Un solveur de contraintes basé sur les domaines abstraits

Marie Pelleau[1] **Antoine Miné**[2] **Charlotte Truchet**[1] **Frédéric Benhamou**[1]

[1] Université de Nantes, 2 rue de la Houssinière, Nantes

[2] École normale supérieure, 45, rue d'Ulm, Paris

{Prénom.Nom}@univ-nantes.fr Antoine.Mine@ens.fr

La version anglaise de cet article a été publié à la conférence VMCAI 2013 [17].

Résumé

Dans cet article, nous utilisons des techniques de l'interprétation abstraite (une théorie d'approximation des sémantiques) dans le cadre de la programmation par contraintes (basée sur la logique du premier ordre qui permet de résoudre des problèmes combinatoires). Nous mettons en évidence certains liens et différences entre ces domaines de recherches : tous deux calculent itérativement des points fixes mais emploient des extrapolations et stratégies de raffinement différentes. De plus, nous pouvons mettre en correspondance les consistances en programmation par contraintes et les domaines abstraits non relationnels. Nous utilisons ensuite ces correspondances pour construire un solveur de contraintes abstrait qui s'appuie sur des techniques d'interprétation abstraite (comme les domaines relationnels) pour aller au-delà des solveurs classiques. Les résultats expérimentaux obtenus avec notre prototype sont encourageants.

1 Introduction

L'interprétation abstraite (IA) est une méthode d'approximation des sémantiques de programmes. Elle fournit des réponses garanties à des questions sur leur comportements lors de l'exécution [7, 6]. La programmation par contraintes (PPC) vise à résoudre, avec des techniques génériques, des problèmes combinatoires exprimés de manière déclarative. Cet article étudie l'utilisation de techniques de l'IA en PPC.

1.1 État de l'art

Introduite par Montanari [16], la PPC repose sur l'idée que de nombreux problèmes peuvent être exprimés comme une conjonction de formules logiques du premier ordre, appelées contraintes, chacune représentant une caractéristique combinatoire spécifique du problème [19]. Chaque contrainte vient avec des opérateurs *ad hoc* exploitant sa structure interne afin de réduire la combinatoire. Les contraintes sont ensuite combinées dans des algorithmes de résolution. Une grande partie de l'effort de recherche en PPC porte sur la définition et l'amélioration de contraintes [1] et des algorithmes de résolution. La PPC propose désormais de puissantes techniques d'optimisation combinatoire, avec de nombreuses applications pratiques pour la planification, l'emballage, l'attribution des fréquences, etc. Il subsiste cependant des limitations aux solveurs. Ils sont limités à des domaines non relationnels, tels que des boîtes ou des produits cartésiens d'ensembles d'entiers. De plus, les algorithmes de résolution sont séparés en deux familles : l'une manipule des variables discrètes et l'autre des variables continues. Plusieurs méthodes ont été proposées pour traiter les problèmes mixtes, tels que la discrétisation de variables continues afin de les manipuler dans un solveur discret (comme dans Choco [21]). Malheureusement, le solveur reste purement discret et ne bénéficie pas d'heuristiques développées pour les solveurs continus. Alternativement, il est possible d'ajouter des contraintes mixtes [5] ou des contraintes d'intégrité [3] à un solveur continu, avec des inconvénients similaires.

Dans un autre domaine de recherche, celui de la vérification de programmes, l'interprétation abstraite (IA) est utilisée pour concevoir des analyseurs statiques de programmes qui sont corrects et se terminent toujours (comme Astrée [4]) en développant des approximations calculables de problèmes essentiellement indéci-

1. Voir `http://www.emn.fr/z-info/sdemasse/gccat` pour un catalogue de contraintes globales existantes.

dable. La sémantique concrète (incalculable) exprime sous forme de point fixe l'ensemble des comportements observables du programme. Une approximation de la sémantique est calculée dans un domaine abstrait limitant l'expressivité à un ensemble de propriétés intéressantes. Les domaines abstraits fournissent des structures de données pour les représentations, des algorithmes efficaces pour calculer des versions abstraites de la sémantique concrète, des opérateurs de points fixes et des opérateurs d'accélération pour calculer des approximations en un temps fini.

La sûreté garantit que l'analyseur observe un ensemble plus grand que celui des comportements du programme. Les domaines abstraits numériques mettent l'accent sur les variables et les propriétés numériques. Ils en existent de nombreux et ils sont particulièrement bien développés. Les plus connus sont les intervalles [6] et les polyèdres [8]. Ces dernières années, de nouveaux domaines ont été développés tels que les octogones [15] et des bibliothèques de domaines abstraits telles que Apron [12] ont vu le jour. Les domaines abstraits numériques peuvent traiter tous les types de variables numériques, y compris les entiers, les rationnels, les réels et les nombres flottants, et même exprimer des relations entre les variables de différents types [14, 4]. Chaque domaine correspond à un certain compromis entre le coût et la précision. Enfin, les domaines peuvent être modifiés et combinés par des opérateurs génériques, tels que la complétion disjonctive et les produits réduits.

Dans cet article, nous cherchons à utiliser les techniques d'IA pour construire un solveur de PPC qui soit abstrait et générique.

Cet articles est organisé comme suit. La section 2 introduit brièvement l'IA et la PPC, et donne des éléments de comparaison. La section 3 définit la PPC à l'aide de techniques de l'IA et présente notre solveur abstrait. Notre prototype ainsi que des résultats expérimentaux sont présentés section 4.

1.2 Travaux connexes

Certaines interactions entre la PPC et les techniques de vérification ont déjà été explorées dans des travaux antérieurs. Par exemple, la PPC a été utilisée pour générer automatiquement des tests de configuration [11], ou pour vérifier des modèles en PPC [13].

Plus récemment, des travaux tels que [9, 22], établissent des liens entre l'IA et les algorithmes de résolution SAT. Notre but est similaire mais lie la PPC à l'IA. Bien que connexes, la PPC et SAT diffèrent assez significativement dans les modèles choisis et les algorithmes de résolution pour que les résultats précédents ne s'appliquent pas. Notre travail s'inscrit dans la continuité de [18] qui étend les méthodes de résolution en PPC afin d'utiliser des représentations plus expressives, telles que les octogones. Cependant, l'ajout d'un nouveau domaine abstrait demande de définir des opérateurs *ad hoc* (consistance, coupe, ...) Dans cet article, nous effectuons le processus inverse : nous concevons un solveur abstrait reposant sur les domaines abstraits et utilisant les opérateurs existants en IA. Ceci permet entre autres choses de définir la consistance de façon unique pour tous les domaines abstraits.

2 Notations et rappels

Dans cette section nous présentons uniquement les notions d'IA et de PPC nécessaires pour la suite (voir [7, 6] et [19] pour une présentation plus détaillée).

2.1 Bases de l'interprétation abstraite

Nous présentons d'abord des éléments d'IA nécessaire pour la conception de notre solveur.

2.1.1 Abstractions de point fixe

Le domaine concret, noté $\mathcal{D}$, correspond aux valeurs que peuvent prendre les variables tout au long du programme. Calculer la sémantique concrète d'un programme peut être indécidable et une approximation est acceptée. L'approximation du domaine concret est appelée domaine abstrait et est notée $\mathcal{D}^\sharp$. Les domaines abstraits sont regroupés en deux familles, les domaines abstraits non-relationnels et les domaines abstraits relationnels. Un domaine abstrait non-relationnel associe une propriété indépendante à chaque variable (comme ses bornes), tandis qu'un domaine abstrait relationnel exprime des relations entre les variables (comme des relations linéaires), ce qui est bien plus expressif mais aussi plus coûteux.

Une fonction de concrétisation $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$ associe à un élément abstrait un élément concret. Un opérateur abstrait $F^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ est une abstraction correcte de F si $F \circ \gamma \sqsubseteq \gamma \circ F^\sharp$. Il peut exister une fonction d'abstraction $\alpha : \mathcal{D} \to \mathcal{D}^\sharp$ telle que (α, γ) forme une correspondance de Galois (notée $\mathcal{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{D}^\sharp$), assurant que chaque élément concret X admet une abstraction optimale $\alpha(X)$. Si une correspondance de Galois existe entre le domaine concret et le domaine abstrait, toute fonction F dans $\mathcal{D}$ admet une abstraction $F^\sharp$ dans $\mathcal{D}^\sharp$ telle que $\forall X^\sharp \in \mathcal{D}^\sharp, (\alpha \circ F \circ \gamma)(X^\sharp) \sqsubseteq F^\sharp(X^\sharp)$. La fonction abstraite $F^\sharp$ est dite optimale ssi $\alpha \circ F \circ \gamma = F^\sharp$.

Afin d'analyser un programme, chaque ligne de code est analysée. Pour cela, chaque instruction du programme est associée à une fonction, appelée fonction de transfert, qui modifie les valeurs possibles pour les variables.

Définition 2.1 (Fonction de transfert) *Soit C une ligne de code à analyser. Une fonction de transfert $F : \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$ retourne, en fonction d'un ensemble de départ, un ensemble d'environnements correspondant à tous les états accessibles après avoir exécuté C.*

Exemple 2.1 *Considérons une expression booléenne, la fonction de transfert ne conserve que les environnements satisfaisant l'expression booléenne.*

Soit deux variables x et y dont les valeurs possibles sont comprises dans $[-10, 10]$, après l'expression booléenne $x \leq 0$ la fonction de transfert filtre les valeurs de x afin de satisfaire la condition. On a maintenant x dans $[-10, 0]$ et y dans $[-10, 10]$.

Chaque instruction du programme est associée à une fonction de transfert, le programme est donc associé à une composition de ces fonctions. Prouver que le programme est correct revient à calculer le plus petit point fixe de cette composition de fonctions. La sémantique concrète d'un programme est donc donnée comme le plus petit point fixe $\mathrm{lfp}_\perp F$ d'un opérateur $F : \mathcal{D} \to \mathcal{D}$ dans une structure partiellement ordonnée $(\mathcal{D}, \sqsubseteq, \perp, \sqcup)$, telle qu'un treillis ou un ordre partiel complet. De la même façon, notons $(\mathcal{D}^\sharp, \sqsubseteq^\sharp, \perp^\sharp, \sqcup^\sharp)$ le domaine abstrait. Le plus petit point fixe, $\mathrm{lfp}_\perp F$ peut être approximé par $\bigsqcup^\sharp_{i \in \mathrm{Ord}} F^{\sharp i}(\perp^\sharp)$. Cette limite peut ne pas être calculable, même si $F^\sharp$ l'est, ou peut nécessiter un très grand nombre d'itérations. Elle est donc généralement remplacée par la limite d'une séquence croissante : $X^\sharp_0 = \perp^\sharp$, $X^\sharp_{i+1} = X^\sharp_i \triangledown F^\sharp(X^\sharp_i)$ utilisant un opérateur d'élargissement $\triangledown$ afin d'accélérer la convergence. L'élargissement est conçu afin de sur-approximer $\sqcup$ et converger en un temps fini δ en un post point fixe $X^\sharp_\delta$ de $F^\sharp$, ce qui implique $\gamma(X^\sharp_\delta) \sqsupseteq \mathrm{lfp}_\perp F$. Cette limite est généralement améliorée par des itérations décroissantes : $Y^\sharp_0 = X^\sharp_\delta$, $Y^\sharp_{i+1} = Y^\sharp_i \vartriangle F^\sharp(Y^\sharp_i)$, utilisant un opérateur de rétrécissement $\vartriangle$ conçu pour rester au-dessus de tout point fixe de F et converger en un temps fini. Comme tous les $Y^\sharp_i$ sont des abstractions de $\mathrm{lfp}_\perp F$, les itérations peuvent être arrêtées à tout moment.

2.1.2 Itérations locales

Les itérations décroissantes permettent d'améliorer le point fixe calculé, mais ont aussi été utilisées localement par Granger [10], au sein du calcul de $F^\sharp$. Granger observe que l'opérateur concret F utilise souvent des opérateurs de clôture inférieure, *i.e.,* des opérateurs ρ qui sont croissants, idempotents ($\rho \circ \rho = \rho$) et contractants ($\rho(X) \sqsubseteq X$). Étant donnée une abstraction $\rho^\sharp$ de ρ, la limite $Y^\sharp_\delta$ de la séquence $Y^\sharp_0 = X^\sharp$, $Y^\sharp_{i+1} = Y^\sharp_i \vartriangle \rho^\sharp(Y^\sharp_i)$ est une abstraction de $\rho(\gamma(X^\sharp))$. Si $\rho^\sharp$ n'est pas une abstraction optimale de ρ, $Y^\sharp_\delta$ peut être significativement plus précis que $\rho^\sharp(X^\sharp)$. Une application pertinente est celle de l'analyse de conjonctions de tests complexes $C_1 \wedge \cdots \wedge C_p$ où chaque test atomique C_i est modélisé dans l'abstraction par $\rho^\sharp_i$. Généralement, $\rho^\sharp = \rho^\sharp_1 \circ \cdots \circ \rho^\sharp_p$ n'est pas optimal, même si chaque $\rho^\sharp_i$ l'est. Une application complémentaire est l'analyse d'un seul test C_i utilisant une séquence non-optimale d'abstractions de tests relaxés. Par exemple, les expressions non-linéaires peuvent être remplacées par des intervalles calculés avec les bornes des variables [14]. Comme l'exécution des tests relaxés améliore ces bornes, la relaxation n'est pas idempotente et peut être améliorée par les itérations locales. Le lien entre les itérations locales et l'amélioration du plus petit point fixe repose sur le fait que $\rho(X)$ calcule un point fixe trivial : le plus grand point fixe de ρ plus petit que X : $\mathrm{gfp}_X\, \rho$. Dans ces deux cas, une itération décroissante commence par une abstraction d'un point fixe ($\mathrm{lfp}_\perp F$ dans un cas, $\mathrm{gfp}_X\, \rho$ dans l'autre) et calcule une abstraction plus petite de ce point fixe.

2.2 Programmation par contraintes

Nous présentons maintenant des définitions basiques de PPC. Dans cette section, nous utilisons la terminologie de PPC, et précisons les termes avec un sens différents en IA et PPC.

Les problèmes sont modélisés sous forme d'un problème de satisfaction de contraintes (CSP) tel que défini ci-dessous :

Définition 2.2 *Un CSP est défini par un ensemble de variables $(v_1, \ldots, v_n)$ prenant leurs valeurs dans les domaines $(\hat{D}_1, \ldots, \hat{D}_n)$ et un ensemble de contraintes $(C_1, \ldots, C_p)$.*

Soit D_i le domaine de la variable v_i, notons $D = D_1 \times \cdots \times D_n$ l'*espace de recherche*. Comme l'espace de recherche est modifié durant la résolution, nous distinguons l'espace de recherche initial et le notons $\hat{D} = \hat{D}_1 \times \cdots \times \hat{D}_n$. Les problèmes peuvent être discrets ($\hat{D} \subseteq \mathbb{Z}^n$) ou continus ($\hat{D} \subseteq \mathbb{R}^n$). Cependant, les domaines sont toujours bornés.

Soit D_i le domaine de la variable v_i, et $x_i \in D_i$. On note $C(x_1, \ldots, x_n)$ le fait que la contrainte est satisfaite quand chaque variable v_i prend la valeur x_i. L'ensemble de solutions est $S = \{(s_1, \ldots, s_n) \in \hat{D} \,|\, \forall i \in [\![1, p]\!],\, C_i(s_1, \ldots, s_n)\}$, avec p le nombre de contraintes et où $[\![a, b]\!] = \{x \in \mathbb{Z} \,|\, a \leq x \leq b\}$ correspond à l'intervalle des entiers entre a et b.

Pour les problèmes discrets, deux représentations des domaines sont généralement utilisées : les en-

sembles et les intervalles. Nous ne présentons que la représentation ensembliste.

Définition 2.3 (Produit cartésien d'entiers) *Soient $v_1 \dots v_n$ des variables de domaines discrets finis $\hat{D}_1 \dots \hat{D}_n$. Un produit cartésien d'ensembles d'entiers est appelé produit cartésien d'entiers et est exprimé dans $\mathcal{D}$ par :* $\mathcal{S}^\sharp = \left\{ \prod_i X_i \,|\, \forall i, X_i \subseteq \hat{D}_i \right\}$

Pour les problèmes continus, les domaines sont représentés par des intervalles à bornes flottantes. Soit $\mathbb{F}$ l'ensemble des flottants représentables en machine. Soit $a, b \in \mathbb{F}$, notons $[a, b] = \{x \in \mathbb{R} \,|\, a \leq x \leq b\}$ l'intervalle des réels compris entre a et b, et $\mathbb{I} = \{[a, b] \,|\, a, b \in \mathbb{F}\}$ l'ensemble de tous ces intervalles.

Définition 2.4 (Boîte) *Soient $v_1 \dots v_n$ des variables de domaines continus $\hat{D}_1 \dots \hat{D}_n$. Un produit cartésien d'intervalles est appelé boîte et est exprimé dans $\mathcal{D}$ par :* $\mathcal{B}^\sharp = \left\{ \prod_i I_i \,|\, I_i \in \mathbb{I}, I_i \subseteq \hat{D}_i \right\} \cup \emptyset$

Résoudre un CSP revient à calculer exactement ou une approximation de l'ensemble des solutions S.

Définition 2.5 (Approximation) *Une* approximation complète *(resp.* correcte*), de l'ensemble des solutions est un ensemble $\mathcal{A}$ de produits de domaines telle que $\forall (D_1, \dots, D_n) \in \mathcal{A}, \forall i,\ D_i \subseteq \hat{D}_i$ et $S \subseteq \bigcup_{(D_1,\dots,D_n)\in\mathcal{A}} D_1 \times \dots \times D_n$ (resp. $\bigcup_{(D_1,\dots,D_n)\in\mathcal{A}} D_1 \times \dots \times D_n \subseteq S$).*

La correction garantit que toutes les solutions sont trouvées, alors que la complétude garantit qu'aucune solution n'est perdue. En pratique, un solveur de contraintes sur les domaines finis doit être complet et correct, ce qui revient à calculer les solutions. Cela n'est pas possible dans le cas de domaines continus car les réels ne sont pas représentables exactement en machine, et un solveur continu sera soit complet (la plupart du temps) soit correct. Les notions de correction et complétude sont différentes en IA et en PPC. Afin d'éviter toute ambiguïté, le terme sur-approximation (resp. sous-approximation) sera utilisé pour une approximation PPC-complète IA-correcte (resp. PPC-correcte IA-complète).

Dans cet article, nous considérons les méthodes de résolution qui sur-approximent les solutions des problèmes continus et calculent les solutions exactes pour les problèmes discrets. Ces méthodes alternent deux étapes : la propagation et l'exploration.

2.2.1 Propagation

Le but de la propagation est d'utiliser les contraintes afin de réduire les domaines. Intuitivement, nous supprimons des domaines les valeurs inconsistantes, *i.e.,* les valeurs ne pouvant pas être dans une solution. Plusieurs formes de consistance ont été proposées. Nous présentons ici les plus communes.

Définition 2.6 (Consistance d'arc généralisée) *Soient $v_1 \dots v_n$ des variables de domaines discrets finis $D_1 \dots D_n, D_i \subseteq \hat{D}_i$, et C une contrainte. Les domaines sont dits* arc-consistants généralisés *(GAC) pour C ssi $\forall i \in [\![1, n]\!], \forall x_i \in D_i, \forall j \neq i, \exists x_j \in D_j$ tel que $C(x_1, x_2, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_n)$.*

Définition 2.7 (Consistance d'enveloppe) *Soient $v_1 \dots v_n$ des variables de domaines continus représentés par les intervalles $D_1 \dots D_n \in \mathbb{I}, D_i \subseteq \hat{D}_i$, et C une contrainte. Les domaines $D_1 \dots D_n$ sont dits Hull-consistant (HC) pour C ssi $D_1 \times \dots \times D_n$ est la plus petite boîte à bornes flottantes contenant les solutions de C dans $D_1 \times \dots \times D_n$.*

Chaque contrainte C et consistance vient avec un algorithme appelé *propagateur*, qui essaie d'atteindre la consistance. Quand plusieurs contraintes sont considérées, une *boucle de propagation* exécute les propagateurs des contraintes jusqu'à ce qu'un point fixe soit atteint. Comme indiqué dans [2], l'ordre d'exécution des propagateurs n'a pas d'importance car l'ensemble des domaines forme un treillis fini ($\mathcal{B}^\sharp$, $\mathcal{I}^\sharp$ ou $\mathcal{S}^\sharp$) et le point fixe consistant est son unique plus petit élément. Lorsque la consistance est trop coûteuse à réaliser, les propagateurs et les boucles de propagations calculent une sur-approximation (*e.g.,* enlève seulement quelques valeurs inconsistantes). En plus de restreindre l'espace de recherche, la propagation est parfois en mesure de découvrir qu'il n'existe pas de solution, ou que tous les points sont des solutions.

2.2.2 Exploration

En règle générale, la propagation seule ne permet pas de calculer les solutions exactes (dans le cas discret) ou une sur-approximation assez précise (dans le cas continu). Ainsi, dans une deuxième étape, différentes hypothèses sur les valeurs des variables sont testées. Dans le cas discret, une variable est choisie et est instanciée pour chaque valeur dans son domaine. Dans le cas continu, son domaine est divisé en deux sous-domaines plus petits. L'algorithme de résolution continue en sélectionnant un espace de recherche et applique une nouvelle étape de propagation (car il peut ne plus être consistant), puis fait d'autres choix. Ces itérations de propagation et de choix s'arrêtent lorsqu'il peut être prouvé que l'espace de recherche ne contient pas de solution, seulement des solutions ou, dans le cas continu, lorsque sa taille est inférieure à un seuil spécifié par l'utilisateur. Dans le cas discret, dans le pire des cas, toutes les variables sont instanciées.

Après avoir exploré une branche, en cas d'erreur ou si toutes les solutions doivent être calculées, l'algorithme revient à un point de choix précédent (instanciation ou coupe) par *backtracking* et tente une autre hypothèse.

2.3 Comparaison entre l'IA et la PPC

Cette section présente, de manière informelle, certains liens entre l'IA et la PPC. La section suivante formalisera ces liens en exprimant la PPC dans le cadre de l'IA.

Les techniques de ces deux domaines reposent sur la théorie des points fixes dans les treillis. Ils poursuivent des objectifs similaires : calculer ou sur-approximer les solutions d'équations complexes en manipulant des vues abstraites d'ensembles de solutions potentielles, comme les boîtes (appelés domaines en PPC et éléments de domaines abstraits en IA). Cependant, les méthodes diffèrent, les méthodes de résolution visent la complétude et améliore donc jusqu'à une précision arbitraire. Au contraire, la précision des analyseurs abstraits est fixée par le choix du domaine abstrait : ils peuvent rarement calculer des sur-approximations de précision arbitraire. Le choix du domaine abstrait définit le coût et la précision d'un analyseur, alors que le choix du domaine en PPC définit le coût d'un solveur pour atteindre une précision donnée.

Même si ils visent la complétude, les solveurs utilisent néanmoins des domaines non relationnels simples. Ils s'appuient sur les collections de domaines simples (disjonctions en IA) pour atteindre la précision souhaitée. Les domaines sont homogènes et ne peuvent pas mélanger des variables de type différent. Au contraire, l'IA possède de nombreux domaines abstraits, pouvant être relationnels et hétérogènes.

Sur le plan algorithmique, l'IA et la PPC partagent des idées communes. L'itération des propagations en PPC est similaires aux itérations locales en IA. En effet, approximer une consistance en PPC est similaire à approximer l'effet d'un test complexe en IA. Cependant, les méthodes de résolution en PPC utilisent des techniques telles que les points de choix et le backtracking, qui n'ont pas d'équivalent en IA. Inversement, l'élargissement de l'IA n'a pas d'équivalent en PPC, car la PPC n'emploie pas d'itérations croissantes.

Enfin, alors que les analyseurs abstraits sont généralement définis de façon très générique et paramétrable grâce aux domaines abstraits, les solveurs sont bien moins flexibles et incorporent les domaines et les consistances ainsi que le type des variables dans leur conception. Par la suite, nous concevons un solveur abstrait qui permet d'éviter ces désavantages et permet de bénéficier de la grande bibliothèque de domaines abstraits conçus pour l'IA.

3 Un solveur de contraintes abstrait

Nous présentons maintenant notre principale contribution : exprimer la résolution de contraintes comme un analyseur abstrait. Nous définissons donc les domaines concrets et abstraits, les opérateurs abstraits pour la consistance et le choix, et le schéma itératif.

3.1 Résolution concrète

Un CSP est similaire à l'analyse d'une conjonction de tests et peut être formalisé en terme d'itérations locales. Considérons comme domaine concret $\mathcal{D}$ les sous-ensembles de l'espace de recherche $\hat{D} = \hat{D}_1 \times \cdots \times \hat{D}_n$ du CSP, *i.e.,* $(\mathcal{P}(\hat{D}), \subseteq, \emptyset, \cup)$. Chaque contrainte C_i vient avec un opérateur de clôture inférieure $\rho_i : \mathcal{P}(\hat{D}) \to \mathcal{P}(\hat{D})$, tel que $\rho_i(X)$ conserve uniquement les points de X satisfaisant C_i. Les solutions concrètes du problèmes sont $S = \rho(\hat{D})$, où $\rho = \rho_1 \circ \cdots \circ \rho_p$, et peuvent être exprimées sous forme de point fixe $\mathrm{gfp}_{\hat{D}}\, \rho$.

3.2 Domaines abstraits

Les solveurs ne manipulent pas des points dans $\hat{D}$, mais plutôt des collections de points d'une certaine forme, telles que des boîtes, appelées domaines en PPC. Nous montrons que les domaines en PPC sont des éléments d'un domaine abstrait $(\mathcal{D}^\sharp, \sqsubseteq^\sharp, \bot^\sharp, \sqcup^\sharp)$ en IA, dépendant d'une consistance. Nous ajoutons aussi une fonction *de précision* $\tau : \mathcal{D}^\sharp \to \mathbb{R}^+$ qui sera utilisée dans la condition d'arrêt (Def. 3.2).

Exemple 3.1 *La consistance d'arc généralisée (Def. 2.6) correspond au domaine abstrait des produits cartésiens d'entiers $\mathcal{S}^\sharp$ (Def. 2.3), ordonné par l'inclusion sur les ensembles. Elle est liée au domaine concret $\mathcal{D}$ par une correspondance de Galois standard :*
$\mathcal{D} \underset{\alpha_a}{\overset{\gamma_a}{\leftrightarrows}} \mathcal{S}^\sharp$
$\gamma_a(S_1, \ldots, S_n) = S_1 \times \cdots \times S_n$
$\alpha_a(X) = \lambda i.\{x \mid \exists (x_1, \ldots, x_n) \in X,\, x_i = x\}$

La fonction de précision τ_a utilise la taille du plus grand ensemble d'entiers, moins un, de sorte que les singletons aient une taille 0 : $\tau_a(S_1, \ldots, S_n) = \max_i(|S_i| - 1)$

Exemple 3.2 *La consistance d'enveloppe (Def. 2.7) correspond au domaine des boîtes à bornes flottantes $\mathcal{B}^\sharp$ (Def. 2.4). Nous utilisons la correspondance de Galois et la fonction de précision suivantes :*
$\mathcal{D} \underset{\alpha_h}{\overset{\gamma_h}{\leftrightarrows}} \mathcal{B}^\sharp$
$\gamma_h([a_1, b_1], \ldots, [a_n, b_n]) = [a_1, b_1] \times \cdots \times [a_n, b_n]$

$\alpha_h(X) = \lambda i.[\ \max\{x \in \mathbb{F} \mid \forall (x_1, \ldots, x_n) \in X,\ x_i \geq x\},$
$\min\{x \in \mathbb{F} \mid \forall (x_1, \ldots, x_n) \in X,\ x_i \leq x\}]$
$\tau_h([a_1, b_1], \ldots, [a_n, b_n]) = \max_i(b_i - a_i)$

Notons qu'à chacun des choix correspond un domaine abstrait non-relationnel classique en IA. De plus, il correspond à un produit cartésien homogène d'un seul et même ensemble de base (type de représentation pour une variable). Cependant, ceci n'est pas obligatoire, et de nouveaux solveurs peuvent être conçus en modifiant davantage les domaines abstraits. Une première idée est d'appliquer différentes consistances à différentes variables, ce qui permettrait en particulier de mélanger des variables discrètes avec des variables continues. Une seconde idée est de paramétrer le solveur avec d'autres domaines abstraits existant en IA, en particulier les domaines abstraits relationnels ce que nous illustrons ci-dessous.

Exemple 3.3 *Le domaine abstrait des octogones $\mathcal{O}^\sharp$ [15] assigne une borne (flottante) supérieure à chaque expression binaire unitaire $\pm v_i \pm v_j$ sur les variables $v_1, \ldots, v_n$. Il admet une correspondance de Galois et la fonction de précision définie dans [18] :*
$\mathcal{D} \underset{\alpha_o}{\overset{\gamma_o}{\leftrightarrows}} \mathcal{O}^\sharp$
$\mathcal{O}^\sharp = \{\alpha v_i + \beta v_j \mid i, j \in [\![1, n]\!],\ \alpha, \beta \in \{-1, 1\}\} \to \mathbb{F}$
$\gamma_o(X^\sharp) = \{(x_1, \ldots, x_n) \in \mathbb{R}^n \mid \forall i, j, \alpha, \beta, \alpha x_i + \beta x_j \leq X^\sharp(\alpha v_i + \beta v_j)\}$
$\alpha_o(X) = \lambda(\alpha v_i + \beta v_j).\min\{x \in \mathbb{F} \mid \forall (x_1, \ldots, x_n) \in X,\ \alpha x_i + \beta x_j \leq x\}$
$\tau_o(X^\sharp) = \min(\max_{i,j,\beta}(X^\sharp(v_i + \beta v_j) + X^\sharp(-v_i - \beta v_j)),$
$\max_i(X^\sharp(v_i + v_i) + X^\sharp(-v_i - v_i))/2)$
En représentant un octogone comme une intersection de boîtes, la fonction de précision retourne la plus grande dimension de la plus petite boîte.

Le domaine abstrait des polyèdres $\mathcal{P}^\sharp$ peut aussi être défini comme montré dans [17].

3.3 Contraintes et consistance

À partir de maintenant, nous supposons qu'un domaine abstrait $\mathcal{D}^\sharp$ sous-jacent au solveur est fixé. Pour une sémantique concrète de contraintes $\rho = \rho_1 \circ \cdots \circ \rho_p$ donnée, et si $\mathcal{D}^\sharp$ admet une correspondance de Galois $\mathcal{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{D}^\sharp$, alors la sémantique du propagateur optimal réalisant la consistance pour toutes les contraintes est simplement : $\alpha \circ \rho \circ \gamma$. Les solveurs réalisent ceci de façon algorithmique, ils exécutent le propagateur de chaque contrainte jusqu'à atteindre un point fixe ou, quand ce processus est estimé trop coûteux, s'arrêtent avant que le point fixe ne soit atteint. En observant que chaque propagateur correspond à une fonction abstraite modélisant un test $\rho_i^\sharp$ dans $\mathcal{D}^\sharp$, nous retrouvons les itérations locales de Granger pour analyser des conjonctions de tests [10].

3.4 Complétion disjonctive et coupe

Afin d'approximer les solutions à une précision arbitraire, les solveurs utilisent un pavage fini d'éléments abstraits de $\mathcal{D}^\sharp$. En IA, un pavage fini correspond à une complétion disjonctive [7]. Considérons le domaine abstrait $\mathcal{E}^\sharp = \mathcal{P}_{\text{fini}}(\mathcal{D}^\sharp)$, et équipons-le d'un ordre de Smyth $\sqsubseteq_{\mathcal{E}}^\sharp$, un ordre classique pour les complétions disjonctives défini ainsi : $X^\sharp \sqsubseteq_{\mathcal{E}}^\sharp Y^\sharp \iff \forall B^\sharp \in X^\sharp, \exists C^\sharp \in Y^\sharp,\ B^\sharp \sqsubseteq^\sharp C^\sharp$

Définition 3.1 (Opérateur de coupe) *Un* opérateur de coupe *est un opérateur $\oplus : \mathcal{D}^\sharp \to \mathcal{E}^\sharp$ tel que*

1. $\forall e \in \mathcal{D}^\sharp,\ |\oplus(e)|$ *est fini,*
2. $\forall e \in \mathcal{D}^\sharp, \forall e_i \in \oplus(e),\ e_i \sqsubseteq^\sharp e$, *et*
3. $\forall e \in \mathcal{D}^\sharp,\ \gamma(e) = \bigcup\{\gamma(e_i) \mid e_i \in \oplus(e)\}$.

Chaque élément de $\oplus(e)$ étant inclus dans e (condition 2), on a $\oplus(e) \sqsubseteq_{\mathcal{E}}^\sharp \{e\}$. De plus, la condition 3 montre que $\oplus$ est une abstraction de l'identité. Et donc, $\oplus$ peut être utilisé à n'importe quel moment de la résolution sans en altérer la correction. Nous en présentons ici quelques exemples.

Exemple 3.4 (Coupe dans $\mathcal{S}^\sharp$) *L'instanciation d'une variable v_i de domaine discret $X^\sharp = (S_1, \ldots, S_n) \in \mathcal{S}^\sharp$ est un opérateur de coupe : $\oplus_a(X^\sharp) = \{(S_1, \ldots, S_{i-1}, x, S_{i+1}, \ldots, S_n) \mid x \in S_i\}$*

Exemple 3.5 (Coupe dans $\mathcal{B}^\sharp$) *Couper une boîte en deux le long d'une variable v_i de domaine continu $X^\sharp = (I_1, \ldots, I_n) \in \mathcal{B}^\sharp$ est un opérateur de coupe : $\oplus_h(X^\sharp) = \{(I_1, \ldots, I_{i-1}, [a, h], I_{i+1}, \ldots, I_n), (I_1, \ldots, I_{i-1}, [h, b], I_{i+1}, \ldots, I_n)\}$ où $I_i = [a, b]$ et $h = (a+b)/2$ arrondi dans $\mathbb{F}$ de manière quelconque.*

Exemple 3.6 (Coupe dans $\mathcal{O}^\sharp$) *Étant donnée une expression binaire $\alpha v_i + \beta v_j$, nous définissons l'opérateur de coupe pour les octogones $X^\sharp \in \mathcal{O}^\sharp$ le long de cette expression comme : $\oplus_o(X^\sharp) = \{X^\sharp[(\alpha v_i + \beta v_j) \mapsto h],\ X^\sharp[(-\alpha v_i - \beta v_j) \mapsto -h]\}$ où $h = (X^\sharp(\alpha v_i + \beta v_j) - X^\sharp(-\alpha v_i - \beta v_j))/2$, arrondi dans $\mathbb{F}$ de manière quelconque.*

De même, nous pouvons définir un opérateur de coupe pour les polyèdres (voir [17]).

Ces opérateurs de coupe sont paramétrés par le choix d'une direction de coupe (le long d'une variable ou d'une expression). Pour un domaine non-relationnel, nous pouvons utiliser les deux stratégies classiques en PPC : définir un ordre et couper les variables tour à tour suivant l'ordre choisi, ou couper

le long de la variable ayant le plus grand ou le plus petit domaine (*i.e.,* $|S_i|$ ou $b_i - a_i$). On peut naturellement étendre ces stratégies aux octogones en remplaçant l'ensemble des variables par l'ensemble (fini) des expressions binaires ou utiliser un opérateur de coupe plus rapide et plus simple tel que couper le long de la variable ayant le plus grand domaine.

Afin d'assurer la terminaison du solveur, nous imposons que toute suite de réductions, coupes et sélections retourne à partir d'un moment un élément assez petit pour τ :

Définition 3.2 *Les deux opérateurs* $\tau : \mathcal{D}^\sharp \to \mathbb{R}^+$ *et* $\oplus : \mathcal{D}^\sharp \to \mathcal{E}^\sharp$ *sont dits compatibles si, pour tout opérateur de réduction* $\rho^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ *(i.e.,* $\forall X^\sharp \in \mathcal{D}^\sharp, \rho^\sharp(X^\sharp) \sqsubseteq^\sharp X^\sharp$*) et pour toute famille d'opérateurs de sélection* $\pi_i : \mathcal{E}^\sharp \to \mathcal{D}^\sharp$ *(i.e.,* $\forall Y^\sharp \in \mathcal{E}^\sharp, \pi_i(Y^\sharp) \in Y^\sharp$*) :* $\forall e \in \mathcal{D}^\sharp, \forall r \in \mathbb{R}^{>0}, \exists K$ *tel que* $\forall j \geq K$, $(\tau \circ \pi_j \circ \oplus \circ \rho^\sharp \circ \cdots \circ \pi_1 \circ \oplus \circ \rho^\sharp)(e) \leq r$.

Les opérateurs de coupes définis précédemment, $\oplus_a$, $\oplus_h$ et $\oplus_o$ sont compatibles avec les fonctions de précision τ_a, τ_h et τ_o.

La procédure d'exploration peut être représentée par un arbre de recherche où chaque nœud correspond à un espace de recherche et dont les fils sont construits par l'opérateur de coupe. Avec cette représentation, l'ensemble des nœuds à une profondeur donnée correspond à une disjonction sur-approximant l'ensemble des solutions. De plus, une série d'opérateurs de réduction (ρ), de sélection (π) et de coupe ($\oplus$) correspond à une branche. La définition 3.2 stipule que chaque branche de l'arbre de recherche est finie.

3.5 Résolution abstraite

Notre algorithme de résolution est présenté Fig. 1. Il maintient dans toExplore et sols deux disjonctions dans $\mathcal{E}^\sharp$, et itère les étapes suivantes : choisir un élément abstrait e de toExplore (**pop**), appliquer la consistance ($\rho^\sharp$), et soit supprimer e, soit l'ajouter à la liste de solutions sols, soit le couper ($\oplus$). La résolution commence avec le plus grand élément $\top^\sharp$ de $\mathcal{D}^\sharp$ tel que $\gamma(\top^\sharp) = \hat{D}$.

3.5.1 Correction et terminaison

À chaque itération, $\bigcup\{\gamma(x) \mid x \in \text{toExplore} \cup \text{sols}\}$ est une sur-approximation de l'ensemble des solutions, car la consistance $\rho^\sharp$ est une abstraction de la sémantique concrète ρ pour les contraintes et l'opérateur de coupe $\oplus$ est une abstraction de l'identité. Notons que les éléments abstraits dans sols sont consistants et soit contiennent uniquement des solutions soit sont plus petits que r. L'algorithme termine quand toExplore est vide, et donc sols sur-approxime l'ensemble des solutions avec des éléments consistants contenant uniquement des solutions ou plus petits que r. Afin de calculer exactement l'ensemble des solutions dans le cas discret, il suffit de choisir $r = 0$.

La terminaison est assurée par la proposition suivante :

Proposition 3.1 *Si* τ *et* $\oplus$ *sont compatibles, l'algorithme 1 se termine.*

Preuve Supposons par l'absurde que l'arbre de recherche est infini. Sa largeur étant finie par Def. 3.1, il devrait exister une branche infinie (lemme de König), ce qui contredit Def. 3.2.

L'arbre de recherche étant fini, l'algorithme 1 se termine.

L'algorithme 1 utilise une structure de données de file et coupe le plus ancien élément abstrait en premier. Des stratégies de sélection plus élaborées, telles que couper l'élément le plus grand pour τ, peuvent être définies. Quelle que soit la stratégie choisie, l'algorithme reste correct et termine.

3.5.2 Comparaison avec l'IA

Comme les itérations locales en IA, notre solveur effectue des itérations abstraites décroissantes. Plus précisément, toExplore$\cup$sols est décroissant pour $\sqsubseteq^\sharp_{\mathcal{E}}$ dans le domaine de complétion disjonctive $\mathcal{E}^\sharp$ à chaque itération de la boucle. En effet, $\rho^\sharp$ est contractant dans $\mathcal{D}^\sharp$, et $\oplus(e) \sqsubseteq^\sharp_{\mathcal{E}} \{e\}$. Cependant, notre solveur diffère de l'IA classique sur deux points. Premièrement, il n'existe pas d'opérateurs de coupe en IA, les nouveaux éléments d'une disjonction étant généralement ajoutés au niveau des jointures dans le graphe de flot de contrôle. Deuxièmement, la stratégie de résolution itérative est plus élaborée qu'en IA. L'utilisation de l'opérateur de rétrécissement est remplacée par une structure de données permettant de maintenir une liste ordonnée des éléments abstraits et une stratégie de coupe permet une réduction et assure sa terminaison.

4 Le solveur Absolute

Nous avons implanté un prototype de solveur abstrait, d'après les idées ci-dessus. Les principales caractéristiques et résultats préliminaires sont présentés ici.

4.1 Implémentation

Notre prototype de solveur, appelé Absolute, est implanté au-dessus de Apron, une bibliothèque OCaml de domaines abstraits numériques pour l'analyse statique [12]. Différents domaines abstraits sont implantés dans

```
liste de domaines abstraits sols ← ∅
queue de domaines abstraits toExplore ← ∅
domaine abstrait e ∈ D♯
push ⊤♯ dans toExplore
while toExplore ≠ ∅ do
    e ← ρ♯(pop(toExplore))
    if e ≠ ∅ then
        if τ(e) ≤ r ou isSol(e) then
            sols ← sols ∪ e
        else
            push ⊕(e) dans toExplore
        end
    end
end
```

Algorithm 1: Solveur abstrait générique

Apron, tels que les intervalles, les octogones et les polyèdres. De plus, Apron offre une API uniforme permettant de cacher les algorithmes internes des domaines abstraits, et permet de traiter les variables entières et réelles et les contraintes non linéaires.

4.1.1 Consistance

Apron fournit un langage de contraintes arithmétiques suffisant pour exprimer la plupart des CSP numériques : égalités et inégalités d'expressions numériques, comprenant les opérateurs $+$, $-$, $\times$, $/$, $\sqrt{}$, puissance, modulo et arrondi vers des entiers. Les fonctions de transfert de tests fournissent naturellement des propagateurs pour les contraintes. En interne, chaque domaine abstrait implante ses propres algorithmes afin de gérer les tests, comprenant des méthodes sophistiquées permettant de traiter les contraintes non-linéaires (telles que HC4 et la linéarisation [14]). Notre solveur effectue des itérations locales jusqu'à ce qu'un point fixe ou un nombre maximum d'itérations soit atteint (fixé à trois afin d'assurer une résolution rapide). Dans les solveurs de PPC, seules les contraintes contenant au moins une variable dont le domaine a été modifié à l'itération précédente sont propagées. Cependant, pour des raisons de simplicité, notre solveur propage toutes les contraintes à chaque itération.

4.1.2 Opérateur de coupe

Notre solveur utilise un opérateur de coupe très simple, consistant à couper selon la variable ayant le plus grand domaine, y compris dans le cas d'éléments abstraits relationnels. Pour des raisons de simplicité, nous ne coupons pas ceux-ci selon des directions non parallèles aux axes, alors que c'est autorisé par l'exemple 3.6. Ceci sera traité dans des travaux futurs, de même que la prise en compte des stratégies sophistiquées de choix de variables utilisées dans les solveurs PPC actuels.

4.2 Résultats

Nous avons testé Absolute sur deux classes de problèmes : la première est constituée de problèmes continus, le but étant de comparer l'efficacité d'Absolute à celle d'un solveur classique en PPC. La deuxième est composée de problèmes mixtes, que les solveurs classiques ne peuvent pas traiter.

4.2.1 Résolution continue

Nous avons utilisé des problèmes du benchmark COCONUT [2] contenant uniquement des variables continues et comparé les résultats obtenus avec Absolute à ceux obtenus avec Ibex [3], un solveur continu basé sur les intervalles. De plus, nous avons comparé Absolute à notre extension d'Ibex pour les octogones [18], ce qui nous permet de comparer le choix du domaine abstrait (intervalles ou octogones) indépendamment du choix du solveur (solveur classique de PPC ou solveur basé sur l'IA). Le tableau 1 donne les temps de calculs en secondes et le nombre de nœuds crées pour trouver toutes les solutions pour chacun des problèmes.

En moyenne, Absolute est compétitif comparé à l'approche de PPC. Plus précisément, il est globalement plus lent sur les problèmes contenant des égalités et plus rapide sur les problèmes contenant des inégalités. Ces différences de performances ne semblent pas être dues au type des contraintes mais plutôt au ratio : nombre de contraintes dans lesquelles une variable apparait sur le nombre total de contraintes. Comme précisé précédemment, à chaque itération, toutes les contraintes sont propagées, même celles dont aucune des variables n'a été modifiée. Ceci augmente le temps de calcul à chaque itération et donc augmente le temps de calcul total. Ceci explique les mauvaises performances d'Absolute sur les problèmes `brent-10` et `nbody5.1`.

De plus, dans Absolute, la boucle de propagation est arrêtée après trois itérations, alors que dans l'approche de PPC la boucle de propagation atteint le point fixe. De ce fait, la consistance dans Absolute peut être moins précise que celle utilisée dans Ibex, ce qui réduit le temps passé lors de la propagation mais peut augmenter la phase d'exploration. Ceci explique le fait que le nombre de nœuds crées par Absolute est souvent plus grand que celui par Ibex.

Ces expériences montrent que notre prototype, bien que n'ayant que très peu des stratégies de PPC, se

2. Disponible à l'adresse `http://www.mat.univie.ac.at/~neum/glopt/coconut/`.

3. Disponible à l'adresse `http://www.emn.fr/z-info/ibex/`.

			$\mathcal{B}^\sharp$				$\mathcal{O}^\sharp$			
nom	# vars	ctr type	Ibex		Absolute		Ibex		Absolute	
b	4	=	*0,02*	551	*0,10*	577	*0,26*	147	*0,14*	1057
nbody5.1	6	=	*96,0*	598 521	*1538,3*	5 536 283	*27,1*	7 925	$\geq 1h$	-
ipp	8	=	*38,8*	237 445	*39,2*	99 179	*279,4*	39 135	*817,9*	2 884 925
brent-10	10	=	*21,6*	211 885	*263,9*	926 587	*330,7*	5 527	$\geq 1h$	-
KP	2	$\leq$	*59,0*	847 643	*23,1*	215 465	*60,8*	520 847	*31,1*	215 465
biggsc4	4	$\leq$	*800,9*	3 824 249	*414,9*	6 038 844	*1772,5*	2 411 741	*688,6*	6 037 260
o32	5	$\leq$	*27,4*	161 549	*22,7*	120 842	*40,7*	84 549	*33,2*	111 194

TABLE 1 – Comparaison du temps CPU, en secondes et du nombre de nœuds crées, pour trouver toutes les solutions, avec l'implantation en Ibex et Absolute.

				First Solution			All solutions			
nom	# vars		ctr type	$\mathcal{B}^\sharp$		$\mathcal{O}^\sharp$	$\mathcal{B}^\sharp$		$\mathcal{O}^\sharp$	
	int	real								
gear4	4	2	=	*0,02*	43	*0,04* 226	*0,02*	67	*0,05*	501
st_miqp5	2	5	$\leq$	*0,67*	2 247	*1,15* 2 247	*2,64*	7 621	*3,64*	7 621
ex1263	72	20	= $\leq$	*8,75*	8544	$\geq 1h$ -	*473,93*	493 417	$\geq 1h$	-
antennes_4_3	6	2	$\leq$	*3,30*	17 625	*22,55* 40 861	*520,77*	2 959 255	*1562,34*	6 657 237

TABLE 2 – Comparaison du temps CPU, en secondes, et du nombre de nœuds crées par Absolute.

comporte raisonnablement bien sur un benchmark classique. De futurs travaux comprendront une analyse plus poussée des performances et bien sûr des améliorations d'Absolute sur ses faiblesses identifiées (stratégie de coupe, boucle de propagation).

4.2.2 Résolution mixte discret-continu

Comme les solveurs en PPC traitent rarement des problèmes mixtes, il n'existe pas de benchmark standard. Nous avons donc rassemblé des problèmes de MinLPLib [4], une bibliothèque de problèmes mixtes d'optimisation issus de la communauté de Recherche Opérationnelle. Ces problèmes ne sont pas des CSP mais des problèmes d'optimisation, c'est-à-dire des problèmes ayant une fonction à minimiser. Nous les avons donc traduits en problèmes de satisfaction en utilisant la même méthode que celle présentée dans [3]. Nous avons remplacé chaque critère d'optimisation $\min f(x)$ par une contrainte $|f(x) - \text{best\_known\_value}| \leq \epsilon$. Nous comparons les résultats obtenus par Absolute au schéma de résolution proposé dans [3], utilisant le même ϵ et les mêmes problèmes, et trouvons des résultats similaires en termes de temps d'exécution (nous ne fournissons pas une comparaison détaillée, ce serait apporter de l'information dénuée de sens en raison des différences de machines). Par manque de place, les résultats obtenus avec les polyèdres n'apparaissent pas ici.

4. Disponible à l'adresse `http://www.gamsworld.org/minlp/minlplib.htm`.

Plus intéressant, nous observons qu'Absolute peut résoudre des problèmes mixtes en un temps raisonnable et se comporte mieux avec les intervalles qu'avec les domaines relationnels. Une raison possible est que les propagations et heuristiques actuelles ne sont pas en mesure d'utiliser pleinement l'information relationnelle disponible avec les octogones ou les polyèdres. De précédents travaux [18] suggèrent qu'un opérateur de coupe soigneusement conçu est la clef pour obtenir une résolution octogonale efficace. De futurs travaux incorporeront les idées développées pour la résolution octogonale en PPC dans notre solveur. Cependant, Absolute est en mesure de faire face naturellement à des problèmes mixtes de PPC en un temps raisonnable, ouvrant la voie à de nouvelles applications de la PPC telles que le problème de restauration et réparation du réseau électrique après une catastrophe naturelle [20], ou des problèmes géométriques [1].

5 Conclusion

Dans cet article nous avons exploré certains liens entre l'IA et la PPC, et les avons utilisés pour concevoir un schéma de résolution en PPC entièrement basé sur les domaines abstraits. Les résultats obtenus avec notre prototype sont encourageants et ouvrent la voie au développement de solveurs PPC–IA permettant de traiter naturellement les problèmes mixtes. Dans de futurs travaux, nous souhaitons améliorer notre solveur en adaptant et intégrant les méthodes avancées

de la littérature en PPC. Les améliorations incluent des opérateurs de coupe pour les domaines abstraits, des propagations spécialisées, et plus important encore, l'amélioration de la boucle de propagation. Nous avons conçu notre solveur sur des abstractions de façon modulaire, ainsi les nouvelles méthodes et les méthodes existantes peuvent être combinées, comme ce qui est fait pour le produit réduit en IA. Finalement, chaque problème sera résolu dans le domaine abstrait qui lui sied le mieux comme c'est la norme en IA.

Un autre développement intéressant est d'utiliser certaines méthodes de la PPC dans un analyseur statique d'IA, telles que l'utilisation d'opérateurs de coupe dans les complétions disjonctives, ainsi que la capacité de la PPC d'affiner un élément abstrait pour atteindre ou approcher la complétude.

Au final, il reste à comprendre comment les opérateurs d'élargissement très utilisés en IA pourraient être utilisés dans des solveurs de PPC.

Références

[1] Nicolas Beldiceanu, Mats Carlsson, Emmanuel Poder, Rida Sadek, and Charlotte Truchet. A generic geometrical constraint kernel in space and time for handling polymorphic k-dimensional objects. In *Proc. of CP'07*, pages 180–194, 2007.

[2] Frédéric Benhamou. Heterogeneous constraint solvings. In *Proc. of ALP'96*, pages 62–76, 1996.

[3] Nicolas Berger and Laurent Granvilliers. Some interval approximation techniques for minlp. In *Proc. of SARA'09*, 2009.

[4] Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace 2010*. American Institute of Aeronautics and Astronautics, 2010.

[5] Gilles Chabert, Luc Jaulin, and Xavier Lorca. A constraint on the number of distinct vectors with application to localization. In *Proc. of CP'09*, pages 196–210, 2009.

[6] Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conf. Rec. of POPL'77*, pages 238–252, 1977.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4) :511–547, August 1992.

[8] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of POPL'78*, pages 84–96, 1978.

[9] Vijay D'Silva, Leopold Haller, and Daniel Kroening. Satisfiability solvers are static analysers. In *Proc. of SAS'12*, 2012.

[10] Philippe Granger. Improving the results of static analyses of programs by local decreasing iterations. In *Proc. of FSTTCS'92*, 1992.

[11] Aymeric Hervieu, Benoit Baudry, and Arnaud Gotlieb. Pacogen : Automatic generation of pairwise test configurations from feature models. In *Proc. of ISSRE 2011*, pages 120–129, 2011.

[12] Bertrand Jeannet and Antoine Miné. Apron : A library of numerical abstract domains for static analysis. In *Proc. of CAV 2009*, volume 5643, pages 661–667, 2009.

[13] Nadjib Lazaar, Arnaud Gotlieb, and Yahia Lebbah. A cp framework for testing cp. *Constraints*, 17(2) :123–147, 2012.

[14] Antoine Miné. *Domaines numériques abstraits faiblement relationnels*. PhD thesis, École Normale Supérieure, December 2004.

[15] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1) :31–100, 2006.

[16] Ugo Montanari. Networks of constraints : Fundamental properties and applications to picture processing. *Information Science*, 7(2) :95–132, 1974.

[17] Marie Pelleau, Antoine Miné, Charlotte Truchet, and Frédéric Benhamou. A constraint solver based on abstract domains. In *Proc. of VMCAI 2013*, 2013.

[18] Marie Pelleau, Charlotte Truchet, and Frédéric Benhamou. Octagonal domains for continuous constraints. In *Proc. of CP'11*, volume 6876, pages 706–720, 2011.

[19] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier, 2006.

[20] Ben Simon, Carleton Coffrin, and Pascal van Hentenryck. Randomized adaptive vehicle decomposition for large-scale power restoration. In *Proc. of CPAIOR'12*, pages 379–394, 2012.

[21] Choco Team. Choco : an open source java constraint programming library. Research report 10-02-INFO, Ecole des Mines de Nantes, 2010.

[22] Aditya Thakur and Thomas Reps. A generalization of stØAlmarck's method. In *Proc. of SAS'12*, 2012.