Domaines de congruence pour la programmation par contraintes

Michel Leconte (*) **Bruno Berstel** (*,†)

* ILOG
9, rue de Verdun
93250 Gentilly
France

† Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken
Allemagne

{leconte,berstel}@ilog.fr

Résumé

Dans un système de Programmation par Contraintes (PPC), une *convergence lente* se produit lorsque la propagation prend un temps proportionnel à la taille des domaines des variables. Ce phénomène survient par exemple sur des équations entières aussi simples que $x = 2y + 1$ et $x = 2z$. Les contraintes qui interviennent dans l'analyse de programmes portent souvent sur des variables dont les domaines s'étendent à tous les entiers représentables en machine. Dans ce cas, la convergence lente empêchera en pratique d'atteindre le point fixe de réduction des domaines.

Pour lutter contre ce phénomène de convergence lente, nous proposons d'utiliser les *domaines de congruence* pour enrichir les domaines de la PPC. Cette idée a été introduite par Ph. Granger [12] dans la communauté de l'interprétation abstraite ; nous montrons ici comment un système de PPC peut en tirer parti, par exemple pour découvrir que $12x+|y| = 3$ et $4x+7y = 0$ n'ont pas de solution entière.

Abstract

A Constraint Programming Solver (CP Solver) encouters *slow convergence* when propagation takes a time that is linear in the size of the domains. Such a phenomenon occurs for instance on as simple integer equations as $x = 2y + 1$ and $x = 2z$. Constraints generated for Program Verification tasks very often involve variables ranging on all the machine-representable integers. Slow convergence then prevents the CP Solver from reaching a fix point in practical time. To address the slow convergence phenomenon, we propose to enrich a CP Solver with *congruence domains*. This idea has been introduced by Ph. Granger [12] in the abstract interpretation community ; we show how a CP Solver can benefit from it, for example in discovering immediately that $12x + |y| = 3$ and $4z + 7y = 0$ have no integer solution.

1 Introduction

Pour certains problèmes, la taille des domaines initiaux des variables est très grande. Il peut être difficile, voire impossible, de les réduire *a priori* lors de l'énoncé des contraintes du problème. C'est notamment le cas des problèmes de vérification de programmes, où on cherche à déterminer si un programme satisfait ou non une certaine propriété. Les domaines des variables d'entrée sont très grands, car typiquement composés de tous les entiers représentables en machine. Les problèmes de vérification de programmes, parce qu'ils s'appuient sur la réfutation, tendent à produire des problèmes de contraintes insatisfaisables. Si le temps pris par un système de programmation par contraintes pour conclure à l'absence de solution est proportionnel à la taille des domaines des variables, il en résulte une grande inefficacité, en particulier sur les programmes sans erreur pour lesquels les contraintes associées sont insatisfaisables (voir l'exemple 3).

Dans cet article nous proposons d'étendre les domaines finis de la Programmation par Contraintes (PPC) avec des domaines de congruence, afin de prouver l'insatisfaisabilité de certaines contraintes entières d'égalité plus efficacement, à savoir en un temps indépendant de la taille des domaines des variables. Les congruences concernent la division d'un entier par un autre et son reste dans la division entière. L'analyse de congruence vient de la communauté de l'interprétation abstraite. Introduite par Granger [12], elle n'est pas restreinte aux équations linéaires mais peut aussi traiter les multiplications. Nous étendons ici son champ d'application à l'expression *si-alors-sinon*, et donc à la valeur absolue, au minimum et au maximum.

Dans la section 2, nous décrivons le phénomène de convergence lente et comment il survient naturellement en vérification de programmes. Les domaines de congruence sont présentés en section 4. La section 5 fournit les formules pour la propagation de ces domaines tandis que la section 6 étudie la coopération des domaines finis et des domaines de congruence. Enfin, la section 7 en présente une utilisation dans un Système de Gestion de Règles Métier.

2 Convergence lente et vérification de programmes

Dans cet article nous utiliserons les notations suivantes (toutes les variables ci-dessous sont à valeurs dans $\mathbb{Z}$, l'ensemble des entiers relatifs).

- Nous noterons $a\mathbb{Z}+b$ l'ensemble $\{az+b \mid z \in \mathbb{Z}\}$.
- Pour $x \in a\mathbb{Z}+b$, nous utiliserons également la notation $x \equiv b\,[a]$.
- Nous utiliserons $a \wedge b$ pour dénoter le plus grand diviseur commun à a et b.
- Nous utiliserons $a \vee b$ pour dénoter le plus petit multiple commun à a et b.

Comme nous l'avons mentionné dans l'introduction, une spécificité des problèmes de contraintes dérivant de la vérification de programmes est que les variables d'entrée sont essentiellement non bornées. Par exemple les variables entières sont supposées pouvoir prendre n'importe quelle valeur des entiers machine en fonction de leur type. Ceci est très différent de l'utilisation traditionnelle de la programmation par contraintes pour résoudre des problèmes combinatoires, où les domaines initiaux des variables sont réduits *a priori* dans la phase de modélisation autant que possible.

Exemple 1 *Prouver que* $2x + 2y = 1$*, où* $x, y \in [-10^d, 10^d]$*, n'a pas de solution.*

Considérons la simple contrainte d'égalité de l'exemple 1, dans lequel les valeurs possibles des variables entières x et y vont de -10^d à 10^d, pour un certain entier d. Il n'y a évidemment pas de solution à cette contrainte, et la réduction « habituelle » des domaines d'intervalles le découvrira en réduisant les domaines de x et y jusqu'à l'ensemble vide. Mais pour atteindre ce point, la réduction aura eu à parcourir tous les 10^d domaines $[-10^d, 10^d]$, puis $[-10^d+1, 10^d-1]$, etc. jusqu'au domaine vide. Nous donnons dans la table 1 le temps pris pour conclure à l'insatisfaisabilité de cette équation, par la propagation des bornes. Ces mesures ont été effectuées sur notre machine avec la bibliothèque ILOG JSolver4Verif de PPC en Java. On constate que ce temps peut atteindre plusieurs minutes.

Valeur de d	Temps de propagation
4	0.12 s
5	0.06 s
6	0.62 s
7	6.16 s
8	61.96 s
9	871.05 s

Tab. 1 – Mesures du temps de propagation pour l'exemple 1.

Insistons sur le fait que cette convergence lente apparaît lors de la propagation des contraintes : le temps pris pour atteindre le point fixe est asymptotiquement proportionnel à la largeur des domaines. Puisque les contraintes sont propagées aussi bien avant que pendant la recherche, la convergence lente peut également survenir lors de la recherche d'une solution. C'est pour lutter contre ces deux cas que nous proposons l'utilisation des domaines de congruence.

Exemple 2 *Résoudre* $2x + 3y + 6z = 2$*, où* $x, y, z \in [-10^d, 10^d]$.

Dans l'exemple 2, nous supposerons que l'ordre d'instanciation des variables est x, y, z et que les valeurs sont énumérées en ordre croissant. Ainsi, la variable x est d'abord instanciée à -10^d. L'équation est alors équivalente à $3y + 6z = 2 + 2.10^d$. Remarquons qu'elle n'a pas de solution puisque son membre gauche est divisible par 3, mais pas son membre droit. Toutefois, la variable y va être instanciée sans succès aux valeurs possibles de son domaine. La variable x est alors instanciée à $-10^d + 1$, ce qui provoque à nouveau le parcours des valeurs possibles du domaine de y. Enfin, l'instanciation $x = -10^d + 2$ permet de trouver une solution. La table 2 indique le temps pris pour engendrer cette solution, en fonction de la valeur de d.

Valeur de d	Temps de résolution
4	0.04 s
5	0.19 s
6	1.88 s
7	18.85 s
8	241.82 s
9	946.57 s

Tab. 2 – Mesures du temps de résolution de l'exemple 2.

Du point de vue de la vérification de programmes, il est malheureusement fréquent de se heurter à la convergence lente de la propagation. Prenons par

exemple un programme contenant la boucle suivante : `tantque (pair(x)) incrémenter x`. Ce programme termine évidemment toujours. Comme l'indique [25], une démonstration consiste à prouver que la conjonction du test de boucle avant et après un tour de boucle est insatisfaisable. Ici l'état du programme est représenté par la valeur de x. Après un tour de boucle, il devient $x+1$. Pour prouver la terminaison de la boucle, il suffit donc de prouver que `pair`(x) et `pair`$(x+1)$ n'a pas de solution. Cela se réduit a $x = 2y$ et $x = 2z + 1$ qui, encore une fois, présente un cas de convergence lente de la propagation.

Un autre exemple de problème de vérification de programmes qui se ramène à la preuve de l'insatisfaisabilité d'égalités entières est celui du non-recouvrement des conditions dans un programme à gardes entières.

Exemple 3 *Le programme suivant exprime, dans un pseudo-langage gardé, une version simple du calcul du nombre de jours dans une année du calendrier Grégorien.*

```
fonction nbDeJours (a : entier) : entier
début
   a === 0 mod 4  ->  366  |
   a === 1 mod 4  ->  365  |
   a === 2 mod 4  ->  365  |
   a === 3 mod 4  ->  365
fin
```

La question est : les gardes de ce programme se recouvrent-elles ?

Bien sûr, les gardes ne se recouvrent pas et le programme est correct à cet égard. Pour le démontrer, on commencera par traduire les gardes en contraintes, ce qui donnera les quatre contraintes $x = 4y_i + i$, pour $i = 0, 1, 2, 3$. Dans ces contraintes, x et les y_i sont des variables entières à valeur dans $[-d, d]$ pour un certain entier d (potentiellement $2^{31} - 1$ pour un entier codé sur 32 bits). Puis, on prouvera que pour deux i et j distincts entre 0 et 3, la conjonction de $x = 4y_i + i$ et $x = 4y_j + j$ est insatisfaisable. Comme nous l'avons déjà vu, la réduction des domaines finis le découvrira mais aura besoin de l'ordre de d réductions pour cela.

Le reste de cet article explique comment les domaines de congruence permettent à un solveur de programmation par contraintes de le démontrer en un nombre de pas indépendant de la taille des domaines.

3 Liens avec d'autres travaux

L'analyse de congruence a été introduite par Philippe Granger [12, 13], pour être appliquée à la vectorisation automatique. Aujourd'hui l'analyse de congruence est une technique importante, en particulier pour la vérification des propriétés d'alignement des pointeurs [5, 28]. Dans cet article, nous étendons le champ d'application au-delà des opérateurs arithmétiques $+$, $-$, $\times$, $\div$, à l'expression *si-alors-sinon*, et donc à la valeur absolue, au minimum et au maximum.

Les domaines de congruence ont été étendus [23, 2] aux contraintes de la forme $x - y \equiv b\,[c]$. Toman *et al.* ont proposé dans [27] une procédure de normalisation de telles contraintes en $O(n^4)$. Miné a amélioré cette complexité en $O(n^3)$ dans [23]. Les grilles (*grids*) [4, 24] sont une autre généralisation des domaines de congruence *relationnels* en présence d'équations de la forme $\sum a_i x_i \equiv b\,[c]$, alors que nous considérons ici les domaines plus spécifiques non-relationnels $x \equiv b\,[c]$. Granger [14] propose une extension de l'analyse de congruence en considérant des ensembles de nombres rationnels de la forme $p\mathbb{Z} + q$, avec $p, q \in \mathbb{Q}$.

Par ailleurs, la convergence lente de la propagation sur les nombres réels a motivé le développement de techniques spéciales de surréduction [22]. Elles ne s'appliquent pas lorsque les variables sont entières.

Dans cet article, nous étendons une bibliothèque de programmation par contraintes à domaines finis, ILOG JSolver4Verif, avec des domaines de congruence pour enrichir les domaines finis. Très peu de systèmes de programmation par contraintes utilisent des raisonnements basés sur les congruences. Une exception notable est le système Alice [20] et son successeur Rabbit [21] qui mettent en œuvre des capacités de raisonnement de congruence dans le cadre des simplifications formelles de contraintes.

Comme nous le verrons, les domaines finis et les domaines de congruence coopèrent naturellement [12, 5, 28]. Les domaines numériques abstraits, tels que les intervalles, sont disponibles dans les systèmes d'analyse statique comme Astrée [8] ou la PPL [3, 26], qui fournit aussi les domaines de congruence.

Enfin, une approche toute différente pour éviter la convergence lente de contraintes telles que $x = 2y + 1$ et $x = 2z$ serait d'utiliser un solveur de contraintes linéaires, par exemple basé sur l'élimination de Gauss, comme une contrainte globale supplémentaire.

4 Raisonnement par congruence

Nous considérons ici toutes les contraintes d'égalité entre des variables ou expressions entières. Les expressions sont construites à partir des opérateurs arithmétiques $+$, $-$, $\times$, $\div$ usuels, ainsi qu'à partir de la valeur absolue et des opérateurs de minimum et de maximum. Nous considèrerons aussi des expressions *si-alors-sinon*. Elles correspondent à l'operateur `?:` de certains langages de programmation comme C, Java ou C#. Une expression *si-alors-sinon* de la forme

$\mathsf{si}(c, e_1, e_2)$, où c est une contrainte et les e_i sont des expressions entières, a pour valeur e_1 si la contrainte c est vraie et e_2 sinon.

Quoique l'ensemble des contraintes entières soit couvert par l'analyse de congruence, celle-ci ne constitue pas une procédure de décision en elle-même. Ainsi, l'analyse de congruence ne détectera pas dans tous les cas l'insatisfaisabilité d'un ensemble de contraintes entières d'égalité. Ceci n'est pas gênant puisque notre objectif est avant tout de renforcer la propagation des contraintes.

4.1 Utilisation des congruences sur les domaines d'intervalles

Exemple 4 *Déterminer si l'équation* $2x+4y+6z = 1$ *a des solutions entières.*

Dans l'exemple 4 ci-dessus, la propagation de la contrainte sur les domaines d'intervalles ne produit aucune réduction des bornes de ces domaines. En particulier, la réduction des bornes ne détectera pas l'insatisfaisabilité de la contrainte. Pourtant, un simple raisonnement sur les congruences montre que $2x+4y+6z$ est pair, et ne pourra donc jamais être égal à 1 qui est impair. C'est pour le moins l'illustration d'une propagation manquante.

Rappelons-nous qu'une contrainte $\sum a_i x_i = c$ admet une solution si et seulement si le plus grand diviseur commun des coefficients, soit $\bigwedge_i a_i$, divise la constante c.

Cette propriété nous donne directement un propagateur pour les équations linéaires. Nous calculons d'abord le plus grand diviseur commun des coefficients des variables non instanciées, et nous vérifions qu'il divise la constante c moins la somme des $a_i x_i$ pour les variables x_i instanciées.

Cette utilisation d'une propriété de congruence, bien que **passive**, est déjà très utile. Elle permet de prouver lors de la propagation initiale que les exemples 1, 3 et 4 précédents n'ont pas de solution, et ce en un temps indépendant de la taille des domaines puisque seul un calcul de P.G.C.D. est effectué. Utilisée a chaque instanciation d'une variable, elle permet de trouver une solution à l'exemple 2 sans énumérer les valeurs de y. Plus précisément, les instanciations de x à -10^d puis à $-10^d + 1$ déclencheront immédiatement un échec. Notons enfin que cette utilisation passive n'ajoute que très peu au temps de propagation, puisque les P.G.C.D. ne sont calculés que lors de la propagation initiale, puis seulement à chaque instanciation d'une variable.

4.2 Représentation des congruences par des domaines propres

Exemple 5 *Trouver toutes les solutions entières du système d'équations* $2x + 4y + 3z = 1$ *et* $z = 2t + 12$.

L'utilisation passive de l'information de congruence telle que nous l'avons juste décrite ne permet pas de déduire que z est impair au vu de $2x + 4y + 3z = 1$. La vacuité de l'ensemble des solutions du système de l'exemple ci-dessus ne sera donc pas détectée. Toutefois, ce serait une fort mauvaise idée que d'utiliser activement l'information de congruence sans précaution.

Imaginons par exemple que la contrainte de congruence, non seulement vérifie que le P.G.C.D. divise la constante, mais aussi *réduise* les bornes des domaines des variables pour éliminer les valeurs qui conduiraient à un échec. Disons par exemple que la propagation de la contrainte $2x + 4y + 3z = 1$ ajuste les bornes de z de telle façon que ces bornes soient impaires, et que la contrainte $z = 2t + 12$ ajuste les bornes de z à des valeurs paires. Si cette propagation est utilisée dans l'exemple 5, un domaine vide sera effectivement trouvé pour z puisque les contraintes réduiront successivement les bornes du domaine de z à des valeurs paires et impaires. C'est exact, mais c'est encore une occurrence de la convergence lente de la propagation puisque les bornes de z seront modifiées d'une unité à chaque fois.

Pour éliminer cette convergence lente de la propagation, l'information de congruence déduite par les contraintes doit être partagée par ces mêmes contraintes. Il s'agit donc d'équiper les variables d'une information de congruence, plutôt que de fondre cette information dans la valeur des bornes de ces variables. Cela nous conduit tout naturellement, de la même façon que nous associons un domaine fini — i.e. un ensemble de points — à chaque variable entière, à associer à chacune d'elles un domaine de congruence. Ce domaine de congruence est une paire d'entiers (a, b) qui représente l'ensemble $a\mathbb{Z} + b$.

Ainsi nous pourrons calculer un domaine de congruence pour chaque expression à partir des domaines de congruence de ses sous-expressions, ainsi que nous le verrons dans la prochaine section. Comme avec les intervalles, les contraintes d'égalité propageront les domaines de congruence calculés sur les expressions pour finir par réduire les domaines de congruence des variables elles-mêmes.

Ainsi l'insatisfaisabilité des deux contraintes $x = 2y$ et $x = 2z + 1$ est trouvée par ce raisonnement de congruence en déduisant que x est à la fois pair et impair. Ce raisonnement, au contraire de la réduction des bornes du domaine intervalle de x, ne demande qu'un temps indépendant de la taille de cet intervalle.

La complexité est en fait celle du calcul du plus grand diviseur commun, et dépend logarithmiquement de la plus grande des valeurs absolues des coefficients, qui sont eux-mêmes bornés par la taille du plus grand entier représentable.

Cette utilisation **active** de l'information de congruence subsume complètement l'utilisation passive, qui n'effectue que la vérification des tests de divisibilité.

5 Propagation des domaines de congruence

5.1 Propagation sur les expressions

Chaque variable entière a un domaine de congruence, noté $a\mathbb{Z}+b$, qui représente l'ensemble des valeurs possibles de cette variable. Nous utiliserons $0\mathbb{Z}+b$ pour représenter la constante b et $1\mathbb{Z}+0$ comme domaine d'une variable pouvant prendre toutes les valeurs entières possibles ($\mathbb{Z}$).

Indiquons maintenant comment calculer les domaines de congruence des expressions à partir de leurs sous-expressions. Nous ne donnerons ici que les formules pour l'addition et la multiplication. Ces formules, ainsi que celles pour la soustraction et la division, peuvent être trouvées dans [12]. Soient $x \in a\mathbb{Z}+b$ et $y \in a'\mathbb{Z}+b'$, on a :

$$x + y \in (a \wedge a')\mathbb{Z} + (b + b') \quad (1)$$

$$x \times y \in (aa' \wedge a'b \wedge ab')\mathbb{Z} + bb' \quad (2)$$

Considérons maintenant une expression d'appartenance, qui dériverait de contraintes de la forme $z \in \{x, y\}$. Soient $x \in a\mathbb{Z}+b$ et $y \in a'\mathbb{Z}+b'$, on a :

$$\text{si } z \in \{x, y\} \text{ alors } z \in (a \wedge a' \wedge |b - b'|)\mathbb{Z} + b \quad (3)$$

L'asymétrie dans cette formule entre b et b' n'est qu'apparente. En effet, si α est le plus grand diviseur commun de a, a' et $|b-b'|$ dans (3), il divise en particulier $|b-b'|$, donc $b \equiv b'\ [\alpha]$.

Par ailleurs, notons bien que dans l'expression d'appartenance, comme pour les domaines d'intervalles, on ne réalise pas l'union des domaines de congruence de x et de y, mais on retient le plus petit domaine de congruence contenant à la fois les domaines de x et de y.

De la formule pour l'appartenance, on déduit aisément la formule d'une expression *si-alors-sinon*. Par définition, $\mathsf{si}(c, e_1, e_2)$ est une expression qui vaut e_1 si c est vraie et qui vaut e_2 si c est fausse. Donc, si la contrainte c est fixée à vrai (resp. faux), alors le domaine de congruence du *si-alors-sinon* est le domaine de congruence de e_1 (resp. e_2). Si toutefois la valeur de vérité de la contrainte n'est pas fixée, alors l'expression a un domaine de congruence qui est celui de son appartenance à $\{e_1, e_2\}$. Ainsi, l'expression d'appartenance est une sur-approximation de l'expression *si-alors-sinon*.

Soient $x \in a\mathbb{Z}+b$, $y \in a'\mathbb{Z}+b'$, et une contrainte c :

$$\mathsf{si}(c, x, y) \in \begin{cases} a\mathbb{Z}+b & \text{si } c \text{ est fixée à vrai} \\ a'\mathbb{Z}+b' & \text{si } c \text{ est fixée à faux} \\ (a \wedge a' \wedge |b-b'|)\mathbb{Z}+b & \text{sinon} \end{cases} \quad (4)$$

Cela nous donne aussi directement les formules du minimum (et du maximum) puisque $\min(x, y)$ est équivalent à $\mathsf{si}(x < y, x, y)$, ainsi que de la valeur absolue puisque $|x| = \mathsf{si}(x < 0, -x, x)$.

5.2 Propagation des contraintes d'égalité

Nous explicitons maintenant comment les contraintes d'égalité vont propager les domaines de congruence à leurs membres. Comme d'habitude pour l'égalité, le domaine commun de ces membres sera l'intersection de leurs domaines initiaux.

Soient $x \in a\mathbb{Z}+b$ et $y \in a'\mathbb{Z}+b'$:

$$\text{si } x = y \text{ alors } x \in \begin{cases} (a \vee a')\mathbb{Z}+b'' & \\ \qquad \text{si } (a \wedge a') \text{ divise } (b-b') & \\ \emptyset \quad \text{sinon} & \end{cases}$$

Le nombre b'' peut être calculé de la façon suivante. Soient $x = au+b$ et $y = a'v+b'$, l'équation $x = y$ donne $au+b = a'v+b'$, ou encore $au-a'v = b'-b$. Comme $a \wedge a'$ divise $b-b'$, l'équation peut être simplifiée par $a \wedge a'$ en $\alpha u - \alpha' v = \beta$. Comme α et α' sont premiers entre eux, le lemme de Bachet-Bezout indique qu'il existe u_0 et v_0 tels que $\alpha u_0 + \alpha' v_0 = 1$. Ces nombres peuvent être calculés par l'algorithme étendu d'Euclide [19]. La combinaison des deux dernières équations donne $\alpha(u-\beta u_0) = \alpha'(v+\beta v_0)$. Comme α et α' sont premiers entre eux, $u - \beta u_0$ est un multiple de α', c'est-à-dire $u \in \alpha'\mathbb{Z} + \beta u_0$. Finalement de $x = au+b$ on tire $x \in a\alpha'\mathbb{Z} + b''$, où $b'' = b + a\beta u_0$.

Quand la contrainte d'égalité porte sur des expressions (par opposition à des variables), les domaines de congruence de ces expressions sont d'abord calculés à la demande de la contrainte, puis l'intersection de ces domaines est elle-même calculée suivant la formule ci-dessus. Le domaine résultant est alors propagé sur les sous-expressions des membres de l'égalité, jusqu'aux variables.

Exemple 6 *Trouver toutes les solutions entières de* $4x = 3|y| + 2$.

Illustrons ce mécanisme sur l'exemple ci-dessus. En l'absence d'information supplémentaire, nous partons

avec $x, y \in 1\mathbb{Z} + 0$. Les formules pour la multiplication (2), l'addition (1) et la valeur absolue (4) nous donnent $4x \in 4\mathbb{Z} + 0$ et $3|y| + 2 \in 3\mathbb{Z} + 2$.

La contrainte d'égalité déduit maintenant que ces expressions appartiennent toutes les deux à $12\mathbb{Z} + 8$. Puisque $4x \in 12\mathbb{Z} + 8$, on a $x \in 3\mathbb{Z} + 2$. De la même façon, puisque $3|y| + 2 \in 12\mathbb{Z} + 8$, on a $|y| \in 4\mathbb{Z} + 2$. La valeur absolue se décompose en deux cas : soit $y \in 4\mathbb{Z} + 2$, soit $-y \in 4\mathbb{Z} + 2$. Remarquons que ce dernier cas se ramène à $y \in -4\mathbb{Z} - 2$, c'est-à-dire $y \in 4\mathbb{Z} - 2$ qui est le même ensemble que $4\mathbb{Z} + 2$. Donc dans tous les cas $y \in 4\mathbb{Z} + 2$. Les domaines ne peuvent plus être réduits, le point fixe est atteint.

5.3 Propagation du si-alors-sinon

Enfin, pour une expression *si-alors-sinon*, on peut déduire la valeur de vérité de la contrainte de choix, c'est-à-dire réduire le domaine de cette contrainte depuis sa valeur initiale $\{\mathsf{vrai}, \mathsf{faux}\}$.

Plus précisément, soient $x \in a\mathbb{Z} + b$, $y \in a'\mathbb{Z} + b'$, une contrainte c et $z = \mathsf{si}(c, x, y) \in a''\mathbb{Z} + b''$, on a :

$$\begin{array}{ll} \text{si } a\mathbb{Z} + b \cap a''\mathbb{Z} + b'' = \emptyset & \text{alors } c = \mathsf{faux} \\ \text{si } a'\mathbb{Z} + b' \cap a''\mathbb{Z} + b'' = \emptyset & \text{alors } c = \mathsf{vrai} \end{array} \quad (5)$$

Remarquons que la réduction du domaine de la contrainte c à $\{\mathsf{vrai}\}$ (resp. $\{\mathsf{faux}\}$) permet non seulement de propager $z = x$ (resp. $z = y$), mais aussi de propager la valeur de vérité de la contrainte c elle-même (resp. de sa négation), ce qui pourra encore réduire les domaines des variables.

6 Coopération des domaines de congruence et d'intervalles

L'idée ici est de rassembler les deux notions et de considérer des domaines de la forme $a\mathbb{Z} + b \cap [min, max]$. De tels domaines sont appelés Reduced Interval Congruence (RIC) dans [5, 28]. En combinant les deux notions, l'information venant des domaines finis peut être utilisée par les domaines de congruence et vice versa.

Commençons par examiner comment les domaines de congruence exploitent les domaines finis. Dès que le domaine d'une variable est instanciée à une valeur, disons $x = b$, cela se traduit pour le domaine de congruence par $x \in 0\mathbb{Z} + b$. Plus généralement, si l'on trouve que $x \in \{b_i\}$ pour des constantes b_i, alors $x \in (\bigwedge_{i \neq 0} |b_i - b_0|)\mathbb{Z} + b_0$.

Réciproquement, les congruences sont utilisés par les domaines finis, en exploitant le fait que les bornes du domaine d'une variable doivent appartenir au même domaine de congruence que la variable elle-même. Ainsi, si $x \in [min, max]$ et $x \in a\mathbb{Z} + b$, alors min et max doivent être ajustés pour tomber dans $a\mathbb{Z} + b$. Si $a \neq 0$, alors min devient $a\lceil (min - b)/a \rceil + b$ tandis que le max devient $a\lfloor (max - b)/a \rfloor + b$, où $\lceil x \rceil$ et $\lfloor x \rfloor$ dénotent respectivement les parties entières supérieure et inférieure du nombre x.

Si nous reprenons l'exemple 1, le domaine de x passera lors de la propagation initiale de $[-10^d, 10^d]$ à $[-10^d + 2, 10^d]$ puisque de $2x + 3y + 6z = 2$ on tire que $x \in 3\mathbb{Z} + 1$. Lors de la recherche, le premier essai $x = -10^d + 2$ conduira alors à une solution sans aucun échec.

De plus, si le diamètre $max - min$ est plus petit que a, le domaine de la variable sera soit un singleton soit vide. Par exemple, si le domaine d'une variable se réduit à $a\mathbb{Z} + b \cap [0, a - 1]$ avec $0 \leq b < a$, alors la variable est directement instanciée à b, qui est la seule valeur contenue dans ce domaine. Similairement, si le domaine intervalle a été réduit à $[0, |b| - 1]$, alors la contrainte déclenche un échec puisque le domaine $a\mathbb{Z} + b \cap [0, |b| - 1]$ est vide.

Terminons cette section avec un exemple de réductions non triviales.

Exemple 7 *Considérons les deux contraintes* $4x = 3y + 2$ *et* $|x| - 12z = 2$.

Nous avons vu dans l'exemple 6 que la première de ces contraintes réduisait les domaines des variables à $x \in 3\mathbb{Z} + 2$ et $y \in 4\mathbb{Z} + 2$. De la seconde contrainte, nous déduisons que $|x| \in 12\mathbb{Z} + 2$.

Nous allons utiliser la formule du *si-alors-sinon* (5) avec $|x| = \mathsf{si}(x < 0, -x, x)$ pour propager $|x| \in 12\mathbb{Z}+2$. Puisque $x \in 3\mathbb{Z} + 2$, nous notons d'abord que $-x \in 3\mathbb{Z}+1$. Nous remarquons ensuite que $12\mathbb{Z}+2 \cap 3\mathbb{Z}+1 = \emptyset$ ce qui nous permet d'inférer que la contrainte $x < 0$ est fausse, donc que sa négation $x \geq 0$ est vraie. Ainsi $|x| = x$.

Puisque $|x| \in 12\mathbb{Z}+2$, nous avons $x \in 12\mathbb{Z}+2$. Ajustant alors la borne minimale (0) de x, nous obtenons $x \geq 2$. Propageant alors $|x| \geq 2$ (car $|x| = x$) dans la seconde contrainte, nous avons $z \geq 0$, Pour finir, nous propageons $x \geq 2$ dans $4x = 3y + 2$ pour en déduire que $y \geq 2$.

Finalement, nous atteignons le point fixe avec :

$$\begin{array}{rcl} x \in 12\mathbb{Z} + 2 & \text{et} & x \geq 2 \\ y \in 4\mathbb{Z} + 2 & \text{et} & y \geq 2 \\ & & z \geq 0 \end{array}$$

7 Utilisation industrielle

Les règles de production (ou règles condition-action) [1, 9, 11, 6], au travers de produits nommés Systèmes de Gestion de Règles Métier (SGRM) [17,

18], suscitent un intérêt croissant dans l'industrie. Les règles sont utilisées pour isoler des applications logicielles, la part de leur comportement susceptible de changer rapidement ou fréquemment, en réaction par exemple à des modifications réglementaires ou à la pression concurrentielle.

Dans ce contexte, la vérification de programmes est une fonction importante d'un SGRM [15]. En effet, on compte parmi les utilisateurs des SGRM des experts métiers, souvent non-informaticiens, qui attendent de l'outil qu'il l'aide à maîtriser des bases de règles en croissance rapide. Cette aide passe aussi par la vérification de propriétés du programme, et par des outils de navigation dans le code qui prennent en compte la sémantique du programme.

Les problèmes de vérification de programmes, c'est-à-dire la question de savoir si un programme vérifie une propriété donnée, peuvent être formulés comme des problèmes de satisfaisabilité ou d'insatisfaisabilité. L'utilisation d'un système de programmation par contraintes pour résoudre ces problèmes présente l'avantage de traiter une large classe de formules. Cet avantage a un prix, à savoir la complétude, mais l'expérience montre que cette utilisation donne la plupart du temps des résutats concluants [7]. Les solutions trouvées par le système de PPC correspondent à des réponses (témoins ou contre-exemples) aux questions de vérification de programmes.

Par exemple, une des fonctions de vérification de programmes à base de règles que nous proposons, consiste à détecter les règles qui ne seront jamais applicables, c'est-à-dire dont la partie condition est toujours insatisfaisable. Ainsi, lorsque l'utilisateur utilisera l'environnement de développement Eclipse [10] pour écrire ses règles, une règle qui n'est jamais applicable sera immédiatement signalée, et les tests de la partie condition qui la rendent insatisfaisable seront soulignés.

Nous avons mis en œuvre les domaines de congruence dans ILOG JSolver4Verif, une bibliothèque de programmation par contraintes à domaines finis dérivée de ILOG JSolver [16]. Cette bibliothèque est utilisée pour l'analyse de règles dans le produit ILOG JRules.

8 Conclusion

Nous disons que la propagation de contraintes souffre de *convergence lente* lorsque le temps pris pour atteindre un point fixe (ou pour détecter un échec) est proportionnel à la taille initiale des domaines (finis) des variables. Nous mettons notamment en évidence ce phénomène dans le cadre de la vérification de programmes, en particulier sur des programmes corrects (vis-à-vis d'une propriété donnée).

Pour éviter la convergence lente pour les équations entières, nous proposons d'ajouter aux systèmes de propagation par contraintes des capacités de raisonnement sur les relations de congruence. Nous avons repris l'idée d'enrichir les domaines finis avec des domaines de congruence des travaux de la communauté de l'interprétation abstraite [12].

Nous avons montré à l'aide d'exemples comment un système de propagation par contraintes peut benéficier de la coopération de ces domaines. Nous avons detaillé les propagations multiples entre les domaines finis et les domaines de congruence des variables entières ainsi que les domaines booléens des contraintes.

Références

[1] Herbert A. Simon Allen Newell. *Human problem solving*. Prentice Hall, Englewood Cliffs, NJ, USA, 1972.

[2] R. Bagnara. *Data-Flow Analysis for Constraint Logic-Based Languages*. PhD thesis, Dipartimento di Informatica, Università di Pisa, March 1997.

[3] R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In M. V. Hermenegildo and G. Puebla, editors, *Static Analysis : Proceedings of the 9th International Symposium*, volume 2477 of *LNCS*, pages 213–229. Springer-Verlag, Berlin, 2002.

[4] Roberto Bagnara, Katy Dobson, Patricia M. Hill, Matthew Mundell, and Enea Zafanella. Grids : A domain for analyzing the distribution of numerical values. In *LOPSTR*, 2006.

[5] Gogul Balakrishnan and Thomas W. Reps. Analyzing memory accesses in x86 executables. In Evelyn Duesterwald, editor, *CC*, volume 2985 of *LNCS*, pages 5–23. Springer, 2004.

[6] Elena Baralis and Jennifer Widom. An algebraic approach to static analysis of active database rules. *ACM Trans. Database Syst.*, 25(3) :269–332, 2000.

[7] Hélène Collavizza and Michel Rueher. Exploration of the capabilities of constraint programming for software verification. In Holger Hermanns and Jens Palsberg, editors, *TACAS*, volume 3920 of *LNCS*, pages 182–196. Springer, 2006.

[8] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The Astrée analyzer. In *ESOP'05*, 2005.

[9] Randall Davis, Bruce G. Buchanan, and Edward H. Shortliffe. Production rules as a repre-

sentation for a knowledge-based consultation program. *Artif. Intell.*, 8(1) :15–45, 1977.

[10] The Eclipse Consortium. ECLIPSE 3.0, 2005. `http://www.eclipse.org`.

[11] Charles Forgy. Rete : A fast algorithm for the many patterns/many objects match problem. *Artif. Intell.*, 19(1) :17–37, 1982.

[12] Philippe Granger. Static analysis of arithmetic congruences. *International Journal of Computer Math*, pages 165–199, 1989.

[13] Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In Samson Abramsky and T. S. E. Maibaum, editors, *TAPSOFT, Vol.1*, volume 493 of *LNCS*, pages 169–192. Springer, 1991.

[14] Philippe Granger. Static analyses of congruence properties on rational numbers (extended abstract). In Pascal Van Hentenryck, editor, *SAS*, volume 1302 of *LNCS*, pages 278–292. Springer, 1997.

[15] Stephen D. Hendrick. *Business Rule Management Systems : Addressing Referential Rule Integrity.* IDC, 2006. `http://www.idc.com/getdoc.jsp?containerId=201262`.

[16] ILOG. ILOG JSOLVER, 2000. `http://www.ilog.com`.

[17] ILOG. ILOG JRULES, 2006. `http://www.ilog.com`.

[18] JBoss. DROOLS, 2006. `http://www.drools.org`.

[19] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley, Reading, Massachusetts, second edition, 1981.

[20] Jean-Louis Laurière. A language and a program for stating and solving combinatorial problems. *Artif. Intell.*, 10(1) :29–127, 1978.

[21] Jean-Louis Laurière. Programmation de contraintes ou programmation automatique. Technical report, L.I.T.P., 1996. `http://www.lri.fr/~sebag/Slides/Lauriere/Rabbit.pdf`.

[22] Olivier Lhomme, Arnaud Gotlieb, and Michel Rueher. Dynamic optimization of interval narrowing algorithms. *J. Log. Program.*, 37(1-3) :165–183, 1998.

[23] Antoine Miné. A few graph-based relational numerical abstract domains. In Manuel V. Hermenegildo and Germán Puebla, editors, *SAS*, volume 2477 of *LNCS*, pages 117–132. Springer, 2002.

[24] Markus Müller-Olm and Helmut Seidl. A generic framework for interprocedural analysis of numerical properties. In Chris Hankin and Igor Siveroni, editors, *SAS*, volume 3672 of *LNCS*, pages 235–250. Springer, 2005.

[25] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In Bernhard Steffen and Giorgio Levi, editors, *VMCAI*, volume 2937 of *LNCS*, pages 239–251. Springer, 2004.

[26] PPL. *Parma Polyhedra Library*, 2006. `http://www.cs.unipr.it/ppl`.

[27] David Toman, Jan Chomicki, and David S. Rogers. Datalog with integer periodicity constraints. In *SLP*, pages 189–203, 1994.

[28] Michael Venable, Mohamed R. Chouchane, Md. Enamul Karim, and Arun Lakhotia. Analyzing memory accesses in obfuscated x86 executables. In Klaus Julisch and Christopher Krügel, editors, *DIMVA*, volume 3548 of *LNCS*, pages 1–18. Springer, 2005.