

Implémenter des contraintes globales expliquées

Guillaume Rochart^{1,2}

Narendra Jussien²

¹ École des Mines de Nantes – LINA CNRS FRE 2729
4 rue Alfred Kastler – BP 20722 – F-44307 Nantes Cedex 3, France

² e-lab – Bouygues SA
1 av Eugène Freyssinet – F-78280 Guyancourt, France
grochart@bouygues.com jussien@emn.fr

Résumé

Modifier le comportement d'un solveur de contraintes pour lui ajouter de nouvelles capacités est souvent une tâche complexe : l'optimisation extrême, fruit d'une expérience de développement de plusieurs années, du cœur de propagation et des algorithmes de filtrage rend cette opération particulièrement délicate si on souhaite maintenir le niveau de qualité atteint par l'outil. Dès lors, cette difficulté constitue bien souvent un frein pour mettre en place au sein du solveur de nouvelles techniques ou outils (explications, trace précise, etc.). Nous montrons dans cet article les problèmes posés par l'ajout d'explications dans des contraintes globales et proposons deux nouvelles méthodes pour ajouter des contraintes globales expliquées dans un solveur : l'une basée sur l'utilisation d'un cadre générique de description des contraintes globale, l'autre utilisant les possibilité de la programmation par aspects.

Abstract

Adding new features to a constraint solver may be a tedious and difficult task. Indeed, propagation engines and filtering algorithms have been optimized for a long time, by numerous developers. To modify them without decreasing the quality of the software, one has to check that the new feature will not interfere with previous works. This is why developing new features like on-the-fly explanations or new trace formats can be hard and long to add in a commercial constraint solver, even it is well known that these features can be very useful for the user. In this paper, we show what can be the issues when one wants to add explanations for global constraints and then propose two new ways to achieve this goal.

1 Introduction

Pouvoir résoudre un problème combinatoire est d'une grande utilité pour améliorer de nombreux pro-

cessus industriels et en optimiser les revenus, la qualité ou l'ergonomie : gestion du personnel, optimisation d'un parc de machines, emploi du temps (des universités aux formations d'employés), etc. Cependant, dans les outils actuels, lorsqu'il n'est pas possible de trouver une solution (car toutes les contraintes dures – qui ne peuvent être remises en cause – forment un problème sur-contraint) il est rarement possible d'en connaître la cause. De même, il est pratiquement impossible dans les solveurs de simuler après résolution le retrait d'une ou plusieurs contraintes pour modifier le problème afin d'améliorer les solutions trouvées.

Cette problématique n'est d'ailleurs pas propre à la programmation par contraintes ni même aux outils de résolutions de problèmes combinatoires, mais constitue un domaine de recherche en intelligence artificielle, cherchant à expliciter le résultat retourné par un logiciel. L'*abduction*, ou la recherche d'*explication*, dans le cadre de la programmation par contraintes, consiste en particulier à fournir des outils ou à instrumenter un algorithme pour permettre la génération, le maintien et le stockage d'ensembles d'informations permettant de justifier logiquement un état courant du système (comme une contradiction, une valeur de la fonction objectif, etc.). Mais générer de telles explications, les exploiter et les rendre accessibles à l'utilisateur final peut nécessiter des modifications profondes de l'outil.

Par exemple, dans le cas de la programmation par contraintes – mise à part la technique de QuickXPlain [6] qui est non intrusive mais peut coûter particulièrement cher – la plupart des techniques proposées nécessite de modifier le code des contraintes, voire de modifier l'algorithme de résolution [7]. Or, dans le cas de la programmation par contraintes, modifier le cœur d'un solveur de contraintes peut se révéler très com-

pliqué voire inconcevable étant donnés les risques que cela représente pour l'entreprise vendant la solution.

Nous proposons donc dans cet article plusieurs méthodes pour implémenter des contraintes globales générant des explications¹. Ainsi, après une brève introduction sur la programmation par contraintes et les explications, nous rappelons tout d'abord des travaux précédents [4] qui ont proposé une méthodologie pour modifier de manière *ad hoc* les contraintes (ce qui permet d'obtenir des implémentations efficaces mais potentiellement coûteuses à mettre en place). Nous soulignerons ainsi quels sont les problèmes liés à la génération d'explication pour les contraintes globales. Puis nous proposons deux nouvelles solutions consistant à utiliser d'une part, des cadres génériques de développement de contraintes globales et d'autre part, la programmation par aspects pour rendre les modifications indépendantes du code originel (et ainsi ne pas prendre le risque de mettre en péril la stabilité du solveur) permettant de réduire notablement le coût de mise en place des explications dans les solveurs. Nous concluons en exposant les avantages et inconvénients de chaque méthode.

2 Contexte

2.1 La programmation par contraintes

La programmation par contraintes est une technique maintenant bien connue de résolution de problèmes combinatoires modélisés sous la forme de problèmes de satisfaction de contraintes (CSP). Un CSP se caractérise par trois ensembles principaux :

- les variables v_1, v_2, \dots, v_n du problème ;
- leurs domaines tels que chaque variable v_i doit être instanciée à une valeur de son domaine d_i ;
- des contraintes reliant les différentes variables.

Le but est alors de trouver une affectation pour chaque variable telle que toutes les contraintes sont vérifiées simultanément.

Pour cela, la programmation par contraintes s'appuient sur deux phases qui coopèrent :

- une phase de **branchement** qui consiste à se focaliser sur une portion de l'espace de recherche en prenant une décision (généralement, en affectant une valeur à une variable) pour essayer de trouver une solution dans cette portion (si cela échoue un mécanisme classique de retour-arrière est mis en place) ;
- une phase de **propagation** qui consiste à réduire *a priori* l'espace de recherche en appelant les

contraintes concernées par les modifications apportées sur les variables pour que ces dernières puissent déduire des valeurs à retirer d'autres domaines (grâce à un algorithme de **filtrage**)

Si l'on souhaite ajouter une fonctionnalité dans un solveur de contraintes, il est souvent nécessaire de modifier ces deux blocs qui représentent tous les deux du code qui a été optimisé. Il est donc délicat, comme nous le verrons, d'ajouter de telles fonctionnalités sur des plateformes commerciales qui demandent une forte qualité de service ou des plateformes d'une taille telle qu'il est impossible d'ajouter du code dans toutes les contraintes, tous les branchements, etc. C'est pourquoi nous proposons ici de nouvelles méthodes pour instrumenter les solveurs et les contraintes globales avec de nouvelles fonctionnalités comme les explications.

2.2 Les explications et les algorithmes les exploitant

Une explication est, dans le cas général, une justification logique d'un état (problème inconsistant, domaine courant d'une variable, valeur de la borne durant une optimisation, etc.). Dans le cadre de la programmation par contraintes, il est nécessaire de déterminer quelles sont les hypothèses que l'on souhaite étudier pour pouvoir justifier un état. Parmi les hypothèses que l'on peut considérer, on peut citer par exemple : les domaines des variables, les choix pris par un algorithme de branchement, les contraintes (que l'utilisateur a posées), etc.

Selon ce que l'on souhaite faire des explications que l'on calcule, différentes parties de ces ensembles seront considérées comme des hypothèses. Par exemple si l'on souhaite juste conserver de l'information sur la recherche pour faire une recherche *intelligente*, seuls les choix du branchement nous intéressent. Par contre si l'on souhaite transmettre une information à l'utilisateur, la plupart du temps, les hypothèses sont les contraintes (c'est-à-dire quelles sont les contraintes qui impliquent qu'un problème soit inconsistant?). Comme les deux applications semblent intéressantes et peuvent être générées de manière équivalentes, nous définissons les explications à l'aide de ces deux types d'hypothèses que sont les choix pris par le branchement et les contraintes du modèle.

Définition 1 (Explication) *L'explication de l'état X (notée $\text{expl}(X)$) est un sous-ensemble C' des contraintes du modèle $C' \subseteq C$ et un sous-ensemble des décisions prises jusqu'à présent par l'algorithme de branchement $(d_{i_1}, \dots, d_{i_m})$ tels que la conjonction de ces deux ensembles conduit logiquement à l'état X étant donnée la théorie de la PPC (respect des domaines, des contraintes, etc.).*

¹Nous nous focalisons en effet sur les contraintes globales dans cet article : d'une part, les contraintes *simples* sont plus aisément instrumentables [11] et d'autre part, leur importance dans le domaine n'est plus à démontrer.

Pour évaluer la qualité d'une explication, on mesure sa *précision*. On dira ainsi qu'une explication est *plus précise* qu'une autre si : elle est la plus petite au sens de l'inclusion, ou bien si elle a une plus petite cardinalité, ou encore si elle est de poids agrégé le plus faible (si l'on a affecté des poids aux contraintes), etc. La définition souvent utilisée en programmation par contrainte est la définition de la précision au sens de l'inclusion. Les résultats de ce papier ne garantissent pas l'optimalité de la précision des explications, mais essaient de fournir des algorithmes déterminant naturellement les explications les plus précises possible en ce sens.

2.3 La difficulté des explications pour les contraintes globales

L'état d'un problème de satisfaction de contraintes se décrit comme nous l'avons vu à l'aide des domaines, des contraintes du problème, et si l'on est en cours de résolution des choix faits par l'algorithme de recherche. Il est donc possible de justifier un état (contradiction, valeur d'une borne, valeur impossible), en fonction de ces données. Il est donc possible de justifier tout état du problème.

Ainsi, dans le cas de contraintes *simples* (portant sur un nombre de variables limité et avec des raisonnements simples), il est aisé d'expliquer chacun des filtrages déduits par l'algorithme de filtrage [11]. Par exemple dans le cas d'une contrainte d'égalité $x = y$, si la valeur 5 est retirée du domaine de y , alors cette même valeur doit être retirée du domaine x . La valeur est alors retirée de x pour la même raison (explication) que le retrait du domaine y et la contrainte d'égalité. On voit ainsi qu'il est possible de construire incrémentalement des explications pour chaque valeur retirée.

Cependant on cache déjà ici la difficulté liée à cette construction incrémentale au cours de la recherche qui nécessite de modifier substantiellement l'algorithme de branchement (cf. [8]). De plus, lorsque l'on considère une contrainte globale, un tel raisonnement devient particulièrement compliqué. En effet, sans accéder aux structures internes de la contrainte, il est difficile de savoir pourquoi l'algorithme de filtrage a pris cette décision². Une solution simple serait de dire que le retrait est justifié par l'état courant de tous les domaines des variables de la contrainte et de cette dernière contrainte. Cependant, ceci générerait des expli-

²Par exemple, dans le cas de la contrainte `all_different` : si l'on considère quatre variables $a = \llbracket 1, 4 \rrbracket$, $b = \llbracket 1, 4 \rrbracket$, $c = \llbracket 3, 4 \rrbracket$ et $d = \llbracket 3, 4 \rrbracket$ qui doivent être toutes différentes, le filtrage proposé par [15] déduirait que a et b sont différentes toutes deux de 3 et 4. Sur un exemple aussi simple, on voit déjà qu'il n'est pas facile de trouver *a priori* un algorithme général permettant de déduire une explication précise de ces retraits de valeurs.

cations bien peu précises et qui n'apporteraient d'informations utiles ni à l'utilisateur (que ce soit pour l'utilisateur final ou le développeur) ni à l'algorithme de recherche si l'on souhaite améliorer la recherche en fournissant des explications sur les échecs.

Il est donc nécessaire de fournir des outils pour générer des explications valides (qui justifient bien logiquement ce que l'on souhaite) aussi précises que possibles, dans un temps raisonnable (si possible une complexité du même ordre de grandeur que le filtrage en lui-même) et pour chaque valeur que l'algorithme de filtrage peut retirer des domaines.

Nous introduisons ici trois techniques différentes tant au niveau de leur efficacité et de leur portée sur les différents type de problèmes à résoudre qu'au niveau de leur coût en développement pour les mettre en place. Nous commençons par la solution la plus évidente consistant à reprendre les algorithmes de filtrage et à justifier chacun de leurs calculs [4], puis présentons deux nouvelles méthodes : l'une qui exploite les cadres génériques de contraintes globales pour générer des explications pour un nombre important de contraintes et adapter ces contraintes à des problématiques dynamiques, et l'autre basée sur la programmation par aspects, une technique récente issue du génie logiciel.

3 Implémentation intrusive

La solution la plus intuitive, mais la plus coûteuse en temps de développement, consiste à modifier d'une part le cœur du solveur de contraintes pour prendre en compte et maintenir les explications au cours de la recherche, et d'autre part les algorithmes de filtrage pour générer des explications à chaque fois qu'une valeur est filtrée [4]. Nous allons tout d'abord mettre en évidence quelles sont les difficultés à prendre en compte lorsqu'on souhaite générer et maintenir des explications avec des contraintes globales. Nous verrons comment surmonter ces difficultés avec les deux nouvelles solutions que nous proposons dans la suite de l'article.

3.1 Expliquer chaque règle du filtrage

La première difficulté consiste à générer les explications elles-mêmes. En effet, on pourrait souhaiter, comme dans le cas des contraintes basiques et/ou binaires (arithmétiques par exemple), générer les explications une fois que la valeur à filtrer a été trouvée. En effet, il peut sembler aisé à première vue de justifier le filtrage d'une valeur étant donnée la sémantique de la contrainte. C'est malheureusement rarement le cas avec les contraintes globales puisque les algorithmes de filtrage s'appuient bien souvent sur des règles subtiles. Expliquer sans exploiter les calculs de l'algorithme de

filtrage risque donc d'impliquer soit des explications peu précises, voire d'oublier des cas ou de générer des explications qui n'en sont pas (ce qui risque de remettre en cause la complétude de la recherche).

De plus, même si l'on génère les explications sans utiliser l'algorithme de filtrage et ses structures, il risque d'être nécessaire d'utiliser des algorithmes très coûteux. Or le but est d'essayer, autant que possible, de générer des explications avec des algorithmes de complexité équivalente à celles des algorithmes de filtrage.

C'est pourquoi il est plus prudent de reprendre les algorithmes de filtrage, comme `all_different`, `gcc`, `stretch`, `flow`, etc. pour instrumenter ces algorithmes à l'aide d'informations nécessaires à la génération d'explications. Par exemple, dans le cas de la première version de la contrainte `stretch` [12], l'algorithme de filtrage s'appuie sur des calculs de bornes des blocs de valeurs identiques et successives, et sur quelques règles exploitant ces bornes. Pour justifier une valeur filtrée, il est donc nécessaire, d'une part de générer les explications justifiant les étendues minimale et maximale du bloc courant et de justifier le domaine des certaines variables justifiant la règle de filtrage [17] (notamment le domaine des variables voisines au bloc d'étude courant).

De même, dans le cas du `all_different` ou du `gcc`, l'algorithme proposé par [15] propose de décomposer un graphe biparti reliant valeur et variable en composantes fortement connexes. On peut alors prouver qu'une valeur qui ne se trouve pas dans la même composante fortement connexe que l'une des variables ne peut pas être affectée à cette variable. Pour pouvoir filtrer cette valeur du domaine, il est donc nécessaire d'expliquer dans ce cas pourquoi la variable et la valeur n'appartiennent pas à la même composante fortement connexe, ce qui implique bien d'instrumenter l'algorithme pour construire au fur et à mesure de la recherche des composantes cette explication [4].

3.2 Maintien de la structure dans un cadre dynamique

Une des applications des explications est de résoudre des problèmes *dynamiques*, c'est-à-dire, dont l'ensemble des contraintes peut varier. C'est d'ailleurs sur cette hypothèse que tient la proposition d'utiliser l'algorithme `mac-dbt` [8], qui remplace le retour-arrière classique par le retrait dynamique d'une des contraintes responsables de l'échec actuel.

Cet aspect dynamique pose des problèmes dans la gestion des contraintes. En effet, un problème dynamique demande de retirer une contrainte sans recommencer les calculs; il est alors nécessaire d'assurer à tout moment que l'on se trouve dans le même état que

celui dans lequel on devrait être si l'on avait tout recalculé statiquement avec le nouveau problème, que ce soit au niveau des domaines des variables (les valeurs inconsistantes doivent être effectivement retirées, mais celles qui sont consistantes doivent être remises dans le domaine si besoin), ou au niveau de la structure de données des contraintes.

Pour maintenir un domaine cohérent pour les variables, toutes les valeurs qui ont été filtrées à cause d'une contrainte que l'on vient de retirer sont remises dans leurs domaines respectifs. Ainsi, on est sûr de ne pas retirer de solutions en ignorant une valeur d'un domaine injustement. Toutefois, il est alors nécessaire de vérifier si ces nouvelles valeurs sont bien consistantes à l'aide d'une phase dite de repropagation [2].

Le deuxième problème provient du maintien des structures de données. En effet, les structures de données sont souvent prévues pour fonctionner dans un algorithme de type *retour-arrière*, c'est-à-dire qu'une pile stocke les différentes valeurs attribuées à la structure dans l'arbre de recherche. Mais si l'on résout dynamiquement un tel problème, ce mécanisme n'est plus valable; il est alors primordial de pouvoir réparer la structure de données associée à une contrainte, par exemple, en stockant la dernière structure consistante, et en l'adaptant au nouvel état de la recherche [4].

Tous ces problèmes représentent un frein pour l'utilisation des explications générées à la volée dans les solveurs de contraintes, bien qu'il s'agisse souvent d'une solution intéressante car elle ne nécessite pas de nouveaux calculs, mais fournit une explication pour chaque problème dont il n'est pas possible de trouver une solution. Nous présentons deux nouvelles solutions qui permettent de réduire ces inconvénients pour faciliter l'utilisation des explications dans des solveurs utilisant non seulement des contraintes basiques, mais aussi des contraintes globales engendrant ces nouvelles difficultés.

4 Utilisation de cadres génériques de contraintes globales

La première nouvelle solution que nous proposons consiste à utiliser des travaux qui proposent de fournir un cadre générique pour dériver de nombreuses contraintes à partir d'une description simple de celle-ci soit sous la forme d'un automate [1, 13], soit sous la forme de propriétés de graphes [1]. Ainsi, si l'on accepte le surcoût lié à la perte de la complétude des algorithmes de filtrage ou à des algorithmes moins efficaces, il devient possible de générer au même titre que le filtrage la génération d'explications associées. Nous commençons par présenter succinctement ces cadres puis montrons comment il est possible de générer des

explications automatiquement si l'on fournit un algorithme propageant ces nouvelles contraintes.

4.1 Les contraintes globales comme des automates ou propriétés de graphes

L'univers des contraintes globales semblant infini, plusieurs cadres ont été proposés pour essayer de les décrire, de les comparer, voire de dériver automatiquement l'algorithme de filtrage d'une contrainte depuis sa description. Parmi ces travaux on peut citer par exemple deux types de descriptions :

- **une description à base d'automates** qui consiste à simuler l'algorithme de filtrage en le modélisant sous la forme d'un automate [13, 1]; ainsi, il est possible par exemple, de décrire une contrainte en fournissant un automate vérifiant si une solution satisfait ou non la contrainte, et le cadre permet d'en dériver automatiquement un algorithme de filtrage; pour cela, le cadre utilise un nombre bien défini de contraintes élémentaires pour décrire sémantiquement le filtrage que l'on souhaite obtenir; si l'on souhaite ajouter une fonctionnalité comme les explications dans les contraintes globales, il suffit alors de l'ajouter dans ce petit noyau de contraintes élémentaires.
- **une description à base de propriétés de graphes** qui consiste à décrire à l'aide de propriétés de graphes (où schématiquement les nœuds sont des variables et les arcs des contraintes qui peuvent être satisfaites ou non) la contrainte que l'on souhaite maintenir au cours de la recherche [1]. Ainsi, pour modéliser l'exemple bien connu du `all_different`, il suffit de considérer chaque variable comme un nœud, de générer des arcs potentiels entre chaque paire de variables différentes avec des contraintes élémentaires d'égalité, et de forcer que la taille maximale de toutes les composantes connexes soit de 1. Dans ce cas, l'algorithme propageant la propriété qui doit être vérifiée déduira directement que chaque arc doit être absent du graphe (sinon les composantes seraient de taille supérieure à 1). Cela se traduit par le fait que pour chacun de ces arcs retirés il faut propager l'inverse d'une égalité, c'est-à-dire une différence, ce qui garantit bien que toutes les variables doivent être différentes. Si au contraire, un arc était devenu présent de manière sûre, la contrainte élémentaire associée aurait été activée (au lieu de son opposée).

Pour résumer, ces cadres permettent de décrire sémantiquement le filtrage que doit effectuer une contrainte, voire dans certains cas de dériver automatiquement l'algorithme de filtrage associé [1, 5]. Étant donné ces résultats, il semble donc très intéressant

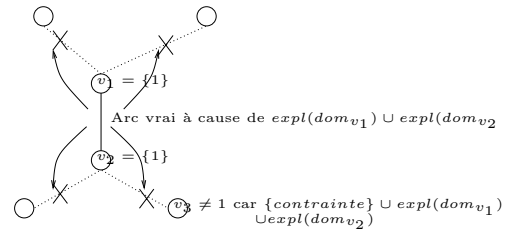


FIG. 1 – Exemple de génération d'explications : supposons que l'on veuille que seules deux variables puissent avoir des valeurs égales par exemple à cause d'une contrainte de `global_cardinality` [16] (taille maximale des composantes de 2, avec des contraintes élémentaires d'égalité sur les arcs), et que nous étions dans l'état décrit sur le dessin (seules les variables reliées avaient des domaines dont l'intersection était non vide), alors si v_1 et v_2 sont instanciées à 1, l'arc potentiel les reliant devient présent de manière sûre (car la contrainte d'égalité est forcément vraie), on peut donc en déduire que les autres arcs sortant de v_1 et v_2 doivent être absent pour la même raison, à laquelle on ajoute la contrainte en cours qui maintient la propriété de graphe. On peut alors filtrer des valeurs depuis les variables voisines avec cette explications.

d'en profiter pour générer automatiquement un grand nombre de contraintes globales avec les fonctionnalités voulues à moindre coût (il suffit d'instrumenter les contraintes élémentaires).

4.2 Dériver de telles contraintes pour créer des contraintes expliquées

Si l'on reprend le cas du `all_different` et que l'on souhaite générer des explications pour chaque retrait de valeur, il faut justifier pourquoi la contrainte a déduit ce retrait. Pour cela, dans le cas de contraintes modélisées à l'aide de propriétés de graphes, il est donc nécessaires d'expliquer :

- l'état des arcs (puisque à chaque arc est associée une contrainte dont la propagation dépend de l'état de cet arc – **absent** ou **présent**),
- la propagation de la contrainte élémentaire associée à l'arc.

Dans le cas de la contrainte élémentaire (une contrainte d'égalité, différence, infériorité, etc.), les outils permettent déjà de fournir les explications nécessaires [7]. Il suffit donc d'expliquer la propagation qui s'effectue sur les propriétés de graphes pour pouvoir expliquer l'ensemble. Par exemple, dans le cas du `all_different`, la contrainte en elle-même (via la propriété de graphe associée) permet directement de déduire que chaque arc doit être à **absent**. Ainsi, il est possible d'expliquer l'état de chaque arc, juste avec la contrainte en elle-même. Donc le filtrage qui sera effectué par chaque contrainte élémentaire (des différences ici), utilisera l'explication du domaine de la deuxième variable et l'explication de l'arc, c'est-à-dire la contrainte uniquement.

Dans des cas plus compliqués (voir la figure 1), l'état d'un arc peut être modifié car le solveur déduit qu'une contrainte élémentaire est forcément vérifiée. Ainsi, dans l'exemple de la figure 1, si on avait comme propriété de graphe que les composantes était au plus de

taille 2, il serait alors possible de déduire que tous les autres arcs reliant ces deux variables à d'autres variables doivent être absent du graphe final. Pour expliquer cet état, il suffit de justifier l'état de l'arc qui est forcément présent (ce qui est comme nous l'avons vu possible puisqu'il s'agit de contraintes élémentaires dont on sait expliquer le comportement), et d'ajouter à l'explication la contrainte en cours qui a permis de modifier l'état de ces arcs (passage d'un arc potentiel à un arc absent du graphe).

Outre la factorisation du code pour une multitude de contraintes, l'avantage d'utiliser ce cadre basé sur les propriétés de graphes pour générer des contraintes expliquées provient de la précision des explications que l'on peut générer. En effet, on voit bien dans le dernier exemple que seules les explications des domaines de variables qui ont vraiment un rapport avec le filtrage ont été prises en compte.

5 Utilisation des aspects

Le principal défaut de la méthode intrusive était de devoir reprendre tous les algorithmes de filtrage un à un, en modifiant le code directement : il est nécessaire d'avoir accès au code (ce qui n'est pas toujours le cas), le solveur doit être modifié pour prendre en compte les explications, et chaque contrainte doit être dupliquée, ce qui engendre bien entendu des problèmes de versions, oublis de corrections d'erreurs, etc. L'avantage de la solution basée sur les cadres génériques de description de contraintes est qu'elle permet de factoriser la difficulté à développer de telles contraintes. Ainsi l'investissement de départ est plus lourd mais peut être amorti étant donné le nombre de contraintes que l'on peut générer à l'aide de ces cadres.

Cependant, le principal défaut d'un cadre générique est souvent de perdre un peu d'efficacité (même si ce n'est pas toujours le cas comme l'ont montré les travaux de [1]). Nous proposons donc ici de trouver un intermédiaire entre ces deux solutions. En effet, moyennant la supposition que l'on ne souhaite pas résoudre de problèmes dynamiques, on peut remarquer que générer des explications ne consiste qu'à générer de l'information lors du déroulement de l'algorithme de filtrage à différents points du code. C'est typiquement ce que permet d'effectuer la programmation par aspects qui consiste à ajouter du code en des points précis du code original (ou du *bytecode* si l'on ne possède pas les sources mais l'API précise des contraintes et de l'algorithme de recherche) : il n'est donc plus nécessaire de modifier le code lui-même (ou un clone) et permet donc de rendre l'ajout des explications complètement modulaire (au même titre qu'un aspect de trace, les explications peuvent être débranchées pour retrouver

l'efficacité d'origine de l'outil).

Nous commençons par proposer une description brève de la programmation par aspects, ses notions de base et une implémentation pour Java, AspectJ. Nous en profitons pour présenter une application classique de cette technique que l'on peut appliquer pour la PPC, la génération de traces. Nous montrons ensuite comment l'appliquer à la génération d'explications pour la PPC, puis nous montrerons les résultats obtenus avec une première implémentation.

5.1 La programmation par aspects

La programmation par aspects [9] est une technique de génie logiciel qui permet de factoriser du code que l'on souhaite déployer en de nombreux endroits. Les deux principaux avantages de cette technique sont, pour la programmation par contraintes, d'une part de simplifier l'étape de développement en proposant des outils simples pour éviter la duplication de code, et d'autre part de fournir du contenu modulable. En effet les différents aspects peuvent être ou non ajoutés au programme original, ce qui permet de brancher ou débrancher très facilement de nouvelles fonctionnalités.

La programmation par aspects (AOP) s'appuie autour de trois notions particulières :

- les *join points* (points de jonction) sont des points précis dans le déroulement d'un programme où l'on souhaite ajouter du comportement,
- les *pointcuts* (coupes) représentent des ensembles de *join points* décrits à l'aide d'un langage dédié,
- les *advice* (code *advice*) sont les fonctionnalités que l'on souhaite ajouter.

Ainsi, pour utiliser l'AOP, il suffit de définir des *point cuts* pour décrire où l'on souhaite ajouter du code, et les *advice* qui doivent contenir ce code factorisé. De plus, il peut être intéressant d'ajouter de l'information aux objets traités, sans devoir passer par le mécanisme d'héritage ou tout autre mécanisme proposé par le langage. Pour cela l'AOP permet de faire des modifications *inter type*, c'est-à-dire modifier la structure même d'un objet (ajouter un champ, une méthode, etc.). À l'aide de ces notions, l'AOP fournit des outils pour *tisser* ces nouvelles fonctionnalités avec le logiciel cible (sous forme de *bytecode* ou de code compilé), c'est-à-dire que le code compilé est modifié pour prendre en compte ces ajouts.

L'application la plus évidente de l'AOP reste la génération de trace. C'est d'ailleurs une des premières applications qu'en avaient faite les auteurs pour implémenter la trace GENTRA4CP du projet OADYMP-PAC[3] pour Choco [10]. En effet, générer de telles traces demande d'accéder à des informations qui ne

sont accessibles qu’au cœur du solveur (gestion des événements, suivi de la propagation, etc.). Cependant, sa génération reste coûteuse (tout comme pour les explications, comme nous le verrons plus tard). Il est donc important, d’une part d’avoir à modifier le code dans tout le solveur (chaque contrainte, chaque type de branchement, chaque domaine de variables, etc.) et d’autre part de pouvoir débrancher facilement cette trace, pour ne pas perdre d’efficacité lorsqu’elle n’est pas nécessaire.

Comme le montre la section suivante, ces problématiques sont très proches de celles de la génération d’explications, ce qui explique que l’AOP soit aussi intéressante dans ce cas. Nous illustrons nos propos ici avec `AspectJ`, car il s’agit d’un des outils les plus connus pour l’AOP. Il ne s’agit cependant pas de l’outil le plus rapide ni le plus adapté à notre problème, mais semble suffisant pour illustrer nos propos. De plus, les exemples seront appliqués à `Choco` dont on a désactivé la gestion des explications, pour réimplémenter à partir de rien un système avec explications³.

5.2 Application aux explications

La programmation par aspects peut être utilisée de deux manières : soit il s’agit d’un outil de génie logiciel classique que l’on exploite dès la conception d’un outil principalement pour rendre l’outil modulaire; soit il s’agit d’un outil permettant de faire évoluer/modifier un outil existant (les deux objectifs n’étant bien entendu pas à opposer). Dans un précédent article, il a été montré comment les explications pouvaient être vues comme un aspect dans un solveur en partant d’un squelette de solveur. Ceci exposait bien le premier cas dont nous venons de parler, mais ne montrait pas qu’il était possible d’appliquer cela d’une part à des solveurs existants (ce qui est nécessaire pour pouvoir *industrialiser* l’utilisation d’explication dans des solveurs commerciaux) et d’autre part dans des configurations plus complexes mettant en œuvre des contraintes globales.

Pour utiliser les explications avec la PPC et avec les contraintes globales, il est nécessaire d’ajouter quelques informations et comportements dans le solveur pour générer et maintenir les explications au long de la recherche de sorte à exploiter un minimum l’information à l’aide de l’algorithme `mac-cbj` [14]. Les principales modifications à apporter sont les suivantes :

- créer une classe d’objets contenant une explication (`CustomExplanation` dans la suite des exemples);

³On notera cependant qu’historiquement `Choco` n’a pas été développé pour générer des explications, mais qu’il s’agit simplement d’une intégration de `Palm` [7] pour rendre l’accès aux explications plus simple

- pour chaque choix du branchement, la modification sur le domaine concerné doit être justifiée avec une explication contenant une référence vers ce choix;
- à chaque retour arrière ajouter une explication au retrait du choix sur la variable que l’on vient d’annuler, pour pouvoir construire incrémentalement l’explication de l’absence de solution d’un problème;
- modifier les domaines et variables pour prendre en compte les explications.

Pour rendre l’exemple lisible, nous illustrons ici le cas le plus simple, à savoir ajouter une explication au niveau des choix du branchement. Pour cela, on crée un *pointcut* décrivant un tel choix, le but étant de réussir à isoler l’endroit où se situe la prise de décision lorsqu’une variable est instanciée par l’algorithme de branchement. On peut obtenir une telle coupe de la manière suivante :

```
pointcut assignVal(IntVar var) :
    call(* IntVar+.setVal(*)
    && target(var)
    && cflow (execution
    (* AssignVar+.goDownBranch(..)));
```

Ce *pointcut* permet de décrire un point d’exécution du programme ayant les caractéristiques suivantes :

- la méthode `setVal` est appelée sur une variable : le `call` représente un appel à une méthode auquel on fournit une description de ce que peut être cette méthode (ici une méthode qui renvoie n’importe quoi `*`, sur un objet de type `IntVar` ou une classe qui en hérite `+`, dont la méthode se nomme `setVal` et enfin qui prend n’importe quels arguments);
- on unifie alors la variable en paramètre du *pointcut* vers la variable modifiée (la cible `target` de la méthode définie ci-dessus);
- cet appel doit se faire depuis l’exécution de la méthode `goDownBranch` du branchement utilisé; c’est ce que permet de vérifier l’instruction `cflow` : la pile d’appels des méthodes est utilisée pour vérifier que l’on exécute bien une méthode nommée `goDownBranch` de la classe `AssignVar` ou une classe dérivée de cette dernière.

Maintenant que l’on a décrit l’endroit où il convient d’ajouter de l’information, il faut maintenant dire précisément ce qui doit y être ajouté, et si cela doit avoir lieu, *avant*, *après*, *à la place*... dudit endroit. Pour cela, on ajoute alors un *advice* :

```
before(IntVar var) : assignVal(var) {
    CustomExplanation exp =
        new CustomExplanation(
```

```

    var.getProblem().getWorldIndex() - 1,
    var.getProblem());
var.getDomain().explanation = exp;
}

```

Comme on le voit, l'*advice* doit se dérouler *avant* le *pointcut* (*before*), et doit construire une nouvelle explication comprenant une information à propos du *monde* actuel, c'est-à-dire la taille de la pile de la recherche. Cette explication est alors ajoutée dans les informations contenues dans les variables. Par souci de place, nous ne détaillerons pas toutes les modifications à apporter au solveur, mais il est intéressant de remarquer qu'au final, moins de 200 lignes sont ainsi nécessaires pour instrumenter Choco avec des fonctionnalités de base sur les explications.

Nous nous intéressons maintenant aux contraintes, le cœur du sujet. Comme nous l'avons vu dans la première solution pour implémenter des contraintes globales, ajouter des explications demandent deux types de modifications : générer les explications au fur et à mesure du filtrage, et maintenir les structures de données dans le cas du dynamique. Seul le premier problème nous intéresse ici puisque nous avons fait l'hypothèse que les problèmes traités ne seront pas dynamiques (ou bien seront traités comme plusieurs résolutions statiques consécutives).

Tout d'abord commençons par une contrainte basique et très simple : la différence entre deux variables. On voit alors très bien, que dès lors qu'aucune structure de données n'est utilisée, il est très simple d'instrumenter une contrainte avec des explications. En effet l'algorithme trivial de cette contrainte est de vérifier, à chaque fois que cela peut arriver, si l'une des variables est instanciée, et dans ce cas retirer la valeur de cette variable du domaine de l'autre variable. Donc si l'on suppose que l'on possède un *pointcut* `neqXYCAnyFiltering` qui référence l'ensemble des points où un tel filtrage peut avoir lieu, on peut lui associer l'*advice* suivant :

```

before(NotEqualXYC cst, IntVar var) :
  neqXYCAnyFiltering(cst, var) {
    IntVar otherVar = cst.getIntVar(0);
    if (otherVar == var)
      otherVar = cst.getIntVar(1);
    CustomExplanation expl = new
      CustomExplanation(var.getProblem());
    cst.self_explain(expl);
    otherVar.self_explain(
      CustomExplanation.DOM, expl);
    var.getDomain().explanation = expl;
  }

```

Ainsi, on voit que la génération de l'explication se déroule en trois temps :

1. on cherche l'autre variable de la contrainte `otherVar` (car c'est son domaine qui a déclenché ce filtrage);
2. on génère une explication justifiant le domaine de cette seconde variable et on lui ajoute la contrainte de différence elle-même `cst.self_explain(expl)`;
3. on fournit l'explication au domaine pour que ce dernier puisse la prendre en compte lors de l'exécution du filtrage (juste après cet *advice* par définition de celui-ci).

Dans le cas d'une contrainte globale, exactement le même mécanisme devra être mis en place. La différence vient surtout du fait que les algorithmes sont plus compliqués, exploitent des structures de données, et qu'il est donc nécessaire d'en tenir compte pour générer les explications. Dans un souci de clarté, nous considérons ici une contrainte simple d'occurrence. Le but est d'assurer qu'à tout moment, le nombre d'occurrences d'une valeur fixe donnée λ dans un ensemble de variables v_1, v_2, \dots, v_n est compris entre la borne inférieure et supérieure d'une variable d'occurrence *occ* (il s'agit en fait d'un cas très simple d'application de la contrainte de cardinalité bien connue). Pour filtrer efficacement, cette contrainte maintient incrémentalement le nombre de variable pouvant prendre la valeur fixée λ et le nombre de variables instanciées à cette valeur. Dès lors, les deux règles suivantes sont appliqués :

- si le nombre de variables pouvant être instanciées à λ est égal à la borne inférieure de *occ*, alors toutes ces variables doivent être instanciées;
- si le nombre de variables devant être instanciées à λ est égal à la borne supérieure de *occ*, alors toutes les autres variables ne doivent pas être instanciées à cette valeur et donc λ est retiré de leurs domaines.

Pour expliquer ce comportement, il faut donc :

- à chaque fois que le nombre de variables instanciées ou pouvant être instanciées à λ est modifié, stocker l'explication ou tout au moins un moyen de retrouver l'explication de cette nouvelle valeur (maintenir la liste des variables qui permettent de justifier ce nombre suffit),
- à chaque fois qu'une règle de filtrage est appelée, il faut que l'*advice* utilise ces explications stockées, y ajoute la contrainte en cours, et informe le domaine filtré que cette explication justifie le retrait de la ou des valeurs en question.

Prendre en compte des contraintes plus compliquées comme le `all_different` ou le `flow` n'apporte pas plus de difficulté. La seule limitation qu'il peut y avoir provient de l'accès à certaines données qui ne sont pas publiques. Dans ce cas, une solution revient à hériter

Problème	Sans expl.	Expl. intrus.	AOP
Diff.(14)	0.19s	0.62s	0.78s
Diff.(16)	0.77s	1.4s	1.8s
Diff.(18)	6.1s	12.1s	15s
Occur.(14)	12.1s	8.1s	9.6s

TAB. 1 – Résultats des premières expérimentations (avec entre parenthèse la taille n du problème) sur un Celeron 900 MHz et les options par défaut de compilation Java

de la contrainte, ajouter des accesseurs aux données dont on a besoin dans l’aspect et de modifier le comportement du constructeur pour renvoyer une instance de la nouvelle classe au lieu de l’ancienne.

5.3 Évaluation

Pour apporter une première évaluation de l’utilisation de l’AOP pour générer des explications à la volée durant la résolution d’un problème de satisfaction de contraintes, nous proposons ici dans un premier temps de tenter de résoudre un problème surcontraint comprenant uniquement des contraintes basiques de différence. En effet, utiliser de telles contraintes à la place de contraintes globales, permet d’augmenter artificiellement la phase de propagation et ainsi d’évaluer de manière pessimiste le coût que pourrait avoir l’utilisation des explications.

Ainsi, nous tentons de résoudre un problème avec n variables, et avec une clique de différence entre les $n/2$ premières variables (ce qui signifie que chaque variable doit avoir une valeur différente de celles de toutes les autres du même ensemble comme pour la contrainte `all_different`), et pour les $n/2$ dernières variables. De plus, une contrainte lie une variable de la première clique avec la deuxième clique. Enfin, chacune des variables contient les mêmes $n/2 - 1$ valeurs. Le problème est donc trivialement sur-contraint.

Si on lance la résolution sans explication, avec des explications implémentées de manière intrusive, et enfin à l’aide d’aspects, on obtient les résultats reportés dans le tableau 1. On remarque d’une part que le coût de la gestion des explications est acceptable (environ deux fois plus de temps dès que le problème devient suffisamment compliqué) mais aussi et surtout que la version traitée avec les aspects est relativement efficace, le surcoût par rapport à la version intrusive étant largement acceptable. De plus, bien entendu, le solveur ne se limite plus à répondre uniquement qu’il n’y a pas de solutions, mais précise aussi qu’une explication peut être constituée d’une seule des deux cliques de différences, ce qui est bien le comportement souhaité.

Afin de tester la solution sur une contrainte glo-

bale, qui comprend notamment des structures de données à maintenir et utiliser pour générer les explications comme nous l’avons vu dans la section précédente, nous proposons de résoudre un problème sur-contraint illustratif avec des contraintes d’occurrence. Pour cela, considérons un problème avec n variables. Les $n - 1$ premières ont un domaine allant de 1 à 5 et la dernière a un domaine qui vaut $\llbracket 5, 6 \rrbracket$. On ajoute ensuite des contraintes d’occurrence sur l’ensemble de ces contraintes pour assurer que la valeur 3 apparaît $n/2$ fois, que la valeur 4 apparaît exactement $n/2$ fois aussi et la valeur 6 une fois (en fait cette dernière contrainte et la dernière variable ne font qu’ajouter du bruit pour tester l’explication retournée). Encore une fois, il est trivial de voir que le problème est sur-contraint. Si on lance la résolution avec les explications, on obtient bien une explication contenant les deux premières occurrences qui sont bien inconsistantes.

Les expérimentations permettent d’obtenir les résultats reportés dans le tableau 1. Comme on peut le voir, l’exploitation des explications a même permis d’améliorer le temps nécessaire pour déterminer le fait que le problème est sur-contraint grâce au gain que l’on peut tirer de `mac-cbj` (même s’il s’agit d’un phénomène qui ne se généralise malheureusement pas souvent, cela montre bien que la génération des explications peut être amortie par l’exploitation de cette information durant la recherche en évitant d’explorer des régions de l’espace de recherche vouées à l’échec).

Il ne s’agit ici que de tests préliminaires, une campagne de tests étant en cours de réalisation pour tester ces résultats sur des problèmes plus réalistes. Cependant ces premiers résultats montrent bien d’une part que la proposition d’utiliser les aspects fonctionne bien (il est possible de modifier un solveur déjà existant sans modifier son code de manière intrusive, en ce sens il s’agit déjà d’un test réaliste) et que son surcoût par rapport à une méthode classique est minime.

6 Comparaisons des techniques et conclusion

Nous avons présenté dans ce papier les difficultés que peut poser l’ajout d’une nouvelle fonctionnalité comme les explications dans un solveur de contraintes. Outre la solution classique qui consiste à implémenter la nouvelle fonctionnalité dans tout le solveur, nous proposons deux nouvelles méthodes pour tenter de réduire le coût d’un tel développement. Ainsi, le tableau 2 résume les avantages de chaque proposition, depuis l’utilisation de cadres génériques qui peut demander un temps de développement conséquent au début mais fournit ensuite à moindre coût un très grand nombre de contraintes avec la fonctionnalité souhaitée (que ce

	Méthode ad hoc	Cadres génériques	AOP
Investissement initial	faible	fort	moyen
Développement pour chaque contrainte	fort	faible	moyen
Efficacité	fort	moyen	assez fort
Problèmes dynamiques	gérés	gérés	non gérés

TAB. 2 – Avantages et inconvénients des différentes méthodes

soit les explications ou la génération de trace) si l'on accepte de perdre une partie de la complétude du filtrage sur certaines contraintes ; jusqu'à l'utilisation de la programmation par aspects qui permet d'ajouter ou de modifier le code d'un logiciel de manière non intrusive en définissant des coupes où l'on souhaite ajouter un nouveau code (en effet, cette méthode ne nécessite pas de modifier le code original ni de réduire l'efficacité du solveur lorsque l'on utilise pas les explications).

Comme nous l'avons vu cette dernière technique peut se révéler très efficace puisque le surcoût est minime. La principale limitation actuellement est certainement la connaissance intime nécessaire du solveur. Nous travaillons actuellement au développement d'un langage d'aspects pour la programmation par contraintes indépendant du solveur utilisé manipulant des concepts haut niveau spécifiques à la programmation par contraintes (à l'image de ce qui a été fait pour la trace trace GENTRA4CP du projet OADYMPPAC[3].

Dans de futurs travaux, il semble intéressant d'implémenter la deuxième proposition basée sur les cadres génériques pour pouvoir évaluer de manière pratique d'une part son efficacité en terme de temps de calcul mais aussi en terme de qualité d'explications générées. En effet, cette solution semble très intéressante dans le sens où elle répond à un deuxième problème de la programmation par contraintes, à savoir le coût de développement des contraintes globales qui entraîne souvent la présence d'un nombre limité de contraintes dans les solveurs académiques.

Références

- [1] Nicolas Beldiceanu, Mats Carlsson, and Thierry Petit. Deriving filtering algorithms from constraint checkers. In *Principles and Practice of Constraint Programming (CP'04)*, pages 107–122, 2004.
- [2] Romuald Debruyne, Gérard Ferrand, Narendra Jussien, Willy Lesaint, Samir Ouis, and Alexandre Tessier. Correctness of constraint retraction algorithms. In *FLAIRS'03*, pages 172–176, St. Augustine, Florida, USA, 2003.
- [3] Pierre Deransart. Projet OADYMPPAC, 2004. Site : contraintes.inria.fr/OADymPPaC/.
- [4] Étienne Gaudin, Narendra Jussien, and Guillaume Rochart. Implementing explained global constraints. In *CP04 Workshop on Constraint Propagation and Implementation (CPAI'04)*, pages 61–76, 2004.
- [5] Dávid Hanák. Implementing global constraints as graphs of elementary constraints. *Acta Cybern.*, 16(2) :241–258, 2003.
- [6] Ulrich Junker. QUICKXPLAIN : Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints*, Seattle, WA, USA, August 2001.
- [7] Narendra Jussien and Vincent Barichard. The PaLM system : explanation-based constraint programming. In *TRICS a post-conference workshop of CP'00*, pages 118–133, September 2000.
- [8] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *CP'00*, number 1894 in LNCS, pages 249–261, 2000.
- [9] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, pages 220–242, 1997.
- [10] François Laburthe. Choco : implementing a cp kernel. In *TRICS, a post-conference workshop of CP 2000*, 2000. Site : choco.sourceforge.net.
- [11] Ludovic Langevine. Explication systématique des contraintes indexicales. In *JFPC'05*, Lens, France, May 2005.
- [12] Gilles Pesant. A filtering algorithm for the stretch constraint. In *Principles and Practice of CP (CP 2001)*, LNCS, pages 183–195, 2001.
- [13] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In *CP'04*, pages 482–495, 2004.
- [14] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3) :268–299, August 1993.
- [15] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI '94*, pages 362–367, 1994.
- [16] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *AAAI/IAAI, Vol. 1*, pages 209–215, 1996.
- [17] Guillaume Rochart and Narendra Jussien. Une contrainte stretch expliquée. *JEDAI : Journal Électronique d'Intelligence Artificielle*, 3(31), 2004.