Stratégies de recherche pour les systèmes de contraintes sur les flottants *

Heytem Zitoun[1] †**Claude Michel**[1] **Michel Rueher**[1] **Laurent Michel**[2]

[1] Université Côte d'Azur, CNRS, I3S, France
[2] University of Connecticut, Storrs, CT 06269-2155
prenom.nom@unice.fr ldm@engr.uconn.edu

Résumé

La vérification logicielle est un enjeu clé pour les applications critiques telles que l'aviation, l'aérospatiale, ou les systèmes embarqués. Les approches basées sur le « bounded model checking (BMC) » et la programmation par contraintes (CBMC, CBPV, ...) recherchent des contre-exemples qui violent une propriété du programme. La recherche de tels contre-exemples peut-être particulièrement longue et couteuse lorsque le programme à vérifier contient des opérations sur les flottants. En effet, les stratégies de recherche existantes en programmation par contraintes ont été conçues pour les domaines finis et, dans une moindre mesure, les domaines continus. Dans cet article, nous proposons de nouvelles stratégies de recherches adaptées aux spécificités des flottants. Ces stratégies se basent sur des propriétés propres aux domaines des variables (e.g., densité des domaines) et aux contraintes sur les flottants (e.g., arithmétique propre aux flottants et structure même des problèmes). Les stratégies que nous introduisons exploitent ces propriétés et les combinent en utilisant, par exemple, la notion de score introduite dans la stratégie « Impact Based Search ». Pour évaluer ces nouvelles stratégies, nous avons porté le solveur sur les flottants FPCS dans Objective-CP afin de pouvoir bénéficier des facilités offertes par cet environnement.

1 Introduction

Un point important lors de la vérification de programme contenant des calculs sur les flottants est la recherche des erreurs de calcul dues à l'arithmétique spécifique des flottants qui peuvent entrainer un résultat sensiblement différent du résultat attendu sur les réels. Soit le programme foo suivant :

```
void foo(){
        float a = 1e8f;
        float b = 1.0f;
        float c = -1e8f;
        float r = a + b + c;
        if(r >= 1.0f){
                doThenPart();
        }else{
                doElsePart();
        }
}
```

En interprétant le programme foo sur les réels, l'instruction doThenPart devrait être exécutée. Cependant, un problème d'absorption sur les flottants (la valeur 1 est absorbée par 1e8f[1]) conduit le programme à passer par la branche Else.

Lors de la vérification de programmes avec des techniques de bounded model checking il faut aussi s'assurer qu'une propriété ne peut pas être violée du fait des erreurs de calcul sur les flottants. La programmation par contraintes [4, 5] a été utilisée pour traiter de tels problèmes, mais la recherche de contre-exemple reste longue et coûteuse. Les stratégies de recherche standards sont en effet peu efficaces pour trouver de tels contre-exemples. De nombreuses stratégies de recherche ont été proposées sur les entiers [3, 6, 12, 14, 15] et, dans une moindre mesure, sur les réels [10, 9]. Or, les stratégies propres aux entiers et aux réels sont mal adaptées aux flottants. Le sous-ensemble des entiers d'un domaine borné par deux entiers est fini et réparti de manière uniforme ; il est donc possible d'en énumérer toutes les valeurs lorsque

*Ces travaux ont été partiellement supportés par l'ANR COVERIF (ANR-15-CE25-0002).

†Papier doctorant : Heytem Zitoun[1] est auteur principal.

1. sur les flottants simple précision et avec un arrondi au plus près.

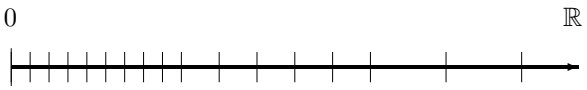

FIGURE 1 – Répartition des flottants

le domaine est suffisamment petit. Le sous-ensemble des réels borné par deux flottants étant infini et uniforme, il n'est pas énumérable. Aussi les stratégies de recherche adaptées au continu utilisent des techniques sur les intervalles telles que la bissection ainsi que des propriétés mathématiques pour prouver l'existence où l'absence de solution dans un petit intervalle. A contrario, l'ensemble des flottants est lui fini, mais sa cardinalité est très élevée et la répartition des flottants n'est pas du tout uniforme (la moitié des flottants sont dans le domaine [-1,1]). Les techniques précédentes, comme l'énumération, sont donc inenvisageables dans le cas général. Les flottants correspondent à une approximation des réels, mais n'hérite pas des mêmes propriétés mathématiques, telles que la continuité. Il est donc difficile de tirer profit des stratégies de recherches sur les réels comme celles qui exploite la continuité.

Dans cet article, nous présentons de nouvelles stratégies de recherches adaptées aux spécificités des nombres flottants pour résoudre plus efficacement les problèmes de vérification. Ces stratégies sont en cours d'évaluation sur un ensemble de benchmarks. Pour cela, nous avons porté le solveur FPCS [2] (Floating Point Constraint Solver) [13] développé par Claude Michel dans Objective-CP. Objective-CP est un outil d'optimisation introduit dans [17]. Ce portage nous permet de bénéficier de l'ensemble des facilités (e.g continuations, contrôleurs) offertes par Objective-CP pour implanter ces stratégies de recherche.

L'article est organisé comme suit : la section suivante présente quelques notations et définitions nécessaires à la compréhension de ce document. La section 3 explicite plusieurs propriétés basées sur les domaines des variables du système, puis des propriétés définies sur les contraintes. Ces propriétés servent de base aux nouvelles stratégies de recherches explicitées dans la section 4. La section 5 présente les choix liés à l'implantation des stratégies de recherche et enfin la section 6 discute des travaux en cours et des perspectives.

2. `http://www.i3s.unice.fr/~cpjm/misc/fpcs.html`

2 Notations et définitions

2.1 Les nombres à virgule flottante

Les nombres à virgule flottante ont été introduits pour approximer les nombres réels dans un ordinateur. La norme IEEE 754 [8] spécifie la représentation, ainsi qu'un ensemble de propriétés arithmétiques spécifiques aux flottants. Les deux principaux formats de représentations des flottants introduits dans IEEE 754 sont les formats *simple* et *double*. Selon le format utilisé, le nombre de bits de représentation est de 32 bits pour les flottants à simple précision, et de 64 bits pour les flottants à double précision. Un nombre à virgule flottante est composé de trois parties : le **signe** s (0 ou 1), la **mantisse** m et **l'exposant** e [7]. Un flottant normalisé est représenté par :

$$(-1)^s 1.m \times 2^e$$

Pour représenter les nombres flottants ayant une valeur absolue très petite, la norme IEEE 754 introduit la notion de nombre dénormalisé. Un flottant dénormalisé est représenté par :

$$(-1)^s 0.m \times 2^0$$

La **mantisse** m est représentée par 23 bits pour les flottants à simple précision, et par 52 bits pour le format double précision. Dans la suite de l'article, la **taille** de mantisse sera notée p.
L'exposant e est représenté par 8 bits pour le format simple (11 bits pour le format double).

2.2 Absorption

L'absorption est un phénomène qui apparait lors de l'addition de deux flottants d'ordres de grandeur très différents. Le résultat de l'addition est alors le flottant le plus grand.

Par exemple, en C, sous unix avec le format simple précision et un arrondi au plus près :

$$10^8 + 1.0 = 10^8$$

Dans cet exemple, la valeur 1.0 est absorbée par la valeur 10^8.

2.3 Cancellation

La cancellation correspond à une élimination des chiffres significatifs de poids les plus élevés. Elle apparait lors de la soustraction de deux flottants très proches résultant d'un calcul. Le phénomène est d'autant plus important lors d'une accumulation de calculs avec des erreurs arrondis. Cette accumulation d'erreurs de calculs est mise en évidence par la soustraction qui est exacte lorsque les opérandes sont proches [16].

Par exemple, en C, sous unix avec le format simple précision et un arrondi au plus près :

$$((1.0 - 10^{-8}) - 1.0) * 10^8 = 0$$

Dans cet exemple, la soustraction entre les valeurs correspondant au résultat de $(1.0 - 10^{-8})$ et 1.0 entraine une perte des chiffres significatifs. Or, le résultat de la soustraction est utilisé dans une multiplication pour illustrer le phénomène de cancellation intervenu lors de la soustraction. Le résultat, ici, devrait être -1 alors qu'il est de 0.

2.4 Notations

Dans la suite de cet article, x, y, z dénotent des variables flottantes, et D_x, D_y, D_z leurs domaines associés. De la même manière, $[\underline{x}, \overline{x}]$ représente tous les flottants compris entre les valeurs $\underline{x}$ et $\overline{x} \in F$, où F représente l'ensemble des flottants. Pour un nombre flottant f, f^+ et f^- dénotent respectivement le successeur et le prédécesseur de f. Enfin $|D_x|$ dénote la cardinalité du domaine de la variable x.

3 Propriétés sur les domaines des variables et des contraintes

Cette section définit plusieurs propriétés sur les domaines des variables et sur les contraintes. Ces propriétés sont utilisées pour construire de nouvelles stratégies de recherche. Les propriétés sur les domaines des variables telles que la cardinalité ou la densité capturent la structure des domaines des variables flottantes du problème. Les propriétés sur les contraintes prennent en compte les problèmes arithmétiques des flottants comme l'absorption ou la cancellation. Elles cherchent aussi à saisir la structure du problème à travers des propriétés telles que la dérivée.

3.1 Propriétés basées sur les domaines des variables

Les propriétés explicitées dans cette section servent de base aux nouvelles stratégies de recherches proposées. Ces propriétés se concentrent sur le domaine des variables du système de contraintes. Chaque propriété contient une description, une définition, et ses intérêts.

3.1.1 La taille

Description $width(D_x)$ dénote la taille du domaine de la variable x.

Définition

$$width(D_x) = \overline{x} - \underline{x} \tag{1}$$

Intérêts La taille du domaine correspond à la distance entre les deux bornes. C'est un critère plutôt historique. En effet, sur les domaines finis, de nombreuses stratégies sont basées sur ce critère, notamment une des plus populaires, minDom [12]. La sélection des variables avec le plus petit domaine dans les stratégies sur les entiers a pour but de favoriser les variables qui contraignent le plus le problème. Sur les flottants, ce critère est plus problématique à cause de la répartition non uniforme des flottants. Pour des domaines de taille différente, le domaine le plus petit n'est pas forcément le domaine avec le moins de flottants. Par exemple, si x et y sont deux variables ayant pour domaine respectif, $D_x = [1, 2]$ et $D_y = [10, 12]$, le plus petit domaine en termes de taille étant D_x. Or c'est D_y qui contient le moins de flottants ($|D_x| = 8388608$ et $|D_y| = 2097152$ alors que $width(D_x) = 1$ et $width(D_y) = 2$). Le critère de taille et celui de la cardinalité sont identiques sur domaines répartis de manière uniforme comme sur les domaines finis alors que dans le cas des flottants ces deux critères divergent.

3.1.2 La cardinalité

Description $|D_x|$ dénote la cardinalité du domaine de la variable x.

Définition Soit $D_x = [\underline{x}, \overline{x}]$ et $\underline{x} > 0$

$$|D_x| = 2^p * (e_{\overline{x}} - e_{\underline{x}}) + m_{\overline{x}} - m_{\underline{x}} + 1 \tag{2}$$

où $m_{\underline{x}}$ et $m_{\overline{x}}$ sont les mantisses de $\underline{x}$ et $\overline{x}$, $e_{\underline{x}}$ et $e_{\overline{x}}$ sont les exposants de $\underline{x}$ et $\overline{x}$

Cette formule est aisément extensible par symétrie aux autres cas.

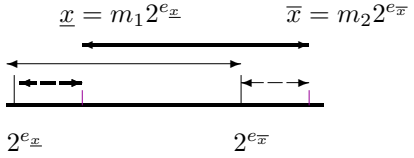

FIGURE 2 – Illustration de la formule

L'objectif est de calculer le nombre de flottants contenus dans l'intervalle D_x. Cet intervalle est représenté par la flèche en gras sur la figure 2. L'idée est donc de calculer le nombre de flottants contenus dans l'intervalle $[2^{e_{\underline{x}}}, 2^{e_{\overline{x}}}]$ (par la formule $(2^p * (e_{\overline{x}} - e_{\underline{x}}))$)

représenté par la flèche continue. Puis, de retirer le nombre de flottants compris entre $[2^{e_{\underline{x}}}, \underline{x}]$ (par la formule $-m_{\underline{x}}$) représenté par la flèche en gras et discontinue. Enfin il faut ajouter le nombre de flottants contenus dans l'intervalle $[\overline{x}, 2^{e_{\overline{x}}}]$ (par la formule $+m_{\overline{x}}$) représenté par la flèche droite discontinue. Il est nécessaire de comptabiliser un flottant supplémentaire $(+1)$ car la soustraction de la partie droite a supprimé le flottant $\underline{x}$.

Intérêts La taille et la cardinalité des domaines sont deux notions différentes sur les flottants. L'utilité d'utiliser la cardinalité pour guider la recherche est double. Tout d'abord, elle permet d'identifier les variables dont les domaines compte le moins de flottants dans le problème et donc les variables qui contraignent le plus le problème. Elle permet aussi de savoir quelle variable a le domaine le plus grand en nombre de flottants, ce qui permet de sélectionner des variables qui ont potentiellement beaucoup de valeurs appartenant aux solutions.

3.1.3 La densité

Description Cette propriété caractérise la proximité entre les flottants au sein du domaine d'une variable. On notera $\rho(D_x)$, l'application de la propriété au domaine de la variable x.

Définition

$$\rho(D_x) = \frac{|D_x|}{width(D_x)} \tag{3}$$

Intérêts La densité d'une variable peut permettre d'identifier les variables dont le domaine a des valeurs disparates (densité faible). La sélection de variables dont le domaine a une faible densité permet d'atteindre des valeurs sensiblement différentes et donc d'obtenir des comportements potentiellement différents au sein du système de contraintes. Dans le cas où les variables ont des domaines à forte densité, le nombre de valeurs du domaine potentiellement solution est plus conséquent. La densité d'un domaine augmente à proximité de 0.

3.1.4 La magnitude

Description Cette propriété permet d'identifier si le domaine est constitué uniquement de grands flottants ou de petit flottants. Par exemple, pour la magnitude du domaine $[0, 1]$, une valeur proche de 0 est attendue. La magnitude du domaine $[10^{36}, 10^{37}]$ quant à elle doit avoir une valeur proche de 1. Soit $mag(D_x)$, l'application de la propriété magnitude au domaine de la variable x.

Définition

$$mag(D_x) = \frac{e_{\underline{x}} + e_{\overline{x}}}{2 * e_{max}} \tag{4}$$

où e_{max} est l'exposant le plus grand.

Intérêts Cette propriété a un double objectif. Dans un premier temps, cette propriété peut être un moyen de sélectionner des variables qui sont potentiellement impliquées dans une absorption en combinant la sélection de variables à forte magnitude avec la sélection de variables à faible magnitude. Dans ce cas, cette propriété est simple à implanter, mais moins précise que la propriété d'absorption définie dans la section 3.2.1. Dans un second temps, à l'aide de cette propriété, les variables ayant des domaines avec des valeurs extrêmes peuvent être testées. Les valeurs extrêmes sont des valeurs particulières qui sont susceptibles d'entrainer des comportements non désirés.

3.2 Propriétés basées sur les contraintes

En plus des propriétés basées sur les domaines des variables, nous proposons des propriétés qui prennent en compte la structure des contraintes. Comme dans la section précédente, chacune des propriétés contient une description, une définition, et les intérêts. Ces propriétés sont de natures différentes, certaines sont des prédicats, d'autres retournent des valeurs.

3.2.1 Absorption

Description retourne vrai si la contrainte peut mener à une absorption, faux sinon. Cette propriété ne s'applique qu'aux contraintes d'additions. On notera cette propriété **canLeadToAnAbsorption**.

Définition Soit la contrainte $x + y = z$ avec y absorbée et un arrondi au plus près la condition :

$$canLeadToAnAbsorption = D_y \subseteq [\frac{x_m - x_m^+}{2}, \frac{x_m + x_m^+}{2}] \tag{5}$$

où $x_m = max(abs(\underline{x}), abs(\overline{x}))$ avec $abs(\underline{x})$ la valeur absolue de $\underline{x}$

Intérêts L'idée de cette propriété est de réduire la sélection des variables à celles impliquées dans des absorptions. Les problèmes d'absorptions sont récurrents lors de calcul sur les flottants. Orienter la recherche en prenant en compte ce phénomène ne peut qu'aider la recherche de contre-exemples lorsque l'absorption est mise en cause dans la correction du programme.

3.2.2 Cancellation

Description Cette propriété ne s'applique qu'aux contraintes de soustractions. Notons cette propriété **canLeadToACancellation**. La formule 6 est tirée de [2].

Définition Soit z = x - y

$$canLeadToACancellation = max(e^x, e^y) - e^z \quad (6)$$

Intérêts Cette propriété identifie les contraintes qui peuvent être impliquées dans une cancellation. Elle capture le taux de cancellation d'une contrainte. Plus le taux de cancellation est fort, plus la perte de chiffre significatif est forte. Si le taux est égal à zéro, il n'y a pas de cancellation. Tout comme les problèmes d'absorption, les problèmes de cancellation sont récurrents dans les programmes de vérification sur les flottants. Guider la recherche en tenant compte de ces phénomènes doit permettre de trouver plus efficacement un contre-exemple pour ces problèmes.

3.2.3 La dérivée

Description calcule la dérivée de la contrainte. On notera cette propriété **derivative**.

Définition

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \quad (7)$$

avec h est un flottant proche de 0.

Intérêts Dans une contrainte mono-variée, lorsque la dérivée est proche de 0, les valeurs de f(x) forment un palier. Le nombre de valeurs de x pour lesquelles la valeur de $f(x)$ est constante est donc potentiellement plus important. Dans ce cas, il y a plus de chance de trouver une solution. A contrario, lorsque la dérivée tend vers l'infini, il y a moins de valeurs solution. Un raisonnement identique peut-être fait sur les contraintes multivariées. L'objectif des stratégies qui utilisent la dérivée est de permettre de sélectionner une zone ayant potentiellement beaucoup de solutions ou au contraire très peu.

3.2.4 Le degré

Description La propriété **degré** est identique à la propriété sur les entiers. Elle se concentre sur le nombre de contraintes dans lesquels apparait la variable.

Définition

$$\sum_{i=0}^{|C|} implied(x, C_i) \quad (8)$$

où
$|C|$ est le nombre de contraintes du système
$implied(x, C_i)$ est un prédicat qui vaut 1 si la variable x est impliquée dans la contrainte C_i

Intérêts Cette propriété identifie les variables qui ont un impact fort sur la résolution du problème. Sur les domaines finis, de nombreuses stratégies utilisent cette propriété telle que weighted degree [3].

3.2.5 Les occurrences

Description Cette propriété compte le nombre maximal d'occurrences d'une variable dans les contraintes du problème.

Définition Soit $S = \{\forall\ C_i \in C,\ count(x, C_i)\}$

$$max(S) \quad (9)$$

avec
$count(x, C_i)$ est une fonction qui compte le nombre d'occurrences de x dans C_i
max est une fonction qui retourne le maximum.

Intérêts Les occurrences multiples d'une variable sont une problématique récurrente dans la vérification de programmes basés sur la programmation par contraintes. De nombreux travaux ont été faits pour gérer ce critère surtout en termes de consistance [11, 1]. L'idée ici est donc d'essayer de capturer l'importance d'une variable au sein d'une contrainte. En plus de permettre d'adapter le type de consistance au nombre d'occurrences présente au sein d'une contrainte, ce critère permet d'identifier les variables qui ont un impact fort sur une contrainte spécifique.

4 Nouvelles stratégies de recherche

Dans cette section, nous proposons un ensemble de stratégies de recherche basées sur les propriétés des sections 3.1 et 3.2. Dans un premier temps, nous définissons des stratégies simples qui se concentrent sur une seule propriété. Puis dans un second temps, nous introduisons des stratégies qui tentent d'exploiter plusieurs propriétés en même temps.

4.1 Stratégies mono-propriété

Les stratégies mono-propriété proposées suivent toutes le même schéma. Pour chaque propriété basée sur les variables, deux stratégies sont définies : une qui sélectionne la variable qui minimise la propriété et l'autre qui sélectionne la variable qui la maximise. Par exemple, pour la propriété cardinalité, deux stratégies sont obtenues, la première sélectionne la variable qui minimise la cardinalité, la seconde sélectionne la variable qui maximise la cardinalité lors de la sélection de la variable. Pour la propriété magnitude, une stratégie supplémentaire est définie qui alterne entre la sélection de variable qui minimise la magnitude et celle qui la maximise. Cette stratégie a pour objectif d'essayer de déclencher artificiellement les phénomènes d'absorption.

Pour les propriétés basées sur les contraintes, les propriétés *degree*, *occurrences* et *derivative*, sont utilisées pour définir deux stratégies. Ces deux là sont :

1. Pour les propriétés *degree* et *occurences*, les stratégies sont similaires à celles pour les variables. Une minimise le critère, alors que la seconde le maximise.

2. Pour la propriété *derivative*, la première sélectionne la variable qui est impliquée dans le plus de contraintes dont la dérivée est proche de 0. Cette sélection cherche les variables ayant potentiellement beaucoup de valeurs solutions. La seconde sélectionne d'abord celle qui est le plus impliquée dans les contraintes dont la dérivée tend vers l'infini.

Changer de stratégie durant la recherche peut-être nécessaire. Certaines propriétés sur les contraintes sont statiques (e.g degree et occurences), d'autres sont dynamiques (e.g la dérivée) ou passent de faux à vrai une seule fois lors de la descente dans l'arbre de recherche (e.g canLeadToAnAbsorption). Cette remarque est également vraie pour les stratégies de la section 4.2.2. Lorsqu'une propriété n'a plus d'intérêt, il est important d'en changer pour en choisir une qui améliore la recherche.

4.2 Stratégies multi-propriétés

Dans cette section, différentes propriétés sont combinées. Une première manière de combiner les propriétés définies dans les sections 3.1 et 3.2 est de les agréger avec une formule. Une seconde manière de les combiner est de les appliquer les unes à la suite des autres.

4.2.1 Stratégies avec score

Cette section propose la mise en relation de plusieurs propriétés en utilisant une formule. Cette proposition est inspirée de la stratégie de recherches IBS [15] (Impact Based Search). IBS est une stratégie qui mesure l'impact de l'affectation d'une variable en définissant un score. Ce score correspond à la réduction de l'espace de recherche suite à la propagation de l'affectation de la variable. Ce score peut-être diminué ou augmenté selon l'efficacité du choix précédemment effectué lors de la recherche. À chaque étape, la variable ayant le meilleur score est choisie. Les stratégies présentées dans cette section reprennent cette idée de privilégier en premier les variables qui satisfont au mieux différents scores. Les grandes lignes de la sélection de variables sur le principe d'un score sont les suivantes :

1. Calculer le score de chaque variable.
2. Sélectionner la variable qui maximise le score.

Le premier score S_1 présenté essaye de lier la propriété de taille à celle de cardinalité. Le choix de ces deux propriétés est du à leur proximité. En effet, l'une définit la distance les bornes du domaine et l'autre le nombre de flottants présents dans ce même domaine.

$$S_1(x) = \omega_1|D_x| + \omega_2\rho(D_x)$$

Les ω_i représentent des pondérations. Ces pondérations seront à définir lors des expérimentations.

Le second score S_2 réunit les mêmes propriétés que S_1 en ajoutant le degré de la variable.

$$S_2(x) = -\frac{S_1(x)}{degree(x)}$$

Ce choix tente de prendre en compte les variables qui contraignent le plus le problème pour avoir un plus grand impact sur la résolution.

4.2.2 Autres stratégies multi-propriétés

Dans cette section, nous proposons des stratégies qui allient les propriétés basées sur les contraintes, sur les domaines des variables, et les scores. Combiner ces stratégies de recherche permet d'être plus fin au niveau de la recherche en prenant en compte plusieurs critères. Par exemple, utiliser une stratégie de recherche basée sur une propriété statique ne prend en compte que la structure initiale du problème. Or, la descente dans l'arbre de recherche fait qu'on est confronté à un sous-problème du problème initial où la propriété peut ne plus être significative. Pour ce faire, les propriétés définies sur les contraintes sont utilisées pour réduire le choix des variables à celle qui ont une structure particulière (e.g qui peuvent mener à une absorption). Une fois cette réduction effectuée, les propriétés établies

sur les variables sont utilisées pour sélectionner celle qui nous intéresse le plus. L'utilisation d'un score peut aussi remplacer la propriété sur les variables.

Le premier groupe de stratégies par propriétés sur les contraintes que nous proposons. Le premier utilise la propriété **canLeadToAnAbsorption** pour réduire le choix des variables à celle impliquée dans une absorption. À la suite de l'application de cette propriété, une des stratégies mono-propriété est utilisée. L'idée de ce groupe de stratégies est le suivant :

1. Extraire les variables qui peuvent mener à une absorption (en utilisant le prédicat canLeadToAnAbsorption).
2. Appliquer la propriété choisie aux variables extraites.
3. Sélectionner la variable qui maximise ou minimise la propriété.

Le second groupe de stratégies proposé, repose sur la propriété **canLeadToACancellation** pour restreindre le choix des variables à celles qui sont impliquées dans une cancellation. La propriété canLeadToACancellation calcule un taux de cancellation. La restriction peut donc être modulée en choisissant le taux minimal d'acceptation d'une variable (e.g toutes les variables impliquées dans une cancellation avec un taux supérieur à 0.5). Suite à l'application de cette propriété, une des stratégies définies à la section 4.1 est appliquée.

Enfin pour les propriétés **degree**, **occurrences** et **dérivative** deux choix sont à évaluer. Le premier consiste à minimiser la propriété choisie, le second à la maximiser. Une fois le choix appliqué, une des stratégies monopropriété est mise en œuvre. On peut également envisager de composer d'autres propriétés sur les contraintes par exemple *canLeadToAnAbsorption* ou *canLeadToACancellation* à la suite de propriétés choisies, pour ensuite finir par une propriété sur les variables.

Ces stratégies peuvent également être combinées en les appliquant à des sous-domaines. Par exemple, une fois une variable sélectionnée, il est possible de diviser le domaine de la variable en plusieurs sous-domaines pour appliquer différentes stratégies sur chaque sous-domaines.

5 Implantation

Les stratégies présentées sont en cours d'évaluation sur un ensemble de benchmarks. Pour ce faire, le solveur FPCS (Floating Point Constraint Solver) [13] développé par un des auteurs a été porté dans Objective-CP. Objective-CP est un outil d'optimisation introduit dans [17]. Ce portage a été effectué pour apporter une flexibilité dans le développement des stratégies de recherche que nous proposons. Dans cette section nous discutons des choix d'implantation effectués.

5.1 Choix du solveur

Pour explorer de nouvelles stratégies de recherche sur les problèmes de vérification, nous avons besoin d'un solveur sur les flottants et d'un solveur avec la possibilité de définir des stratégies flexibles. Un solveur sur les flottants FPCS a été développé au sein de notre équipe de recherche. Ce solveur est basé sur un filtrage fort. Cependant, l'écriture de stratégie de recherche dans FPCS est une tâche longue et fastidieuse. Objective-CP quant à lui est une boite à outils d'optimisation sur les domaines finis, avec de nombreuses facilités pour le développement de stratégies de recherche flexible. Pour avoir toute l'efficacité de FPCS et la flexibilité des stratégies de recherche d'Objective-CP nous avons décidés d'incorporer FPCS dans Objective-CP. Ces deux outils sont complémentaires. Dans les deux parties qui suivent, nous présentons d'abord ces deux solveurs, leurs avantages et limites.

5.1.1 FPCS

Présentation FPCS est un solveur de contraintes sur les flottants conçu et développé par C.Michel. Ce solveur est écrit en C++, il a été développé dans la perspective de la vérification de programme.

Avantages FPCS a de nombreux avantages. C'est un solveur complet et efficace supportant les filtrages fort tel que la KB-consistance [11], les différents modes d'arrondi, et un catalogue de contraintes issu des problèmes de vérification de programmes avec flottants. De plus, l'ajout des contraintes au modèle est simple (voir la figure 3).

```
model.add(a = 2 * b + 3 * c);
```

FIGURE 3 – Ajout de contrainte dans FPCS

Limites Le principal défaut de FPCS est qu'il n'a pas été conçu pour développer des stratégies de recherche complexes, mais a plutôt vocation à être utilisé avec des stratégies simples comme la bissection. De ce fait, le code des stratégies de recherche peut vite devenir long et fastidieux et l'arbre de recherche doit être traité en ajoutant les différents points de choix un à un. Par exemple, 80 lignes de codes sont nécessaires pour implanter la bissection. De plus, l'ajout d'un point de choix entraine de nombreuses copies non désirées.

```
1       choice->domain->setLBTo(&low);
2       choice->domain->setUBTo(&low);
3       push();
4
5       choice->domain->setLBTo(&up);
6       choice->domain->setUBTo(&up);
7       push();
```

FIGURE 4 – Définition d'une stratégie de recherche dans FPCS

Dans l'exemple de la figure 4, les deux points de choix successifs correspondant aux bornes de l'intervalle courant sont ajoutés à la stratégie. Ici, *choice* est une variable. La première ligne force la borne inférieure du domaine de la variable *choice* à la valeur low. La seconde force la borne supérieure du domaine de la variable *choice* à la valeur low également. À la troisième ligne le choix de la réduction du domaine de la variable *choice* à $[low, low]$ est ajouté à la pile des choix de la recherche. De la cinquième ligne à la septième, le domaine de la variable *choice* est réduit à $[up, up]$ et le point de choix est ajouté.

5.1.2 Objective-CP

Présentation Objective-CP est un outil d'optimisation conçu par P. VanHentenryck et L.Michel [17]. Cet outil est développé en Objective-C et contient plusieurs solveurs, dont un solveur CP (Constraint Programming), un solveur LP (Linear Programming), et un solveur MIP (Mixed-Integer Programming).

Avantages Objective-CP est un outil qui sépare la partie modélisation de la partie résolution (solveur) et stratégie de recherche. En effet, lors de la modélisation du problème les variables et les contraintes définies sont abstraites. Le passage de la modélisation à la résolution s'effectue par la définition d'un programme (CPProgram ou LPProgram ou MIPProgram). Ce programme correspond au solveur (cf. figure 5) qui sera appelé pour concrétiser les variables et contraintes abstraites en variables et contraintes concrètes (interne à chaque solveur). Dans la figure 5, les deux premières lignes définissent un programme CP. Ce programme correspond au modèle concrétisé par un solveur CP. Aux deux dernières lignes, un processus identique est effectué avec un programme LP. La partie recherche est basée sur les concepts de continuations et de contrôleurs qui permettent l'écriture de stratégies flexibles, faciles à prendre en main et concises en termes de lignes de code. Par exemple, la figure 6 montre comment implanter la bissection en 10 lignes. Enfin, cet outil incorpore également une partie qui permet de paralléliser en quelques lignes les solveurs.

```
id<CPProgram> p;                                    1
p = [ORFactory createCPProgram:model];              2
                                                    3
id<LPProgram> p2;                                   4
p2 = [ORFactory createLPProgram:model];             5
```

FIGURE 5 – Appel au solveur CP

```
while (![xi bound]) {                               1
  ORFloat theMax = xi.max;                          2
  ORFloat theMin = xi.min;                          3
  ORFloat mid = (theMin + theMax)/2.0f;             4
  if(mid == theMax)                                 5
        mid = theMin;                               6
  [_search try: ^{                                  7
    [self floatGthenImpl:xi with:mid];              8
    }                                               9
   alt: ^{                                          10
    [self floatLEqualImpl:xi with:mid];             11
    }                                               12
   ];                                               13
}                                                   14
```

FIGURE 6 – Exemple d'une stratégie de recherche avec bissection en Objective-CP

Dans la figure 6, aux seconde et troisième lignes les bornes de la variable xi sont extraites. La quatrième ligne correspond au calcul du milieu de l'intervalle *mid*. Le test à la cinquième ligne permet de gérer le cas où il n'y a que 3 flottants. La septième ligne correspond au branchement. Le *try* représente la branche où $xi > mid$. Le *alt* correspond à la branche $xi \leq mid$. Ce processus est répété tant que le domaine de x_i n'est pas dégénéré.

Limites Le solveur CP implanté dans Objective-CP ne supporte pas les flottants donc tout un travail d'implantation est nécessaire pour l'ajout de la partie flottante. Du fait que Objective-CP est basé sur Objective-C l'ajout des contraintes est moins élégant que dans FPCS (figure 7), mais dans le cadre de la vérification de programme les modèles sont généralement créés de manière automatique.

```
[model add:[a eq:[[b mul:@(2)]
                 plus:[c mul:@(3)]]]];
```

FIGURE 7 – Ajout de la contrainte $a = 2b + 3c$ en Objective-CP

6 Discussions et perspectives

Cet article propose plusieurs stratégies de recherche dédiées aux systèmes de contraintes sur les flottants, et plus particulièrement aux problèmes de vérification de programme avec du calcul sur les flottants. Les différentes stratégies proposées sont basées sur un ensemble de propriétés basées sur deux critères : les domaines des variables et les contraintes. Les premières cherchent à capturer la structure spécifique des flottants telles que leurs nombres ou leurs densités. Les secondes tendent à prendre en compte les problèmes liés à l'arithmétique propre aux flottants et à la structure même des problèmes. Un travail d'expérimentation pour évaluer ces stratégies sur un ensemble d'exemple de problèmes liés à la vérification de programme est actuellement en cours.

Références

[1] Mohammed Said Belaid, Claude Michel, and Michel Rueher. Boosting local consistency algorithms over floating-point numbers. In *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, pages 127–140, 2012.

[2] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. A dynamic program analysis to find floating-point accuracy problems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 453–462, 2012.

[3] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. *ECAI'04*, pages 146–150, 2004.

[4] Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. Cpbpv : A constraint-programming framework for bounded program verification. *Constraints*, 15(2) :238–264, 2010.

[5] Hélène Collavizza, Nguyen Le Vinh, Michel Rueher, Samuel Devulder, and Thierry Gueguen. A dynamic constraint-based BMC strategy for generating counterexamples. *26th ACM Symposium On Applied Computing*, 2011.

[6] Steven Gay, Renaud Hartert, Christophe Lecoutre, and Pierre Schaus. Conflict ordering search for scheduling problems. In *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*, pages 140–148, 2015.

[7] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1) :5–48, 1991.

[8] IEEE. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Standard*, 754, 2008.

[9] Narendra Jussien and Olivier Lhomme. Dynamic domain splitting for numeric csps. *ECAI*, pages 224–228, 1998.

[10] R. Baker Kearfott. Some tests of generalized bisection. *ACM Trans. Math. Softw.*, 13(3) :197–220, September 1987.

[11] Olivier Lhomme. Consistency techniques for numeric csps. In *Proceedings of the 13th International Joint Conference on Artifical Intelligence - Volume 1*, IJCAI'93, pages 232–238, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[12] Jeff T. Linderoth and Martin W. P. Savelsbergh. A computational study of search strategies for mixed integer programming. *INFORMS Journal on Computing*, 11(2) :173–187, 1999.

[13] C. Michel, M. Rueher, and Y. Lebbah. Solving constraints over floating-point numbers. In *Principles and Practice of Constraint Programming — CP 2001 : 7th International Conference, CP 2001 Paphos, Cyprus, November 26 – December 1, 2001 Proceedings*, pages 524–538, Berlin, Heidelberg, 2001.

[14] Laurent Michel and Pascal Van Hentenryck. Activity-based search for black-box constraint programming solvers. In Nicolas Beldiceanu, Narendra Jussien, and Éric Pinson, editors, *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems : 9th International Conference, CPAIOR 2012, Nantes, France, May 28 – June1, 2012. Proceedings*, pages 228–243, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[15] Philippe Refalo. Impact-based search strategies for constraint programming. *CP 2004*, 3258 :557–571, 2004.

[16] P.H. Sterbenz. *Floating-point computation.* Prentice-Hall series in automatic computation. Prentice-Hall, 1973.

[17] Pascal Van Hentenryck and Laurent Michel. The objective-cp optimization system. In *Principles and Practice of Constraint Programming : 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, pages 8–29, Berlin, Heidelberg, 2013.