ÉCOLE DOCTORALE SYSTÈMES

Configuration, CSP et base de connaissances

Mémoire de DEA – Systèmes Industriels

Tuteur : Paul Gaborit

Thomas van Oudenhove

— Année universitaire : 2002-2003 —

Avant-propos

Conventions de notation

Les références à la bibliographie sont faites entre crochets, généralement par les initiales des auteurs suivies de l'année de publication (ou les trois premières lettres du nom, en cas d'auteur unique) – exemple : [Aldanondo *et al.* 01].

Les mots mis en valeur *(italique)* sont généralement des mots anglais.

Les mots en petites capitales (par exemple, CSP) font généralement référence à des acronymes et sont explicités dans le glossaire, page 27. En particulier, l'acronyme CSP est utilisé même au pluriel.

À propos de la cohérence d'un CSP, l'adjectif « consistant » sera aussi employé (issu de l'anglais *consistent*).

Remerciements

Je souhaite remercier toue l'équipe du centre *Génie Industriel* de l'École des Mines d'Albi, et plus particulièrement :

- Paul GABORIT, mon tuteur de stage, pour m'avoir aidé sur la bibliographie de ce mémoire, et pour m'avoir proposé un sujet de thèse ;
- Michel ALDANONDO pour ses orientations générales ;
- Élise VAREILLES pour les publications plus récentes qu'elle m'a fournies ;
- enfin, Hervé PINGAUD et Lionel DUPONT pour s'être penchés sur l'épineux problème de mon sujet de mémoire ...

Résumé

Résumé

La configuration est une des interfaces du processus de vente se situant entre la vente et la gestion de production. Son but est de fournir au client le produit configurable le plus adapté à ses besoins. Dans ce cadre, les CSP sont un formalisme adapté pour représenter des problèmes de configuration, car le système des contraintes permet à l'utilisateur d'exprimer ses choix. Cependant, les CSP utilisés en configuration ne sont pas standards, et doivent donc faire appel à de nouvelles méthodes de résolution (directes ou par transformation en jeu de CSP standards). Un autre enjeu de la configuration est de pouvoir générer un CSP à partir de modèles de produits (représentation UML, ...) existants et utilisés en entreprise. Le *(J)Configurator* d'ILOG essaie d'exploiter toutes ces techniques. En conclusion, les étapes amont et aval d'un problème de configuration global sont bien maîtrisées (modélisation du produit et résolution de CSP standards) ; mais la transition entre ces deux phases fait actuellement l'objet de recherches.

Abstract

The configuration aims at a better communication between sales and production. Its goal is to supply the customer with a product which fits to his needs. The CSP paradigm allows to solve configuration problems, thanks to constraints, which allow the user to specify his needs. However, standard CSP solving methods rarelly fit real configuration problems. Indeed, the generated CSP may contain conditional or mixed constraints and need new solving methods. Another problem is the generation of these CSP ; it can be done from a standard design language as UML, modeling a configurable product. We will see an example of all these emergent techniques with the ILOG (J)Configurator. In conclusion, the steps of modeling a product and solving a standard CSP are mastered, but researchs are done on the transition from the model to a set of standard CSP.

Table des matières

Chapitre 1

Introduction

Depuis les débuts de la gestion industrielle, les entreprises ont essayé d'améliorer les phases de configuration de leurs produits pour mieux satisfaire le client, tout essayant de réduire leurs coûts de production. En effet, la configuration vise à fournir au client le produit le plus adapté à ses besoins, à ses envies. Par exemple, dans le domaine automobile, nous sommes passé du modèle unique de la FORD T au début du XX$^{\text{ième}}$ siècle à des catalogues contenant différents modèles en quelques décénnies, et diverses options permettant au client de façonner *sa* voiture.

Ainsi, nous allons voir dans un premier temps quelles sont les nouvelles problématiques de la configuration et quelle est sa situation dans l'entreprise, entre la vente et la gestion de production. Nous allons ensuite présenter quelques techniques de résolution utilisées pour des problèmes de configuration réels (CSP complexes).

Dans une troisième partie, nous allons voir une méthode permettant de générer un problème de type CSP (contraintes) à partir d'un modèle objet du produit configurable. Ces techniques ont été mises en application dans le *(J)Configurator* d'ILOG, dont nous verrons le fonctionnement en dernière partie.

Chapitre 2

Problématique

2.1 Configuration

La configuration doit permettre à un utilisateur ou client de disposer du produit le plus adapté à ses besoins fonctionnels et/ou esthétiques. Ainsi, les entreprises essayent depuis quelques années de disposer d'un produit générique qu'elles pourront ensuite différencier, le plus tard possible dans la chaîne de conception/production, en une déclinaison de sous-produits, afin de réduire les délais, les coûts de fabrication et de satisfaire les demandes des client.

Ainsi, lorsqu'un produit est complexe (nombreuses pièces optionnelles, certaines étant incompatibles), le vendeur du produit doit pouvoir disposer d'un outil d'aide à l'expression du besoin du client. Ainsi, les premiers configurateurs sont en fait des catalogues recensant toutes les possibilités (assimilables à une base de connaissances simplifiée). Le but de la configuration est de définir, avec le client, le produit souhaité, tout en étant guidé par un logiciel, le configurateur, qui s'assure de l'adéquation entre le besoin client et l'offre produit fournisseur (par exemple, j'ai besoin d'un micro-ordinateur pour effectuer du traitement d'images).

Dans l'idéal, le configurateur doit renvoyer le produit configuré, ainsi qu'une nomenclature, une gamme de fabrication, et un prix de revient (ou autres : prix conseillé de vente, coûts des matières premières, ...), en respectant les contraintes fonctionnelles exprimées par le client. C'est en effet l'adéquation entre le besoin client et l'offre fournisseur qui permet de limiter les coûts liés au processus de vente du produit fini.

2.2 Définitions

2.2.1 Configuration

Il existe plusieurs définitions de la configuration. La première a été donnée par Mittal et Frayman en 1989 (cf. [Mittal, Frayman 89]) :

> *Étant donné un produit configurable, [configurer, c'est] construire une ou plusieurs configurations satisfaisant toutes les contraintes, où une configuration est un ensemble de composants et la description des connections entre eux, ou bien détecter l'incohérence dans les contraintes.*

Cependant, nous retiendrons la définition donnée par [Veron 01] :

> *Étant donné un modèle repésentant 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 terme d'une nomenclature de fonctions et/ou de produits.*

Cette définition est en effet plus complète, car elle apporte la notion de cohérence, en rapport avec les CSP. De plus, elle introduit le modèle du produit, qui doit être le point de départ de la configuration, et le souhait de l'utilisateur, qui doit en être le point d'arrivée.

2.2.2 Produit configurable

De même que pour la configuration, MITTAL et FRAYMAN (cf. [Mittal, Frayman 89]) ont été les premiers à en proposer une définition :

> *Un produit configurable est constitué :*
> *A1 d'au moins une architecture fonctionnelle, chacune définie de manière abstraite par un ensemble de fonctions rf_i et of_i : $\{rf_1, \ldots, rf_n, of_1, \ldots, of_m\}$. Les fonctions rf_i sont requises et les fonctions of_i optionnelles ;*
> *A2 d'un ensemble fixé et prédéfini de composants, où un composant est décrit par un ensemble de propriétés, de ports pour la connexion aux autres composants, de contraintes décrivant les composants pouvant se lier à ces ports et d'autres contraintes structurelles ;*
> *A3 de méthodes de correspondance entre chaque fonction f_i et un composant c_i, considéré comme un composant essentiel pour remplir la fonction f_i, et la description des autres fonctions nécessaires à c_i pour remplir la fonction f_i.*

Cette définition ne prend pas du tout en compte l'existence de contraintes pouvant limiter les choix possibles et l'éventualité de conditions gérant l'existence de composants ou de fonctions. Pour ces raisons, nous retiendrons la définition plus générale proposée par [Aldanondo *et al.* 01] :

> *Un produit configurable peut être défini par un ensemble de variables. Chaque variable appartient à un domaine de définition (continu ou discret). Il existe des contraintes pouvant conditioner l'existence de variables et/ou restreindre leur domaine de définition. Configurer vise alors à donner une valeur à toutes les variables qui existent dans un produit configurable, en respectant toutes les contraintes.*

2.3 Situation de la configuration

Avec l'apparition de produits génériques pour la production de séries, le problème de la configuration s'est posé. Ainsi, la plupart des entreprises disposent d'un produit générique à décliner en plusieurs produits spécifiques.

Le processus de configuration est utilisé pour répondre au mieux aux besoins du client. Ainsi, plusieurs allers-retour sont effectués entre le client la gestion de production, la vente et la configuration assurant l'interface (cf. schéma 2.1, page 4).

Le client exprime ses besoins à la vente, qui affecte des valeurs aux variables de configuration. Le configurateur renvoie alors des solutions réalistes à la vente, qui fait ses propositions au client. Une fois que le client a arrêté son choix, une commande est envoyée à la production (avec éventuellement une nomenclature technique et des procédés de fabrication, selon le type de configuration).

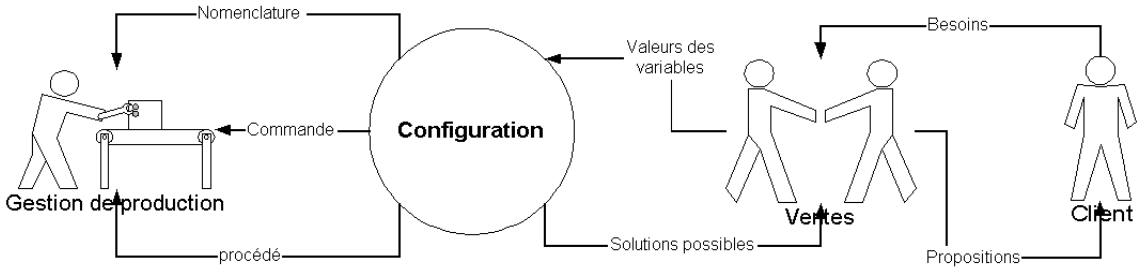

FIG. 2.1 – Du client à la gestion de production en passant par la configuration

La fonction configuration sert donc à faire une interface entre le client et le produit désiré final. Il existe plusieurs types de configuration, cités dans [Aldanondo *et al.* 01] :

***Pick to order* (PTO) :** configuration pour la vente, qui se déroule pendant le dialogue client/fournisseur. Cette approche est adaptée à des grandes séries, pour des produits simples ou techniquement parfaitement maîtrisés (voitures, ...).

***Assemble to order* (ATO) :** configuration adaptée aux moyennes séries, pour des produits un peu plus complexes que le PTO, qui peuvent nécessiter quelques calculs.

***Engineer to order* (ETO) :** pour des petites séries. Ce type de configuration recquiert des variables et des calculs pour définir une nomenclature technique détaillée et éventuellement un procédé de fabrication.

***Design to order* (DTO) :** cette démarche est adaptée à des fabrications unitaires, en interaction avec des outils de CAO, et peut répondre à des problèmes complexes.

2.4 Outils et problématique

Il existe plusieurs outils permettant de configurer un produit. Le premier à apparaître a été le catalogue papier. Puis, nous le verrons dans la section 3.1, page 6, les CSP (CSP : *Constraint Satisfaction Problem*) sont apparus comme un bon moyen de résoudre certains problèmes, exprimés à l'aide de contraintes. À partir de là, un nouveau problème se pose : comment arriver à exprimer un besoin industriel de configuration en termes de CSP ? Ainsi, [Felfernig *et al.* 01] proposent d'utiliser la méthode de modélisation UML, et en particulier OCL (partie d'UML) pour aider à exprimer le problème sous forme de CSP dynamiques, qui seront ensuite « transformés » en CSP, de manière à être résolus. Nous pouvons voir ce cheminement, du besoin industriel jusqu'à l'obtention d'un problème potentiellement solvable, sur la figure 2.2, page 5.

Partant d'un produit configurable, modélisé selon une méthode objet, nous cherchons à avoir un problème CSP solvable [**1**] (CSP standard). Pour générer ce problème, il faut passer par une étape intermédiaire : les CSP complexes [**2**] qui peuvent ensuite être résolus directement ou transformés en CSP standards.

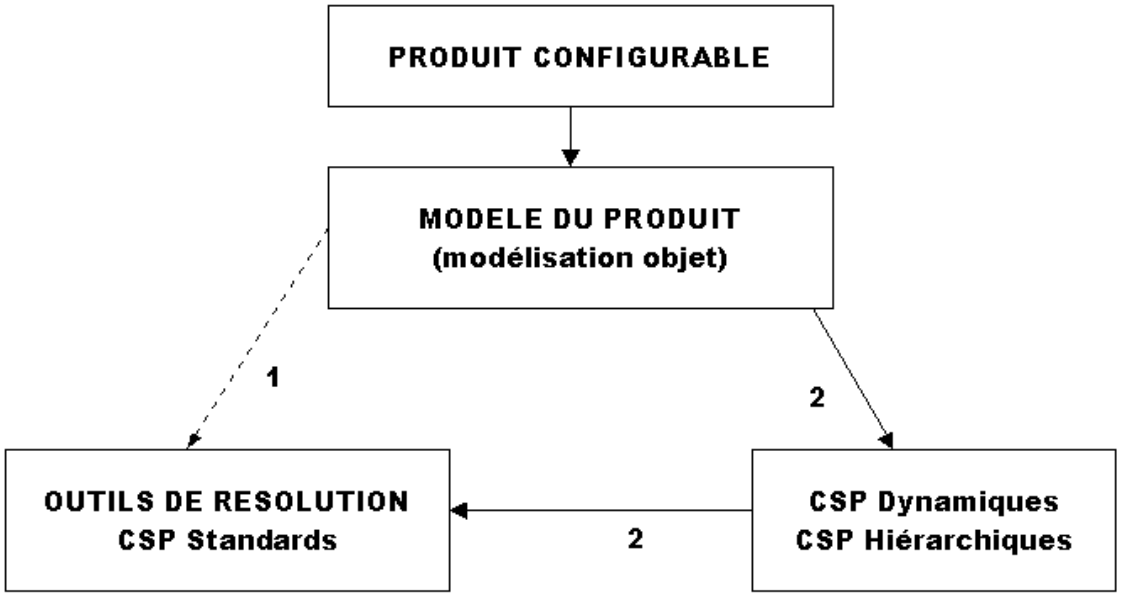

FIG. 2.2 – D'un modèle du produit à la configuration

Chapitre 3

Application des techniques de CSP à la configuration

3.1 Présentation des CSP

Le formalisme des problèmes à satisfaction de contraintes (CSP : *Constraint Satisfaction Problem*) a été introduit par [Montanari 74]. À la base utilisés pour des techniques relevant de l'intelligence artificielle (au départ, traitement d'images en robotique), les CSP sont maintenant beaucoup utilisés dans des domaines relevant de la recherche opérationnelle et de l'optimisation : logisitique, ordonnancement, ...

Les bases fondamentales de ce formalisme sont les contraintes et les variables. De la manière dont nous l'avons défini (cf. définition page 3), un produit configurable est exprimé par un ensemble de variables, sur lesquelles s'appliquent des contraintes. Nous pouvons donc imaginer traiter des problèmes de configuration en utilisant le formalisme CSP. En effet, l'utilisation de variables définies sur un domaine et de contraintes dans les deux concepts justifie cette application des CSP aux problèmes de configuration.

La suite de cette section présente brièvement les CSP.

3.1.1 Définition

Un CSP est défini par un triplet (X, D, C) où :
- X est un ensemble de variables ;
- D l'ensemble des domaines de définition des variables ;
- C un ensemble de contraintes.

Les contraintes définissent les relations entre les variables en limitant les combinaisons de leurs valeurs. Nous distinguons deux natures de contraintes :

les contraintes de compatibilité : elles expriment les compatibilités de valeurs entre plusieurs variables. Elles restreignent les domaines de définitions des variables en autorisant certaines combisaisons de valeurs.

les contraintes d'activation : elles déterminent l'existence ou non d'une ou plusieurs variables, c'est à dire leur appartenance ou non à la solution courante (cf. [Mittal, Falkeinhainer 90]).

Les contraintes peuvent être exprimées sous différentes formes : de table de valeurs compatibles, formules mathématiques, etc. Elles s'appliquent aussi bien aux variables discrètes (domaine de valeurs symboliques ou numériques dénombrables) qu'aux variables continues (domaine de définition numérique indénombrable).

3.1.2 Résolution

La résolution d'un CSP consiste à trouver au moins une valeur pour chaque variable en respectant toutes les contraintes. Un CSP est dit cohérent si chaque variable peut prendre une valeur dans son domaine de définition, sans violer aucune contrainte. Dans le cas contraire, le CSP est dit incohérent.

Une des solutions pour résoudre ce type de problème est celle de la « force brute ». Cette méthode consiste à parcourir l'arbre de résolution[1] branche par branche. Si une branche devient incohérente, elle est abandonnée, et l'exploration d'une autre branche commence.

Cependant, cette méthode peut s'avérer très coûteuse en temps. En nommant n le nombre de variables et d la taille de leur domaine de définition, l'arbre de résolution compte $d^{\frac{n(n+1)}{2}}$ nœuds. Le nombre de nœuds croît donc exponentiellement et peut rapidement atteindre une taille (d^n) « insurmontable ». De plus, cette méthode est seulement applicable aux cas où les variables X sont discrètes et ne peut pas traiter les variables continues (variables géométriques par exemple).

Pour tenter de pallier aux inconvénients des méthodes complètes[2], des méthodes locales ont été introduites à partir de la fin des années 1970, en particulier :

- la cohérence d'arc introduite par [Mackworth 77]. Le principe est de vérifier par couple de variables si tous les produits cartésiens des valeurs de leur domaine sont corrects. Donc, pour chaque couple de variables liées (X, Y), valuer X successivement à toutes les valeurs de son domaine et vérifier si une valeur du domaine de Y satisfait les contraintes qui les lient, et inversement ;
- la k-cohérence ([Cooper 89]). Le principe est le même, sauf que ce sont des n-uplets de $(k-1)$ variables valuées qui sont testés. La possibilité de valuation pour la $k^{\text{ième}}$ variable est alors étudiée (si une valeur maintient la cohérence). Ensuite, l'ordre d'instanciation des $(k-1)$ premières variables est changé, et ainsi de suite ...

Ce type de méthode explore l'espace des solutions en se basant sur une heuristique. Ces méthodes ont l'avantage d'être très rapides ; en revanche, elles ne peuvent garantir ni la complétude, ni l'optimalité des résultats.

3.1.3 Propagation de contraintes

Ce mécanisme permet de réduire la taille des branches de l'arbre des solutions. Le principe est d'éliminer des branches non valides lors de l'affectation d'une valeur à une variable. En faisant faire un premier choix au client, des valeurs des domaines de variables sont éliminées par les contraintes de compatibilité. Les contraintes de compatibilité permettent d'éliminer encore d'autres valeurs possibles pour les variables restantes. Selon les problèmes, la taille de l'arbre de résolution peut être significativement réduite. De plus, ce type de mécanisme est itératif, ce qui permet une prise en charge relativement facile par une machine.

Il existe plusieurs méthodes de propagation de contraintes, en particulier :

- *Backtracking* (BT) : exploration de l'arbre de recherche et retour en arrière s'il y a incohérence ;
- *Forward-Checking* (FC) : vérification de contraintes impliquant la variable courante et les variables non encore instanciées ;

[1] Affectation d'une valeur à une variable, puis à une deuxième, et ainsi de suite

[2] Comme par exemple celle de la « force brute ».

- *Maintenance de cohérence d'arc* (MAC) : méthode aussi appelée *look-ahead*, apparentée au *forward-checking*, mais vérifiant la cohérence globale des arcs.

Pour plus de détails sur ces méthodes et les algorithmes, cf. [Barták].

3.2 Configurateurs

Il s'agit, lors de la création d'un configurateur, de modéliser le produit générique et toutes ses variantes de manière déclarative, ainsi que les différentes propriétés des variantes possibles.

3.2.1 Modélisation

À chaque variable est associée un état (variable booléenne : vrai ou faux) permettant de savoir si cette variable participe au problème courant. Ainsi, la variable en question peut être prise en compte dans le mécanisme de résolution (ou non, si elle n'est pas « active »).

Les contraintes de compatibilité découlent naturellement des contraintes de faisabilité du produit, qu'elles soient discrètes ou continues, ou mixtes. Les contraintes sur des variables discrètes sont généralement exprimées en termes de compatibilité ou d'incompatibilité, alors que les contraintes numériques le sont grâce à des formules algébriques lorsqu'elles sont continues, mais peuvent aussi l'être sous forme de tables.

Les contraintes d'activation représentent plus facilement les contraintes structurelles du produit configurable : c'est à dire qu'elles jouent sur le nombre de composants qui vont entrer dans la composition finale du produit. Elles permettent en effet d'ajouter ou de retirer une ou plusieurs variables de la solution courante et donc de modifier la structure interne et/ou externe du produit fini. Ainsi, un certain composant peut être requis si un autre composant est présent et si un troisième composant a une de ses propriétés fixée au-dessus d'une valeur seuil.

Prenons l'exemple d'une voiture. Il existe des contraintes de compatibilité telles que : le volume du moteur ne doit pas dépasser 400 litres. L'utilisateur ajoute alors des contraintes qui réduisent les valeurs possibles dans le somaine de définition, par exemple : la consommation ne doit pas dépasser 7 litres pour 100 km. Enfin, l'utilisateur, par ses choix, peut activer des contraintes d'activation. Si l'utilisateur veut des jantes de 17 pouces, la contrainte d'activation suivante limitera aussi les valeurs possibles : « si la taille des jantes est supérieure à 16 pouces, alors le moteur doit développer une puissance d'au moins 90 chevaux ».

3.2.2 Configuration

Le principal problème lors de la propagation de contraintes est le maintien de la cohérence. Idéalement, le CSP doit rester globalement cohérent et restreindre la configuration aux produits faisables. Cependant, cette cohérence globale est très difficile à maintenir, il peut être laissé à l'utilisateur le choix soit de l'algorihme de cohérence, soit du niveau de cohérence, mais cela implique que l'utilisateur ait une bonne connaissance du problème.

La configuration doit donc permettre plusieurs tâches :

- assurer la cohérence du CSP, au besoin en restaurant celle-ci en cas de forçage (cf. 3.5.1, page 16) ;
- intégrer des fonctions d'optimisation d'une (ou plusieurs) fonctions objectifs pour permettre à l'utilisateur de ne pas faire certains choix et des fonctions de réduction du domaine de recherche (pour une variable continue) ou de restriction (par exemple, ôter une valeur possible d'une variable discrète de son domaine) ;

- fournir des explications lorsqu'une configuration n'est pas possible, et permettre la remise en cause de certains choix, si possible en évitant de devoir recommencer tout le procédé de configuration.

3.2.3 Maintenance et validation

La maintenance d'une configuration est utile lorsque le produit générique change. Il faut alors vérifier que toutes les configurations cohérentes correspondent à des produits physiquement réalisables. Pour cela, il faut tester les configurations, en enlevant de l'espace de recherche les informations qui ne participent à aucune solution. Cela revient à rendre le CSP minimal (le plus compact possible) ; cependant, cette opération est particulièrement complexe.

3.2.4 Avantages et inconvénients

Le formalisme CSP permet de modéliser facilement un produit générique et ses variantes grâce aux variables, et aux contraintes entre ces variables. Plusieurs méthodes permettent ensuite la résolution de CSP « classiques », parmi lesquelles nous pouvons citer les techniques :

- par exploration de l'arbre de recherche (peu efficace) :
  - ✓ *Generate & Test* ;
  - ✓ *Backtracking* ;
- par suppression de valeurs inconsistantes (propagation de contraintes grâce à des propriétés de cohérence locale) :
  - ✓ la cohérence de nœud *(node consistency – NC)* ;
  - ✓ la cohérence d'arc *(arc consistency – AC)* ;
  - ✓ la cohérence de chemin *(path consistency – PC)* ;
  - ✓ la k-cohérence *(k-consistency).*

Cependant, modéliser des produits complexes (réels) demande l'utilisation de CSP conditionnels[2] (ou dynamiques[2]) et mixtes, qui sont plus difficiles à résoudre. Ainsi, nous allons voir dans la section suivante un algorithme de résolution de CondCSP. Pour la suite du rapport, nous adopterons la définition d'un CondCSP donnée par [Gelle, Faltings 03]. Un CondCSP est un problème répondant aux caractéristiques suivantes :

- la formulation du problème contient des variables discrètes *et* numériques, il est donc mixte. Par voie de conséquence, des contraintes peuvent impliquer les deux types de variables, ce sont alors des contraintes mixtes ;
- certaines variables n'existent que sous certaines conditions ; l'existence de ces variables dans une solution donnée est donc conditionnelle (par exemple, sur un micro-ordinateur, la présence d'un disque dur SCSI provoque immanquablement la présence d'une carte SCSI, contrainte notée ainsi : $AC_1 : DD_{SCSI} \xrightarrow{ACT} \text{Carte}_{SCSI}$) ;
- les contraintes numériques, en particulier les inégalités, définissent des intervalles de valeurs autorisées. Un problème de contraintes numériques débouche souvent sur des intervalles de solutions, appelés sous-espaces de solutions[4].

Les CSP sont donc un bon moyen de modéliser un produit configurable, mais le nombre de solutions possibles est trop élevé. De plus, il n'est pas possible de prendre en compte les informations de relations entre certaines entités, telles qu'elles sont décrites dans les formalismes objet (cf. chapitre 4, page 18).

[2] Ayant un nombre de variables non fixé, variant selon les contraintes d'activation.

[4] Les méthodes de cohérence locale permettent une bonne approximation de ces intervalles en un temps de calcul raisonnable

3.3 Méthode de résolution : réduction

Les méthodes que nous allons voir ont été proposées par E. Gelle, en collaboration avec B. Faltings ou M. Sabin (cf. [Gelle, Faltings 03] et [Gelle, Sabin 03]).

Elles datent de cette année (2003) et proposent des méthodes de résolution de csp pour arriver à obtenir des résultats à partir de csp dynamiques ou mixtes. En effet, les méthodes actuelles ne permettent pas de prendre en compte de telles contraintes (mixtes et conditionnelles). Nous allons d'abord voir en quoi consistent de telles méthodes puis analyser les comparaisons faites par les auteurs sur des problèmes générés aléatoirement.

Les variables du csp généré sont les composants et leurs propriétés. Les contraintes définissent des relations entre les variables ; celles-ci peuvent être discrètes, mixtes[5] ou conditionnelles (contraintes d'activation, notées ainsi : $AC : X_1 \stackrel{ACT}{\longrightarrow} X_2$ signifie que la variable X_1 entraîne l'existence de la variable X_2).

3.3.1 Satisfaction de contraintes d'activation – définitions

Les contraintes d'un problème sont générallement complexes : le problème de configuration implique des contraintes aussi bien discrètes que numériques et mixtes.

Il existe différentes façons de représenter une contrainte ; pour une contrainte discrète, les valeurs admissibles sont généralement toutes listées. Les contraintes numériques sont définies par des formules et les contraintes mixtes peuvent être exprimées des deux façons. Une contrainte d'activation définit les conditions sous lesquelles une variable (conditionnelle) existe. Ces contraintes peuvent ainsi décrire l'existence nécessaire d'un composant requis par l'existence d'autres composants, ou par la valeur de certaines de leurs propriétés.

Le modèle Condcsp *(Conditional Constraint Satisfaction Problem)* est introduit pour utiliser des techniques de résolution de csp standards afin de résoudre des problèmes ayant un nombre évolutif de variables. Ainsi, pour définir le problème, nous avons :

- un ensemble de variables $\mathcal{V}$ – chaque variable X_i est associée à son domaine D_i ;
- un ensemble non-vide de variables initiales $V_1 \subseteq \mathcal{V}$, qui font partie de toutes les solutions ;
- un ensemble de contraintes de compatibilité, $\mathscr{C}^C \subseteq \mathscr{C}$ représentant les contraintes « classiques » ;
- un ensemble de contraintes d'activation, $\mathscr{C}^A \subseteq \mathscr{C}$ pour représenter « l'apparition » de nouvelles variables selon les valeurs des variables de V_1.

Une solution S à un tel problème de configuration (Condcsp) est l'affectation de valeurs aux variables de $\mathcal{V}$, tel que S satisfait $\mathscr{C}^A \cup \mathscr{C}^C$; de plus, des valeurs doivent être affectées à toutes les variables de V_1.

Nous définissons une solution *minimale* comme le plus petit sous-ensemble respectant les contraintes. Ainsi, si S est solution minimale, il n'existe pas de solution S' telle que $S' \subset S$ (cf. [Gelle, Faltings 03]).

Un premier algorithme de résolution donné par [Mittal, Falkeinhainer 90] fonctionne de la façon suivante :

1. pour une variable donnée, vérification de la satisfaction des conditions des contraintes d'activation ;
2. chaque contrainte de compatibilité déjà satisfaite est revérifiée ;
3. une variable non encore affectée est choisie et affectée ;

[5] Une contrainte mixte est définie par une formule entre des variables discrètes et continues

4. si une incompatibilité est détectée durant le processus de vérification des contraintes, retour sur la dernière variable affectée, qui peut être enlevée des variables actives si elle crée l'incohérence.

Cependant, cet algorithme de résolution a certaines faiblesses :

- quand la condition d'activation est une variable discrète avec beaucoup de valeurs possibles ;
- quand la condition d'activation est une variable numérique d'un intervalle réel.

Ainsi, cette méthode n'est pas très efficace pour résoudre un CondCSP ; elle peut en effet s'avérer très coûteuse en temps. [Gelle, Faltings 03] proposent donc un nouvel algorithme de résolution qui réduit un CondCSP en un ensemble de CSP standards ; les algorithmes existants peuvent alors être utilisés pour résoudre les CSP standards.

3.3.2 Réduction de CSP conditionnels en un jeu de CSP standards

Créer un graphe de dépendance

Le premier algorithme, introduit par MITTAL et FALKENHEINER en 1990, boucle sur toutes les contraintes d'activation. Pour une efficacité accrue dans la résolution de CondCSP, il faut prendre en compte l'ordre dans lequel les variables sont activées par les contraintes d'activation. L'ordre d'activation est déterminé par analyse du contexte des variables actives et des conditions pour lesquelles une contrainte d'activation devient satisfaite. Analyser les dépendances entre les contraintes d'activation a permis aux auteurs d'identifier un ordre dans lequel elles devraient être appliquées et d'éliminer les variables qui ne seront jamais activées. Ces dépendances peuvent être représentées dans un graphe, où chaque contrainte d'activation est un nœud, et un lien direct existe entre deux nœuds si la contrainte d'activation du premier nœud active une variable utilisée dans la contrainte d'activation du deuxième nœud. La création du graphe de dépendances fait donc intervenir les étapes suivantes :

1. créer un graphe représentant les dépendances entre les contraintes, depuis l'ensemble V_1 des variables initiallement actives ;
2. éliminer certains cycles dans le graphe ;
3. dériver un ordre total (ou plusieurs) depuis l'ordre partiel donné par les graphes acycliques directs, parcourir le graphe dans l'ordre, et appliquer les contraintes d'activation ;
4. quand une contrainte d'activation s'applique à un contexte de variables déjà actives, les solutions sont dans la combinaison de deux sous-espaces : celui où la nouvelle variable est active, et la condition doit être satisfaite, et celui où la condition ne doit pas être satisfaite.

Le problème de CSP conditionnel est alors réduit à un ensemble de CSP standards, chacun défini sur des ensembles de variables différents, donc un ensemble différent de containtes de compatibilité.

Une relation de dépendance DR est introduite pour déterminer l'ordre dans lequel les contraintes d'activation doivent être appliquées. Ainsi, si une contrainte d'activation $C_n \overset{ACT}{\longrightarrow} X_n$ dépend d'une autre contrainte d'activation $C_1 \overset{ACT}{\longrightarrow} X_1$, on note cette dépendance de la façon suivante : $C_n \overset{ACT}{\longrightarrow} X_n \; DR \; C_1 \overset{ACT}{\longrightarrow} X_1$. En cas de cycle de dépendance, les dépendances sont représentées dans un graphe de dépendances $\mathcal{G} = \langle \mathcal{H}, \mathcal{U} \rangle$ où :

- l'ensemble des nœuds $\mathcal{H}$ forme l'ensemble des contraintes d'activation. V_1 est alors le nœud « racine ».

- Les liens $\mathcal{U}$ sont définis comme suit : il y a un lien entre deux nœuds si la condition du second nœud contient une variable activée par le premier nœud.

Pour réduire la complexité du graphe de dépendance, il faut faire disparaître le phénomène de cycle. Pour celà, [Gelle, Faltings 03] cherchent des nœuds dont la connectivité est forte ; c'est à dire que tous leurs liens directs participent à au moins un cycle.

Le graphe réduit $\mathscr{G}_R$ est ainsi construit, en regroupant en particulier les cycles à l'intérieur du graphe des dépendances. Les liens de $\mathscr{G}_R$ représentent les relations de dépendance entre contraintes d'activation.

Les contraintes peuvent alors être ordonnées en comparant les chemins pour aller du nœud « racine » à un nœud donné. Certains nœuds étant à la même distance du nœud « racine », leur ordre de traitement est alors fixé de manière arbitraire. Ces nœuds sont dits « incomparables ».

Appliquer les contraintes d'activation

Le graphe des dépendances est alors traité en appliquant les contraintes d'activation de chaque nœud. Deux cas se distinguent : le premier, où un nœud de $\mathscr{G}_R$ correspond à une seule contrainte d'activation ; le deuxième, où un nœud correspond à plusieurs contraintes d'activation (ce cas est en fait plus simple à traiter).

Cas 1 Pour résoudre les problèmes d'espace, [Gelle, Faltings 03] introduisent le complément $\overline{C}$ de C, qui est l'ensemble des valeurs non autorisées dans C pour une contrainte discrète. Pour une contrainte numérique, c'est le complément sur l'ensemble de définition des valeurs du domaine de définition autorisées par C, réduit aux inégalités (pour une égalité, deux inégalités seraient nécessaires pour décrire $\overline{C}$).

Le symbole P est ensuite défini comme l'espace du problème, constitué par un jeu de variables actives $\mathcal{V}_P$, un ensemble de contraintes $\mathscr{C}_P$ et une contrainte d'activation $C \xrightarrow{ACT} X$. Supposons alors que toutes les variables de C soient actives ; la contrainte d'activation $C \xrightarrow{ACT} X$ divise alors P en trois parties :

- P_1 tel que :
$$\mathcal{V}_{P_1} = \mathcal{V}_P \cup \{X\},$$
$$\mathscr{C}_{P_1} = \mathscr{C}_P \cup \{C\},$$
- P_2 tel que :
$$\mathcal{V}_{P_2} = \mathcal{V}_P \cup \{X\},$$
$$\mathscr{C}_{P_2} = \mathscr{C}_P \cup \{\overline{C}\},$$
- P_3 tel que :
$$\mathcal{V}_{P_3} = \mathcal{V}_P,$$
$$\mathscr{C}_{P_3} = \mathscr{C}_P \cup \{\overline{C}\},$$

[Gelle, Faltings 03] pouve alors le lemme suivant :

Lemme 3.1 ([Gelle, Faltings 03]) *Soient un espace du problème P et une contrainte d'activation $AC : C \xrightarrow{ACT} X$, tels que Vars(C) $\subseteq \mathcal{V}_P$. Seuls les sous-espaces P_1 et P_3 du Cond*CSP *$P \wedge AC$ contiennent des solutions minimales.*

En considérant ensuite un ensemble de contraintes d'activation, et en se basant sur $\mathscr{G}_R$ (pas de dépendances cycliques), on obtient le théorème suivant :

Théorème 3.1 ([Gelle, Faltings 03]) *Les solutions minimales d'un ensemble de contraintes d'activation $AC_i : C_i \stackrel{ACT}{\longrightarrow} X_i$ $(i = 1, \ldots, n)$ d'un Cond*CSP *se trouvent dans le produit cartésien $\{P_{11}, P_{13}\} \times \cdots \times \{P_{n1}, P_{n3}\}$ formé par tous les sous-espaces P_{i1}, P_{i3}.*

Cas 2 Dans ce cas, il faut appliquer toutes les contraintes aux *super*-nœuds (qui en contiennent plusieurs). Certaines contraintes d'activation peuvent alors se révéler redondantes ; dans ce cas, elles peuvent être supprimées de l'espace de recherche.

Corollaire 3.1 ([Gelle, Faltings 03]) *Soient un espace de problème P avec une variable X active et une contrainte d'activation $AC : C \stackrel{ACT}{\longrightarrow} X$, telle que les variables de C soient aussi actives. L'union des sous-espaces minimaux P_1 et P_3 est égale à l'espace d'origine P.*

L'algorithme pour les CSP conditionnels

Pour un CondCSP donné – $\mathcal{P} = \langle \mathcal{V}, \mathscr{C}, \mathscr{D}, V_1 \rangle$ – comportant des variables numériques et discrètes, l'algorithme CondCSP détaillé figure 3.1, page 14 (tiré de [Gelle, Faltings 03]) génère successivement tous les espaces de problèmes (CSP standards) contenant les solutions minimales. Cet algorithme commence au nœud « racine », composé d'un ensemble de variables actives V_1.

- La fonction **relevant-constraint** sert à identifier toutes les contraintes de compatibilité dont les variables sont actives.
- En utilisant les relations de dépendances (DR) et le graphe réduit $\mathscr{G}_R$, la fonction **compute-order** génère une liste ordonnée des contraintes d'activation : $\mathscr{C}^A_{ord}$.

L'algorithme principal **CondCSP** fonctionne alors de la façon suivante. Une contrainte d'activation AC est extraite de $\mathscr{C}^A$ si toutes les variables de sa condition sont actives (**get-relevant-AC**). L'algorithme construit alors les deux sous-espaces par application des deux fonctions **with-condition** et **with-neg-condition**. Les deux appels donnent un nouvel ensemble de variables actives, et un nouveau jeu de contraintes pertinentes, soit un nouveau CSP standard. Cet algorithme est récursivement appliqué, jusqu'à ce que toutes les contraintes de $\mathscr{C}^A_{ord}$ aient été traitées.

procedure ConCSP$(\langle \mathcal{V}, \mathscr{C}^C \cup \mathscr{C}^A, V_1 \rangle)$:
$Vact \leftarrow V_1$
$\mathscr{C}^C_{rel} \leftarrow$ **relevant-constraint**$(Vact, \mathscr{C}^C)$
$\mathscr{C}^A_{ord} \leftarrow$ **compute-order**$(\mathscr{C}^A)$
CondCSP-main$(Vact, \mathscr{C}^C_{rel}, \mathscr{C}^A_{ord}, \mathscr{C}^C)$

function compute-order$(\mathscr{C}^A)$:
$\mathscr{G} \leftarrow$ graphe des dépendances depuis $\mathscr{C}^A$ et V_1
$\mathscr{G}_R \leftarrow$ graphe réduit issu de $\mathscr{G}$ (nœud racine N_0)
$\mathscr{C}^A_{ord} \leftarrow$ ordonne N_1 selon les chemins les plus longs dans $\mathscr{G}_R$
return $\mathscr{C}^A_{ord}$

procedure CondCSP-main$(Vact, \mathscr{C}^C_{rel}, \mathscr{C}^A_{ord}, \mathscr{C}^C)$:
$AC \leftarrow$ **get-relevant-AC**$(Vact, \mathscr{C}^A_{ord})$
enlever AC de $\mathscr{C}^A_{ord}$
if $AC \neq \emptyset$ then
$P_1 \leftarrow$ **with-condition**$(AC, Vact, \mathscr{C}^C_{rel}, \mathscr{C}^C)$
if $P_1 \neq \emptyset$ then **CondCSP-main**$(V_{P_1}, C_{P_1}, \mathscr{C}^A_{ord}, \mathscr{C}^C)$ fi
$P_2 \leftarrow$ **with-neg-condition**$(AC, Vact, \mathscr{C}^C_{rel})$
if $P_2 \neq \emptyset$ then **CondCSP-main**$(V_{P_2}, C_{P_2}, \mathscr{C}^A_{ord}, \mathscr{C}^C)$ fi
else nouvelle solution : $\langle Vact, C_{rel} \rangle$ fi

function with-condition$(C_i \xrightarrow{ACT} X_i, Vact, \mathscr{C}^C_{rel}, \mathscr{C}^C)$:
$Cnew \leftarrow \mathscr{C}^C_{rel} \cup \{C_i\}$
$Vnew \leftarrow Vact \cup \{X_i\}$
$Cnew \leftarrow Cnew \cup$ **relevant-constraint**$(Vnew, \mathscr{C}^C)$
if **locally-consistent ?**$(\langle Vnew, Cnew \rangle)$ then
return$(\langle Vnew, Cnew \rangle)$
else **return**$(\emptyset)$ fi

function with-neg-condition$(C_i \xrightarrow{ACT} X_i, Vact, \mathscr{C}^C_{rel})$:
$Cnew \leftarrow \mathscr{C}^C_{rel} \cup \{\overline{C_i}\}$
if **locally-consistent ?**$(\langle Vact, Cnew \rangle)$ then
return$(\langle Vact, Cnew \rangle)$
else **return**$(\emptyset)$ fi

FIG. 3.1 – Algorithme générant un CSP standard contenant les solutions minimales

3.4 Résultats

Il existe des algorithmes de résolution directe de CondCSP [6]. Cette section présente les comparaisons entre ces algorithmes et l'algorithme de réduction en CSP standards ; les résultats expérimentaux sont tirés de [Gelle, Sabin 03].

Avec un générateur de CondCSP aléatoires, 100 problèmes différents ont été testés. Chacun comportait 8 variables discrètes de 6 valeurs différentes possibles. Les auteurs ont montré que l'algorithme de réduction en CSP standards était beaucoup plus rapide. Six combinaisons de résolution ont été testées :

- avec l'algorithme de résolution directe des CondCSP (CCSP)
  - ✓ et la méthode du *backtracking* (BT) ;
  - ✓ et la méthode *forward checking* (FC) ;
  - ✓ et la maintenance de cohérence d'arc (MAC - *maintaining arc-consistency*) ;
- avec l'algorithme de réduction en CSP standards (SCSP)
  - ✓ et la méthode du *backtracking* (BT) ;
  - ✓ et la méthode *forward checking* (FC) ;
  - ✓ et la maintenance de cohérence d'arc (MAC - *maintaining arc-consistency*).

Le résultat est visible sur la figure 3.2, page 16. Tout d'abord, nous pouvons constater que la méthode du *backtraking* demande beaucoup plus de temps que les autres. Les deux méthodes *forward checking* et par maintenance de cohérence d'arc ont des résultats comparables, il est donc possible de juger de la performance de la réduction en CSP standards par rapport à la résolution directe. Pour des degrés de satisfaction peu importants (jusqu'à 0,7), la méthode de réduction est environ deux fois plus rapide. En augmentant le degré de satisfaction, cette proportion augmente encore, jusqu'à atteindre un ratio de 3 pour 0,9.

Les auteurs en ont donc déduit l'efficacité de cette nouvelle méthode de résolution, en particulier en termes de rapidité, par rapport à une résolution directe du CondCSP. D'autres algorithmes de recherche de solutions existent, on peut citer [Codognet, Diaz] pour une méthode de recherche locale.

[6] cf. [Sabin 03]

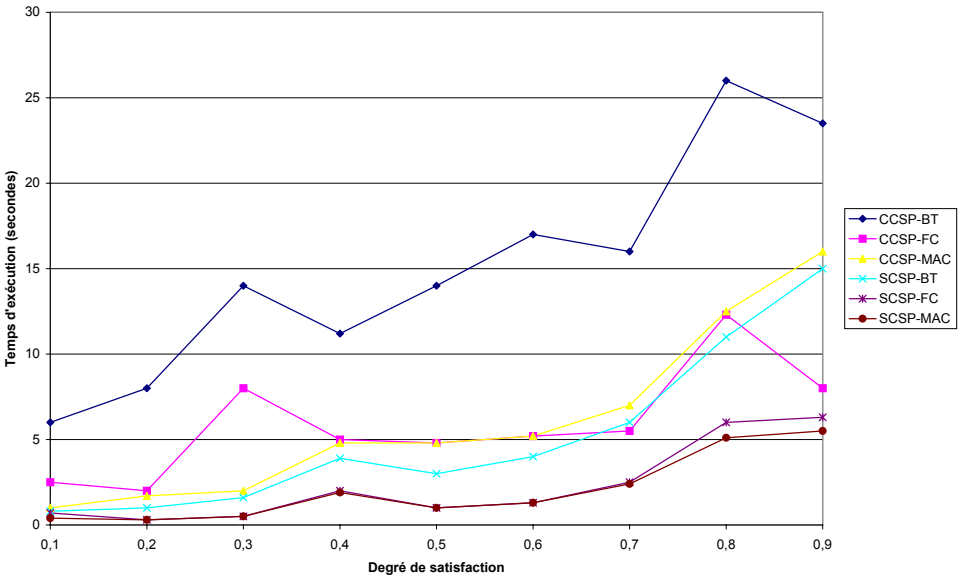

FIG. 3.2 – Temps d'exécution moyen de différentes méthodes de résolution en fonction du degré de satisfaction des contraintes de compatibilité

3.5 Cohérence et optimisation

3.5.1 Cohérence – dialogue avec l'utilisateur

Parmi les objectifs d'une tâche de configuration, il faut aussi citer la restauration de la cohérence et fournir des explications à l'utilisateur. Ces deux objectifs sont liés. En remontant à l'étape qui fait perdre la cohérence au système, il faut essayer d'expliquer à l'utilisateur pourquoi l'étape suivante a constitué un problème.

Pour maintenir la cohérence, il faut s'assurer qu'à chaque instant, les choix de l'utilisateur sont cohérents avec le CSP modélisant le système. Dans l'idéal, cette cohérence doit être globale ; il faut au minimum détecter une incohérence le plus tôt possible. Il faut aussi évaluer les conséquences des choix de l'utilisateur en restaurant ou effaçant toutes les valeurs compatibles ou incompatibles avec ces choix.

Ensuite, il faut fournir à l'utilisateur des fonctionnalités permettant de revenir en arrière et de répondre à des questions du type : « Quel choix puis-je changer pour retrouver la cohérence ? » ou « Quelles valeurs changer pour avoir telle valeur disponible pour cette variable ? ». Le problème devient alors d'identifer les sous-espaces cohérents des choix en cours. Deux choix de stratégies se présentent : soit maximiser la taille de ces sous-espaces, soit identifier des sous-espaces permettant d'optimiser une fonction objectif.

Enfin, conséquence logique, le système doit pouvoir répondre à des questions du type : « D'où vient l'incohérence (de quel sous-espace ou plus précisément, de quelle variable) ? » ou « Pourquoi cette valeur n'est plus disponible pour telle variable ? ». Il faut alors identifer des sous-espaces incohérents minimaux.

Pour résumer, [Aldanondo *et al.* 01] parlent de « forçage ». Les auteurs distinguent trois cas :

pas de forçage possible : sans forçage, il faut absolument permettre à l'utilisateur de pouvoir identifier les choix qui sont à l'origine de l'incohérence, puis de pouvoir les modifier. Ce mode de fonctionnement est très robuste, mais il doit fournir des explications à l'utilisateur ;

forçage non destructif avec perte de cohérence : l'utilisateur peut forcer le système et choisir une valeur interdite. Le CSP devient alors incohérent. Ce fonctionnement est très souple, mais il nécessite des fonctionnalités d'aide à la restauration de cohérence. En effet, il ne sert à rien pour l'utilisateur de forcer une variable s'il n'a pas la possibilité de revenir sur un produit faisable ;

forçage destructif sans perte de cohérence : l'utilisateur peut forcer une variable, mais un processus autonome va alors modifier une autre variable pour que le CSP reste cohérent – ce mécanisme annule des choix antérieurs sans demander de validation, il peut donc être dangereux (*i.e.* aboutir à un produit ne correspondant pas aux souhaits de l'utilisateur).

Pour éviter de se trouver dans des cas devant faire appel à ces mécanismes, il est préférable d'utiliser une fonction d'aide à la configuration, par exemple en demandant à l'utilisateur de réduire les domaines de variables, mais en laissant plusieurs possibilités au système.

3.5.2 Optimisation

Une fois que l'utilisateur a effectué certains choix, il peut décider de laisser le système optimiser pour lui une fonction objectif. C'est par exemple le cas lorsqu'un client veut acheter un micro-ordinateur ayant 40 Go de disque dur, 256 Mo de mémoire vive et un processeur cadencé à au moins 1,4 GHz ; il veut alors trouver l'offre la moins chère disposant de ces caractéristiques. La fonction objectif est alors le prix, mais toutes sortes d'autres objectifs peuvent être introduits : délai, coût de fabrication, ...

Chapitre 4

Méthodes objet adaptées à la configuration

4.1 Concepts

L'idée générale d'utiliser des modèles UML est venue du fait que beaucoup de produits sont déjà modélisés par cette méthode, qui permet une maintenance relativement facile. Le modèle sert ici à générer des CSP dynamiques, dont la résolution est abordée dans le chapitre précédant.

La première étape est de modéliser la structure du produit générique, englobant toutes les variantes possibles. Les solutions non viables sont écartées par l'expression de contraintes par rapport aux objets ; par exemple, une voiture a besoin *(requires)* d'un moteur, qui peut être soit diesel, soit essence (ou autre ...) – ces deux types de moteurs sont mutuellement exclusifs (« essence » *incompatible* « diesel »). Ces contraintes entre objets sont exprimées dans le diagramme de classe. Certaines contraintes ne peuvent pas être exprimées sur le diagramme de classe. La méthode OCL est alors utilisée, en particulier pour des contraintes de cardinalité entre différents objets, ou pour toute contrainte numérique (par exemple, le poids du moteur ne doit pas dépasser 250 kg).

L'avantage d'utiliser UML est de disposer de toutes les fonctionnalités de cette méthode pour structurer l'information et ainsi de gagner la maintenabilité assurée par les technologies orientées objet.

4.2 Construction d'une base de connaissances

Une fois le modéle UML du produit fait, des régles de passage permettent de passer du modèle conceptuel à une représentation logique, plus proche des CSP. Ces techniques débouchent alors sur des CSP hiérarchiques, qui reflètent le concept d'héritage issu des méthodes de modélisation orientées objet. Une contrainte valable pour un type d'objet sera aussi valable pour tous ses sous-types ; si le poids du moteur ne doit pas dépasser 250 kg, c'est valable aussi bien pour un moteur essence que diesel.

Dans [Felfernig *et al.* 01], les auteurs définissent un problème de configuration de la façon suivante :

- un domaine de description (DD) qui contient des informations sur les types de composants, leurs propriétés et des contraintes ;
- une description des besoins de l'utilisateur (SRS) – la résolution du problème de configuration devant satisfaire ces exigences ;

- un ensemble $CONL$ de variables utilisées pour décrire les résultats de la configuration.

Ainsi, étant donné un problème de configuration (DD, SRS, $CONL$), une configuration $CONF$ est cohérente (respecte les contraintes) si et seulement si la condition $DD \cup SRS \cup CONF$ est remplie. Par extension, une configuration valide $\overline{CONF}$ est définie comme une configuration cohérente *et* complète.

En définissant un modèle logique de configuration et une notation donnée pour le niveau conceptuel, il est alors possible d'obtenir la base de connaissances DD automatiquement, en fixant des règles de transformation déterministes. De plus, les contraintes additionnelles, comme les relations *requires* et *incompatible* peuvent aussi être transformées et ajoutées à DD. Par exemple, une relation impossible entre deux composants pour cause d'incompatibilité entre deux composants X_1 et X_2 sera notée ainsi (avec *port* un type de connexion) :

$$type(X_1) \wedge type(X_2) \wedge conn(X_1, X_2, port) =: false$$

Pour plus de détails sur les règles de transformation, cf. les documents suivants :

- Felfernig A, Friedrich G, Jannach D. UML as domain specific language for the construction of knowledge-based configuration systems. Int J Software Engng Knowledge Engng (IJSEKE) 2000 ;10(4) :449-70 ;
- Felfernig A, Friedrich G, Jannach D. Generating product configuration knowledge bases from precise domain extended UML models. *In* : Proceedings of the 12^{th} International Conference on Software Engineering and Knowledge Engineering (SEKE'00), Chicago, USA, 2000, p. 284-93.

4.3 Diagnostic de la base de connaissances

En vue de la maintenance de la base de connaissances (issue de la modélisation UML), il faut fournir des exemples de configuration, complets ou non. Il faut fournir des exemples cohérents *et* incohérents. Soient E^+ l'ensemble des exemples cohérents et E^- l'ensemble des exemples sur-contraints.

Lors de la phase de validation de la base de connaissances, il faut bien vérifier que tous les exemples $e^+ \in E^+$ soient bien consistants ; de même, tous les exemples $e^- \in E^-$ doivent être incohérents. Si ce n'est pas le cas, il faut alors changer des contraintes.

Bien entendu, ces deux ensembles sont complémentaires ; ils servent à vérifier qu'une configuration valide soit acceptée, et que les contraintes soient assez restrictives pour refuser une « mauvaise » configuration.

Ainsi, [Felfernig *et al.* 01] définissent un problème de diagnostic CKB (*Configuration Knowledge Base*) comme un triplet (DD, E^+, E^-). En nommant NE l'ensemble des exemples négatifs niés ($NE = \bigwedge(\neg e^-)$, $e^- \in E^-$) et S le diagnostic, les auteurs en arrivent au corollaire suivant :

Corollaire 4.1 ([Felfernig *et al.* 01]) *S est un diagnostic si et seulement si, $\forall e^+ \in E^+$, $DD - S \cup e^+ \cup NE$ est cohérent.*

Pour identifier les problèmes dans la configuration, ils introduisent CS comme l'ensemble conflictuel : $CS \in DD$ et $\exists e^+ \in E$ tel que $CS \cup e^+ \cup NE$ est incohérent. C'est alors l'exemple e^+ qui induit le conflit.

Ce procédé de diagnostic génère un ensemble de diagnostics permettant d'interpréter les comportements inattendus du configurateur. Ainsi, chaque diagnostic S contient un

ensemble de contraintes de DD à revoir pour disposer d'une base de connaissances efficace. Il est ensuite possible d'envisager plusieurs cas pour la réparation de la base : soit une réparation automatique (en cas de contraintes trop contraignantes, par exemple) ou produire une sortie décrivant les contraintes posant des problèmes pour les testeurs. C'est alors un ingénieur métier qui est chargé de la reconstruction de la base (vérification de contraintes, etc.).

4.4 Diagnostic et reconfiguration

4.4.1 Vœux de l'utilisateur

Une fois la base de connaissance correctement générée, il faut pouvoir évaluer les vœux de l'utilisateur, en termes de cohérence. La méthode de diagnostic présentée dans [Felfernig *et al.* 01] permet d'assister l'utilisateur dans la recherche du (ou des) choix qui « créent » l'incohérence. En utilisant les souhaits de l'utilisateur SRS et en les comparant au domaine de définition DD, on peut retrouver les propositions appartenant à SRS qui contradisent DD. Ainsi, le but du diagnostic est d'identifier un ensemble S ($S \subseteq SRS$) tel que $SRS - S \cup DD$ soit cohérent. Il est aussi possible de chercher l'ensemble conflictuel CS défini ainsi : $CS \subseteq SRS$ tel que $CS \cup DD$ soit incohérent.

4.4.2 Reconfiguration

Cette fonctionnalité est particulièrement demandée pour l'après-vente (maintenance, ...). Lorsque le client veut ajouter des fonctionnalités à un produit ou changer une pièce par sa nouvelle version, il faut assurer la compatibilité avec la configuration initiale, ceci parce que les anciens composants peuvent ne plus être disponibles.

La principale différence entre configuration et reconfiguration repose sur le fait que la reconfiguration est basée sur une configuration déjà existante. Pour la reconfiguration, le nombre de paramètres à changer doit être le plus petit possible afin de préserver la plus grande partie possible de la configuration initiale. On peut ensuite associer à la reconfiguration différentes fonctions d'optimisation, basées sur de nouveaux critères, pour affecter de nouveaux coûts à certains changements. Enfin, il faut aussi veiller à ce que les fonctionnalités additionnelles soient apportées avec le moins de changements possible.

Dans ce procédé de reconfiguration, la configuration existante $CONF$ n'est plus cohérente avec les nouveaux souhaits de l'utilisateur SRS et le domaine de défintion DD. S'il est impossible d'étendre la configuration $CONF$ pour remplir les nouveaux besoins, il faut tout d'abord chercher des sous-ensembles de $CONF$ qui peuvent être ôtées du problème ou modifiées. Cette configuration doit alors être complétée – en opérant le moins de changements possibles – pour qu'elle convienne aux nouveaux besoins.

Reconstruire une configuration complète est évidemment possible, mais on ne prend alors pas en compte ce dont dispose déjà l'utilisateur, et on court donc le risque de devoir tout changer ; pour l'utilisateur, cela revient à se procurer un nouveau produit. Pour prévenir ce risque, l'utilisateur peut, dans ses nouveaux souhaits SRS, ajouter des expressions concernant les parties de sa configuration actuelle qui ne doivent surtout pas changer.

Trouver une solution à un problème de reconfiguration (DD, SRS, $CONF$, $CONL$), où $CONF$ représente la configuration existante, revient donc à résoudre le problème de configuration (DD, $SRS \cup (CONF - S)$, $CONL$), où $S \subseteq CONF$ tel que $CONF - S \cup DD \cup SRS$ soit cohérent. Les algorithmes de résolution de problèmes de configuration peuvent alors être appliqués.

4.4.3 Application à des produits complexes

Certains produits étant particulièrement complexes à (re)configurer, on se contente alors de fournir non pas une solution optimale à l'utilisateur, mais un ensemble de solutions convenables (de coûts pas trop exhorbitants, ...). C'est ensuite l'utilisateur qui fait son choix entre ces différentes configurations possibles.

Chapitre 5

Un exemple de configurateur industriel

L'exemple étudié ici est celui de ILOG *(J)Configurator*, d'après [Junker, Mailharro 03]. Ce système se base sur la description logique de contraintes et la résolution à l'aide de techniques de programmation sous contraintes.

5.1 Logique du configurateur

Tout d'abord, le produit est modélisé grâce à une technique orientée objet. Chaque composant est représenté par une classe (éventuellement abstraite, s'il existe plusieurs types d'un même composant). Ainsi, une classe concrète est une classe primitive, alors qu'une classe abstraite est l'agrégation d'autres classes (qui peuvent être abstraites ou concrètes).

Chaque classe possède un certain nombre de propriétés, qui modélisent les attributs des composants et les relations entre ces composants. Il existe deux types de propriétés : les propriétés d'objet, qui désignent une relation entre deux objets et les propriétés de données, relations entre un objet et une donnée (numérique, ...). La distinction entre des propriétés qui ne peuvent avoir qu'une valeur pour un objet (prix, ...) et d'autres qui peuvent avoir plusieurs valeurs (par exemple, nombre et type de disques durs pour un micro-ordinateur, ...) peut aussi être faite. Une propriété additionnelle est déclarée pour chaque objet, qui représente le type de l'objet en question (le nom de la classe concrète à laquelle il appartient).

Pour faciliter l'interprétation de ces propriétés, chacune d'elles est transformée en une fonction :

- une propriété avec une seule valeur devient une fonction π_P ; la propriété $P(x, y)$ devient donc la fonction $\pi_P(x) = y$;
- une propriété avec plusieurs valeurs $P(x, y)$ devient la fonction Π_P telle que $y \in \Pi_P(x)$.

Le domaine de la fonction détermine ensuite le type du champ. Pour une fonction π_P :

- les champs numériques sont des entiers ou des réels ;
- les champs textes ont des valeurs choisies parmi une liste (valeurs discrètes) ;
- les champs objets ont une valeur choisie parmi les autres objets, ils représentent une relation entre les deux objets ;
- le champ de classe est utilisé pour représenter le type de l'objet.

Pour des champs à multiples valeurs possibles (fonction Π_P), on retrouve les champs de

type objet ou classe, mais sous forme d'ensembles. À chacun de ces champs est alors associé une cardinalité k. On peut ainsi définir un ordre et transformer une fonction Π_P en une fonction $\pi_{p,i}$ avec $i = 1, \ldots, k$.

5.2 Programmation sous contraintes

Ces fonctions sont ensuite exprimées en termes d'expressions, en utilisant les symboles suivants :

- pour les expressions numériques, les opérateurs classiques sont utilisés : somme, différence, produit, division, valeur absolue, ... ;
- pour les expressions portant sur des objets ou des classes, la logique de fonctionnement demande d' utiliser les opérateurs logiques suivants : intersection, union, différence, inclusion ;
- les opérateurs de négation et la cardinalité sont aussi disponibles pour quasiment toutes les expressions.

Les contraintes peuvent donc être exprimées sous forme d'inégalités pour les contraintes numériques et d'expressions logiques pour les variables discrètes. Ainsi, cette approche permet de générer un problème de configuration proche du formalisme CSP.

5.3 Exemple

Soit `monPC` un produit configurable (ordinateur personnel). Le constructeur du processeur doit être choisi parmi les entreprises suivantes : *Intel*, *IBM*, *Apple*, ... Des objets sont alors ajoutés (connectés) à `monPC`, par exemple des diques durs. La propriété objet `avoirDD` est alors définie ainsi : `avoirDD(monPC, DD1)`. Il est aussi possible de définir des propriétés numériques, par exemple, la propriété `coûte` : `coûte(DD1, 100)`. Ici, la fonction `coûte` a toujours une cardinalité de 1 : un objet ne peut avoir deux prix différents.

Une fois toutes les relations et propriétés des objets exprimées, il faut fixer les choix de l'utilisateur. L'utilisateur fixe donc les contraintes : par exemple, `somme(DD(monPC), prix)` ≤ 200. D'autres contraintes peuvent être fixées par les ingénieurs métier, pour éviter d'obtenir un produit « inefficace » ou non réalisable, par exemple, le nombre de disques durs IDE ne peut être supérieur au nombre de ports IDE du micro-ordinateur.

5.4 Conclusion

Même avec cette description particulièrement simpliste du configurateur d'ILOG, nous constatons que la démarche suivie s'approche du schéma 2.2, page 5. Ainsi, un produit configurable est modélisé selon une approche objet (UML). En passant par une logique descriptive de ce produit, le configurateur permet de générer un CSP complexe en plusieurs étapes (passage par des fonctions, puis des expressions). Une fois un problème de type CSP obtenu, il est possible d'utiliser tout l'arsenal des techniques de résolution de ces problèmes.

Pour trouver plus d'informations sur la société ILOG et ses produits, cf. [ILO].

5.5 Autres configurateurs industriels

Depuis quelques années, quasiment tous les éditeurs de Progiciels de Gestion Intégré (PGI – ERP : *Enterprise Ressources Planning*) proposent un configurateur. Ce type d'outil ne fait pas appel aux techniques de progammation sous contraintes que nous avons vues. Il s'agit plutôt d'un type de catalogue, car ses logiciels ne gèrent pas les problèmes complexes, avec beaucoup de contraintes.

En revanche, d'autres éditeurs sont spécialisés dans ces logiciels, et fournissent des solutions pour résoudre des CSP de plus en plus complexes. Le point faible de ces logiciels reste leur convivialité.

Chapitre 6

Conclusion

Configuration, CSP et modèles objets

La configuration est une problématique industrielle qui vise à répondre parfaitement aux attentes du client. Nous l'avons vu dans le chapitre 2, page 2, la configuration est un intermédiaire entre la vente et la production.

L'utilisation des techniques de résolution de CSP est apparue assez tôt dans les problèmes de configuration exprimés en termes de contraintes. Cependant, les CSP générés par des problèmes de configuration ne sont généralement pas des CSP standards, car les contraintes peuvent impliquer des variables mixtes, et activer d'autres variables, inutilisées au départ. Nous avons vu un bref aperçu de techniques de résolution de ces problèmes dans le chapitre 3, page 6, en particulier en générant des CSP standards à partir de CSP conditionnels.

Enfin, une des nouvelles problématiques de la configuration est de générer des problèmes de type CSP à partir des méthodes de modélisation utilisées en entreprise. Nous avons vu dans les deux derniers chapitres comment utiliser des modélisations objets de produits pour générer des problèmes exprimés en termes de contraintes.

Avancement de la recherche

Les techniques de résolution de CSP standards sont au point. En revanche, des recherches sont en cours pour proposer de nouvelles techniques de résolution de CSP complexes (Gelle, Sabin, ...).

De même, sur le sujet de la génération de CSP à partir de modèles objets de produits, de nombreuses recherches sont en cours, à la limite de la gestion industrielle et de l'intelligence artificielle. On peut citer en particulier Felfernig, Soininen, Friedrich et Männistö.

Enfin, il faut souligner qu'au moment où j'écris ces lignes, le dix-huitième congrès de l'*IJCAI (International Joint Conference on Artificial Intelligence)* a lieu à Acapulco, au Mexique.

Sujet de thèse

Le sujet a été proposé par le centre *Génie Industriel* de l'École des Mines d'Albi.

L'objet de la thèse est d'étudier les différentes méthodes qui permettraient éventuellement de prendre en compte des contraintes issues de problèmes industriels (donc mixtes, et en particulier géométriques) via des CSP. Les méthodes envisagées peuvent être par

discrétisation de l'espace, par modélisation directe au sein d'un moteur de CSP mixte, par l'ajout d'un module externe spécifique, etc. Toutes ces méthodes devront être évaluées et comparées tant en terme de performance qu'en terme d'adaptation des réponses fournies aux besoins réels. Il pourra aussi être envisagé de fournir une interface adaptée à l'utilisateur, lui permettant de définir ses besoins en termes de fonctions.

Glossaire

CSP : Constraint Satisfaction Problem – Problème devant satisfaire des contraintes i, ii, 1, 3, 4, 6–18, 23–25, 29

OCL : Object Constraint Language, méthode d'expression de contraintes entre les objets, fait partie d'UML 4, 18

UML : Unified Modeling Language, méthode de modélisation orientée objet ii, 4, 18, 19, 23

Bibliographie

Références bibliographiques

[Aldanondo *et al.* 01] Michel Aldanondo, Hélène Fargier & Matthieu Veron. Configuration, configurateur et gestion de production, chapitre 7. Hermes Science, Paris, 2001. Traité IC2 Productique.

[Amilhastre *et al.* 02] Jérôme Amilhastre, Hélène Fargier & Pierre Marquis. *Consistency restoration and explanations in dynamic* CSP*s – Application to configuration.* Artificial Intelligence, vol. 135, 2002.

[Barták 99] Roman Barták. *Constraint Programming – What is Behind.* In Proc. Workshop on Constraint Programming in Decision and Control, Pologne, 1999.

[Bensana, Mulyanto 00] Éric Bensana & Taufiq Mulyanto. *A generic approach for conceptual design based on object oriented and constraint logic programming.* In International Conference on Engineering Design and Automation, Orlando, Floride, 2000.

[Cooper 89] Martin Cooper. *An optimal k-consistency algorithm.* Artificial intelligence, vol. 41, 1989.

[Felfernig *et al.* 01] Alexander Felfernig, Gerhard Friedrich & Dietmar Jannach. *Conceptual modeling for configuration of mass-customizable products.* Artificial intelligence in Engineering, vol. 15, 2001.

[Gelle, Faltings 03] Esther Gelle & Boi Faltings. *Solving mixed and conditional constraint satisfaction problems*, 2003. Constraints.

[Gelle, Sabin 03] Esther Gelle & M. Sabin. *Solving Methods for Conditional Constraint Satisfaction.* In Proc. IJCAI'03, Acapulco, Mexique, 2003.

[Junker, Mailharro 03] U. Junker & D. Mailharro. *The Logic of ILOG (J)Configurator : Combining Constraint Programming with a Description Logic.* In Proc. IJCAI'03, Acapulco, Mexique, 2003.

[Lottaz 99] Claudio Lottaz. *Rewriting numerical constraint satisfaction problems for consistency algorithms*, 1999.

[Mackworth 77] Alan Mackworth. *Consistency in networks of relations.* Artificial intelligence, vol. 8, 1977.

[Männistö *et al.* 01] Tomi Männistö, Timo Soininen & Reijo Sulonen. *Modelling configurable products and software product families.* In Proc. IJCAI'01, Seattle, Washington, 2001.

[Mittal, Falkeinhainer 90] Sanjay Mittal & Brian Falkeinhainer. *Dynamic Constraint Satisfaction Problem.* In Proc. AAAI'90, Boston, Massachusetts, 1990.

[Mittal, Frayman 89] S. Mittal & F. Frayman. *Toward a Generic Model of Configuration Tasks*. In Proc. IJCAI'89, Detroit, Michigan, 1989.

[Montanari 74] H. Montanari. *Networks of constraints : fundamental properties and application to picture processing*. Information sciences, vol. 7, 1974.

[Sabin 03] Mihaela Sabin. *Towards Improving Solving Conditional Constraint Satisfaction Problems*. PhD thesis, University of New Hamshire, 2003.

[Schiex 00] Thomas Schiex. *Réseaux de contraintes*, 2000. HDR.

[Silaghi *et al.* 00] Marius-Calin Silaghi, Djamila Sam-Haroud & Boi Faltings. *Fractionnement intelligent de domaine pour* CSP *avec domaines ordonnés*. In Proc. RFIA'00, Paris, 2000.

[Soininen *et al.* 99] Timo Soininen, Esther Gelle & Ilkka Niemelä. *A fixpoint definition of dynamic constraint satisfaction*. In Proc. CP'99, 1999. Principles and practices of constraint programming.

[Veron 01] Matthieu Veron. *Modélisation et résolution du problème de configuration industrielle : utilisation des techniques de satisfaction de contraintes*. PhD thesis, Institut National Polytechnique de Toulouse, ENI Tarbes, November 2001.

[Vu *et al.* 02] Xuan-Ha Vu, Djamila Sam-Haroud & Marius-Calin Silaghi. *Numerical constraint satisfaction problems with non-isolated solutions*, 2002.

Références Internet

[Barták] Roman Barták. *Constraint propagation*. http://www.enstimac.fr/~hadjhamo/CSP/CSP-Discrets/propagation.html.

[Bensana] Éric Bensana. *Approche par contraintes pour les problèmes de configuration*. http://www.laas.fr/FERIA/SD/feria.pdf.

[Codognet, Diaz] Philippe Codognet & Daniel Diaz. *An efficient library for solving* CSP *with local search*. http://ww.ailab.sztaki.hu/Codognet_Diaz.pdf.

[Fargier, Henocque] Hélène Fargier & Laurent Henocque. *Configuration à base de contraintes*. sis.univ-tln.fr/gdri3/fichiers/assises2002/slides/07-ConfigurationABaseDeContraintes.ppt.

[ILO] *Site Internet de ILOG*. http://www.ilog.com.

[Jussien] Narendra Jussien. *Publications de Narendra* JUSSIEN. http://njussien.e-constraints.net/publications.html.