.4 page 137 d'arc-cohérence permet de revérifier les différents arcs.

Cette version de l'*Arc-Consistency* n'est pas vraiment satisfaisante car elle refait un passage de vérification sur **tous** les arcs dès qu'un seul domaine a été réduit.

En fait, les seuls domaines directement affectés par une réduction du domaine de X_j sont les domaines D_i des variables X_i liées à X_j par un arc $(X_i \rightarrow X_j)$. Seuls ces domaines doivent être revérifiés. Si ces domaines sont réduits, il faut remonter un cran plus loin via les arcs $(X_k \rightarrow X_i)$ pour supprimer les valeurs dans D_k n'ayant plus de support dans D_i. AC-3 (cf. algorithme A.5 page 137, MACKWORTH & FREUDER (1985)), l'une des variantes les plus connues de l'*Arc-Consistency*, utilise ce mécanisme.

Il existe beaucoup d'autres algorithmes implémentant la cohérence d'arc :

- AC-4 (MOHR & HENDERSON, 1986; MOHR & MASINI, 1988), moindre complexité en temps, plus complexe en espace ;
- AC-5 (VAN HENTENRYCK *et al.*, 1992), amélioration de AC-4 ;
- AC-6 (BESSIÈRE, 1994), cf. algorithmes A.6 page 137, A.7 et A.8 page 138, algorithme arrivant à une complexité en temps équivalent à l'AC-3 et en espace équivalent à AC-5 ;
- AC-7 (BESSIÈRE *et al.*, 1995), amélioration de AC-6 ;
- AC-2000 et AC-2001 (BESSIÈRE & RÉGIN, 2001), algorithmes basés sur AC-3, relativement équivalents à AC-6 ;
- AC-* (RÉGIN, 2005), permettant de choisir le meilleur des algorithmes précédents selon l'aspect local du problème.

2.4 Filtrage sur des domaines continus

Étant donné les problèmes posés par la résolution de CSP continus ou mixtes, nous devons utiliser des méthodes permettant de réduire les domaines continus de façon à s'approcher des solutions du CSP. Ces méthodes sont l'équivalent des méthodes de filtrage sur domaines discrets. Nous allons étudier trois types de méthodes.

2.4.1 Arithmétique des intervalles

MOORE (1966) a posé les bases de l'arithmétique des intervalles. Il définit les opérateurs de base sur les intervalles suivants ($[a;b]$ et $[c;d]$ sont deux intervalles) :

- $[a;b] \oplus [c;d] = [a+c; b+d]$;
- $[a;b] \ominus [c;d] = [a-d; b-c]$;
- $[a;b] \otimes [c;d] = [min(ac,ad,bc,bd); max(ac,ad,bc,bd)]$;
- $[a;b] \oslash [c;d] = [min(\frac{a}{c}, \frac{a}{d}, \frac{b}{c}, \frac{b}{d}); max(\frac{a}{c}, \frac{a}{d}, \frac{b}{c}, \frac{b}{d})]$ si $0 \notin [c;d]$,

qui correspondent aux quatres opérations de base étendues aux intervalles. Cette arithmétique peut être étendue aux fonctions élémentaires (cos, log... cf. NEUMAIER (1990)), et d'autres opérateurs peuvent être définis (puissance...).

Les contraintes numériques sont alors étendues aux intervalles en remplaçant les opérateurs « classiques » par les opérateurs sur les intervalles, comme le montre l'exemple 2.8 page ci-contre.

Nous utilisons la technique étendue aux multi-intervalles ; SHAPIRO *et al.* (1998) ont montré que cette extension aux multi-intervalles ne provoquait pas d'explosion combinatoire (qui pourrait être due aux opérations sur les multi-intervalles). Les opérations de base sur les multi-intervalles deviennent les unions des opérations de base sur les produits cartésiens des intervalles, par exemple si $\boldsymbol{i}$ et $\boldsymbol{j}$ sont deux multi-intervalles, avec $\boldsymbol{i} = \cup\{\boldsymbol{i_1}; \boldsymbol{i_2}\}$ et

EX. 2.8 — Extension d'une contrainte aux intervalles

La contrainte :

$$z = 2 \times a + b/c - d$$

(où a, b, c, d et z sont des variables continues) devient (avec $D_a = D_b = D_c = D_d = [1\,;2]$ et $D_z = [0\,;10]$) :

$$\begin{aligned} z &= (2 \otimes a) \oplus (b \oslash c) \ominus d \\ &= (2 \otimes [1\,;2]) \oplus ([1\,;2] \oslash [1\,;2]) \ominus [1\,;2] \\ &= [2\,;4] \oplus [0{,}5\,;2] \ominus [1\,;2] \\ &= [2{,}5\,;6] \ominus [1\,;2] \\ &= [0{,}5\,;5] \end{aligned}$$

Le domaine de z sera donc réduit à $[0{,}5\,;5]$.

$\boldsymbol{j} = \cup\{\boldsymbol{j_1}\,;\boldsymbol{j_2}\})$:

- l'addition :

$$\boldsymbol{i} \oplus \boldsymbol{j} \equiv \bigcup\{i_1 \oplus j_1, i_1 \oplus j_2, i_2 \oplus j_1, i_2 \oplus j_2\}$$

- soustraction :

$$\boldsymbol{i} \ominus \boldsymbol{j} \equiv \bigcup\{i_1 \ominus j_1, i_1 \ominus j_2, i_2 \ominus j_1, i_2 \ominus j_2\}$$

- multiplication :

$$\boldsymbol{i} \otimes \boldsymbol{j} \equiv \bigcup\{i_1 \otimes j_1, i_1 \otimes j_2, i_2 \otimes j_1, i_2 \otimes j_2\}$$

- division :

$$\boldsymbol{i} \oslash \boldsymbol{j} \equiv \bigcup\{i_1 \oslash j_1, i_1 \oslash j_2, i_2 \oslash j_1, i_2 \oslash j_2\}$$

Ainsi, nous disposons d'une approximation supérieure des intervalles englobant les projections des variables, qui nous permet de réduire le domaine de chaque variable si celui-ci est plus grand. L'exemple 2.9 illustre le fonctionnement du filtrage direct par l'arithmétique des intervalles.

EX. 2.9 — Filtrage direct par arithmétique des intervalles

Soient $x \in [0\,;1]$, $y \in [2\,;3]$, $z \in [-10\,;10]$ et la formule suivante : $z = x^2 - 2 \times x\,y + y^2$. Nous pouvons alors calculer le domaine de z grâce à l'arithmétique des intervalles et filtrer son domaine :

$$\begin{aligned} z &= x^2 - 2 \times x\,y + y^2 \\ D'_z &= [0\,;1] \otimes [0\,;1] \ominus 2 \otimes [0\,;1] \otimes [2\,;3] \oplus [2\,;3] \otimes [2\,;3] \\ &= [0\,;1] \ominus 2 \otimes [0\,;3] \oplus [4\,;9] \\ &= [-6\,;1] \oplus [4\,;9] \\ &= [-2\,;10] \end{aligned}$$

Ainsi, le domaine filtré de z sera réduit à $[-2\,;10]$, par rapport à son domaine initial de $[-10\,;10]$.

2.4.2 2B-cohérence

LHOMME (1993) définit la 2B-cohérence comme un filtrage aux bornes des intervalles. Une contrainte est donc 2B-cohérente si pour chacune des variables qu'elle relie prenant la valeur des bornes de son intervalle de définition, il existe au moins une valeur dans le domaine des autres variables permettant de satisfaire la contrainte. La 2B-cohérence — ou *Hull* cohérence — est définie ainsi :

Définition 2.6 : *2B-cohérence* LHOMME & RUEHER *(1997)*

Soient $P = (\boldsymbol{V}, \boldsymbol{D}, \boldsymbol{C})$ un CSP *et $c \in \boldsymbol{C}$ une contrainte numérique d'arité k portant sur les variables continues $(v_1, \ldots, v_k)$. La contrainte c est 2B-cohérente si et seulement si pour chacune des bornes des intervalles de définition des variables de c, il existe au moins une valeur dans le domaine des autres variables telle que c soit cohérente ($\underline{v_i}$ et $\overline{v_i}$ représentent respectivement les bornes inférieure et supérieure de l'intervalle de définition de v_i et D_{v_i} le domaine de la variable v_i) :*

- $c(D_{v_1}, \ldots, D_{v_{i-1}}, \underline{v_i}, D_{v_{i+1}}, \ldots, D_{v_k})$
- $c(D_{v_1}, \ldots, D_{v_{i-1}}, \overline{v_i}, D_{v_{i+1}}, \ldots, D_{v_k})$.

Un CSP *est 2B-cohérent si et seulement si toutes ses contraintes sont 2B-cohérentes.*

La 2B-cohérence permet de ne pas perdre de valeurs pouvant mener à des solutions. Cependant, nous ne retiendrons pas cette méthode de filtrage numérique car ses hypothèses d'application sont trop contraignantes. Pour pouvoir appliquer la 2B-cohérence, les contraintes numériques doivent être des fonctions :

- sans multi-occurence de variables (par exemple, $x = x^2 - y$) ;
- monotones ;
- projetables sur chacune de leurs variables.

Généralement, en conception, ces trois conditions ne sont pas remplies. L'exemple 2.10 page ci-contre illustre la 2B-cohérence sur la contrainte $x + y = 2$.

2.4.3 *Box*-cohérence

BENHAMOU *et al.* (1994) proposent la *Box*-cohérence pour s'affranchir des conditions d'application de la 2B-cohérence. Une contrainte est *Box*-cohérente si pour toute variable X_i appartenant à cette contrainte, les bornes de l'intervalle de X_i vérifient la contrainte obtenue en remplaçant chaque occurence de X_j ($X_j \neq X_i$) par l'intervalle de valeurs de X_j.

La *Box*-cohérence fait ensuite appel à des méthodes numériques (comme celle de NEWTON). En pratique, la fonction représentant la contrainte doit être dérivable ; la *Box*-cohérence fait appel aux méthodes numériques pour trouver un zéro de la dérivée. Nous pouvons alors trouver un intervalle englobant cette solution autour des points d'inflexion de la fonction.

2.4.4 Discussion

DELOBEL (2000) a réalisé une classification de méthodes de filtrage pour CSP numériques. Chacune de ces méthodes est plus adaptée à certaines formes de contraintes numériques. Pour des fonctions continues monotones, la 2B-cohérence procure un filtrage suffisant. Pour des fonctions non-monotones (mais dérivables), il faudra faire appel à la *Box*-cohérence ; de même si les contraintes numériques à filtrer contiennent des multi-occurences de contraintes (par exemple, $x = x^2 - y$). Le filtrage par l'arithmétique des intervalles a plusieurs

EX. 2.10 — Contrainte 2B-cohérente

Considérons la contrainte $C_1 : x + y = 2$ avec, dans un premier cas, $D_x = [0\,;1]$ et $D_y = [1\,;2]$, puis, dans un second cas, $D_x = [0\,;1]$ et $D_y = \{[0\,;2]$.
Dans le premier cas, la contrainte C_1 est 2B-cohérente car :
- il existe une valeur ($y = 2$) dans le domaine de y qui satisfait C_1 lorsque x est valuée à sa borne inférieure ($x = 0$),
- il existe une valeur ($y = 1$) dans le domaine de y qui satisfait C_1 lorsque x est valuée à sa borne supérieure ($x = 1$),
- il existe une valeur ($x = 1$) dans le domaine de x qui satisfait C_1 lorsque y est valuée à sa borne inférieure ($y = 1$),
- il existe une valeur ($x = 0$) dans le domaine de x qui satisfait C_1 lorsque y est valuée à sa borne supérieure ($y = 2$).

Ainsi :

$$\begin{cases} \forall x \in \{\underline{x}\,,\overline{x}\}\,, \exists y \in D_y \text{ tel que } C_1 \text{ est satisfaite} \\ \forall y \in \{\underline{y}\,,\overline{y}\}\,, \exists x \in D_x \text{ tel que } C_1 \text{ est satisfaite} \end{cases}$$

Dans le second cas, la contrainte C_1 n'est pas 2B-cohérente car il n'existe pas de valeur dans le domaine de x lorsque y est valuée à sa borne inférieure ($y = 0$) telle que C_1 soit satisfaite.

avantages : il est facile à mettre en œuvre (il suffit de surdéfinir les opérateurs mathématiques) et permet de ne pas trop alourdir les calculs. Il garantit de ne pas perdre de valeurs pouvant mener à des solutions et il permet de filtrer plus ou moins efficacement tout type de fonction (pas d'hypothèses d'application contraignantes).

Pour filtrer des problèmes numériques, nous utiliserons donc l'arithmétique des intervalles étendue aux multi-intervalles. Ce choix nous permet de conserver toutes les solutions d'un problème de conception sans trop alourdir les calculs et avec des hypothèses d'application peu contraignantes.

GELLE & FALTINGS (2003) proposent des méthodes de filtrage pour les contraintes mixtes (entre variables discrètes et variables continues). Ces méthodes sont basées sur une discrétisation des domaines continus et ne peuvent garantir la cohérence globale du CSP mixte.

2.5 Diversité compilée

Certaines approches par contraintes sont basées sur la transformation d'un CSP par compilation en un automate à états finis pour générer toutes les successions d'instanciations possibles (VEMPATY, 1992).

Ce type de méthode a l'avantage d'être optimal (sans retours arrière) une fois l'automate compilé et ne pose pas de problèmes de qualité de filtrage (cf. section 2.3 page 27).

Cependant, cette approche a trois inconvénients par rapport à nos besoins :
- les CSP continus ou mixtes ne peuvent être pris en compte. En effet, un CSP continu ou mixte ne peut fondamentalement pas être modélisé sous forme d'un automate à états finis, étant donné que certaines de ses variables ont des domaines infinis ;
- le temps de compilation peut constituer un problème suivant la taille du modèle de connaissances ;

– il n'existe pas de mécanismes de gestion de la pertinence (de variables, de sous-ensembles, de contraintes) : un automate compilé est une structure statique.

Des mécanismes permettant d'ajouter des contraintes unaires (typiquement, des contraintes de valuation de variables posées par l'utilisateur) au problème ont été abordés par AMILHASTRE *et al.* (2002), mais ceux-ci n'ont été étudiés que dans un cadre de configuration interactive.

À cause de ces inconvénients, nous ne prendrons pas en compte les systèmes basés sur des automates à états finis dans la suite de ce mémoire.

2.6 Amélioration de la résolution par le filtrage

Basés sur des techniques de parcours d'arbre type *BackTracking*, des algorithmes utilisent le principe du filtrage avant chaque nouvelle valuation de variable pour éviter de tester des valeurs dont on est sûr qu'elles méneront à des incohérences. En général, les mécanismes de filtrage interactif sont basés sur la cohérence d'arc, ce qui permet d'avoir un système rapide.

2.6.1 *Forward-Checking*

Le *Forward-Checking* est plus complet que le *BackTracking* (cf. section 2.2.2 page 25) et le *BackJumping* (cf. section 2.2.3 page 26) ; il permet de prévenir les conflits qui risquent de se produire à l'instanciation suivante.

Contrairement à *BackTracking* qui vérifie la cohérence sur les variables déjà instanciées, *Forward-Checking* anticipe l'arc-cohérence sur des variables non encore instanciées (cf. algorithme A.9 page 138). Il repose sur une implémentation locale de la cohérence d'arc : ainsi, quand une valeur d'une variable rentre en conflit avec l'intanciation courante, cette valeur est ôtée du domaine de la variable à instancier en question. Si le domaine d'une variable devient vide, toutes les branches dérivées de l'instanciation courante sont incohérentes ; il est donc possible de supprimer des branches entières de l'arbre des solutions bien en amont de ce que proposait le *BackTracking*.

De plus, lorsqu'une nouvelle variable est instanciée, les valeurs restantes de son domaine sont obligatoirement compatibles avec les variables déjà instanciées ; il n'est donc pas nécessaire de vérifier la cohérence avec les variables précédentes.

2.6.2 *Look-Ahead*

Forward-Checking vérifie l'arc-cohérence des variables courantes avec les variables futures. Il est possible de pousser plus loin ce concept en vérifiant l'arc-cohérence (et donc en réduisant les domaines) non seulement avec la variable qui sera prochainement instanciée, mais aussi avec les suivantes. On pratique ainsi la maintenance de cohérence d'arc (MAC : *Maintaining Arc-Consistency*), appelée aussi (*Full*) *Look-Ahead*. Cet algorithme repose sur une application plus prospective de la cohérence d'arc (par rapport au *Forward-Checking*).

Cet algorithme, visible sur l'algorithme A.10 page 139 procure une meilleure qualité de filtrage par rapport au *Forward-Checking*, puisqu'il permet de détecter les incohérences futures encore plus tôt. De même qu'avec *Forward-Checking*, il n'est pas nécessaire de vérifier la cohérence avec les variables précédemment instanciées puisque les domaines sont réduits pour s'ajuster automatiquement aux premières variables.

Cependant, s'il existe beaucoup de variables ayant des grands domaines, *Look-Ahead* fera plus de vérifications à chaque étape que *Forward-Checking*. Il s'ensuit donc, malgré le nombre réduit d'étapes, une augmentation du nombre de vérifications pouvant réduire l'efficacité de *Look-Ahead*.

2.6.3 Discussion

La figure 2.11 situe les algorithmes *BackTracking*, *Forward-Checking* et *Look-Ahead* par rapport aux variables instanciées et à instancier. *BackTracking* vérifie seulement les contraintes dont toutes les variables sont instanciées. *Forward-Checking* vérifie en plus les contraintes dont l'une des variables n'est pas encore instanciée alors que *Look-Ahead* vérifie aussi les contraintes reliant ces dernières variables non-instanciées avec d'autres variables non-instanciées.

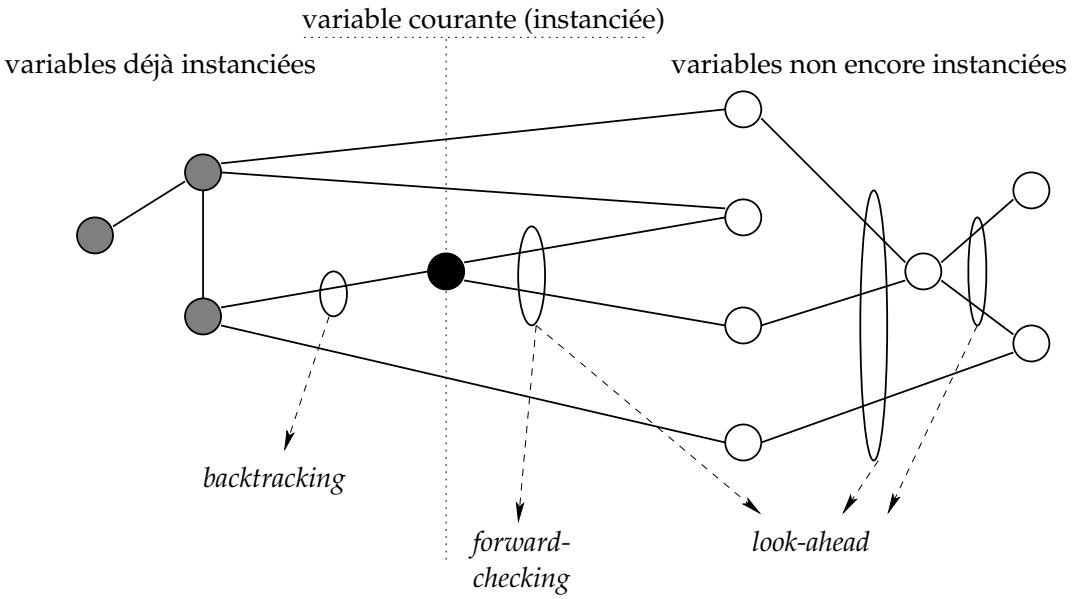


FIG. 2.11 — Position des variables instanciées sur plusieurs algorithmes (BARTÁK, 1998)

Plus on propage les contraintes à chaque nœud, moins il y a de nœuds à explorer, mais le coût (en temps de calcul) peut augmenter significativement.

Il est possible, sur n nœuds, d'exécuter de la n-cohérence forte, et d'avoir donc un seul nœud exploré, mais cette démarche devient de moins en moins efficace au fur et à mesure que n augmente, et ses performances passent rapidement en-dessous de *BackTracking*. Même *Look-Ahead* est parfois trop « cher », c'est pourquoi *BackTracking* et *Forward-Checking* sont très souvent utilisés dans des applications.

2.7 Conclusion

Dans ce chapitre, nous avons commencé par introduire les CSP, qui sont un concept fréquemment utilisé pour créer des systèmes d'aide à la conception (comme cela a été montré par LOTTAZ *et al.*, 1999; MULYANTO, 2002; VERNAT, 2004; VAREILLES, 2005). Nous avons

ensuite présenté les différentes techniques permettant d'assister la conception de manière autonome (résolution de CSP) ou interactive (filtrage de CSP). Ces différentes techniques doivent être adaptées au type de CSP : continu, discret ou mixte. Cependant, les approches permettant de traiter des contraintes mixtes sont rares, nous essaierons de remédier à ce point dans la suite de ce mémoire, en proposant une implémentation originale au chapitre 5. Plusieurs méthodes permettent de tirer parti des avantages du filtrage pour aboutir plus rapidement à une solution dans le cas d'un processus de conception autonome.

Nous venons de voir que les CSP peuvent couvrir un besoin que nous avons évoqué au chapitre 1 : la modélisation de paramètres discrets et de paramètres continus. En revanche, les modèles mixtes (regroupant les deux types d'éléments) sont plus délicats à traiter.

Nous définissons ici l'appellation CSP « classique », qui regroupe les concepts que nous avons vus au cours de ce chapitre.

Notation 2.7 : CSP *« classique »*

Un CSP dit « classique » est un CSP (symbolique, numérique ou mixte) ne disposant pas de système de gestion de pertinence de ses éléments, ni de système de gestion hiérarchique.

Dans le chapitre suivant, nous étudions les évolutions de CSP permettant de traiter des problèmes définis hiérarchiquement ou de gérer la pertinence des éléments (voire les deux).

Chapitre 3

Éléments optionnels et hiérarchie dans les approches par contraintes

Dans le chapitre 1, nous avons présenté les besoins pour l'aide à la conception à base de contraintes. Un de ces besoins est la gestion d'éléments optionnels (paramètres, composants standard ou paramétrables, sous-ensembles) ; ceux-ci sont modélisés par des variables, contraintes ou groupes d'un CSP, dont il faut gérer la pertinence.

Dans ce chapitre, nous allons passer en revue certains types de CSP de la littérature afin d'étudier comment ceux-ci répondent au besoin de gestion de la pertinence de certains éléments du modèle.

La pertinence est une notion permettant de qualifier la participation d'un composant, d'un paramètre, d'un sous-ensemble ou d'une contrainte à la solution d'un problème de conception.

Du point de vue du modeleur, nous pouvons établir deux grandes catégories de mécanismes permettant la gestion de la pertinence :

① l'élément dont la pertinence est indéterminée existe toujours dans le problème, le moteur gère alors explicitement la pertinence de l'élément en question ;

② l'élément dont la pertinence est indéterminée n'existe pas vis-à-vis du moteur ; un mécanisme permet alors d'ajouter cet élément au cours de la résolution ou du filtrage du problème selon une condition (valuation de variable, vérification d'une condition...).

Les premiers CSP proposant une gestion de la pertinence (Mittal & Falkenhainer, 1990) se nomment CSP dynamiques ; ils ont ensuite été renommés CSP conditionnels ou CondCSP par Gelle & Faltings (2003). Verfaillie & Jussien (2005) ont repris cette appellation pour proposer une classification de ces problèmes, identifiant les CSP conditionnels au point ① et les CSP dynamiques au point ②.

Généralement, ces travaux attachent à la notion de dynamisme celle d'activité. Ainsi, un élément dynamique est un élément dont l'activité va être déterminée au cours de la résolution ou du filtrage du CSP. Nous emploierons les termes d'activité, prise en compte ou pertinence indifféremment dans la suite de ce mémoire.

3.1 Les DCSP de MITTAL & FALKENHAINER (1990)

MITTAL & FALKENHAINER (1990) ont été les premiers à proposer une approche pour gérer la pertinence de variables dans un modèle de contraintes. Ils ont donc défini les *Dynamic* CSP (DCSP), autorisant ou interdisant l'activation d'un ensemble de variables. Les DCSP appartiennent au type ① des CSP conditionnels d'après VERFAILLIE & JUSSIEN (2005).

Définition 3.1 : DCSP

Un DCSP *est un quadruplet* $(\boldsymbol{X}, \boldsymbol{X_I}, \boldsymbol{D}, \boldsymbol{C})$ *où l'on identifie* $\boldsymbol{X_I}$ *comme un sous-ensemble de* $\boldsymbol{X}$ *regroupant les variables initialement actives et le triplet* $(\boldsymbol{X}, \boldsymbol{D}, \boldsymbol{C})$ *comme un* CSP *discret. Les contraintes sont divisées en deux groupes : les contraintes d'activité (*$\boldsymbol{C_A}$*) et les contraintes de compatibilité (*$\boldsymbol{C_C}$*). Les contraintes de compatibilité sont équivalentes aux contraintes* $\boldsymbol{C}$ *d'un* CSP *discret. Les contraintes d'activité, notées ainsi :* $C_A : P \xrightarrow{ACT} X_i$ *ou* $P \xrightarrow{\cancel{ACT}} X_i$ *(où* X_i *représente une variable) servent à gérer la pertinence de variables dont l'existence est conditionnée.*

Définition 3.2 : *Solution d'un* DCSP

Dans le cas d'un DCSP*, une solution est une instanciation de toutes les variables actives telle que toutes les contraintes de compatibilité et toutes les contraintes d'activité sont satisfaites.*

Les contraintes d'activité

MITTAL & FALKENHAINER (1990) ont introduit quatre types de contraintes permettant de gérer la pertinence des variables appelées contraintes d'activité.

La contrainte *Require Variable* active une variable à partir d'un prédicat dépendant d'un ensemble d'autres variables. On note ce type de contrainte de la façon suivante :

$$\begin{gathered} C_A(\text{RV}) : P(X_i, \ldots, X_j) \underset{RV}{\Longrightarrow} X_k \quad (\text{avec } X_k \notin \{X_i, \ldots, X_j\}) \\ \Longleftrightarrow \\ C_A(\text{RV}) : P(X_i, \ldots, X_j) \xrightarrow{ACT} X_k. \end{gathered} \tag{3.1}$$

La contrainte *Always Require Variable* active une variable en se basant sur un prédicat mettant en jeu l'activité d'autres variables. Ce type de contrainte prend la forme :

$$\begin{gathered} C_A(\text{ARV}) : X_i \wedge \cdots \wedge X_j \underset{ARV}{\Longrightarrow} X_k \quad (\text{avec } X_k \notin \{X_i, \ldots, X_j\}) \\ \Longleftrightarrow \\ C_A(\text{ARV}) : \{X_i = x \vee \cdots \vee X_i = x_{m_i}\} \wedge \cdots \wedge \{X_j = x \vee \cdots \vee X_j = x_{m_j}\} \xrightarrow{ACT} X_k \\ \text{Avec } m_i \text{ taille du domaine } D_i. \end{gathered} \tag{3.2}$$

La contrainte *Require Not* sert à empêcher l'activation d'une variable ; nous pouvons donc établir la forme de l'équation 3.3.

$$\begin{gathered} C_A(\text{RN}) : P(X_i, \ldots, X_j) \underset{RN}{\Longrightarrow} X_k \quad (\text{avec } X_k \notin \{X_i, \ldots, X_j\}) \\ \Longleftrightarrow \\ C_A(\text{RN}) : P(X_i, \ldots, X_j) \xrightarrow{\cancel{ACT}} X_k. \end{gathered} \tag{3.3}$$

La contrainte *Always Require Not* fonctionne de la même façon que la contrainte *Always Require Variable* par rapport à *Require Not*. Ainsi, ce n'est plus un prédicat d'un ensemble de variables, mais leur activité qui est prise en compte, comme le montre l'équation 3.4.

$$\begin{aligned} C_A(\text{ARN}) : X_i \wedge \cdots \wedge X_j \underset{ARN}{\Longrightarrow} X_k \quad (\text{avec } X_k \notin \{X_i, \ldots, X_j\}) \\ \Longleftrightarrow \\ C_A(\text{ARN}) : \{X_i = x \vee \cdots \vee X_i = x_{m_i}\} \wedge \cdots \wedge \{X_j = x \vee \cdots \vee X_j = x_{m_j}\} \xrightarrow{\cancel{ACT}} X_k. \end{aligned} \tag{3.4}$$

Ce jeu de contraintes autorise ou interdit l'activation de variables grâce à une formule.

L'algorithme d'implémentation des DCSP

L'algorithme A.11 page 140 exploite les différents types de contraintes proposés par MITTAL & FALKENHAINER (1990) (contraintes de compatibilité et contraintes d'activation) et reprend l'idée du *BackTrack*. En revanche, aucun algorithme de filtrage n'est proposé pour ce type de CSP.

Extensions des DCSP

SOININEN & GELLE (1999) et SOININEN *et al.* (1999) ont proposé une nouvelle définition des DCSP permettant de mélanger les contraintes de type RV, ARV, RN et ARN pour établir des prédicats autorisant ou interdisant l'activation d'une variable, en se basant sur une combinaison de l'activité, de l'inactivité et/ou de la valeur d'autres variables.

Les CondCSP (*Conditionnal* CSP, cf. GELLE & FALTINGS (2003)) sont basés sur le même principe de contraintes d'activité et de compatibilité, mais utilisent des algorithmes générant un CSP « classique » à chaque nouvelle activation ou désactivation de variables[1].

DCSP et aide à la conception

Grâce à la gestion de l'activité des variables, les DCSP permettent de répondre au besoin de composants ou paramètres discrets optionnels, modélisés par une variable. Les variables ne faisant pas partie de l'ensemble de variables initialement actives sont soit activées au fil des valuations successives (*via* les contraintes d'activité), soit elles ne font pas partie de la solution. Les composants optionnels (ou paramètres optionnels) sont modélisés par des variables du DCSP inactives au début du processus de conception. Lors du processus de conception, les variables représentant des composants ou paramètres optionnels deviendront actives ou inactives et rendront compte de la participation du composant ou paramètre qu'elles représentent à la solution du problème de conception. Ainsi, les composants optionnels, modélisés par des variables à pertinence non déterminée, deviennent partie prenante de la nomenclature du produit fini ou non lors de la résolution du problème.

L'exemple publié dans MITTAL & FALKENHAINER (1990) montre comment utiliser les DCSP pour un problème de configuration de voiture ; les composants toit ouvrant et air conditionné, par exemple, ne font pas toujours partie de la solution.

[1] Cette méthode a fait l'objet d'une évaluation (GELLE & SABIN, 2003).

3.2 Les CCSP de SABIN & FREUDER (1996)

Les CCSP (*Composite* CSP) ont été introduits par SABIN & FREUDER (1996). Leur principale spécificité est l'ajout de sous-problèmes grâce à des méta-variables, ce qui permet d'établir une hiérarchie de variables, contraintes et sous-groupes, et de gérer la pertinence de ces éléments. Les CCSP correspondent au point ② page 37 : les variables et contraintes qui appartiennent aux groupes n'existent pas dans le problème de départ.

Définition 3.3 : CCSP

Un CCSP *est défini par un triplet* $(\boldsymbol{X}, \boldsymbol{D}, \boldsymbol{C})$*, de la même façon qu'un* CSP *« classique ». Cependant, la différence entre* CCSP *et* CSP *« classiques » est que les valeurs des variables peuvent être des sous-problèmes entiers (et pas seulement des valeurs « standard », issues de domaines) :* $P' = (\boldsymbol{X'}, \boldsymbol{D'}, \boldsymbol{C'})$*. Lorsqu'une variable* X_i *prend pour valeur* P'*, le problème à résoudre est modifié et devient :* $P'' = (\boldsymbol{X''}, \boldsymbol{D''}, \boldsymbol{C''})$*, avec :*

- $\boldsymbol{X''} = \boldsymbol{X} \cup \boldsymbol{X'} \setminus X_i$;
- $\boldsymbol{D''} = \boldsymbol{D'} \cup \boldsymbol{D} \setminus D_i$;
- $\boldsymbol{C''} = \boldsymbol{C} \cup \boldsymbol{C'} \setminus \boldsymbol{C(X_i)}$*, où* $\boldsymbol{C(X_i)}$ *est l'ensemble des contraintes portant sur* X_i.

Les variables dont les valeurs sont des sous-problèmes entiers sont appelées des méta-variables. Lorsqu'une méta-variable est valuée, elle est remplacée dans le problème par le sous-problème correspondant à cette valeur. La structure du modèle est donc augmentée dynamiquement de l'ensemble des variables et contraintes présentes dans le sous-problème.

CCSP et aide à la conception

Cette approche permet une manipulation aisée de groupes de variables et/ou contraintes symbolisant un sous-ensemble du produit, en créant des variables au cours de la résolution du problème. En particulier, l'exemple 1.6 page 17 peut être traité par une méta-variable : dès que celle-ci est valuée à essence ou diesel, le sous-problème comportant la variable cylindrée (éventuellement augmentée d'autres variables et contraintes) remplace cette variable.

Cependant, cette démarche n'est pas adaptée pour traiter des variables optionnelles isolées ; en effet, cela oblige à ajouter une variable pour la remplacer par une autre. De plus, il apparaît que les variables ajoutées par substitution d'une méta-variable ne peuvent être reliées par contrainte avec le reste du problème. Il n'est donc pas possible d'utiliser ce formalisme pour une contrainte reliant des variables initiales à des variables ajoutées au cours de la résolution.

Nous retiendrons donc que cette approche est très intéressante pour les sous-ensembles indépendants du reste du problème, mais souffre de défauts concernant les autres besoins en configuration et aide à la conception : pertinence de variables seules et contraintes conditionnelles.

3.3 L'ajout d'une valeur au domaine des variables

Les CSP★ (AMILHASTRE, 1999; MCDONALD & PROSSER, 2002) proposent de gérer la pertinence d'une variable *via* l'ajout d'une valeur « spéciale » au domaine. Cette méthode n'ajoute pas d'éléments au cours du traitement du problème, elle fait donc partie des CSP

conditionnels (point ① page 37). Nous utiliserons la valeur ★ dans la suite de ce mémoire, comme le propose AMILHASTRE (1999)[2].

Ainsi, lorsqu'une variable prend la valeur ★, cela signifie que :
- pour le CSP, cette variable ne participe pas à la solution ;
- du point de vue de la conception, l'élément (composant ou paramètre) que représente cette variable ne fait pas partie du produit.

CSP★ et aide à la conception

Cette modélisation permet de répondre au besoin de gestion de la pertinence de paramètres discrets ou de choix de composants standard optionnels. Pour ce qui concerne les variables symboliques, les CSP★ constituent un très bon moyen de gérer leur pertinence : il suffit d'ajouter le symbole ★ à leur domaine. En revanche, pour des variables numériques, il est plus délicat de gérer des domaines mixtes (domaines numériques continus auxquels on ajoute une valeur symbolique). L'adaptation des techniques de résolution et filtrage aux domaines mixtes représentant un travail fastidieux, nous nous bornerons à utiliser cette méthode de modélisation pour des variables symboliques.

Enfin, un travail de traduction est nécessaire pour obtenir les solutions d'un produit modélisé par un CSP★. Les composants ou paramètres représentés par des variables valuées à ★ n'existent pas dans la solution du problème de conception. Il faut donc prendre ceci en compte lors du passage des solutions du CSP★ aux solutions du problème de conception en ne conservant que les variables dont la valeur est différente de ★ comme solutions du problème de conception.

3.4 Les CSPe de VERON (2001)

VERON & ALDANONDO (2000) ont introduit les CSPe (CSP à états) pour permettre une prise en compte explicite de la pertinence des variables. Ainsi, une variable n'est pas seulement active ou indéfinie, on lui attache un état, ce qui peut permettre de gérer non seulement l'activité, mais éventuellement d'autres paramètres. Les CSPe correspondent au point ① page 37.

Définition 3.4 : *CSPe*

Un CSPe est donc un triplet $(\boldsymbol{V},\boldsymbol{A},\boldsymbol{F})$ tel que :
- $\boldsymbol{V}$ *est un ensemble de variables comprenant chacune un attribut d'état ;*
- $\boldsymbol{A}$ *est l'ensemble des domaines de définition des variables ;*
- $\boldsymbol{F}$ *est un ensemble de contraintes sur les variables (cf. définition 3.5).*

Définition 3.5 : *Contrainte dans un* CSPe

Une contrainte est une condition logique sur les valeurs ou états d'un ensemble de variables, notée $(V_i, V_j, ...)$.

Une contrainte est donc une formule logique $F(V_i, \ldots, V_j)$, avec, pour chaque variable impliquée dans la formule, une condition sur la valeur de cette variable ou sur son attribut d'état.

[2] MCDONALD & PROSSER (2002) proposent la valeur `DC` (pour *Don't Care*).

Définition 3.6 : *Solution d'un CSPe*

Une solution d'un CSPe *est une instanciation de tous les attributs d'état et de toutes les variables dont l'attribut d'état a pour valeur « actif » telle que toutes les contraintes à prendre en compte sont satisfaites.*

Le concept de CSPe se base sur les DCSP de MITTAL & FALKENHAINER (1990), en réifiant la pertinence des variables. Cette pertinence est attachée à un attribut d'état booléen (*actif* ou *inactif*). L'extension des DCSP apportée par SOININEN *et al.* (1999) peut aussi être utilisée : il est possible de contraindre dans une même formule logique des conditions sur l'état des variables et leur valeur.

CSPe et aide à la conception

Les CSPe permettent de répondre au besoin de modélisation de paramètres ou composants optionnels. Suivant la nature de ces paramètres ou composants, les variables les représentant peuvent être booléennes, numériques ou symboliques. Pour les CSPe, la notion de pertinence des variables est réifiée et supportée grâce à un attribut d'état attaché à chaque variable. Cet attribut d'état permet de rendre compte de la participation du composant ou paramètre modélisé par la variable à la solution du problème de conception.

Ainsi, la valeur de l'attribut d'état d'une variable à la fin du processus de conception permet de savoir si le composant ou paramètre qu'elle représente fait partie de la solution du problème de conception.

Enfin, la pertinence de sous-ensembles peut être gérée grâce à une combinaison des attributs d'état des différentes variables inclues dans le sous-ensemble. Ce développement de tous les attributs d'état n'est cependant pas naturel lorsque l'on utilise un modèle de produit ; il serait donc souhaitable de pouvoir factoriser cet attribut d'état.

3.5 Les ACSP de GELLER & VEKSLER (2005)

Les ACSP (*Activity* CSP) sont basés sur les DCSP ; ils font donc partie des CSP conditionnels (point ① page 37). Les ACSP utilisent des notions développées auparavant pour gérer la pertinence d'éléments du CSP avec des variables d'activité.

Définition 3.7 : ACSP

Un ACSP *est un sextuplet* $(\boldsymbol{X}, \boldsymbol{X_I}, \boldsymbol{X_A}, \boldsymbol{D}, \boldsymbol{C}, \boldsymbol{A})$ *où :*

- *les symboles* $\boldsymbol{X}$, $\boldsymbol{X_I}$ *et* $\boldsymbol{D}$ *sont équivalents à ceux d'un* DCSP *(variables, variables initialement actives et domaines) ;*
- $\boldsymbol{X_A}$ *est un ensemble de variables d'activité (dont le domaine est booléen) —* $\boldsymbol{X_A}$ *est inclus dans* $\boldsymbol{X_I}$ *;*
- $\boldsymbol{C}$ *est un ensemble de contraintes ;*
- $\boldsymbol{A}$ *est l'ensemble des conditions d'activation pour chaque variable non active initialement.*

Définition 3.8 : *Solution d'un* ACSP

Une solution S d'un ACSP *est une valuation d'un ensemble de variables* $\boldsymbol{X_S}$ $(\boldsymbol{X_S} \subseteq \boldsymbol{X})$ *tel que :*

– *toutes les variables actives sont valuées :*

$$\boldsymbol{X_I} \subseteq \boldsymbol{X_S} \textit{ et } \forall x \in \boldsymbol{X} \setminus \boldsymbol{X_I} \ : \ x \in \boldsymbol{X_S} \Leftrightarrow S(A(x)) = \textit{ Vrai}$$

– *toutes les contraintes pertinentes sont satisfaites.*

Les ACSP explicitent aussi la notion de pertinence par le biais d'un jeu de variables $\boldsymbol{X_A}$. Par rapport aux CSPe, les ACSP ne cherchent pas à attribuer un attribut d'état à chaque variable, mais chaque élément (contrainte ou groupe) non initialement actif est associé à une variable d'activité[3]. Ils permettent ainsi de modéliser des paramètres ou composants optionnels, mais aussi des sous-ensembles définis hiérarchiquement, grâce à la factorisation de la variable d'activité.

La différenciation entre contraintes de compatibilité et contraintes d'activation est reprise de la même façon que dans les DCSP ; GELLER & VEKSLER (2005) montrent l'équivalence entre les deux modèles.

L'algorithme AMAC[4] propose un « pré-filtrage » dans les groupes non encore pertinents pour améliorer la qualité de résolution.

ACSP et aide à la conception

Les ACSP permettent de répondre au besoin d'éléments optionnels : ces éléments sont des paramètres ou composants discrets, ou des sous-ensembles. Pour cela, les ACSP font appel à la notion de variable d'activité explicite (comme les CSPe). L'activité de contraintes ou de sous-ensembles peut être gérée grâce à des variables d'activité, lesquelles peuvent être factorisées (une seule variable d'activité pour un sous-ensemble).

Ainsi, lors de la phase de modélisation, une variable ou un groupe de variables représente un paramètre, composant ou sous-ensemble du produit. La variable d'activité de cette variable ou de ce groupe rend compte de la participation de celle-ci (ou celui-ci) à la solution du problème.

Les auteurs étendent les ACSP pour que les contraintes d'activité puissent influer sur les variables d'activité, ce qui permet d'interdire l'activation d'éléments.

3.6 Conclusion

Nous avons abordé dans ce chapitre les formalismes basés sur des CSP, permettant de gérer la pertinence d'éléments ou la hiérarchie du modèle. Nous avons ainsi analysé les besoins auxquels ils répondent, en termes de gestion d'éléments optionnels et de gestion hiérarchique.

Les DCSP, premier formalisme proposé pour la gestion de variables optionnelles, constituent la base de la plupart des autres formalismes (CSPe, CondCSP, ACSP). Ils permettent la manipulation de la pertinence de variables optionnelles grâce à un attribut, ou variable d'état, exprimé implicitement ou explicitement. Les CCSP proposent des possibilités de gestion hiérarchique de sous-problèmes grâce à une méta-variable pour chaque sous-ensemble de variables et/ou contraintes.

Nous pouvons synthétiser les démarches utilisées par chacun de ces formalismes dans le tableau 3.1 page suivante.

[3] La pertinence des variables est gérée par des mécanismes similaires aux CSP★

[4] *Activity Maintaining Arc-Consistency*, cf. GELLER & VEKSLER (2005).

TAB. 3.1 — Résumé des possibilités des différents formalismes CSP étudiés

Formalisme	Gestion de la pertinence de l'élément	Pertinence de groupes de variables et contraintes
DCSP	implicite, par contrainte d'activité ①	non abordé
CCSP	par ajout au problème ②	oui, pertinence factorisée
CSP★	explicite, par valeur particulière ①	oui, sans factorisation de pertinence
CSPe	explicite, par attribut d'état ①	oui, factorisation non prévue
ACSP	explicite, par variable d'activité ①	oui, avec pertinence factorisée

Transition

DANS le chapitre 1, nous avons introduit les notions de conception routinière et de configuration, ainsi que les différentes approches pour aider ces processus ; nous avons choisi de nous concentrer sur les approches à base de contraintes. Après avoir mis en évidence le modèle générique du produit, correspondant au modèle de connaissances, nous avons identifié les besoins suivants pour l'aide à la conception :

- traiter des éléments de différentes natures (symboliques, numériques...) ;
- traiter des modèles de produits organisés de façon hiérarchique ;
- permettre une gestion de la pertinence des différents éléments du produit.

Le deuxième chapitre s'attache aux différentes techniques de traitement de CSP « classiques ». Nous avons présenté différentes méthodes qui permettent d'assister des tâches de conception autonome ou interactive. Nous avons ainsi montré que les CSP permettaient de gérer le premier besoin identifié : modéliser et traiter des éléments de différentes natures (discrets, numériques continus...). Cependant, peu de travaux concernent les CSP mixtes.

Le troisième chapitre propose un état de l'art de différentes adaptations de CSP pour gérer la notion de hiérarchie ou de pertinence des éléments. Nous avons évoqué la différence entre pertinence en conception et pertinence du point de vue du CSP : en conception, un élément (composant, paramètre, sous-ensemble) est pertinent lorsqu'il prend part à la définition du produit ; il fait alors partie de la nomenclature. Pour un CSP, un élément (variable, contrainte, groupe) est pertinent lorsqu'il participe à la solution du CSP.

Au vu de cet état de l'art et de nos besoins, nous allons, dans la partie suivante, proposer une intégration de ces approches pour remédier aux problèmes suivants :

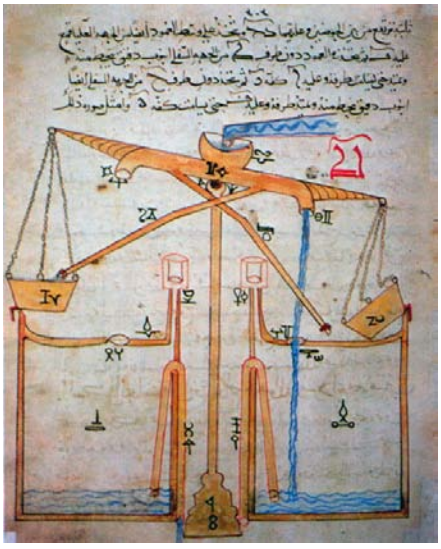
- il existe différentes techniques permettant de modéliser la pertinence d'éléments en CSP ; afin de tirer parti des avantages de chacune, nous proposons, dans le chapitre 4, de les étudier plus en détails et de les intégrer dans un formalisme commun nommé RCSP.

– bien que les CSP discrets et continus puissent être manipulés par des concepts existants, peu de travaux concernent les CSP mixtes ; l'implémentation que nous proposons dans le chapitre 5 permet le traitement de CSP mixtes.

Nous proposons donc une intégration de tous ces concepts dans les RCSP. Dans un premier temps, nous présentons les RCSP, ainsi que des phases de traduction de RCSP en CSP « classique » afin d'utiliser les méthodes de résolution et de filtrage étudiées pour les résoudre. Dans un deuxième temps, nous présentons notre implémentation basée sur des arbres syntaxiques pour traiter des CSP « classiques ». Cette implémentation nous permet de répondre aux besoins énoncés au chapitre 1 : le traitement de paramètres de différentes natures et de contraintes mixtes, saisies sous des formes diverses. Nous terminons cette deuxième partie par une présentation des compléments apportés à l'implémentation pour exploiter les spécificités des RCSP.

Deuxième partie

Propositions de modélisation et implémentation pour la résolution de problèmes d'aide à la conception ou à la configuration



Chapitre 4

Intégration et principes de modélisation : les RCSP

Nous avons abordé dans le chapitre précédent les mécanismes issus de la littérature présentant différentes solutions pour gérer la pertinence de variables, contraintes ou groupes. Dans ce chapitre, nous allons explorer plus finement les différents mécanismes abordés auparavant pour proposer une approche exploitant leurs atouts respectifs, tout en traitant les autres besoins identifiés au chapitre 1.

Ce chapitre propose une synthèse et une tentative d'intégration des mécanismes étudiés, ainsi que des principes de modélisation pour réaliser un modèle générique de produit. Pour synthétiser toutes ces approches, nous utiliserons le terme de RCSP (*Relevancy Constraint Satisfaction Problem*). Cette extension des CSP gérant la pertinence sera introduite progressivement au cours du chapitre, ce qui nous permet d'aboutir à une définition formelle des RCSP.

Afin d'évaluer nos propositions, nous explicitons les principes de traduction d'un RCSP en un CSP, ce qui nous permettra d'utiliser les algorithmes existants pour les CSP « classiques ».

La conclusion de ce chapitre portera sur des préconisations de modélisation pour chaque type de composant ou paramètre que le modeleur peut avoir à traiter pour créer un modèle générique et effectuer de l'aide à la conception ou à la configuration.

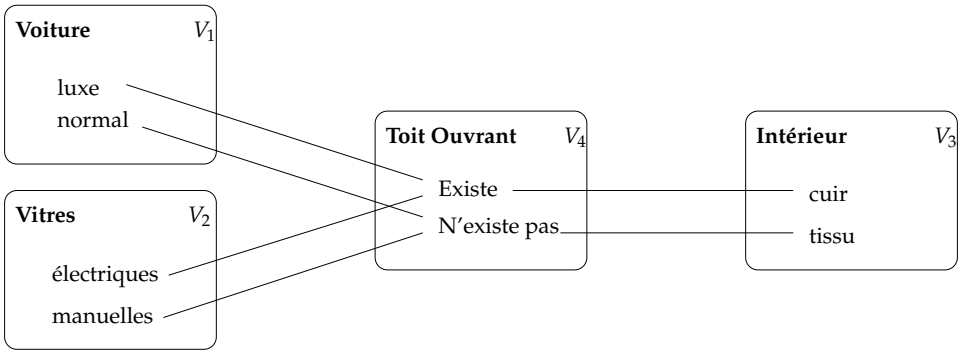
4.1 Modéliser un composant standard optionnel

Pour modéliser un composant standard optionnel (dont l'existence doit être déterminée au cours de la conception), nous préconisons l'utilisation d'une variable booléenne.

Ce mécanisme est illustré sur l'exemple 4.1 page suivante, qui est une adaptation simplifiée de l'exemple 1.7 page 18, dans lequel le toit ouvrant est un composant standard optionnel. Dans cet exemple, la valuation d'une seule variable entraîne la valuation (et donc la pertinence ou la non-pertinence) de la variable *Toit Ouvrant*. Le toit ouvrant existe, mais

n'est pas paramétrable. Nous retrouvons ce comportement dans certains *Benchmarks*[1], pour lesquels les composants optionnels sont modélisés de cette façon.

EX. 4.1 — Modélisation de composants standard optionnels



Une voiture est de type normal ou luxe, comporte des vitres (électriques ou manuelles) et un intérieur (cuir ou tissu). Si (et seulement si) la voiture est de type luxe, elle est équipée d'un toit ouvrant. L'existence du toit ouvrant contraint l'intérieur en cuir et les vitres électriques. Dans ce cas, nous obtenons le RCSP suivant :

– les variables : voiture (V_1), vitres (V_2), intérieur (V_3), toit ouvrant (V_4) ;
– les domaines : $D_1 = \{\text{luxe, normal}\}$, $D_2 = \{\text{électriques, manuelles}\}$, $D_3 = \{\text{cuir, tissu}\}$, $D_4 = \{\text{existe, n'existe pas}\}$, ;
– les contraintes :

$$C_1 : V_1 = \text{luxe} \Leftrightarrow V_4 = \text{existe}$$
$$C_2 : V_2 = \text{électriques} \Leftrightarrow V_4 = \text{existe}$$
$$C_3 : V_3 = \text{cuir} \Leftrightarrow V_4 = \text{existe}$$

Cette solution a l'avantage d'être particulièrement simple à mettre en œuvre : tous les moteurs CSP savent gérer des variables booléennes, l'adaptation de cette solution de gestion des composants optionnels est donc immédiate (aucune extension n'est nécessaire pour passer du RCSP au CSP). À ce stade, RCSP et CSP sont équivalents.

4.2 Modélisation de contraintes conditionnelles

Dans les besoins énoncés au chapitre 1, nous avons introduit la notion de *contrainte conditionnelle*. Une contrainte conditionnelle est une contrainte qui est pertinente sous certaines conditions. Par exemple, une contrainte entre deux composants n'est pertinente que si un troisième composant a une propriété particulière.

Comme nous l'avons vu au chapitre 2, nous avons défini les contraintes comme des formules logiques (cf. définition 2.2 page 22). La définition que nous avons adoptée pour les contraintes permet de modéliser des contraintes dont la pertinence dépend d'autres contraintes par des implications, tout en restant dans le cadre de CSP « classiques ». Ainsi, cette définition nous permet de modéliser des contraintes conditionnelles facilement. Une implication est un opérateur logique et son utilisation répond au besoin des contraintes conditionnelles en restant dans le cadre de CSP « classiques ».

[1] CONFIGIT, cf. http://www.itu.dk/research/cla/externals/clib/

Ainsi, lorsque la pertinence d'une contrainte est soumise à une condition, il est possible de modéliser ce comportement par l'utilisation d'une implication.

Sur l'exemple 4.2, la relation entre la largeur et la hauteur de la fenêtre est prise en compte si et seulement si la fenêtre est « bien proportionnée » ; cette implication forme la contrainte C_1. Lorsque la condition ($T =$ bien proportionnée) n'est pas vérifiée, la contrainte C_1 est vraie ; ainsi, la relation entre hauteur et largeur ne sera pas prise en compte. Nous pouvons aussi noter que lorsque le rapport entre hauteur et largeur est égal au nombre d'or, la contrainte C_1 sera vérifiée (que la condition ($T =$ bien proportionnée) soit vérifiée ou non).

EX. 4.2 — Modélisation d'une contrainte conditionnelle

Soit une fenêtre de hauteur h et largeur l. L'utilisateur a le choix entre plusieurs types de fenêtres : « haute », « large » ou « bien proportionnée ». Lorsque l'utilisateur choisit une fenêtre « bien proportionnée », le rapport entre la hauteur et la largeur doit être égal au nombre d'or. La configuration de la fenêtre sera modélisée par le RCSP suivant :

- les variables : hauteur h, largeur l et type T ;
- les domaines : $D_h = [50, 220]$, $D_l = [70, 300]$, $D_T =$ { haute, large, bien proportionnée } ;
- une seule contrainte : C_1 : $(T =$ bien proportionnée $) \Rightarrow (h = \frac{1+\sqrt{5}}{2} \times l)$.

Dans ce cas, aucune extension n'est nécessaire pour passer du CSP au RCSP.

4.3 Modéliser un composant paramétrable optionnel

4.3.1 Préliminaires

Avant de modéliser un composant paramétrable optionnel, nous devons clarifier deux problèmes :

- comment établir la pertinence de contraintes portant sur des variables non-pertinentes ou dont la pertinence n'est pas encore déterminée ;
- une fois les solutions d'un CSP traduit depuis un RCSP obtenues, comment obtenir les solutions du problème de conception.

Du traitement de contraintes portant sur des variables non pertinentes

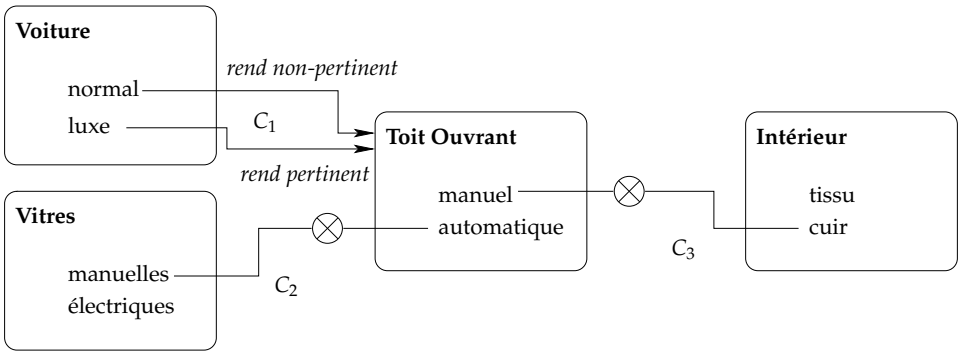
Un problème se pose lorsque le moteur de résolution ou filtrage doit traiter des contraintes portant sur des variables non-pertinentes ou dont la pertinence n'est pas déterminée. Nous posons donc le postulat 1.

Postulat 1 *Une contrainte sera considérée comme pertinente si et seulement si toutes les variables sur lesquelles elle porte sont pertinentes. Dans tous les autres cas, la contrainte est considérée comme satisfaite.*

Ainsi, les contraintes comportant des variables non-pertinentes ou dont la pertinence n'est pas déterminée sont considérées comme non pertinentes. Sur l'exemple 1.7 page 18, il est logique de ne pas considérer les contraintes reliant le toit ouvrant aux vitres ou à l'intérieur de la voiture lorsque le toit ouvrant n'est pas pertinent (le composant n'existe pas dans la voiture configurée).

Le choix inverse (considérer pertinentes les contraintes même lorsque certaines variables ne sont pas nécessairement pertinentes) pourrait procurer une meilleure qualité de filtrage. Cependant, ce choix provoquerait des phases de propagation sur les domaines de variables à pertinence non déterminée, qui risqueraient d'augmenter le temps de filtrage, et ce pour filtrer des domaines de variables à pertinence non déterminée (donc pouvant encore ne pas être pertinentes). De plus, ces phases de propagation inutiles pourraient mener à des incohérences : par le biais de variables à pertinence non déterminée, des domaines de variables pertinentes pourraient ne plus contenir de valeurs cohérentes. L'exemple 4.3 illustre ce choix inverse sur la voiture simple (exemple 1.7 page 18). En définitive, le choix illustré par le postulat 1 page précédente nous paraît le seul choix raisonnable à ce sujet.

EX. 4.3 — Illustration du filtrage sur des variables à pertinence non-déterminée



Supposons que le moteur effectue des réductions de domaine sur les variables à pertinence non déterminée. Si l'utilisateur choisit un intérieur cuir, le toit ouvrant ne peut qu'être automatique. Si l'utilisateur choisit ensuite des vitres manuelles, le problème deviendra incohérent. En effet, il ne reste plus de solutions pour le type du toit ouvrant.

Solutions d'un problème de conception

Étant donné que nous nous proposons d'effectuer de l'aide à la conception aussi bien autonome qu'interactive, nous devons définir ce que nous entendons par la solution de ces problèmes.

En ce qui concerne l'aide à la conception autonome, une solution correspond à une instanciation de toutes les variables pertinentes du RCSP telle que toutes les contraintes pertinentes sont satisfaites.

Pour l'aide à la conception interactive, nous définissons la solution comme un ensemble de variables pertinentes dont les domaines ont été filtrés pour satisfaire les contraintes pertinentes après une phase de filtrage. Ainsi, en conception interactive, un problème atteint une solution intermédiaire après chaque phase de filtrage (une phase de filtrage correspond à l'ajout d'une contrainte utilisateur : valuation de variable...).

Parmi les mécanismes que nous allons présenter, il sera souvent nécessaire d'adapter les solutions du RCSP aux solutions du problème de conception. Ainsi, une variable non-pertinente fait partie de la solution du RCSP parce qu'il est nécessaire de savoir qu'elle n'est pas pertinente, alors que l'élément du produit qu'elle représente (paramètre, composant, sous-ensemble) ne fait pas partie de la solution du problème de conception.

4.3.2 Modélisation d'un composant paramétrable optionnel par l'ajout d'une valeur au domaine

Pour modéliser un composant paramétrable optionnel, nous préconisons l'utilisation d'un symbole particulier (AMILHASTRE, 1999). Généralement, une étoile (★) est ajoutée au domaine des variables dont l'existence est conditionnée. Pour les RCSP, nous ajoutons donc la possibilité de rajouter ★ au domaine d'une variable discrète.

Pour des variables symboliques, un RCSP avec ★ possède l'avantage d'être assimilable à un CSP discret. En effet, l'ajout d'une valeur dans le domaine des variables symboliques a pour seul effet d'augmenter la combinatoire, sans nécessiter d'adaptation du moteur de résolution.

En revanche, un problème se pose pour les variables numériques : comment traiter la valeur ★ vis-à-vis des opérations mathématiques ? La plupart des moteurs CSP pouvant gérer des variables numériques ne permettent pas l'insertion de valeurs symboliques dans leurs domaines.

La figure 4.4 montre la modélisation du problème de la voiture illustré par l'exemple 1.7 page 18 sous forme d'un RCSP. Une voiture est de type luxe ou normal ; elle comporte des vitres électriques ou manuelles et un intérieur tissu ou cuir. Si (et seulement si) la voiture est de type luxe, elle est équipée d'un toit ouvrant, qui peut être manuel ou automatique (C_1). Un toit ouvrant automatique interdit des vitres manuelles (C_2) et un toit ouvrant manuel interdit un intérieur en cuir (C_3). Le RCSP modélisant ce problème en utilisant ★ pour la variable à pertinence non déterminée est formulé dans l'équation 4.1.

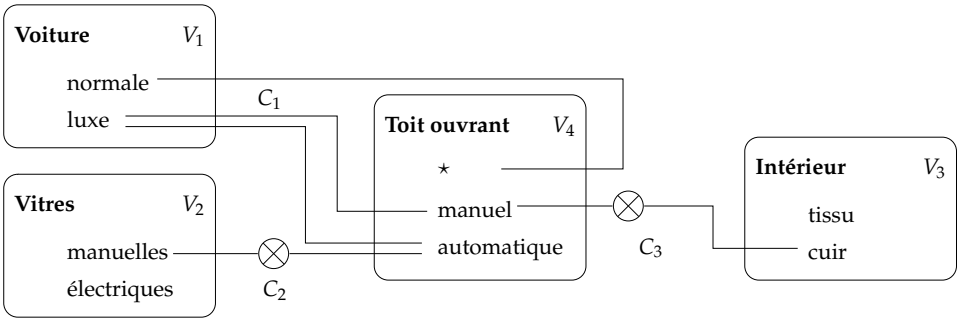


FIG. 4.4 — Configuration de voiture — modélisation par RCSP et ★

$$
\begin{array}{lll}
\boldsymbol{V} & : & \{V_1,\ V_2,\ V_3,\ V_4\} \\
\boldsymbol{D} & : & \{D_1(\text{ normal, luxe }),\ D_2(\text{ manuelles, électriques }), \\
 & & \ D_3(\text{ tissu, cuir }),\ D_4(\text{ ★, manuel, automatique })\} \\
\boldsymbol{C} & : & \{C_1,\ C_2\ C_3\} \\
 & & C_1 :\ V_1 = \text{luxe} \Leftrightarrow V_4 \neq \text{★} \\
 & & C_2 :\ V_4 = \text{automatique} \Rightarrow V_2 \neq \text{manuelles} \\
 & & C_3 :\ V_4 = \text{manuel} \Rightarrow V_3 \neq \text{cuir}
\end{array}
\tag{4.1}
$$

Les solutions au problème de conception représenté par la figure 4.4 sont données dans le tableau 4.5 page suivante. Dans ce tableau, nous pouvons constater que lorsque la *variable Toit Ouvrant* n'est pas pertinente, le *composant* toit ouvrant ne fait pas partie du produit (donc n'apparaît pas dans la nomenclature du produit fini).

TAB. 4.5 — Solutions au problème de conception de l'exemple 1.7 page 18

Variables / Solutions	Voiture	Vitres	Intérieur	Toit Ouvrant
Solution n° 1	normale	électriques	tissu	∄
Solution n° 2	normale	électriques	cuir	∄
Solution n° 3	normale	manuelles	tissu	∄
Solution n° 4	normale	manuelles	cuir	∄
Solution n° 5	luxe	manuelles	tissu	manuel
Solution n° 6	luxe	électriques	tissu	manuel
Solution n° 7	luxe	électriques	tissu	automatique
Solution n° 8	luxe	électriques	cuir	automatique

Traduction du RCSP en CSP— cas des domaines avec ★

Enfin, le postulat 1 page 51 implique une phase de traduction des contraintes pour pouvoir prendre en compte le fait que les variables non-pertinentes ne sont pas contraintes. Ainsi, l'équation 4.1 page précédente représentant le RCSP est traduite en un CSP « classique » de la forme suivante :

$$\begin{array}{lcl} & & \text{L'ensemble de variables :} \\ \boldsymbol{V} & = & \{\text{Voiture, Vitres, Intérieur, Toit ouvrant}\} \\ & & \{V_1, V_2, V_3, V_4\}, \\ & & \text{L'ensemble des domaines :} \\ \boldsymbol{D} & = & \{\{\text{normale, luxe}\}, \{\text{manuelles, électriques}\}, \\ & & \{\text{cuir, tissu}\}, \{\text{manuel, automatique, } \bigstar\}\} \\ & & \text{Les contraintes :} \\ \boldsymbol{C} & = & C_1 : (V_1 = \text{ luxe }) \Leftrightarrow (V_4 \neq \bigstar) \\ & & C_2 : (V_4 = \bigstar) \vee (V_4 = \text{ automatique } \Rightarrow V_2 \neq \text{ manuelles }) \\ & & C_3 : (V_4 = \bigstar) \vee (V_4 = \text{ manuel } \Rightarrow V_3 \neq \text{ cuir }) \end{array} \tag{4.2}$$

Nous pouvons retenir de cette solution qu'elle est bien adaptée à la modélisation par variables discrètes, par le fait qu'elle ne change pas la nature du problème.

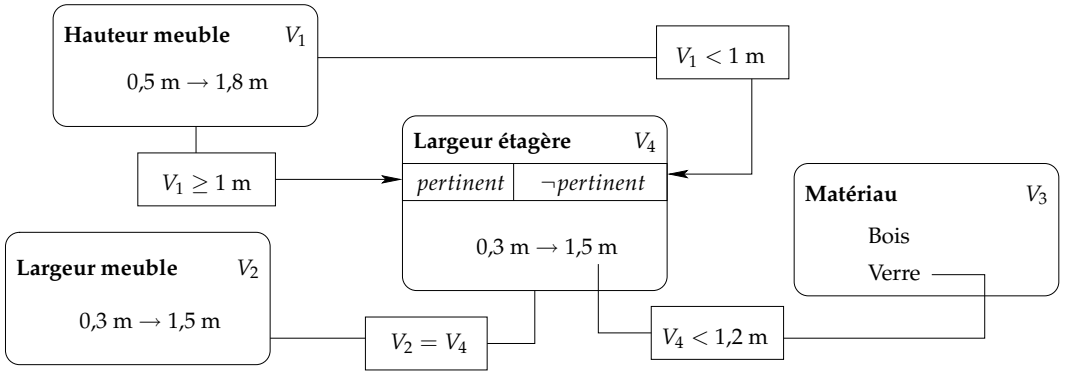
4.3.3 Modélisation par l'ajout d'un attribut de pertinence

Dans le cas d'un composant optionnel à un paramètre numérique, nous reprenons les propositions de VERON (2001). Nous attachons à chaque variable dont la pertinence n'est pas déterminée un « attribut de pertinence ». Dans un RCSP, un attribut peut donc être attaché à une variable.

L'avantage de cette modélisation par rapport à la modélisation avec le symbole ★ est que la variable garde son domaine original. Dans le cas où la variable dont nous devons déterminer la pertinence est numérique, elle peut être traitée par un moteur CSP numérique en utilisant les principes que nous avons détaillés dans le chapitre 2. Cependant, il est alors nécessaire de traduire un RCSP en CSP de façon à pouvoir le traiter avec un moteur CSP « classique ». Un moteur « classique » pourra alors traiter un CSP issu d'un RCSP, sans avoir à gérer le cas où l'on ajouterait une valeur symbolique (★) au domaine d'une variable numérique.

L'exemple 4.6 illustre la modélisation d'un problème de configuration mixte comportant des variables numériques à pertinence conditionnée.

EX. 4.6 — Configuration de meuble



Un meuble est composé d'un caisson de hauteur V_1 et de largeur V_2. Le meuble peut être en bois ou en verre. Si le caisson est suffisamment haut, il existe une planche au milieu (de la même longueur que le caisson), mais le problème devient incohérent si le meuble est en verre et le caisson trop long : la planche en verre ne peut excéder 1 m de long.

L'équation 4.3 montre le RCSP modélisant l'exemple 4.6 en utilisant un attribut de pertinence pour la variable *Planche*.

$$\begin{array}{rcl} \boldsymbol{V} & : & \{V_1,\ V_2,\ V_3,\ V_4\} \\ \boldsymbol{D} & : & \{D_1 = [0,5;\ 1,8]\text{m}, D_2 = [0,3;\ 1,5]\text{m}, \\ & & D_3 = \{\text{Bois, Verre}\}, D_4 = [0,3;\ 1,5]\text{m}\} \\ \boldsymbol{C} & : & \{C_1,\ C_2\ C_3\} \\ & & C_1 : \quad (V_1 \geq 1\text{m}) \Leftrightarrow (V_4\ \textit{pertinent}) \\ & & C_2 : \quad V_2 = V_4 \\ & & C_3 : \quad (V_3 = \text{ Verre }) \Rightarrow (V_4 < 1\text{m}) \end{array} \tag{4.3}$$

Dans ce cas, il y aura, comme pour la modélisation avec ★, une étape de traduction pour obtenir un CSP « classique ».

Traduction d'un RCSP en CSP— cas des attributs

Pour un moteur CSP « classique », il existe seulement des variables, domaines et contraintes. Ainsi, VERON (2001) a proposé de traduire chacun des attributs d'état par une variable d'état. Nous utiliserons le même mécanisme. Pour chaque variable dont la pertinence est à déterminer, nous obtiendrons dans le CSP deux variables :

- une variable de base qui est la variable à pertinence non-déterminée du RCSP ;
- une variable de pertinence, booléenne, qui représente la pertinence de la variable de base associée — nous noterons ses deux valeurs *pertinent* et ¬*pertinent*.

Pour le RCSP schématisé dans l'exemple 4.6, dont les contraintes sont reproduites dans l'équation 4.3, nous obtenons donc le CSP suivant :

$$
\begin{array}{lcl}
\text{L'ensemble de variables :} & & \\
\boldsymbol{V} & = & \{\text{hauteur du caisson, largeur du caisson, matériau,} \\
 & & \text{pertinence de la planche, planche}\} \\
 & & \{V_1, V_2, V_3, V_4^p, V_4^b\}, \\
\text{L'ensemble des domaines :} & & \\
\boldsymbol{D} & = & \{D_1 = [0,5;\ 1,8]\text{m}, D_2 = [0,3;\ 1,5]\text{m}, D_3 = \{\text{Bois, Verre}\}, \\
 & & D_4^p = \{pertinent, \neg pertinent\}, D_4 = [0,3;\ 1,5]\text{m}\} \\
\text{Les contraintes :} & & \\
\boldsymbol{C} & = & C_1 : (V_1 \geq 1\text{m}) \Leftrightarrow (V_4^p = pertinent) \\
 & & C_2 : (V_4^p = \neg pertinent) \vee (V_2 = V_4) \\
 & & C_3 : (V_4^p = \neg pertinent) \vee \big((V_3 = \text{ Verre }) \Rightarrow (V_4 < 1\text{m})\big)
\end{array}
\tag{4.4}
$$

La traduction des contraintes C_2 et C_3 fait apparaître une disjonction supplémentaire. Ceci est dû à l'utilisation du postulat 1 page 51 pour l'interprétation des variables non-pertinentes. Ainsi, si la variable *planche* est non-pertinente (donc la variable V_4^p est valuée à $\neg pertinent$), la contrainte est toujours satisfaite.

4.3.4 Conclusion sur la pertinence des composants

Nous avons vu trois méthodes pour modéliser la pertinence de composants optionnels (standard ou à un paramètre) ou de paramètres optionnels :

- l'utilisation d'une variable booléenne, qui permet de modéliser des composants standard optionnels ;
- l'ajout d'une valeur ★ au domaine, que nous préconisons pour gérer des composants optionnels à un paramètre symbolique ;
- l'ajout d'un attribut de pertinence, que nous préconisons pour gérer des composants optionnels à un paramètre numérique ou des paramètres optionnels numériques.

Les RCSP que nous avons proposés dans l'introduction permettent de regrouper ces trois méthodes dans un même formalisme. Pour modéliser des composants ou paramètres non optionnels, nous utiliserons des variables, de la même façon que dans un CSP « classique ». Nous établissons une première différenciation entre CSP « classique » et RCSP par le fait que des composants optionnels puissent être modélisés grâce à un attribut de pertinence attaché à la variable ou l'ajout d'une valeur au domaine. Cependant, lorsqu'un composant comporte plusieurs paramètres, nous devons pouvoir grouper ceux-ci dans un même sous-ensemble du RCSP. La section suivante propose deux méthodes de modélisation pour les sous-ensembles du produit, applicables aux composants à plusieurs paramètres.

4.4 Modélisation de sous-ensembles optionnels

La pertinence de sous-ensembles du produit permet d'éviter la redondance de la pertinence de plusieurs variables lorsqu'un composant optionnel est modélisé par plusieurs variables (éventuellement contraintes entre elles).

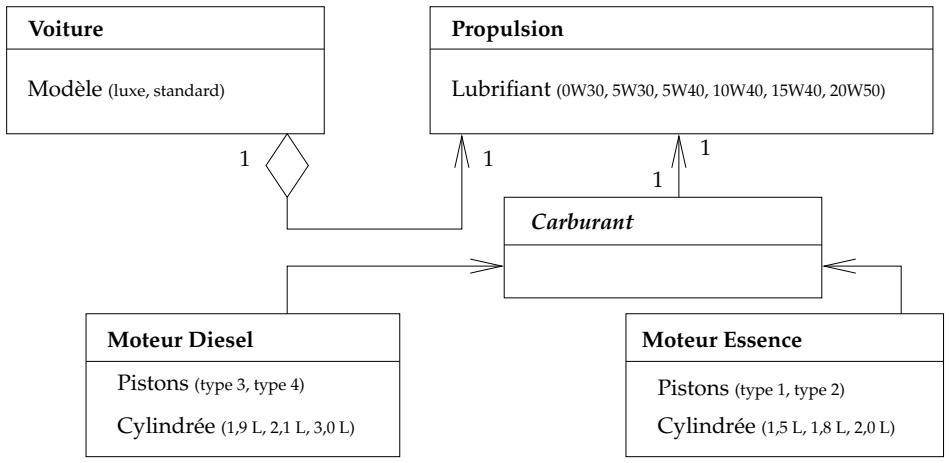
La solution consistant à gérer la pertinence de chacune des variables séparément oblige à ajouter des contraintes pour signifier leur regroupement et a pour conséquence de perdre l'aspect hiérarchique du modèle. En termes d'approches hiérarchiques, le principal besoin est de disposer d'un pouvoir expressif dans les CSP suffisant pour exprimer des groupes de composants, et de pouvoir ensuite les traiter pour l'aide à la conception. En effet, un

modèle produit est généralement exprimé grâce à des méthodes de modélisation utilisant les concepts d'entités et de relations (pour MERISE, cf. TARDIEU *et al.* (2000)) ou de classes, attributs et agrégation (pour UML, cf. RUMBAUGH *et al.* (2004)).

Nous allons utiliser l'exemple 4.7, qui est une adaptation de l'exemple d'une voiture présenté dans SABIN & FREUDER (1996), pour illustrer la factorisation d'attributs de pertinence.

EX. 4.7 — CSP hiérarchique, dérivé de SABIN & FREUDER (1996)

La figure ci-dessous représente une voiture, modélisée grâce à un diagramme de classe du formalisme UML.



Une voiture est d'un certain modèle, possède un élément « propulsion », qui utilise un certain lubrifiant. Cet élément possède un moteur, qui peut être soit essence soit diesel. Ce moteur comprend lui-même des pistons d'un certain type et une cylindrée (le type de piston contraint le volume de la cylindrée).

Pour traiter des modèles de produit définis hiérarchiquement, nous avons identifié dans le chapitre 3 deux approches : soit utiliser la méthode des attributs, factorisés au niveau du groupe, soit procéder par l'ajout de sous-problèmes au cours de la résolution ou du filtrage.

4.4.1 Factorisation de l'attribut de pertinence

Comme GELLER & VEKSLER (2005) pour les ACSP, nous préconisons de factoriser l'attribut de pertinence de variables et/ou contraintes. Ainsi, l'attribut booléen de pertinence peut être affecté à une variable, mais aussi à un groupe de variables et/ou contraintes. Nous ajoutons donc deux éléments aux RCSP : les groupes, éventuellement pourvus d'attributs.

En utilisant un attribut d'état pour le groupe, nous pouvons non seulement contrôler l'activité d'un ensemble de variables et/ou contraintes, mais aussi (grâce aux mécanismes étudiés dans la section 4.3) contrôler la pertinence des variables à l'intérieur de ces ensembles. Un groupe sera noté G, et son attribut de pertinence G^p.

La figure 4.8 page suivante illustre l'exemple 4.7 adapté à ce type de modélisation. Le RCSP en résultant fait l'objet de l'équation 4.5 page suivante.

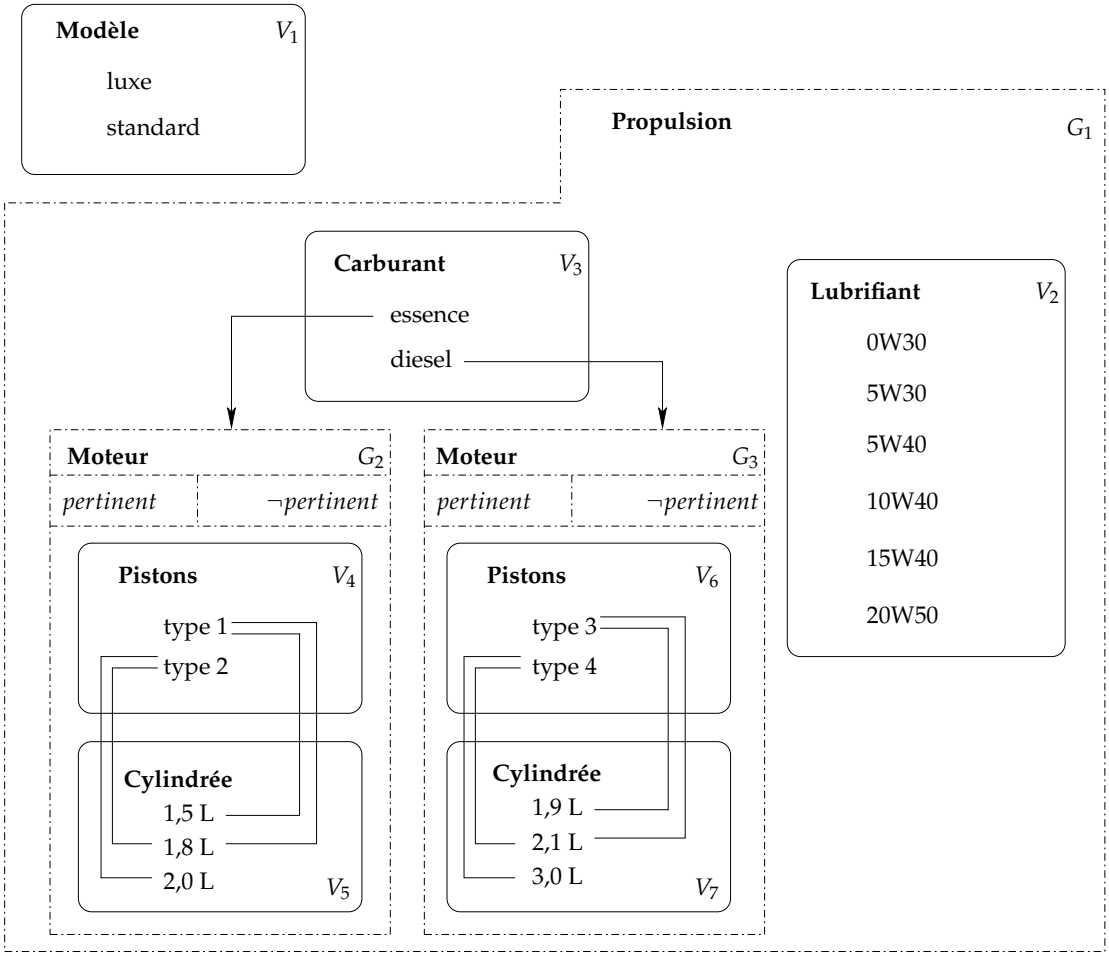


FIG. 4.8 — Configuration d'une voiture — factorisation des attributs de pertinence

Variables :
$V_1,\ V_2,\ V_3,\ V_4,\ V_5,\ V_6,\ V_7$
Domaines :
$D_1(\text{ luxe , standard }), D_2(\text{ 0W30, 5W30, 5W40, 10W40, 15W40, 20W50 }),$
$D_3(\text{ essence, diesel }),\ D_4(\text{ type 1, type 2 }),\ D_5(1,5\text{L},\ 1,8\text{L},\ 2,0\text{L}),$
$D_6(\text{ type 3, type 4 }),\ D_7(1,9\text{L},\ 2,1\text{L},\ 3,0\text{L})$
Groupes :
$G_1\big(V_2, V_3, G_2(G_2^p, V_4, V_5), G_3(G_3^p, V_6, V_7)\big)$ (4.5)
Contraintes :
$C_1 \rightarrow V_3 = \text{ essence } \Leftrightarrow (G_2^p = \ pertinent \wedge G_3^p = \neg pertinent)$
$C_2 \rightarrow V_3 = \text{ diesel } \Leftrightarrow (G_2^p = \neg pertinent \wedge G_3^p = \ pertinent)$
$C_3 \rightarrow (V_4 = \text{ type 1 } \Rightarrow V_5 \in \{1,5\text{L},\ 1,8\text{L}\})$
$\wedge (V_4 = \text{ type 2 } \Rightarrow V_5 \in \{1,8\text{L},\ 2,0\text{L}\})$
$C_4 \rightarrow (V_6 = \text{ type 3 } \Rightarrow V_7 \in \{1,9\text{L},\ 2,1\text{L}\})$
$\wedge (V_6 = \text{ type 4 } \Rightarrow V_7 \in \{2,1\text{L},\ 3,0\text{L}\})$

Nous pouvons noter sur l'équation 4.5 que la traduction en RCSP de l'exemple 4.7 ne contient pas de variables à pertinence conditionnée. Ainsi, aucun domaine n'est augmenté

de la valeur ★ et aucune variable n'est attachée à un attribut de pertinence.

Traduction d'un RCSP en CSP « classique »

Les contraintes du CSP traduit doivent être adaptées pour prendre en compte le fait que les variables G_i représentent la pertinence d'un ensemble de variables et contraintes. De la même façon que pour les attributs de pertinence par variable, nous proposons de traduire ces attributs factorisés par des variables, et d'utiliser le postulat 1 page 51 étendu, conduisant au postulat 2.

Postulat 2 *Un élément ne peut être pertinent que si le sous-ensemble auquel il appartient est pertinent.*

Par exemple, la contrainte C_3 de l'équation 4.5 page précédente sera traduite de la façon suivante :

$$\begin{array}{lll} C_1 & : & V_3 = \text{ essence } \Leftrightarrow (G_2^p = \ pertinent \wedge G_3^p = \neg pertinent) \\ C_2 & : & V_3 = \text{ diesel } \Leftrightarrow (G_2^p = \neg pertinent \wedge G_3^p = \ pertinent) \\ C_3 & : & (G_2^p = \neg pertinent) \vee \\ & & \quad \big((V_4 = \text{ type 1 } \Rightarrow V_5 \in \{1,5\text{L},\ 1,8\text{L}\}) \\ & & \qquad \wedge (V_4 = \text{ type 2 } \Rightarrow V_5 \in \{1,8\text{L},\ 2,0\text{L}\})\big) \\ C_4 & : & (G_2^p = \neg pertinent) \vee \\ & & \quad \big((V_6 = \text{ type 3 } \Rightarrow V_7 \in \{1,9\text{L},\ 2,1\text{L}\}) \\ & & \qquad \wedge (V_6 = \text{ type 4 } \Rightarrow V_7 \in \{2,1\text{L},\ 3,0\text{L}\})\big). \end{array}$$

Comme pour le premier postulat, celui-ci est nécessaire pour interpréter les contraintes dont l'une des variables appartiendrait à un élément non-pertinent (ou dont la pertinence n'est pas déterminée). Combiné au fait qu'une variable à l'intérieur d'un groupe à pertinence variable peut ne pas être active même si le groupe est actif, nous pouvons obtenir des modélisations permettant de traiter quasiment tous les modèles hiérarchiques de produits.

4.4.2 Ajout d'éléments

Introduction — les besoins

Nous voyons plusieurs intérêts à l'ajout d'éléments au cours de la conception ou configuration.

- Cette approche permet de ne pas évaluer toute une partie du problème sous une certaine condition, ce qui peut dans certains cas amener de bien meilleurs temps de réponses. À ce titre, cette approche peut être particulièrement intéressante pour maintenir l'interactivité.
- Plus particulièrement, ajouter des éléments au problème permettrait d'y connecter des moteurs de calcul externes (par exemple, du calcul de structures, calculs numériques de mécanique des fluides...). Ces calculs nécessitent un certain nombre de conditions (valuations de constantes...) avant de pouvoir être lancés ; l'ajout dynamique d'éléments permettrait de pouvoir vérifier ces conditions.

Ce type d'approche (l'ajout dynamique d'éléments) est défini par VERFAILLIE & JUSSIEN (2005) comme la seule pouvant mériter le nom de CSP dynamique (contrairement au vocabulaire employé par MITTAL & FALKENHAINER (1990)). La figure 4.9 page suivante illustre le fonctionnement d'ajouts d'éléments en cours de problème.

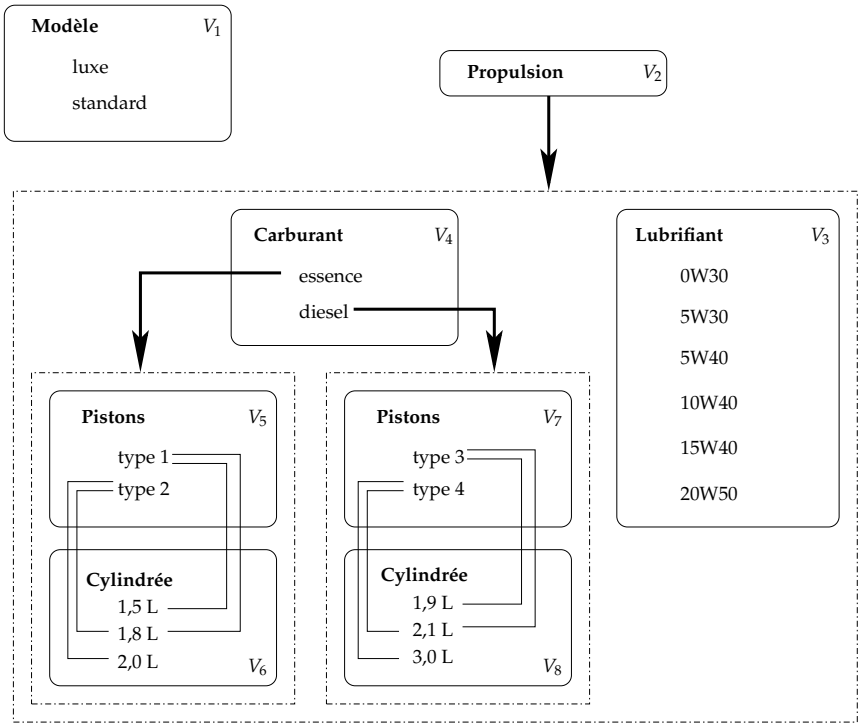


FIG. 4.9 — Configuration d'une voiture — modélisation par ajout de sous-problèmes

Nous identifions deux types de méthodes pour pouvoir ajouter des éléments au cours de la résolution d'un problème de conception :

– l'ajout au moyen d'une *méta-variable* ;
– l'ajout au moyen d'un *méta-opérateur*, composé d'une condition et d'une formule.

Nous nous basons sur la méthode utilisant une méta-variable proposée dans les CCSP (SABIN & FREUDER, 1996) pour proposer le méta-opérateur *Sure*.

Introduction — modélisation par une *méta-variable*

Cette méthode a été proposée par SABIN & FREUDER (1996). Elle consiste à considérer un groupe de variables et/ou contraintes comme une seule variable tant que celle-ci n'intervient pas dans le problème.

La figure 4.10 page suivante illustre le fonctionnement d'une méta-variable : dès que X_3 est valuée à une valeur particulière, le sous-problème contenant X_4, X_5, X_6 et les contraintes les reliant remplacent cette variable X_3 dans le problème initial.

Ainsi, sur l'exemple 4.7, adapté sur la figure 4.9, lorsque l'on prendra la propulsion en compte, cette variable sera remplacée par le groupe constitué des variables *Lubrifiant* et *Carburant*. Une fois la variable *Carburant* valuée, elle sera remplacée dans le problème par les variables *Pistons* et *Cylindrée*, qu'il faudra alors valuer pour arriver au terme du processus de configuration. En utilisant des méta-variables, les seules entités du CSP existant au début de la configuration sont celles de l'équation 4.6 page suivante.

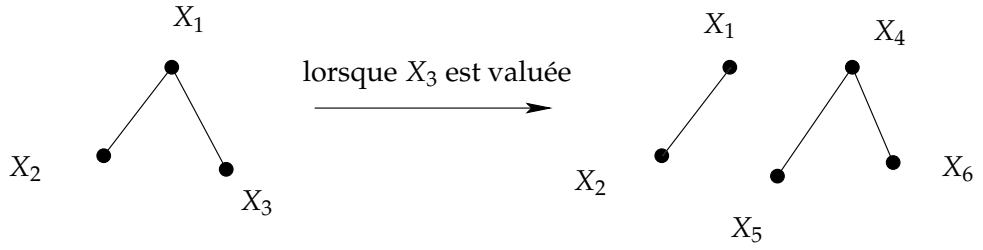


FIG. 4.10 — Fonctionnement de l'ajout d'éléments par méta-variable (CCSP) ; X_3 est la méta-variable

$$\begin{array}{l}\text{Variables :}\\ \quad V_1,\ V_2\\ \text{Domaines :}\\ \quad D_1(\text{ standard, luxe }),\ D_2()\\ \text{Contraintes :}\\ \quad C_1\ :\ \text{choix de } V_2 \Rightarrow (\nexists V_2,\ \exists V_3,\ V_4)\end{array} \tag{4.6}$$

Lorsque l'on prend en compte la propulsion de la voiture, la variable V_2 est alors remplacée dans le problème par les variables V_3 et V_4 et le problème devient :

$$\begin{array}{l}\text{Variables :}\\ \quad V_1,\ V_3,\ V_4\\ \text{Domaines :}\\ \quad D_1(\text{ standard, luxe }),\ D_3(\text{ 0W30, 5W30, 5W40, 10W40, 15W40, 20W50 }),\\ \quad D_4(\text{ essence, diesel })\\ \text{Contraintes :}\\ \quad C_1\ :\ (V_4 = \text{ essence }) \Rightarrow (\nexists V_4,\ \exists V_5,\ V_6)\\ \quad C_2\ :\ (V_4 = \text{ diesel }) \Rightarrow (\nexists V_4,\ \exists V_7,\ V_8)\end{array} \tag{4.7}$$

Enfin, si l'utilisateur choisit de valuer la variable V_4 à essence, le CSP final est représenté par l'équation 4.8 :

$$\begin{array}{l}\text{Variables :}\\ \quad V_1,\ V_3,\ V_5,\ V_6\\ \text{Domaines :}\\ \quad D_1(\text{ standard, luxe }),\ D_3(\text{ 0W30, 5W30, 5W40, 10W40, 15W40, 20W50 }),\\ \quad D_5(\text{ type 1, type 2 }),\ D_6(1,5\text{L},\ 1,8\text{L},\ 2,0\text{L})\\ \text{Contraintes :}\\ \quad C_1 \rightarrow V_5 = \text{ type 1 } \Leftrightarrow V_6 \in \{1,5\text{L},\ 1,8\text{L}\}\\ \quad C_2 \rightarrow V_5 = \text{ type 2 } \Leftrightarrow V_6 \in \{1,8\text{L},\ 2,0\text{L}\}\end{array} \tag{4.8}$$

Cette méthode a l'avantage de ne pas considérer le groupe avant qu'il ne soit activé par la valuation de la variable. Comme nous l'avons montré sur les équations 4.6 à 4.8, l'expression d'un modèle de produit est alors très simple. En revanche, il nous semble impossible d'exprimer des contraintes entre une variable membre du groupe et une autre extérieure

au groupe. En effet, s'il existait une contrainte entre le type de pistons et le lubrifiant, il serait impossible de la représenter par cette méthode. De plus, l'ajout de ces sous-problèmes ne peut être déclenché que par la valuation d'une variable à une certaine valeur. Pour pallier cet inconvénient et permettre l'utilisation d'une formule logique pour déclencher l'ajout d'un sous-problème, nous proposons la modélisation par méta-opérateur.

Modélisation par un *méta-opérateur*

Pour généraliser l'ajout d'éléments par méta-variable, en particulier l'impossibilité d'exprimer des conditions d'activation non limitées à une valeur, nous proposons un nouvel opérateur : l'opérateur *Sure* (sûr). Il se compose d'une condition (formule logique) et d'un sous-problème.

Cet opérateur permet d'*attendre* qu'une condition soit vraie pour la *remplacer* par un sous-ensemble. L'opérateur *Sure* est nommé méta-opérateur car il permet de raisonner sur *l'état* du raisonnement à un moment donné. Le fait d'être vrai implique l'attente que les domaines de certaines variables soient assez réduits pour que la condition d'activation ne puisse plus être fausse, quelles que soient les valeurs des variables. Cette condition d'activation est appelée une méta-contrainte car elle raisonne sur l'état du filtrage, et non sur les domaines des variables. Ce sous-problème permet la déclaration de nouvelles variables ainsi que de nouvelles contraintes.

La figure 4.11 montre le fonctionnement du méta-opérateur *Sure*. Lorsque les variables X_1 et X_2 ont leurs domaines suffisamment filtrés pour que la condition soit toujours vraie, celle-ci est remplacée par les variables X_4, X_5, X_6 et les contraintes les reliant entre elles et au reste du problème.

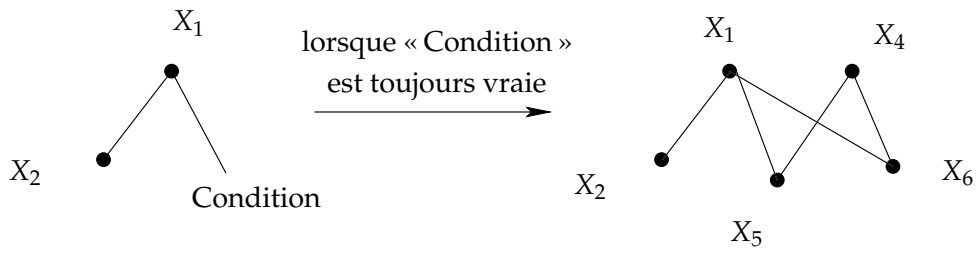


FIG. 4.11 — Fonctionnement du méta-opérateur *Sure*

Adapté à l'exemple 4.7 page 57, la contrainte permettant la création de l'un des groupes « Moteur » pourrait s'exprimer ainsi :

$$\begin{array}{ll} \mathit{Sure} & (\mathit{Carburant} = \text{ essence }) \\ & \{\exists\, V_5(\mathit{Pistons}), V_6(\mathit{Cylindrée}), C_3(V_5, V_6)\} \\ \wedge & \\ \mathit{Sure} & (\mathit{Carburant} = \text{ diesel }) \\ & \{\exists\, V_7(\mathit{Pistons}), V_8(\mathit{Cylindrée}), C_4(V_7, V_8)\}. \end{array}$$

De cette façon, les variables V_5 et V_6 (*resp.* V_7 et V_8) ne sont créées que si (et seulement si) la variable *Carburant* est valuée à essence (*resp.* diesel).

La formulation par méta-opérateur possède un avantage sur la formulation par méta-variable. En effet, nous avons vu que la modélisation par méta-variable convenait seulement pour des sous-problèmes indépendants. La modélisation par méta-opérateur permet

de s'affranchir de cette limitation des méta-variables. Il est donc possible d'exprimer des contraintes entre des variables ajoutées par le sous-problème et des variables existantes.

Sur l'exemple 4.7 page 57, s'il existait une contrainte entre le type de lubrifiant et le type des pistons ($V_3 \in \{ \text{0W30, 5W30} \} \Rightarrow V_5 \neq \text{type 2}$), nous pourrions la formuler avec un méta-opérateur de la façon suivante :

$$\begin{aligned} \mathit{Sure} \quad & (\mathit{Carburant} = \text{essence}) \\ & \{\exists\, V_5, V_6, C_3(V_5, V_6), C_5(V_3 \in \{ \text{0W30, 5W30} \} \Rightarrow V_5 \neq \text{type 2})\}. \end{aligned}$$

L'utilisation d'un méta-opérateur implique le fait pour le moteur de connaître son état lors de son fonctionnement. Ce contrôle du moteur étant dépendant de l'implémentation, nous reviendrons sur ce point dans le chapitre 6, à partir de la page 112. Du point de vue de la logique formelle, nous pouvons noter que le méta-opérateur *Sure* est équivalent à une implication. Nous définissons donc ce méta-opérateur comme un opérateur logique, qu'il est possible d'utiliser dans des formules au même titre que les opérateurs de la logique booléenne (et, ou...).

4.4.3 Conclusion sur la modélisation de sous-ensembles optionnels

Les méthodes pour gérer la pertinence de groupes ou sous-ensembles du produit dans la littérature sont basées sur l'utilisation d'un attribut de pertinence lié au groupe ou l'ajout de sous-problèmes. L'utilisation de groupes peut être contournée en passant par la pertinence particulière de chacun des éléments, mais ceci alourdirait la modélisation et obligerait le modeleur à saisir des « contraintes de groupe » pour lier tous ces éléments entre eux.

Les RCSP comprennent donc la possibilité de saisir des attributs de pertinence pour des sous-ensembles du produit, qui permettent de rendre compte de leur participation à la solution du problème de conception. La pertinence de tous les éléments du groupe (variables, contraintes, groupes) est ainsi factorisée. L'utilisation de cet attribut de pertinence factorisé par groupe permet ainsi de modéliser des composants optionnels à plusieurs paramètres.

La modélisation par méta-variable ou méta-opérateur permet de retarder l'évaluation de certaines parties du problème. Ce retard dans l'évaluation a plusieurs conséquences, en particulier sur l'aide à la conception interactive.

Tout d'abord, aucun raisonnement n'est effectué sur les sous-problèmes venant se connecter. Lors du filtrage, ceci réduit la qualité du filtrage. Cependant, cette absence de raisonnement peut être intéressante sur des sous-problèmes complexes et facultatifs : l'évaluation retardée permettra alors de maintenir l'aide à la conception interactive, avec des temps de réponse acceptables pour l'utilisateur.

Enfin, les procédés consistant à remplacer des pans de problèmes permettent de connecter facilement un moteur de calcul externe (par exemple, pour des calculs de structures, des moteurs de résolution numériques, des systèmes de *quad-trees* (SAMET, 1984; SAM, 1995)...).

Pour conclure sur l'ajout de sous-problèmes au cours de l'aide à la conception, nous proposons un nouveau méta-opérateur qui permet de résoudre certaines difficultés de la méthode à base de méta-variable proposée par SABIN & FREUDER (1996) : en particulier, il est alors possible de relier par contrainte des variables du sous-problème avec des variables déjà existantes. Nous utiliserons donc le méta-opérateur *Sure*, qui est une généralisation de la méthode par méta-variable.

4.5 Les RCSP

4.5.1 Définitions

Nous appellerons RCSP (pour *Relevancy* CSP, ou CSP à pertinence, cf. définition 4.1) un CSP permettant de gérer la pertinence de variables, de contraintes ou de groupes. Les RCSP sont issus des CSP★, des CSPe (VERON, 2001) et des ACSP (GELLER & VEKSLER, 2005). La pertinence des éléments peut être gérée par trois moyens différents :

- l'ajout d'une valeur ★ au domaine, pour les variables discrètes ;
- l'ajout d'un attribut de pertinence, pour les variables continues ou les groupes de variables et/ou contraintes ;
- l'ajout de sous-problèmes grâce à un méta-opérateur *Sure*.

Définition 4.1 : RCSP— Relevancy Constraint Satisfaction Problem

Un RCSP *est un quadruplet (V, D, G, C) où :*

- $\boldsymbol{V}$ *est un ensemble de variables ; les variables ayant une pertinence conditionnée peuvent être pourvues d'un attribut de pertinence booléen (variables numériques), attribut noté V_i^p pour la variable V_i ;*
- $\boldsymbol{D}$ *est l'ensemble des domaines de définition des variables ; les variables discrètes ayant une pertinence conditionnée peuvent avoir un domaine augmenté de la valeur ★, signifiant leur non-pertinence ;*
- $\boldsymbol{G}$ *est l'ensemble des groupes (un groupe est un quadruplet V, D, G, C)) ; les groupes ayant une pertinence conditionnée sont pourvus d'un attribut de pertinence noté G_i^p pour le groupe G_i ;*
- $\boldsymbol{C}$ *est un ensemble de contraintes ; une contrainte est une formule logique pouvant impliquer des variables du problème ou de l'un de ses sous-groupes, des attributs de variables et des attributs de groupes.*

Définition 4.2 : *Solution d'un* RCSP

Une solution d'un RCSP *est une instanciation de tous les attributs de pertinence pertinents (qui font partie de groupes pertinents) et de toutes les variables pertinentes telle que toutes les contraintes sont satisfaites.*

Les attributs de pertinence héritent eux-mêmes de la pertinence des groupes qui les contiennent. En effet, le postulat 2 page 59 implique que les attributs de pertinence compris à l'intérieur d'un groupe ne seront pertinents que si leur groupe englobant est pertinent.

4.5.2 Agrégation des solutions

L'étape d'agrégation des solutions correspond à la traduction inverse : des CSP « classiques » en RCSP. Une fois les solutions du CSP « classiques » identifiées, nous devons alors les agréger pour retrouver les solutions du RCSP, puis déterminer les solutions du problème de conception.

En ce qui concerne la modélisation par le symbole ★, cette agrégation est triviale : chaque paramètre ou composant modélisé par une variable valuée à ★ (respectivement dont la valeur ★ appartient au domaine de définition) n'existe pas dans la solution du problème (*resp.* peut ne pas appartenir à la solution du problème).

Pour la modélisation par attribut de pertinence, chaque solution du problème de conception correspond à une solution du RCSP. Chaque solution du RCSP est une classe d'équivalence de solutions du CSP traduit. Lorsque nous traduisons un RCSP en CSP pour le traiter, chaque variable à pertinence indéterminée est modélisée par deux variables (une variable de base et une variable de pertinence). Nous pouvons alors distinguer trois cas :

- la variable de pertinence est valuée à *pertinent* : la solution du RCSP est alors équivalente à la solution du CSP traduit et le composant du problème de conception modélisé par la variable dont la pertinence était indéterminée existe ;
- la variable de pertinence n'est pas valuée[2] : il n'est alors pas encore possible de se prononcer sur l'existence du composant dans le problème de conception (la conception n'est pas finie — il s'agit d'un résultat intermédiaire) ;
- la variable de pertinence est valuée à $\neg pertinent$: il existe alors plusieurs solutions du CSP traduit équivalentes pour le RCSP. Ainsi, toutes les valuations de la variable de base correspondent à la même solution du RCSP : celle pour laquelle cette variable n'est pas pertinente ; pour le problème de conception, le composant n'existe pas.

La figure 4.12 illustre l'agrégation des solutions pour les RCSP au travers de l'exemple 1.7 page 18 et des deux modélisations que nous en avons fait (par ★ ou attribut de pertinence).

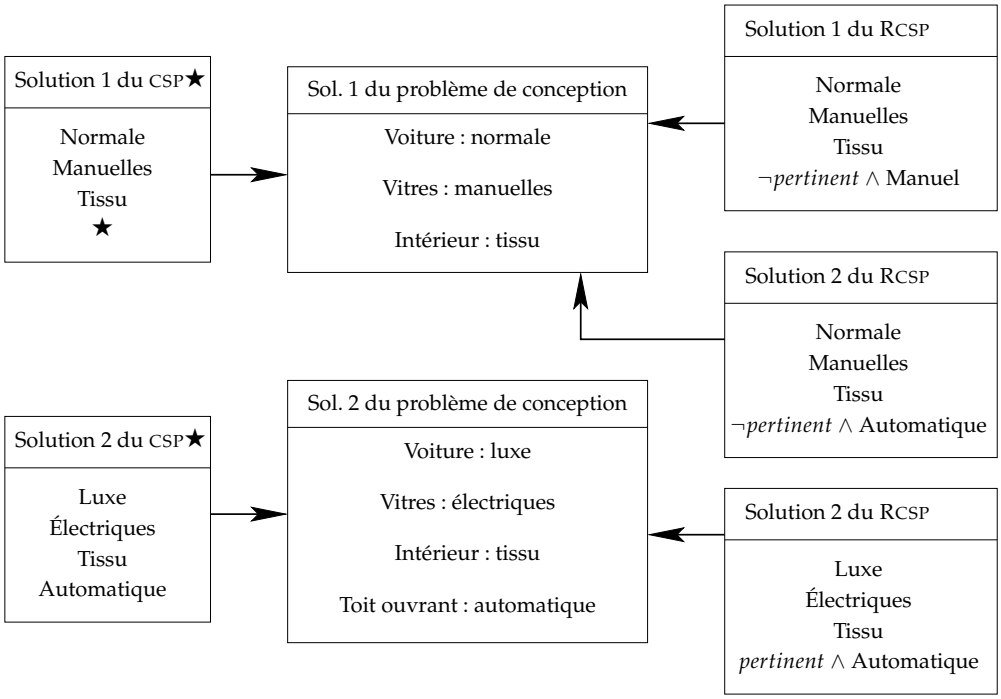


FIG. 4.12 — Agrégation des solutions — des solutions du RCSP aux solutions du problème de conception

Le tableau 4.13 page suivante propose une synthèse des différentes opérations d'agrégation selon le type du CSP et le problème de conception. Dans ce tableau, le composant dont nous voulons déterminer l'existence (première colonne) est modélisé par la variable X_i (deuxième colonne) si l'on utilise une ★ (composant ou paramètre discret) et par les variables V_i^p et V_i^b (troisième et quatrième colonnes) si l'on utilise un attribut de pertinence.

[2]Ce cas ne peut se produire que lors d'une aide à une conception interactive, pour la présentation intermédiaire à l'utilisateur de résultats du filtrage.

4.5.3 RCSP— synthèse

Nous avons proposé les RCSP, qui constituent une synthèse des CSP permettant une manipulation des concepts que nous avons détaillés dans les besoins : traitement de différents types de variables, gestion hiérarchique des éléments et gestion de leur pertinence.

Ce formalisme permet de modéliser des problèmes de conception pour fournir des outils informatiques d'aide à la conception. Cependant, plusieurs étapes de traduction et d'agrégation sont nécessaires. Les étapes de traduction consistent à obtenir un CSP « classique », permettant d'utiliser des moteurs existants. L'étape d'agrégation comporte deux phases : la première consiste à passer des solutions du CSP « classique » aux solutions du RCSP ; la deuxième consiste à extraire des solutions du RCSP les solutions du problème de conception.

4.6 Conclusion

Nous avons présenté dans ce chapitre ce que nous entendions par pertinence pour chacun des éléments d'un CSP, et adapté ce concept de pertinence aux groupes de variables et/ou contraintes, de façon à prendre plus facilement en compte des modèles produits, généralement formalisés de façon hiérarchique.

Nous pouvons donc conclure sur les différentes méthodes à utiliser pour chacun des cas.

- Pour un composant standard optionnel, nous utiliserons préférentiellement une variable booléenne.
- Pour un composant optionnel à un paramètre symbolique, l'ajout d'une valeur signifiant la non-pertinence dans son domaine est généralement la meilleure solution.
- Pour un composant optionnel à un paramètre numérique, nous utiliserons un attribut de pertinence, transformé par traduction en variable booléenne de pertinence.
- Pour les composants optionnels à plusieurs paramètres, nous proposons de factoriser l'attribut de pertinence.

En ce qui concerne les contraintes conditionnelles, nous pouvons gérer leur activité par le biais de formules logiques.

Nous avons également abordé l'ajout de sous-problèmes en cours de résolution ou de filtrage, ce qui consiste à retarder l'évaluation de sous-problèmes pour améliorer l'interactivité, ou attendre certaines conditions pour le lancement d'un moteur de calcul externe.

TAB. 4.13 — Synthèse des agrégations de solutions pour les problèmes de conception

Problème de conception : existence du composant	modélisation du RCSP par valeur ★ domaine de X_i	par attribut de pertinence valeur de V_i^p	valuation de V_i^b
$\exists$	$\bigstar \notin D_{X_i}$	*pertinent*	nécessaire
$\nexists$	$D_{X_i} = \{\bigstar\}$	$\neg$*pertinent*	inutile
non déterminé[a]	$\bigstar \in D_{X_i}$	{*pertinent*, $\neg$*pertinent*}	à déterminer

[a] En cours de configuration interactive...

Pour une activité de conception, cet ajout de sous-problèmes peut être utile pour gérer des sous-ensembles du produit.

Nous avons ainsi proposé les RCSP (*Relevancy* CSP), qui permettent d'exploiter plusieurs principes pour modéliser la pertinence d'éléments et leur hiérarchie. Ce formalisme de modélisation intègre les différents formalismes issus des CCSP, CSP★, CSPe et ACSP.

Chapitre 5

Implémentation des CSP sous forme d'arbres syntaxiques

Dans le chapitre précédent, nous avons retenu les différents mécanismes à mettre en œuvre pour traiter tous nos besoins en conception. Nous nous proposons de mettre au point un moteur CSP (une implémentation) permettant de traiter les problèmes à base de contraintes.

Notre approche se base sur le fait que les tables de compatibilité (symboliques, ou éventuellement étendues au numérique) et les formules mathématiques constituent les outils majoritairement utilisés pour les problèmes de conception ou de configuration à base de contraintes. Nous avons donc besoin d'une implémentation exploitant simultanément des contraintes symboliques, numériques et mixtes.

Nous nous sommes donc appuyés sur les expériences précédentes du laboratoire pour développer une implémentation basée sur :

- un langage de description de CSP permettant des compositions logiques, plus expressives que les tables de compatibilité, et plus adaptées à notre définition des contraintes (qui sont des formules logiques, cf. définition 2.2 page 22) ;
- un analyseur de ce langage, qui construit un arbre syntaxique représentant un CSP (discret, numérique ou mixte) ;
- les moteurs de résolution et de filtrage utilisent cette structure d'arbre syntaxique.

Ce chapitre détaille donc ces trois points : le langage, la structure d'arbre syntaxique et les moteurs de résolution et de filtrage ; le moteur de résolution se limite aux CSP discrets, alors que le moteur de filtrage prend en compte les CSP mixtes. Comme nous l'avons vu dans le chapitre précédent, cette implémentation nous permettra de traiter des CSP « classiques » issus de la traduction de RCSP. Nous présenterons alors dans le chapitre suivant des enrichissements nous permettant de tenir compte des spécificités des RCSP.

5.1 Le langage de description

Un CSP « classique » est un triplet (variables, domaines, contraintes) non pourvu de systèmes de gestion de la pertinence ou de la hiérarchie. Le langage doit donc pouvoir exprimer des déclarations de variables, de domaines et de contraintes.

5.1.1 Les domaines

Nous avons identifié trois types de domaines :
- booléens — de domaine $\{0, 1\}$;
- symboliques — dont les domaines peuvent comprendre tout symbole, ces domaines sont de cardinal fini, ou définis par leur complémentaire ;
- numériques ou définis par multi-intervalles, le cardinal est alors infini.

Ces trois types ont un comportement par défaut. Un domaine booléen contiendra les deux valeurs 0 et 1 si sa taille d'origine n'est pas précisée. Un domaine symbolique par défaut (sans symboles précisés) est infini : il comprend *tous* les symboles ; de même pour les domaines numériques, qui contiennent tous les réels par défaut.

Les domaines symboliques peuvent être déclarés comme une liste de symboles exclus : les accolades désignant les symboles sont alors inversées et les valeurs à exclure sont listées. Pour les domaines numériques, il est possible de déclarer un intervalle, une liste d'intervalles (inclus ou exclus), des opérations sur les domaines (union, intersection, différence...) ou des « intervalles implicites » (par exemple : $D = \{> 0\}$).

La syntaxe de déclaration de domaine est donc la suivante : le mot-clef `domain`, le type du domaine suivi du nom choisi ; il est alors possible de saisir le domaine (liste de valeurs ou d'intervalles suivant le type du domaine) ou de saisir des domaines par opérations sur des intervalles (union, intersection...). La syntaxe est la suivante :

Syntaxe : `domain <type> <nom> [description] ;`

avec { `type` compris dans `boolean`, `symbolic` ou `float`
`description` peut être une liste de valeurs ou
une liste d'intervalles ou
des opérations sur des domaines $(\cup, \cap, \ldots)$

Par exemple, les déclarations de domaine peuvent se faire ainsi :
- un domaine booléen par défaut qui contient les deux valeurs 0 et 1 :

```
domain boolean A ;
```

- un domaine symbolique comportant une liste de valeurs exclues :

```
domain symbolic B = }"c", "d"{ ;
```

- un domaine numérique comme complémentaire (exclusion d'un intervalle) :

```
domain float C = }]-oo, -10]{ ;
```

- un domaine numérique à l'aide d'un opérateur de comparaison :

```
domain float D = {> 0} ;
```

- un domaine numérique comme opérations sur des domaines :

```
domain float E = intersection(C, D) ;
```

Il est aussi possible (en particulier pour les domaines numériques) de composer plusieurs types de déclarations. Ces différentes déclarations de domaines permettent ensuite de les assigner à des variables.

5.1.2 Les variables

Pour correspondre aux trois types de domaines, il existe trois types de variables : booléenne, symbolique ou numérique. Les variables se déclarent en leur assignant un domaine préalablement déclaré, ou en leur associant un domaine (de la même façon que pour les déclarations de domaines). Elles sont déclarées par leur type, suivi du mot-clef `variable`, suivi du nom de la variable et d'un domaine (celui-ci peut être un domaine préalablement déclaré ou un domaine saisi « à la volée »). La syntaxe générale de définition d'une variable est :

Syntaxe : `<type> variable in <domain definition>;`

avec { `type` compris dans `boolean`, `symbolic` ou `float` ; `domain definition` peut être un nom de domaine ou une description de domaine }

Ainsi, nous pourrons avoir les formes suivantes de déclarations :

```
boolean variable a in A;
symbolic variable b in B;
symbolic variable c in {"i", "j", "k"};
float variable d in D;
float variable e in {[-10, 10]};
```

5.1.3 Les contraintes

Comme nous l'avons vu, les contraintes sont des formules logiques. Ainsi, chaque contrainte est saisie grâce à des opérateurs logiques permettant d'exprimer des conjonctions et/ou disjonctions entre des formules de comparaison.

Les opérations logiques classiques sont la négation (not), la conjonction (and), la disjonction (or), l'implication (=>) et l'équivalence (<=>). Ces cinq opérateurs logiques sont implémentés dans la syntaxe, ainsi que les opérateurs de comparaison symbolique et numérique permettant de saisir des formules logiques comprenant des variables symboliques et/ou numériques. Ces opérateurs de comparaison sont l'égalité et l'inégalité (symbolique et numérique), l'appartenance et la non-appartenance symboliques et les opérateurs de comparaison numériques suivants : $<$, $>$, $\geq$, $\leq$. Les opérateurs mathématiques courants ($+$, $-$, $\times$, $\div$, puissance) ont aussi été implémentés. Les contraintes se saisissent donc grâce à ces opérateurs, aux opérateurs de comparaison (symboliques ou numériques) et aux opérateurs logiques, par exemple comme ceci pour les contraintes de l'exemple de la voiture (cf. page 22) :

```
(voiture eq "luxe") <=> (toit_ouvrant eq "automatique") ;
(toit_ouvrant eq "automatique") => (vitres eq "électriques") ;
(intérieur eq "cuir") => (toit_ouvrant eq "automatique") ;
```

5.1.4 Conclusion

Ce langage de description nous permet de décrire des CSP en déclarant des domaines, des variables auxquelles on associe un domaine puis des contraintes, qui sont des formules logiques, conformément à notre définition 2.2 page 22. Les particularités de cette implémentation d'un langage de description se situent au niveau des domaines et des contraintes :

- il est possible de décrire des domaines en effectuant des opérations assemblistes sur des intervalles (unions, intersections, différences...)
- les contraintes sont saisies comme des formules logiques : nous pouvons ainsi utiliser des contraintes définies sous quelque forme que ce soit.

Les opérateurs implémentés sont détaillés dans la section suivante.

5.2 L'analyseur

Le langage décrit à la section précédente doit ensuite être interprété pour pouvoir traiter les CSP qu'il décrit. Pour exploiter la décomposition des contraintes en opérateurs logiques, puis en opérateurs de comparaison, et éventuellement en opérateurs de calcul numériques, nous avons choisi de représenter un CSP par un arbre syntaxique.

Un arbre syntaxique est donc une représentation d'un CSP, dont la racine est la conjonction de toutes les contraintes (cf. figure 5.1).

5.2.1 Opérateurs logiques

De la racine sont issues les formules logiques représentant les contraintes, comprenant un ou plusieurs opérateurs logiques. Par exemple, la contrainte ci-dessous comporte deux niveaux logiques que nous voyons apparaître dans l'arbre des contraintes, comme l'illustre la figure 5.1 avec la contrainte suivante :

```
(c eq "j") => ((e < 100) and (e > 70));
```

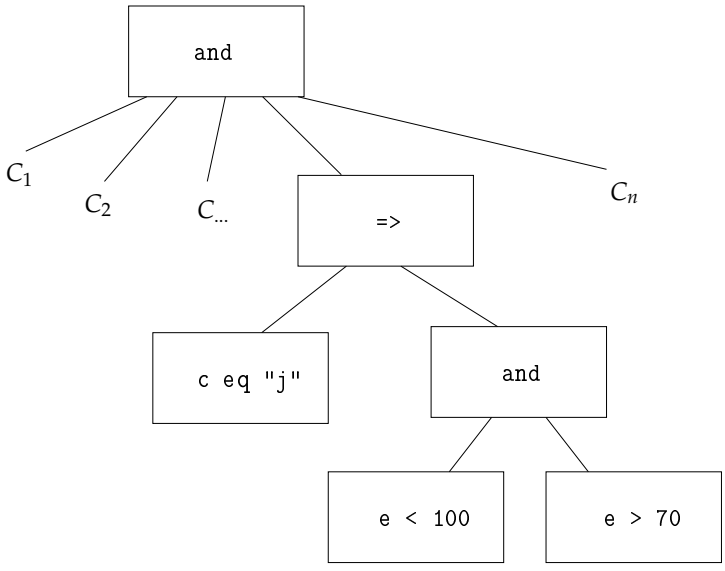


FIG. 5.1 — Racine et niveau logique de l'arbre syntaxique : conjonction des contraintes

Les opérateurs logiques constituent les nœuds de l'arbre ; les opérateurs logiques suivants ont été implémentés et sont reconnus par l'analyseur :
- conjonction : and (par exemple : `A and B`), d'arité n,
- disjonction : or (par exemple : `A or B`), d'arité n,
- implication : => (par exemple : `A => B`), d'arité 2,
- équivalence : <=> (par exemple : `A <=> B`), d'arité 2,
- négation : not (par exemple : `not A`), d'arité 1.

Pour les nœuds logiques, il faut préciser l'arité des opérateurs. En effet, les nœuds `and` et `or` peuvent avoir plusieurs opérandes, alors que les nœuds `=>` et `<=>` n'en ont que deux, et le nœud `not` seulement un. Il est possible d'ajouter des opérateurs (par exemple, le OU EXCLUSIF) ; il faut alors ajouter un mot-clef dans la grammaire pour l'analyseur (par exemple, `xor`) et préciser son arité, ainsi que la façon de construire les branches issues de ce nœud.

Les opérateurs logiques constituent les nœuds de l'arbre. Les feuilles de l'arbre syntaxique sont des opérateurs de comparaison, symboliques ou numériques.

5.2.2 Feuilles de comparaison symbolique

Les feuilles de comparaison symbolique sont les filles de nœuds logiques ; elles doivent donc retourner une valeur logique et servent à comparer une variable symbolique à une valeur symbolique, à un domaine symbolique ou à une autre variable symbolique. Elles peuvent être de trois types :
- égalité : eq (par exemple : `a eq "value"` ou `a eq b`),
- inégalité : ne (par exemple : `a ne "value"` ou `a ne b`),
- appartenance à un ensemble : in (par exemple : `a in {"v1", "v2"}`, qui peut aussi s'exprimer grâce à un domaine : `a in A` si le type du domaine correspond au type de la variable),
- non-appartenance à un ensemble : notin (par exemple : `a notin {"v1", "v2"}`, qui peut aussi s'exprimer grâce à un domaine : `a notin A` si le type du domaine correspond au type de la variable).

5.2.3 Feuilles de comparaison numérique

Les feuilles de comparaison numérique doivent aussi retourner une valeur logique. Elles servent à comparer deux formules numériques (une formule peut être une variable, un domaine ou une fonction mathématique). Les feuilles de comparaison numérique implémentées sont les suivantes :
- égalité : = (par exemple : `X = Y`),
- inégalité : != (par exemple : `X != Y`),
- inférieur ou égal : <= (par exemple : `X <= Y`),
- strictement inférieur : < (par exemple : `X < Y`),
- strictement supérieur : > (par exemple : `X > Y`),
- supérieur ou égal : >= (par exemple : `X >= Y`).

Il faut noter que les feuilles de comparaison numérique ne sont pas réellement des feuilles car elles peuvent contenir des formules mathématiques, qui sont implémentées sous la forme d'un arbre numérique. La figure 5.2 page suivante montre une feuille numérique représentant la contrainte numérique $(a - b)^2 < c$: chaque opérateur de calcul forme alors un nouveau nœud.

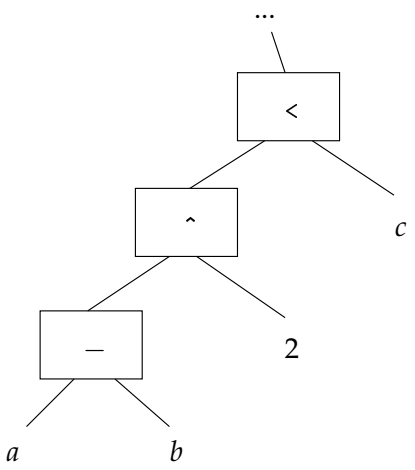


FIG. 5.2 — Feuille de comparaison numérique de la formule $(a-b)^2 < c$

5.2.4 Domaines dans l'arbre syntaxique

L'analyseur transforme les déclarations de domaines en contraintes de type in. Ainsi, ces déclarations de domaines :

```
domain float A = [0, 10];
symbolic variable x in {"i", "j", "k"};
float variable a in A;
```

ont pour effet d'ajouter à la racine de l'arbre syntaxique des contraintes d'appartenance, comme illustré sur la figure 5.3.

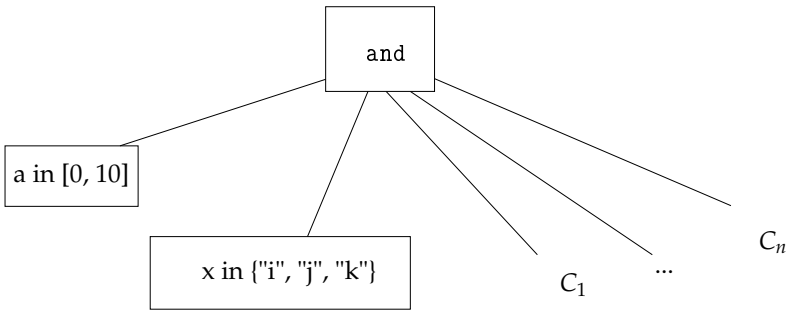


FIG. 5.3 — Analyse des déclarations de domaine : feuilles résultantes

5.2.5 Bilan

Nous avons implémenté un analyseur permettant d'interpréter un CSP saisi avec notre langage de description et produisant un arbre syntaxique représentant le CSP : variables, domaines et contraintes.

Si nous reprenons l'exemple de la voiture simple (exemple 2.1 page 22) vu au chapitre 2, nous obtenons l'arbre syntaxique suivant de la figure 5.4 page ci-contre.

Les moteurs de résolution et de filtrage se basent sur cette structure d'arbre syntaxique pour respectivement rechercher des solutions et réduire les domaines des variables.

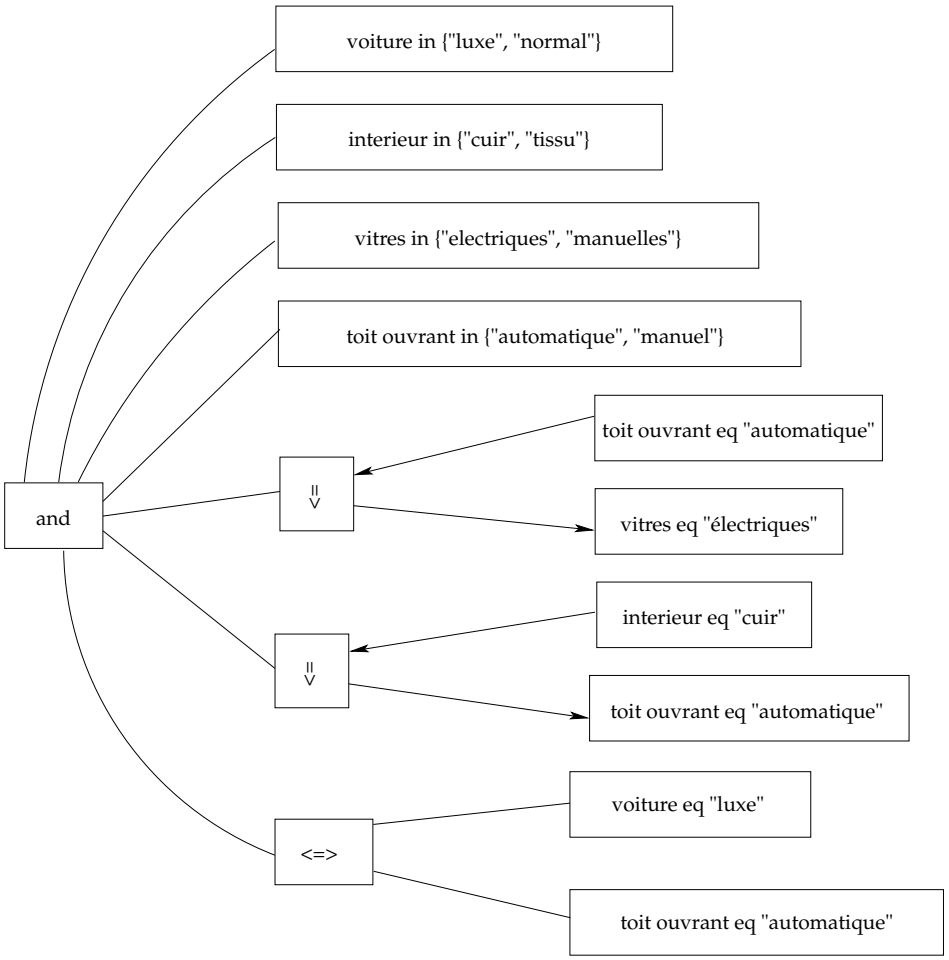


FIG. 5.4 — Arbre complet de l'exemple « voiture simple »

5.3 Le moteur de résolution

Le moteur de résolution se limite aux CSP discrets. En effet, comme nous l'avons vu au chapitre 2, il est très délicat de fournir les solutions d'un CSP numérique sans approximations. Ne souhaitant pas nous limiter à la recherche d'une (ou de quelques) solution, le moteur de résolution que nous avons implémenté ne tiendra donc pas compte des variables numériques continues.

5.3.1 Évaluation d'un nœud

À chaque nœud est associé un domaine booléen ($\{0,1\}$). Lorsque l'instanciation permet de tester le nœud (toutes les variables du nœud sont valuées), ce domaine booléen est alors filtré pour ne laisser que la valeur 0 si l'instanciation est incohérente ou la valeur 1 si l'instanciation est cohérente. La définition 5.3 décrit le fonctionnement de la procédure CALC qui effectue cette vérification.

Procédure 5.3 : CALC

La procédure est implémentée pour chacun des types de nœuds dans l'arbre syntaxique. Son argument d'appel est une contrainte (un nœud, comprenant tous ses fils).

Elle utilise ensuite les domaines (éventuellement réduits) des variables pour déterminer la valeur de vérité du nœud :

- *"0" si la contrainte est insatisfaite ;*
- *"1" si la contrainte est satisfaite.*

Ainsi, la variable « coherent » dans l'algorithme 5.5 peut prendre plusieurs valeurs. Lorsque tous les domaines booléens de tous les nœuds sont valués à 1, l'instanciation est une solution du problème. Lorsqu'il existe une contrainte pour laquelle « coherent » vaut "0", l'instanciation n'est plus cohérente et il faut alors retourner en arrière (*id est*, choisir une autre valeur pour la dernière variable instanciée).

5.3.2 Mécanisme

Nous utilisons un algorithme proche du *BackTrack* pour résoudre un CSP de façon autonome. Nous instancions les variables les unes après les autres, en vérifiant que les contraintes sont satisfaites. Du point de vue de la résolution, l'arbre syntaxique est vu comme une conjonction de contraintes (cf. figure 5.1 page 72).

Une fois que toutes les variables d'une contrainte C_i ont été valuées, nous évaluons la contrainte C_i. Si cette contrainte est satisfaite, d'autres variables sont alors valuées jusqu'à atteindre une solution. Si cette contrainte n'est pas satisfaite, il faut alors tester d'autres valeurs pour les variables précédemment instanciées (principe du *BackTrack*).

L'algorithme 5.5 montre l'adaptation du *BackTrack* conformément à notre implémentation. Sa principale différence avec un algorithme de *BackTrack* traditionnel se situe au niveau de la procédure CALC.

Alg. 5.5 BACKTRACKING (V_v, V_n)

```
– ∴ – Adaptation du BackTrack à notre implémentation — trouve toutes les solutions du CSP (X, D, C)
– ∴ – Deux ensembles ordonnés :
– ∴ – V_v l'ensemble des couples (variable valuée, valeur choisie)
– ∴ – V_n l'ensemble des couples (variable à instancier, domaine d'instanciation)
coherent ← 1
Si (V_n = ∅) Alors
    V_v est une solution
Sinon
    CHOISIR & EFFACER x dans V_n
    Pour Chaque i ∈ D_x Faire
        IDENTIFIER C_x ∈ C tel que C_x contraint x et des variables appartenant à V_v
        coherent ← CALC (C_x, V_v)      – ∴ – cf. définition 5.3 page précédente
        Si (coherent) Alors
            BACKTRACKING (V_v ∪ (x, i), V_n \ x)
        Fin Si
    Fin Pour
Fin Si
```

La procédure CHOISIR & EFFACER consiste à prendre la variable suivante à instancier, étant donné que les ensembles V_v et V_n sont ordonnés.

5.3.3 Ordre d'instanciation des variables

Dans l'algorithme 5.5 page ci-contre, la procédure CHOISIR & EFFACER peut faire appel à des heuristiques pour sélectionner les variables à instancier. Nous pouvons citer ici les heuristiques de classement de variables à instancier pour les CSP (SAGAUT, 1996) :

- la taille des domaines des variables qui consistent à instancier les variables ayant les plus petits domaines en premier.
- le nombre de contraintes dans lesquelles apparaît une variable : instancier en premier les variables qui apparaissent dans le plus de contraintes.

Il est possible de combiner ces deux types d'heuristiques.

DECHTER (1997) identifie des heuristiques dynamiques, qui permettent, à chaque nouvelle instanciation, de choisir la variable à valuer grâce à une phase de filtrage pour déterminer les variables ayant les plus petits domaines, ou reliées au maximum de contraintes.

5.3.4 Conclusion

Nous avons présenté une adaptation de l'algorithme *BackTrack* à notre implémentation. Cet algorithme tient compte de la structure en arbre syntaxique fournie par l'analyseur et nous permet de résoudre des CSP discrets ; des heuristiques de classement des variables à instancier peuvent ensuite être utilisées pour améliorer la résolution.

5.4 Le moteur de filtrage

5.4.1 Introduction

Nous voulons pouvoir fournir à l'utilisateur l'espace des solutions dans lequel il peut encore faire des choix. Pour montrer cet espace des solutions, nous proposons de fournir à l'utilisateur les projections de cet espace sur les domaines des variables. L'espace des choix possibles pour l'utilisateur est constitué de l'ensemble de ces projections. L'utilisateur a alors le choix de valuer une variable ou de réduire un domaine et le moteur fournira alors une nouvelle projection de l'espace des solutions sur les domaines des variables.

Ce mode de fonctionnement ne se limite pas aux CSP discrets : il doit nous permettre de filtrer aussi des CSP continus ou mixtes.

Le but du filtrage est d'arriver à maintenir un espace dans lequel il reste des choix cohérents pour l'utilisateur au niveau du CSP (la racine de l'arbre syntaxique, conjonction de toutes les contraintes), connaissant les domaines initiaux des variables. Pour ce faire, nous procédons par cohérence locale dans chacune des branches de l'arbre en retirant du domaine des variables les valeurs qui n'appartiennent plus à l'espace des solutions (ces valeurs sont localement incohérentes). L'espace des choix au niveau du CSP est une vision des domaines filtrés à travers toutes les contraintes.

Chaque nœud calcule les projections de son espace de choix connaissant les domaines des variables qu'il contraint. Chaque nœud ne peut déterminer que les projections de variables qu'il contraint (directement ou via ses enfants).

Nous voulons obtenir une approximation de l'espace des solutions de chaque nœud à la racine de l'arbre, soit un espace des valeurs possibles. Ainsi, nous utilisons la logique modale pour calculer un espace possiblement vrai pour chacun des nœuds, et ensuite le remonter à la racine de l'arbre. Pour obtenir son espace possiblement vrai, chaque nœud se base sur l'espace des solutions de ses fils et sur les domaines initiaux des variables.

5.4.2 Logique modale

La logique modale est dérivée de la logique propositionnelle classique par l'ajout de modalités, permettant de « nuancer » le sens des propositions. La logique modale est basée sur les travaux de KRIPKE (1959, 1963).

Il existe plusieurs types de logique modale, appelés « modes ». Ceux-ci sont les modes :

- classiques, dont les opérateurs sont : nécessaire, contingent, possible et impossible (mode utilisé par défaut) ;
- épistémiques (relatifs à la connaissance), dont les propositions sont : établi, contestable, exclu, plausible ;
- déontiques (moraux), dont les opérateurs sont : obligatoire, interdit, permis et facultatif.

Nous allons retenir le mode classique, en adoptant les notations classiques suivantes : nécessaire ($\Box$), contingent ($\neg\Box$), possible ($\Diamond$) et impossible ($\neg\Diamond$). Nous désignerons certains sous-ensembles des domaines (filtrés ou non) du CSP comme des espaces nécessairement vrai, possiblement vrai, possiblement faux ou nécessairement faux.

Nous définissons ainsi les espaces possiblement vrai et possiblement faux :

Définition 5.4 : *Espace possiblement vrai — $\Diamond V$*

Un espace possiblement vrai d'une formule logique est un ensemble de domaines (un par variable) pour lequel chaque affectation d'une et une seule variable de la formule à l'intérieur de ce domaine peut encore aboutir à une solution.

Définition 5.5 : *Espace possiblement faux — $\Diamond F$*

Un espace possiblement faux d'une formule logique est un ensemble de domaines (un par variable) pour lequel chaque affectation d'une et une seule variable de la formule à l'intérieur de ce domaine peut ne pas *aboutir à une solution.*

Pour calculer ces espaces possiblement vrais et faux, nous nous basons sur les domaines de définition des variables. Ces espaces sont donc calculés *sachant* D. Par rapport à une variable, l'espace possiblement vrai correspond à son domaine de définition duquel ont été ôtées toutes les valeurs qui *ne peuvent pas mener à une solution* ; c'est l'espace que nous voulons obtenir à la racine de l'arbre. L'espace possiblement faux correspond à son domaine de définition duquel ont été ôtées toutes les valeurs *menant obligatoirement à une solution* ; cet espace doit être calculé, il est utilisé principalement pour gérer la négation et l'élagage (cf. section 5.4.8 page 91).

Les définitions des espaces nécessaires sont les négations de celles-ci. En effet, un des axiomes de la logique modale stipule que le possible de A est la négation du nécessaire de non-A, et le nécessaire de A est la négation du possible de non-A (cf. équation 5.1).

$$\begin{aligned} \Diamond \mathrm{A} &= \overline{\Box \overline{A}} \\ \Box \mathrm{A} &= \overline{\Diamond \overline{A}} \end{aligned} \tag{5.1}$$

L'exemple 5.6 page suivante illustre le calcul des espaces sur un exemple très simple.

Remarque : sur cet exemple très simple, les espaces possiblement vrai et possiblement faux sont complémentaires. Ceci est dû à la simplicité de l'exemple ; dès que des formules logiques interviennent, les espaces possiblement vrais et possiblement faux ne sont plus complémentaires, et leur intersection est non-nulle.

EX. 5.6 — Calcul des espaces — un exemple très simple

Soit le CSP suivant :
– variable : X,
– domaine : $D = \{$ bleu , rouge , vert $\}$,
– contrainte : C : $X \in \{$ rouge , vert $\}$.
L'espace possiblement vrai est l'ensemble des domaines des variables auxquels on a ôté les valeurs *ne pouvant plus mener à une solution* :

$$\diamond\mathrm{V} = \{D \ : \{ \text{ rouge , vert } \}\}.$$

L'espace possiblement faux est l'ensemble des domaines des variables auxquels on a enlevé les valeurs *menant obligatoirement à une solution* :

$$\diamond\mathrm{F} = \{D \ : \{ \text{ bleu } \}\}.$$

5.4.3 Calcul des espaces pour les feuilles de comparaison symbolique

Dans le cas de feuilles de comparaison symbolique, l'espace possiblement vrai (respectivement faux) est égal à l'espace nécessairement vrai (respectivement faux). Dans ce cas, le calcul des différents espaces est simple et parfait. Les seules projections sont liées à la variable concernée.

Nous calculons la réduction du domaine D_X en prenant en compte seulement les valeurs dans la partie de l'égalité (ou d'un autre opérateur simple) ne contenant pas la variable. La réduction d'un domaine D sera notée D' (réduction au niveau du nœud père).

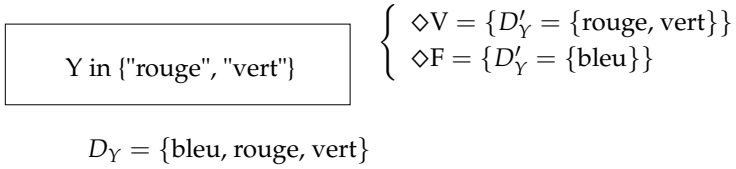


FIG. 5.7 — Espaces possiblement vrai et possiblement faux pour une feuille symbolique

Sur la figure 5.7, nous pouvons voir la réduction de domaine s'opérant dans le cas d'une feuille « appartenance » : $Y \in \{$ rouge, vert $\}$.

5.4.4 Calcul des espaces pour les feuilles de comparaison numérique

Les feuilles de comparaison numérique seront présentées « éclatées » en arbres numériques dans les figures. Nous divisons ce type de feuilles en deux sous-types : les feuilles de comparaison numérique directes et indirectes.

Feuilles de comparaison numérique directes

Une feuille de comparaison numérique directe peut représenter toutes les contraintes de la forme : $x \odot f(y_1, ..., y_n)$ ($\odot$ représentant un opérateur de comparaison numérique). L'arbre numérique correspondant à une telle contrainte est donc de la forme suivante :

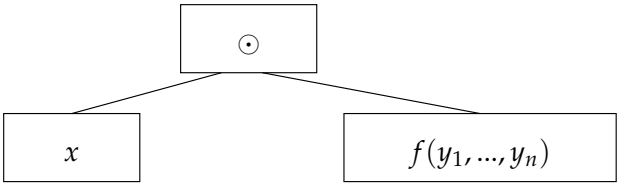


FIG. 5.8 — Forme générale d'un arbre numérique représentant une feuille de comparaison numérique directe

La figure 5.9 illustre une feuille de comparaison numérique directe, dont une branche comporte une seule variable. L'équation correspondante à la contrainte illustrée est la suivante :

$$a = b^2 - 2 \times c.$$

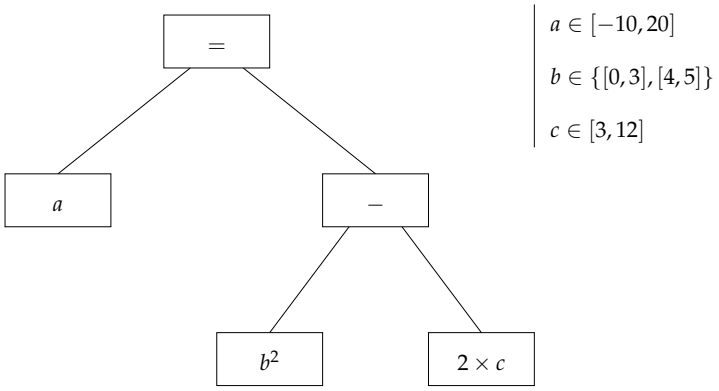


FIG. 5.9 — Feuille de comparaison numérique directe

Dans ces cas-là, nous pouvons filtrer le domaine de la variable a (variable directe). Nous utilisons alors les domaines des variables non-directes (b et c) pour réduire le domaine de a. En utilisant une méthode de filtrage basée sur l'arithmétique des intervalles, nous obtenons l'espace possiblement vrai pour le nœud « égal » (=). Cet espace est exprimé dans l'équation 5.2.

$$\Diamond\mathrm{V}(=) \quad \left\{ \begin{array}{ll} D'_a = [-10, 19] & = (D_b^{②} \ominus 2 \otimes D_c) \cap D_a \\ & = (([0,3],[4,5])^{②} \ominus 2 \otimes [3,12]) \cap [-10,20] \\ & = ([-24,3] \cup [-8,19]) \cap [-10,20] \\ D'_b = \{[0,3],[4,5]\} & \\ D'_c = [3,12] & \end{array} \right\} \tag{5.2}$$

Feuilles de comparaison numérique indirectes

Une feuille de comparaison numérique indirecte représente toutes les formules où l'on ne peut isoler une variable, de la forme : $f(x_1, ..., x_n) \odot g(x_1, ..., x_m)$. Elles sont représentées sous la forme d'un arbre numérique, comme l'illustre la figure 5.10 page suivante.

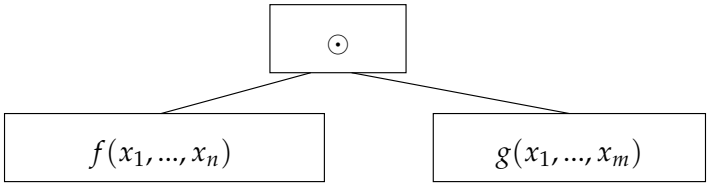


FIG. 5.10 — Forme générale d'un arbre numérique représentant une feuille de comparaison numérique indirecte

Dans le cas illustré par la figure 5.11, nous ne pouvons plus filtrer une variable en fonction des autres. La formule est indirecte, et donc plus complexe à filtrer.

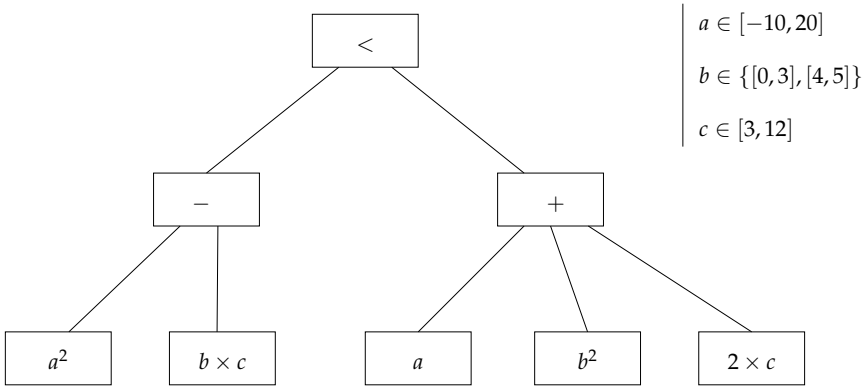


FIG. 5.11 — Feuille de comparaison numérique indirecte

Des pistes ont été proposées pour filtrer ce type de contraintes (BENHAMOU *et al.*, 1999; BATNINI, 2005). Actuellement, notre implémentation de feuilles de comparaison numérique indirectes ne fait que vérifier que les intervalles des différentes variables soient compatibles (le nœud n'est pas nécessairement faux).

Remarque : le modeleur doit attacher une importance toute particulière à la formulation des contraintes. En effet, pour obtenir un filtrage sur les trois variables, il peut alors utiliser des contraintes redondantes, et exprimer chacune des variables en fonction des autres (lorsque c'est possible).

Nous sommes conscients que le filtrage numérique peut provoquer des problèmes de stabilité numérique (erreurs de calculs et convergence). Ce point est détaillé dans la section 5.4.9 page 96.

5.4.5 Calcul des espaces pour les nœuds logiques

Lorsque les espaces possiblement vrais et possiblement faux ont été déterminés au niveau de chacune des feuilles de comparaison (symboliques et numériques), ils sont agrégés au niveau des nœuds de composition logique jusqu'à la racine de l'arbre afin de vérifier la cohérence globale du modèle. L'opération d'agrégation des espaces pour chacun des nœuds de composition logique est spécifique à ce nœud.

Dans la suite de ce chapitre, le CSP sera noté (X, D, C), n sera le nœud considéré et ses fils seront nommés f_i. Pour une variable x, nous adopterons les notations suivantes : D_x est

son domaine avant le filtrage au niveau des feuilles de l'arbre, D'_x est son domaine réduit après filtrage au niveau de la feuille, D''_x est son domaine réduit après un premier filtrage au niveau logique et D'''_x son domaine réduit après un deuxième filtrage au niveau logique.

Postulat 3 *L'approximation des espaces d'un nœud peut s'obtenir avec seulement les espaces de ses nœuds fils.*

Opérations assemblistes sur les espaces

Il faut noter que cette opération dépend des variables apparaissant dans chacune des branches du nœud. Nous devons alors distinguer deux cas pour effectuer les opérations assemblistes sur les espaces de projections :

- les branches du nœud comportent des variables différentes ;
- les branches du nœud comportent toutes les mêmes variables.

Les branches du nœud comportent des variables différentes

Lorsque les branches comportent des variables différentes, il faut alors calculer la projection des variables n'apparaissant que sur une branche sur les autres branches du nœud, connaissant les projections sur lesquelles cette variable apparaît. Nous appelons ces projections de l'espace possiblement vrai sur une variable non-contrainte des espaces possiblement vrais (ou possiblement faux) *virtuels* (cf. figure 5.12). Nous distinguons alors deux cas selon la cohérence locale du problème.

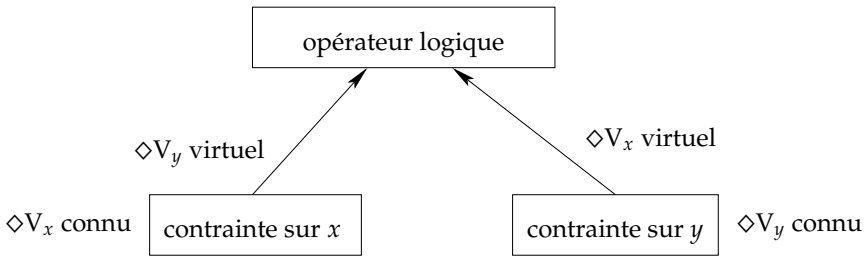


FIG. 5.12 — Utilisation d'espaces possiblement vrais virtuels

Lorsque le problème est localement cohérent, les projections de cette variable dans les branches où elle apparaît sont non-vides ; le domaine de la projection de cette variable sur les branches où elle n'apparaît pas sera alors son domaine initial. Sur la figure 5.13 page suivante, l'espace possiblement vrai relatif à Y sur le fils `X in {"0", "1"}` correspond au domaine initial D_Y, car ce nœud est localement cohérent (il existe au moins une valeur de X pour laquelle l'appartenance est vraie) : $\diamond V_Y \text{virtuel} = D_Y$.

Quand une des projections de cette variable dans les branches où elle apparaît est vide, le problème est alors (localement) incohérent. La projection de cette variable dans les espaces des autres branches est alors l'ensemble vide : $\diamond V_Y \text{virtuel} = \emptyset$. Ainsi, quelle que soit l'opération effectuée, le problème sera toujours incohérent au niveau du nœud étudié[1]. Sur la figure 5.14 page ci-contre, l'espace possiblement vrai relatif à Y sur le fils `X in {"0", "1"}` correspond à l'ensemble vide $\emptyset$ car ce nœud est localement incohérent (il n'existe aucune valeur de X permettant de vérifier l'appartenance).

[1]Il faut noter que cette incohérence est locale ; la présence de certains nœuds (`not`, par exemple) au niveau supérieur peut inverser la tendance.

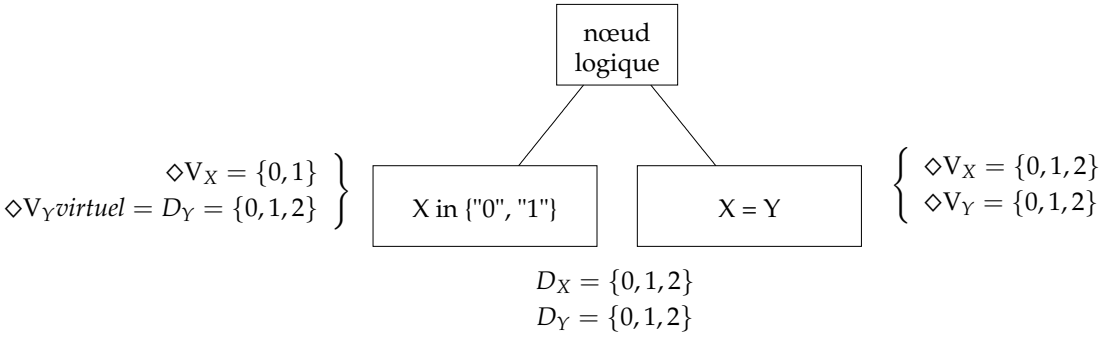


FIG. 5.13 — Utilisation d'espaces possiblement vrais virtuels — le problème est localement cohérent

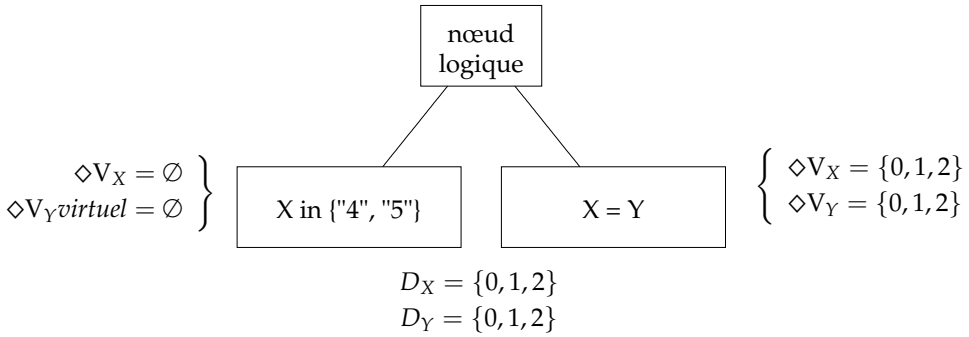


FIG. 5.14 — Utilisation d'espaces possiblement vrais virtuels — le problème est localement incohérent (branche à gauche)

Pour résumer, afin de pouvoir calculer l'espace possiblement vrai au niveau de l'opérateur logique, il faut utiliser les espaces possiblement vrais relatifs à chacune des variables pour les nœuds fils. Ainsi, nous créons un espace possiblement vrai virtuel pour chacune des variables sur le fils qui ne la contraint pas. Cet espace virtuel dépend de la cohérence locale du fils auquel il est rattaché :

- dans le cas d'un fils localement cohérent, cet espace virtuel relatif à une variable non contrainte par le fils correspond au domaine initial de cette variable ;
- dans le cas d'un fils localement incohérent (l'espace possiblement vrai de l'une des variables qu'il contraint est alors vide), cet espace virtuel relatif à une variable non contrainte par le fils est vide aussi (pour ne pas changer artificiellement la cohérence du fils).

Les branches du nœud comportent toutes les mêmes variables

Lorsque les nœuds fils ont tous les mêmes variables, il n'y a aucune adaptation à effectuer.

L'équation 5.3 résume le calcul des espaces possiblement vrais virtuels d'un fils f par rapport à une variable x_j ($\Diamond V_{x_j}(f)$).

$$\Diamond V_{x_j}(f) = \left\| \begin{array}{l} D'_{x_j} \text{ si } x_j \in f \\ D_{x_j} \text{ si } x_j \notin f \text{ et } \forall k / x_k \in f, D'_{x_k} \neq \emptyset \\ \emptyset \text{ si } x_j \notin f \text{ et } \exists k / x_k \in f, D'_{x_k} = \emptyset \end{array} \right. \tag{5.3}$$

Les deux opérateurs ⋒ et ⋓ notent les opérations d'intersection et d'union sur les espaces. Ces deux opérateurs dénotent le fait que les deux espaces à agréger doivent comporter les mêmes variables. Les espaces doivent donc être augmentés d'espaces virtuels par rapport aux variables manquantes (par rapport au père). Chacun des nœuds de composition logique possède un mode d'agrégation spécifique des espaces possiblement vrais et possiblement faux de ses fils ; les sections suivantes détaillent ces opérations.

Nœuds not

Pour un nœud not, les espaces possiblement vrais et possiblement faux s'interchangent. En effet, ce qui est nécessairement vrai devient nécessairement faux et inversement. D'après le théorème 5.1 page 78, il faut donc complémenter les espaces possiblement vrais et faux pour obtenir les espaces possiblement faux et vrais. La figure 5.15 illustre ce fonctionnement.

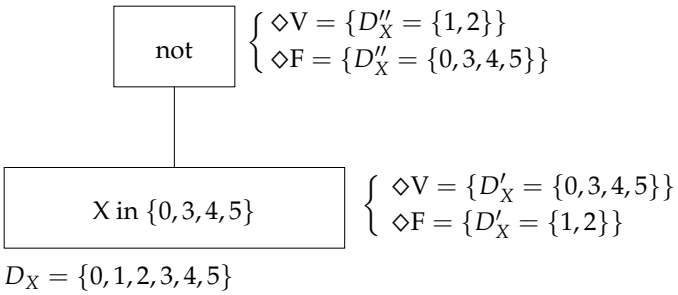


FIG. 5.15 — Espaces possiblement vrai et possiblement faux pour un nœud not

Nœuds and

Ce type de nœud est plus complexe ; il nécessite d'effectuer des opérations assemblistes sur les espaces possiblement vrais ou faux des nœuds fils. Les opérations définies pour obtenir les espaces possiblement vrais et faux à partir des espaces des nœuds fils sont détaillées sur les équations 5.4 et 5.5 page ci-contre. Pour déterminer ces espaces, nous avons utilisé les opérations assemblistes sur les espaces dont les principes sont détaillés plus haut.

L'espace possiblement vrai d'un nœud and correspond à l'ensemble des espaces possiblement vrais relatifs à chacune de ses variables : $\Diamond\mathrm{V}(n) = \{\Diamond\mathrm{V}_x(n)\}_{\forall x \in X}$ (avec X l'ensemble des variables du nœud n).

Pour obtenir l'espace possiblement vrai d'un nœud and, il faut intersecter les domaines de l'espace possiblement vrai de chacun de ses fils pour chacune de ses variables ($\Diamond\mathrm{V}_x(n) = \bigcap_{\forall i}(\Diamond\mathrm{V}_x(f_i))$). Si une variable (donc un domaine) n'apparait pas dans l'un des fils, nous prenons son domaine initial si le problème peut encore être cohérent (si $\forall j$ tel que $x_j \in f_i$, $D'_{x_j} \neq \emptyset$) et un domaine vide s'il n'existe plus de solutions au problème ($\emptyset$ si l'un des domaines de l'espace étudié devient l'ensemble vide). Si la variable apparaît dans le fils, nous utilisons alors l'espace filtré remonté depuis les feuilles de l'arbre ($\Diamond\mathrm{V}_{x_j}(f_i) = D'_{x_j}$ si $x_j \in f_i$).

Ainsi, lorsqu'un nœud comporte une branche dans laquelle une variable n'apparaît pas, le domaine utilisé pour l'intersection de la projection de cette variable est son domaine initial si son domaine filtré dans les autres branches n'est pas vide. Si son domaine filtré dans les autres branches devient vide, l'ensemble vide est alors utilisé pour sa projection dans les branches où cette variable n'apparaît pas.

$$\Diamond \mathrm{V}(\mathit{and}) \quad \begin{cases} \Diamond \mathrm{V}/D & = \{\Diamond \mathrm{V}_x(n)\}_{\forall x \in X} \\ \Diamond \mathrm{V}_x(n) & = \bigcap_{\forall i}(\Diamond \mathrm{V}_x(f_i)) \\ \Diamond \mathrm{V}_{x_j}/D(f_i) = \begin{cases} \Diamond \mathrm{V}_{x_j}(f_i) = D'_{x_j} \text{ si } x_j \in f_i \\ \Diamond \mathrm{V}_{x_j}(f_i) = \left\| \begin{array}{ll} D_{x_j} & \text{si } \forall k / x_k \in f_i, D'_{x_k} \neq \emptyset \\ \emptyset & \text{sinon} \end{array} \right. \end{cases} \Big/ \; x_j \notin f_i \end{cases} \tag{5.4}$$

Les opérations pour obtenir l'espace possiblement faux d'un nœud sont similaires, à la différence que l'on fait l'union des espaces possiblement faux de ses fils ($\Diamond \mathrm{F}_x(n) = \bigcup_{\forall i}(\Diamond \mathrm{F}_x(f_i))$).

$$\Diamond \mathrm{F}(\mathit{and}) \quad \begin{cases} \Diamond \mathrm{F}/D & = \{\Diamond \mathrm{F}_x(n)\}_{\forall x \in X} \\ \Diamond \mathrm{F}_x(n) & = \bigcup_{\forall i}(\Diamond \mathrm{F}_x(f_i)) \\ \Diamond \mathrm{F}_{x_j}/D(f_i) = \begin{cases} \Diamond \mathrm{F}_{x_j}(f_i) = D'_{x_j} \text{ si } x_j \in f_i \\ \Diamond \mathrm{F}_{x_j}(f_i) = \left\| \begin{array}{ll} D_{x_j} & \text{si } \forall k / x_k \in f_i, D'_{x_k} \neq \emptyset \\ \emptyset & \text{sinon} \end{array} \right. \end{cases} \Big/ \; x_j \notin f_i \end{cases} \tag{5.5}$$

Nous illustrons dans un premier temps (cf. figure 5.16) les réductions de domaine sur un nœud and dont toutes les branches ont les mêmes variables, puis sur un nœud dont les branches ont des variables différentes (cf. figure 5.17).

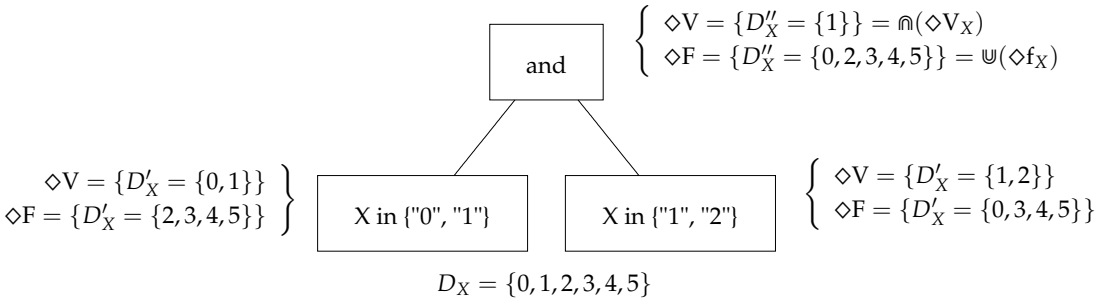


FIG. 5.16 — Espaces possiblement vrais et faux pour un nœud and à une seule variable

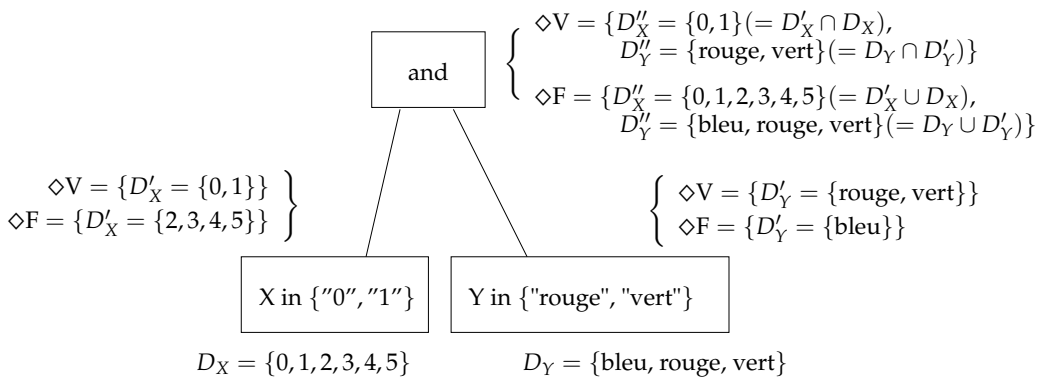


FIG. 5.17 — Espaces possiblement vrais et faux pour un nœud and à deux variables

Nœuds or

Dans le cas d'un nœud or, les opérations à effectuer pour calculer les espaces possiblement vrais et faux sont les symétriques du nœud and. Ainsi, les équations 5.6 et 5.7 résument les opérations à effectuer.

Pour calculer l'espace possiblement vrai, nous faisons donc l'union des domaines de l'espace possiblement vrai ($\diamond \mathrm{V}_x(n) = \bigcup_{\forall f}(\diamond \mathrm{V}_x(f))$, cf. équation 5.6). Nous utilisons aussi les espaces virtuels si l'une des variables n'apparaît pas dans l'un des fils du nœud or.

$$\diamond \mathrm{V}(or) \quad \begin{cases} \diamond \mathrm{V}/D & = \{\diamond \mathrm{V}_x(n)\}_{\forall x \in X} \\ \diamond \mathrm{V}_x(n) & = \bigcup_{\forall i}(\diamond \mathrm{V}_x(f_i)) \\ \diamond \mathrm{V}_{x_j}/D(f_i) = & \begin{cases} \diamond \mathrm{V}_{x_j}(f_i) = D'_{x_j} \text{ si } x_j \in f_i \\ \diamond \mathrm{V}_{x_j}(f_i) = \left\| \begin{array}{ll} D_{x_j} & \text{si } \forall k / x_k \in f_i, D'_{x_k} \neq \emptyset \\ \emptyset & \text{sinon} \end{array} \right. \Big/ x_j \notin f_i \end{cases} \end{cases} \tag{5.6}$$

L'équation 5.7 montre les opérations de calcul de l'espace possiblement faux pour un nœud or. Nous effectuons ici l'intersection ($\diamond \mathrm{F}_x(n) = \bigcap_{\forall f}(\diamond \mathrm{F}_x(f))$) des domaines des espaces possiblement faux des nœuds fils.

$$\diamond \mathrm{F}(or) \quad \begin{cases} \diamond \mathrm{F}/D & = \{\diamond \mathrm{F}_x(n)\}_{\forall x \in X} \\ \diamond \mathrm{F}_x(n) & = \bigcap_{\forall i}(\diamond \mathrm{F}_x(f_i)) \\ \diamond \mathrm{F}_{x_j}/D(f_i) = & \begin{cases} \diamond \mathrm{F}_{x_j}(f_i) = D'_{x_j} \text{ si } x_j \in f_i \\ \diamond \mathrm{F}_{x_j}(f_i) = \left\| \begin{array}{ll} D_{x_j} & \text{si } \forall k / x_k \in f_i, D'_{x_k} \neq \emptyset \\ \emptyset & \text{sinon} \end{array} \right. \Big/ x_j \notin f_i \end{cases} \end{cases} \tag{5.7}$$

La figure 5.18 illustre l'obtention des espaces possiblement vrais et faux dans le cas d'un nœud or.

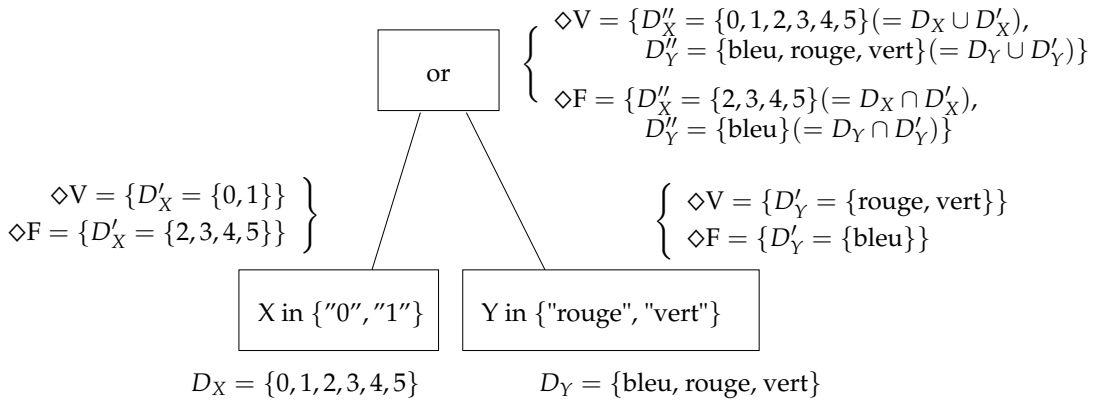


FIG. 5.18 — Espaces possiblement vrais et faux pour un nœud or

Nœuds => et <=>

Ces deux types de nœuds sont en fait des combinaisons de not, and et or. Cependant, ce type de nœud possède seulement deux fils : f_1 et f_2. L'équation 5.8 page ci-contre montre les équivalences logiques entre ces deux opérateurs et les opérateurs logiques « de base ».

Soient A et B deux propositions :

$$A \Rightarrow B \equiv \neg A \vee B \tag{5.8}$$
$$A \Leftrightarrow B \equiv (A \Rightarrow B) \wedge (B \Rightarrow A) \equiv (\neg A \vee B) \wedge (\neg B \vee A)$$

Nous pouvons donc établir le calcul des espaces possiblement vrais et possiblement faux pour chacun de ces opérateurs. L'équation 5.9 montre les formules de calcul pour le nœud « implication » (=>). Pour une implication, nous devons noter que l'ordre des fils de n est important ; l'équation suivante considère donc l'implication suivante : $n \equiv f_1 \Rightarrow f_2$.

$$\text{=>} \begin{cases} \Diamond \mathrm{V}(n) = \Diamond \mathrm{F}(f_1) \Cup \Diamond \mathrm{V}(f_2) \\ \\ \Diamond \mathrm{F}(n) = \Diamond \mathrm{V}(f_1) \Cap \Diamond \mathrm{F}(f_2) \end{cases} \tag{5.9}$$

Remarque : pour ces deux nœuds, nous utilisons les opérations assemblistes de la même façon que précédemment : lorsqu'une variable n'apparaît pas dans une branche, le domaine que l'on considère est soit son domaine initial (si le problème est localement cohérent), soit l'ensemble vide (si le problème est localement incohérent). Dans les équations 5.9 et 5.10, les espaces ne comprenant pas une variable sont donc augmentés des espaces virtuels.

La figure 5.19 illustre le calcul des espaces possiblement vrais et faux pour l'opérateur => sur un exemple.

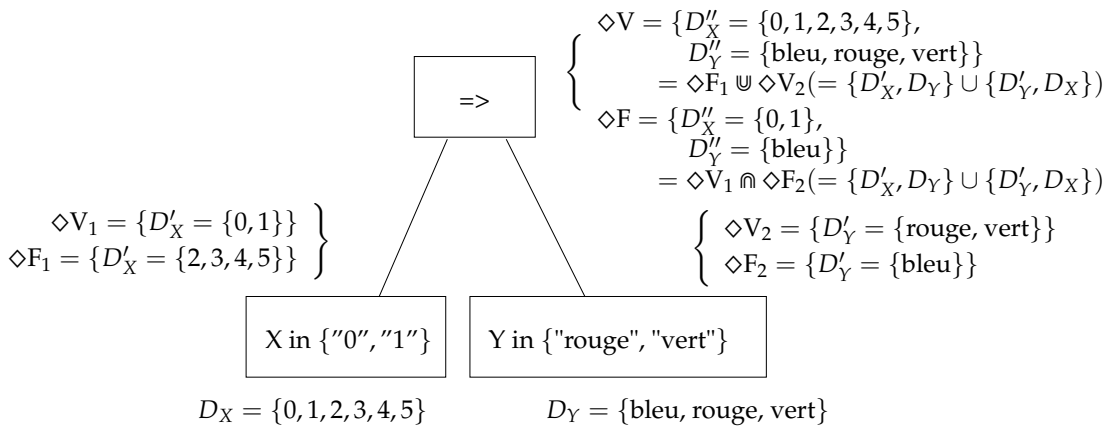


FIG. 5.19 — Espaces possiblement vrais et faux pour un nœud =>

L'équation 5.10 montre les formules de calcul pour le nœud « équivalence » (<=>). Dans le cas d'une équivalence, l'ordre des fils n'est pas important (ce n'est pas le premier qui implique le deuxième, ils sont équivalents).

$$\text{<=>} \begin{cases} \Diamond \mathrm{V}(n) = \left(\Diamond \mathrm{F}(f_1) \Cap \Diamond \mathrm{F}(f_2)\right) \Cup \left(\Diamond \mathrm{V}(f_1) \Cap \Diamond \mathrm{V}(f_2)\right) \\ \\ \Diamond \mathrm{F}(n) = \left(\Diamond \mathrm{F}(f_1) \Cup \Diamond \mathrm{F}(f_2)\right) \Cap \left(\Diamond \mathrm{V}(f_1) \Cup \Diamond \mathrm{V}(f_2)\right) \end{cases} \tag{5.10}$$

La figure 5.20 page suivante illustre le calcul des espaces possiblement vrais et faux pour l'opérateur <=> sur un exemple.

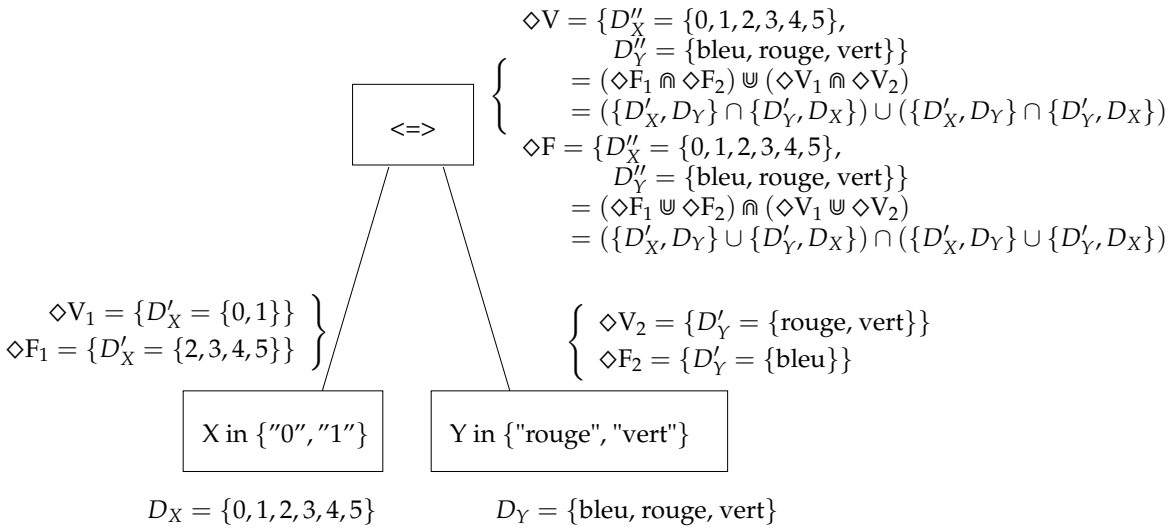


FIG. 5.20 — Espaces possiblement vrais et faux pour un nœud <=>

5.4.6 Filtrage

Le principe du filtrage est le suivant : les espaces possiblement vrais et faux sont calculés depuis les feuilles jusqu'à la racine de l'arbre. Il faut alors réinjecter les domaines réduits aux feuilles de l'arbre et remonter les espaces jusqu'à stabilisation de l'espace possiblement vrai à la racine de l'arbre. Une fois l'espace possiblement vrai stabilisé à la racine, l'utilisateur peut effectuer des choix à l'intérieur de cet espace.

Pour le fonctionnement de notre raisonnement en arbres, nous nous proposons de revérifier à chaque réduction de domaine seulement les nœuds dont une des variables a vu son domaine réduit (pour la cohérence d'arc, AC-3 proposait cette amélioration par rapport à AC-1).

Lors de la première itération, les domaines sont injectés au niveau des feuilles de l'arbre et les domaines possiblement vrais et faux remontés jusqu'à la racine de l'arbre. À la racine de l'arbre, nous obtenons ainsi une cohérence globale, qui correspond à l'agrégation de toutes les cohérences locales aux nœuds.

À partir de la deuxième itération, l'espace possiblement vrai est considéré comme l'agrégation des domaines réduits des variables. Nous réinjectons dans chacun des fils les domaines filtrés (espaces possiblement vrais) des variables qui le concernent. Ainsi, à chaque passe de propagation, l'espace possiblement vrai global de la passe précédente est réinjecté, et chacun des nœuds utilise seulement les domaines le concernant (par exemple, un nœud contraignant la variable x n'utilisera que l'espace possiblement vrai global relatif à x).

La figure 5.21 page suivante illustre le passage entre deux passes de propagation. L'espace global possiblement vrai de la phase $n-1$ ($\diamond V^{n-1}$) est réinjecté lors de la phase n. Les nœuds considèrent alors que les domaines des variables sont issus de cet espace possiblement vrai, et non plus des domaines initiaux des variables.

Pour chaque nœud, nous revérifions la cohérence de ses fils avec les domaines réduits si et seulement si les nœuds fils portent sur des variables dont les domaines ont été réduits. Chaque nœud étant capable de remarquer un changement dans ses espaces possiblement vrais d'entrée, nous évitons donc les vérifications pour des variables dont le domaine est inchangé.

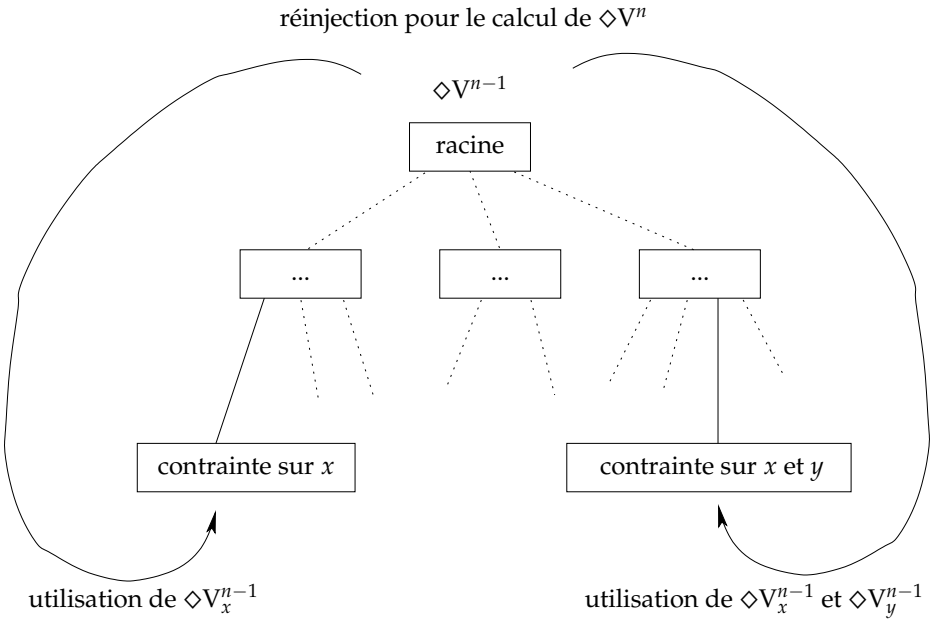


FIG. 5.21 — Réinjection des domaines de l'espace $\diamond V^{n-1}$ pour le filtrage de la phase n

Nous pouvons noter que d'autres travaux en programmation logique et par contraintes proposent des mécanismes qui s'apparentent à ceux proposés ici, en particulier RATSCHAN (2002).

5.4.7 Un exemple de filtrage

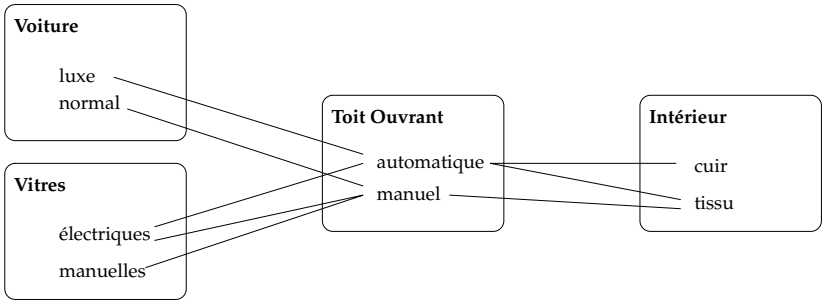


FIG. 5.22 — Exemple simple pour l'illustration du filtrage

Reprenons l'exemple 2.1 page 22, illustré par la figure 5.22. Soient les variables V_1 (*Voiture*), V_2 (*Vitres*), V_3 (*Intérieur*) et V_4 (*Toit Ouvrant*), dont les domaines sont D_1 (normal, luxe), D_2 (électriques, manuelles), D_3 (tissu, cuir) et D_4 (automatique, manuel). Les contraintes sont les suivantes :

- C_1 : $V_1 = \text{luxe} \Leftrightarrow V_4 = \text{automatique}$;
- C_2 : $V_4 = \text{automatique} \Rightarrow V_2 = \text{électriques}$;
- C_3 : $V_3 = \text{cuir} \Rightarrow V_4 = \text{automatique}$.

Afin d'illustrer le filtrage, nous ne nous intéressons qu'à l'espace possiblement vrai de chaque nœud. En faisant abstraction des définitions de domaine dans l'arbre syntaxique,

celui-ci est tel que l'illustre la figure 5.23 (les domaines des variables ont été reportés sous les feuilles). Nous supposons que l'utilisateur a choisi un intérieur cuir ; le domaine de V_3 se réduit donc à cuir dans $\diamond V^1$, l'espace possiblement vrai après la première phase de propagation.

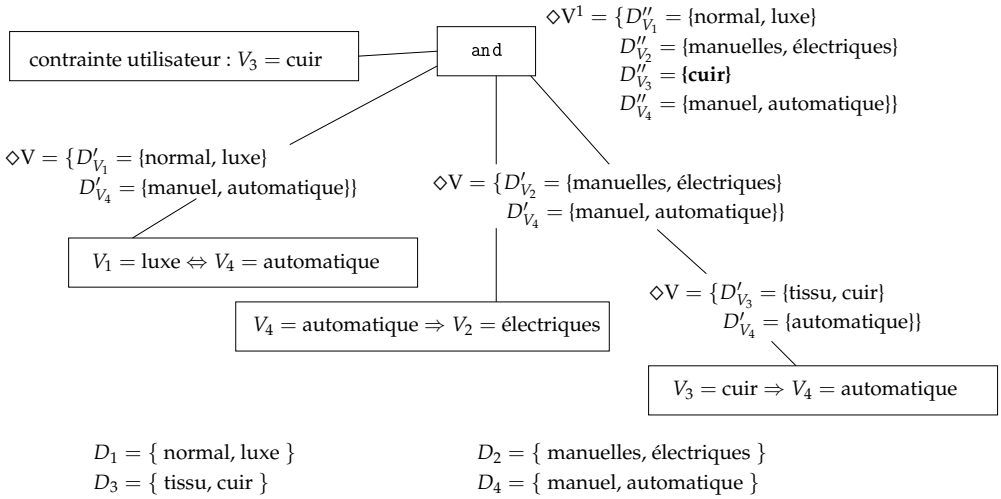


FIG. 5.23 — Arbre syntaxique et espaces possiblement vrais pour un exemple simple — le domaine de V_3 est réduit à cuir

Lors de la première phase de propagation, les espaces possiblement vrais et faux sont remontés depuis les feuilles de l'arbre jusqu'à la racine, en utilisant les opérations décrites dans les sections précédentes. À la racine, après une phase de propagation, nous obtenons donc l'espace possiblement vrai :

$$\diamond V^1 = \{ \quad D''_{V_1} = \{ \text{normal, luxe} \},\ D''_{V_2} = \{ \text{manuelles, électriques} \} \\ D''_{V_3} = \{ \text{cuir} \},\ D''_{V_4} = \{ \text{manuel, automatique} \} \} \tag{5.11}$$

Cet espace est alors réinjecté aux feuilles de l'arbre ; lorsque les domaines filtrés (contenus dans $\diamond V^1$) sont différents de l'espace possiblement vrai relatif à une variable d'une feuille de la phase précédente, ces feuilles sont filtrées et nous remontons ainsi jusqu'à la racine de l'arbre, comme l'illustre la figure 5.24 page suivante.

À la racine, après une nouvelle phase de propagation, nous obtenons donc l'espace possiblement vrai :

$$\diamond V^2 = \{ \quad D''_{V_1} = \{ \text{normal, luxe} \},\ D''_{V_2} = \{ \text{manuelles, électriques} \} \\ D''_{V_3} = \{ \text{cuir} \},\ D''_{V_4} = \{ \text{automatique} \} \} \tag{5.12}$$

Nous réinjectons alors l'espace possiblement vrai obtenu après la deuxième phase de propagation ($\diamond V^2$), étant donné qu'un domaine supplémentaire a été réduit. Les valeurs des variables seront donc prises dans $\diamond V^2$. Nous obtenons ainsi l'arbre syntaxique et les espaces possiblement vrais de la figure 5.25 page 92.

À la racine de l'arbre, nous obtenons l'espace possiblement vrai suivant :

$$\diamond V^3 = \{ \quad D''_{V_1} = \{ \text{luxe} \},\ D''_{V_2} = \{ \text{électriques} \} \\ D''_{V_3} = \{ \text{cuir} \},\ D''_{V_4} = \{ \text{automatique} \} \} \tag{5.13}$$

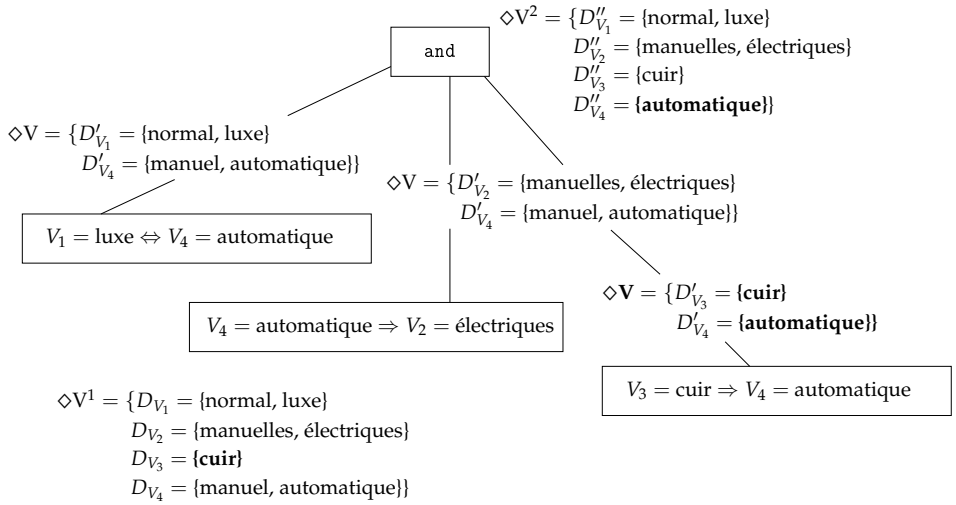


FIG. 5.24 — Arbre syntaxique et espaces possiblement vrais pour un exemple simple — le domaine de V_4 est réduit à automatique

Dans l'espace possiblement vrai, chacune des variables est valuée (il existe une seule valeur disponible). Nous venons donc d'atteindre une solution du problème.

Sur cet exemple simple, si l'utilisateur choisit un intérieur cuir, une solution est atteinte après trois phases de propagation. Elle correspond à $\diamond V^3$, soit une voiture de luxe avec vitres électriques, intérieur cuir et toit ouvrant automatique.

5.4.8 Élagage de l'arbre au cours du filtrage

Nous définissons la notion de branche vide ainsi :

Définition 5.6 : *Branche vide*

Une branche vide est une branche dont au moins l'un des deux espaces (possiblement vrai ou possiblement faux) est vide.

L'élagage peut se produire lorsque les domaines sont suffisament filtrés pour que certaines branches aient l'un des espaces possiblement vrais ou possiblement faux vides. Selon les nœuds concernés, il est alors possible de conclure que la branche en question est toujours vraie ou toujours fausse (sauf relaxation de contraintes), quelles que soient les valeurs adoptées pour chacune des variables.

La première étape consiste donc à identifier les nœuds dont les espaces possiblement vrais ou faux sont vides. Une fois cette identification effectuée, nous pouvons alors remonter les espaces jusqu'à un nœud où ils pourront être utiles pour couper certaines branches de l'arbre, et ainsi éviter le calcul d'espaces inutiles.

Nous introduisons les nœuds ***Vrai*** et ***Faux*** ; un nœud ***Vrai*** est un nœud dont l'espace possiblement faux est vide, qui est donc nécessairement vrai. Un nœud ***Faux*** est un nœud dont l'espace possiblement vrai est vide, qui est donc nécessairement faux. La suite de cette section présente les différentes opérations d'élagage pour chacun des opérateurs logiques (dans les formules, l'opérateur $\Rrightarrow$ représente l'opération d'élagage).

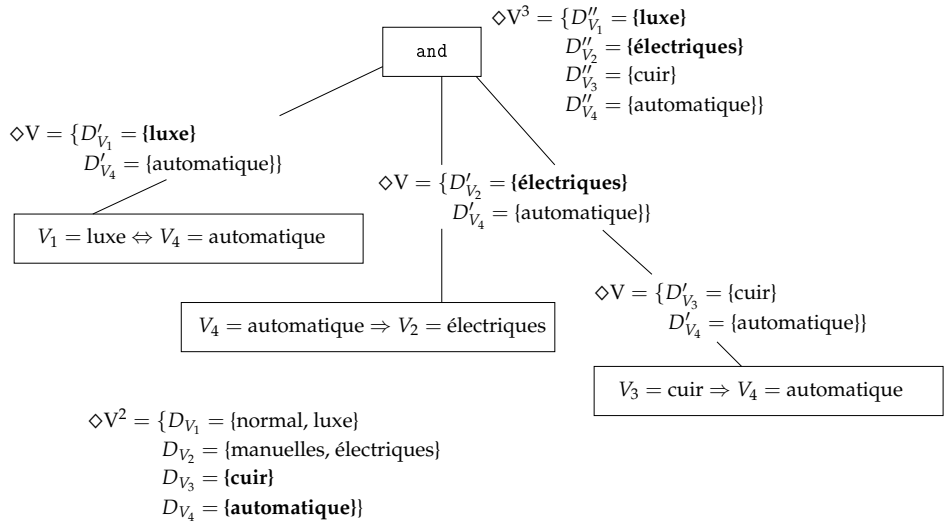


FIG. 5.25 — Arbre syntaxique et espaces possiblement vrais pour un exemple simple — les domaines D_3 et D_4 sont réduits

Élagage des feuilles

Pour les feuilles de comparaison (symboliques ou numériques), l'opération d'élagage est triviale. Dans le cas où la comparaison est toujours vraie ou toujours fausse, on remplace lors de l'élagage la feuille représentant l'opérateur de comparaison par un nœud ***Vrai*** ou un nœud ***Faux***.

Élagage d'un nœud and

Si un des fils d'un nœud and a son espace possiblement vrai vide, ce nœud est alors considéré comme un nœud ***Faux***. L'espace possiblement vrai du nœud concerné est alors toujours vide ; en effet, l'intersection d'un ensemble quelconque avec l'ensemble vide est vide. Ainsi, le nœud est toujours faux.

$$\begin{aligned}
&\text{Soient } A_1, ..., A_{n-1} \text{ les } n-1 \text{ premiers fils d'un opérateur and et } \boldsymbol{Faux} \text{ le } n\text{-ième} \\
&\quad \Diamond\mathrm{V}(\mathsf{and}) = \Diamond\mathrm{V}(\boldsymbol{Faux}) \cap \bigcap_{i=1...n-1} \Diamond\mathrm{V}(A_i) \\
&\qquad\qquad = \emptyset \cap \bigcap_{i=1...n-1} \Diamond\mathrm{V}(A_i) \\
&\qquad\qquad = \emptyset \\
&\Longrightarrow A_1 \wedge ... \wedge A_{n-1} \wedge \boldsymbol{Faux} \Rrightarrow \boldsymbol{Faux}
\end{aligned} \tag{5.14}$$

Si un des fils d'un nœud and est toujours vrai (son espace possiblement faux est vide), nous ne le considérons plus. En effet, l'intersection d'un ensemble quelconque avec l'espace entier sur lequel il est défini est égale à cet ensemble. Ainsi, le nœud considéré perd un fils. Si tous ses fils sont ***Vrai*** (et donc éliminés par élagage), le nœud considéré est alors ***Vrai***.

$$\begin{aligned}
&\text{Soient } A_1, ..., A_{n-1} \text{ les } n-1 \text{ premiers fils d'un opérateur and et } \boldsymbol{Vrai} \text{ le } n\text{-ième} \\
&\quad \Diamond\mathrm{V}(\mathsf{and}) = \Diamond\mathrm{V}(\boldsymbol{Vrai}) \cap \bigcap_{i=1...n-1} \Diamond\mathrm{V}(A_i) \\
&\qquad\qquad = \neg\emptyset \cap \bigcap_{i=1...n-1} \Diamond\mathrm{V}(A_i) \\
&\qquad\qquad = \bigcap_{i=1...n-1} \Diamond\mathrm{V}(A_i) \\
&\Longrightarrow A_1 \wedge ... \wedge A_{n-1} \wedge \boldsymbol{Vrai} \Rrightarrow A_1 \wedge ... \wedge A_{n-1}
\end{aligned} \tag{5.15}$$

Élagage d'un nœud or

Si un des fils d'un nœud or a son espace possiblement faux vide, l'espace possiblement faux du nœud concerné est alors logiquement toujours vide. En effet, l'union d'un ensemble quelconque avec l'espace entier sur lequel il est défini est égale à l'espace entier. Ainsi, le nœud or a son espace possiblement faux vide et est toujours vrai.

$$\begin{aligned}
&\text{Soient } A_1, ..., A_{n-1} \text{ les } n-1 \text{ premiers fils d'un opérateur and et } \boldsymbol{Vrai} \text{ le } n\text{-ième} \\
&\quad \Diamond\mathrm{V}(\mathsf{and}) = \Diamond\mathrm{V}(\boldsymbol{Vrai}) \cup \bigcup_{i=1...n-1} \Diamond\mathrm{V}(A_i) \\
&\quad = \neg\emptyset \cup \bigcup_{i=1...n-1} \Diamond\mathrm{V}(A_i) \\
&\quad = \neg\emptyset \\
&\Longrightarrow A_1 \vee ... \vee A_{n-1} \vee \boldsymbol{Vrai} \Rrightarrow \boldsymbol{Vrai}
\end{aligned} \tag{5.16}$$

Si un des fils d'un nœud or est toujours faux (son espace possiblement vrai est vide), nous ne le considérons plus. En effet, l'union d'un ensemble quelconque avec l'ensemble vide est égale à cet ensemble quelconque. Ainsi, le nœud considéré perd un fils. Si tous ses fils sont ***Faux*** (et donc éliminés par élagage), le nœud considéré est alors ***Faux***.

$$\begin{aligned}
&\text{Soient } A_1, ..., A_{n-1} \text{ les } n-1 \text{ premiers fils d'un opérateur and et } \boldsymbol{Faux} \text{ le } n\text{-ième} \\
&\quad \Diamond\mathrm{V}(\mathsf{and}) = \Diamond\mathrm{V}(\boldsymbol{Faux}) \cup \bigcup_{i=1...n-1} \Diamond\mathrm{V}(A_i) \\
&\quad = \emptyset \cup \bigcup_{i=1...n-1} \Diamond\mathrm{V}(A_i) \\
&\quad = \emptyset \\
&\Longrightarrow A_1 \vee ... \vee A_{n-1} \vee \boldsymbol{Faux} \Rrightarrow A_1 \vee ... \vee A_{n-1}
\end{aligned} \tag{5.17}$$

Élagage d'un opérateur not

Si le fils d'un nœud not a son espace possiblement faux (*resp.* possiblement vrai) vide, l'espace possiblement vrai (*resp.* possiblement faux) du nœud concerné est alors toujours vide. Ainsi, le nœud est toujours faux (*resp.* toujours vrai).

$$\begin{aligned}
&\text{Soit } \boldsymbol{Faux} \text{ le fils d'un opérateur not} \\
&\quad \Diamond\mathrm{V}(\mathsf{not}) = \neg\Diamond\mathrm{V}(\boldsymbol{Faux}) \\
&\quad = \neg\emptyset \\
&\Longrightarrow \neg\boldsymbol{Faux} \Rrightarrow \boldsymbol{Vrai} \\
&\text{Soit } \boldsymbol{Vrai} \text{ le fils d'un opérateur not} \\
&\quad \Diamond\mathrm{V}(\mathsf{not}) = \neg\Diamond\mathrm{V}(\boldsymbol{Vrai}) \\
&\quad = \emptyset \\
&\Longrightarrow \neg\boldsymbol{Vrai} \Rrightarrow \boldsymbol{Faux}
\end{aligned} \tag{5.18}$$

Élagage d'un opérateur =>

Si la condition A d'un nœud A => B ne peut plus être vraie (l'espace possiblement vrai de A est alors vide) ou si la formule B est toujours vraie (l'espace possiblement faux de B est alors vide), alors le nœud => est toujours vrai. En effet, une implication avec une condition fausse est toujours vraie, de même pour une implication avec une formule vraie. Dans ces deux cas, l'opérateur => est remplacé par un nœud ***Vrai***.

$$
\begin{array}{l}
\text{Soit l'implication } \boldsymbol{Faux} \Rightarrow A (A \text{ est une formule logique quelconque}) \\
\quad \Diamond \mathrm{V}(\texttt{=>}) = \Diamond \mathrm{F}(\boldsymbol{Faux}) \cup \Diamond \mathrm{V}(A) \\
\quad\quad\quad\quad = \neg\emptyset \cup \Diamond \mathrm{V}(A) \\
\quad\quad\quad\quad = \neg\emptyset \\
\Longrightarrow \boldsymbol{Faux} \Rightarrow A \Rrightarrow \boldsymbol{Vrai} \\
\text{Soit l'implication } A \Rightarrow \boldsymbol{Vrai} (A \text{ est une formule logique quelconque}) \\
\quad \Diamond \mathrm{V}(\texttt{=>}) = \Diamond \mathrm{F}(A) \cup \Diamond \mathrm{V}(\boldsymbol{Vrai}) \\
\quad\quad\quad\quad = \Diamond \mathrm{F}(A) \cup \neg\emptyset \\
\quad\quad\quad\quad = \neg\emptyset \\
\Longrightarrow A \Rightarrow \boldsymbol{Vrai} \Rrightarrow \boldsymbol{Vrai}
\end{array}
\tag{5.19}
$$

Si la condition A d'un nœud A => B est toujours vraie et si la formule B est toujours fausse, alors le nœud => est toujours faux.

$$
\begin{array}{l}
\text{Soit l'implication } \boldsymbol{Vrai} \Rightarrow \boldsymbol{Faux} \\
\quad \Diamond \mathrm{V}(\texttt{=>}) = \Diamond \mathrm{F}(\boldsymbol{Vrai}) \cup \Diamond \mathrm{V}(\boldsymbol{Faux}) \\
\quad\quad\quad\quad = \emptyset \cup \emptyset \\
\quad\quad\quad\quad = \emptyset \\
\Longrightarrow \boldsymbol{Vrai} \Rightarrow \boldsymbol{Faux} \Rrightarrow \boldsymbol{Faux}
\end{array}
\tag{5.20}
$$

Si la condition A d'un nœud A => B est toujours vraie, celui-ci ne dépend plus que de sa formule B. L'équation 5.21 détaille ce résultat : une implication dont la condition est toujours vraie se résume à sa condition.

$$
\begin{array}{l}
\text{Soit l'implication } \boldsymbol{Vrai} \Rightarrow A \\
\quad \Diamond \mathrm{V}(\texttt{=>}) = \Diamond \mathrm{F}(\boldsymbol{Vrai}) \cup \Diamond \mathrm{V}(A) \\
\quad\quad\quad\quad = \emptyset \cup \Diamond \mathrm{V}(A) \\
\quad\quad\quad\quad = \Diamond \mathrm{V}(A) \\
\Longrightarrow \boldsymbol{Vrai} \Rightarrow A \Rrightarrow A
\end{array}
\tag{5.21}
$$

Si la formule B d'un nœud A => B est toujours fausse, le nœud => devient la négation de sa condition $\neg A$. En effet, pour qu'une implication dont la formule est fausse soit vraie, il faut que sa condition ne soit pas vérifiée.

$$
\begin{array}{l}
\text{Soit l'implication } A \Rightarrow \boldsymbol{Faux} \\
\quad \Diamond \mathrm{V}(\texttt{=>}) = \Diamond \mathrm{F}(A) \cup \Diamond \mathrm{V}(\boldsymbol{Faux}) \\
\quad\quad\quad\quad = \Diamond \mathrm{F}(A) \cup \emptyset \\
\quad\quad\quad\quad = \Diamond \mathrm{F}(A) \\
\Longrightarrow A \Rightarrow \boldsymbol{Faux} \Rrightarrow \neg A
\end{array}
\tag{5.22}
$$

Élagage d'un opérateur <=>

Si les deux fils sont toujours simultanément faux ou vrais, le nœud <=> est toujours vrai. En effet, s'ils sont simultanément égaux, les deux fils vérifient l'équivalence. L'opérateur <=> est ainsi remplacé par un nœud ***Vrai***.

$$
\begin{array}{l}
\text{Soit l'équivalence } \boldsymbol{Faux} \Leftrightarrow \boldsymbol{Faux} \\
\quad \Diamond\mathrm{V}(\texttt{<=>}) = \big(\Diamond\mathrm{F}(\boldsymbol{Faux}) \cap \Diamond\mathrm{F}(\boldsymbol{Faux})\big) \cup \big(\Diamond\mathrm{V}(\boldsymbol{Faux}) \cap \Diamond\mathrm{V}(\boldsymbol{Faux})\big) \\
\qquad\qquad = (\neg\emptyset \cap \neg\emptyset) \cup (\emptyset \cap \emptyset) \\
\qquad\qquad = \neg\emptyset \\
\Longrightarrow \boldsymbol{Faux} \Leftrightarrow \boldsymbol{Faux} \Rrightarrow \boldsymbol{Vrai} \\
\text{Soit l'équivalence } \boldsymbol{Vrai} \Leftrightarrow \boldsymbol{Vrai} \\
\quad \Diamond\mathrm{V}(\texttt{<=>}) = \big(\Diamond\mathrm{F}(\boldsymbol{Vrai}) \cap \Diamond\mathrm{F}(\boldsymbol{Vrai})\big) \cup \big(\Diamond\mathrm{V}(\boldsymbol{Vrai}) \cap \Diamond\mathrm{V}(\boldsymbol{Vrai})\big) \\
\qquad\qquad = (\emptyset \cap \emptyset) \cup (\neg\emptyset \cap \neg\emptyset) \\
\qquad\qquad = \neg\emptyset \\
\Longrightarrow \boldsymbol{Vrai} \Leftrightarrow \boldsymbol{Vrai} \Rrightarrow \boldsymbol{Vrai}
\end{array}
\tag{5.23}
$$

Si les deux fils d'un nœud <=> sont nécessairement opposés, le nœud <=> est toujours faux. En effet, si les deux fils sont opposés, ils ne pourront jamais être équivalents. L'opérateur <=> est ainsi remplacé par un nœud ***Faux***.

$$
\begin{array}{l}
\text{Soit l'équivalence } \boldsymbol{Faux} \Leftrightarrow \boldsymbol{Vrai} \text{ (l'équivalence est symétrique)} \\
\quad \Diamond\mathrm{V}(\texttt{<=>}) = \big(\Diamond\mathrm{F}(\boldsymbol{Faux}) \cap \Diamond\mathrm{F}(\boldsymbol{Vrai})\big) \cup \big(\Diamond\mathrm{V}(\boldsymbol{Faux}) \cap \Diamond\mathrm{V}(\boldsymbol{Vrai})\big) \\
\qquad\qquad = (\neg\emptyset \cap \emptyset) \cup (\emptyset \cap \neg\emptyset) \\
\qquad\qquad = \emptyset \\
\Longrightarrow \boldsymbol{Faux} \Leftrightarrow \boldsymbol{Vrai} \Rrightarrow \boldsymbol{Faux}
\end{array}
\tag{5.24}
$$

Si une branche A d'un nœud A <=> B est toujours vraie (*resp.* toujours fausse), le nœud <=> ne dépendra plus que de la deuxième branche B ; il est alors remplacé par cette deuxième branche B (*resp.* la négation de la deuxième branche $\neg B$).

$$
\begin{array}{l}
\text{Soit l'équivalence } \boldsymbol{Faux} \Leftrightarrow A \text{ (l'équivalence est symétrique)} \\
\quad \Diamond\mathrm{V}(\texttt{<=>}) = \big(\Diamond\mathrm{F}(\boldsymbol{Faux}) \cap \Diamond\mathrm{F}(A)\big) \cup \big(\Diamond\mathrm{V}(\boldsymbol{Faux}) \cap \Diamond\mathrm{V}(A)\big) \\
\qquad\qquad = \big(\neg\emptyset \cap \Diamond\mathrm{F}(A)\big) \cup \big(\emptyset \cap \Diamond\mathrm{V}(A)\big) \\
\qquad\qquad = \Diamond\mathrm{F}(A) \cup \emptyset \\
\qquad\qquad = \Diamond\mathrm{F}(A) \\
\Longrightarrow \boldsymbol{Faux} \Leftrightarrow A \Rrightarrow \neg A \\
\text{Soit l'équivalence } \boldsymbol{Vrai} \Leftrightarrow A \text{ (l'équivalence est symétrique)} \\
\quad \Diamond\mathrm{V}(\texttt{<=>}) = \big(\Diamond\mathrm{F}(\boldsymbol{Vrai}) \cap \Diamond\mathrm{F}(A)\big) \cup \big(\Diamond\mathrm{V}(\boldsymbol{Vrai}) \cap \Diamond\mathrm{V}(A)\big) \\
\qquad\qquad = \big(\emptyset \cap \Diamond\mathrm{F}(A))\big) \cup \big(\neg\emptyset \cap \Diamond\mathrm{V}(A)\big) \\
\qquad\qquad = \emptyset \cup \Diamond\mathrm{V}(A) \\
\qquad\qquad = \Diamond\mathrm{V}(A) \\
\Longrightarrow \boldsymbol{Vrai} \Leftrightarrow \boldsymbol{Vrai} \Rrightarrow \boldsymbol{Vrai}
\end{array}
\tag{5.25}
$$

Remarque : actuellement, dans notre implémentation, l'élagage est définitif. En effet, lorsque celui-ci est utilisé, il est alors impossible de revenir en arrière en relaxant des contraintes de façon à augmenter les domaines de certaines variables. Les branches coupées par l'élagage ne sont pas stockées en mémoire et le problème est alors transformé au cours de sa résolution.

Pour pallier cet inconvénient de l'élagage, nous avons implémenté un drapeau booléen qui permet de spécifier si les branches vides doivent être élaguées ou laissées telles quelles. Ainsi, l'élagage permet d'améliorer le temps de réponse aux dépens de la possibilité pour

l'utilisateur de revenir sur ces choix. Les algorithmes de filtrage (avec élagage) des opérateurs de composition logique sont présentés en annexes, pages 141 à 145.

5.4.9 Problèmes de stabilité numérique (convergence)

La réinjection de domaines numériques peut causer des problèmes de stabilité numérique. En effet, les domaines continus peuvent être indéfiniment réduits à chaque passe de façon asymptotique par rapport au point fixe. Par exemple, une équation comme :

$$x \leq x^2, \quad \forall x \in [0,1[$$

peut provoquer un certain nombre d'itérations avant d'en arriver à la solution $x = 0$.

Ainsi, nous devons implémenter un système de « pas numérique » permettant d'arrêter les itérations lorsque la différence entre les valeurs d'entrée et les valeurs après filtrage sont inférieures à un certain seuil. Cette idée est reprise de la 2B-ω cohérence (LHOMME & RUEHER, 1997).

5.5 Discussion et conclusion

Dans ce chapitre, nous avons présenté notre implémentation comprenant un langage de description de CSP, un analyseur syntaxique et deux moteurs : un pour la résolution et un pour le filtrage.

Le langage de description permet la saisie de domaines et de variables de CSP « classiques » et la saisie de contraintes exprimées comme des formules logiques.

L'analyseur syntaxique transforme le CSP en un arbre dont la racine est une conjonction de toutes les contraintes et des domaines des variables.

Le moteur de résolution (de CSP discrets) se base sur cette structure en arbre pour évaluer les branches au fur et à mesure de la valuation des variables que ces branches contraignent.

Le moteur de filtrage utilise cette structure en arbres et attache à chaque nœud de l'arbre deux espaces : un espace possiblement vrai et un espace possiblement faux. Le filtrage consiste alors à maintenir l'espace possiblement vrai au niveau de la racine de l'arbre en agrégeant les espaces possiblement vrais de ses fils depuis chacune des feuilles de comparaison. Nous proposons aussi un élagage de l'arbre permettant de couper des branches vides.

L'interêt de cette implémentation est sa possibilité d'exprimer les contraintes les plus diverses. La structure en arbre syntaxique nous permet de décrire des contraintes assemblant tables de compatibilité et formules logiques (ce qui est souvent nécessaire en conception). Cette implémentation nous permet aussi de filtrer facilement des CSP mixtes, grâce à la décomposition en deux niveaux de l'arbre syntaxique (niveau logique et feuilles de comparaison). Une contrepartie de ces possibilités est le stockage mémoire des domaines associés à chaque nœud.

Ce chapitre présentait l'implémentation pour une utilisation sur des CSP « classiques », donc sur des RCSP traduits. Dans le chapitre suivant, nous allons détailler l'adaptation de cette implémentation pour exploiter les spécificités des RCSP. La puissance de cette implémentation en termes d'expressivité des contraintes sera utile pour permettre la description de contraintes traduites depuis un RCSP.

Chapitre 6

Enrichissements de l'implémentation pour exploiter le formalisme RCSP

Dans le chapitre précédent, nous avons présenté notre implémentation pour résoudre ou filtrer des CSP « classiques ». Ce chapitre propose des améliorations et adaptations de cette implémentation pour :

- faciliter la traduction depuis les RCSP en CSP exploitables par notre implémentation ;
- améliorer des performances en résolution en tenant compte des spécificités des RCSP ;
- ajouter des mécanismes de filtrage spécifiques aux RCSP (le méta-opérateur *Sure* en particulier).

Dans un premier temps, nous présentons l'enrichissement du langage de description de CSP permettant de décrire des RCSP. Ensuite, nous présentons des améliorations du moteur de résolution permettant de tenir compte des spécificités des RCSP pour la résolution. Enfin, nous illustrons en quoi les préconisations de modélisation du chapitre 4 peuvent améliorer la qualité du filtrage puis nous présentons des adaptations tenant compte de méta-informations pour améliorer l'interactivité du filtrage.

6.1 Enrichissement du langage

Cette section présente les fonctionnalités du langage de description permettant d'attacher des attributs aux variables, ou de grouper certains éléments d'un CSP.

6.1.1 Les attributs de variables

Dans les RCSP, les variables peuvent posséder un certain nombre d'attributs, (un attribut de pertinence par exemple). Pour saisir ces attributs, nous utiliserons le mot-clé *with* suivi d'accolades pour lister les attributs et leur type selon la syntaxe suivante :

Syntaxe : `<variable> with {<type> attribute <nom> [domain]} ;`

avec
- `variable` est une déclaration de variable, cf. page 71
- `type` compris dans `boolean`, `symbolic` ou `float`
- `domain` est un domaine, cf. page 71

Ainsi, la déclaration d'une variable réelle a munie d'un attribut de pertinence se fait de la façon suivante :

```
float variable a in {> 0} with {
  boolean attribute pertinence;
};
```

Il est alors possible de contraindre la valeur de cet attribut *via* l'accesseur « . ». En supposant que la pertinence de la variable a soit soumise à la vérification d'une formule `X`, nous obtenons alors la contrainte suivante :

```
X => (a.pertinence eq "1");
```

Toutes les contraintes impliquant la variable a font alors intervenir son attribut de pertinence ; en effet, la traduction de RCSP en CSP et le postulat 1 page 51 impliquent que chacune des contraintes n'est pertinente que si les variables qu'elle contraint sont pertinentes.

Cette traduction est du ressort du modeleur ; l'analyseur n'effectue pas de traduction automatique. Du point de vue du moteur, un attribut intervient dans les contraintes au même titre qu'une variable.

Par exemple, soit la variable a (a réelle, $a > 0$) à pertinence non déterminée liée à une variable b ($b \in \{"i", "j", "k"\}$) toujours pertinente par la contrainte :

$$(b = "i") \Leftrightarrow (a \in [0,5]) ;$$

Cette contrainte est alors traduite en utilisant l'attribut de pertinence de la façon suivante :

$$(\neg\, a.pertinence) \vee \big((b = "i") \Leftrightarrow (a \in [0,5])\big).$$

Cette formulation est ainsi toujours vérifiée lorsque `a` n'est pas pertinente. Cependant, cette utilisation des attributs pour la pertinence n'est pas utilisée pour les variables symboliques. L'ajout d'une valeur ★ dans le domaine de la variable est du ressort du modeleur ; pour la suite des travaux, il serait intéressant de prévoir un langage de description de RCSP dont l'analyseur transformerait les attributs de pertinence en attributs de variables ou en augmentant le domaine de celles-ci de la valeur ★, suivant les préconisations que nous avons établies au chapitre 4.

Enfin, nous pouvons imaginer d'autres attributs de variables ; par exemple, un attribut de visibilité pour l'utilisateur qui permet de choisir si l'utilisateur voit la variable et peut donc la contraindre directement (valuer ou réduire son domaine) ou non.

6.1.2 L'aspect hiérarchique

Cette particularité de notre implémentation se base sur le besoin de traiter des sous-ensembles de composants, paramètres et/ou contraintes.

Le langage de description prévoit deux étapes pour la déclaration de groupes de variables et/ou contraintes : la définition puis l'instanciation.

La première étape consiste à définir les groupes grâce aux mots-clé *define* et *group*. La définition du groupe proprement dite suit entre accolades : déclaration de domaines, de variables, puis contraintes.

Syntaxe : `define group <nom> (<paramètres>) {<sous-problème>}`

avec { `paramètres` est une liste de `<type> nom`
`sous-problème` peut comprendre des déclarations de :
domaines, variables, contraintes et groupes

Pour utiliser le groupe défini ainsi, il faut alors l'instancier grâce au mot-clé *group* :

Syntaxe : `group <nom> (<paramètres>) ;`

avec { `nom` est le nom d'un groupe défini
`paramètres` est une liste de variables définies à l'extérieur du groupe

L'analyseur créera alors un nœud and auquel se rattachent les déclarations de variables et de contraintes ; ces contraintes ne seront pas directement rattachées à la racine de l'arbre, comme l'illustre la figure 6.1.

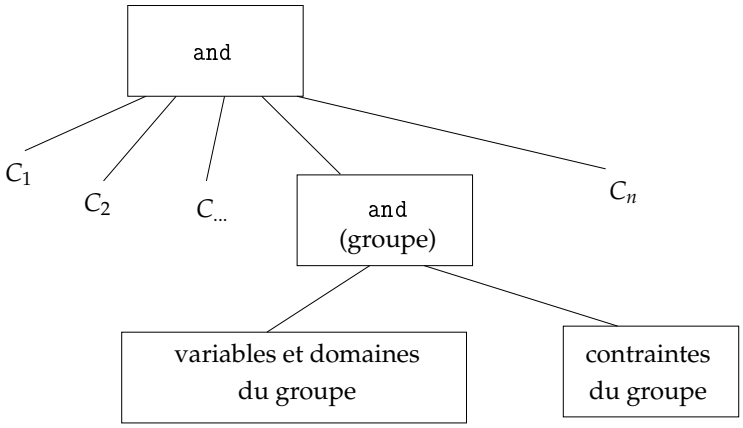


FIG. 6.1 — Place d'un groupe dans l'arbre syntaxique

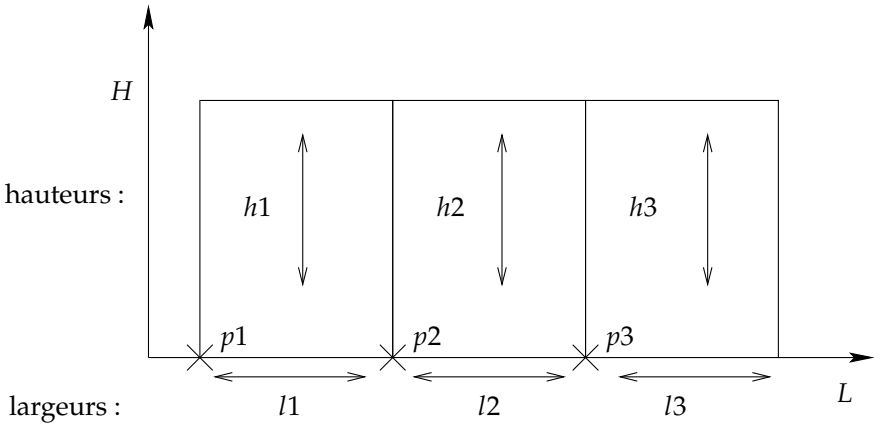
Nous avons implémenté une approche paramétrique des groupes. Nous entendons par approche paramétrique la possibilité de fournir au groupe des variables existantes, dont la valeur pourra être reprise à l'intérieur du groupe, sans avoir à faire appel à une variable extérieure au groupe.

Cette approche paramétrique peut être utile pour l'utilisation de modèles objets d'un produit qui contiendrait plusieurs fois le même sous-ensemble. Un même groupe peut ainsi être instancié plusieurs fois avec des paramètres différents.

L'exemple 6.2 page suivante illustre la définition puis l'instanciation de groupes sur un problème de configuration de meubles. Sur cette exemple, une définition suffit pour ensuite instancier le groupe `Biblio` trois fois, avec des paramètres différents.

EX. 6.2 — Groupes paramétriques — placement et configuration de bibliothèques

Un utilisateur veut disposer trois bibliothèques (de hauteurs variables) le long d'un mur de dimension $H \times L$. Les positions (p, coin inférieur gauche) des bibliothèques dépendent des largeurs (l) des bibliothèques (les bibliothèques se touchent).



Dans le langage de description, nous pouvons alors définir un groupe `Biblio`, qui sera instancié avec des paramètres différents pour chacune des bibliothèques (H et L représentent respectivement la hauteur et la longueur du mur) :

```
float variable H in [2, 5] ;  // hauteur du mur
float variable L in [3, 15] ; // largeur du mur
float variable l1 in [0, 3] ; // largeur bibliothèque 1
float variable l2 in [0, 3] ; // largeur bibliothèque 2
float variable l3 in [0, 3] ; // largeur bibliothèque 3
float variable p1 in [0, L] ; // position bibliothèque 1
float variable p2 in [0, L] ; // position bibliothèque 2
float variable p3 in [0, L] ; // position bibliothèque 3
define group Biblio (float l, float p) {
  symbolic variable couleur in  {"bleu", "rouge", "vert"} ;
  float variable h in [0, 4] ; // hauteur de la bibliothèque
  h < H ; // la hauteur est inférieure à la hauteur du mur
  l + p < L ; // la bibliothèque ne doit pas dépasser le mur
  h < 2 * l ; // pour des proportions harmonieuses, la hauteur est
  h > l ;    //  comprise entre la largeur et 2 fois la largeur
}
l1 + l2 + l3 < L ;
group Biblio (l1, p1); // instanciation de la bibliothèque 1
group Biblio (l2, p2); // instanciation de la bibliothèque 2
group Biblio (l3, p3); // instanciation de la bibliothèque 3
```

Pour un groupe dont nous n'identifions aucun paramètre, les définitions et instanciations suivent le même principe. Il n'est alors pas nécessaire de spécifier des paramètres entre parenthèses :

```
define group A () {
  ...
}
group A();
```

Cette implémentation des groupes paramétriques nous permet donc de définir une seule fois un groupe pour l'instancier ensuite plusieurs fois. Cependant, cette utilisation des groupes paramétriques différencie les instances par le nom des paramètres. Il n'est ainsi pas possible de créer différentes instances d'un même groupe en utilisant les mêmes paramètres, ce qui pourrait être utile. Pour cela, nous proposons de pointer plusieurs fois vers la même instance dans l'arbre syntaxique. Un fils pourrait donc avoir plusieurs pères, et l'arbre deviendrait un treillis.

Pour la description des attributs de groupes, nous identifions deux artifices. Ces artifices sont nécessaires car il n'est pas possible, avec ce langage, de contraindre l'attribut d'un groupe si celui-ci n'existe pas (n'est pas pertinent). Ainsi, pour attacher un attribut à un groupe, il est possible d'utiliser une variable locale (par exemple, la couleur dans l'exemple 6.2 page précédente). Pour des attributs de pertinence, une solution consiste à englober le groupe à pertinence non déterminée dans un autre groupe, dont une variable locale rend compte de la participation du groupe à la solution du problème. Par exemple, un groupe A à pertinence non-déterminée peut être inclus dans un groupe B pour gérer sa pertinence de la façon suivante :

```
define group B () {
  boolean variable pertinence ;
  define group A () {
    ...
  }
  pertinence eq "0" or group A ;
}
```

Cette solution pour gérer un attribut de pertinence pour un groupe a l'avantage de ne pas nécessiter de traduction pour les contraintes. La traduction est comprise dans la formulation de la seule contrainte du groupe B.

6.1.3 Conclusion

Ces enrichissements du langage de description permettent la saisie de RCSP uniquement par des modifications sur le nom des variables. L'analyseur interprète les attributs comme de nouvelles variables et les groupes comme des espaces de nommage. Cette utilisation de noms de variables permet de résoudre ou filtrer un RCSP sans modifier les moteurs développés pour les CSP ; l'analyseur ne fait que traduire les attributs ou groupes en noms spécifiques. Ceci permet au modeleur de saisir un RCSP tout en maintenant les moteurs de résolution ou de filtrage identiques.

La section suivante présente des changements dans le moteur tenant compte des spécificités des RCSP pour en améliorer la résolution.

6.2 Amélioration de la résolution par la prise en compte de la pertinence

Nous allons proposer deux types d'améliorations pour la résolution : en premier lieu, nous nous basons sur les heuristiques décrites dans le chapitre précédent (cf. section 5.3.3 page 77) pour les améliorer, sachant que certaines variables ne seront pas pertinentes. Ensuite, nous proposons une amélioration de l'algorithme de *BackTracking*.

6.2.1 Critères d'évaluation et problèmes considérés

Pour évaluer nos propositions en résolution (algorithme *BackTrack*), nous nous basons sur les critères suivants :

- nombre de solutions atteintes ;
- nombre de *BackTracks* ;
- nombre de choix de valuations (nombre de nœuds de *BackTrack*) ;
- nombre de vérifications de contraintes ;
- le temps de calcul.

Le nombre de solutions atteintes peut différer en fonction de l'ordre de valuation des variables. En effet, si une variable de base est valuée avant sa variable de pertinence associée, pour chaque valeur de la variable de base, la solution du RCSP sera équivalente.

Le nombre de *BackTracks* est incrémenté pour chaque retour-arrière. Le nombre de nœuds parcourus donne une indication un peu plus précise pour l'évaluation, mais nous ne pouvons disposer du temps d'évaluation par contrainte.

Le critère le plus intéressant est le nombre d'évaluations de contraintes. En effet, cette évaluation est l'opération la plus coûteuse. Nos propositions seront d'autant plus efficaces qu'elles permettront une réduction significative du nombre d'évaluations de contraintes.

Le temps CPU est donné à titre purement indicatif. La machine ayant servi à faire les tests est la suivante :

- Processeur : Intel Celeron 2,4 GHz ;
- Mémoire vive : 479 Mo, mémoire virtuelle : 1256 Mo ;
- Système d'exploitation : FreeBSD 5.5 ;
- Version du langage : Perl 5.8.8 ;
- Chaque résultat est la moyenne de cinquante tests.

Pour évaluer nos propositions en résolution, nous nous basons sur différents problèmes :

- la voiture simple, détaillée dans l'exemple 1.7 page 18 — cet exemple comporte quatre variables (dont une à pertinence non-déterminée), le produit cartésien des domaines est de cardinal 16 ;
- la voiture de MITTAL & FALKENHAINER (1990) — cet exemple est reproduit en annexe (cf. exemple B.1 page 147), il comporte huit variables (dont cinq dont la pertinence doit être déterminée) et le produit cartésien des domaines est de cardinal 1296.

De façon à pouvoir tester les heuristiques spécifiques aux attributs de pertinence, nous avons testé les exemples de la voiture simple et de la voiture de MITTAL & FALKENHAINER (1990) aussi bien avec des valeurs ★ qu'avec des attributs de pertinence. Ceci nous permettra de vérifier la cohérence de nos préconisations de modélisation par rapport à la résolution.

6.2.2 Adaptation des heuristiques de choix de variables

Présentation des différentes heuristiques

Dans cette section, nous cherchons à comparer les heuristiques avec de nouvelles heuristiques que nous avons identifiées pour les CSP pourvus de systèmes de gestion de la pertinence. Les heuristiques suivantes constituent notre base de tests :

- ordre aléatoire $\boldsymbol{H_0}$: ceci constitue notre référence ;
- heuristique $\boldsymbol{H_1}$: instancier en premier les variables apparaissant dans le plus de contraintes, en cas d'égalité, choisir la variable ayant le plus petit domaine (HARALICK & ELLIOTT, 1980; SMITH, 1995) — heuristique *Most Constrained Variables*.

Étant donné que nous traitons des RCSP, et donc que des variables peuvent ne pas être pertinentes, nous pouvons identifier d'autres heuristiques. Celles-ci sont basées sur le fait qu'une variable non-pertinente ne fera pas partie de la solution, et sa valeur n'a donc aucune importance. Nous rappellons que lors de la traduction d'un RCSP en CSP, les attributs de pertinence sont considérés comme des variables, que nous appelons variables de pertinence. Pour les trois heuristiques basées sur ce principe, les variables sont divisées en deux sous-ensembles issus de la traduction d'un RCSP en CSP : l'ensemble des variables de base et l'ensemble des variables de pertinence. Le moteur de résolution fait la différence entre variables de base et variables de pertinence grâce aux noms fournis par l'analyseur (une variable `a` a un attribut de pertinence de la forme `a.pertinence`).

- heuristique $\boldsymbol{H_2}$: instancier en premier l'ensemble des variables de pertinence — à l'intérieur de chacun des ensembles, l'ordre est aléatoire ;
- heuristique $\boldsymbol{H_3}$: appliquer $\boldsymbol{H_2}$ puis $\boldsymbol{H_1}$ — les variables de base et de pertinence sont séparées ; les variables de pertinence sont instanciées en premier, puis pour chacun des ensembles (variables de base et variables de pertinence), on instancie en premier les variables les plus contraintes (en cas d'égalité, on choisit celle qui a le plus petit domaine) ;
- heuristique $\boldsymbol{H_4}$: appliquer $\boldsymbol{H_1}$ puis $\boldsymbol{H_2}$ — on instancie d'abord les variables apparaissant dans le plus de contraintes, en cas d'égalité on utilise la taille des domaines ($\boldsymbol{H_1}$), s'il reste des égalités, on instancie les variables de pertinence en premier ($\boldsymbol{H_2}$).

Résultats avec l'algorithme *BackTrack*

Le tableau 6.3 page suivante montre les différents résultats obtenus pour la modélisation RCSP utilisant les valeurs ★. Le nombre de solutions du CSP et du problème de conception sont équivalents en utilisant la modélisation par la valeur ★ (NA indique que ces heuristiques ne sont pas applicables — Non Applicable — en effet, il n'y a pas de variable de pertinence sur des problèmes dont la pertinence est modélisée par une valeur ★).

Nous pouvons constater sur l'exemple de la voiture simple que les résultats aléatoires sont meilleurs que les résultats utilisant les heuristiques ($\boldsymbol{H_2}$ ne classe pas les variables, étant donné qu'il n'y a pas de variables de pertinence). Cet exemple ayant été choisi pour sa simplicité, nous pouvons cependant dire qu'il n'est pas significatif étant donné sa taille. En revanche, sur l'exemple de MITTAL & FALKENHAINER (1990), l'heuristique $\boldsymbol{H_1}$ fournit de bien meilleurs résultats sur tous les critères étudiés.

Le tableau 6.4 page 105 montre les résultats obtenus en modélisant la pertinence des variables grâce à un attribut de pertinence. Ici, les heuristiques RCSP que nous avons développées ont leur interêt ; en effet, le fait de valuer en premier les variables de pertinence est alors utile pour le temps de vérification des contraintes traduites.

Dans ce cas, nous constatons l'efficacité des heuristiques consistant à valuer les variables de pertinence en premier. Sur l'exemple de la voiture de MITTAL & FALKENHAINER (1990), nous pouvons constater que les deux heuristiques $\boldsymbol{H_2}$ et $\boldsymbol{H_4}$ sont à peu près équivalentes ; nous proposons donc de choisir $\boldsymbol{H_4}$ (instancier d'abord les variables apparaissant dans le plus de contraintes, en cas d'égalité utiliser la taille des domaines, s'il reste des égalités, instancier les variables de pertinence en premier).

Le fait de choisir l'heuristique $\boldsymbol{H_4}$ nous permet de traiter aussi des problèmes modélisés avec le symbole ★ pour la non-pertinence. En effet, le moteur de résolution se base sur les

TAB. 6.3 — Résultats en configuration autonome sur les deux exemples avec l'algorithme *BackTrack* — modélisation par ★

Heuristique	*BackTracks*	Vérifications de contraintes	Nœuds parcourus	Solutions du CSP (du problème)	temps CPU
Exemple de la voiture simple					
$\boldsymbol{H_0}$	11, 15	81, 65	32, 3	8 (8)	0, 14
$\boldsymbol{H_1}$	16, 0	114, 0	38, 0	8 (8)	0, 17
$\boldsymbol{H_2}$	NA	NA	NA	NA	NA
$\boldsymbol{H_3}$	NA	NA	NA	NA	NA
$\boldsymbol{H_4}$	NA	NA	NA	NA	NA
Exemple de la voiture de MITTAL & FALKENHAINER (1990)					
$\boldsymbol{H_0}$	1472, 65	13476, 2	2837, 5	450 (450)	16, 84
$\boldsymbol{H_1}$	694, 8	6905, 05	1619, 7	450 (450)	8, 1
$\boldsymbol{H_2}$	NA	NA	NA	NA	NA
$\boldsymbol{H_3}$	NA	NA	NA	NA	NA
$\boldsymbol{H_4}$	NA	NA	NA	NA	NA

noms des variables attribués par l'analyseur pour établir l'ordre d'instanciation. Lorsqu'il n'existe pas de variable de pertinence, le classement des deux ensembles de variables (de base et de pertinence) n'intervient pas ; $\boldsymbol{H_4}$ est alors équivalente à $\boldsymbol{H_1}$ (heuristique *Most Constrained Variables*). De plus, $\boldsymbol{H_4}$ peut aussi s'adapter aux groupes.

6.2.3 Adaptation du *BackTrack* aux RCSP

Nous l'avons vu, les variables non-pertinentes n'ont pas à être valuées pour atteindre une solution. Nous choisissons donc de leur assigner une valeur arbitraire et de ne pas tester les contraintes les impliquant. Ainsi, nous pouvons éviter au moins une vérification de contrainte par variable non-pertinente. L'algorithme 6.5 page 106 est la version adaptée à ce fonctionnement de l'algorithme 5.5 page 76. Cette adaptation correspond à une prise en compte du postulat 1 page 51 : les variables non-pertinentes ne sont pas prises en compte, et n'ont donc pas besoin d'être valuées.

Nous pouvons noter que cet algorithme atteint une efficacité maximale en l'utilisant conjointement avec les heuristiques $\boldsymbol{H_2}$, $\boldsymbol{H_3}$ ou $\boldsymbol{H_4}$, qui permettent de s'assurer que les variables de pertinence sont valuées en premier. Pour illustrer les gains que nous pouvons attendre par rapport aux attributs de pertinence des groupes, nous avons effectué ces tests sur la modélisation non préconisée avec les attributs de pertinence.

Le tableau 6.6 page 106 montre les résultats obtenus avec cet algorithme. Ainsi, nous pouvons vérifier que les heuristiques spécifiques aux RCSP que nous avons proposées procurent de réelles améliorations, sur tous les critères retenus. Les performances de cet algorithme *BackTrack* modifié avec la modélisation par attribut de pertinence rivalisent avec celles obtenues avec la modélisation avec la valeur ★.

Cependant, dans le chapitre 4, nous n'avons pas retenu la modélisation par l'ajout d'un attribut de pertinence pour les variables symboliques. En revanche, ces heuristiques peuvent être mises à profit pour les attributs de pertinence de groupes. Ainsi, lorsqu'un groupe n'est

TAB. 6.4 — Résultats en configuration autonome sur les deux exemples avec l'algorithme *BackTrack* — modélisation par attribut de pertinence

Heuristique	*BackTracks*	Vérifications de contraintes	Nœuds parcourus	Solutions du CSP (du problème)	temps CPU
Exemple de la voiture simple					
$\boldsymbol{H_0}$	12,3	109,7	46,6	12 (8)	0,17
$\boldsymbol{H_1}$	12,1	104,6	46,2	12 (8)	0,16
$\boldsymbol{H_2}$	9,25	85,45	40,5	12 (8)	0,14
$\boldsymbol{H_3}$	9,95	89,55	41,9	12 (8)	0,15
$\boldsymbol{H_4}$	9,2	85,5	40,4	12 (8)	0,15
Exemple de la voiture de MITTAL & FALKENHAINER (1990)					
$\boldsymbol{H_0}$	5371,05	44722,35	11635,1	1072 (450)	64,49
$\boldsymbol{H_1}$	2051,0	17855,65	4792,8	1072 (450)	29,14
$\boldsymbol{H_2}$	1675,1	16794,4	4653,9	1072 (450)	30,07
$\boldsymbol{H_3}$	2607,3	22223,2	5597,3	1072 (450)	32,76
$\boldsymbol{H_4}$	1934,15	16801,55	4592,4	1072 (450)	26,62

pas pertinent, nous pouvons alors attribuer arbitrairement une valeur à chacune de ses variables, et le composant modélisé ainsi ne fera pas partie de la solution du problème de conception.

6.2.4 Conclusion

Nous avons présenté dans cette section des tests sur la performance en résolution de RCSP discrets pouvant s'appliquer aux problèmes de conception. Ces tests justifient nos préconisations de modélisation pour la pertinence de variables (donc de composants à choisir dans une liste ou de composants à un paramètre discret). Ces résultats montrent que l'exploitation des informations spécifiques aux RCSP améliore clairement les performances.

Nous identifions d'autres heuristiques qu'il serait intéressant de tester : des heuristiques dynamiques. Le principe est de remettre à jour l'ordre des variables avant chaque valuation, ce qui permet de toujours faire le choix localement optimal pour la variable à valuer. Ces heuristiques sont basées sur le même principe que $\boldsymbol{H_1}$: à chaque pas, il s'agit d'instancier la variable la plus contrainte ou ayant le plus petit domaine. Ces heuristiques ont été détaillées par BESSIÈRE & RÉGIN (1996).

Alg. 6.5 RCSP-BT (V_v, V_n)

– ∴ – *Amélioration du* BackTrack *pour notre implémentation*
– ∴ – *Deux ensembles ordonnés :* V_v *l'ensemble des couples (variable valuée, valeur choisie)*
– ∴ – V_n *l'ensemble des couples (variable à instancier, domaine d'instanciation)*
– ∴ – x *une variable et* x^p *son éventuel attribut de pertinence*
coherent ← 1
Si ($V_n = \emptyset$) **Alors**
  V_v est une solution
  Retour V_v
Sinon
  CHOISIR & EFFACER x dans V_n
  Si ($x^p = \neg pertinent$) **Alors**
    $x = x_0 / x_0 \in D_x$
  Sinon
    Pour Chaque $i \in D_x$ **Faire**
      IDENTIFIER $C_x \in C$ tel que C_x contraint x
      coherent ← CALC (C_x) – ∴ – *cf. définition 5.3 page 75*
      Si (coherent) **Alors**
        BACKTRACKING ($V_v \cup (x, i)$, $V_n \setminus x$)
      Fin Si
    Fin Pour
  Fin Si
Fin Si

TAB. 6.6 — Résultats en configuration autonome sur les deux exemples avec l'algorithme *RCSP-BT* — modélisation par attribut de pertinence

Heuristique	*BackTracks*	Vérifications de contraintes	Nœuds parcourus	Solutions du CSP (du problème)	temps CPU
Exemple de la voiture simple					
H_0	10,45	72,8	37,3	9,2 (8)	0,15
H_1	11,8	83,3	42,0	10,2 (8)	0,18
H_2	8,7	57,1	31,4	8 (8)	0,12
H_3	8,3	55,2	30,6	8 (8)	0,13
H_4	8,35	55,5	30,7	8 (8)	0,12
Exemple de la voiture de MITTAL & FALKENHAINER (1990)					
H_0	3882,2	27954,6	8610,0	772,5 (450)	50,76
H_1	848,05	7806,2	2458,9	724,8 (450)	14,15
H_2	553,45	5331,95	1675,1	450 (450)	10,31
H_3	524,95	5284,37	1497,11	450 (450)	9,8
H_4	492,68	4896,05	1447,84	450 (450)	9,06

6.3 Filtrage de RCSP— résultats et adaptation

Les adaptations de notre implémentation pour le filtrage de RCSP se situent sur deux points :

- la déduction de la non-pertinence des variables, que nous illustrons par les résultats du filtrage ;
- l'exploitation de méta-informations, et en particulier du méta-opérateur *Sure*.

6.3.1 Déduction de la non-pertinence de variables

Pertinence de variables discrètes

Pour étudier la différence de comportement des RCSP selon la modélisation choisie (★ ou attribut de pertinence), nous nous basons sur l'exemple 1.7 page 18. Nous proposons de valuer la variable *Vitres* à manuelles et la variable *Intérieur* à cuir. Avec notre implémentation, un filtrage de qualité devrait pouvoir conclure que la voiture est obligatoirement de type normal *via* la non-pertinence de la variable *Toit Ouvrant*.

Modélisation par ★

La figure 6.7 (déjà vue au chapitre 4) illustre la modélisation du problème sous forme d'un RCSP dont l'équation 6.1 montre les contraintes.

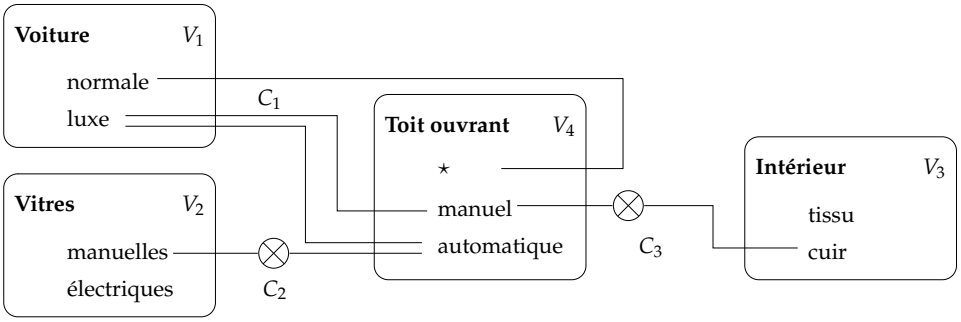


FIG. 6.7 — Configuration de voiture — modélisation par RCSP et valeur ★

$$\begin{array}{lll} C_1 : & V_1 = \text{luxe} \Leftrightarrow V_4 \neq \bigstar \\ C_2 : & V_4 = \text{automatique} \Rightarrow V_2 \neq \text{manuelles} \\ C_3 : & V_4 = \text{manuel} \Rightarrow V_3 \neq \text{cuir} \end{array} \tag{6.1}$$

Après les deux valuations (vitres manuelles et intérieur cuir), le domaine de *Toit Ouvrant* est réduit à ★. Lors de la seconde phase de propagation, le moteur déduit que *Voiture* ne peut plus être valuée à luxe ; son domaine est donc réduit à normal (seule combinaison valide). Au niveau de la racine de l'arbre syntaxique, nous obtenons donc l'espace possiblement vrai suivant :

$$\Diamond \mathrm{V} : \left\{ \begin{array}{ll} \textit{Voiture} & = \{\text{normale}\} \\ \textit{Vitres} & = \{\text{manuelles}\} \\ \textit{Intérieur} & = \{\text{cuir}\} \\ \textit{Toit Ouvrant} & = \{\bigstar\} \end{array} \right\}$$

Cet espace possiblement vrai est obtenu après la seconde phase de propagation ; la figure 6.8 illustre cette seconde phase de filtrage.

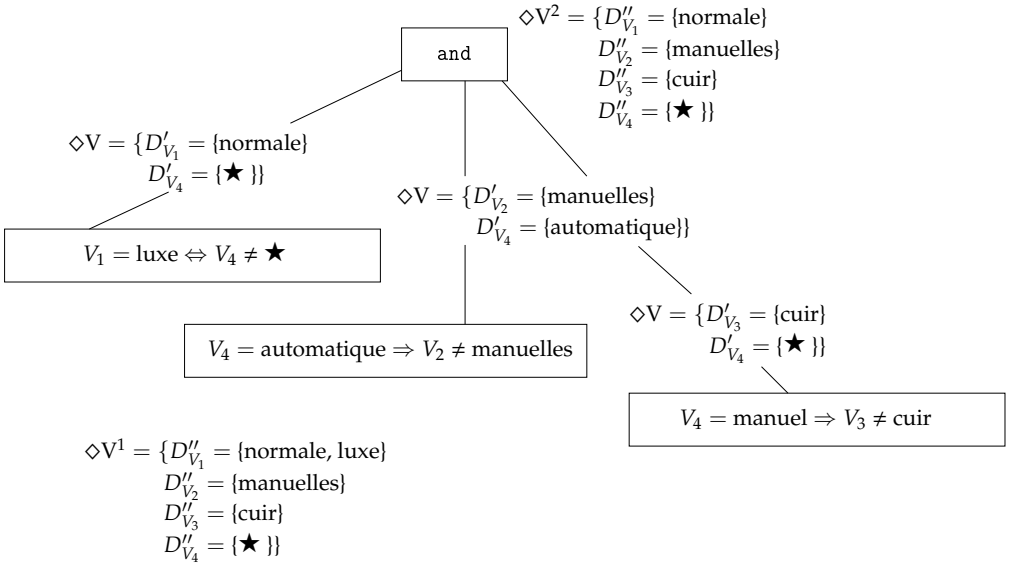


FIG. 6.8 — Arbre syntaxique — modélisation par RCSP et valeur ★

Modélisation par attribut de pertinence

La figure 6.9 illustre la modélisation du problème sous forme d'un RCSP dont l'équation 6.2 montre les contraintes.

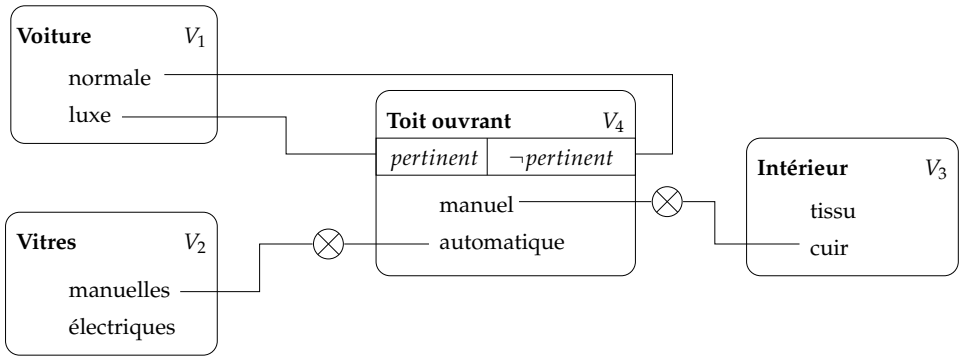


FIG. 6.9 — Configuration de voiture — modélisation par RCSP et attribut de pertinence

$$\begin{array}{lll} C_1 : & V_1 = \text{luxe} \Leftrightarrow V_4 \textit{ pertinent} \\ C_2 : & V_4 = \text{automatique} \Rightarrow V_2 \neq \text{manuelles} \\ C_3 : & V_4 = \text{manuel} \Rightarrow V_3 \neq \text{cuir} \end{array} \tag{6.2}$$

Le RCSP est ensuite traduit en un CSP : chaque attribut de pertinence devient une variable de pertinence et chaque variable une variable de base ; le postulat 1 page 51 est utilisé pour exprimer les contraintes.

L'équation 6.3 page suivante montre le CSP issu d'une traduction directe du RCSP.

$$\begin{array}{ll} C_1: & V_1 = \text{ luxe } \Leftrightarrow V_4^p = \textit{ pertinent} \\ C_2: & (V_4^p = \neg \textit{pertinent}) \vee (V_4 = \text{ automatique } \Rightarrow V_2 \neq \text{ manuelles }) \\ C_3: & (V_4^p = \neg \textit{pertinent}) \vee (V_4 = \text{ manuel } \Rightarrow V_3 \neq \text{ cuir }) \end{array} \tag{6.3}$$

Après les deux valuations (vitres manuelles et intérieur cuir), aucun domaine n'est filtré. En effet, les contraintes C_2 et C_3 peuvent encore être satisfaites si l'attribut de pertinence de *Toit Ouvrant* est $\neg \textit{pertinent}$. Les opérations de filtrage sur les espaces possiblement vrais ne permettent pas au moteur de détecter la future incohérence si l'utilisateur value *Voiture* à luxe. Nous obtenons donc l'espace possiblement vrai à la racine de l'arbre syntaxique suivant :

$$\Diamond \mathrm{V} : \left\{ \begin{array}{ll} \textit{Voiture} & = \{\text{normale, luxe}\} \\ \textit{Vitres} & = \{\text{manuelles}\} \\ \textit{Intérieur} & = \{\text{cuir}\} \\ \textit{Toit Ouvrant} & = \{\text{automatique, manuel}\} \\ \textit{Toit Ouvrant}_p & = \{\textit{pertinent}, \neg \textit{pertinent}\} \end{array} \right\}$$

Cet espace est obtenu après une seule phase de propagation ; l'arbre syntaxique de la figure 6.10 illustre l'obtention de cet espace possiblement vrai.

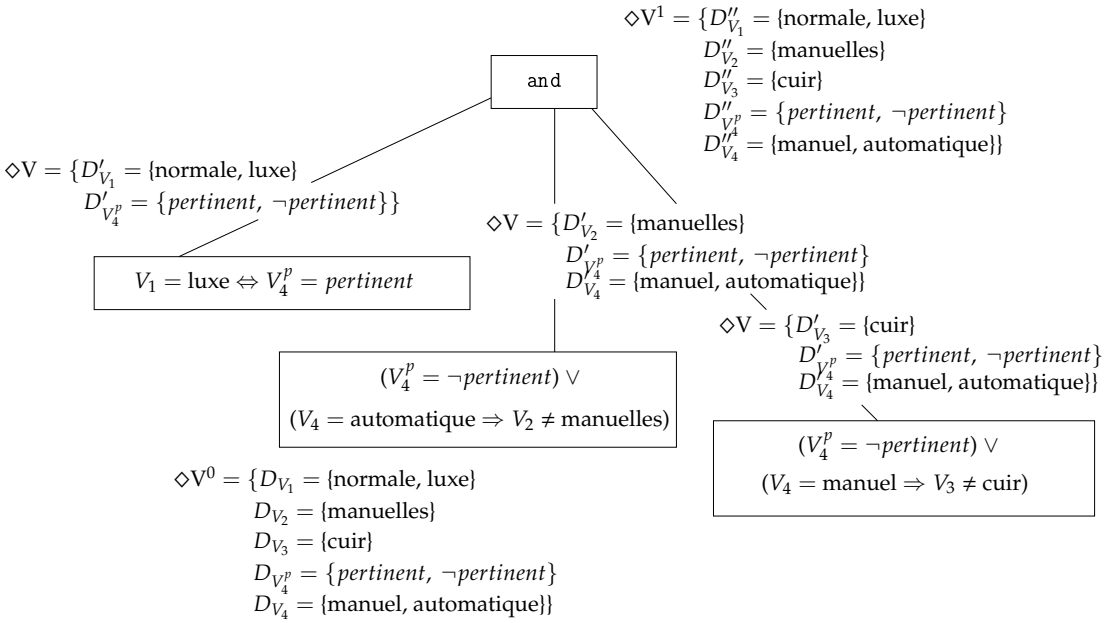


FIG. 6.10 — Arbre syntaxique — modélisation par RCSP et attribut de pertinence

En traduisant le RCSP en CSP de façon plus élaborée, nous obtenons les contraintes de l'équation 6.4. Cette traduction demande une factorisation dans l'arbre syntaxique. Elle est obtenue en factorisant l'opérande $(V_4^p = \neg \textit{pertinent})$ des contraintes C_2 et C_3 de l'équation 6.3 pour obtenir une seule contrainte : C'_2.

$$\begin{array}{ll} C_1: & V_1 = \text{ luxe } \Leftrightarrow V_4^p = \textit{ pertinent} \\ C'_2: & (V_4^p = \neg \textit{pertinent}) \vee \\ & \big((V_4 = \text{ automatique } \Rightarrow V_2 \neq \text{ manuelles }) \wedge \\ & \quad (V_4 = \text{ manuel } \Rightarrow V_3 \neq \text{ cuir })\big) \end{array} \tag{6.4}$$

En utilisant cette formulation, les deux valuations (vitres manuelles et intérieur cuir) permettent d'effectuer les mêmes déductions que la modélisation avec ★. Pour vérifier la contrainte C_2', la variable *Toit Ouvrant* devra être non-pertinente. En effet, les deux implications que C_2' contient ne peuvent être vérifiées, donc V_4^p doit être valuée à $\neg pertinent$; ainsi, *Voiture* sera valuée à normale suite à la propagation sur C_1. À la racine de l'arbre, nous obtenons donc l'espace possiblement vrai suivant :

$$\Diamond\mathrm{V} : \left\{ \begin{array}{ll} \textit{Voiture} & = \{\text{normale}\} \\ \textit{Vitres} & = \{\text{manuelles}\} \\ \textit{Intérieur} & = \{\text{cuir}\} \\ \textit{Toit Ouvrant} & = \{\text{automatique, manuel}\} \\ \textit{Toit Ouvrant}_p & = \{\neg pertinent\} \end{array} \right\}$$

L'arbre de la figure 6.11 illustre l'obtention de l'espace possiblement vrai, lors de la première phase de propagation. La seconde phase de propagation ne filtrera que le domaine de V_1 qui sera alors réduit à normale grâce à la contrainte C_1.

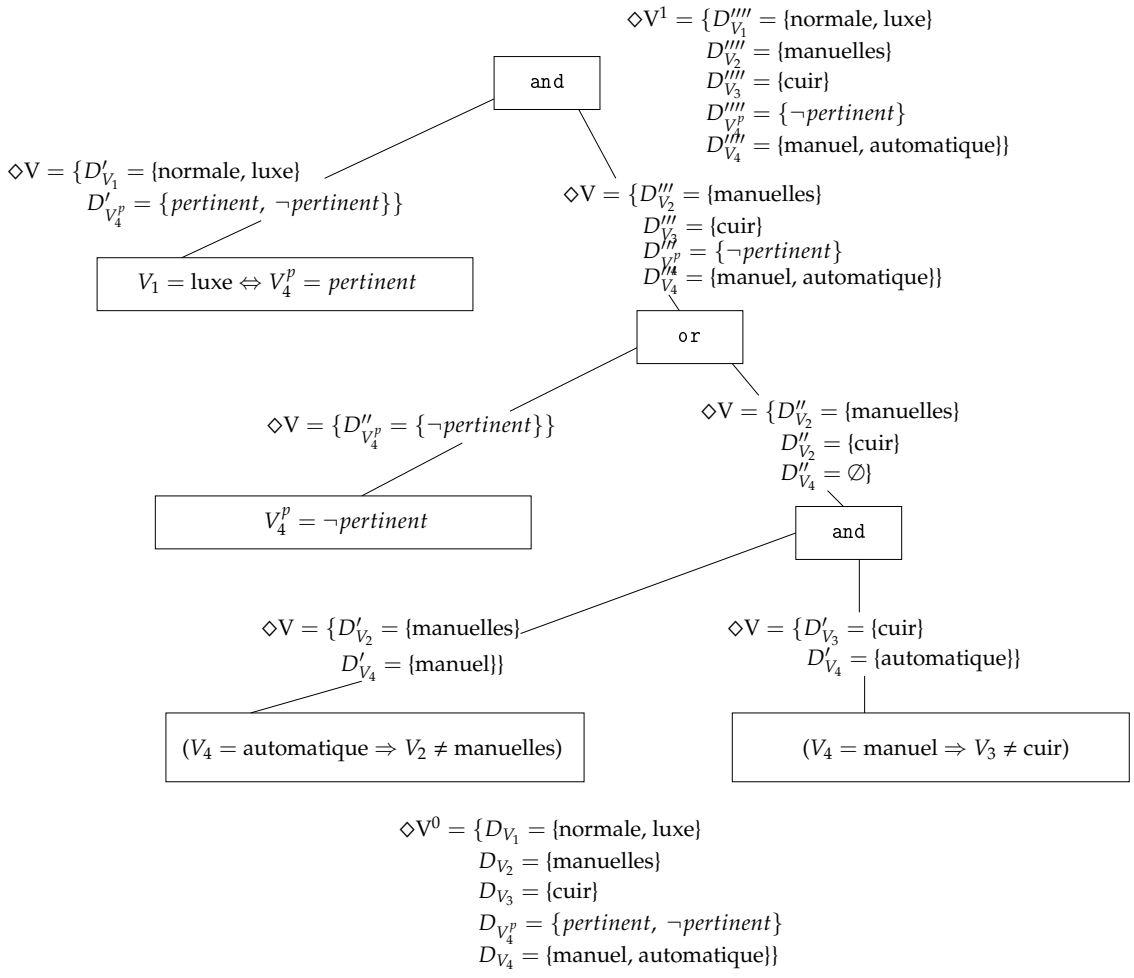


FIG. 6.11 — Arbre syntaxique — modélisation par RCSP et attribut de pertinence — traduction évoluée

Pertinence de variables continues

Pour illustrer le filtrage sur les variables numériques dont la pertinence doit être déterminée, nous utilisons l'exemple 4.6 page 55, dont la figure 6.12 reprend le schéma.

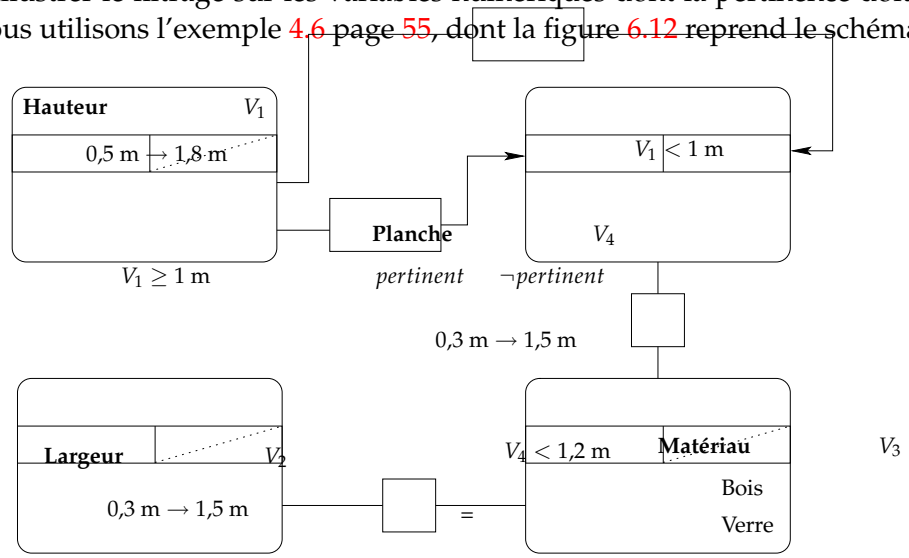


FIG. 6.12 — Configuration de meuble — modélisation par RCSP et attribut de pertinence

Pour les variables numériques, nous ne pouvons pas utiliser la modélisation avec ★ ; nous obtenons donc un RCSP dont l'équation 6.5 montre les contraintes.

$$\begin{aligned} C_1 &: \quad (V_1 \geq 1\text{m}) \Leftrightarrow (V_4 \; pertinent) \\ C_2 &: \quad V_2 = V_4 \\ C_3 &: \quad V_3 = \text{ Verre } \Rightarrow V_4 < 1,2\,\text{m} \end{aligned} \tag{6.5}$$

Cet exemple est l'équivalent de la voiture simple adapté au numérique ; en valuant la variable *Largeur* à plus d'un mètre vingt (par exemple, 1,25 m) et la variable *Matériau* à verre, la variable *Planche* devrait être obligatoirement non-pertinente et donc la hauteur être inférieure à un mètre.

La traduction directe du RCSP donne un CSP avec les contraintes suivantes :

$$\begin{aligned} C_1 &: \quad (V_1 \geq 1\,\text{m}) \Leftrightarrow (V_4^p = \; pertinent) \\ C_2 &: \quad (V_4^p = \neg pertinent) \vee V_2 = V_4 \\ C_3 &: \quad (V_4^p = \neg pertinent) \vee (V_3 = \text{ Verre } \Rightarrow V_4 < 1,2\,\text{m}) \end{aligned} \tag{6.6}$$

Avec cette formulation, de la même façon que sur l'exemple de la voiture simple, la condition sur la pertinence de la planche ne permet pas d'effectuer des déductions sur la taille du meuble. L'espace possiblement vrai à la racine de l'arbre est alors :

$$\Diamond\text{V} : \left\{ \begin{array}{l} \mathit{Hauteur} = [0,5\,;\,1,8]\ \text{m} \\ \mathit{Largeur} = 1,25\ \text{m} \\ \mathit{Matériau} = \{\text{Verre}\} \\ \mathit{Planche} = [0,5\,;\,1,8] \\ \mathit{Planche}_p = \{pertinent, \neg pertinent\} \end{array} \right\}$$

Pour permettre au moteur de déduire la non-pertinence de la planche, et donc de réduire le domaine de la hauteur, il faut utiliser la traduction factorisée. Les contraintes C_2 et C_3 de l'équation 6.5 deviennent alors C_2' (cf. équation 6.7 page suivante).

$$\begin{array}{lll} C_1 & : & (V_1 \geq 1\,\text{m}) \Leftrightarrow (V_4^p = \mathit{pertinent}) \\ C_2' & : & (V_4^p = \neg \mathit{pertinent}) \vee \\ & & \big((V_2 = V_4) \wedge (V_3 = \text{ Verre } \Rightarrow V_4 < 1{,}2\,\text{m})\big) \end{array} \tag{6.7}$$

En utilisant cette formulation, les deux valuations (*Largeur* à 1,25 m et *Matériau* à verre) permettent d'effectuer les déductions amenant à la non-pertinence de la variable *Planche* et donc à la réduction du domaine de *Hauteur*. En effet, les deux opérandes de la conjonction que C_2' contient ne peuvent être vérifiées, donc V_4^p doit être valuée à $\neg \mathit{pertinent}$; ainsi, la variable *Hauteur* sera limitée à 1 m suite à la propagation sur C_1. À la racine de l'arbre, nous obtenons donc l'espace possiblement vrai suivant :

$$\Diamond\text{V} : \left\{ \begin{array}{l} \mathit{Hauteur} = [0{,}5\,;\,1[\,\text{m} \\ \mathit{Largeur} = 1{,}25\,\text{m} \\ \mathit{Matériau} = \{\text{Verre}\} \\ \mathit{Planche} = [0{,}5\,;\,1{,}8] \\ \mathit{Planche}_p = \{\neg \mathit{pertinent}\} \end{array} \right\}$$

Il ne reste alors plus qu'un choix à l'utilisateur : la hauteur du meuble, entre 0,5 m et 1 m.

Conclusion

Par ces tests, nous justifions les préconisations de modélisation vues au chapitre 4. Pour des variables discrètes, il est plus intéressant d'ajouter la valeur ★ dans leur domaine. Pour des variables numériques, il faut utiliser l'attribut de pertinence. Si la qualité de filtrage par rapport à cette variable numérique est déterminante, il est possible d'utiliser une procédure de traduction factorisée, qui permet de filtrer correctement le problème.

Cependant, la procédure de traduction évoluée demande une étape de factorisation de certains éléments des formules logiques. Il faut donc dans un premier temps pouvoir identifier quels sont les éléments à factoriser. C'est le point difficile de cette traduction factorisée, et elle demande l'intervention du modeleur.

Nous venons de voir que la manière de décrire les contraintes a une grande influence sur la qualité du filtrage. Par conséquent, le modeleur joue un rôle crucial : c'est lui qui doit adapter la forme des contraintes du modèle pour le modèle RCSP.

6.3.2 Méta-informations

Ce type d'informations permet au moteur de prendre en compte son état afin de piloter son fonctionnement. Nous avons déjà introduit l'opérateur *Sure* et son fonctionnement général au chapitre 4. Cet opérateur permet de prendre en compte l'état du filtrage pour remplacer certaines parties du problème par des sous-problèmes.

Méta-opérateur *Sure*

Nous rappelons le fonctionnement général du méta-opérateur *Sure* : le nœud *Sure* se compose de deux opérandes : une condition et une formule. Lorsque la condition est nécessairement vérifiée (son espace possiblement faux connaissant les domaines des variables est vide) alors la formule remplace la condition dans le problème. L'algorithme 6.13 page suivante montre les grandes étapes du filtrage du méta-opérateur *Sure*.

Alg. 6.13 FILTRAGE DU NŒUD *Sure*

Si (les variables de la condition ont été filtrées) **Alors**
  Remettre à jour les espaces possiblement vrai et possiblement faux de la condition.
  Si (la condition est toujours fausse) **Alors**
    le nœud est toujours vrai
  Sinon Si (la condition est toujours vraie) **Alors**
    le nœud devient la formule
  Sinon
    les espaces possiblement vrai et possiblement faux du nœud sont ceux de la condition
  Fin Si
Fin Si

La figure 6.14 illustre la forme d'un nœud *Sure*. Avant que la condition ne devienne toujours vraie ou toujours fausse, les espaces possiblement vrai et possiblement faux du nœud *Sure* sont égaux à ceux de la condition, et la formule n'intervient pas dans le filtrage.

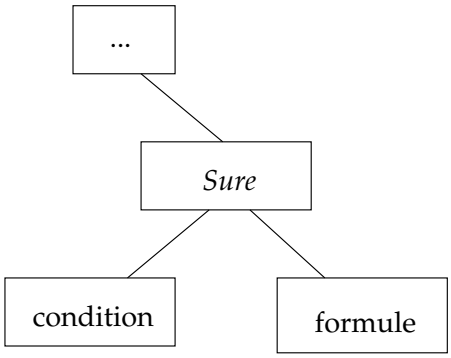
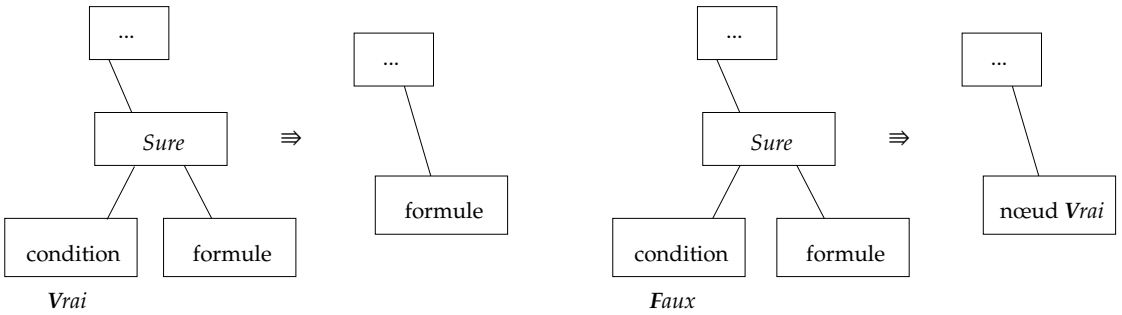


FIG. 6.14 — Extrait d'arbre syntaxique — nœud *Sure*

Lorsque les domaines des variables de la condition ont été suffisament filtrés pour que la condition soit nécessairement vraie ou nécessairement fausse, le nœud *Sure* change alors d'espaces possiblement vrai et possiblement faux :

- si la condition devient nécessairement vraie (son espace possiblement faux est alors vide), les espaces possiblement vrais et possiblement faux du nœud *Sure* deviennent alors les espaces possiblement vrai et possiblement faux de la formule ;
- si la condition devient nécessairement fausse (son espace possiblement vrai est alors vide), les espaces possiblement vrais et possiblement faux du nœud *Sure* deviennent alors les domaines initiaux.

Enfin, il est aussi possible de faire appel à l'élagage pour le nœud *Sure*. Lorsque la condition est nécessairement vraie, c'est alors la formule qui remplace le nœud *Sure*, comme le montre la figure 6.15 page suivante. Lorsque la condition est nécessairement fausse, le nœud *Sure* est remplacé par un nœud ***Vrai***, et celui-ci peut alors provoquer d'autres simplifications plus près de la racine de l'arbre.



(a) La condition est nécessairement vraie (b) La condition est nécessairement fausse

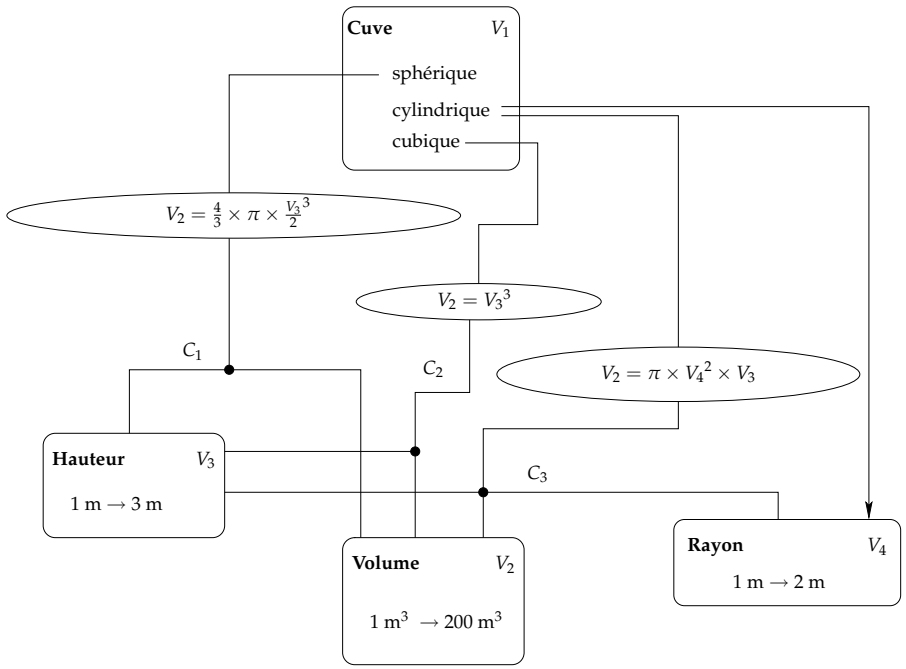
FIG. 6.15 — Formes filtrées du nœud *Sure* avec élagage

Résultats sur un exemple numérique — pertinence de contraintes

L'exemple 6.16 peut être modélisé de plusieurs façons. Le modeleur peut décider :
- que le filtrage doit pouvoir être effectué même sans connaître la forme de la cuve ;
- que les calculs de volumes sont complexes et qu'il faut donc éviter de les propager inutilement (en utilisant un méta-opérateur).

EX. 6.16 — Exemple du réacteur chimique

Cet exemple est une adaptation simplifiée de l'exemple du *mixer* de GELLE (1998). Un mélangeur chimique est composé d'une cuve, dont le volume varie en fonction de la géométrie (en particulier la formule de calcul).
La figure ci-dessous illustre les contraintes selon la forme de la cuve.



Les deux points de vue que peut adopter le modeleur correspondent à deux modélisations des contraintes différentes :

– la première modélisation regroupe deux formulations différentes :

$$\begin{cases} - \text{ première formulation, une seule contrainte} \\ \qquad (V_1 = \text{ cube } \wedge (V_2 = V_3^3)) \quad \vee \\ \qquad (V_1 = \text{ sphere } \wedge (V_2 = \frac{4}{3} \times \pi \times \left(\frac{V_3}{2}\right)^3)) \quad \vee \\ \qquad (V_1 = \text{ cylindre } \wedge (\exists V_4 \in [1,2],\ V_2 = \pi \times V_4^2 \times V_3)) \\ \\ - \text{ deuxième formulation, trois contraintes conditionnelles} \\ \qquad (V_1 = \text{ cube }) \Rightarrow (V_2 = V_3^3) \\ \qquad (V_1 = \text{ sphere }) \Rightarrow (V_2 = \frac{4}{3} \times \pi \times \left(\frac{V_3}{2}\right)^3) \\ \qquad (V_1 = \text{ cylindre }) \Rightarrow (\exists V_4 \in [1,2],\ V_2 = \pi \times V_4^2 \times V_3) \end{cases}$$

– la deuxième modélisation permet une troisième formulation ; trois contraintes, avec l'utilisation de méta-opérateurs :

$$\begin{cases} \mathit{Sure}\,(V_1 = \text{ cube }, \{V_2 = V_3^3\}) \\ \mathit{Sure}\,(V_1 = \text{ sphere }, \{V_2 = \frac{4}{3} \times \pi \times \left(\frac{V_3}{2}\right)^3\}) \\ \mathit{Sure}\,(V_1 = \text{ cylindre }, \{\exists V_4 \in [1,2],\ V_2 = \pi \times V_4^2 \times V_3\}) \end{cases}$$

Ainsi, avec la première formulation, le moteur sera capable de déduire sans contrainte supplémentaire que le volume maximal (200 m^3) ne peut pas être atteint (le maximum est atteint pour une cuve cylindrique de 3 m de hauteur et 2 m de rayon avec environ $37,7$ m^3). La deuxième formulation est un hybride entre les deux autres ; les calculs sont effectués, mais la formulation logique employée ne permet pas de faire de déductions sur le volume maximal de la cuve. En revanche, avec la troisième formulation, le moteur attend la valuation de la variable *Cuve* avant d'effectuer un calcul quelconque.

Les résultats du tableau 6.17 page suivante illustrent bien les avantages et les inconvénients du méta-opérateur *Sure*. Dans les trois cas, l'avantage est de diminuer le temps de filtrage par rapport à la formulation par contraintes conditionnelles, grâce au fait qu'aucun calcul n'est effectué tant que la variable *Cuve* n'est pas valuée. Par rapport à la formulation avec une seule contrainte, nous économisons des phases de réinjection.

L'inconvénient est que le domaine de la variable *Volume* n'est pas filtré, ce qui peut entraîner des valuations incohérentes de l'utilisateur.

Nous constatons dans la deuxième partie du tableau que la modélisation par méta-opérateur reste plus rapide, ceci parce que les deux autres formulations doivent filtrer les formes cubique et sphérique (contrairement à la formulation par méta-opérateur).

Lorsque l'utilisateur impose un volume supérieur à 35 m^3, les formulations sans méta-opérateurs déduisent que la géométrie de la cuve ne peut être que cylindrique (troisième partie du tableau). Par ailleurs, nous distinguons une différence en temps de calcul et en nombres de réinjections entre la formulation avec une seule contrainte (respectivement 3 réinjections et $0,59$ s) et les trois contraintes conditionnelles (*resp.* 5 réinjections et $0,67$ s). Avec un méta-opérateur, aucun calcul n'étant effectué si V_1 n'est pas valuée, le moteur ne peut filtrer le domaine de V_1.

Ainsi, la façon de modéliser ce type de problème doit prendre en compte l'utilisation que l'on compte en faire : si l'utilisateur doit pouvoir contraindre le volume, il faut utiliser

TAB. 6.17 — Résultats en filtrage numérique selon la formulation

Formulation	Nombres de réinjections	Temps CPU	Domaine de V_2
sans valuations			
une seule contrainte	3	0,65 s	$[1; 37, 70]$
contraintes conditionnelles	1	0,45 s	$[1; 200]$
avec un méta-opérateur	1	0,31 s	$[1; 200]$
avec la valuation : $V_1 =$ cylindre			
une seule contrainte	4	0,62 s	$[3, 14; 37, 70]$
contraintes conditionnelles	4	0,58 s	$[3, 14; 37, 70]$
avec un méta-opérateur	4	0,37 s	$[3, 14; 37, 70]$
avec la contrainte : $V_2 > 35$			Domaine de V_1
une seule contrainte	3	0,59 s	cylindre
contraintes conditionnelles	5	0,67 s	cylindre
avec un méta-opérateur	2	0,31 s	cube, cylindre, sphère

la modélisation sans méta-opérateur. Si l'utilisateur doit choisir avant tout la forme de la cuve, il est préférable d'utiliser un méta-opérateur.

L'utilisation d'un méta-opérateur peut être particulièrement intéressante pour plusieurs aspects :

- en cas de contraintes numériques lourdes : nous l'avons évoqué sur l'exemple de la cuve (nous pouvons retrouver ce cas sur des *quad-trees*). Dans le cas de formules de calculs très complexes, l'ajournement de ces calculs par la condition de connaître le calcul à utiliser peut faire gagner un temps précieux et garantit l'interactivité de l'outil ;
- pour l'utilisation d'un moteur externe : en utilisant un méta-opérateur, il est alors possible d'attendre que des variables remplissent une condition avant d'appeler un moteur externe effectuant des calculs complexes sur ces variables (calcul de structure, de mécanique des fluides...).

Discussion et perspectives d'implémentation

Nous avons vu l'interêt du méta-opérateur *Sure* pour maintenir l'interactivité lors de la configuration. Son deuxième avantage est de pouvoir attendre que certaines conditions soient remplies pour lancer un module de calcul externe. Pour ces modules externes, les conditions à remplir peuvent être : disposer de variables valuées pour lancer le calcul.

Nous proposons sur le même principe l'utilisation de méta-attributs pour vérifier que les variables ne puissent prendre qu'une seule valeur. Dans les espaces associés au nœud, ceci revient à ce que le domaine associé aux variables en question soit réduit à un singleton. Le méta-attribut permettant d'exprimer cette condition sera donc `is_singleton` et retournera un booléen, vrai si le domaine est réduit à un singleton, faux sinon. Nous pouvons aussi concevoir des méta-informations autour de l'arité des domaines.

La combinaison de méta-attributs et de méta-opérateurs peut ainsi nous permettre d'attendre que certaines variables soient valuées (ou réduites à un singleton) pour faire appel à des applications externes pour le filtrage d'une partie du problème.

6.4 Conclusion

Dans ce chapitre, nous avons présenté les améliorations et adaptations que nous avons apportées à notre implémentation pour permettre une meilleure exploitation des RCSP. Ces améliorations concernent aussi bien le langage (attributs de variables et groupes) que la résolution (modification de l'algorithme *BackTrack* et différentes heuristiques) ou le filtrage (élagage, méta-opérateurs et méta-attributs).

L'implémentation basée sur des arbres syntaxiques et les RCSP est intéressante en termes de filtrage : cela nous permet d'utiliser des contraintes très particulières tout en maintenant l'interactivité et en gardant une qualité de filtrage satisfaisante.

Cependant, un aspect n'est pas encore traité grâce à notre implémentation : les feuilles numériques indirectes ne permettent pas encore un filtrage efficace.

Enfin, le modeleur a la lourde responsabilité d'exprimer les contraintes sous la meilleure forme possible. Cela lui laisse le choix de la formulation, mais il doit posséder une bonne connaissance du modèle pour obtenir une meilleure qualité de filtrage tout en maintenant l'interactivité.

Conclusion générale

Synthèse

Dans le premier chapitre, nous situons le cadre de nos travaux dans les domaines de la conception et de la configuration. Plus précisément, nos travaux ont pour objectif de fournir des outils d'aide à la décision dans ces deux domaines. Nous nous positionnons sur les approches par contraintes, puis exprimons les besoins suivants :

- traiter des éléments de différentes nature (composants ou paramètres discrets et/ou numériques) ;
- pouvoir exprimer des hiérarchies entre ces éléments et ainsi pouvoir en regrouper certains au sein d'une même entité ;
- gérer la pertinence de ces éléments ou sous-ensembles d'éléments : certains ne font alors pas partie de la solution du problème de conception ou de configuration.

Ces besoins nous conduisent à la problématique d'intégration de différentes approches au sein d'un même formalisme.

Dans le deuxième chapitre, nous passons en revue différentes techniques de la littérature permettant de traiter les CSP discrets et continus. Nous proposons ainsi un rapide panorama des différentes méthodes de résolution de CSP— pour un processus d'aide à la conception ou à la configuration autonome — et des méthodes de filtrage — pour un processus interactif, où l'utilisateur doit faire des choix.

Ensuite, le troisième chapitre se consacre plus spécifiquement à la gestion de la pertinence d'éléments pouvant être organisés hiérarchiquement dans un CSP. Cet état de l'art souligne le peu de travaux dans le domaine et le fait qu'aucune approche ne couvre tous les besoins.

Nos premières propositions concernent la modélisation de problèmes de conception ou de configuration afin d'effectuer de l'aide à la décision. Nous proposons le modèle RCSP, qui complète les CSP mixtes par des mécanismes de gestion de la pertinence ou de la hiérarchie d'éléments du modèle. De plus, nous établissons des préconisations de modélisation

pour les différents éléments d'un problème de conception ou de configuration, ainsi que des procédures de traduction de RCSP en CSP permettant d'utiliser des techniques existantes de résolution ou de filtrage.

Nous présentons ensuite notre implémentation permettant le traitement de CSP. Cette implémentation est composée de trois éléments : un langage de description de CSP, un analyseur du langage qui produit des arbres syntaxiques et deux moteurs (résolution et filtrage) qui reposent sur l'interprétation des arbres syntaxiques.

Pour terminer, nous proposons des enrichissements du langage de description qui permettent de mieux décrire certaines particularités des RCSP. Le moteur de résolution est amélioré grâce à la prise en compte de spécificités des RCSP par des heuristiques et une modification de l'algorithme de *BackTrack*. Le moteur de filtrage permet d'utiliser le nouveau méta-opérateur *Sure*, qui permet de gérer le compromis entre qualité du filtrage et temps de réponse.

Notre proposition d'implémentation permet de traiter des CSP mixtes grâce aux arbres syntaxiques. Ces arbres syntaxiques permettent l'expression des contraintes les plus diverses. Associée au modèle RCSP, cette implémentation permet d'aider à la conception ou à la configuration de produits, services ou procédés.

Le modeleur dispose ainsi d'un modèle lui permettant de décrire des problèmes de conception ou configuration et de plusieurs formulations pour privilégier la qualité des déductions ou les performances (en termes de temps de calcul). Les différents moteurs proposent alors une assistance utile au concepteur lors de ses prises de décision en ne lui proposant que des choix pouvant encore mener à des solutions cohérentes avec le modèle du produit, service ou procédé.

Perspectives de recherches

Filtrage numérique

En ce qui concerne le filtrage des contraintes numériques indirectes, BENHAMOU *et al.* (1999) ont proposé des méthodes dont l'adaptation aux arbres syntaxiques mériterait d'être étudiée. Lors de cette adaptation, il sera aussi nécessaire de prendre en compte les problèmes de stabilité numérique et de propagation d'erreurs.

Qualité de la formulation

La définition du modèle comprenant l'expression des contraintes est la tâche du modeleur. Notre moteur reposant sur l'interprétation des contraintes, leur formulation a un gros impact sur la qualité et sur les performances du moteur. Les préconisations de modélisation que nous avons établies au chapitre 4 et les résultats du chapitre 6 permettent d'orienter les choix de formulation du modeleur. Des procédures de traduction automatisée pourraient éventuellement être mises en œuvre, en utilisant les formes factorisées.

Multi-occurences et pertinence

Certains aspects pour la conception n'ont pas été traités : la multi-occurence de composants et ses conséquences sur la pertinence. Plusieurs questions se posent alors sur la prise

en compte de ces éléments et leur pertinence. Nous devrions pouvoir prendre en compte une contrainte globale portant sur un ensemble d'éléments dont certains peuvent être pertinents ou non (par exemple, la somme des poids des éléments). Nous devrions pouvoir gérer un nombre variable d'occurences d'un élément et pouvoir poser des contraintes portant sur l'ensemble de ces éléments. De plus, des contraintes d'interaction peuvent exister entre les différentes occurences d'un élément ; il faut aussi pouvoir les prendre en compte.

Autres implémentations

Pour terminer, nous identifions de futurs travaux pour tester le formalisme RCSP en utilisant d'autres moteurs de propagation de contraintes, par exemple ILOG SOLVER, ECLiPSe, COPRIN, etc. L'implémentation que nous avons créée est adaptée aux RCSP, mais un moteur de propagation de contraintes plus efficace pourrait obtenir de meilleurs résultats.

Références bibliographiques

A

A. AAMODT & E. PLAZA. Case-Based Reasoning : Foundational Issues, Methodological Variations, and System Approaches. *AI Communications*, 7(1):39–52, 1994.

AMERICAN ASSOCIATION FOR ARTIFICIAL INTELLIGENCE, *coord*. *AAAI-90, Proceedings of the Eighth National Conference on Artificial Intelligence*, Boston, Massachusetts, USA, juillet 1990. AAAI Press / The MIT Press. ISBN 0-262-51057-X.

AMERICAN ASSOCIATION FOR ARTIFICIAL INTELLIGENCE, *coord*. *AAAI-96, Proceedings of the 13th National Conference on Artificial Intelligence*, vol. 1, Portland, Oregon, USA, août 1996. AAAI Press / The MIT Press. ISBN 0-262-51091-X.

Jérôme AMILHASTRE. *Représentation par automate d'ensemble de solutions de problèmes de satisfaction de contraintes*. Thèse de doctorat, Université Montpellier II, Montpellier, France, janvier 1999.

Jérôme AMILHASTRE, Hélène FARGIER & Pierre MARQUIS. Consistency Restoration and Explanations in Dynamic CSPs – Application to Configuration. *Artificial Intelligence*, 135, 2002.

Krzysztof R. APT. The essence of constraint propagation. *Theor. Comput. Sci.*, 221(1-2):179–210, 1999.

B

Ruzena BAJCSY, *coord*. *IJCAI-93, Proceedings of the 13th International Joint Conference on Artificial Intelligence*, Chambéry, France, août 1993. Morgan Kaufmann. ISBN 1-55860-300-X.

Roman BARTÁK. Constraint Propagation, 1998.

Heikel BATNINI. *Contraintes globales et heuristiques de recherche pour les CSPs continus*. Thèse de doctorat, Université de Nice, Sophia Antipolis, décembre 2005.

Frédéric BENHAMOU, Frédéric GOUALARD, Laurent GRANVILLIERS & Jean-Francois PUGET. Revising Hull and Box Consistency. *In* DE SCHREYE (1999), pages 230–244. ISBN 0-262-54104-1.

Frédéric BENHAMOU, David A. MCALLESTER & Pascal VAN HENTENRYCK. CLP(Intervals) Revisited. *In* BRUYNOOGHE (1994), pages 124–138. ISBN 0-262-52191-1.

P. BERLANDIER. Improving Domain Filtering using Restricted Path Consistency. *In* CONFERENCE ON ARTIFICIAL INTELLIGENCE FOR APPLICATIONS (1995), pages 32–37.

A. BERNARD. Modèles et approches pour la conception et la production intégrée. *APII — Journal Européen des Systèmes Automatisés*, 34:163–193, 2000.

C. BESSIÈRE, E. C. FREUDER & J. C. RÉGIN. Using interference to reduce arc-consistency computation. *In* INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE (1995).

C. BESSIÈRE, P. MESEGUER, E.C. FREUDER & J. LARROSA. On Forward Checking for Nonbinary Constraint Satisfaction. *In* JAFFAR (1999), pages 88–102. ISBN 3-540-66626-5.

Christian BESSIÈRE. Arc-Consistency and Arc-Consistency Again. *Artificial Intelligence*, 65 (1):179–190, Janvier 1994.

Christian BESSIÈRE & Jean-Charles RÉGIN. MAC and Combined Heuristics : Two Reasons to Forsake FC (and CBJ ?) on Hard Problems. *In* FREUDER (1996), pages 61–75.

Christian BESSIÈRE & Jean-Charles RÉGIN. Refining the Basic Constraint Propagation Algorithm. *In* NEBEL (2001), pages 309–315. ISBN 1-55860-777-3.

D. C. BROWN & B. CHANDRASEKARAN. *Design problem solving : knowledge structures and control strategies*. Morgan Kaufmann Publishers Inc., San Francisco, Californie, USA, 1989.

Maurice BRUYNOOGHE, *coord*. *Proceedings of the 1994 International Symposium on Logic Programming*, Ithaca, New York, USA, novembre 1994. MIT Press. ISBN 0-262-52191-1.

C

Jacques CALMET, Belaid BENHAMOU, Olga CAPROTTI, Laurent HENOCQUE & Volker SORGE, *coord*. *AISC-02, Artificial Intelligence, Automated Reasoning, and Symbolic Computation, Joint International Conferences*, vol. 2385 de *Lecture Notes in Computer Science*, Marseille, France, juillet 2002. Springer. ISBN 3-540-43865-3.

B. CHANDRASEKARAN. Design Problem Solving : a Task Analysis. *Artificial Intelligence Magazine*, 11:59–71, 1990.

Peter CLARK & Rob HOLTE. Generalised Backjumping. Technical Report TR-92-20, Univ. Ottawa, Ottawa, Canada, 1992.

CONFERENCE ON ARTIFICIAL INTELLIGENCE FOR APPLICATIONS, *coord*. *CAIA-95, Proceedings of the 11th Conference on Artificial Intelligence for Applications*, Los Angeles, Californie, USA, 1995. IEEE.

CONFIGIT. CLib : Configuration Benchmarks Library.

D

Danny DE SCHREYE, *coord*. *ICLP-99, Proceedings of the 16th International Conference on Logic Programming*, Las Cruces, Nouveau-Mexique, USA, décembre 1999. The MIT Press. ISBN 0-262-54104-1.

Romuald DEBRUYNE & Christian BESSIÈRE. From restricted path consistency to max-restricted path consistency. *In* SMOLKA (1997), pages 312–326.

Romuald DEBRUYNE & Christian BESSIÈRE. Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. *In* INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE (1997), pages 412–417.

Rina DECHTER. Enhancement Schemes for Constraint Processing : Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence*, 41(3):273–312, janvier 1990.

Rina DECHTER. Backtracking Algorithms for Constraint Satisfaction Problems — a survey, janvier 1997.

F. DELOBEL. *Résolution de contraintes réelles non-linéaires*. Thèse de doctorat, Université de Nice — Sophia Antipolis, janvier 2000.

E

Mathieu ESTRATAT & Laurent HENOCQUE. Parsing Languages with a Configurator. *In* LÓPEZ DE MÁNTARAS & SAITTA (2004), pages 591–595. ISBN 1-58603-452-9.

F

Hélène FARGIER & Laurent HENOCQUE. Configuration à base de contraintes. Rapport Technique 0010.2002, Laboratoire des Sciences de l'Information et des Systèmes, LSIS, (UMR CNRS 6168), 2002.

E. FREUDER. A Sufficient Condition for Backtrack-Bounded Search. *Journal of the ACM*, 32 (4):755–761, 1985.

Eugene C. FREUDER, *coord*. *CP-96, Second International Conference on Principles and Practice of Constraint Programming*, vol. 1118 de *Lecture Notes in Computer Science*, Cambridge, Massachusetts, USA, août 1996. Springer.

Eugene C. FREUDER & Charles D. ELFE. Neighborhood Inverse Consistency Preprocessing. *In* AMERICAN ASSOCIATION FOR ARTIFICIAL INTELLIGENCE (1996), pages 202–208. ISBN 0-262-51091-X.

G

James GARSON. Modal Logic. *In* ZALTA (2005).

John GASCHNIG. *Performance Measurement and Analysis of Certain Search Algorithms*. PhD Thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvanie, USA, mai 1979.

P. A. GEELEN. Dual ViewPoint Heuristics for Binary Constraint Satisfaction Problems. *In* NEUMANN (1992), pages 31–35.

Esther GELLE. *On the generation of locally consistent solution spaces in mixed dynamic constraint problems*. Thèse de doctorat, École Polytechnique Fédérale de Lausanne, Lausanne, Suisse, 1998.

Esther GELLE & Boi FALTINGS. Solving Mixed and Conditional Constraint Satisfaction Problems. *Constraints*, 8:107–141, 2003.

Esther GELLE & Mihaela SABIN. Solving Methods for Conditional Constraint Satisfaction. *In* GOTTLOB & WALSH (2003).

Felix GELLER & Michael VEKSLER. Assumption-Based Pruning in Conditional CSP. *In* VAN BEEK (2005), pages 241–255.

Solomon W. GOLOMB & Leonard D. BAUMERT. Backtrack Programming. *Journal of the ACM*, 12(4):516–524, 1965.

Georg GOTTLOB & Toby WALSH, *coord*. *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, Acapulco, Mexique, août 2003. Morgan Kaufmann.

Laurent GRANVILLIERS & Barry O'SULLIVAN, *coord*. *Proceedings of the First International Workshop on Constraints and Design*, Sitges (Barcelone), Espagne, octobre 2005.

H

Khaled HADJ-HAMOU. *Contribution à la conception de produits à forte diversité et de leur chaîne logistique : une approche par contraintes*. Thèse de doctorat, Institut National Polytechnique de Toulouse, École des Mines d'Albi-Carmaux, décembre 2002.

Khaled HADJ-HAMOU. Approches par contraintes : du discret vers le numérique, juin 2003.

R. M. HARALICK & G. L. ELLIOTT. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.

Laurent HENOCQUE, *coord*. *JFPC-06, Actes des Deuxièmes Journées Francophones de Programmation par Contraintes*, Nîmes, France, juin 2006. AFPC, École des Mines d'Alès.

I

INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, *coord*. *IJCAI-95, Proceedings of the 14th International Joint Conference on Artificial Intelligence*, vol. 1, Montréal, Québec, août 1995. Morgan Kaufmann.

INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, *coord*. *IJCAI-97, Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, Nagoya, Japon, août 1997. Morgan Kaufmann.

J

Joxan JAFFAR, *coord*. *CP-99, Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming*, vol. 1713 de *Lecture Notes in Computer Science*, Alexandria, Virginie, USA, octobre 1999. Springer. ISBN 3-540-66626-5.

K

Yves KODRATOFF, *coord*. *ECAI-88, Proceedings of the 8th European Conference on Artificial Intelligence*, Münich, Allemagne, août 1988. Pitmann Publishing. ISBN 0-273-08798-3.

Janet KOLODNER. *Case-Based Reasoning*. Morgan Kaufmann Publishers Inc., San Francisco, Californie, USA, 1993. ISBN 1-55860-237-2.

S. A. KRIPKE. A Completeness Theorem in Modal Logic. *Journal of Symbolic Logic*, 24:1–14, 1959.

Saul KRIPKE. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

L

E. LEMMON & D. SCOTT. *An Introduction to Modal Logic*. Oxford : Blackwell, 1977.

Olivier LHOMME. Consistency Techniques for numeric CSP. *In* BAJCSY (1993), pages 232–238. ISBN 1-55860-300-X.

Olivier LHOMME & Michel RUEHER. Application des techniques CSP au raisonnement sur les intervalles. *Revue d'Intelligence Artificielle*, 11(3):283–311, 1997.

Ramon LÓPEZ DE MÁNTARAS & Lorenza SAITTA, *coord*. *ECAI-04, Proceedings of the 16th European Conference on Artificial Intelligence*, Valence, Espagne, août 2004. IOS Press. ISBN 1-58603-452-9.

C. LOTTAZ, D. CLEMENT, B. FALTINGS & I. SMITH. Constraint Based Support for Collaboration in Design and Construction. *Journal of Computing in Civil Engineering*, 13(1):23–35, 1999.

M

A.K. MACKWORTH & E.C. FREUDER. The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. *Artificial intelligence*, 25:65–74, 1985.

Alan MACKWORTH. Consistency in networks of relations. *Artificial intelligence*, 8:99–118, 1977.

Daniel MAILHARRO. A classification and constraint-based framework for configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 12(4):383–397, 1998.

Kevin MCDONALD & Patrick PROSSER. A case study of constraint programming for configuration problems. Technical Report APES-45-2002, APES Research Group, mars 2002.

Sanjay MITTAL & Brian FALKENHAINER. Dynamic Constraint Satisfaction Problem. *In* AMERICAN ASSOCIATION FOR ARTIFICIAL INTELLIGENCE (1990), pages 25–32. ISBN 0-262-51057-X.

Sanjay MITTAL & Felix FRAYMAN. Toward a Generic Model of Configuration Tasks. *In* SRIDHARAN (1989). ISBN 1-55860-094-9.

Martin G. MOEHRLE. What is TRIZ ? From Conceptual Basics to a Framework for Research. *Creativity and Innovation Management*, 14(1):3, mars 2005.

R. MOHR & T.C. HENDERSON. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28: 225–233, 1986.

Roger MOHR & Gérald MASINI. Good Old Discrete Relaxation. *In* KODRATOFF (1988), pages 651–656. ISBN 0-273-08798-3.

Ugo MONTANARI. Networks of constraints : Fundamental properties and application to picture processing. *Information sciences*, 7:95–132, 1974.

Ramon E. MOORE. *Interval Analysis*. Prentice Hall, 1966.

Taufiq MULYANTO. *Utilisation des techniques de programmation par contraintes pour la conception d'avions*. Thèse de doctorat, École Nationale Supérieure de l'Aéronautique et de l'Espace, Toulouse, France, 2002.

N

Bernhard NEBEL, *coord*. *IJCAI-01, Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, Seattle, Washington, USA, août 2001. Morgan Kaufmann. ISBN 1-55860-777-3.

A. NEUMAIER. Interval Methods for System Equations. *In Encyclopedia of Mathematics and its Applications*, vol. 37. 1990.

Bernd NEUMANN, *coord*. *ECAI-92, Proceedings of the 10th European Conference on Artificial Intelligence*, Vienne, Autriche, août 1992. John Wiley and Sons.

P

G. PAHL & W. BEITZ. *Engineering Design : a Systematic Approach*. Springer-Verlag, Londres, Royaume-Uni, 2e édition, 1996.

Patrick PROSSER. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9(3):268–299, 1994.

R

Stefan RATSCHAN. Continuous first-order constraint satisfaction. *In* CALMET *et al.* (2002), pages 181–195. ISBN 3-540-43865-3.

James RUMBAUGH, Ivar JACOBSON & Grady BOOCH. *UML 2.0 Guide de référence*. Campus Press, 2004. ISBN 2-7440-1820-1.

Jean-Charles RÉGIN. AC-* : A Configurable, Generic and Adaptive Arc Consistency Algorithm. *In* VAN BEEK (2005), pages 505–519.

S

Daniel SABIN & Eugene C. FREUDER. Configuration as Composite Constraint Satisfaction. *In Proceedings of the First Artificial Intelligence and Manufacturing Research Planning Workshop*, 1996.

M. SAGAUT, *coord*. *La programmation par contraintes*, Chapitre 4 — Contraintes et Problèmes à Satisfaction de Contraintes. Agence pour la Diffusion de l'Information Technologique, Avril 1996.

Djamila SAM. *Constraint Consistency Techniques for Continuous Domains*. PhD Thesis, École Polytechnique Fédérale, Lausanne, Suisse, 1995.

Hanan SAMET. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

Rony SHAPIRO, Yishai A. FELDMAN & Rina DECHTER. On the Complexity of Interval-Based Constraint Networks. Rapport Technique r76, University of California, Irvine, 1998.

Barbara M. SMITH. A Tutorial on Constraint Programming. Technical report, University of Leeds, 1995.

Gert SMOLKA, *coord*. *CP-97, Third International Conference on Principles and Practice of Constraint Programming*, vol. 1330 de *Lecture Notes in Computer Science*, Linz, Autriche, octobre 1997. Springer.

Timo SOININEN & Esther GELLE. Dynamic Constraint Satisfaction in Configuration. *In AAAI'99, Workshop on Configuration*, pages 95–100, Orlando, Floride, USA, 1999.

Timo SOININEN, Esther GELLE & Ilkka NIEMELÄ. A Fixpoint Definition of Dynamic Constraint Satisfaction. *In* JAFFAR (1999), pages 419–433. ISBN 3-540-66626-5.

N. S. SRIDHARAN, *coord*. *IJCAI-89, Proceedings of the 11th International Joint Conference on Artificial Intelligence*, Detroit, Michigan, USA, août 1989. Morgan Kaufmann. ISBN 1-55860-094-9.

R. M. STALLMAN & G. S. SUSSMAN. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, octobre 1977.

William R. SWARTOUT, *coord*. *AAAI-92, Proceedings of the 10th National Conference on Artificial Intelligence*, San Jose, Californie, USA, juillet 1992. AAAI Press / The MIT Press. ISBN 0-262-51063-4.

T

H. TARDIEU, A. ROCHFELD & R. COLLETTI. *La méthode Merise — principes et outils*. Éditions d'Organisation, 2e édition, 2000. ISBN 2-7081-2473-0.

M. TOLLENAERE. Contribution à la modélisation de connaissances pour la conception mécanique. HDR, Université Joseph Fourier, Grenoble, France, 1994.

Edward TSANG. *Foundations of Constraint Satisfaction*. Academic Press, Londres (UK) et San Diego (USA), 1993. ISBN 0-12-701610-4.

V

Peter VAN BEEK, *coord*. *CP-05, Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming*, vol. 3709 de *Lecture Notes in Computer Science*, Sitges (Barcelone), Espagne, octobre 2005. Springer.

P. VAN HENTENRYCK, Y. DEVILLE & C.-M. TENG. A Generic Arc-Consistency Algorithm and its Specializations. *Artificial Intelligence*, 57:291–321, 1992.

Thomas VAN OUDENHOVE. Configuration, CSP et base de connaissances. Rapport de DEA, École des Mines d'Albi-Carmaux, Albi, France, septembre 2003.

Thomas VAN OUDENHOVE. Configuration et CSP avec variables à existence conditionnée. *In Congrès des doctorants EDSYS*, mai 2005.

Thomas VAN OUDENHOVE, Paul GABORIT, Michel ALDANONDO & Élise VAREILLES. CSP dynamiques en configuration. *In* HENOCQUE (2006), pages 397–404.

Thomas VAN OUDENHOVE DE SAINT GÉRY, Paul GABORIT & Michel ALDANONDO. A Specificity of CSP in Design : Controlling the Relevance of the Variables in the Problem. *In* GRANVILLIERS & O'SULLIVAN (2005), pages 34–41.

M. VAN VELZEN. A Piece of CAKE, Computer Aided Knowledge Engineering on KADSified Configuration Tasks. Master's thesis, University of Amsterdam, Amsterdam, Pays-Bas, 1993.

Élise VAREILLES. *Conception et approches par propagation de contraintes : contribution à la mise en œuvre d'un outil d'aide interactif*. Thèse de doctorat, Institut National Polytechnique de Toulouse, École des Mines d'Albi-Carmaux, juin 2005.

Nageshwara Rao VEMPATY. Solving Constraint Satisfaction Problems Using Finite State Automata. *In* SWARTOUT (1992), pages 453–458. ISBN 0-262-51063-4.

Gérard VERFAILLIE & Narendra JUSSIEN. Constraint Solving in Uncertain and Dynamic Environments : A Survey. *Constraints*, 10(3):253–281, juillet 2005.

Yoann VERNAT. *Formalisation et qualification de modèles par contraintes en conception préliminaire*. Thèse de doctorat, École Nationale des Arts et Métiers, Bordeaux, France, novembre 2004.

Mathieu VERON. *Modélisation et résolution du problème de configuration industrielle : utilisation des techniques de satisfaction de contraintes*. Thèse de doctorat, Institut National Polytechnique de Toulouse, ENI Tarbes, novembre 2001.

Mathieu VERON & Michel ALDANONDO. Yet Another Approach to CCSP for configuration problem. *In ECAI'00, European Conference on Artificial Intelligence, Workshop on Configuration*, pages 59–62, Berlin, Allemagne, 2000.

W

D. WALTZ. Understanding Line Drawings of Scenes with Shadows. *In* Patrick WINSTON, *coord*, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975. ISBN 0070710481.

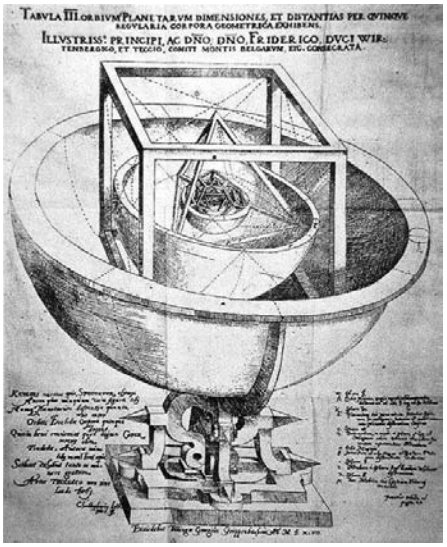
I. WATSON & F. MARIR. Case-Based Reasoning : A Review. *The Knowledge Engineering Review*, 9(4):355–381, 1994.

Z

Edward N. ZALTA, *coord*. *The Stanford Encyclopedia of Philosophy*, 2005. The Metaphysics Research Lab, Center for the Study of Language and Information, Stanford University.

Troisième partie

Annexes



ANNEXE A

Algorithmes

A.1 Algorithmes pour CSP « classiques »

Alg. A.1 BACKTRACKING (V_v, V_n)

```
– ∴ – Retourne une solution (sauf si le CSP est incohérent)
– ∴ – Deux ensembles ordonnés : V_v l'ensemble des couples (variable valuée, valeur choisie)
– ∴ –                          V_n l'ensemble des couples (variable à instancier, domaine d'instanciation)
coherent ← 1
Si (V_n = ∅) Alors
    V_v est une solution
    Retour V_v
Sinon
    CHOISIR & EFFACER x dans V_n
    Sinon
        Pour Chaque i ∈ D_x Faire
            IDENTIFIER C_x ∈ C tel que C_x contraint x
            coherent ← C_x (valeur de vérité de C_x)
            Si (coherent) Alors
                BACKTRACKING (V_v ∪ (x, i), V_n \ x)
            Fin Si
        Fin Pour
    Fin Si
Fin Si
```

Alg. A.2 BACKJUMPING $(x_1, \ldots, x_i, X_i)$

– $\therefore$ – *Retourne le niveau auquel remonter en cas d'un échec dans le parcours de l'arbre*
$niveaupourbackjump \leftarrow 0$
Pour Chaque $x \in D_i$ **Faire**
  Action : TROUVER $X_k, k \in [\![1, \ldots, i-1]\!]$, dont la valeur y est incohérente avec $X_i = x$
  Si $(\exists y \in D_k)$ **Alors**
    $X_i = x$ échoue au niveau $L \leftarrow k$
  Fin Si
  Si $(\nexists y \in D_k)$ **Alors**
    Si ($i = n$ (n nombre de variables)) **Alors**
      $succes \leftarrow$ Vrai
      Action : OUTPUT$(x_1, \ldots, x_j)$
      Retour 0
      Action : SORTIE
    Sinon
      Action : BACKJUMPING$(x_1, \ldots, x_i, X_i)$
      Si (BackJumping = 0) **Alors**
        $succes \leftarrow$ Vrai
        Action : SORTIE
      Sinon
        Si $(L < i)$ **Alors**
          Retour L
          Action : SORTIE
        Fin Si
      Fin Si
    Fin Si
  Fin Si
  $niveaupourbackjump \leftarrow$ MAX$(niveaupourbackjump, L)$
Fin Pour
Retour *niveaupourbackjump*

Alg. A.3 REVISE$(X_i \rightarrow X_j)$

– $\therefore$ – *reduction indique si le domaine de X_i a été réduit.*
$reduction \leftarrow$ Faux
Pour Chaque $x \in D_i$ **Faire**
  Si $(\nexists y \in D_j$ / $(X_i = x, X_j = y)$ cohérent) **Alors**
    $D_i \leftarrow D_i \setminus \{x\}$
    $reduction \leftarrow$ Vrai
  Fin Si
Fin Pour
Retour *reduction*

Alg. A.4 AC-1

– ∴ – *appelle* REVISE *pour chacune des contraintes, ceci jusqu'à ce qu'il n'y ait plus de réductions.*
$Q \leftarrow \{(X_i \rightarrow X_j) \in C, i \neq j\}$
Répéter
  changement ← Faux
  Pour Chaque $(X_i \rightarrow X_j) \in Q$ **Faire**
    changement ← REVISE$(X_i \rightarrow X_j) \vee$ *changement*
  Fin Pour
Jusqu'à (¬*changement*)

Alg. A.5 AC-3

– ∴ – *Même fontionnalité que l'algorithme A.4, en évitant des vérifications inutiles*
$Q \leftarrow C$
Tant Que $(Q \neq \emptyset)$ **Faire**
  $(X_k \rightarrow X_m) \leftarrow$ un élément pris dans Q
  $Q \leftarrow Q \setminus \{(X_k \rightarrow X_m)\}$
  Si (REVISE$(X_k \rightarrow X_m)$) **Alors**
    – ∴ – *Ajout à Q des contraintes liées à la variable dont le domaine a été réduit*
    $Q \leftarrow Q \cup \{(X_i \rightarrow X_k) \in C\}$
  Fin Si
Fin Tant Que

Alg. A.6 INITIALISATION AC-6

– ∴ – *Initialise les variables (et ensembles) pour l'AC-6*
Pour Chaque $(X_i, x) \in D$ **Faire**
  $S_{X_i x} \leftarrow \emptyset$
  $M(X_i, x) \leftarrow$ Vrai
Fin Pour
Pour $(X_i \rightarrow X_j) \in C$ **Faire**
  Pour $x \in D_i$ **Faire**
    Si $(D_j = \emptyset)$ **Alors**
      $supportvide \leftarrow$ Vrai
    Sinon
      $y \leftarrow$ PREMIER(D_j)
      Action : SUPPORTSUIVANT$(X_i, X_j, x, y, supportvide)$
    Fin Si
    Si $(supportvide)$ **Alors**
      $D_i \leftarrow D_i \setminus \{x\}$
      $M(X_i, x) \leftarrow$ Faux
      $liste \leftarrow liste \cup (X_i, x)$
    Sinon
      $S_{X_j y} \leftarrow S_{X_j y} \cup (X_i, x)$
    Fin Si
  Fin Pour
Fin Pour

Alg. A.7 SUPPORTSUIVANT($X_i, X_j, x, y, supportvide$) AC-6

– ∴ – *Trouve le support après y de la valeur x de X_i dans D_j*
Tant Que ($\neg M(X_j, y) \wedge b <$ DERNIER(D_j)) **Faire**
    $supportvide \leftarrow \neg M(X_j, y)$
Fin Tant Que
Tant Que ($\neg(X_i(x) \rightarrow X_j(y)) \wedge \neg supportvide$) **Faire**
    Si ($y <$ DERNIER(D_j)) **Alors**
        $y \leftarrow$ SUIVANT(y, D_j)
    Sinon
        $supportvide \leftarrow$ Vrai
    Fin Si
Fin Tant Que

Alg. A.8 PROPAGATION AC-6

– ∴ – *Équivaut à l'algorithme AC-3, avec une optimisation sur la complexité temporelle*
Tant Que ($liste \neq \emptyset$) **Faire**
    Action : SELECT & DELETE($(X_j, y) \in liste$)
    Pour $(X_i, x) \in S_{X_j y}$ **Faire**
        Action : DELETE($(X_i, x) \in S_{X_j y}$)
        Si ($M(X_i, x)$) **Alors**
            $z \leftarrow y$
            SUPPORTSUIVANT($X_i, X_j, x, y, supportvide$)
            Si ($supportvide$) **Alors**
                $D_i \leftarrow D_i \setminus \{x\}$
                $M(X_i, x) \leftarrow$ Faux
                $liste \leftarrow liste \cup (X_i, x)$
            Sinon
                $S_{X_j,z} \leftarrow S_{X_j,z} \cup (X_i, x)$
            Fin Si
        Fin Si
    Fin Pour
Fin Tant Que

Alg. A.9 AC-3 – FORWARD-CHECKING(cv)

– ∴ – *Établit l'arc-cohérence sur les domaines des futures variables instanciables*
$Q \leftarrow \{(X_i \rightarrow X_{cv}) \in C, i > cv\}$
$coherent \leftarrow$ Vrai
Tant Que ($Q \neq \emptyset \wedge coherent$) **Faire**
    Action : CHOISIT & EFFACE($(X_k \rightarrow X_m) \in Q$)
    Si (REVISE($X_k \rightarrow X_m$)) **Alors**
        $coherent \leftarrow \neg(D_k = \emptyset)$
    Fin Si
Fin Tant Que
Retour *coherent*

Alg. A.10 AC-3 – LOOK-AHEAD(cv)

– $\therefore$ – *Établit l'arc-cohérence sur les domaines de toutes les variables futures*
$Q \leftarrow \{(X_i \rightarrow X_{cv}) \in C, i > cv\}$
coherent $\leftarrow$ Vrai
Tant Que ($Q \neq \emptyset \wedge$ *coherent*) **Faire**
  CHOISIT & EFFACE($(X_k \rightarrow X_m) \in Q$)
  Si (REVISE($X_k \rightarrow X_m$)) **Alors**
    $Q \leftarrow Q \cup \{(X_i \rightarrow X_k) \,/\, (X_i \rightarrow X_k)C, i \neq k, i \neq m, i > cv\}$
    coherent $\leftarrow \neg D_k = \emptyset$
  Fin Si
  Retour coherent
Fin Tant Que

A.2 Algorithme DCSP

Alg. A.11 DCSP (V_I)

– ∴ – *Propage les contraintes d'activité avant de vérifier les contraintes de compatibilité.*
$V \leftarrow V_I$
$solution \leftarrow \emptyset$
Action : VERIFIER toutes les contraintes actives
Si (CONTRADICTION($V, solution$)) **Alors**
  Retour *echec* – ∴ – *Le problème initial est incohérent*
Fin Si
retour? ← Faux
Tant Que ($V \neq \emptyset$) **Faire**
  Si (*retour*? ∨ CONTRADICTION($V, solution$)) **Alors**
    Action : BACKTRACK(V_{cv}) ∧ CHANGE(V ET $solution$)
    Si (BackTrack(V_{cv}) = Faux) **Alors**
      Retour *echec*
    Fin Si
    retour? ← Faux
  Sinon Si ($\exists C_A$(ARV) active) **Alors**
    Action : EXECUTE(C_A(ARV))
    $V \leftarrow V \cup$ (nouvelles variables actives)
  Sinon Si ($\exists C_A$(ARN) active) **Alors**
    Action : EXECUTE(C_A(ARN))
    $V \leftarrow V \setminus$ (variables désactivées)
  Sinon Si ($\exists C_A$(RV) active) **Alors**
    Action : EXECUTE(C_A(RV))
    $V \leftarrow V \cup$ (nouvelles variables actives)
  Sinon Si ($\exists C_A$(RN) active) **Alors**
    Action : EXECUTE(C_A(RN))
    $V \leftarrow V \setminus$ (variables désactivées)
  Sinon Si ($\exists C_C$ active) **Alors**
    Action : EXECUTE(C_C)
  Sinon
    Action : CHOISIT & EFFACE($V_i \in V$)
    $V_{cv} \leftarrow x, x \in D_i$ / x non encore testée
    Si (V_{cv} = Null) **Alors**
      retour? ← Vrai
    Sinon
      $solution \leftarrow solution + V_{cv}$
    Fin Si
  Fin Si
Fin Tant Que

A.3 Algorithmes RCSP — filtrage des nœuds de composition logique

Chacun de ces algorithmes prend en entrée les espaces possiblement vrais et faux de ses parents, ainsi qu'un booléen ($modif_D$ marquant la modification de domaines de variables ; ce booléen est aussi la variable de retour.

Alg. A.12 CALC_SPACE_NOT ($modif_D$)

– ∴ – ***T**his est le nœud considéré, f son fils*
Si (Élagage) **Alors**
  Si (f est toujours faux) **Alors**
    – ∴ – *le not est toujours vrai*
    ***T**his* ← ***V**rai*
    Retour 1
  Fin Si
  Si (f est toujours vrai) **Alors**
    – ∴ – *le not est toujours faux*
    ***T**his* ← ***F**aux*
    Retour 1
  Fin Si
Fin Si
◇V (***T**his*) ← f(◇F)
◇F (***T**his*) ← f(◇V)
Retour $modif_D$

Alg. A.13 CALC_SPACE_AND ($modif_D$)

```
– ∴ – This est le nœud considéré, F l'ensemble de ses fils
Si (Élagage) Alors
    Pour Chaque f ∈ F Faire
        Si (f = Faux) Alors
            – ∴ – le and est toujours faux
            This ← Faux
            Retour 1
        Fin Si
        Si (f = Vrai) Alors
            – ∴ – le and perd un fils
            F ← F \ f
        Fin Si
    Fin Pour
    Si (F = ∅) Alors
        – ∴ – plus de fils : le and est toujours vrai
        This ← Vrai
        Retour 1
    Sinon Si (card(F) = 1) Alors
        – ∴ – il ne reste qu'un fils (f, qui remplace le and)
        This ← f
        Retour 1
Fin Si
◇V (This) ← ⋒ (f_i(◇V) / f_i ∈ F)
◇F (This) ← ⋓ (f_i(◇F) / f_i ∈ F)
Retour modif_D
```

Alg. A.14 CALC_SPACE_OR ($modif_D$)

```
– ∴ – This est le nœud considéré, F l'ensemble de ses fils
Si (Élagage) Alors
    Pour Chaque f ∈ F Faire
        Si (f = Vrai) Alors
            – ∴ – le or est toujours vrai
            This ← Vrai
            Retour 1
        Fin Si
        Si (f = Faux) Alors
            – ∴ – le or perd un fils
            F ← F \ f
        Fin Si
    Fin Pour
    Si (F = ∅) Alors
        – ∴ – plus de fils : le or est toujours faux
        This ← Faux
        Retour 1
    Sinon Si (card(F) = 1) Alors
        – ∴ – il ne reste qu'un fils (f, qui remplace le or)
        This ← f
        Retour 1
    Fin Si
Fin Si
◇V (This) ← ⋓ (f_i(◇V) / f_i ∈ F)
◇F (This) ← ⋒ (f_i(◇F) / f_i ∈ F)
Retour modif_D
```

Alg. A.15 CALC_SPACE_EQUIVALENCE ($modif_D$)

– $\therefore$ – ***This*** *est le nœud considéré,* f_1 *et* f_2 *ses fils*
Si (Élagage) **Alors**
  Si $((f_1 = \textbf{\textit{Vrai}} \wedge f_2 = \textbf{\textit{Vrai}}) \vee (f_1 = \textbf{\textit{Faux}} \wedge f_2 = \textbf{\textit{Faux}}))$ **Alors**
    – $\therefore$ – *les deux opérandes sont équivalents : le* <=> *est toujours vrai*
    This $\leftarrow$ ***Vrai***
    Retour 1
  Fin Si
  Si $((f_1 = \textbf{\textit{Vrai}} \wedge f_2 = \textbf{\textit{Faux}}) \vee (f_1 = \textbf{\textit{Faux}} \wedge f_2 = \textbf{\textit{Vrai}}))$ **Alors**
    – $\therefore$ – *les deux opérandes sont opposés : le* <=> *est toujours faux*
    This $\leftarrow$ ***Faux***
    Retour 1
  Fin Si
  Si $(f_1 = \textbf{\textit{Vrai}})$ **Alors**
    – $\therefore$ – *le* <=> *devient son autre opérande*
    This $\leftarrow f_2$
    Retour 1
  Fin Si
  Si $(f_2 = \textbf{\textit{Vrai}})$ **Alors**
    – $\therefore$ – *le* <=> *devient son autre opérande*
    This $\leftarrow f_1$
    Retour 1
  Fin Si
  Si $(f_1 = \textbf{\textit{Faux}})$ **Alors**
    – $\therefore$ – *le* <=> *devient le* not *de son autre opérande*
    This $\leftarrow (\neg f_2)$
    Retour CALC_SPACE_NOT (***This***)
  Fin Si
  Si $(f_2 = \textbf{\textit{Faux}})$ **Alors**
    – $\therefore$ – *le* <=> *devient le* not *de son autre opérande*
    This $\leftarrow (\neg f_1)$
    Retour CALC_SPACE_NOT (***This***)
  Fin Si
Fin Si
$\Diamond$V (***This***) $\leftarrow (\Diamond \mathrm{V}(f_1) \Cap \Diamond \mathrm{V}(f_2)) \Cup (\Diamond \mathrm{F}(f_1) \Cap \Diamond \mathrm{F}(f_2))$
$\Diamond$F (***This***) $\leftarrow (\Diamond \mathrm{V}(f_1) \Cup \Diamond \mathrm{V}(f_2)) \Cap (\Diamond \mathrm{F}(f_1) \Cup \Diamond \mathrm{F}(f_2))$
Retour $modif_D$

Alg. A.16 CALC_SPACE_IMPLICATION ($modif_D$)

*– ∴ – **T**his est le nœud considéré, f_1 et f_2 ses fils ($f_1 \Rightarrow f_2$)*
Si (Élagage) **Alors**
  Si $((f_1 = \textbf{\textit{F}}aux) \vee (f_2 = Vrai))$ **Alors**
    – ∴ – le => est toujours vrai
    ***T**his* ← ***V**rai*
    Retour 1
  Fin Si
  Si $((f_1 = \textbf{\textit{V}}rai) \wedge (f_2 = \textbf{\textit{F}}aux))$ **Alors**
    – ∴ – le => est toujours faux
    ***T**his* ← ***F**aux*
    Retour 1
  Fin Si
  Si $(f_2 = \textbf{\textit{F}}aux)$ **Alors**
    – ∴ – le => devient le not de f_1
    ***T**his* ← $(\neg f_1)$
    Retour CALC_SPACE_NOT (***T**his*)
  Fin Si
  Si $(f_1 = \textbf{\textit{V}}rai)$ **Alors**
    – ∴ – le => devient son fils f_2
    ***T**his* ← f_2
    Retour 1
  Fin Si
Fin Si
$\Diamond$V (***T**his*) ← $\Diamond \mathrm{F}(f_1) \Cup \Diamond \mathrm{V}(f_2)$
$\Diamond$F (***T**his*) ← $\Diamond \mathrm{V}(f_1) \Cap \Diamond \mathrm{F}(f_2)$
Retour $modif_D$

Annexe B

Autres exemples

B.1 La voiture de Mittal et Falkenhainer

Une voiture est constituée de 8 composants :

- la forme (*break*, « Sedan » ou décapotable) ;
- un *package* (Lux, Delux ou Standard) ;
- un moteur (petit, moyen ou gros) ;
- une batterie (petite, moyenne ou grosse) ;
- l'air conditionné (Ac1 ou Ac2) ;
- un toit ouvrant (Sr1 ou Sr2) pourvu d'un
- système d'ouverture (automatique ou manuel) ;
- des vitres (teintées ou non).

Parmi ces composants, seuls 3 ne sont pas facultatifs : la forme, le *package* et le moteur. L'équation B.1 page suivante montre les contraintes de cet exemple en utilisant la valeur ★ pour symboliser la non-pertinence des variables, dont les variables et leurs domaines sont définies ci-dessous :

Package : $V_1 \in \{$ Delux, Luxury, Standard $\}$

Frame : $V_2 \in \{$ Hatchback, Sedan, Convertible $\}$

Engine : $V_3 \in \{$ Small, Medium, Large $\}$

Battery : $V_4 \in \{$ Small, Medium, Large, ★ $\}$

Opener : $V_5 \in \{$ Auto, Manual, ★ $\}$

AirCond : $V_6 \in \{$ Ac1, Ac2, ★ $\}$

Glass : $V_7 \in \{$ Tinted, Not-tinted, ★ $\}$

Sunroof : $V_8 \in \{$ Sr1, Sr2, ★ $\}$

$$
\begin{aligned}
C_1 &: \ \forall i \in \{1,2,3\}, V_i \neq \bigstar \\
C_2 &: \ V_3 \neq \bigstar \Rightarrow V_4 \neq \bigstar \\
C_3 &: \ (V_3 = \text{Small} \wedge V_4 = \text{Small}) \Rightarrow V_6 = \bigstar \\
C_4 &: \ V_2 = \text{Convertible} \Leftrightarrow V_1 \neq \text{Standard} \\
C_5 &: \ V_2 = \text{Convertible} \Rightarrow V_8 = \bigstar \\
C_6 &: \ V_1 = \text{Standard} \Leftrightarrow V_6 \neq \text{Ac2} \\
C_7 &: \ V_1 = \text{Deluxe} \Rightarrow V_8 \neq \bigstar \\
C_8 &: \ V_1 = \text{Luxury} \Leftrightarrow V_6 \neq \text{Ac1} \\
C_9 &: \ V_1 = \text{Luxury} \Rightarrow (V_6 \neq \bigstar \wedge V_8 \neq \bigstar) \\
C_{10} &: \ (V_5 = \text{Auto} \wedge V_6 = \text{Ac2}) \Rightarrow V_4 = \text{Large} \\
C_{11} &: \ (V_5 = \text{Auto} \wedge V_6 = \text{Ac1}) \Rightarrow V_4 = \text{Medium} \\
C_{12} &: \ V_5 \neq \bigstar \Rightarrow V_8 \neq \bigstar \\
C_{13} &: \ (V_6 = \text{Ac2} \wedge V_8 = \text{Sr1}) \Rightarrow V_7 \neq \text{Tinted} \\
C_{14} &: \ V_7 \neq \bigstar \Leftrightarrow V_8 \neq \bigstar \\
C_{15} &: \ V_8 = \text{Sr1} \Rightarrow V_5 = \bigstar \\
C_{16} &: \ V_8 = \text{Sr2} \Rightarrow V_5 \neq \bigstar \\
C_{17} &: \ V_8 = \text{Sr1} \Rightarrow V_6 \neq \bigstar
\end{aligned}
\tag{B.1}
$$

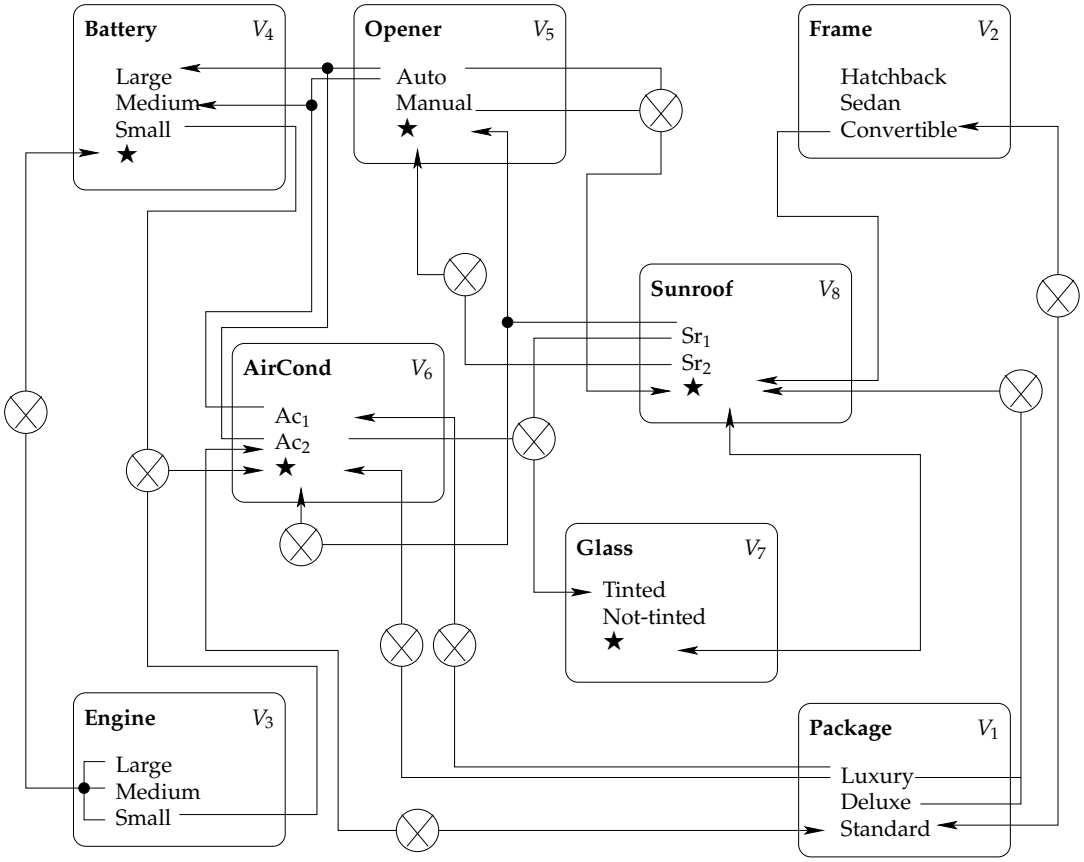


FIG. B.1 — Schéma de la voiture de MITTAL & FALKENHAINER (1990), en utilisant la valeur ★ pour la non-pertinence

Colophon

Ce mémoire a été rédigé en utilisant les logiciels suivants : *XEmacs* (http://www.xemacs.org) pour l'édition du source, *Xfig* (http://www.xfig.org) pour l'édition des schémas et LaTeX 2ε (cf. http://www.ctan.org) puis DVIPS pour le formatage du document.

Les images en tête de partie sont issues de l'encyclopédie libre Wikipedia (http://www.wikipedia.org).

— Partie I : « L'Homme de Vitruve », issu de l'« Étude de proportions du corps humain selon Vitruve » réalisée par Léonard de Vinci (artiste, scientifique et philosophe italien, 1452 – 1519) ;

— Partie II : un mécanisme d'irrigation, dessiné par Ibn Ismail Ibn al-Razzaz Al-Jazari, dit Al-Jazari, ingénieur arabe du XIII^{e} siècle ;

— Partie III : le modèle d'univers de Johannes Kepler (astronome allemand, 1571 – 1630), fondé sur les cinq polyèdres réguliers.

Contribution à l'élaboration d'un formalisme gérant la pertinence pour les problèmes d'aide à la conception à base de contraintes

Résumé :

Les travaux présentés dans cette thèse portent sur l'aide à la conception et à la configuration. Une intégration de différents concepts existant dans les domaines de la programmation par contraintes a été réalisée. Cette intégration a pu être testée sur une implémentation basée sur des arbres syntaxiques représentant un CSP (problème de satisfaction de contraintes) modélisant un problème de conception ou configuration.

La première partie de la thèse présente les domaines de la conception et de la configuration, et en fait ressortir les besoins pour l'aide à la décision : paramètres discrets et continus, organisation hiérarchique et éléments optionnels. Différentes approches à base de contraintes permettant de répondre à ces besoins sont ensuite détaillées.

La seconde partie présente les RCSP (CSP gérant la pertinence), qui intègrent les différents mécanismes vus dans la première partie. Des préconisations de modélisation pour les problèmes de conception et de configuration sont établies. L'outil réalisé est ensuite présenté, dans un premier temps pour le traitement de problèmes CSP et dans un deuxième temps pour le traitement de RCSP.

Mots-clés : *conception, configuration, programmation par contraintes, filtrage, résolution, arbres syntaxiques*

Contribution to the development of a constraint-based formalism managing the relevance to assist design tasks

Abstract:

The research work presented in this thesis deals with assistance to design and configuration tasks. An integration of different existing concepts of constraint programming has been achieved. This integration has been tested on an implementation based upon syntaxic trees. The syntaxic trees allow to express different kinds of CSP (Constraint Satisfaction Problem) which model design or configuration problems.

The first part presents the fields of design and configuration, and aims at identifying the needs for decision aid: different kinds of parameters (discrete and continuous), hierarchical organisation and optionnal elements. Different constraint-based approaches which may fulfill any need are then detailed.

The second part presents the RCSP (Relevancy CSP), which are an integration of different CSP from the literature seen in the first part. Some recommendations for modeling design or configuration problems are set up. The implementation is then presented, on the one hand for CSP processing and on the other hand for RCSP processing.

Keywords: *design, configuration, constraint programming, filtering, resolution, syntaxic trees*

Centre Génie Industriel, École des Mines d'Albi-Carmaux,
Campus Jarlard, 81 013 ALBI CT Cedex 09 – FRANCE.
Tél. : + 33 (0)5 63 49 31 56 — Fax : + 33 (0)5 63 49 31 83
http://www.enstimac.fr