Stratégies de recherches dédiées à la résolution de systèmes de contraintes sur les flottants pour la vérification de programmes

Heytem Zitoun

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Stratégies de recherche dédiées à la résolution de systèmes de contraintes sur les flottants pour la vérification de programmes

Heytem ZITOUN

Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis (I3S)
UMR7271 UNS CNRS

Présentée en vue de l'obtention du grade de docteur en Informatique d'Université Côte d'Azur

Dirigée par :
Michel Rueher et Claude Michel

Soutenue le :
26 Octobre 2018

Devant le jury, composé de :

Goubault Éric,	Professeur,	École Polytechnique
Podelski Andréas,	Professeur,	Université de Freiburg
Collavizza Hélène,	MCF - HDR,	Université Côte d'Azur
Marre Bruno,	Chercheur,	CEA
Martel Matthieu,	Professeur,	Université de Perpignan
Michel Laurent,	Professeur,	Université du Connecticut
Michel Claude,	IR - HDR,	Université Côte d'Azur
Rueher Michel,	Professeur,	Université Côte d'Azur

cnrs

AGENCE NATIONALE DE LA RECHERCHE
ANR

Stratégies de recherche dédiées à la résolution de systèmes de contraintes sur les flottants pour la vérification de programmes

Composition du jury :

Rapporteurs

Goubault Éric,	Professeur,	École Polytechnique
Podelski Andréas,	Professeur,	Université de Freiburg

Examinateurs

Collavizza Hélène,	Maître de Conférences - HDR,	Université Côte d'Azur
Marre Bruno,	Chercheur,	CEA
Martel Matthieu,	Professeur,	Université de Perpignan
Michel Laurent,	Professeur,	Université du Connecticut

Résumé de la thèse

Stratégies de recherche dédiées à la résolution de systèmes de contraintes sur les flottants pour la vérification de programmes

Résumé : La vérification des programmes est un enjeu majeur pour les applications critiques comme l'aviation, l'aérospatiale ou les systèmes embarqués. Les approches Bounded model checking (e.g., CBMC) et de programmation par contraintes (e.g., CPBPV, ...) reposent sur la recherche de contre-exemples qui violent une propriété du programme à vérifier. La recherche de tels contre-exemples peut être très longue et coûteuse lorsque les programmes à vérifier contiennent des calculs en virgule flottante. Ceci est dû en grande partie au fait que les stratégies de recherche existantes ont été conçues pour des domaines finis et, dans une moindre mesure, pour des domaines continus.

Dans cette thèse, nous proposons un ensemble de stratégie de recherche dédié à la vérification de programme avec du calcul sur les flottants. Les stratégies proposées pour les choix de variables et de choix de valeur se basent sur des propriétés propres aux flottants. Ces propriétés utilisent des caractéristiques des domaines des variables, ou de la structure des contraintes. Certaines propriétés qui portent sur les domaines des variables sont classiques comme la taille et la cardinalité et d'autres beaucoup plus spécifiques comme la densité. Les notions de taille et cardinalité sont équivalentes sur les entiers, mais ne le sont pas sur les flottants. Ainsi la densité capture une variabilité qui est très spécifique aux flottants dont la moitié se trouve entre [-1,1]. De manière similaire les propriétés qui portent sur la structure des contraintes sont, pour certaines tels que le degré ou le nombre d'occurrences, issues des domaines finis, et pour d'autres beaucoup plus spécifiques, comme l'absorption, et la cancellation ; ces deux propriétés capturent des phénomènes qui sont généralement la cause de fortes déviations du programme flottant vis-à-

vis son interprétation sur les réels et donc de l'existence même de beaucoup de contre-exemples.

Pour chaque propriété, deux stratégies de choix de variables sont proposées. La première choisit la variable qui minimise la propriété, alors que la seconde choisit la variable qui la maximise. Les stratégies de choix de valeurs essaient quant à elles de tirer profit des phénomènes d'absorption et de cancellation.

L'évaluation de ces stratégies sur un ensemble de programmes réalistes est très encourageante : ces stratégies sont plus efficaces que les stratégies standards.

Search strategies for solving constraint systems over floats for program verification

Abstract : Program verification is a major issue for critical applications such as aviation, aerospace or embedded systems. Bounded model checking (e.g., CBMC) and constraint programming (e.g., CPBPV,...) approaches are based on the search for counter-examples that violate a property of the program to verify. The search for such counter-examples can be very time-consuming and costly when the programs to be verified contain floating point calculations. This is largely due to the fact that existing research strategies have been designed for finite domains and, to a lesser extent, for continuous domains.

In this thesis, we propose a set of search strategies dedicated to program verification with floating point computation. The proposed strategies for variable and value selection are based on specific floating properties. These properties use characteristics of the variable domains, or the constraint structure. Some properties that focus on the domains of the variables are classic such as size and cardinality and others much more specific like density. The notions of size and cardinality are equivalent on the integers, but not on the floats. Density captures a variability that is very specific to the floats, half of which are between[-1.1]. Similarly, the properties that concern the structure of constraints are, for some such as the degree or number of occurrences, derived from finite domains, and for others much more specific, such as absorption, and cancellation ; these two properties capture phenomena that are generally the cause of strong deviations of the floating point program from its interpretation on the reals and hence the existence of many counterexamples.

For each property, two variable selection strategies are proposed. The first one chooses the variable that minimizes the property, while the second one chooses the variable that maximizes it. Value choice strategies try to take advantage of the phenomena of absorption and cancellation.

Remerciement

Tout d'abord je voudrais remercier Éric Gaubault et Andréas Podelsky d'avoir accepté de relire cette thèse et d'en être les rapporteurs. Je remercie également Bruno Marre, Hélène Collavizza, et Matthieu Martel pour avoir accepté d'assister à la présentation de ce travail et de participer à mon jury de thèse.

Je voudrais remercier tout particulièrement Laurent Michel qui m'a permis de progresser énormément grâce à ces conseils et ces nombreuses connaissances théoriques et techniques. J'ai également apprécié sa disponibilité et son humour.

Un grand merci à mes directeurs de thèse : Michel Rueher et Claude Michel. Merci pour ce sujet, ces conseils, ces discussions scientifiques ou non. J'ai apprécié leur patience, leur humour, mais aussi ces repas partagés.

Merci également à Jean Charles Régin, Arnaud Malapert, Marie Pelleau, Jean-Paul Comet, Gilles Bernot, Alexandre Muzi, Enrico Formenti et tous les membres de l'équipe MDSC pour leur gentillesse et bienveillance. Je remercie Sandra Devauchelle, notre assistante pour sa gentillesse et son aide durant ces trois années.

Je remercie également Mohamed Belaid Said, et Mohamed Rezghi pour leurs conseils et discussions.

Je tiens également à remercier tous les doctorants que j'ai pu rencontré grâce à ce doctorat : Ophélie Guinaudeau, Laetitia Laversa, Ingrid Grenet, Assia Kamal Idrissi, Cyrille Mascart, Oussama Sabri, Nicolas Isoart, Benjamin Miraglio, Arthur Finkelstein, et Valentin Montmirail. Je remercie tout particulièrement Rémy Garcia et Jonathan Behaegel (Team114) de m'avoir accompagné aux conférences/écoles d'été.

Mes remerciements s'étendent à mes camarades de promotion devenus amis : Guillaume Perez, Mehdi Ahizoune, Jean-Michel Diaz Vaz, Sami Lazreg, Yassine Ferkouch, Anthony Palmieri et Nicolas Huin.

Un grand merci à mes amis dracenois : Claire Duhamel, Cédric Garcia, Charlotte et Jérémy Garcia, Dalila et Anthony, Jimmy Tourtelot, Romain Agostini, Tristan Legros, Arthur Gellas, Ellie Delaye, Audrey Giessinger et Fahmi Ahammar. Je remercie aussi : Aimen Nouira et Yassin Mezni (TeamBo,Team100), mais aussi à ma grande et fidèle amie Estelle Turrel (TeamDr). Merci à ma famille Sarra, Yasser et Nader. Un merci tout particulier à ma mère sans qui je ne serais rien, merci pour son soutien, ses sacrifices et son amour.

Je dédie cette thèse à mon défunt père.

Table des matières

Chapitre 1

Introduction

De nombreux dispositifs technologiques sont contrôlés par des systèmes informatiques complexes. Ces systèmes informatiques commandent des processus critiques dans l'automobile, l'aviation, l'énergie nucléaire ou encore dans le domaine médical. Des erreurs dans ces systèmes complexes peuvent avoir des conséquences dramatiques. Ces erreurs trouvent souvent leur source dans des calculs impliquant des nombres à virgule flottante. Les exemples les plus marquants sont l'échec du missile Patriot en 1991[Defense, 1992], ou encore l'explosion de la fusée Ariane 5 en 1996 [Einarsson, 2005].
Un missile américain Patriot n'a pas pu suivre et intercepter un missile Scude Irakien à cause d'un problème logiciel. Ce problème venait d'erreurs d'arrondi lors du calcul du temps écoulé depuis le démarrage du système de lancement du missile. Pour calculer la position du missile ennemi, le programme estimait sa vitesse en utilisant des nombres à virgules flottante, ainsi que le temps écoulé depuis le démarrage du système de lancement missile Patriot. Pour connaitre le temps écoulé, l'horloge interne du missile utilise une représentation entière. La vitesse étant en nombre à virgule flottantes, le temps devait être converti en flottant. Pour ce faire, le programme multipliait le temps par une approximation flottante de 0.1 sur 24 bits. Comme la valeur 0.1 n'a pas de représentation exacte en flottant, cette multiplication entraine une d'erreur d'arrondi. Après 100 heures, cette accumulation d'erreurs d'arrondi provoquait une erreur de 687 mètres dans le calcul de la position du missile Irakien.
Une des raisons de l'explosion de la fusée Ariane 5 était la conversion d'un nombre à virgule flottante (64 bits) en entier (16 bits).
Le point commun de ces défaillances est une mauvaise utilisation des calculs sur les nombres à virgule flottante. Dans ces cas, comme dans beaucoup

d'autres, les erreurs proviennent souvent de la non prise en compte des propriétés spécifiques du calcul sur les flottants par le programmeur.
Les calculs sur les nombres flottants sont dérivés de modèles mathématiques sur les réels [Goldberg, 1991]. Néanmoins, ils n'héritent pas de propriétés, telles que la continuité et les théorèmes qui en découlent.
Les calculs sur les nombres à virgule flottante sont différents des calculs sur les réels. Par exemple, considérons les équations :

$$x + 16.0 = 16.0, x \neq 0.0 \tag{1.1}$$

$$x^2 = 2 \tag{1.2}$$

Aucun réel n'est solution des équations 1.1, alors qu'une multitude de nombres flottants sont solution de ces équations. A contrario, $\sqrt{2}$ est une solution sur les réels dans l'équation 1.2, alors qu'aucune solution n'existe sur les nombres flottants [1]. Un autre point important est que de nombreux réels ne sont pas représentables de manière exacte (e.g. 0.1) par les nombres flottants. Sur les flottants, les opérateurs arithmétiques ne sont ni associatifs ni distributifs, et peuvent être sujets à des phénomènes tels que l'absorption et la cancellation.

Le programme de la Figure 1.1 illustre un problème majeur où des erreurs de calcul dues à l'arithmétique spécifique des flottants peuvent conduire à une décision sensiblement différente de celle qui aurait été obtenue sur les nombres réels. En interprétant le programme sur les réels, la valeur de r calculée est 1 (ligne 5) : l'instruction `doThenPart` devrait donc être exécutée. Sur les nombres flottants, un problème d'absorption se produit : la valeur 1 est absorbée par `1e8f` [2] et r est donc égal à 0. Le test de ligne 6 échoue, et la branche `else` est exécutée. Cette différence de comportement peut avoir de lourdes conséquences pour un programme embarqué si, par exemple, elle conditionne le freinage (ou non) dans un système comme l'ABS (système Anti-Blocage des Roues).

Les calculs en virgule flottante sont donc une source supplémentaire d'erreurs dans les programmes embarqués. Ces problèmes viennent du fait que très souvent, les programmes sont conçus par des programmeurs qui ont la sémantique des nombres réels à l'esprit. Pour des décisions critiques, il est vital de s'assurer que certaines propriétés ne peuvent pas être violées à cause des erreurs arithmétiques sur les nombres flottants [Titolo et al., 2018a]. La recherche de contre-exemples (valeurs d'entrées qui violent ces propriétés) est donc un problème important dans la vérification des programmes embarqués.

1. avec un monde d'arrondi au plus près
2. sur les flottants simple précision et avec un arrondi au plus près.

```
1  void foo(){
2    float a = 1e8f;
3    float b = 1.0f;
4    float c = -1e8f;
5    float r = a + b + c;
6    if(r >= 1.0f){
7      doThenPart();
8    } else {
9      doElsePart();
10   }
11 }
```

FIGURE 1.1 – Exemple 1

Dans cette thèse, nous proposons une méthode pour la recherche de tels contre-exemples. Cette méthode est basée sur la programmation par contraintes (PPC). La programmation par contraintes est un paradigme utilisé pour résoudre des problèmes combinatoires difficiles tels que les problèmes de planification ou d'ordonnancement [Bartk, 1970]. Ce paradigme se base sur une séparation entre la modélisation et la résolution du problème. La modélisation formalise le problème de manière déclarative et mathématique, alors que la résolution se concentre sur la manière de trouver une solution. La résolution est basée sur deux principes : le filtrage et la recherche de solution. Le filtrage consiste à supprimer les valeurs qui ne peuvent trivialement pas faire partie de la solution. La recherche de solution intervient quant à elle, lorsqu'il n'est plus possible d'effectuer de nouvelles réductions. Elle consiste à explorer l'espace de recherche en faisant des choix qui permettent d'effectuer de nouvelles réductions. Si les choix effectués mènent à une impasse, un retour en arrière (backtrack) est effectué. La recherche se termine lorsque l'espace de recherche est totalement exploré. La recherche de solution reprend l'idée du "diviser pour régner". Les choix effectués consistent à découper le problème initial en sous problèmes plus simples et plus facile à résoudre.

La PPC a déjà été utilisée pour traiter les problèmes de vérification de programme sur les domaines entiers [Podelski, 2000, Gotlieb et al., 2000, Collavizza et al., 2010, Gotlieb, 2012, Collavizza et al., 2011]. Les contraintes sur les flottants ont été introduites [Michel et al., 2001, Botella et al., 2006] pour permettre de vérifier des programmes avec du calcul sur les nombres flottants [Collavizza et al., 2016, Ponsini et al., 2016]. Sur les domaines flottants, la recherche de contre-exemples reste longue et coûteuse, car les stratégies de recherche classiques sont peu efficaces pour trouver de tels contre-exemples.

De nombreuses stratégies de recherche ont été proposées sur les entiers [Linderoth and Savelsbergh, 1999, Boussemart et al., 2004, Gay et al., 2015, Michel and Van Hentenryck, 2012, Refalo, 2004] et, dans une moindre mesure, sur les réels [Jussien and Lhomme, 1998, Batnini et al., 2005, Kearfott, 1987]. Or, les stratégies dédiées aux entiers ou aux réels sont mal adaptées aux flottants car ces domaines ont des propriétés différentes. Le sous-ensemble des entiers d'un domaine bornés par deux entiers est fini et réparti de manière uniforme ; il est donc possible d'en énumérer toutes les valeurs lorsque le domaine est suffisamment petit. Le sous-ensemble des réels bornés 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 qui exploitent par exemple la jacobienne pour montrer qu'un intervalle contient au moins une solution. 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 généralement inenvisageables dans le cas des flottants. Les flottants correspondent à une approximation des réels, mais n'héritent pas des mêmes propriétés mathématiques, telles que la continuité. Il est donc difficile de tirer profit des stratégies de recherches basées sur les propriétés des réels.

Contributions

Dans cette thèse, nous cherchons à exploiter les spécificités des flottants pour proposer un ensemble de stratégies de recherche efficaces. En PPC, les stratégies de recherche reposent sur un choix de variables et un choix de valeurs. Les stratégies introduites ici pour les choix de variables et de valeurs se basent sur des propriétés propres aux flottants. Ces propriétés utilisent des caractéristiques des domaines des variables, ou de la structure des contraintes. Ainsi, la densité capture une variabilité qui est très spécifique aux flottants dont la moitié se trouve entre [-1,1]. L'absorption, et la cancellation capturent des phénomènes qui sont généralement à l'origine de fortes déviations du programme flottant vis-à-vis de son interprétation sur les réels et donc de l'existence même de nombreux contre-exemples.
Certaines propriétés comme la taille et la cardinalité sont classiques, mais ces notions qui sont équivalentes sur les entiers, ne le sont pas sur les flottants. De manière similaire les propriétés qui portent sur la structure des contraintes comme le degré ou le nombre d'occurrences sont issues des domaines finis.

Les stratégies de recherche que nous avons proposées ont été évaluées sur

un ensemble significatif de benchmarks. Ces expérimentations ont montré que des méthodes de recherche adaptées aux spécificités des flottants permettent d'obtenir des gains de performances significatifs et de réaliser des solveurs bien plus efficaces que les solveurs SMT pour la recherche de contre-exemples.

Première partie

Etat de l'art

Chapitre 2

Les nombres à virgule flottante

Ce chapitre rappelle les notions de bases des nombres à virgules flottantes qui sont nécessaire à la compréhension des travaux présentés dans ce manuscrit.

Les nombres à virgule flottante ont été introduits pour approximer les nombres réels. Le standard IEEE 754-2008 normalise les nombres en virgule flottante [IEEE, 2008], leurs formats, ainsi que certaines propriétés de l'arithmétique en virgule flottante. Les deux formats les plus courants définis dans la norme IEEE 754 sont les simples et les doubles qui utilisent respectivement 32 bits et 64 bits.

2.1 Représentation

Un nombre à virgule flottante est représenté par un triplet (s, m, e) où $s \in \{0, 1\}$ représente le signe, m la mantisse et, e l'exposant [Goldberg, 1991]. La mantisse est de la forme $d_0.d_1 \dots d_p$, où $d_i \in \{0, 1\}$ avec $d_0 = 0$ ssi $e = 0$ sinon $d_0 = 1$. p représente la précision (23 pour les simples et 52 pour les doubles).

Un nombre à virgule flottante est défini par :

$$(-1)^s \times m \times 2^{e-biais} \tag{2.1}$$

Pour pouvoir représenter à la fois petits et grands nombres, le standard IEEE 754 propose l'utilisation d'un *biais* pour calculer la valeur effective de l'exposant. Ce biais permet d'avoir un exposant signé.

2.1.1 Flottant normalisé

Un nombre à virgule flottante *normalisé* est caractérisé par son bit de poids le plus fort, d_0, fixé à 1 et un exposant e différent de 0 et de la valeur maximale (Fixer d_0 à 1 donne à ce nombre une représentation unique). Un nombre flottant normalisé est donc défini par :

$$(-1)^s \times 1 \ . \ d_1 \dots d_p \times 2^{e-biais} \tag{2.2}$$

Exemple:

Le tableau ci-dessous présente quelques nombres flottants normalisés en simple précision. $1.17549435 \times 10{-}38$ est le plus petit nombre normalisé positif, et $3.40282347 \times 10^{38}$ le plus grand positif normalisé.

Valeur décimale (approximation)	signe	exposant	mantisse
$2.35098898 \times 10^{-38}$	0	00000010	00000000000000000000001
1.6125×10^{1}	0	10000011	00000010000000000000000
$3.40282347 \times 10^{38}$	0	11111110	11111111111111111111111

2.1.2 Flottant dénormalisé

La norme IEEE 754 introduit également les nombres dénormalisés qui permettent d'assurer une transition lente avec zéro. Ils permettent également de garantir des propriétés comme : $x = y \iff x - y = 0$.

Un nombre à virgule flottante *dénormalisé* est caractérisé par un bit de poid le plus fort, d_0 fixé à 0 et un exposant à 0. Il est défini par :

$$(-1)^s \times 0 \ . \ d_1 \dots d_p \times 2^{0-biais} \tag{2.3}$$

Exemple:

Le tableau ci-dessous illustre quelques nombres flottants dénormalisés en simple précision. 1.4×10^{-45} est le plus petit nombre dénormalisé positif, et $1.1754942106924411 \times 10^{-38}$ le plus grand positif dénormalisé.

Valeur décimale (approximation)	signe	exposant	mantisse
1.4×10^{-45}	0	00000000	00000000000000000000001
$2.8684579564729005 \times 10^{-42}$	0	00000000	00000000000011111111111
$1.1754942106924411 \times 10^{-38}$	0	00000000	11111111111111111111111

2.1.2.1 ULP

L'**ULP** (*Unit in the Last Place*) est une unité de mesure de la précision. Elle est utilisée pour mesurer l'erreur des opérations sur les flottants et correspond à la distance qui sépare deux flottants consécutifs. La fonction $ulp(x)$ peut-être défini par :

$$ulp(x) = \begin{cases} x^+ - x, & \text{si } x \geq 0 \\ x - x^-, & \text{sinon} \end{cases} \tag{2.4}$$

Les flottants x^+ et x^- représentent respectivement le successeur et le prédécesseur de x.

Néanmoins, cette définition est ambiguë au voisinage de $\pm\infty$ et lors de changement de puissance de 2. Golberg, Muller et Harrison [Goldberg, 1991, Muller, 2005, Harrison, 1999] proposent d'autres définitions plus précises.

Les flottants étant répartis de manière non uniforme, la valeur de l'ULP augmente donc lorsque la distance avec zéro augmente. Plus particulièrement, la fonction $ulp(x)$, est une fonction croissante, positive et en escalier lorsque x est positif. Elle est également symétrique avec l'axe des ordonnées.

2.1.3 Formats

Deux formats sont explicités par la norme pour représenter des nombres à virgule flottante. Le format à simple précision et le format à double précision. Ces formats définissent, la taille (i.e le nombre de bits) de la mantisse et de l'exposant (Figure 2.1).

Format simple précision Le format simple précision est défini sur 32 bits :

- ◇ 1 bit pour le signe
- ◇ 8 bits pour l'exposant
- ◇ 23 bits pour la mantisse

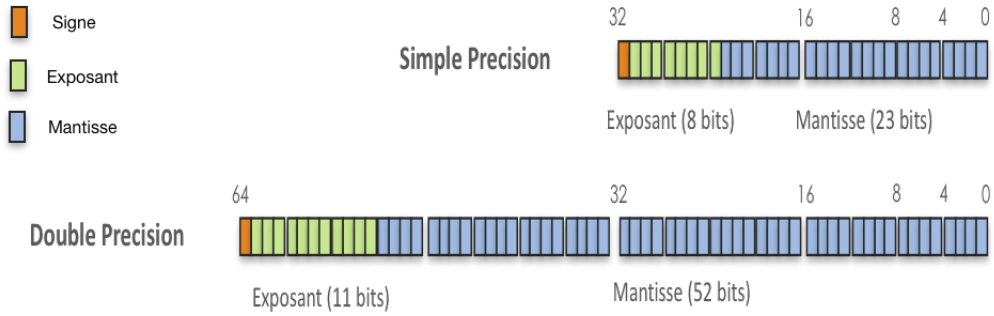


FIGURE 2.1 – Formats des nombres à virgule flottante

Format double précision Le format double précision est défini sur 64 bits :

- ◇ 1 bit pour le signe
- ◇ 11 bits pour l'exposant
- ◇ 52 bits pour la mantisse

Valeurs spéciales
La Table 2.1 exhibe la représentation des valeurs symboliques sur les flottants en simples précisions. Ces valeurs peuvent être signées.

Valeur	exposant	mantisse
zero	0000 0000	000 0000 0000 0000 0000 0000
∞	1111 1111	000 0000 0000 0000 0000 0000
NaN	1111 1111	Au moins un bit non nul

TABLE 2.1 – Représentation des valeurs symboliques sur les flottants simple précision

Les valeurs $\pm\infty$ permettent de gérer les dépassements de capacités (**overflow**). Ils apparaissent lors d'opérations où la magnitude du résultat est trop grande pour être représentée dans le format choisi. Un **NaN** (*Not-a-number*) est le résultat d'une opération non définie sur les réels. Elle est, par exemple, le résultat $\sqrt{-1}$. NaN a plusieurs représentations possibles.

2.1.3.1 Opérations de base

Le standard IEEE normalise 5 opérations de base : l'addition ($+$), la soustraction ($-$), la multiplication ($*$), la division ($/$) et la racine carrée

($\sqrt{}$). Ces opérations doivent être correctement arrondies. Autrement dit, si le résultat sur les réels ne correspond pas à un nombre flottant, ce résultat doit être arrondi au nombre flottant supérieur ou inférieur, suivant le mode (ou direction) d'arrondi choisi.

2.1.3.2 Mode d'arrondi

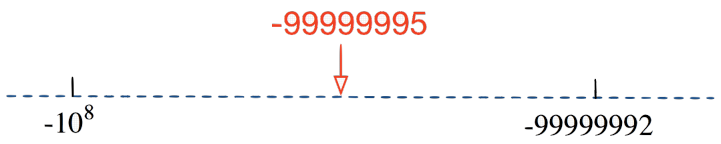
Le standard IEEE 754 définit 4 modes d'arrondi pour les nombres flottants.

- ◇ Vers $-\infty$: le nombre réel est arrondi vers le plus grand nombre flottant inférieur ou égal.
- ◇ Vers $+\infty$: le nombre réel est arrondi vers le plus petit nombre flottant supérieur ou égal.
- ◇ Vers 0 : ce mode d'arrondi correspond à une troncation. Si le nombre réel est positif, l'arrondi agit comme un arrondi vers $-\infty$. Si le nombre réel est négatif, il agit comme un arrondi vers $+\infty$.
- ◇ Au plus près : le nombre réel est arrondi au flottant le plus proche. Dans le cas où le nombre réel se trouve à équidistance de deux nombres flottants, deux variantes de ce mode d'arrondi existent :
  - pair : le nombre réel est arrondi au nombre flottant avec une mantisse paire.
  - extérieur : le nombre réel est arrondi au nombre flottant le plus grand en valeur absolue.

  Dans la suite, pour différencier ces deux variantes, le mode d'arrondi est suivi par sa variante. Par exemple, le premier est notée mode d'arrondi "au plus près pair".

Exemple:

Le nombre réel -99999995 n'étant pas représentable sur les nombres flottants en simple précision, un arrondi est donc nécessaire. La figure ci-dessous explicite le flottant le plus grand (-99999992, et le nombre flottant le plus petit (10^8) qui encadrent ce nombre réel.



Ce nombre réel est arrondi à :

- -10^8 avec un arrondi vers $-\infty$.
- -99999992 avec un arrondi vers $+\infty$, un arrondi vers 0, ou un arrondi "au plus près".

2.1.3.3 Erreurs

Les erreurs d'arrondi s'accumulent au cours des opérations. Les erreurs d'arrondi accumulées peuvent entrainer des comportements différents entre le comportement d'un programme avec la sémantique des réels, et l'execution du même programme sur les flottants. Il est donc important de quantifier ces erreurs pour pouvoir évaluer la déviation possible du programme. L'erreur *absolue* et l'erreur *relative* permettent de quantifier la différence entre la valeur d'une expression sur les réels et l'évaluation de la même expression sur les flottants.

L'erreur absolue (notée ε_a) caractérise la distance entre l'évaluation du résultat sur les réels et l'évaluation du résultat sur les flottants.

L'erreur relative (notée ε_r) est le rapport entre l'erreur absolue et le résultat sur les réels.

Une expression (notée *expr*) sur les nombres flottants peut être de plusieurs formes : opération arithmétique, variable ou constante. Chacune de ces formes peut être porteuse d'erreurs. Soit *expr* :

$$expr := expr \odot expr, \text{avec } \odot \in \oplus, \ominus, \otimes, \oslash \tag{2.5}$$

$$:= constante \tag{2.6}$$

L'erreur absolue d'une expression est définie de la manière suivante :

- $\varepsilon_a(cst) = |cst_{\mathbb{R}} - cst_{\mathbb{F}}|$
- $\varepsilon_a(expr) = |expr_{\mathbb{R}} - expr_{\mathbb{F}}|$

Exemple:

Soit l'équation :

$$z_{\mathbb{F}} = a_{\mathbb{F}} \oplus b_{\mathbb{F}}$$

Pour simplifier, $z_{\mathbb{F}}$, $a_{\mathbb{F}}$ et $b_{\mathbb{F}}$ sont considérés comme des nombres flottants décimaux (base 10) dont la précision est de 3 chiffres significatifs.

Comme les valeurs de $a_{\mathbb{F}}$ et $b_{\mathbb{F}}$ sont exactes, $\varepsilon_a(a_{\mathbb{F}}) = 0$ et $\varepsilon_a(b_{\mathbb{F}}) = 0$.

Avec $a = 3.37$, et $b = 7.18$, le résultat de z sur les réels est 10.55. Le résultat de $z_{\mathbb{F}}$ est 10.6. L'erreur absolue $\varepsilon_a(z)$ est donc 0.05.

De la même manière, **l'erreur relative** de $expr_{\mathbb{F}}$ est définie par :

- $\varepsilon_r(cst) = \frac{\varepsilon_a(cst)}{cst_{\mathbb{R}}}$
- $\varepsilon_r(expr) = \frac{\varepsilon_a(expr)}{expr_{\mathbb{R}}}$

Cas particulier :

- $expr_{\mathbb{R}} = 0$ et $\varepsilon_a(expr_{\mathbb{F}}) = 0 \Rightarrow \varepsilon_r(expr_{\mathbb{F}}) = 0$
- $expr_{\mathbb{R}} = 0$ et $\varepsilon_a(expr_{\mathbb{F}}) \neq 0 \Rightarrow \varepsilon_r(expr_{\mathbb{F}}) = +\infty$

Exemple:

Avec les mêmes données que l'exemple 2.1.3.3, l'erreur relative est $\varepsilon_r(z_{\mathbb{F}}) \approx 0.0047$.

2.1.4 Conséquences des arrondi

Les propriétés mathématiques du calcul sur les nombres réels ne sont pas forcément conservées sur les flottants.

Par exemple :

- L'addition et la multiplication sont rarement associatives. En général :

$$(a \oplus b) \oplus c \neq a \oplus (b \oplus c) \tag{2.7}$$

> ***Exemple:***
> Avec a = 0.3, b = 0.2, c = 0.1 sur les flottants double précision et un arrondi "au plus près"
>
> $$(a \oplus b) \oplus c = 6.0000000000000009e-01$$
> $$a \oplus (b \oplus c) = 5.9999999999999998e-01$$

$$(a \otimes b) \otimes c \neq a \otimes (b \otimes c) \tag{2.8}$$

> ***Exemple:***
> Avec les mêmes valeurs
>
> $$(a \otimes b) \otimes c = 6.0000000000000010e-03$$
> $$a \otimes (b \otimes c) = 6.0000000000000001e-03$$

◇ La multiplication est rarement distributive. En général :

$$a \otimes (b \oplus c) \neq a \otimes b \oplus a \otimes c \tag{2.9}$$

> ***Exemple:***
> Avec les mêmes valeurs
>
> $$a \otimes (b \oplus c) = 9.0000000000000011e-02$$
> $$a \otimes b \oplus a \otimes c = 8.9999999999999997e-02$$

De plus, les arrondis peuvent entrainer des phénomènes pathologiques : **l'absorption** et **la cancellation**

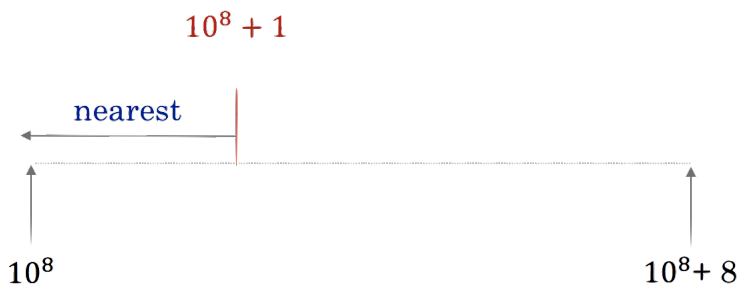
2.1.4.1 Absorption

L'absorption se produit lors de l'addition, ou la soustraction, de deux valeurs d'ordre de grandeurs très différents. Dans ce cas, la valeur la plus petite disparait et est dite "absorbée" par la seconde. Plus formellement, une absorption se produit pour l'opération [1] $x_{\mathbb{F}} \odot y_{\mathbb{F}} = z_{\mathbb{F}}$ avec $\odot \in \{\oplus, \ominus\}$ et $|x_{\mathbb{F}}| > |y_{\mathbb{F}}|$ ssi $|y_{\mathbb{F}}| \leq \frac{1}{2} ulp(x_{\mathbb{F}})$.

Exemple:

Soit l'opération d'addition suivante sur les nombres flottants simple précision et avec un mode d'arrondi "au plus près pair".

$$10^8 \oplus 1 = 10^8$$



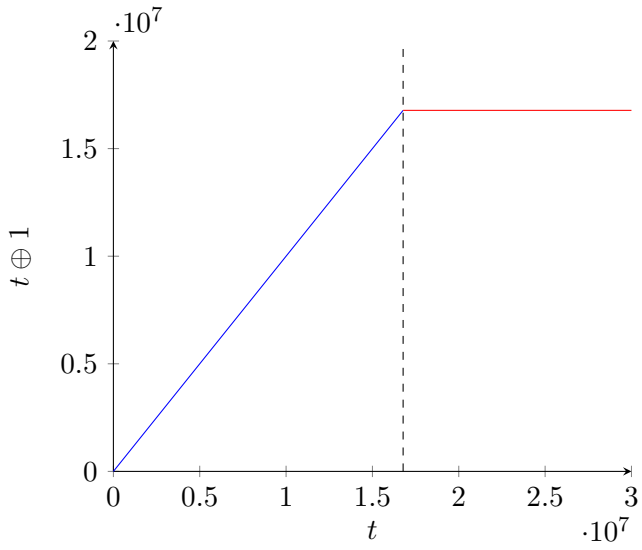
nearest est le mode d'arrondi utilisé.

La Figure ci-dessus illustre le phénomène d'absorption. Le successeur du flottant 10^8 est $10^8 \oplus 8$. Il est donc évident que lorsqu'une quantité inférieure à 8 est ajoutée à 10^8, le résultat est sujet à un arrondi et peu potentiellement entrainer un phénomène d'absorption.
Dans cet exemple, avec un mode d'arrondi "au plus près pair", $10^8 \oplus 1$ est arrondi à 10^8, la valeur 1 est donc absorbée par 10^8.

Si cette opération est répétée un grand nombre de fois, l'erreur peut devenir catastrophique. Par exemple, imaginons un programme évaluant $t = t \oplus 1.0$ toutes les secondes. La courbe ci-dessous montre l'évolution de la valeur de t au cours des itérations. Après la 16777216^{eme} itération,

1. avec un mode d'arrondi "au plus près pair"

la valeur 1.0 est absorbée à chaque appel. La valeur calculée par ce programme n'a donc plus aucun sens.



2.1.4.2 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 [Goldberg, 1991] (ou lors de l'addition de flottants de magnitude très proches mais de signes opposés). Il y a deux types de cancellation : la *bénigne* et la *catastrophique*.

Une cancellation est **bénigne** lorsque les opérandes sont connues exactement (pas d'erreur d'arrondi). Si ces dernières sont proches, la soustraction qui est dans ce cas exacte [Sterbenz, 1973], va engendrer une perte de chiffres significatifs.

Exemple:

Soit l'équation :

$$z_{\mathbb{F}} = x_{\mathbb{F}} \ominus y_{\mathbb{F}}$$

où les valeurs de $x_{\mathbb{F}}$, $y_{\mathbb{F}}$ et $z_{\mathbb{F}}$ sont considérées comme des nombres flottants décimaux (base 10) dont la précision est de 3 chiffres significatifs.

Si les valeurs de $x_{\mathbb{F}}$ et $y_{\mathbb{F}}$ sont respectivement 1.23 et 1.22, il n'y a pas d'erreurs d'arrondi car ils sont représentables exactement.

Cette soustraction est exacte mais elle entraine une perte de chiffre significatif. Comme ni $x_{\mathbb{F}}$, ni $y_{\mathbb{F}}$ ne sont porteurs d'erreurs, cette cancellation est bénigne.

Une cancellation catastrophique apparait lorsque les opérandes sont porteuses d'erreurs d'arrondi accumulées lors de calculs précédents. Les opérandes étant proches, la soustraction est alors exacte [Sterbenz, 1973]. Le résultat a perdu les chiffres significatifs et il ne reste alors que les erreurs. Si ce résultat est utilisé pour calculer d'autres valeurs, les erreurs de calculs peuvent-être amplifiée.

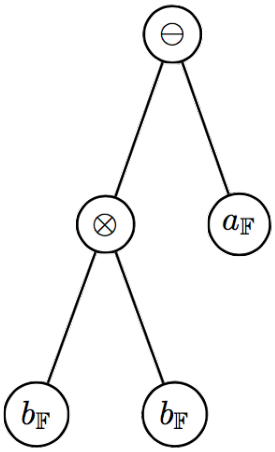
Exemple:

Soit l'équation :

$$z_{\mathbb{F}} = b_{\mathbb{F}}^2 \ominus a_{\mathbb{F}} \tag{2.10}$$

Supposons que les valeurs de $z_{\mathbb{F}}$, $a_{\mathbb{F}}$, $b_{\mathbb{F}}$ sont des nombres flottants décimaux (base 10) dont la précision est de 3 chiffres significatifs.

La figure ci-dessous représente l'arbre syntaxique de cette équation sur les flottants.



Supposons que les valeurs de $a_{\mathbb{F}}$ et $b_{\mathbb{F}}$ sont $a_{\mathbb{F}} = 11.1$ et $b_{\mathbb{F}} = 3.34$. Ces valeurs étant représentables exactement, leurs erreurs sont donc nulles.

$b_{\mathbb{F}}^2$ vaut 11.1556 mais est arrondi à 11.2 qui est proche de $a_{\mathbb{F}}$ et le résultat de la soustraction est 0.1. Les deux chiffres significatifs de poids les plus élevés ont été éliminés. Cette cancellation est catastrophique, car $b_{\mathbb{F}}^2$ est porteur d'une erreur de 0.0444 qui représente 44% du résultat.

Exemple:

Le polynôme de Rump [Rump, 1988], est connu pour produire une cancellation catastrophique. Cette cancellation catastrophique entraine une déviation importante entre le résultat sur les réels et le résultat sur les flottants. Ce polynôme est défini par :

$$Rump(a,b) = 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b}$$

Sur les nombres flottants binaires simples précision avec un mode d'arrondi au plus près, le résultat de $Rump(77617, 33096)$ est environ 6.33825×10^{29} alors que le résultat sur les réels est environ -0.827396. La cancellation catastrophique intervient lors de l'addition de résultat intermédiaire $op13$ et $op7$ (ligne 14 de la Table 2.2). Les Formules 2.11

et 2.12, expriment ces résultats intermédiaires en fonction des valeurs initiales du polynôme.

$$op13 = 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) \tag{2.11}$$

$$op7 = 5.5b^8 \tag{2.12}$$

Pour les valeurs d'entrée $a_{\mathbb{F}} = 77617$ et $b_{\mathbb{F}} = 33096$, les valeurs $op13$ et $op7$ sur les réels sont :

$$op13_{\mathbb{R}} = -7917111340668961361101134701524942850$$

$$op7_{\mathbb{R}} = 7917111340668961361101134701524942848$$

	Opération	valeur sur $\mathbb{F}$	valeur sur $\mathbb{R}$	ε_a	ε_r
1	$op1 = 333.75^6$	4.38606E+29	4.38606E+29	1.67383E+22	3.81625E-08
2	$op2 = a^2$	6024398848	6024398689	159	2.63927E-08
3	$op3 = 11a^2b^2$	7.25868E+19	7.25868E+19	9.1795E+11	1.26462E-08
4	$op4 = b^6$	1.31417E+27	1.31417E+27	1.52361E+20	1.15937E-07
5	$op5 = 121b^4$	1.45174E+20	1.45174E+20	1.8118E+12	1.24803E-08
6	$op6 = 2$	2	2	0	0
7	$op7 = 5.5b^8$	7.91711E+36	7.91711E+36	4.83056E+29	6.10142E-08
8	$op8 = a/2b$	1.172603965	1.17260394	2.47524E-08	2.11089E-08
9	$op9 = op3 - op4$	-1.31417E+27	1.31417E+27	7.97745E+19	6.07031E-08
10	$op10 = op9 - op5$	-1.31417E+27	-1.31417E+27	7.73741E+19	5.88765E-08
11	$op11 = op10 - op6$	-1.31417E+27	-1.31417E+27	7.73741E+19	5.88765E-08
12	$op12 = op2 \times op11$	-7.91711E+36	-7.91711E+36	4.44507E+28	5.61451E-09
13	$op13 = op1 + op12$	-7.91711E+36	-7.91711E+36	1.50769E+29	1.90434E-08
14	**op14 = op13 + op7**	**6.33825E+29**	**-2**	**6.33825E+29**	**3.16913E+29**
15	$op15 = op14 + op8$	6.33825E+29	-0.82739606	6.33825E+29	7.66048E+29

TABLE 2.2 – Détails des erreurs du polynôme de Rump

Exemple : (suite)

Sur les nombres flottants les derniers bits sont contaminés. L'addition de ces deux valeurs ne serait pas problématique si $op13_{\mathbb{F}}$ et $op7_{\mathbb{F}}$ étaient connues exactement. Mais, $op13_{\mathbb{F}}$ et $op7_{\mathbb{F}}$ ont accumulés énormément d'erreurs lors des calculs précédents. L'erreur absolue de $op13_{\mathbb{F}}$ est de l'ordre de 1.50×10^{29} et celle de $op7_{\mathbb{F}}$ est de l'ordre de 4.8×10^{29}. Néanmoins

leurs valeurs restent proches. Les signes étant opposés, l'addition agit comme une soustraction. Les valeurs de $op13_{\mathbb{F}}$ et $op7_{\mathbb{F}}$ correspondent sur les 9 premières décimales et sont donc éliminées, lors de la soustraction. La magnitude des erreurs est très grande par rapport au résultat de la soustraction de $op13_{\mathbb{R}}$ et $op7_{\mathbb{R}}$. Le résultat est donc essentiellement la somme des erreurs.

Il est à noter que la soustraction étant exacte, elle permet seulement de mettre en avant les accumulations d'erreurs des calculs précédents.

Les erreurs absolues et relatives pour chaque opération de ce polynôme sont détaillées dans la Table 2.2.

2.2 Conclusion

Dans ce chapitre, nous avons présenté les nombres à virgule flottante et les spécificités de l'arithmétique des nombres à virgule flottante. Cette dernière est différente de l'arithmétique sur les nombres réels. Certaines opérations ne sont par exemple ni associatives ni distributives. L'utilisation d'arrondi et la perte de certaines propriétés arithmétiques peuvent entrainer des phénomènes d'absorption et de cancellation. Ces phénomènes sont souvent la source de bugs dans de nombreux programmes.
On trouvera une description plus détaillée des notions présentées dans ce chapitre dans [Goldberg, 1991, IEEE, 2008, Einarsson, 2005].

Chapitre 3

Programmation par contraintes

La programmation par contraintes est un paradigme utilisé pour résoudre des problèmes combinatoires difficiles tels que les problèmes de planification ou d'ordonnancement [Bartk, 1970]. Ce paradigme repose sur une séparation claire entre la modélisation et la résolution du problème. La modélisation formalise le problème de manière déclarative, alors que la résolution se concentre sur des techniques d'exploration de l'espace de recherche.

3.1 Définition d'un CSP

En programmation par contraintes, un problème est modélisé par un CSP [1].

Un CSP est un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, où :

- $\mathcal{X} = \{x_1, \ldots, x_n\}$ est l'ensemble des variables du problème.
- $\mathcal{D} = \{D_1, \ldots, D_n\}$ est l'ensemble des domaines des variables D_i ou est le domaine de la variable $x_i \in \mathcal{X}$. D_i est soit un ensemble fini de valeurs énumérées $\{v_1, v_2, \ldots, v_k\}$, soit l'intervalle $[\underline{x}_i, \overline{x}_i] = \{x \in \mathbb{Z} \mid \underline{x}_i \leq x \leq \overline{x}_i\}$, où $\underline{x}_i$ et $\overline{x}_i$ désignent respectivement la borne inférieure et supérieure.
- $\mathcal{C} = \{c_1, \ldots, c_n\}$ est l'ensemble des contraintes. Chaque contrainte $c_i \in C$ représente une relation entre les variables.

Soit $c \in \mathcal{C}$, et $x \in \mathcal{X}$, $Vars(c)$ et $Contraintes(x)$ dénotent respectivement l'ensemble des variables de la contrainte c, et l'ensemble des contraintes impliquant la variable x.

Une solution d'un CSP est une affectation des variables qui vérifie toutes les contraintes.

1. Constraints Satisfaction Problem (Problème de satisfaction de contraintes)

Exemple:
Soit le CSP : $\mathcal{X} = \{x_0, x_1, x_2\}$, $\mathcal{D} = \{D_0, D_1, D_2\}$ avec $D_0 = \{1, 2, 3\}$, $D_1 = D_2 = \{2, 3\}$ et $\mathcal{C} = \{x_0 \neq x_1, x_1 \neq x_2, x_2 \neq x_0\}$
Une solution est $x_0 = 1$, $x_1 = 3$ et $x_2 = 2$.

Plus généralement, un CSP est consistant lorsque pour toutes les valeurs des domaines des variables, il existe une instanciation des variables pour lesquelles toutes les contraintes sont vraies. Plus formellement, un CSP$(\mathcal{X}, \mathcal{D}, \mathcal{C})$ est consistant ssi pour toute variable $x_i \in \mathcal{X}$:

$$\forall c \in \mathcal{C}, \text{ une contrainte k-aire}, \forall v_i \in D_i, \exists v_1 \in D_1, \ldots, \exists v_{i-1} \in D_{i-1}, \exists v_{i+1} \in D_{i+1}, \ldots, \exists v_k \in D_k \text{ tel que } c(v_1, \ldots, v_i, \ldots, v_k)$$

3.2 Modélisation

En PPC, la modélisation consiste à représenter le problème sous forme d'un CSP. Idéalement, cette partie doit se faire sans se soucier de la résolution. Toutefois comme le montre l'exemple ci-dessous, le choix de modélisation peut avoir un impact important sur les temps de résolution.

3.2.1 Exemple : les n-reines

Dans cette section, nous proposons différents modèles du problème des n-reines et nous mettons en évidence l'impact du modèle sur la résolution.

Le problème consiste à placer n reines sur un échiquier $n \times n$ de telle sorte qu'aucune reine ne puisse s'attaquer mutuellement.

La Figure 3.1 présente deux solutions des n-reines pour $N = 4$. La Figure 3.2 présente un placement qui n'est pas solution de ce problème (la reine en b4 attaque celle en c3).

Représentation sous la forme d'un CSP
Plusieurs modèles permettent de représenter ce problème sous la forme d'un CSP. Selon la représentation des variables choisies, il est plus ou moins aisé

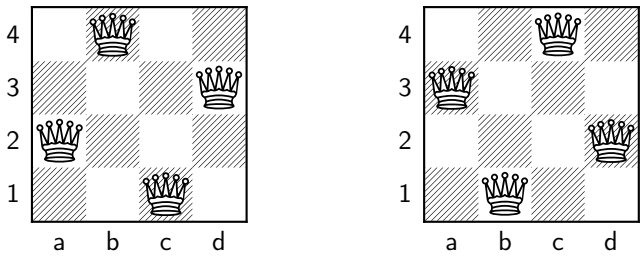


FIGURE 3.1 – N-reines : exemples de solutions

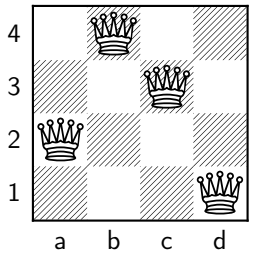


FIGURE 3.2 – N-reines : exemple de non solution

d'exprimer les contraintes du problème. Ceci provient du fait que certaines représentations satisfont de fait des contraintes. Ces contraintes n'ont alors plus besoin d'être explicitées.

Quels sont les variables et leurs domaines ?

Une première représentation naïve consiste à associer une variable à chaque reine. Les domaines de ces variables sont $[1, n * n]$ (par exemple, avec un échiquier de taille 8, le domaine est [1,64]). Les contraintes exprimant que deux reines ne sont pas en prise (lignes, colonnes, et diagonales différentes) sont assez fastidieuses à exprimer dans une telle représentation.

Un modèle plus simple, et plus efficace est d'associer une variable à chaque ligne. La valeur prise par cette variable est la position de l'unique reine qui peut être mise sur cette ligne. Plus formellement, pour un échiquier de taille n, on a n variables X_i avec $i \in [1, n]$. Par exemple, dans la Figure 3.2, les valeurs de i et X_1 sont : $X_1 = 4, X_2 = 1, X_3 = 3, X_4 = 2$.

Quelles sont les contraintes ?
Les contraintes de ce problème sont :

1. Les reines doivent être sur des lignes différentes.
2. Les reines doivent être sur des colonnes différentes.
3. Les reines doivent être sur des diagonales différentes.

Les contraintes 2 et 3 se représentent de la manière suivantes :

2. $X_i \neq X_j$, avec $i, j \in [1, n]$ et $i \neq j$
3. $X_i \neq X_j \pm (i - j)$ avec $i \neq j$ et $i, j \in [1, n]$.

Quant à la première contrainte, elle est obligatoirement satisfaite du fait de la modélisation choisie.

3.3 Résolution des CSP

En général, la programmation par contraintes est utilisée pour résoudre des problèmes difficiles (NP-complet) pour lesquels il n'existe pas d'algorithmes de résolution polynomiaux. La résolution d'un CSP, repose sur deux mécanismes principaux : **le filtrage** et **l'exploration** de l'arbre de recherche. Les algorithmes de filtrages (polynomiaux) réduisent l'espace de recherche. L'exploration cherche quant à elle a réduire le problème initial en sous-problèmes plus simples à résoudre. Ces mécanismes sont appelés l'un après l'autre jusqu'à trouver une solution, ou explorer la totalité de l'espace de recherche. L'Algorithme 1 positionne l'ensemble de ces mécanismes utilisés dans la résolution d'un CSP.

Dans cet algorithme les procédures :

- ◇ *reviser(c,x,y,D)* retourne $VRAI$ si le filtrage de la contrainte $c(x, y)$ a réduit le domaine D_x de la variable x_i.
- ◇ *sauvegarderEtat(D)* sauve l'état actuel des domaines des variables. Pour sauvegarder l'état, cette procédure empile une copy des domaines. À contrarion, la fonction *restaurerEtat(D)*, restaure l'état des domaines des variables à celui de la dernière sauvegarde. Cette fonction retourne $VRAI$, si la restauration a réussi (i.e. si la pile des domaines sauvegardés est non vide avant l'appel à cette fonction), $FAUX$ sinon (i.e. si la pile des domaines est vide).
- ◇ *echec($\mathcal{D}$)* returne $VRAI$ si $\mathcal{D}$ correspond à un échec, autrement dit si tous les domaines des variables $\mathcal{D}$ sont vide, $FAUX$ sinon. De la même manière *solution($\mathcal{D}$)* retourne $VRAI$, si le domaine $\mathcal{D}$ correspond à une solution, autrement dit, s'il existe exactement une valeur dans chaque domaine de chaque variable, et que ces valeurs sont consistantes.

Algorithm 1 Algorithme abstrait de résolution d'un CSP

```
1: procedure FILTRAGE(𝒳,𝒟,𝒞,Q)
2:     while Q ≠ ∅ do
3:         selectionner et supprimer c ∈ Q
4:         {x_i, x_j} ← vars(c)
5:         if reviser(c,x_i,x_j,𝒟) then
6:             Q ← Q ∪ contraintes(x_i)
7:         end if
8:     end while
9: end procedure
10:
11: procedure EXPLORATION(𝒳,𝒟,𝒞,Q)
12:     x ← choixDeVariable(𝒳)
13:     a ← choixDeValeur(𝒟_x)
14:     𝒟_x ← 𝒟_x\{a}
15:     sauvegarderEtat(𝒟)
16:     𝒟_x ← {a}
17:     Q ← Q ∪ contraintes(x)
18: end procedure
19:
20: function RETOURARRIERE(𝒟,𝒞,Q)
21:     if restaurerEtat(𝒟) then
22:         Q ← Q ∪ contraintes(x)
23:         return VRAI
24:     end if
25:     return FAUX
26: end function
27:
28: function RESOLUTION(𝒳,𝒟,𝒞)
29:     S ← {}                                  ▷ Ensemble des solutions
30:     Q ← 𝒞          ▷ Contient l'ensemble des contraintes à filtrer
31:     fin ← FAUX
32:     while ¬fin do
33:         Filtrage(𝒳,𝒟,𝒞,Q)
34:         if solution(𝒟) then S ← S ∪ 𝒟 end if
35:         if echec(𝒟) ou solution(𝒟) then
36:             if retourArriere(𝒟,𝒞,Q) then
37:                 exploration(𝒳,𝒟,𝒞,Q)
38:             else
39:                 fin ← VRAI
40:             end if
41:         end if
42:     end while
43:     return S
44: end function
```

3.3.1 Filtrage et propagation

Le filtrage consiste à considérer les contraintes une à une en supprimant les valeurs qui ne peuvent trivialement pas faire partie de la solution (ligne 3 de l'Algorithme 1). Ces valeurs sont dites inconsistantes. Les réductions effectuées en ne considérant qu'une contrainte à la fois étant locales, beaucoup de valeurs non-solution ne sont pas supprimées. Ces réductions ont besoin d'être **propagées** aux autres contraintes pour permettre de supprimer davantage de valeurs non-solution et atteindre un état ou le CSP est consistant. Seules les contraintes où les variables dont le domaine a été réduit sont donc reconsidérées (ligne 5 de l'Algorithme 1). Les contraintes n'impliquant que les variables dont les domaines n'ont pas été impactés sont ignorées. Cette étape clé du filtrage qui consiste à propager les réductions est appelée propagation. La propagation est effectuée jusqu'à ce qu'aucune autre réduction ne soit possible (ligne 2 de l'Algorithme 1). Le point fixe est alors atteint. Plus précisément, l'algorithme de filtrage étant monotone, c'est le point fixe atteint est le plus grand point fixe.

Un filtrage initial est effectué au début de la résolution (lignes 30 et 33 de l'Algorithme 1). Il consiste à considérer toutes les contraintes au moins une fois. En général, ce filtrage initial effectue peu de réduction d'où l'intérêt de réduire le problème initial en problème plus simple à résoudre à l'aide de la recherche de solutions (ligne 37 de l'Algorithme 1). La recherche de solutions consiste à choisir une variable (ligne 12) et une valeur (ligne 13), puis d'instancier la variable choisie à la valeur choisie (ligne 16). Cette instanciation va permettre de réduire à nouveau les domaines des autres variables lors du filtrage. Ces deux mécanismes vont être appelés l'un après l'autre jusqu'à trouver une solution, où explorer la totalité de l'arbre de recherche.

Dans la suite, les concepts présentés dans cette sous section (e.g., filtrage et point fixe) sont formalisés. Ces formalisations s'inspirent de [Michel et al., 2018].

un algorithme de filtrage peut-être défini comme étant la fonction F suivante :

$$F(\mathcal{D}) = \mathcal{D}' \subseteq \mathcal{D} \wedge S(\mathcal{X}, \mathcal{D}', \mathcal{C}) = S(\mathcal{X}, \mathcal{D}, \mathcal{C})$$

où $S(\mathcal{X}, \mathcal{D}, \mathcal{C})$ représente l'ensemble des solutions du CSP défini par le triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$.

Lorsque $D' = \emptyset$, le CSP n'a pas de solution. Si chaque domaine $D_i \in D'$ contient exactement une valeur dans D_i, alors D' est une solution.

Le point fixe $\mathcal{D}$ est un point fixe $\iff F(\mathcal{D}) = \mathcal{D}$

Le plus grand point fixe $\mathcal{D}^*$ est le plus grand point fixe $\iff \mathcal{D}^* = max\{\mathcal{D}^p \subseteq \mathcal{D} : \mathcal{D}^p = F(\mathcal{D})\}$

Ces différents concepts sont illustrés dans l'exemple suivant.

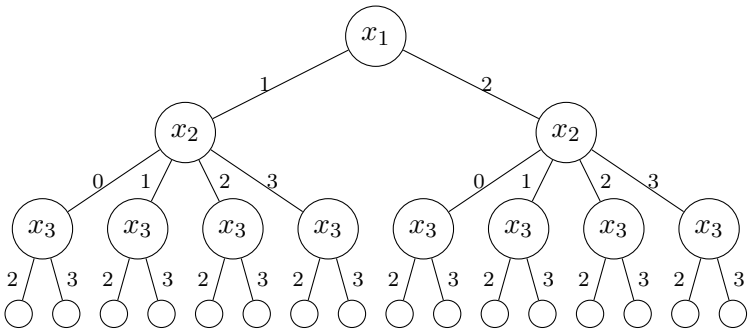
Exemple:

Soit le CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ défini par :

- $\mathcal{X} = \{x_1, x_2, x_3\}$
- $x_1 \in [1, 2], x_2 \in [1, 3], x_3 \in [2, 3]$
- Les contraintes :

$$\begin{aligned} x_1 > x_2 \quad & (c_1) \\ x_1 + x_2 = x_3 \quad & (c_2) \\ x_2 \neq x_3 \quad & (c_3) \\ x_3 \neq x_1 \quad & (c_4) \end{aligned}$$

L'arbre ci-dessous explicite les combinaisons possibles des valeurs des variables pour ce CSP. Le filtrage et la propagation vont permettre d'effectuer des coupes dans cet arbre.



Filtrage et propagation :

Le filtrage initial implique de filtrer au moins une fois la totalité des contraintes. Dans la suite de l'exemple, Q représente l'ensemble des contraintes à filtrer.

$Q = \{c_1, c_2, c_3, c_4\}$

La contrainte c_1 réduit le domaine de x_2. Les domaines deviennent : $x_1 \in [2], x_2 \in [1], x_3 \in [2, 3]$. Toutes les contraintes impliquant x_1 et x_2, sauf $c1$, sont ajoutées dans Q.

$Q = \{c_2, c_3, c_4\}$

La contrainte c_2 permet de réduire le domaine de x_3 à [3]. Toutes les contraintes impliquant x_3, sauf c_2, sont ajoutées dans Q.

$Q = \{c_3, c_4\}$

Les contraintes présentent dans Q n'effectue aucune réduction supplémentaire.

$Q = \{\}$

Toutes les variables sont affectées à une valeur et Q est vide, une solution est donc obtenue. Ici, aucune exploration n'est nécessaire pour permettre d'obtenir une solution.

3.3.1.1 Consistances locales

Les réductions effectuées par le filtrage reposent sur des propriétés de consistance locale. Une consistance locale est une propriété caractérisant une sous-partie du problème.

Sur les domaines finis, la consistance locale communément utilisée est la consistance d'arc (AC). Plus formellement, un CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ est dit arc-consistant ssi :

$$\forall c \in \mathcal{C}, \text{ une contrainte k-aire}, \forall x_i \in Vars(c), \forall v_i \in D_i, \exists v_1 \in D_1, \ldots, \exists v_{i-1} \in D_{i-1}, \exists v_{i+1} \in D_{i+1}, \ldots, \exists v_k \in D_k \; c(v_1, \ldots, v_{i-1}, v_i, v_{i+1} \ldots, v_k)$$

De nombreux algorithmes ont été proposés pour implanter la consistance d'arc [Mackworth and Freuder, 1985, Hentenryck et al., 1992, Mohr and

Henderson, 1986, Bessiére et al., 1999, Bessière, 1994, Perlin, 1992]. Ces algorithmes cherchent à éliminer, pour chaque contrainte, les valeurs qui violent cette contrainte. Ces valeurs sont dites inconsistantes.

Les Figures 3.3a et 3.3b illustrent l'idée de cette consistance. Dans ces figures x_i, x_j et x_k sont des variables. Les noeuds correspondent aux valeurs des domaines de chaque variable. Un arc lie les valeurs de x_i et x_j si ces valeurs sont compatibles. La Figure 3.3a représente le graphe de contraintes initiales. La Figure 3.3b représente le même graphe après application de la consistance d'arc. Dans cette figure, les noeuds grisés sont les valeurs supprimées par la consistance.

(a) Avant réduction (b) Après réduction

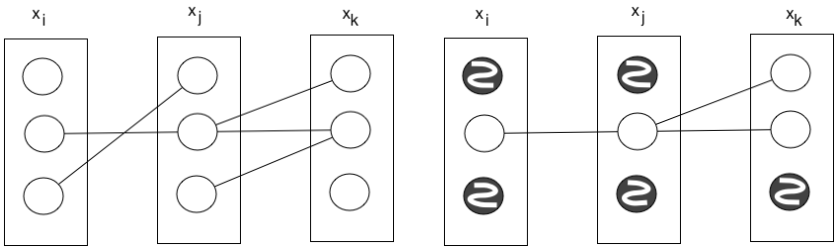


FIGURE 3.3 – Consistance d'arc sur un problème à deux contraintes

Différents algorithmes ont été proposés pour le calcul de la consistance d'arc. AC3 est l'un des algorithmes de consistance d'arc les plus utilisé.

AC3

L'Algorithme 2 présente l'algorithme AC3 [Mackworth, 1977]. L'idée de cet algorithme est de rechercher pour chaque valeur du domaine de x_i une valeur compatible aussi appelée support (ligne 3). Informellement, une valeur $v_i \in D_i$ est une valeur compatible pour la contrainte c s'il existe une instanciation valide des autres variables satisfaisant la contrainte c lorsque x_i est instancié à v_i. Si aucune valeur compatible n'est trouvée, la valeur est supprimée du domaine de x_i (ligne 11). Toutes les contraintes qui impliquent x_i sont alors réétudiées (ligne 22). Ce processus est effectué pour chaque contrainte. L'algorithme s'arrête lorsque le point fixe est atteint (ligne 19).

Limite de la consistance d'arc

La consistance d'arc permet de réduire la combinatoire en supprimant les valeurs trivialement inconsistantes. Néanmoins, ce raisonnement reste fondamentalement un raisonnement local. Ainsi, dans certains CSP trivialement

Algorithm 2 AC3

```
1: function REVISE(c :contrainte, x_i :variable, x_j :variable, D :domaines)
2:     change ← false
3:     for each v_i ∈ D(i) do
4:         del ← true
5:         for each v_j ∈ D(j) do
6:             if consistant(c, v_i,v_j) then
7:                 del ← false
8:             end if
9:         end for
10:        if del then
11:            supprimer v_i de D_i
12:            change ← true
13:        end if
14:    end for
15:    return change
16: end function
17:
18: procedure AC3(X,D,C)
19:    Q ← {c ∈ C}
20:    while Q ≠ ∅ do
21:        selectionner et supprimer c ∈ Q
22:        {x_i, x_j} ← vars(c)
23:        if revise(c,x_i ,x_j,D) then
24:            Q ← Q ∪ contraintes(x_i)
25:        end if
26:    end while
27: end procedure
```

inconsistants, la consistance d'arc est incapable d'effectuer une quelconque réduction.

Exemple:

Soit le CSP : $\mathcal{X} = \{x_1, x_2, x_3\}, \mathcal{D} = \{D_0, D_1, D_2\}$, avec $D_0 = D_1 = D_2 = [1, 2]$ et $\mathcal{C} = \{x_1 \neq x_2, x_2 \neq x_3, x_3 \neq x_1\}$

Les trois contraintes du CSP sont arc-consistantes. Toutes les valeurs des domaines de chaque variable valident individuellement chacune des contraintes. Mais il est évident qu'aucune solution n'existe pour ce CSP.

Les contraintes globales ont été définies pour détecter certaines de ces inconsistances.

3.3.1.2 Contraintes globales

Les contraintes globales ont été introduites pour combler certaines des limitations de la consistance locale. Elles permettent de réduire les domaines d'un ensemble de variables liés par des contraintes spécifiques ; par exemple, un ensemble de variables qui doivent prendre des valeurs différentes. Ces contraintes nécessitent des algorithmes polynomiaux qui permettent de raisonner sur un sous-ensemble de variables du problème.
Les contraintes globales permettent aussi d'avoir un gain en expressivité. Elles permettent de représenter un ensemble de contraintes qui parfois sont difficilement exprimables à partir des contraintes élémentaires.
La contrainte *alldifferent* [Régin, 1994] est un des exemples les plus connus de contrainte globale. Cette contrainte permet de modéliser une conjonction de contraintes d'inégalités. Par exemple, $alldifferrent(x_1, x_2, x_3)$ correspond aux conjonctions : $x_1 \neq x_2 \wedge x_2 \neq x_3 \wedge x_1 \neq x_3$.

[Régin, 1994] a proposé un filtrage efficace basé sur les algorithmes de couplages [2] (matching) dans un graphe biparti [3].
Le filtrage de cette contrainte consiste à calculer un couplage maximal dans le graphe biparti où les sommets de la première partition sont les variables et les valeurs des domaines la seconde. Une fois ce couplage calculé, il suffit de calculer les composantes fortement connexes, puis de supprimer les arcs non couplés qui relient ces composantes.

Ce filtrage raisonne sur un sous-ensemble de variables du problème qui sont fortement corrélées. Ce raisonnement permet de détecter certaines inconsistances comme celles de l'exemple ci-dessous.

Exemple:

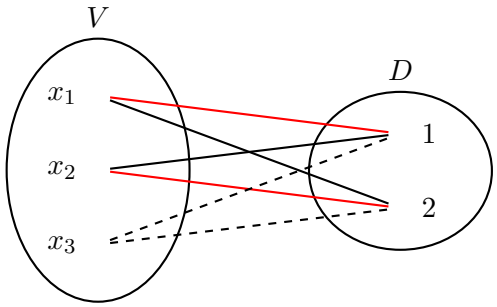
Soit le CSP : $\mathcal{X} = \{x_1, x_2, x_3\}, \mathcal{D} = \{D_0, D_1, D_2\}$, avec $D_0 = D_1 = D_2 = \{1, 2\}$ et $\mathcal{C} = \{alldifferent(x_1, x_2, x_3)\}$

La figure ci-dessous représente ce CSP sous la forme d'un problème de couplage. Les noeuds de l'ensemble V représentent les variables, et les noeuds de l'ensemble D les valeurs des domaines. Les arêtes appartenant à un couplage maximum sont en trait plein, celles qui sont supprimées

2. Un couplage dans un graphe est un ensemble d'arêtes qui ne sont pas adjacentes deux à deux.

3. Un graphe biparti est un graphe où l'ensemble des sommets peut-être partitionné en deux sous-ensembles X_1 et X_2 de telle sorte que toutes les arêtes qui sortent d'un sommet de X_1 arrivent dans X_2 et inversement.

par le filtrage sont en pointillés. À la suite du filtrage, le domaine de x_3 est vide.



3.3.2 Recherche de solutions ou exploration

Cette étape intervient généralement une fois le point fixe atteint. Elle consiste à affecter une valeur à une variable pour relancer de nouvelles réductions via la propagation. La variable et la valeur affectée sont choisies par des heuristiques (ou stratégies) de choix de variables et de choix de valeurs. Si ces choix conduisent à une inconsistance, un retour en arrière (backtrack) est effectué. L'exploration peut être vue comme la réduction du problème initial en sous-problème plus simple.

Les stratégies de recherche consistent à énumérer de manière exhaustive, mais intelligente, l'espace de recherche. Elle détermine l'ordre selon lequel les variables doivent être instanciées. Elles permettent très souvent d'accélérer la recherche de la première solution, et parfois de prouver plus rapidement l'absence de solutions.

3.3.2.1 Stratégie de choix de variables

Sur les domaines finis, les stratégies existantes de choix de variables embrassent souvent le principe *fail-first* énoncé par [Haralick and Elliott, 1980]. Ce principe est le suivant :"To succeed try first where you are most likely to fail". L'idée est d'échouer rapidement pour effectuer des coupes plus importantes dans l'arbre de recherche. Quelques unes des heuristiques les plus utilisées sont présentées dans la suite de cette section. D'autres travaux [Refalo, 2004, Boussemart et al., 2004, Bessiere and Régin, 1996, Gay et al., 2015] proposent également des stratégies de choix de variables efficaces.

Lexicographique Cette stratégie ordonne les variables de manière syntaxique. C'est un ordre statique.

min-domain Cette stratégie choisit la variable qui a le domaine le plus petit. L'intuition de cette stratégie est de sélectionner la variable qui contraint le plus le problème par la taille de son domaine.

degree Cette heuristique a été introduite dans [Brélaz, 1979]. Elle choisit la variable au degré [4] le plus fort. Comme *min-domain*, cette stratégie essaie de choisir la variable qui contraint le plus le problème.

Les choix de variable *min-domain* et *degree* sont des ordres sémantiques ; *degree* est un ordre statique alors que *min-domain* est un ordre dynamique vu que la taille des domaines des variables évolue lors de la propagation par filtrage des choix de valeurs effectués.

3.3.2.2 Stratégie de choix de valeurs

Les heuristiques de choix de valeurs déterminent la prochaine valeur à instancier. Contrairement aux heuristiques de choix de variables, elles suivent souvent le principe *succeed-first*. Ce principe consiste à choisir en premier la valeur qui a le plus de chance d'appartenir à une solution. L'intuition derrière ces principes contradictoire est de trouver une solution (*succeed-first*) tout en limitant la profondeur de l'arbre si cela s'avère être un échec (*fail-first*). Les heuristiques de choix de valeurs les plus communes sont :

min-valeur Cette heuristique choisit en premier la plus petite valeur du domaine.

max-valeur A contrario, cette heuristique choisit la valeur la plus grande du domaine.

Ces deux stratégies se concentrent sur les bornes des domaines. Les solveurs de contraintes évitent, en général, les "trous" dans les domaines. Lorsque les domaines sont assez petits, les trous sont simples à gérer. Néanmoins, quand les domaines sont grands, il est possible de gérer les trous sous forme d'unions de domaines.

4. Le degré d'une variable est le nombre de contraintes où la variable est impliquée

3.3.3 Reprise des n-reines

Cette section propose d'illustrer les principes de résolution énoncés précédemment pour trouver une solution au problème des 4-reines. La modélisation du problème utilisée est celle présentée dans la section 3.2.1. Avec ce choix de modélisation, le filtrage initial ne permet d'effectuer aucune réduction, il est donc nécessaire de rechercher une solution en réduisant le problème à un problème plus simple où une des variables est instanciée. Nous allons utiliser une heuristique de choix de variables basée sur l'ordre lexicographique. L'heuristique de choix de valeurs est un min-valeur.

Dans un souci de clarté, les colonnes sont représentées par des lettres et des lignes par des chiffres. Par exemple, $X_1 = d$, capture la position suivante : la reine sur la première ligne est sur la colonne d.

Le filtrage initial ne permet aucune réduction. L'appliquation de ces heuristiques de choix de variables et de valeurs génère l'échiquier de la Figure 3.4a est obtenu. L'état du CSP correspondant est :

(a) Choix initial (b) Filtrage (1)

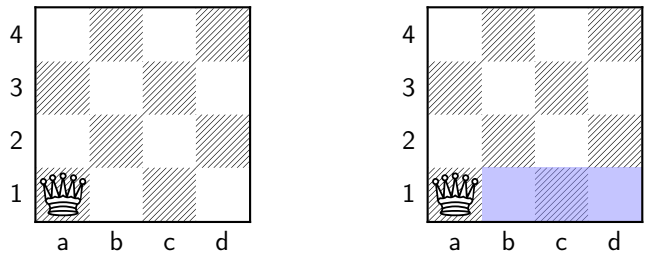


FIGURE 3.4 – Problème des N-reines

$X_1 = a, X_2 = [a, d], X_3 = [a, d], X_4 = [a, d]$

Comme expliqué dans la section modélisation, le modèle choisit implique déjà que les lignes des reines soient toutes différentes. La Figure 3.4b illustre cette contrainte. Les cases en bleu représentent les cases invalidées par cette contrainte implicite.

Il faut maintenant filtrer la contrainte représentant que les colonnes doivent être différentes : $X_i \neq X_j$, avec $i, j \in [1, 4]$ et $i \neq j$. Pour rappel, X_i représente la ligne où est positionnée la reine, et i la colonne.
Cette contrainte permet de supprimer la valeur a des variables X_2, X_3 et X_4. Le CSP devient donc $X_1 = a, X_2 = [b, d], X_3 = [b, d], X_4 = [b, d]$. La

Figure 3.5a présente l'impact de ce filtrage sur l'emplacement des reines. Les cases en rouge indiquent les cases rendues interdites par le filtrage.

(a) Filtrage (2) (b) Filtrage (3)

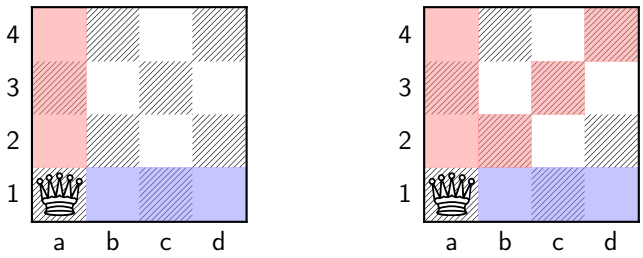


FIGURE 3.5 – Problème des N-reines

La dernière contrainte ($X_i \neq X_j \pm (i-j)$ avec $i \neq j$ et $i, j \in [1,4]$), conduit à l'état : $X_1 = a, X_2 = [c,d], X_3 = \{b,d\}, X_4 = [b,c]$ (voir Figure 3.5b). Comme les domaines ont été réduits, le filtrage reprend avec la première contrainte. Néanmoins, plus aucune réduction ne peut être effectuée. Le point fixe donc est atteint. L'exploration intervient, et la variable X_2 est fixée à la valeur b. La Figure 3.6a illustre ce choix.

(a) Exploration (2) (b) Exploration (3)

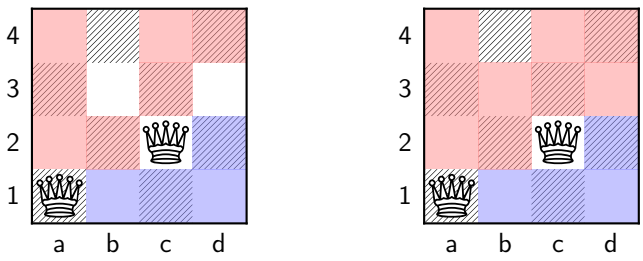


FIGURE 3.6 – Problème des N-reines

Une fois ce choix propagé, la Figure 3.6b est obtenue. Le domaine de la variable représentant la 3^e lgine étant vide, l'état obtenu est inconsistant. Un retour arrière donc est effectué. Le cas $X_2 \neq b$ est alors exploré.

Le même processus est effectué jusqu'à trouver une solution. La Figure 3.7 représente la première solution trouvée par cette recherche.

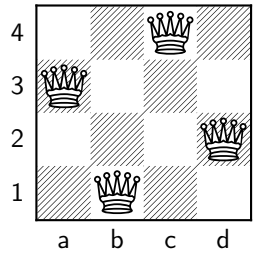


FIGURE 3.7 – N-reines : première solution trouvée par cette heuristique

3.4 PPC sur les domaines continus

Les CSPs sur les domaines continus sont définis de manière analogue à ceux des domaines finis. Seule la représentation des domaines des variables change. Dans les CSPs sur les domaines continus, les domaines des variables sont des intervalles de réels. Le domaine de la variable $x_{\mathbb{R}}$ est représenté par l'intervalle $\mathbf{x}_{\mathbb{R}} = \{x \in \mathbb{R} |\ \underline{x}_{\mathbb{R}} \leq x \leq \overline{x}_{\mathbb{R}}\}$.

Les techniques de filtrage et de recherche introduites sur les domaines finis ne sont pas adaptées aux domaines continus. En effet, contrairement aux domaines finis, les domaines continus représentent un nombre infini de valeurs ; des techniques se basant uniquement sur l'énumération sont donc inutilisable. Un ensemble de consistances [Lhomme, 1993, Benhamou et al., 1995, Freuder, 1978] plus adaptées aux spécificités de ces domaines a été proposé.

3.4.1 Consistances sur les domaines continus

Les consistances introduites pour les domaines continus se basent en général sur l'arithmétique des intervalles [Moore, 1966].

2B-consistance (ou hull-consistance)
Cette consistance a été introduite dans [Lhomme, 1993]. L'objectif de la 2B-consistance est de renforcer la consistance aux bornes. Elle peut être vue comme une relaxation de la consistance d'arc aux bornes de l'intervalle. En d'autres termes, la contrainte c est dite 2B-consistant, pour la variable x, s'il existe des valeurs dans les domaines des autres variables qui satisfont c lorsque x est instancié avec $\underline{x}$ et lorsque x est instancié avec $\overline{x}$. Le CSP est 2B-consistant si toutes les contraintes sont 2B-consistantes.

Plus formellement, soit $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ un CSP et $c \in \mathcal{C}$ une contrainte $k-aire$ sur les variables $(x_1, \ldots, x_k)$. La contrainte c est dite 2B-consistante si :

$$\begin{aligned}\forall i, D_i = \square\{v_i \in D_i \mid \exists v_1 \in D_1, \ldots, \exists v_{i-1} \in D_{i-1},\\ \exists v_{i+1} \in D_{i+1}, \ldots, \exists v_k \in D_k,\\ \text{tel que } c(v_1, \ldots, v_{i-1}, x_i, v_{i+1}, \ldots, v_k)\}\end{aligned}$$

où $\square S$ est le plus petit intervalle I tel que $S \subseteq I$

Cette consistance est réalisée a l'aide d'un ensemble de fonctions de projections. Dans le cas général, calculer les projections des contraintes est impossible. C'est pourquoi, les filtrages comme la 2B-consistance reposent sur une décomposition des contraintes en contraintes élémentaires binaires ou ternaires pour lesquelles, ces projections sont faciles à calculer

L'exemple ci-dessous illustre la 2B-consistance sur les domaines continus.

Exemple:

$$\begin{gathered}\text{Variables : } x_0, x_1\\ \text{Domaines : } D_0 = [1, 4]\ D_1 = [-2, 2]\\ \text{Contrainte : } x_0 = x_1^2\end{gathered}$$

Le CSP ci-dessous est 2B-consistant, car les bornes du domaine D_0 de la variable x_0 ont des valeurs compatibles dans le domaine de la variable x_1 pour la contrainte $x_0 = x_1^2$. La variable x_2 est aussi 2B-consistant pour les mêmes raisons. Ce CSP n'est pas arc-consistant car la valeur 0 du domaine de x_1 n'a pas de valeur compatible dans le domaine de x_0.

2B(w)-consistance
La 2B(w)-consistance est une relaxation de la 2B-consistance. Lors de l'utilisation de la 2B-consistance, une modification des bornes même d'un seul flottant entraine la propagation de cette modification. En pratique, cette propagation systématique peut être couteuse et conduire à des phénomènes de convergence lente. La 2B(w)-consistance permet de relâcher cette condition sur la propagation des modifications. Une modification est propagée seulement lorsqu'aucune valeur compatible n'existe dans l'intervalle $[\underline{x}, \underline{x}^{+^{(w)}}]$, où $\underline{x}^{+^{(w)}}$ (resp. $\underline{x}^{-^{(w)}}$) dénote le w^{ime} flottants après (resp. avant) $\underline{x}$. Plus formellement, Soit $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, $x \in X$, $w \in \mathbb{N}+$, D_x est dit 2B(w)-consistant si pour toute contrainte $c(x, x_1, \ldots, x_k)$, les relations suivantes sont vrai :

1. $\exists v \in [\underline{x}, \underline{x}^{+^{(w)}}], \exists v_1, \ldots, v_k \in D_1 \times \ldots \times D_k \mid c(v, v_1, \ldots, v_k)$
2. $\exists v \in [\overline{x}^{-^{(w)}}, \overline{x}], \exists v'_1, \ldots, v'_k \in D_1 \times \ldots \times D_k \mid c(v', v'_1, \ldots, v'_k)$

box-consistance
La box-consistance est une approximation plus grossière de l'arc-consistance que la 2B–Consistance. En pratique cette consistance est plus précise que la 2B-consistance sur les domaines parce que, contrairement à la 2B, elle ne nécessite pas une décomposition des contraintes initiales en contraintes élémentaires. Elle consiste à générer un système de contraintes univariées et de rechercher les zéros le plus à droite et le plus à gauche. En général, ces systèmes sont résolus en utilisant la méthode de Newton [Moore, 1966].

Plus formellement, [Collavizza et al., 1999] définit la box-consistance de la manière suivante :

Soit $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ un CSP et $c \in \mathcal{C}$ une contrainte $k - aire$ sur les variables $(x_1, \ldots, x_k)$. La contrainte c est dites box-consistante si, pour chaque x_i dans $(x_1, \ldots, x_k)$ les relations suivantes sont valide :

- $\diamond$ $c(D_1, \ldots, D_{i-1}, [\underline{x}_i, \underline{x}_i^+), D_{i+1}, \ldots, D_k)$
- $\diamond$ $c(D_1, \ldots, D_{i-1}, (\overline{x}_i^-, \overline{x}_i], D_{i+1}, \ldots, D_k)$

3B-consistance
La 2B-consistance est souvent trop faible pour calculer des bornes précises. De la même manière que d'autres consistances ont été généralisées à des consistances plus fortes [Mackworth, 1977]. La 3B-consistance est une généralisation de la 2B-consistance. Intuitivement, un CSP est dit 3B-consistant, si pour chaque variable x_i, le CSP reste consistant lorsque la variable x_i est instanciée à l'une de ses bornes. Plus formellement, soit $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ un

CSP, et $P_{x_i \leftarrow \alpha}$ le CSP dérivé de P instanciant x_i à α. P est dit 3B-consistant ssi il vérifie :

- $\diamond$ $P_{x_i \leftarrow [\underline{x}_i, \underline{x}_i^+)}$ est 2B-consistant
- $\diamond$ $P_{x_i \leftarrow (\overline{x}_i^-, \overline{x}_i]}$ est 2B-consistant

kB-consistance et bound-consistance

La kB-consistance est une relaxation de la consistance chemin (path-consistance) [Mackworth, 1977]. Cette consistance est plus forte que la 2B-consistance. Plus particulièrement la kB-consistance vérifie que le système de contraintes est bien (k-1)B-consistant lorsqu'une variable est instanciée à une valeur de ses bornes. La kB(w)-consistance peut également être définie sur le même schéma.

De la même manière, la bound-consistance est basée sur la box-consistance.

La kB-consistance et la bound-consistance sont des consistances fortes. En pratique, ces consistances sont souvent couteuses en temps de calcul, mais permettent d'obtenir des bornes plus précises.

Quad : contrainte globale

Comme sur les domaines finis, les contraintes globales dans les domaines continus permettent de gagner en efficacité. Ce gain est basé sur des algorithmes de filtrage polynomiaux sur un sous-ensemble de variables liées par des contraintes spécifiques. La contrainte globale *Quad* est un exemple de contrainte globale dans les domaines continus. Cette contrainte globale a été introduite dans [Lebbah et al., 2002] et fourni un algorithme de filtrage efficace pour résoudre le sous-système d'équations et inéquations impliquant des contraintes quadratiques. Les algorithmes de filtrage classiques (e.g., 2B-consistance ou box-consistance) ne permettent pas d'effectuer des réductions significatives sur les variables impliquées dans ces contraintes, car ces contraintes sont gérées indépendamment les unes des autres.

La contrainte *Quad* consiste à approximer les contraintes quadratiques du système par un ensemble de contraintes linéaires. Ces contraintes quadratiques sont reformulées en introduisant de nouvelles variables pour les termes non linéaires (e.g., x^2 est remplacée par α_i dans la sous expression linéaire et contraint par la relaxation linéaire de x^2). Les domaines de ces variables sont calculés en utilisant l'arithmétique des intervalles. Une fois la linéarisation effectuée, le simplex est utilisé pour réduire les domaines de ces variables.

Le point critique de cette contrainte globale est que les coefficients de la relaxation linéaire sont calculés avec des nombres flottants. Les implémentations efficaces du simplex ne sont en générale pas conservatrice des solutions.

[Michel et al., 2003] propose une procédure sûre permettant d'utiliser le simplex de manière conservatrice sur les flottants.

La contrainte *Quad* est une contrainte globale efficace et sûre pour traiter les sous-systèmes quadratiques sur les domaines continus. Elle a été étendue avec succés au polynomial.

3.4.2 Recherche de solutions

Les domaines continus exploitent essentiellement des propriétés mathématiques lors de la recherche de solutions. La stratégie de choix de variables *lexicographique* et la stratégie choix de valeurs *bissection* sont souvent utilisée pour explorer l'espace de recherche.

Les réels étant infinis, les choix de valeurs sont en réalité des choix de sous-domaines, c'est-à-dire qu'un intervalle plus petit que le domaine initial va être considéré.

La stratégie *lexicographique* a été présentée dans la Section 3.3.2.1. La *bissection* quant à elle consiste à considérer les domaines $[\underline{x}, \frac{\underline{x}+\overline{x}}{2}]$ et $(\frac{\underline{x}+\overline{x}}{2}, \overline{x}]$.

3.5 PPC sur les domaines flottants

La programmation par contraintes sur les flottants a été développée pour la vérification de programmes. La programmation par contraintes restreinte aux domaines finis, et continus ne suffit pas à représenter et résoudre les problèmes de vérification de programmes contenant des calculs sur les nombres flottants. [Michel et al., 2001] sont, à ma connaissance, les premiers à avoir introduit les contraintes sur les domaines flottants pour permettre de résoudre de tels problèmes.

Les programmes avec du calcul sur les nombres à virgule flottante sont indispensables dans de nombreux systèmes complexes et critiques. Il arrive souvent que même les programmeurs les plus aguerris oublient les différences entre les domaines continus et les domaines flottants. En effet, comme présenté dans le Chapitre 2, les flottants sont une approximation des réels. Ils n'héritent pourtant pas des mêmes propriétés, telles que la continuité et les théorèmes qui en découlent. Considérons, par exemple :

$$x + 16.0 = 16.0, x \neq 0.0 \tag{3.1}$$

$$x^2 = 2 \tag{3.2}$$

Sur les nombres flottant en simple précision avec un arrondi "au plus près" pair, aucun réel n'est solution des équations 3.1, alors qu'une multitude de nombres flottants sont solutions de ces équations. A contrario, $\sqrt{2}$ est une solution sur les réels de l'équation 3.2, alors qu'aucune solution n'existe sur les nombres flottants [5].

Les domaines flottants se trouvent à la frontière entre les domaines finis et les domaines continus. C'est pourquoi, la programmation par contraintes sur les domaines flottants s'inspire de techniques issues de ces deux domaines.

3.5.1 Représentation des domaines

Les CSPs sur les domaines flottants sont définis de la même manière que ceux sur les domaines finis sauf pour la représentation des domaines des variables. Le domaine $\mathbf{x}_{\mathbb{F}}$ de la variable flottante $x_{\mathbb{F}} \in \mathcal{X}$ est défini par l'intervalle : $\mathbf{x}_{\mathbb{F}} = \{x \in \mathbb{F} \mid \underline{x}_{\mathbb{F}} \leq x \leq \overline{x}_{\mathbb{F}}\}$.

3.5.2 Consistances sur les domaines flottants

Les techniques de filtrage comme la consistance d'arc sont clairement inefficaces au vu de la cardinalité importante des domaines traités. Les filtrages comme la *box-consistance* ou la *2B-consistance* ont été adaptés aux domaines flottants [Michel et al., 2001, Michel, 2002, Botella et al., 2006].

box-consistance
[Michel et al., 2001] ont proposé d'adapter la *box-consistance* aux domaines flottants. Cette adaptation se base sur la technique du *shaving*. Cette technique consiste à réduire les domaines en essayant d'en réfuter les bornes. Plus formellement, soit le CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, $x_i \in \mathcal{X}$ et $c \in \mathcal{C}$, une contrainte k-aire :

$$D_i \leftarrow [\underline{x}_i + \epsilon, \overline{x}_i] \text{ ssi } c(D_1, \ldots, [\underline{x}_i, \underline{x}_i + \epsilon), \ldots, D_k) \Rightarrow S(\mathcal{X}, \mathcal{D}, \{c\}) = \emptyset$$

où $S(\mathcal{X}, \mathcal{D}, \{c\})$ dénote l'ensemble des solutions locales de la contrainte c. Cette réduction de domaine utilise l'arithmétique des intervalles pour prouver qu'aucune solution n'existe dans $[\underline{x}, \underline{x} + \epsilon)$.

Ce filtrage permet de conserver toutes les solutions du CSP. Néanmoins les arrondis extérieurs de l'arithmétique des intervalles conduisent à des bornes trop grossières, et le processus de "shaving" est couteux en temps de calcul.

5. avec un monde d'arrondi au plus près

2B-consistance

De la même manière, la 2B-consistance a été adaptée aux domaines flottants dans [Michel, 2002]. L'idée générale de cette consistance est d'effectuer une consistance d'arc aux bornes des domaines des variables. Formellement, un CSP $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ est 2B-consistant ssi la propriété 3.3 est vraie pour toutes les contraintes. Soit $c \in \mathcal{C}$, une contrainte k-aire, c est 2B-consistant si pour toutes variables $x_i \in vars(c)$:

$$\begin{aligned} D_i = \square\{v_i \in D_i | \exists v_1 \in D_1, \ldots, \exists v_{i-1} \in D_{i-1}, \\ \exists v_{i+1} \in D_{i+1}, \ldots, \exists v_k \in D_k \\ \text{tels que } c(v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_k)\} \end{aligned} \tag{3.3}$$

où $\square S$ est le plus petit intervalle I, tel que $S \subseteq I$.

Comme sur les réels, cette consistance est réalisée à l'aide d'un ensemble de fonctions de projections, et reposent sur une décomposition des contraintes en contraintes élémentaires binaires ou ternaires pour lesquelles, ces projections sont faciles à calculer. On distingue deux types de fonctions de projection :

- ◇ *une fonction de projection directe* calcule une projection sur la variable représentant le résultat.
- ◇ *une fonction de projection inverse* calcule une projection sur toutes les autres variables impliquées dans la contrainte.

Ces fonctions de projection sont illustrées sur l'opération d'addition suivante :

$$z = x \oplus_r y$$

avec $x, y, z \in \mathcal{X}$, et $\oplus$ l'opération d'addition sur les flottants avec le mode d'arrondi r. Dans la suite, le mode d'arrondi considéré est "au plus près" pair.

Le calcul des réductions des domaines est possible en utilisant les trois fonctions de projections suivantes [6] :

$$Z \leftarrow Z \cap [\underline{x} \oplus_{-\infty} \underline{y}, \overline{x} \oplus_{+\infty} \overline{y}] \tag{3.4}$$

$$X \leftarrow X \cap [\underline{z} \ominus_{+\infty} \underline{x}^{-}, \overline{z} \ominus_{-\infty} \overline{x}] \tag{3.5}$$

$$Y \leftarrow Y \cap [\underline{z} \ominus_{+\infty} \underline{y}^{-}, \overline{z} \ominus_{-\infty} \overline{y}] \tag{3.6}$$

6. Ces fonctions de projection sont définies de manière un peu plus précise dans [Michel, 2002]

L'équation 3.4 correspond à la fonction de projection directe. À contrario, les équations 3.5 et 3.6 sont les fonctions de projections inverses.

L'ensemble de ces fonctions de projection ainsi que leurs améliorations sont détaillées dans [Michel, 2002, Botella et al., 2006, Marre and Michel, 2010].

Notons que la décomposition utilisée par la 2B-consistance représente une relaxation du problème initial ce qui impacte les réductions possibles.

2B(w)-consistance
Pour les mêmes raisons que sur les domaines continus, la 2B(w)-consistance est utilisée en pratique. Les flottants sont répartis de manière non uniforme, la valeur w est donc en général un pourcentage de la taille du domaine.

3B-consistance et kb-consistance
Les consistances plus fortes, comme la kb-consistance, sont directement adaptables au domaine des flottants. En pratique, la kb-consistance est en général trop couteuses. La 3B-consistance reste un bon compromis entre précision et temps de calcul.

3.5.3 Recherche de solutions

Peu de travaux ont été effectués sur la recherche de solutions sur les domaines flottants. Issus essentiellement des heuristiques de recherche sur les domaines continus, l'ordre lexicographique, et la bissection (Figure 3.8a) sont respectivement les heuristiques de choix de variables et de choix de valeurs (sous-domaines) les plus communément utilisées. Néanmoins, ces heuristiques ne prennent pas en compte la spécificité des nombres à virgule flottante. [Collavizza et al., 2016] ont proposé une nouvelle stratégie de choix de sous-domaine mixant énumération et bissection. Cette stratégie (Figure 3.8a) est plus adaptée aux domaines flottants. Elle permet de recherche des solutions locales au niveau des bornes des sous-domaines. Cette stratégie énumère également le milieu du domaine. Le milieu est énuméré, car on considère possible qu'il ait des continuums de solution.

3.6 Conclusion

La programmation par contrainte permet de résoudre efficacement des problèmes difficiles, i.e avec une forte explosion combinatoire. Elle sépare la modélisation de la résolution du problème. La modélisation formalise le

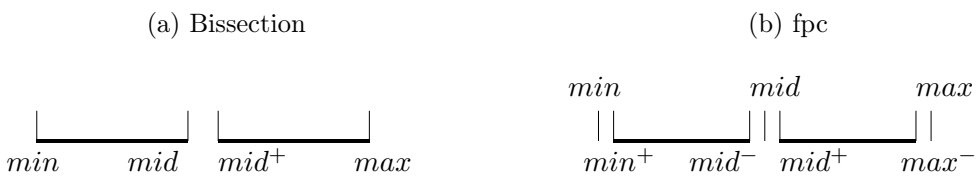


FIGURE 3.8 – Stratégie de choix de sous-domaines

problème de manière déclarative et mathématique. La résolution quant à elle se base sur deux mécanismes : le filtrage et la recherche de solutions. Le premier cherche à réduire les domaines des variables en supprimant les valeurs trivialement inconsistantes. Ces réductions sont effectuées à l'aide d'algorithmes efficaces qui dépendent du type des variables utilisées (discret, continu, flottant). La recherche de solutions applique le principe "divide and conquer" en réduisant le problème initial à un ensemble de sous-problèmes plus facile à résoudre. La recherche de solutions se base sur des heuristiques de choix de variables et valeurs pour réduire le problème. Ces heuristiques de recherche ont un impact important sur les temps de résolution des problèmes.

De nombreuses heuristiques de recherche ont été développées sur les domaines finis. Il y a eu aussi quelques travaux sur les réels [Batnini et al., 2005], mais très peu de travaux ont été effectués pour les domaines constitués de nombre flottants.

Chapitre 4

Vérification de Programmes

De nombreuses méthodes ont été mises au point pour vérifier la conformité des programmes avec du calcul sur les flottants vis-à-vis de leur spécification. L'objectif de la vérification de la conformité d'un programme avec sa spécification est de s'assurer que pour l'ensemble des exécutions possibles, et pour l'ensemble des valeurs d'entrée spécifiées le programme valide le modèle.
Dans ce chapitre nous présentons très rapidement 3 de ces techniques : la preuve formelle, l'interprétation abstraite, et le bounded model checking.

4.1 Preuve Formelle

L'objectif de la preuve formelle est de prouver mathématiquement que le programme est correct. Cette approche est conceptuellement élegante, mais ne permet pas de vérifier la validité de programmes réalistes et il peut être prouvé qu'il n'existe pas d'algorithme général permettant de prouver qu'un programme est correct [Rice, 1953].

Beaucoup de méthodes de preuve formelle sont basées sur la logique de Hoare[Hoare, 1969]. La logique de Hoare est un système formel permettant de raisonner rigoureusement sur un programme. Elle offre un ensemble de règles logiques (Tableau 4.1) qui permettent de caractériser la sémantique des instructions du langage. Le Tableau 4.1 présente quelques règles d'inférence de la logique de Hoare. Ce tableau est issu de [Tucker and Noonan, 2007]. $p \vdash q$ correspond à la déduction logique. La virgule (,) correspond à une conjonction. $Q[t \leftarrow s]$ correspond au prédicat dans lequel t est remplacé par la valeur s. Par exemple si $\{Q\} = \{t = 3 \wedge z = 2\}$, alors $Q[t \leftarrow 3] = \{3 = 3 \wedge z = 2\}$

Prouver la correction d'un programme revient à s'assurer de la validité du triplet suivant, appelé triplet de Hoare :

$$\{P\}\ s\ \{Q\} \tag{4.1}$$

où P dénote la pré-condition (ce qui est vrai avant l'exécution du programme), Q la post-condition (ce qui doit être vrai après l'execution), et s l'ensemble des instructions du programme. Ce triplet capture l'idée que si l'exécution des instructions s commence dans un état satisfaisant P, alors l'état résultant satisfait Q.

Type d'instruction (s)	régle d'inference
Affectation s.target = s.source	$true \vdash \{Q[s.target \leftarrow s.source]\}\ s\ \{P\}$
Block $s_1\ ;\ s_2$	$\{P\}\ s_1\ \{R\},\ \{R\}\ s_2\ \{Q\} \vdash \{P\}\ s_1\ ;\ s_2\ \{Q\}$
Condition if s.test s.thenpart else s.elsepart	$\{s.test \wedge P\}\ s.thenpart\ \{Q\},$ $\{\neg s.test \wedge P\}\ s.elsepart\ \{Q\} \vdash \{P\}\ s\ \{Q\}$
...	

TABLE 4.1 – Régles d'inférence [Tucker and Noonan, 2007]

Limitations Ces preuves nécessitent une expertise en logique et demande de calculer des invariants lorsque le programme à vérifier contient des boucles. Des assistants automatiques de preuve [Coq development team, 2009, Nipkow et al., 2002, Marciszewski, 1994] ont été développés pour aider à la vérification des preuves. Ces assistants accumulent un ensemble de tactiques de preuve, mais ils nécessitent une interaction importante avec l'utilisateur. La mise en oeuvre de ces techniques sur des programmes réalistes est donc longue et fastidieuse.

4.2 Interprétation Abstraite (IA)

L'interprétation abstraite [Cousot and Cousot, 1977a, Cousot and Cousot, 1977b] est une méthode formelle d'analyse de programme. L'objectif de cette approche est de prouver l'absence d'erreurs (e.g., division par 0). Ses fondements théoriques sont les théories des treillis et des points fixes. Contrairement aux méthodes de Bounded Model Checking (BMC), cette approche permet de raisonner sur l'ensemble des exécutions possibles du programme. Elle permet d'extraire des propriétés intéressantes sur le programme telles que les dépassements de capacités, des invariants, ou encore de borner le nombre d'itérations des boucles. L'IA est une approche efficace qui permet de passer à l'échelle et vérifier des programmes significatifs contenant du calcul sur les entiers, ou encore sur les flottants [Cousot et al., 2005, Kästner et al., 2010, Miné, 2004a, Miné, 2004b].

Dans le cas général, le problème d'analyse exhaustive d'un programme est un problème indécidable ([Rice, 1953, Pavlotskaya, 1973]). L'interprétation abstraite calcule donc une approximation de la sémantique du programme appelé abstraction. L'abstraction est plus simple que la sémantique concrète du programme, mais doit couvrir l'ensemble des exécutions possibles. La négation de la spécification fournit quant à elle des zones interdites qu'aucune exécution du programme ne devrait atteindre. Si l'abstraction intersecte ces zones, cela signifie qu'un bug potentiel existe. La Figure 4.1 (issue de [Cousot and Cousot, 2010]) illustre l'idée générale de l'IA.

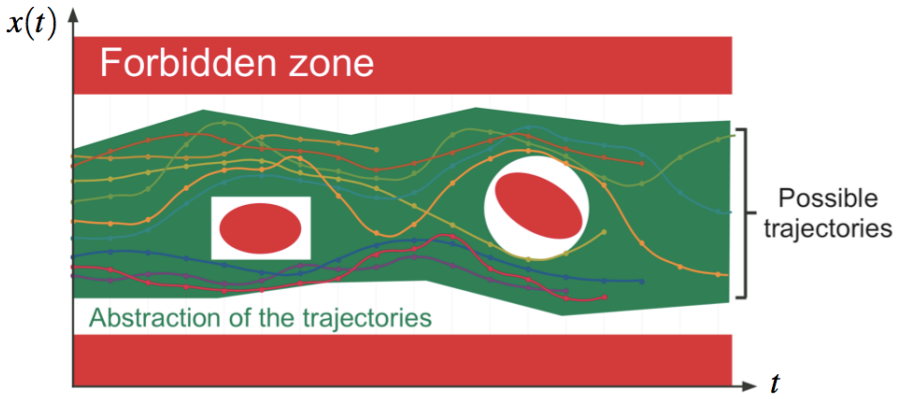


FIGURE 4.1 – Approches IA

Les Figures 4.2 (issue de [Miné, 2006]) présentent des exemples de domaines abstraits utilisés pour calculer l'abstraction de la sémantique d'un programme. Le choix du domaine abstrait impacte directement les propriétés pouvant être extraites. Il implique aussi un compromis entre efficacité et précision. Les domaines abstraits les plus utilisés sont les intervalles, les octogones [Miné, 2006], les zonotopes [Ghorbal et al., 2009], et les polyèdres [Singh et al., 2017].

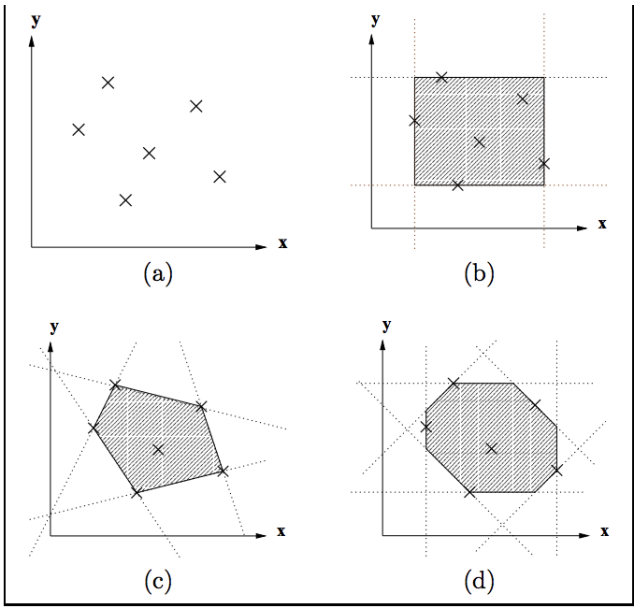


FIGURE 4.2 – un nuage de points (a), son approximation utilisant une abstraction d'intervalle (b), de polyèdres (c) et d'octagones (d)

Limitation Les approximations fournies par les approches IA peuvent conduire à de faux positifs appelés aussi fausses alarmes (Figure 4.3 issue de [Cousot and Cousot, 2010]). Ces faux positifs arrivent lorsque l'approximation intersecte la zone interdite, alors que la sémantique concrète du programme ne l'intersecte pas (Figure 4.3). De plus, ces approches permettent de vérifier des programmes, mais ne fournissent pas les jeux de données violant la spécification, lorsqu'ils en existent.

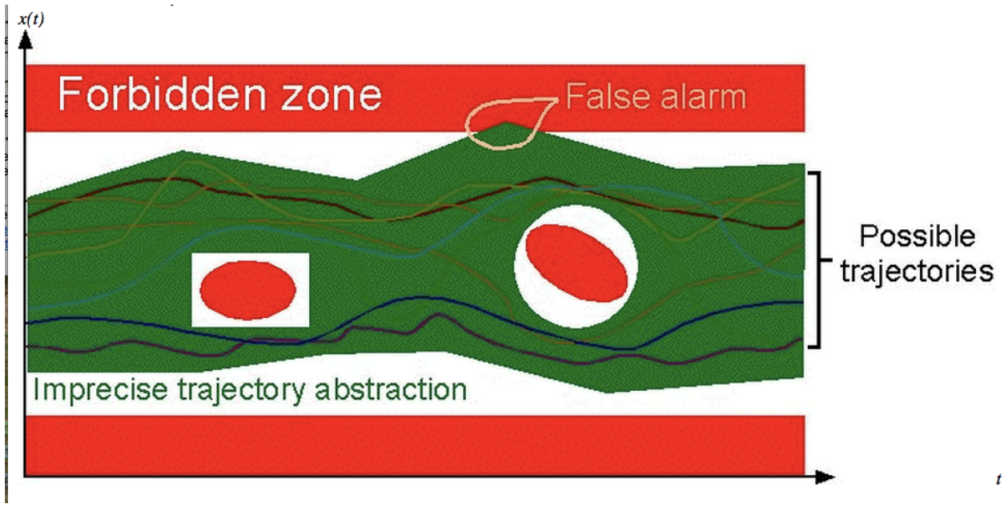


FIGURE 4.3 – Fausse alarme

4.3 Bounded model checking

L'objectif des techniques de bounded model checking est de chercher des valeurs d'entrées du programme pour lesquelles la spécification est invalidée. Ces valeurs d'entrées sont appelées contre-exemples. Si on arrive à montrer qu'il n'existe pas de tels contre exemple on a montré que le programme est correct sous reserve que les hypothèses sur la taille des domaines et le nombre d'itérations, en particulier des boucles, soient respectées.

Les approches bounded model checking [1] (BMC) [Biere et al., 1999] sont des approches efficaces pour rechercher des contre-exemples. L'idée de ces techniques est de rechercher un contre-exemple sur une exécution bornée du programme. En effet, un des problèmes majeurs lors de la vérification de programme est le traitement des boucles. Les boucles sont sujettes à des problèmes de terminaison. Borner l'exécution permet donc d'échapper à ce problème en dépliant les boucles k fois. La Figure 4.4 présente un exemple de programme déplié 2 fois. Si aucun contre-exemple n'est trouvé, cela ne signifie pas pour autant que le programme est correct. Par exemple, en dépliant 2 fois le programme de la Figure 4.4, aucun contre-exemple n'est trouvé ($i = 1$). Il existe néanmoins un contre-exemple lorsque le même programme est déplié 3 fois ($i = 0$).

1. vérification de modéle bornée

En pratique, les approches BMC augmentent la borne si aucun contre-exemple n'est trouvé. En général, après un nombre raisonnable d'augmentation de la borne sans trouver de contre-exemple, le programme est considéré correct vis-à-vis de sa spécification.

(a) Programme original

```
i = 3;
while (i >= 1){
        x = x * i;
        i--;
}
assert(i == 1);
```

(b) Programme déplié

```
i = 3;
if (i >= 1){
        x = x * i;
        i--;
}
if (i >= 1){
        x = x * i;
        i--;
}
assert(i == 1);
```

FIGURE 4.4 – Programme déplié 2 fois

4.3.1 BMC : principes de fonctionnement

Dans la suite de cette section, nous détaillons d'abord le principe général des approches BMC, puis nous présentons les différents solveurs utilisés dans les principaux outils BMC.

Les approches BMC commencent par mettre le programme sous forme SSA (Static Single Assignement).

Cette forme consiste à transformer le programme de telle sorte qu'aucune variable n'apparaissent plus d'une fois du côté gauche d'une affectation. La transformation SSA est un format utilisé par de nombreux compilateurs pour effectuer des simplifications. La forme SSA permet de mettre le programme sous forme relationnelle. La Figure 4.5 donne un exemple de transformation sous forme SSA.

Les approches BMC construisent à partir de ce programme transformé une formule P capturant la sémantique opérationnelle du programme. L'utilisateur doit fournir une formule S capturant sa spécification ou la propriété qui doit être vérifiée. Puis, le solveur BMC cherche une instanciation des variables pour laquelles la formule $P \wedge \neg S$ est vérifiée. S'il existe un tel contre-exemple alors le programme ne respecte pas la spécification.

(a) Programme original

```
x = 2 * a
y = x + 5
x = x + y
```

(b) Programme sous forme SSA

```
x0 = 2 * a0
y0 = x0 + 5
x1 = x0 + y0
```

FIGURE 4.5 – Transformation d'un programme sous forme SSA

À contrario, s'il n'est pas possible de trouver une telle instanciation des variables, alors le programme est conforme à sa spécification pour les domaines donnés.
La vérification de $P \wedge \neg S$ est en général réalisée à l'aide d'un solveur de contraintes. Dans l'état de l'art de la vérification de programme, différents solveurs ont été utilisés : SAT, SMT, PPC.

La Figure 4.6 illustre l'ensemble des processus des approches de type BMC. Dans cette figure, C_p (resp. $C_{\neg S}$) correspond à la transformation du programme (resp. la négation de la spécification) sous forme de contraintes.

Dans les sous sections suivante, nous allons briévement évoquer les avantages et limites des différents solveurs utilisés.

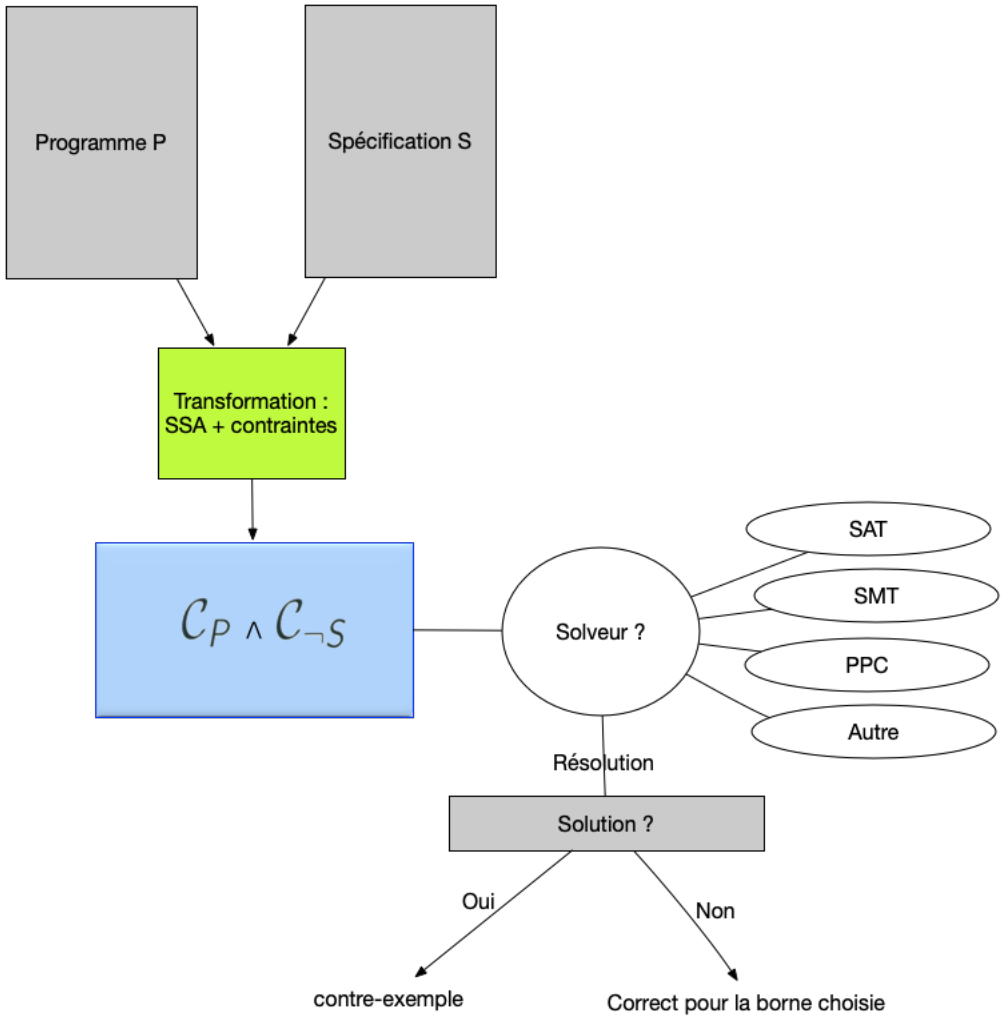
4.3.2 Approches BMC basées sur les solveurs SAT

Les solveurs SAT ont prouvé leur efficacité dans de nombreuses applications. Les premières approches BMC ont exclusivement utilisé des solveurs SAT pour rechercher des contre-exemples.

L'utilisation de ces solveurs permet de bénéficier de techniques efficaces comme l'apprentissage de clause, mais également de profiter des nombreux solveurs disponibles.

CBMC [Clarke et al., 2004] est un exemple d'un outil de vérification de programme qui utilise une approche BMC. Le solveur par défaut utilisé par cet outil est minisat [Eén and Sörensson, 2004].

Limitations Les approches BMC basée sur un solveur SAT nécessitent une formulation booléenne du programme et de sa spécification. La difficulté principale réside dans la transformation des instructions du programme qui travaillent sur les chaines ou des nombres, en formules booléennes. Cette transformation génère souvent des formules de taille considérable. La taille de ces formules est une des raisons pour laquelle les approches BMC basées sur les solveurs SAT ne passent pas à l'échelle.


Programme P
Spécification S
Transformation :
SSA + contraintes
$\mathcal{C}_P \wedge \mathcal{C}_{\neg S}$
Solveur ?
SAT
SMT
PPC
Autre
Résolution
Solution ?
Oui
Non
contre-exemple
Correct pour la borne choisie

De plus, la structure du programme (ordre des instructions, ...) est perdue lors de la transformation ce qui ne permet pas d'exploiter facilement les spécifications du programme pour produire des explications liées aux contre-exemples trouvés.

4.3.3 Approches BMC basées sur les solveurs SMT

Les approches SMT sont des généralisations des approches SAT. Une instance SMT correspond à une instance SAT où les variables sont remplacées par des prédicats issus de formalismes appelés théories logiques. L'utilisation de ces théories logiques permet d'utiliser des solveurs de théories dédiés. Le résultat est, par la suite, traduit sous forme propositionnelle. Par conséquent, les approches SMT ne peuvent s'appliquer qu'à des théories dont les résultats peuvent s'exprimer sous forme propositionnelle. Ces théories permettent d'étendre les capacités de modélisation par rapport aux approches SAT mais leurs efficacité dépend fortement de celles des solveurs pour les différentes théories utilisées.

Ainsi, le solveur SMT [de Moura and Bjørner, 2007, Bruttomesso et al., 2008] a permis d'offrir un cadre plus ou moins générique pour traiter les problèmes de vérification de programmes contenant du calcul sur les entiers, ou encore les flottants, par le biais de théories. Ce cadre permet, entre autres, d'offrir une meilleure expressivité et donc de vérifier plus de programmes.
De nombreux outils sont basés sur des solveurs SMT : CBMC (à partir de la version 3.3), ESBMC [Cordeiro et al., 2012], CDFL [D'Silva et al., 2012a]. Ces outils ont permis d'augmenter la base de programmes pouvant être traités en vérification de programmes.

Limitation Pour la modélisation des opérations numériques les approches SMT se basent souvent sur des techniques comme le *bit-blasting* pour encoder les théories sous la forme de logique propositionnelle. Cette techniques est efficace sur de petits problèmes bien formulés, mais peut entrainer une explosion exponentielle [Kovásznai et al., 2016] de la taille de la formule.

4.3.4 Approches BMC basées sur la PPC

La PPC [2] amène un certain confort en terme d'expressivité pour les calculs numériques, les instructions d'un programme se traduisant assez directement en contraintes qui utilisent des domaines adaptés. La PPC utilise des contraintes et domaines spécifiques à la nature des programmes à vérifier. La Figure 4.8 illustre un extrait du format SMT correspondant au programme Héron [Zitoun et al., 2017] et sa spécification (Figure 4.7). Le fichier SMT (cf Figure 4.8) généré en utilisant ESBMC 5 fait plus de 220 lignes. En comparaison, la Figure 4.9 illustre le même programme transformé en contraintes pour le solveur FPCS (moins de 30 lignes). Cet exemple montre bien la différence d'expressivité, et la taille des contraintes générées.

En plus d'une expressivité accrue, la PPC offre des techniques de filtrage et de recherche efficaces pour trouver des contre-exemples. Que ce soit sur les domaines finis, continus ou encore flottants, la PPC apporte des techniques dédiées permettant de raisonner sur chacun de ces domaines. Cet apport donne la capacité aux approches BMC basées sur la PPC de vérifier des programmes sur les entiers et sur flottants. La PPC permet aussi d'exploiter la structure des programmes comme l'ordre des instructions ou le graphe de flot de contrôle [Collavizza and Rueher, 2006, Collavizza et al., 2010].

Sur les domaines finis, de nombreux travaux ont prouvé l'efficacité de l'utilisation de solveur PPC pour la recherche de contre-exemples, ainsi que la capacité de ces approches à traiter des programmes de taille significative. Par exemple, [Collavizza and Rueher, 2006] ont mis en oeuvre une approche hybride combinant booléens pour la gestion des tests conditionnels et solveur sur entiers pour gérer la partie opératoire du programme.

D'autres approches comme [Aït-Kaci et al., 2007], ont également montré les capacités de la PPC pour vérifier les programmes en incorporant des méthodes de raisonnement issues du SMT.

Des outils comme CPBPV [Collavizza et al., 2010] ont cherché à exploiter la structure du programme en tirant profit du graphe de flot de contrôle. Ces travaux cherchent à couper les chemins qui sont trivialement insatisfiable en parcourant successivement tous les chemins de longueur bornée en générant des systèmes de contraintes à la volée.

[Collavizza and Rueher, 2006] ont également montré, sur les benchmarks testés, de meilleurs temps de résolution que les approches basées sur les solveurs SMT.

2. Le Chapitre 3 détaille les concepts et techniques de la programmation par contraintes.

```
float heron(float a,float b,float c){
        float s;
        float aire;
        s = (a+b+c)/2.0f;
        aire = s*(s-a)*(s-b)*(s-c);
        return aire;
}

int main() {
        float a = nondet_float();
        float b = nondet_float();
        float c = nondet_float();

        __VERIFIER_assume(a > 0);
        __VERIFIER_assume(b > 0);
        __VERIFIER_assume(c > 0);
        __VERIFIER_assume((a+c) > b);
        __VERIFIER_assume((a+b) > c);
        __VERIFIER_assume((b+c) > a);
        __VERIFIER_assume(a > b);
        __VERIFIER_assume(b > c);

        float aire = 0;
        aire = heron(a,b,c);

        assert(aire >= 1e-5f);

}
```

FIGURE 4.7 – Source C du programme heron et de sa spécification

```
(set-info :source |printed by MathSAT|)
(declare-fun __ESBMC_ptr_obj_start_0 () (_ BitVec 32))
(assert (let ((.def_11 (= __ESBMC_ptr_obj_start_0 (_ bv0 32))))
.def_11))

(set-info :source |printed by MathSAT|)
(declare-fun __ESBMC_ptr_obj_end_0 () (_ BitVec 32))
(assert (let ((.def_13 (= __ESBMC_ptr_obj_end_0 (_ bv0 32))))
.def_13))

(set-info :source |printed by MathSAT|)
(declare-fun __ESBMC_ptr_obj_start_1 () (_ BitVec 32))
(assert (let ((.def_16 (= __ESBMC_ptr_obj_start_1 (_ bv1 32))))
.def_16))

(set-info :source |printed by MathSAT|)
(declare-fun __ESBMC_ptr_obj_end_1 () (_ BitVec 32))
(assert (let ((.def_19 (= __ESBMC_ptr_obj_end_1 (_ bv4294967295 32))))
.def_19))

(set-info :source |printed by MathSAT|)
(declare-fun __ESBMC_ptr_obj_start_0 () (_ BitVec 32))
(declare-fun __ESBMC_ptr_obj_end_0 () (_ BitVec 32))
(declare-fun |smt_conv::__ESBMC_ptr_addr_range_0..start0| () (_ BitVec 32))
(declare-fun |smt_conv::__ESBMC_ptr_addr_range_0..end0| () (_ BitVec 32))
(assert (let ((.def_23 (= __ESBMC_ptr_obj_end_0
|smt_conv::__ESBMC_ptr_addr_range_0..end0|)))
(let ((.def_22 (= __ESBMC_ptr_obj_start_0
|smt_conv::__ESBMC_ptr_addr_range_0..start0|)))
(let ((.def_24 (and .def_22 .def_23)))
.def_24))))

(set-info :source |printed by MathSAT|)
(declare-fun __ESBMC_ptr_obj_start_1 () (_ BitVec 32))
(declare-fun __ESBMC_ptr_obj_end_1 () (_ BitVec 32))
(declare-fun |smt_conv::__ESBMC_ptr_addr_range_1..start0| () (_ BitVec 32))
(declare-fun |smt_conv::__ESBMC_ptr_addr_range_1..end0| () (_ BitVec 32))
(assert (let ((.def_28 (= __ESBMC_ptr_obj_end_1
|smt_conv::__ESBMC_ptr_addr_range_1..end0|)))
(let ((.def_27 (= __ESBMC_ptr_obj_start_1
|smt_conv::__ESBMC_ptr_addr_range_1..start0|)))
(let ((.def_29 (and .def_27 .def_28)))
.def_29))))
```

FIGURE 4.8 – Extrait des 220 lignes du programme et de sa spécification transformé en SMT à l'aide de ESBMC 5

```
int main(int argc, char *argv[]) {
  Fpc_Csp CSP;

  Fpc_Variable a(CSP, (char *)"a", 5.0f, 10.0f, FPC_FLOAT);
  Fpc_Variable b(CSP, (char *)"b", 0.0f, 5.0f, FPC_FLOAT);
  Fpc_Variable c(CSP, (char *)"c", 0.0f, 5.0f, FPC_FLOAT);

  Fpc_Variable s(CSP, (char *)"s", FPC_FLOAT);
  Fpc_Variable squared_area(CSP, (char *)"squared_area_1", FPC_FLOAT);

  Fpc_Model model(CSP);

  // a > 0 & b > 0 & c > 0
  model.add(a > 0);
  model.add(b > 0);
  model.add(c > 0);

  //a + b > c & b + c > a & a + c > b
  model.add(a + c > b);
  model.add(a + b > c);
  model.add(b + c > a);

  //a > b > c
  model.add(a > b);
  model.add(b > c);

  model.add(s = (a+b+c)/2.0f);
  model.add(squared_area = s*(s-a)*(s-b)*(s-c));

  model.add(squared_area < 1e-5);
  model.extract();
  return 0;
}
```

FIGURE 4.9 – Modèle PPC complet du même programme et de sa spécification sous FPCS

Sur les domaines de flottants, [Michel et al., 2001, Michel, 2002] ont mis en oeuvre FPCS un solveur de contraintes basées sur des filtrages efficaces pour les approches PPC permettant de vérifier des programmes avec du calcul sur les flottants. Les travaux [Belaid et al., 2012] cherchent quant à eux à approximer sur les réels, les contraintes sur les flottants pour pouvoir tirer profit des techniques existantes sur les domaines continus.

4.3.5 Combinaison de l'IA et de la PPC

Les approches IA permettent de passer à l'échelle et de traiter efficacement les boucles, mais les approximations requisent pour un traitement efficace peuvent entrainer des fausses alarmes. À contrario, la PPC a du mal à passer à l'echelle et traiter les boucles de manière efficace, mais fournis des jeux d'entrée lorsque un contre-exemple existe. Elle calcule également des approximations plus fines que l'IA. De nombreux travaux [Denmat et al., 2007, Ponsini et al., 2010, Ponsini et al., 2012, Pelleau et al., 2014, D'Silva et al., 2012b, Kabi et al., 2016] ont cherché à combiner ces approches complémentaires. Les points faibles de l'une sont comblées par les points forts de l'autre, et inversement.

Par exemple, [Denmat et al., 2007] ont proposé d'utiliser l'interprétation abstraite dans un solveur de contraintes pour permettre de traiter efficacement les boucles.
Ou encore, [Pelleau et al., 2014] ont introduit des domaines abstrait pour la résolution de programmes dans le domaine continu.
Des approches comme [Ponsini et al., 2012] ont cherché à faire communiqué un solveur de contraintes (FPCS) et un analyseur statique (Fluctuat). Cette approche est basée sur une exploration du graphe de flot de contrôle où FPCS et Fluctuat partagent leurs réductions pour affiner les déductions.

En IA, d'autres techniques ont été introduites pour quantifier (donner une borne) de la déviation (l'erreur) dans les programmes contenant du calcul sur les nombres flottants. En d'autres termes, ces techniques consistent à interpréter le programme sur les réels, et sur les flottants puis de calculer une borne de la déviation. Ces techniques ont pour objectif de quantifier la déviation du programme interprété sur les réels par rapport au programme évalué sur les flottants [Goubault and Putot, 2006, Hauser, 1996, Titolo et al., 2018b] mais comme toute technique d'interprétation abstraite, elles sont basées sur une approximation et ne peuvent donc fournir qu'une borne supérieure de la déviation. Ces approches sont complémentaires aux approches IA et BMC présentées dans ce chapitre.

4.4 Conclusion

Dans ce chapitre, nous avons présenté plusieurs approches dont l'objectif est de vérifier que les programmes sont conformes à leurs spécifications.
La preuve formelle cherche à prouver mathématiquement qu'un programme est correct en utilisant la logique de Hoare. Cette approche est élegante, mais elle est difficile à mettre en oeuvre et les outils sur lesquelles elle repose ne sont pas du tout efficace.
Les approches IA se basent sur une approximation de la sémantique du programme et de zones interdites définies par la négation de la spécification. Si aucune intersection n'existe entre l'abstraction et les zones interdites le programme est correct. Si une intersection existe, celle-ci peut correspondre à une erreur ou juste provenir de la sur approximation (fausse alarme). Ces techniques sont très efficaces et permettent de passer à l'échelle, mais ne fournissent pas de valeurs d'entrées des contre-exemples.
Les approches BMC, cherchent quant à elle à exiber l'existence de contre-exemples violant la spécification sur une execution bornée du programme. Ces approches transforment le programme et la négation de sa spécification en un système de contraintes et appellent un solveur de contraintes (SAT, SMT, ou PPC) pour trouver une solution. Lorsqu'une solution existe, elle correspond à un contre-exemple. Si aucune solution n'existe, le programme est correct vis-à-vis de sa spécification pour les bornes considérées au niveau des tableaux et des boucles.

De nombreux travaux ont cherché à combiner ces différentes approches. Combiner les points forts de ces différentes approches est probablement la direction qu'il faudrait adopter pour pouvoir fournir des outils fiables, robustes et utilisables dans les processus de développement industriel pour fournir des programmes plus sûrs. Ce point est d'autant plus pertinent lorsque les programmes contiennent des calculs sur les flottants qui sont une source additionnelle d'erreurs.

Deuxième partie

Contributions : Stratégies de Recherche sur les Flottants

Introduction

Comme mentionné dans les chapitres précédents, 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. Ces calculs peuvent entrainer un résultat sensiblement différent du résultat attendu sur les réels.

Le programme de la Figure 4.10 illustre un problème majeur où des erreurs de calcul dues à l'arithmétique spécifique des flottants peuvent conduire à une décision sensiblement différente de celle qui aurait été obtenue sur les nombres réels. En interprétant le programme sur les réels, la valeur de r calculée est 1 (ligne 5) : l'instruction `doThenPart` devrait donc être exécutée. Sur les nombres flottants, un problème d'absorption se produit : la valeur 1 est absorbée par `1e8f` [3] et r est donc égal à 0. Le test de ligne 6 échoue, et la branche `else` est exécutée. Cette différence de comportement peut avoir de lourdes conséquences pour un programme embarqué si, par exemple, elle conditionne le freinage (ou non) dans un système comme l'ABS (Système Anti-Blocage des roues).

Les calculs en virgule flottante sont donc une source supplémentaire d'erreurs dans les programmes embarqués. Ces problèmes viennent du fait que très souvent, les programmes sont conçus par des programmeurs qui ont la sémantique des nombres réels à l'esprit. Pour des décisions critiques, il est vital de s'assurer que certaines propriétés ne peuvent pas être violées à cause des erreurs arithmétiques sur les nombres flottants [Titolo et al., 2018a]. La recherche de contre-exemples (valeurs d'entrées qui violent ces propriétés) est donc un problème important dans la vérification des programmes embarqués.

De nombreuses stratégies de recherche ont été proposées sur les entiers [Linderoth and Savelsbergh, 1999, Boussemart et al., 2004, Gay et al., 2015, Michel and Van Hentenryck, 2012, Refalo, 2004] et, dans une moindre mesure, sur les réels [Jussien and Lhomme, 1998, Batnini et al., 2005, Kearfott, 1987].

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

```
1  void foo(){
2    float a = 1e8f;
3    float b = 1.0f;
4    float c = -1e8f;
5    float r = a + b + c;
6    if(r >= 1.0f){
7      doThenPart();
8    } else {
9      doElsePart();
10   }
11 }
```

FIGURE 4.10 – Exemple 1

Les stratégies dédiées aux entiers ou aux réels sont mal adaptées aux flottants car ces domaines ont des propriétés différentes. Le sous-ensemble des entiers d'un domaine bornés par deux entiers est fini et réparti de manière uniforme ; il est donc possible d'en énumérer toutes les valeurs lorsque le domaine est suffisamment petit. Le sous-ensemble des réels bornés par deux flottants étant infini et uniforme, il n'est pas énumérable.

Les stratégies de recherche adaptées au continu utilisent des techniques sur les intervalles telles que la bissection ainsi que des propriétés mathématiques qui exploitent par exemple la jacobienne pour montrer qu'un intervalle contient une solution.

A contrario, l'ensemble des flottants est lui fini, mais sa cardinalité est très élevée et la répartition des flottants n'est pas uniforme : la moitié des flottants se trouvent dans le domaine [-1,1]. Les techniques précédentes, comme l'énumération, sont donc généralement inenvisageables dans le cas des flottants. Les flottants correspondent à une approximation des réels, mais n'héritent pas des mêmes propriétés mathématiques, telles que la continuité. Il est donc difficile de tirer profit des stratégies de recherches basées sur les propriétés des réels.

Dans cette partie, nous présentons les contributions de cette thése. Ces contributions sont des stratégies de recherches dédiées aux systèmes de contraintes sur les flottants pour la vérification de programmes. L'objectif des stratégies introduites est de faciliter et accélérer la résolution des problèmes de vérification.

Les stratégies de recherche reposent sur un choix de variables et un choix de valeurs. Les stratégies introduites ici pour les choix de variables et de valeurs se basent sur des propriétés spécifiques aux flottants. Le chapitre 5

présente ces propriétés. Le chapitre 6 détaille les stratégies de choix de variables qui en découlent, alors que le chapitre 7 se focalise sur les stratégies de choix de sous-domaines.

Chapitre 5

Propriétés des domaines et des contraintes

Dans ce chapitre, nous détaillons les différentes propriétés que nous allons utilisé dans le chapitre suivant pour définir des stratégies de choix de variables et domaines dédiées aux systèmes de contraintes sur les flottants. Nous présentons d'abord les propriétés basées sur les domaines des variables, puis celles basées sur les contraintes.

5.1 Propriétés basées sur les domaines des variables

Dans cette section, nous allons étudierles spécificités des propriétés de taille, cardinalité, densité, et de degré pour les variables définies sur les nombres flottants.

5.1.1 Taille

La taille du domaine est définie par la distance entre ses deux bornes. C'est un critère plutôt historique. Sur les domaines finis, de nombreuses stratégies s'appuient sur ce critère, en particulier l'une des plus répandues, à savoir min-domain [Linderoth and Savelsbergh, 1999]. La sélection de variables avec le plus petit domaine vise à se concentrer sur les variables qui contraignent le plus le problème.

Définition 1 (Taille) *Soit $w(\mathbf{x}_\mathbb{F})$, la taille du domaine $\mathbf{x}_\mathbb{F}$, définie par :*

$$w(\mathbf{x}_\mathbb{F}) = \overline{x}_\mathbb{F} - \underline{x}_\mathbb{F}$$

Sur les domaines flottants, l'intérêt de ce critère est discutable en raison de la distribution non uniforme des nombres à virgule flottante. Le fait que la taille d'un domaine soit plus petite que celle d'un autre, n'implique pas nécessairement que sont nombre de flottants soit inférieur.

5.1.2 Cardinalité

La cardinalité $|\mathbf{x}_\mathbb{F}|$ du domaine de la variable $x_\mathbb{F}$ correspond au nombre de flottants contenus dans l'intervalle associé au domaine de $x_\mathbb{F}$. Cette propriété peut être utilisée pour identifier soit le domaine contenant le moins de flottants, i.e, la variable contraignant le plus le problème, soit le domaine contenant le plus de flottant, i.e., la variable avec le plus fort potentiel de solutions. Elle peut être calculée de la manière suivante :

Définition 2 (Cardinalité) *$|\mathbf{x}_\mathbb{F}|$ dénote la cardinalité du domaine $\mathbf{x}_\mathbb{F}$. Soit $\mathbf{x}_\mathbb{F} = [\underline{x}_\mathbb{F}, \overline{x}_\mathbb{F}]$ avec $\underline{x}_\mathbb{F} \geq 0$, $|\mathbf{x}_\mathbb{F}|$ est définie par*

$$|\mathbf{x}_\mathbb{F}| = 2^p * (e_{\overline{x}_\mathbb{F}} - e_{\underline{x}_\mathbb{F}}) + m_{\overline{x}_\mathbb{F}} - m_{\underline{x}_\mathbb{F}} + 1$$

où $e_{\overline{x}_\mathbb{F}}$ et $e_{\underline{x}_\mathbb{F}}$ sont les exposants respectifs de $\overline{x}_\mathbb{F}$ et $\underline{x}_\mathbb{F}$, et $m_{\overline{x}_\mathbb{F}}$ et $m_{\underline{x}_\mathbb{F}}$ leurs mantisses. p correspond à la taille de la mantisse.

Cette formule peut être étendue aux autres cas en exploitant les symétries.

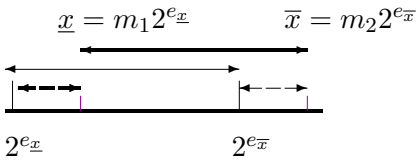


FIGURE 5.1 – Calcul de la cardinalité d'un intervalle flottant

La Figure 5.1 illustre la manière dont est calculée la cardinalité d'un intervalle flottant. La double flèche en gras représente l'intervalle $\mathbf{x}_\mathbb{F}$.

Si les deux bornes de l'intervalle ont le même exposant, soustraire leur mantisse suffit (au flottant près) à comptabiliser le nombre de flottants contenu dans celui-ci. Si les bornes n'ont pas le même exposant, il faut utiliser la propriété suivante : le nombre de flottants représentables avec un exposant est 2^p. En tenant compte de cette remarque, calculer le nombre de flottants contenus dans l'intervalle $[2^{e_{\underline{x}}}, 2^{e_{\overline{x}}}]$ revient à multiplier la différence des exposants par le nombre de flottants représentables sur un exposant (représenté

par la flèche continue simple dans la Figure 5.1). Comme ce calcul ne tient pas compte des flottants séparant $2^{e_{\overline{x}}}$ et la borne supérieure (représenté par la flèche simple discontinue), il faut encore ajouter la mantisse de cette borne. À l'opposé ce calcul prend en compte à tort les flottants qui séparent la borne inférieure de $2^{e_{\underline{x}}}$ (représenté par la flèche en gras discontinue), il faut donc soustraire la mantisse de cette borne. Il est nécessaire de comptabiliser un flottant supplémentaire (+1), car cette soustraction ne tient plus compte du flottant $\underline{x}$.

Sur les domaines finis, taille et cardinalité sont des concepts qui capturent souvent la même chose (particulièrement lorsqu'il n'y a pas de 'trou' dans les domaines). Cependant sur les flottants, ces propriétés ne sont pas du tout corrélées. La taille et la cardinalité jouent des rôles différents sur les flottants.

Exemple : Taille versus cardinalité

Soit $x_{\mathbb{F}}$ et $y_{\mathbb{F}}$ deux variables flottantes en simples précisions et $\mathbf{x}_{\mathbb{F}} = [1, 2]$, $\mathbf{y}_{\mathbb{F}} = [10, 12]$ leurs domaines respectifs. Bien que $w(\mathbf{x}_{\mathbb{F}}) = 1$ et $w(\mathbf{y}_{\mathbb{F}}) = 2$, $\mathbf{x}_{\mathbb{F}}$ contient 8388608 valeurs flottantes et $\mathbf{y}_{\mathbb{F}}$ contient 2097152 valeurs. La variable la plus contraignante est donc $y_{\mathbb{F}}$ plutôt que $x_{\mathbb{F}}$.

Les propriétés de taille et cardinalité peuvent être aussi utilisées pour calculer la densité que nous allons définir dans la sous-section suivante.

5.1.3 Densité

Intuitivement, la densité capture la proximité des nombres flottants pour un domaine donné. La densité d'une variable peut permettre d'identifier les variables dont le domaine a des valeurs disparates (densité faible). Le choix d'une variable 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.

Définition 3 (Densité) *Soit $\rho(\mathbf{x}_{\mathbb{F}})$ la densité de $\mathbf{x}_{\mathbb{F}}$ definie par*

$$\rho(\mathbf{x}_{\mathbb{F}}) = \frac{|\mathbf{x}_{\mathbb{F}}|}{w(\mathbf{x}_{\mathbb{F}})}$$

5.1.4 Magnitude

La propriété de magnitude permet d'identifier les domaines qui contiennent principalement de grandes ou de petites valeurs. Plus précisément, la magnitude a un double objectif. D'une part, 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 5.2. D'autre part, 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 (e.g., **overflow**). Ces comportements introduisent en général des erreurs importantes qui peuvent conduire à violer la spécification.

Définition 4 (Magnitude) *Soit* $mag(\mathbf{x}_{\mathbb{F}})$ *la magnitude de* $\mathbf{x}_{\mathbb{F}}$ *définie par*

$$mag(\mathbf{x}_{\mathbb{F}}) = \frac{e_{\underline{x}_{\mathbb{F}}} + e_{\overline{x}_{\mathbb{F}}}}{2 \cdot e_{max}}$$

où e_{max} *est le plus grand exposant représentable dans le format choisi.*

En pratique, la magnitude du domaine $[0, 1]$ est proche de zéro tandis que la magnitude du domaine $[10^{36}, 10^{37}]$ est proche de 1.

5.1.5 Degré

Le degré 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 [Boussemart et al., 2004].

Définition 5 (Degree) *Soit* $degree(x_{\mathbb{F}})$*, le degré d'une variable* $x_{\mathbb{F}}$ *défini comme le nombre de contraintes dans laquelle* $x_{\mathbb{F}}$ *apparait. Cette propriété est définie par*

$$degree(x_{\mathbb{F}}) = \sum_{c \in C} (x_{\mathbb{F}} \in vars(c))$$

où C *est l'ensemble des contraintes et* $vars(c)$ *les variables impliquées dans la contrainte* c*.*

Cette propriété est une propriété statique.

5.1.6 Occurrences multiples

Les occurrences multiples d'une variable sont un problème récurrent en programmation par contraintes. Les occurrences multiples sont difficiles à gérer du fait de la décomposition des contraintes initiales en contraintes élémentaires. Cette décomposition est une relaxation du problème initiale. Au sein d'une contrainte, les différentes occurrences d'une même variable sont considérées comme des variables indépendantes les unes des autres. De nombreux travaux cherchent à gérer ces occurrences multiples via des consistances adaptées [Lhomme, 1993].

L'idée de cette propriété est 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 également d'identifier les variables qui ont un impact fort sur une contrainte spécifique.

Définition 6 (Occurrences) *Soit $occur(x_{\mathbb{F}})$, le nombre maximum d'occurrences de $x_{\mathbb{F}}$ parmi toutes les contraintes (C). Cette propriété est définie par*

$$occur(x_{\mathbb{F}}) = \max_{c \in C}(x_{\mathbb{F}}, c)$$

où $(x_{\mathbb{F}}, c)$ est le nombre d'occurrences de $x_{\mathbb{F}}$ dans la contrainte c.

5.2 Propriétés basées sur les contraintes

Dans cette section, nous introduisons un ensemble de propriétés qui exploitent les opérations sur les nombres flottants. Ces propriétés ont un rôle essentiel dans la définition de stratégies d'exploration guidées par les contraintes.

Les propriétés présentées dans cette section s'appliquent aux contraintes. Pour chacune de ces propriétés, on considère la contrainte élémentaire $c_i(x_{\mathbb{F}}, y_{\mathbb{F}})$ sur $\mathbb{F}$:

$$x_{\mathbb{F}} \odot y_{\mathbb{F}} = z_{\mathbb{F}} \text{ , avec } \odot \in \{\oplus, \ominus, \otimes, \oslash\}$$

où le mode d'arrondi est "au plus près pair".

5.2.1 Absorption

Si le domaine $x_{\mathbb{F}}$ a une magnitude significativement plus grande que $y_{\mathbb{F}}$, certaines valeurs de $y_{\mathbb{F}}$ peuvent simplement être absorbées lors de l'exécution de l'addition. Mesurer quelle quantité du domaine de $y_{\mathbb{F}}$ est ainsi absorbée est le but de la propriété d'absorption. L'absorption est souvent la source d'une déviation importante du calcul sur les flottants vis-à-vis du même calcul interprété sur les réels. Par conséquent, se concentrer sur ces phénomènes à pour objectif de trouver plus rapidement un contre-exemple.

Cette propriété définie le nombre de valeurs de $\mathbf{y}_{\mathbb{F}}$ qui sont absorbées par au moins une valeur de $\mathbf{x}_{\mathbb{F}}$.

Définition 7 (Absorption) *Soit* $absorb(\mathbf{y}_{\mathbb{F}}, \mathbf{x}_{\mathbb{F}}, c_i)$ *l'absorption de* $\mathbf{y}_{\mathbb{F}}$ *par* $\mathbf{x}_{\mathbb{F}}$ *dans la contrainte élémentaire* $c_i(x_{\mathbb{F}}, y_{\mathbb{F}})$ *définie par :*

$$absorb(\mathbf{y}_{\mathbb{F}}, \mathbf{x}_{\mathbb{F}}, c_i) = \begin{cases} \frac{|[-2^{e_{max}(x_{\mathbb{F}})-p-1}, 2^{e_{max}(\mathbf{x}_{\mathbb{F}})-p-1}] \cap \mathbf{y}_{\mathbb{F}}|}{|\mathbf{y}_{\mathbb{F}}|} & si \odot \in \{\oplus, \ominus\} \\ 0 \ sinon \end{cases}$$

avec $\mathbf{y}_{\mathbb{F}} \neq 0$ *et* $e_{max}(\mathbf{x}_{\mathbb{F}}) = max\{abs(e_{\underline{x}_{\mathbb{F}}}), abs(e_{\overline{x}_{\mathbb{F}}})\}$ *où e représente l'exposant.*

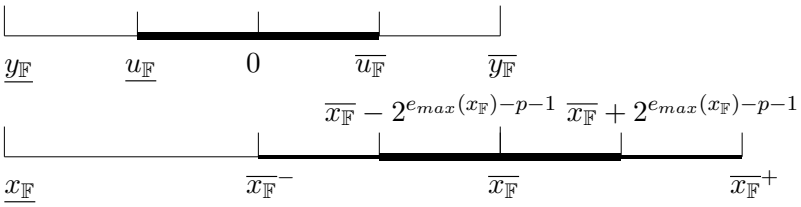


FIGURE 5.2 – illustration d'un phénomène d'absorption

Notons que $\mathbf{u}_{\mathbb{F}} = [-2^{e_{max}(\mathbf{x}_{\mathbb{F}})-p-1}, 2^{e_{max}(\mathbf{x}_{\mathbb{F}})-p-1}] \cap \mathbf{y}_{\mathbb{F}}$ capture la partie de $\mathbf{y}_{\mathbb{F}}$ qui est absorbée par des valeurs de plus grande magnitude de $\mathbf{x}_{\mathbb{F}}$. Les valeurs de plus grande magnitude de $\mathbf{x}_{\mathbb{F}}$ sont celles qui absorbent le plus de valeurs de $\mathbf{y}_{\mathbb{F}}$. D'autre part, étant donné deux nombres flottants x_1 et x_2, appartenant à $\mathbf{x}_{\mathbb{F}}$; si $0 \leq abs(\underline{x}_{\mathbb{F}}) < abs(x_1) < abs(x_2)$, alors $absorb(\mathbf{y}_{\mathbb{F}}, [\underline{x}_{\mathbb{F}}, \mathbf{x_1}_{\mathbb{F}}], c_i) \subset absorb(\mathbf{y}_{\mathbb{F}}, [\underline{x}_{\mathbb{F}}, \mathbf{x_2}_{\mathbb{F}}], c_j)$. Le flottant de plus grande magnitude de $\mathbf{x}_{\mathbb{F}}$ est donc suffisant pour déterminer le plus grand sous-ensemble de valeur de $\mathbf{y}_{\mathbb{F}}$ absorbée par $\mathbf{x}_{\mathbb{F}}$.

Si aucune valeur de $\mathbf{y}_{\mathbb{F}}$ n'est absorbée par $\mathbf{x}_{\mathbb{F}}$, $\mathbf{u}_{\mathbb{F}}$ est vide et $absorb(\mathbf{y}_{\mathbb{F}}, \mathbf{x}_{\mathbb{F}}, c_i)$ est égal à 0. À l'opposé, quand toutes les valeurs de $\mathbf{y}_{\mathbb{F}}$ sont absorbées par

$\mathbf{x}_\mathbb{F}$, $\mathbf{u}_\mathbb{F}$ devient égal à $\mathbf{y}$ et $absorb(\mathbf{y}_\mathbb{F}, \mathbf{x}_\mathbb{F}, c_i)$ est égal à 1.

Choisir les variables qui sont impliquées dans une absorption peut permettre d'améliorer la qualité des logiciels et de fournir des contre-exemples qui génèrent une absorption. La taille des domaines des variables flottantes considérées est, en général, très grande. Identifier les parties des domaines effectivement impliquées dans ces absorptions est aussi important pour pouvoir tirer totalement profit des absorptions dans la recherche de contre-exemples.

5.2.2 Cancellation

De la même manière que les phénomènes d'absorption, les phénomènes de cancellation entrainent souvent des déviations importantes dans le calcalculcule sur les flottants.

Définition 8 (Cancellation) *$cancellation(\mathbf{y}_\mathbb{F}, \mathbf{x}_\mathbb{F}, c_i)$ dénote le nombre de bits éliminés par la soustraction et est définie par*

$$cancellation(\mathbf{y}_\mathbb{F}, \mathbf{x}_\mathbb{F}, c_i) = \begin{cases} max\{e_{\underline{x}_\mathbb{F}}, e_{\overline{x}_\mathbb{F}}, e_{\underline{y}_\mathbb{F}}, e_{\overline{y}_\mathbb{F}}\} - min\{e_{z_\mathbb{F}}, z_\mathbb{F} \in \mathbf{z}_\mathbb{F}\} \ si \odot \in \{\oplus, \ominus\} \\ 0 \ sinon \end{cases}$$

La définition de cette propriété est extraite de [Benz et al., 2012]. Cette valeur augmente avec le nombre de bits éliminés. Lorsqu'elle est strictement positive, certains bits sont potentiellement éliminés.

Néanmoins, cette propriété ne permet de différencier les cancellations bénignes de celles qui sont catastrophiques. Déterminer le type de la cancellation est un point clé. Une cancellation bénigne n'a en général pas d'impact direct sur la déviation du calcul flottant vis a vis de son interpretation sur les réels. À contrario, la cancellation catastrophique peut avoir un impact fort sur la deviation du calcul. Affiner cette propriété est une des perspectives de poursuite de cette thèse.

5.2.3 Dérivée

Dans une contrainte monovariée, 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. Ces informations peuvent être utiles pour orienter la recherche. Cette propriété n'a pas été exploitée dans cette thèse et s'inscrit aussi dans les perspectives de poursuite de ces travaux.

Définition 9 (Derivative) *Soit une contrainte* $c_i : e_1 \Diamond e_2$ *où* $\Diamond \in \{=, \leq, \geq, <, >\}$, c_i *peut être réécrit sous la forme* $f : e_1 - e_2 \Diamond 0$. *Si* f *est une fonction monovariée, sa dérivée peut-être évaluée en utilisant l'arithmétique des intervalles et la définition de la dérivée de la contrainte élémentaire* $c_i(x_{\mathbb{F}}, y_{\mathbb{F}})$ *est :*

$$derive(\mathbf{y}_{\mathbb{F}}, \mathbf{x}_{\mathbb{F}}, c_i) = \mathbf{f}'(\mathbf{x}_{\mathbb{F}}) \approx \frac{\mathbf{f}_{\mathbb{F}}(\mathbf{x}_{\mathbb{F}} + h) - \mathbf{f}_{\mathbb{F}}(\mathbf{x}_{\mathbb{F}})}{h}$$

Cette approche se généralise lorsque f est une fonction multivariée. Sa Jacobienne J donne la variation de chaque variable de f par rapport aux autres variables de f. En utilisant sa matrice, soit par composante, soit en calculant une agrégation de la variation (i.e l'écart type) de chaque variable par rapport aux autres, il est possible d'estimer le rôle d'une variable de f dans la variation de f.

5.3 Conclusion

Dans ce chapitre, nous avons présenté un ensemble de propriétés. Parmi ces propriétés certaines sont classiques sur les domaines finis (e.g., la taille, le degré,...), d'autres sont spécifiques aux domaines flottants (e.g., la densité, l'absorption et la cancellation). Ces propriétés sont la base des stratégies que nous avons développées. Ces stratégies sont détaillées dans le chapitre suivant. L'objectif de ces stratégies est de tirer profit des spécificités des flottants pour guider la recherche de contre-exemple.

Chapitre 6

Stratégies de choix de variables dédiées aux flottants

Dans ce chapitre, nous présentons et évaluons un ensemble de stratégies de choix de variables dédiées à la vérification de programmes dans les approches BMC. Ces stratégies de choix de variables cherchent à tirer profit des spécificités des flottants pour guider la recherche de contre-exemples. Ces stratégies sont basées sur les propriétés présentées dans le chapitre précédent à la page 79. Certaines de ces propriétés sont liées aux domaines des flottants dont la répartition n'est pas uniforme. D'autres reposent sur le comportement particulier des flottants qui conduit à des phénomènes tels que l'absorption ou la cancellation.

Dans la section suivante, nous présentons une première version des stratégies de choix de variables. Nous avons introduit toutes ces stratégies dans [Zitoun et al., 2017]. Nos expérimentations nous ont conduits à améliorer ces stratégies qui sont détaillées dans la section 6.2. Toutes les stratégies présentées dans ce chapitre sont comparées en utilisant une `bissection` comme stratégie de choix de valeurs (i.e sous-domaines). Pour rappel, la `bissection` est une stratégie de choix de sous-domaine qui à partir du domaine $\mathbf{x}_{\mathbb{F}}$, génère deux sous-domaines : $[\underline{x}_{\mathbb{F}}, mid_{\mathbb{F}}]$ et $[mid^{+}_{\mathbb{F}}, \overline{x}_{\mathbb{F}}]$, avec $mid_{\mathbb{F}} = \frac{\underline{x}_{\mathbb{F}} + \overline{x}_{\mathbb{F}}}{2}$.

6.1 Stratégies mono-propriété : version de base

Les stratégies mono-propriété sont des stratégies qui n'utilisent qu'une seule propriété pour choisir la prochaine variable. Ces stratégies choisissent la variable maximisant ou minimisant la propriété choisie. Par exemple, elle peut choisir la variable qui maximise la densité du domaine ou celle qui

minimise cette densité.

Pour chaque propriété `prop`, deux stratégies de choix de variables sont définies : `var_MaxProp` et `var_MinProp`. Pour les définitions, les stratégies basées sur les propriétés des domaines des variables sont indicées par V. Celles basées sur les propriétés des contraintes sont indicées par C.

6.1.1 Stratégies basées sur les domaines des variables

Chaque stratégie choisit la prochaine variable $next\_var_{\mathbb{F}}$ parmi l'ensemble des variables non instanciées $X^*_{\mathbb{F}}$. Son domaine est $\mathbf{next\_var}_{\mathbb{F}}$. Ces stratégies sont définies ci-dessous :

Définition 10 *$var\_MaxProp_V$*

$$prop(\mathbf{next\_var}_{\mathbb{F}}) = max_{x_{\mathbb{F}} \in X^*_{\mathbb{F}}}\{prop(\mathbf{x}_{\mathbb{F}})\} \ avec \ next\_var_{\mathbb{F}} \in X^*_{\mathbb{F}}$$

Définition 11 *$var\_MinProp_V$*

$$prop(\mathbf{next\_var}_{\mathbb{F}}) = min_{x_{\mathbb{F}} \in X^*_{\mathbb{F}}}\{prop(\mathbf{x}_{\mathbb{F}})\} \ avec \ next\_var_{\mathbb{F}} \in X^*_{\mathbb{F}}$$

Dans la suite, nous détaillons les différentes stratégies basée sur les domaines des variables.

6.1.1.1 Stratégies basées sur la taille : `var_MaxDom` et `var_MinDom`

Les stratégies basées sur la taille sont des adaptations directes des stratégies existantes sur les flottants. Par exemple, la stratégie `var_MinDom` est une adaptation directe de la stratégie `min-domain` sur les domaines finis. L'intuition de cette stratégie sur les domaines finis est de choisir la variable avec le plus petit domaine. Cette variable correspond à la variable qui contraint le plus le problème.

Ceci n'est plus valide sur les flottants, mais nous avons quand même défini cette stratégie par souci de complétude. Cette stratégie choisit la variable $next\_var_{\mathbb{F}}$ dont le domaine $\mathbf{next\_var}_{\mathbb{F}}$ est le plus grand parmi les variables non instanciées $X^*_{\mathbb{F}}$. La taille du domaine de la variable $x_{\mathbb{F}}$ est notée $w(\mathbf{x}_{\mathbb{F}}) = \overline{x}_{\mathbb{F}} - \underline{x}_{\mathbb{F}}$.

Plus formellement, `var_MaxDom` est défini de la manière suivante :

$$w(\mathbf{next\_var}_{\mathbb{F}}) = max_{x_{\mathbb{F}} \in X^*_{\mathbb{F}}}\{w(\mathbf{x}_{\mathbb{F}})\} \text{ avec } next\_var_{\mathbb{F}} \in X^*_{\mathbb{F}}$$

De la même manière, `var_MinDom` est défini de la manière suivante :

$$w(\mathbf{next\_var}_{\mathbb{F}}) = max_{x_{\mathbb{F}} \in X^*_{\mathbb{F}}}\{w(\mathbf{x}_{\mathbb{F}})\} \text{ avec } next\_var_{\mathbb{F}} \in X^*_{\mathbb{F}}$$

6.1.1.2 Stratégies basées sur la cardinalité : `var_MaxCard` et `var_MinCard`

La stratégie `var_MaxCard` choisit la variable qui contient le plus de valeurs. L'intuition de cette stratégie est de choisir les domaines contenant potentiellement le plus de solutions, du fait du grand nombre de flottants.

Cette stratégie choisit la variable $next\_var_{\mathbb{F}}$ dont le domaine $\mathbf{next\_var}_{\mathbb{F}}$ est le contient le plus de flottants parmi les variables non instanciées $X^*_{\mathbb{F}}$. La cardinalité du domaine de la variable $x_{\mathbb{F}}$ est notée $|\mathbf{x}_{\mathbb{F}}| = 2^p * (e_{\overline{x}_{\mathbb{F}}} - e_{\underline{x}_{\mathbb{F}}}) + m_{\overline{x}_{\mathbb{F}}} - m_{\underline{x}_{\mathbb{F}}} + 1$. Plus formellement, `var_MaxCard` est défini de la manière suivante :

$$|\mathbf{next\_var}_{\mathbb{F}}| = max_{x_{\mathbb{F}} \in X^*_{\mathbb{F}}}\{|(\mathbf{x}_{\mathbb{F}})|\} \text{ avec } next\_var_{\mathbb{F}} \in X^*_{\mathbb{F}}$$

La stratégie `var_MinCard`, reprend l'intuition de la stratégie `min-domain` sur les entiers. Cette stratégie choisit la variable qui contient le moins de flottants, et donc qui contraint le plus le problème.

Plus formellement, `var_MinCard` est défini de la manière suivante :

$$|\mathbf{next\_var}_{\mathbb{F}}| = min_{x_{\mathbb{F}} \in X^*_{\mathbb{F}}}\{|(\mathbf{x}_{\mathbb{F}})|\} \text{ avec } next\_var_{\mathbb{F}} \in X^*_{\mathbb{F}}$$

6.1.1.3 Stratégies basées sur la densité : `var_MaxDens` et `var_MinDens`

La stratégie `var_MaxDens` repose sur l'intuition suivante : plus un domaine est dense, plus le nombre de solutions potentielles est conséquent.

Cette stratégie choisit la variable $next\_var_{\mathbb{F}}$ dont le domaine $\mathbf{next\_var}_{\mathbb{F}}$ est le plus dense parmi l'ensemble des variables non instanciées $X^*_{\mathbb{F}}$. La densité du domaine de la variable $x_{\mathbb{F}}$ est notée $\rho(\mathbf{x}_{\mathbb{F}}) = \frac{|\mathbf{x}_{\mathbb{F}}|}{w(\mathbf{x}_{\mathbb{F}})}$, où $w(\mathbf{x}_{\mathbb{F}})$ représente la taille du domaine $\mathbf{x}_{\mathbb{F}}$. Plus formellement, `var_MaxDens` est défini de la manière suivante :

$$\rho(\mathbf{next\_var}_{\mathbb{F}}) = max_{x_{\mathbb{F}} \in X^*_{\mathbb{F}}}\{\rho(\mathbf{x}_{\mathbb{F}})\} \text{ avec } next\_var_{\mathbb{F}} \in X^*_{\mathbb{F}}$$

L'un des objectifs de la stratégie `var_MinDens` est de rechercher des solutions avec des valeurs proches des infinis. Formellement, `var_MinDens` est défini par :

$$\rho(\mathbf{next\_var}_{\mathbb{F}}) = min_{x_{\mathbb{F}} \in X^*_{\mathbb{F}}}\{\rho(\mathbf{x}_{\mathbb{F}})\} \text{ avec } next\_var_{\mathbb{F}} \in X^*_{\mathbb{F}}$$

6.1.1.4 Stratégies basées sur la magnitude : `var_MaxMagn` et `var_MinMagn`

La stratégie `var_MaxMagn` permet de choisir la variable dont le domaine est constitué principalement de grandes valeurs. Ces valeurs conduisent souvent à des comportements inattendus (e.g., **overflow**, **absorption**).

Cette stratégie choisit la variable $next\_var_{\mathbb{F}}$ dont le domaine $\mathbf{next\_var}_{\mathbb{F}}$ est à la plus grande a magnitude parmi l'ensemble des variables non instanciées $X^*_{\mathbb{F}}$. La magnitude du domaine de la variable $x_{\mathbb{F}}$ est notée $mag(\mathbf{x}_{\mathbb{F}}) = \frac{e_{\underline{x}_{\mathbb{F}}} + e_{\overline{x}_{\mathbb{F}}}}{2 \cdot e_{max}}$ où e_{max} est le plus grand exposant représentable dans le format choisi. Plus formellement, `var_MaxMagn` est défini de la manière suivante :

$$mag(\mathbf{next\_var}_{\mathbb{F}}) = max_{x_{\mathbb{F}} \in X^*_{\mathbb{F}}}\{mag(\mathbf{x}_{\mathbb{F}})\} \text{ avec } next\_var_{\mathbb{F}} \in X^*_{\mathbb{F}}$$

De la même manière, la stratégie `var_MinMagn` permet de choisir la variable dont le domaine est constitué principalement de petites valeurs (valeurs proches de zéro). Si la magnitude est faible, la densité du domaine est forte et donc le nombre de solutions est potentiellement fort.

Plus formellement, `var_MinMagn` est défini de la manière suivante :

$$mag(\mathbf{next\_var}_{\mathbb{F}}) = min_{x_{\mathbb{F}} \in X^*_{\mathbb{F}}}\{mag(\mathbf{x}_{\mathbb{F}})\} \text{ avec } next\_var_{\mathbb{F}} \in X^*_{\mathbb{F}}$$

6.1.1.5 Stratégies basées sur la degrée : `var_MaxDegre` et `var_MinDegre`

La stratégie `var_MaxDegre` est une adaptation de la stratégie `weighted-degree` [Boussemart et al., 2004] sur les domaines finis. Cette stratégie cherche à identifier les variables qui ont un impact fort sur la résolution du problème.

Cette stratégie choisit la variable $next\_var_{\mathbb{F}}$ dont le degrée est le plus grand l'ensemble des variables non instanciées $X^*_{\mathbb{F}}$. Le degré d'une variable $x_{\mathbb{F}}$ est notée $degree(\mathbf{x}_{\mathbb{F}}) = \sum_{c \in C} (x_{\mathbb{F}} \in vars(c))$ où C est l'ensemble des contraintes et $vars(c)$ les variables impliquées dans la contrainte c. Plus formellement, `var_MaxDegre` est défini de la manière suivante :

$$degree(next\_var_{\mathbb{F}}) = max_{x_{\mathbb{F}} \in X^*_{\mathbb{F}}}\{degree(\mathbf{x}_{\mathbb{F}})\} \text{ avec } next\_var_{\mathbb{F}} \in X^*_{\mathbb{F}}$$

La stratégie `var_MinDegree` à moins d'intérêt, car cette stratégie choisit la variable qui a moins d'impact sur la résolution du CSP. Plus formellement, `var_MinDegree` est défini de la manière suivante :

$$degree(\mathbf{next\_var}_{\mathbb{F}}) = min_{x_{\mathbb{F}} \in X^*_{\mathbb{F}}}\{degree(\mathbf{x}_{\mathbb{F}})\} \text{ avec } next\_var_{\mathbb{F}} \in X^*_{\mathbb{F}}$$

6.1.1.6 Stratégies basées sur les occurrences multiples : `var_MaxOcc` et `var_MinOcc`

L'intuition de la stratégie `var_MaxOcc` est de tirer profit des occurrences multiples qui permettent de capturer l'importance d'une variable au sein d'une contrainte. Les réductions sur ces variables ont une répercussion forte sur les autres variables impliquées dans cette contrainte.

Cette stratégie choisit la variable $next\_var_{\mathbb{F}}$ qui à le plus d'occurrences au sein d'une même contrainte parmi l'ensemble des variables non instanciées $X^*_{\mathbb{F}}$. Le nombre d'occurrence d'une variable $x_{\mathbb{F}}$ est notée $occur(x_{\mathbb{F}}) = \max_{c \in C}(x_{\mathbb{F}}, c)$ où $(x_{\mathbb{F}}, c)$ est le nombre d'occurrences de $x_{\mathbb{F}}$ dans la contrainte c. Plus formellement, `var_MaxOcc` est défini de la manière suivante :

$$occur(next\_var_{\mathbb{F}}) = max_{x_{\mathbb{F}} \in X^*_{\mathbb{F}}}\{occur(\mathbf{x}_{\mathbb{F}})\} \text{ avec } next\_var_{\mathbb{F}} \in X^*_{\mathbb{F}}$$

La stratégie `var_MinOcc` cherche à éviter ces variables aux occurrences multiples. Les occurrences multiples sont difficiles à gérer avec les consistances simples et nécessitent des consistances fortes qui sont couteuses. Cette stratégie évite ces variables, en espérant que les réductions faites sur les autres variables entrainent des réductions sur les domaines de variables aux occurrences multiples.

Plus formellement, `var_MinOcc` est défini de la manière suivante :

$$occur(\mathbf{next\_var}_{\mathbb{F}}) = min_{x_{\mathbb{F}} \in X^*_{\mathbb{F}}}\{occur(\mathbf{x}_{\mathbb{F}})\} \text{ avec } next\_var_{\mathbb{F}} \in X^*_{\mathbb{F}}$$

6.1.2 Stratégies basées sur les contraintes

Les stratégies basées sur les variables n'ont besoin que d'examiner l'ensemble des variables pour choisir la prochaine variable à considérer. Les stratégies basées sur les contraintes doivent aussi examiner pour chaque variable l'ensemble des contraintes élémentaires où la variable considérée est impliquée avant de choisir la variable la plus pertinente vis-à-vis de la propriété. En d'autres termes, étant donnée la propriété sur les contraintes $prop(x_{\mathbb{F}})$, $T\_prop(x_{\mathbb{F}})$ calcul la somme de ces propriétés sur chaque contrainte élémentaire $c_i(x_{\mathbb{F}}, y_{\mathbb{F}})$ où $x_{\mathbb{F}}$ est impliquée. Plus formellement, $T\_prop(x_{\mathbb{F}})$ est calculé de la manière suivante :

$$T\_prop(x_{\mathbb{F}}) = \sum_{c_i(x_{\mathbb{F}}, y_{\mathbb{F}}) \ \in \ Contraintes(x_{\mathbb{F}})} prop(\mathbf{y}_{\mathbb{F}}, \mathbf{x}_{\mathbb{F}}, c_i)$$

Les stratégies $\texttt{var\_MaxProp}_C$ et $\texttt{var\_MinProp}_C$ basées sur les contraintes sont donc définies par :

Définition 12 *$\texttt{var\_MaxProp}_C$*

$$T\_prop(next\_var_{\mathbb{F}}) = max_{x_{\mathbb{F}} \in X^*_{\mathbb{F}}}\{T\_prop(x_{\mathbb{F}})\} \ avec \ next\_var_{\mathbb{F}} \in X^*_{\mathbb{F}}$$

Définition 13 *$\texttt{var\_MinProp}_C$*

$$T\_prop(next\_var_{\mathbb{F}}) = min_{x_{\mathbb{F}} \in X^*_{\mathbb{F}}}\{T\_prop(x_{\mathbb{F}})\} \ avec \ next\_var_{\mathbb{F}} \in X^*_{\mathbb{F}}$$

Notons que les contraintes prises en compte ici sont des contraintes élémentaires binaires ou ternaires. Ces contraintes proviennent de la décomposition des contraintes du problème initial en un ensemble de contraintes de base. Rappelons que cela se justifie par le fait que la décomposition des contraintes en contraintes élémentaires est basée sur le mode d'évaluation des expressions correspondantes. Toutes les contraintes non élémentaires peuvent être décomposées en introduisant des variables supplémentaires. Ces variables supplémentaires permettent de représenter les sous-expressions de la contrainte initiale.

Dans la suite, nous détaillons les stratégies basées sur les contraintes.

6.1.2.1 Stratégies basées sur l'absorption : `var_MaxAbs` et `var_MinAbs`

L'absorption est un phénomène propre à l'arithmétique des flottants. Ces phénomènes qui n'existent ni sur les domaines réels, ni sur les entiers sont souvent une des sources des bugs dans les programmes. L'absorption peut entrainer une déviation importante du calcul sur les flottants vis-à-vis du même calcul interprété sur les réels. L'idée de cette stratégie est donc de tirer profit de ces phénomènes pour trouver des contre-exemples plus rapidement.

La stratégie `var_MaxAbs` est une stratégie axée sur les contraintes. Pour l'absorption, nous avons donc besoin de mesurer le taux d'absorption local à chaque contrainte. Dans la suite, $absorb(\mathbf{y}_\mathbb{F}, \mathbf{x}_\mathbb{F}, c_i)$ définit le taux d'absorption de la variable $x_\mathbb{F}$ pour la partie du domaine de $\mathbf{y}_\mathbb{F}$ qui est absorbée par celui de la variable $x_\mathbb{F}$ dans la contrainte c_i.

Plus formellement, soit la contrainte binaire $c(x_\mathbb{F}, y_\mathbb{F})$ suivante :

$$x_\mathbb{F} \odot y_\mathbb{F} = z_\mathbb{F} \text{ , avec } \odot \in \{\oplus, \ominus, \otimes, \oslash\}$$

Le taux d'absorption de $x_\mathbb{F}$ pour cette contrainte est :

$$absorb(\mathbf{y}_\mathbb{F}, \mathbf{x}_\mathbb{F}, c_i) = \begin{cases} \frac{|[-2^{e_{max}(x_\mathbb{F})-p-1}, 2^{e_{max}(\mathbf{x}_\mathbb{F})-p-1}] \cap \mathbf{y}_\mathbb{F}|}{|\mathbf{y}_\mathbb{F}|} \text{ si } \odot \in \{\oplus, \ominus\} \\ 0 \text{ sinon} \end{cases}$$

avec $\mathbf{y}_\mathbb{F} \neq 0$ et $e_{max}(\mathbf{x}_\mathbb{F}) = max\{abs(e_{\underline{x}_\mathbb{F}}), abs(e_{\overline{x}_\mathbb{F}})\}$ où e représente l'exposant.

Cette propriété est détaillée dans la section 5.2.1 du chapitre 5. La Figure 6.1 rappelle les domaines impliqués par cette propriété.

La stratégie `var_MaxAbs` choisit la variable $next\_var_\mathbb{F}$ qui a le plus grand potentiel d'absorption parmi l'ensemble des variables non instanciées $X^*_\mathbb{F}$. Le

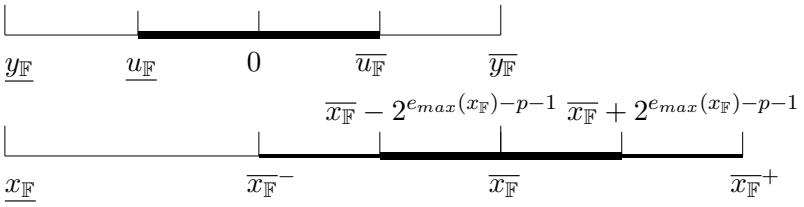


FIGURE 6.1 – illustration de l'absorption

potentiel d'absorption de $x_{\mathbb{F}}$, noté $T\_absorb(x_{\mathbb{F}})$, est la somme des taux d'absorption $absorb(\mathbf{y}_{\mathbb{F}}, \mathbf{x}_{\mathbb{F}}, c_i)$ pour chaque contrainte élémentaire $c_i(x_{\mathbb{F}}, y_{\mathbb{F}})$ où $x_{\mathbb{F}}$ est impliquée. Plus formellement,

$$T\_absorb(x_{\mathbb{F}}) = \sum_{c_i(x_{\mathbb{F}},y_{\mathbb{F}}) \in Contraintes(x_{\mathbb{F}})} absorb(\mathbf{y}_{\mathbb{F}}, \mathbf{x}_{\mathbb{F}}, c_i)$$

Notons que cette stratégie privilégie une variable qui absorbe globalement le plus à une variable qui absorbe davantage localement. En d'autres termes, cette stratégie peut choisir une variable qui absorbe peu sur beaucoup de contraintes, plutôt qu'une variable qui absorbe beaucoup sur une seule contrainte. La seconde alternative mériterait aussi d'être explorée et s'inscrit dans les perspectives de ces travaux.

Finallement, `var_MaxAbs` est défini par :

$$T\_absorb(next\_var_{\mathbb{F}}) = max_{x_{\mathbb{F}} \in X^*_{\mathbb{F}}}\{T\_absorb(x_{\mathbb{F}})\} \text{ avec } next\_var_{\mathbb{F}} \in X^*_{\mathbb{F}}$$

La stratégie `var_MinAbs`, choisit les variables qui n'ont pas où peu d'absorption en premier. Cette stratégie n'a à priori aucun intérêt pour la recherche de contre exemples. Ce qui a été confirmé par les expérimentations ; par soucis de concision nous ne la mentionneront pas dans la partie évaluation.

6.1.2.2 Stratégies basées sur la cancellation : `var_MaxCanc` et `var_MinCanc`

Comme les absorption, les cancellation sont la source de fortes déviations. L'intuition de cette stratégie est donc de tirer profit de ces phénomènes, pour trouver plus rapidement un contre-exemple.

La stratégie `var_MaxCanc` calcule le taux de cancellation de chaque variable. Et choisit la variable $next\_var_{\mathbb{F}}$ qui maximise ce taux parmis les variables non instanciées $X^*_{\mathbb{F}}$. Le taux de cancellation de $x_{\mathbb{F}}$, noté $T\_canc(x_{\mathbb{F}})$,

est la somme des taux de cancellation $cancellation(\mathbf{y}_{\mathbb{F}}, \mathbf{x}_{\mathbb{F}}, c_i)$ pour chaque contrainte élémentaire $c_i(x_{\mathbb{F}}, y_{\mathbb{F}})$ où $x_{\mathbb{F}}$ est impliquée. Plus formellement,

$$T\_canc(x_{\mathbb{F}}) = \sum_{c_i(x_{\mathbb{F}}, y_{\mathbb{F}}) \in Contraintes(x_{\mathbb{F}})} cancellation(\mathbf{y}_{\mathbb{F}}, \mathbf{x}_{\mathbb{F}}, c_i)$$

Comme `var_MaxAbs` la stratégie `var_MaxCanc` privilégie une variable impliquée dans un ensemble de cancellation plutôt qu'une variable qui va être impliqué dans une seule cancellation plus importante mais avec un taux de cancellation plus faible.

Finallement, `var_MaxCanc` est défini par :

$$T\_canc(next\_var_{\mathbb{F}}) = max_{x_{\mathbb{F}} \in X^*_{\mathbb{F}}}\{T\_canc(x_{\mathbb{F}})\} \text{ avec } next\_var_{\mathbb{F}} \in X^*_{\mathbb{F}}$$

La stratégie `var_MinCanc`, choisit les variables qui ne sont pas impliquées dans une cancellation en premier. Comme `var_MinAbs`, cette stratégie n'a aucun intérêt pour la recherche de contre exemples.

6.1.3 Resélection de variable

Lors de la recherche, se focaliser sur une même variable peut être contre-productif et il peut être judicieux d'interdire la resélection de la même variable au nœud suivant. Le fait de se concentrer sur la même variable dans le haut de l'arbre de recherche peut conduire à une bonne réduction pendant la propagation. Mais après quelques resélections, ces réductions ont moins d'impact. Forcer à changer la variable lors de la recherche mène à un problème plus contraint et devrait avoir globalement plus d'impact sur le filtrage. De plus, les stratégies de choix de variables utilisées ici maximisent ou minimisent une propriété. En refusant la resélection, d'autres bons candidats sont également choisis. C'est pourquoi, nous avons évalué l'ensemble des stratégies avec et sans resélection de variable.

6.1.4 Évaluation de ces stratégies

Dans cette section, nous présentons une comparaison expérimentale de deux stratégies de choix de variables spécifiques `var_MaxAbs` et `var_MaxDens` avec la stratégie `var_Lex` sur un ensemble de benchmarks issus de la vérification de programmes. Ces deux stratégies sont bien plus efficaces que les stratégies moins dédiées comme le degré, ou la taille. Les expérimentations

sur les autres stratégies (moins spécifique) ne sont pas détaillées dans ce chapitre, mais elles sont disponibles dans [Zitoun et al., 2017] et dans l'annexe 1.

Les sources principales des benchmarks sont la SMTLib [Barrett et al., 2016], la suite FPBench [Damouche et al., 2017], et CBMC [Clarke et al., 2004] (mais aussi [Collavizza et al., 2010, Collavizza et al., 2016, D'Silva et al., 2012a]). L'ensemble des benchmarks utilisés est constitué de 48 benchmarks avec solutions, et 47 benchmarks sans solution. L'ensemble de ces benchmarks est disponible à l'adresse `http://www.github.com/zitounh/benchmarks/`. Les stratégies présentées ne tiennent pas compte du fait qu'un benchmark est satisfiable ou non. Une liste exhaustive détaillant les sources, le nombre de variables et contraintes de ces benchmarks est explicitée en annexe. Sur l'ensemble des 48 benchmarks "satisfiables" (i.e Il existe au moins une solution qui viole la spécification), 17 benchmarks contiennent au moins une solution liée à un problème d'absorption.

6.1.4.1 Protocole expérimental

Toutes les expérimentations ont été réalisées avec un système Linux, et un processeur Intel Xeon fonctionnant à 2,40 GHz et 12 Go de mémoire. Toutes les stratégies ont été implantées dans le solveur Objective-CP. Tous les calculs en virgule flottante sont effectués avec des nombres flottants simples précisions et avec un mode d'arrondi "au plus près pair". Tous les temps sont en secondes, et le timeout est d'une minute.

6.1.4.2 Filtrage utilisé

Les expérimentations présentées dans ce chapitre utilisent deux filtrages différents : la 2B-consistance et la 3B-consistance. Ces consistances sont définies dans la section 3.5.2 à la page 53. Intuitivement, la 2B-consistance correspond à une relaxation de la consistance d'arc aux bornes des intervalles. La 3B-consistance quant à elle est une consistance plus forte que la 2B-consistance. L'idée de la 3B-consistance est de vérifier si le CSP reste consistant lorsque la variable x_i est instanciée à l'une de ces bornes.

Sur les domaines flottants, ces filtrages peuvent conduire à une énumération complète des domaines en retirant un flottant après l'autre. Lors de ces comportements de convergence lente, chaque modification est propagée ce qui peut conduire à une 2B-consistance (ou 3B-consistance) très couteuse. Pour éviter ces convergences lentes, nous avons préféré implanter la 2B(w)-consistance et la 3B(w)-consistence. Ces consistances, ne propage les

modifications que lorsqu'elles sont supérieures à $w\%$ de la taille du domaine restant. La valeur de w utilisé pour les expérimentations est 5%.

Notons que la 3B(5)-consistance que nous avons implantée est encore plus restrictive : biys ne cherchons pas à atteindre le point fixe, nous n'appliquons le filtrage qu'une seule fois.

6.1.4.3 Résultats

La Table 6.1 présente les résultats de ces différentes stratégies sur cet ensemble de benchmarks. La table 6.1a donne la somme des temps des stratégies pour les benchmarks avec solutions, c'est-à-dire le temps nécessaire pour montrer qu'il existe un contre-exemple sur tous ces benchmarks. La table 6.1b donne quant à elle la somme des temps de résolution sur les benchmarks sans solution, c'est-à-dire le temps nécessaire pour montrer qu'il n'existe pas de contre-exemple sur tous ces benchmarks. Ces stratégies sont comparées à un choix de variables `lexicographique` [1] (notée `var_lex`).

Dans ces tables les stratégies sont triées par temps de résolutions décroissant.

La première colonne présente le choix de variables utilisé. Cette colonne est divisée en deux sous colonnes : la stratégie de choix de variables (notée "strat.") et l'autorisation "y", ou non "n" de la resélection de la même variable au noeud suivant (colonne "sel."). La colonne "Filt." explicite le filtrage utilisé. La colonne "TO" comptabilise le nombre de benchmarks non résolus (le timeout est d'une minute). Enfin, la colonne $\sum t$ donne le nombre de secondes nécessaires pour la résolution de l'ensemble des 48 benchmarks avec solutions, et 47 benchmarks sans solution, selon les stratégies choisies. Pour chaque stratégie, en cas de timeout le temps comptabilisé est d'une minute.

La stratégie `var_Lex` est meilleure que les stratégies plus spécifique `var_MaxAbs` et `var_MaxDens` sur les benchmarks avec et sans solution. Les stratégies `var_MaxDens` et `var_MaxAbs` ont des résultats comparables sur les benchmarks avec solutions. À l'inverse, la stratégie basée sur la densité est peu efficace sur les benchmarks sans solution. Sur les benchmarks sans solution, la somme des temps de résolution de la stratégie `var_MaxAbs` est assez proche de celle de la stratégie `var_Lex`.

La stratégie `var_MaxAbs` est moins bonne que la stratégie `var_Lex` pour résoudre l'ensemble de ces benchmarks avec solution, car tous ces benchmarks n'ont pas forcément une absorption qui joue un rôle déterminant pour trouver

1. Du fait du renommage des variables par des entiers dans ObjectifCP, l'ordre LEX correspond en fait à l'ordre de déclaration des varaibles dans le CSP (cf. `http://www.github.com/zitounh/benchmarks/`)

choix de var.		Filt.	TO	$\sum$ t
strat.	sel.			(s)
var_Lex	n	2B(5)	9	650
var_Lex	y	2B(5)	13	798
var_MaxDens	n	3B(5)	13	889
var_MaxAbs	n	3B(5)	12	949
var_MaxDens	y	3B(5)	15	955
var_Lex	y	3B(5)	15	958
var_MaxAbs	y	3B(5)	13	1002
var_Lex	n	3B(5)	15	1039
var_MaxDens	n	2B(5)	20	1203
var_MaxAbs	y	2B(5)	20	1219
var_MaxAbs	n	2B(5)	20	1219
var_MaxDens	y	2B(5)	23	1396

(a) SAT

choix de var.		Filt.	TO	$\sum$ t
strat.	sel.			(ms)
var_Lex	n	2B(5)	4	294
var_Lex	n	3B(5)	4	393
var_MaxAbs	n	3B(5)	4	413
var_Lex	y	3B(5)	5	458
var_MaxAbs	y	2B(5)	5	472
var_Lex	y	2B(5)	13	819
var_MaxDens	n	3B(5)	17	1062
var_MaxDens	y	3B(5)	17	1063
var_MaxAbs	y	3B(5)	19	1151
var_MaxAbs	n	2B(5)	24	1468
var_MaxDens	n	2B(5)	26	1561
var_MaxDens	y	2B(5)	31	1898

(b) UNSAT

TABLE 6.1 – Comparaison des stratégies `var_MaxAbs`, `var_MaxDens` et `var_Lex` avec un choix de sous-domaines `bissection`

le contre-exemple. De plus, lorsqu'il n'y a plus d'absorption, cette stratégie choisit les variables en utilisant un ordre lexicographique sans exploiter la structure du problème. Enfin, les domaines des variables impliquées dans une absorption sont en général grands et seule une petite partie de ces domaines est effectivement impliquée dans une absorption.

La stratégie `var_MaxDens` n'est pas efficace sur les benchmarks sans solution, car elle favorise les variables avec de fortes densités. Une variable avec une densité forte a potentiellement un nombre de solutions plus important, ce qui est contre-productif lorsque les benchmarks n'ont pas de solution.

Notons également que la 3B-consistance est globalement meilleure que la 2B-consistance pour résoudre cet ensemble de benchmarks avec et sans solutions.

6.2 Stratégies de choix de variables : extensions

Dans cette section, nous présentons deux d'améliorations pour les stratégies basées sur l'absorption. Ces améliorations nous ont conduits à définir trois nouvelles stratégies `var_MaxAbs`*, `var_MaxAbs_Dens` et `var_MaxAbs_Dens`*.

6.2.1 `var_MaxAbs`* ou la prise en compte des absorptions multiples

La stratégie sur l'absorption que nous avons définie, ne détecte pas les phénomènes d'absorption multiples sur une même expression. Par exemple, dans la formule 6.1 une première absorption peut avoir lieu avec $x+y$. Puis une seconde lors de l'addition avec z. Techniquement, cette formule est réécrite en un ensemble de contraintes élémentaires (formules 6.2 et 6.3). Puis les

stratégies calculent les taux d'absorption des variables initiales du modèle. Or α n'est pas une variable initiale.

$$t = (x + y) + z \tag{6.1}$$

$$\alpha = x + y \tag{6.2}$$
$$t = \alpha + z \tag{6.3}$$

Les variables α qui peuvent mener à une absorption sont également considérées lors de la recherche.

Notons que le cas où z absorbe $(x + y)$ est détecté. La version 1 de la stratégie se concentre sur les variables qui absorbent, plutôt que sur les variables absorbées. L'amélioration proposée, reste dans la même optique, mais considère les variables α pouvant mener à une absorption. Cette amélioration détecte également les absorptions sur des expressions complexes comme $(x * y) + (w * z)$.

Dans la suite, nous illustrons l'intérêt de cette amélioration sur un petit exemple simple. Puis nous comparons les temps de résolutions des stratégies `var_MaxAbs` et `var_MaxAbs`*.

6.2.1.1 var_MaxAbs$*$: exemple

Le CSP $\mathcal{A}$ ci-dessous est un exemple simple contenant une double absorption. Dans toutes les solutions de ce CSP, une absorption existe et est déterminante pour résoudre ce problème.

CSP $\mathcal{A}$:
$x \in [1e3, 1e11]$, $y \in [1.0, 40000.0]$, $z \in [1.0, 40000.0]$, $t \in [-\infty, +\infty]$
$w \in [-5000.0, 40000.0]$, $a \in [-5000.0, 40000.0]$
$t = x + y + z$
$t == x$
$w \geq a + y$

La stratégie `var_MaxAbs` avec un filtrage 2B(5) et une resélection autoriser génère 786432 points de choix, alors que la stratégie `var_MaxAbs`* avec les mêmes conditions n'en génère que 115.

6.2.1.2 Résultats expérimentaux

La Table 6.2 présente les résultats de la stratégie `var_MaxAbs`*, et les compare à la stratégie `var_MaxAbs`.

choix de var.		Filt.	TO	$\sum$ t
strat.	sel.			(s)
var_MaxAbs*	n	2B(5)	10	622
var_Lex	n	2B(5)	9	650
var_MaxAbs*	y	3B(5)	9	765
var_MaxAbs*	y	2B(5)	13	798
var_MaxAbs*	n	3B(5)	12	872
var_MaxAbs	n	3B(5)	12	949

(a) SAT

choix de var.		Filt.	TO	$\sum$ t
strat.	sel.			(s)
var_Lex	n	2B(5)	4	294
var_MaxAbs*	n	3B(5)	4	407
var_MaxAbs	n	3B(5)	4	413
var_MaxAbs*	y	3B(5)	5	466
var_MaxAbs*	n	2B(5)	10	652
var_MaxAbs*	y	2B(5)	13	828

(b) UNSAT

TABLE 6.2 – Comparaison des stratégies `var_MaxAbs` et `var_MaxAbs`* avec un choix de sous-domaines `bissection`

Sur l'ensemble des benchmarks avec solutions, la stratégie `var_MaxAbs`* permet un gain de 30% sur la stratégie `var_MaxAbs` avec le meilleur temps de résolution. Cette stratégie permet également un petit gain sur la meilleure stratégie `var_Lex`. Sur l'ensemble des benchmarks sans solution, les temps des stratégies `var_MaxAbs`* et `var_MaxAbs` sont comparables.

6.2.2 Combinaison entre `var_MaxAbs` et `var_MaxDens`

Lorsqu'il n'y plus d'absorption, le choix de variables basé sur l'absorption n'a plus d'intérêt. Nous comparons ci-dessous les combinaisons suivantes :

1. La stratégie de choix de variables `var_MaxAbs` avec le meilleur temps de résolution.
2. La stratégie de choix de variables `var_MaxAbs` qui laisse place à la stratégie `var_MaxDens` lorsqu'il n'y a plus d'absorption. Cette stratégie est notée `var_MaxAbs_Dens`.
3. La stratégie de choix de variables `var_MaxAbs`* avec le meilleur temps de résolution
4. La stratégie de choix de variables `var_MaxAbs`* qui laisse place à la stratégie `var_MaxDens` lorsqu'il n'y a plus d'absorption. Cette stratégie est notée `var_MaxAbs_Dens`*.
5. La stratégie `var_Lex` avec le meilleur temps de résolution.

Les stratégies combinant absorption et densité : `var_MaxAbs_Dens` et `var_MaxAbs_Dens`*, ne permettent pas de résoudre plus rapidement cet ensemble de benchmarks que la meilleure stratégie `var_MaxAbs`*. Les stratégies `var_MaxAbs_Dens` perdent 20% par rapport à la meilleure stratégie basée sur l'absorption sur les benchmarks avec solutions. Pour les benchmarks sans solution, les stratégies `var_MaxAbs_Dens`, `var_MaxAbs_Dens`* et `var_MaxAbs`* sont comparables sur les benchmarks sans solution. Néanmoins, les stratégies

choix de var.		Filt.	TO	$\sum$ t (ms)
strat.	sel.			
var_MaxAbs*	n	2B(5)	10	622
var_Lex	n	2B(5)	9	650
var_MaxAbs_Dens	y	3B(5)	10	837
var_MaxAbs_Dens	n	3B(5)	12	870
var_MaxAbs	n	3B(5)	12	949
var_MaxAbs_Dens*	n	3B(5)	12	951
var_MaxAbs_Dens	n	2B(5)	16	977
var_MaxAbs_Dens*	y	3B(5)	13	1006
var_MaxAbs_Dens*	y	2B(5)	20	1220
var_MaxAbs_Dens*	n	2B(5)	20	1220
var_MaxAbs_Dens	y	2B(5)	24	1456

(a) SAT

choix de var.		Filt.	TO	$\sum$ t (ms)
strat.	sel.			
var_Lex	n	2B(5)	4	294
var_MaxAbs*	n	3B(5)	4	407
var_MaxAbs_Dens	n	3B(5)	4	408
var_MaxAbs_Dens*	n	3B(5)	4	412
var_MaxAbs	n	3B(5)	4	413
var_MaxAbs_Dens	y	3B(5)	5	468
var_MaxAbs_Dens*	y	3B(5)	5	472
var_MaxAbs_Dens	n	2B(5)	10	653
var_MaxAbs_Dens	y	2B(5)	13	828
var_MaxAbs_Dens*	n	2B(5)	19	1227
var_MaxAbs_Dens*	y	2B(5)	24	1468

(b) UNSAT

TABLE 6.3 – Comparaison des stratégies `var_MaxAbs`*, `var_MaxAbs_Dens` et `var_MaxAbs_Dens`* avec une `bissection`

combinant absorption et densité améliorent et donnent de meilleur temps de résolution que les stratégies mono-propriété `var_MaxAbs` et `var_MaxDens`.

Les résultats de la stratégie `var_MaxAbs_Dens`* sur les benchmarks avec solutions s'expliquent par le fait que de nombreuses variables α peuvent être ajoutées aux variables à considérer. Ces variables ne sont pas des variables initiales du modèle. Les sous-domaines générés sur ces variables ont en général un impact moins important sur la totalité des variables. Sur les problème sans solution, choisir les variables qui maximisent la densité est souvent contre-productif.

6.3 Autres stratégies

Dans cette section, nous présentons d'autres stratégies non évaluées dans le cadre de cette thèse, mais s'inscrivant dans les perspectives de ces travaux.

6.3.1 Stratégies multi-propriétés :

Nous proposons de composer deux propriétés spécifiques aux flottants : la densité et l'absorption. L'intuition de ces stratégies est de discriminer les candidats potentiels en utilisant une autre propriété. Par exemple, considérons deux stratégies : `absWDens` et `densWAbs`. Ces stratégies sont basées sur deux propriétés.

`absWDens` sélectionne la variable qui maximise la densité à partir du sous-ensemble de variables impliquées dans une absorption ($absorb > 0$).

`densWAbs` sélectionne la variable qui maximise l'absorption parmi le sous-ensemble de variables qui satisfait $density \geq \frac{maxDens+minDens}{2}$.

D'autres stratégies basées sur la combinaison de propriétés sont imaginables.

6.3.2 Stratégies avec score

Les stratégies avec score ont pour objectif de mettre en relation de plusieurs propriétés en utilisant une formule. Cette proposition est inspirée de la stratégie de recherches IBS [Refalo, 2004] (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 dû 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 reste à définir après 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.

6.4 Conclusion

Ce chapitre introduit un ensemble de propriétés permettant de choisir une variable pour guider la recherche lors de la résolution d'un système de contraintes sur les nombres à virgule flottante. Ces propriétés capturent des

spécificités des domaines et des contraintes flottantes dans un CSP sur les flottants. Ces propriétés sont la base d'heuristique de choix de variables. Ces heuristiques maximisent ou minimisent ces dernières. Ces heuristiques ont pour objectif de tirer profit des spécificités des nombres flottants et de l'arithmétique qui leur est propre pour guider la recherche lors de la résolution.

Chapitre 7

Stratégies de sélection de sous-domaines

Dans le chapitre précédent, nous avons introduit un ensemble de stratégies de choix de variables basées sur les propriétés spécifiques des flottants comme la densité du domaine, la cancellation ou les phénomènes d'absorption. Les stratégies de recherche qui en résultent sont plus efficaces et indépendantes de l'ordre de déclaration des variables, mais ne permettent pas vraiment de passer à l'échelle pour des benchmarks plus difficiles et plus réalistes. En effet, comme dans d'autres applications de techniques de contraintes, les solveurs efficaces nécessitent non seulement des stratégies de choix de variables appropriées, mais aussi des stratégies de choix de valeurs pertinentes. Ainsi, ce chapitre se concentre sur les stratégies de choix de valeurs pour les solveurs de contraintes sur les nombres à virgule flottante dédiés à la recherche de contre-exemples dans les applications de vérification de programmes.

Les problèmes sur les nombres à virgule flottante sont caractérisés par des domaines très grands et des valeurs non uniformément distribuées. Par conséquent, des stratégies d'énumération comme celle qui sont utilisées sur les domaines finis n'ont aucune chance de trouver une solution.

Les stratégies de choix des valeurs standard sur les flottants sont dérivées des stratégies de choix des sous-domaines utilisées sur les réels ; les sous-domaines sont générés en utilisant diverses techniques de splitting, par exemple, $x_{\mathbb{F}} \leq v_{\mathbb{F}}$ ou $x_{\mathbb{F}} > v_{\mathbb{F}}$ avec $v_{\mathbb{F}} = \frac{\overline{x}_{\mathbb{F}} + \underline{x}_{\mathbb{F}}}{2}$.

Dans ce chapitre, nous présentons de nouvelles stratégies de choix de valeurs (i.e sous-domaines) et leurs évaluations. Les premières sont des stratégies simples introduites dans [Zitoun et al., 2017] qui sont dérivées de la stratégie *fpc* [Collavizza et al., 2016] (présentée dans le chapitre 3). Puis, nous

présentons une stratégie plus spécifique qui exploite les phénomènes d'absorption. Enfin, la dernière stratégie introduite embrasse les idées dérivées des consistances fortes.

7.1 Stratégies mixant énumération de valeurs et décomposition en sous-domaines

Dans cette section, nous présentons un ensemble de stratégies de choix de sous-domaines simples, mais adaptées aux domaines sur les flottants. Ces stratégies généralisent la stratégie fpc introduite dans [Collavizza et al., 2016].

7.1.1 Stratégies de base

La stratégie de choix de sous-domaine de base dans les domaines continues et sur les flottants est la bissection. En présence d'un grand nombre de solutions, une simple bissection (Figure 7.1c) atteint rapidement ses limites, le filtrage appliqué après chaque bissection n'étant souvent pas en mesure de réduire la taille des domaines. Pour surmonter ces difficultés, [Collavizza et al., 2016] ont introduit une stratégie de choix de sous-domaines `fpc` qui mélangent l'énumération de valeurs à la bissection classique. Cette stratégie commence toujours par l'énumération des valeurs sélectionnées avant de traiter les deux sous-domaines restants. La figure 7.1b présente cette stratégie.

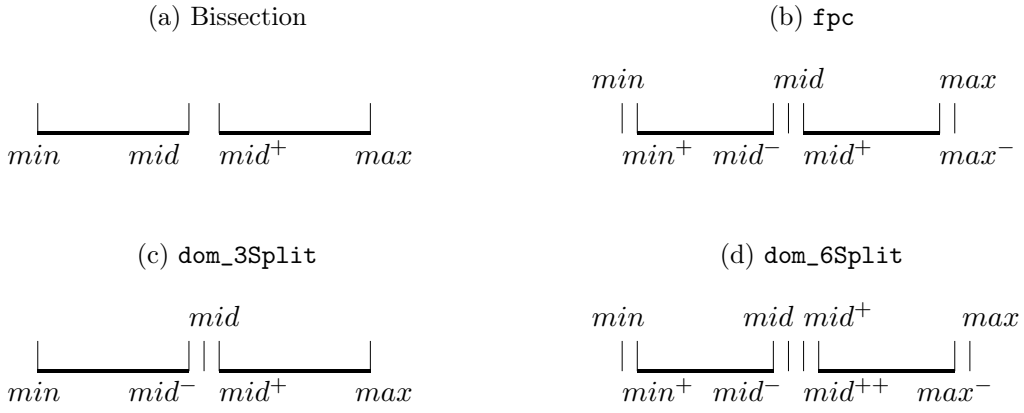


FIGURE 7.1 – Stratégies de choix de sous-domaines

Nous avons d'abord proposé deux variations de cette stratégie :
- ◇ `dom_3Split` qui ne fait qu'énumérer le milieu
- ◇ `dom_6Split` qui énumère trois valeurs : les bornes et le milieu

Les Figures 7.1c à 7.1d illustrent ces variations.

Nous proposons deux généralisations de ces stratégies. L'intuition de ces stratégies est d'explorer les bords, puis le milieu et enfin les sous-domaines restants. La première généralisation `Enum-N`, énumère les `N` flottants de chaque borne, puis le flottant du milieu du domaine, avant dans considérer le reste du domaine. Le second, `Delta-N`, plutôt que d'énumérer les `N` flottants des bornes, construit les sous-domaines impliqués par ces `N` valeurs.

La stratégie `dom_Enum-N`

Cette stratégie de choix de sous-domaines est une généralisation directe des stratégies présentées dans la sous-section précédente. Généralement, le processus de filtrage resserre les bornes jusqu'à ce qu'un support soit trouvé. Cette stratégie de choix de sous-domaines est optimiste et espère que le filtrage conduira à trouver une solution proche des bornes. Pour atteindre cet objectif, elle énumère quelques valeurs aux bornes. La figure 7.2 illustre cette stratégie. Le domaine est divisé en $2 * N + 3$ sous-domaines. Par exemple, avec $N = 5$, les 13 domaines suivants seront générés :

- $\diamond$ $[min, min]$
- $\diamond$ $\cdots$
- $\diamond$ $[min^{+(N)}, min^{+(N)}]$
- $\diamond$ $[min^{+(N+1}, mid^{-}]$
- $\diamond$ $[mid, mid]$
- $\diamond$ $[mid^{+}, max^{-(N+1)}]$
- $\diamond$ $[max^{-(N)}, max^{-(N)}]$
- $\diamond$ $\cdots$
- $\diamond$ $[max, max]$

Si la cardinalité du domaine est inférieure à $2 * N + 3$, toutes les valeurs du domaine sont énumérées.

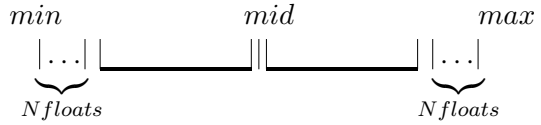


FIGURE 7.2 – Illustration de la strategie `dom_Enum-N`

La stratégie `dom_Delta-N`

Le premier objectif de cette stratégie est de trouver une solution aux bornes. L'énumération peut mener la recherche à explorer un sous-arbre profond avant de trouver une solution ou d'effectuer une réduction. Ici, considérer le sous-domaine impliqué par les N flottants, au lieu d'un seul, donne l'opportunité au processus de filtrage d'opérer certaines réductions par propagation. Il améliore également la possibilité de trouver une solution ou de refuter par simple filtrage l'ensemble des N flottants contenus dans ces petits sous-domaines sans les énumérer. Cette stratégie de choix de sous-domaines est très flexible. En effet, en adaptant la valeur de N, l'ensemble de la stratégie change de comportement. Par exemple, si $N = |\alpha|$, avec $\alpha = [\underline{x}_{\mathbb{F}}, \frac{\underline{x}_{\mathbb{F}} + \overline{x}_{\mathbb{F}}}{2} 2]$, cette stratégie de sous-domaine devient une bissection classique. Enfin, une modification dynamique de la valeur de N pendant la recherche peut être intéressante. Commencer la recherche par une bissection classique ($N = |\alpha|$) et réduire sa valeur pourrait être une bonne idée. La figure 7.3 illustre cette stratégie. Indépendamment de la valeur de N et de la cardinalité du domaine, au maximum les 5 sous-domaines suivants seront générés :

- $[min, min^{+(N)}]$
- $[min^{+(N+1)}, mid^{-}]$
- $[mid, mid]$
- $[mid^{+}, max^{-(N+1)}]$
- $[max^{-(N)}, max]$

Si la cardinalité du domaine est inférieure à $2 * N + 3$, les deux derniers sous-domaines ne sont pas générés, et les deux premiers sont équilibrés par rapport au milieu du domaine. Lorsqu'il y a moins de deux flottants, ils sont énumérés.

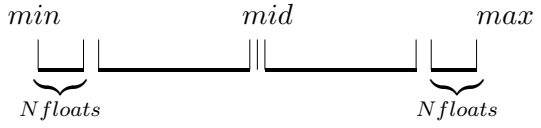


FIGURE 7.3 – Illustration de la strategie `dom_Delta-N`

7.1.2 Expérimentations

Nos expérimentations ont montré que la bissection était bien meilleure pour les problèmes sans solution. L'intuition est qu'énumérer sur un problème sans solution est une perte de temps pour supprimer une seule valeur, alors que le filtrage finira obligatoirement par la supprimer. Sur les problèmes avec solutions, énumérer permet en général de trouver plus rapidement une solution.

7.2 Stratégie de choix de sous-domaines basée sur l'absorption : dom_splitAbs

Dans cette section, nous présentons la stratégie de choix de sous-domaine basée sur l'absorption `dom_splitAbs`.

Rappelons que l'absorption se produit lorsque l'on ajoute deux nombres à virgule flottante avec un ordre de grandeur différent. Le résultat d'une telle addition est le nombre de plus grande magnitude.

Les choix de variables suffissent rarement pour exploiter pleinement l'absorption. Généralement, les domaines impliqués dans une absorption sont grands, et seulement une petite part de ces domaines est effectivement impliquée dans l'absorption en question. L'idée de cette stratégie est de cibler les sous-domaines en question.

7.2.1 Adaptation de var_MaxAbs

Pour tirer pleinement profit de l'absorption, la stratégie `var_MaxAbs` doit non seulement être combinée avec une stratégie de choix de sous-domaines basée sur l'absorption, mais elle doit aussi être modifiée : il faut choisir un couple de variables $< x_{\mathbb{F}}, y_{\mathbb{F}} >$. La variable $x_{\mathbb{F}}$ est la variable qui absorbe le plus. Après avoir choisi la variable $x_{\mathbb{F}}$, la stratégie `var_MaxAbs` examine les contraintes $c_i(y_{\mathbb{F}}, x_{\mathbb{F}})$ où la variable $x_{\mathbb{F}}$ est impliquée et sélectionne la variable $y_{\mathbb{F}}$ parmi les variables non instanciées X^*, dont les valeurs sont le plus absorbées par les valeurs de $\mathbf{x}_{\mathbb{F}}$. Le branchement coordonné sur ces variables est effectué pour exploiter l'absorption latente. Plus formellement, la variable $y_{\mathbb{F}} \in X^*$ est choisie selon le critère suivant :

$$absorb(y_{\mathbb{F}}, x_{\mathbb{F}}, c_i) = max_{y'_{\mathbb{F}} \in X^*}\{absorb(y'_{\mathbb{F}}, x_{\mathbb{F}}, c'_i) | c'_i \in Contraintes(x_{\mathbb{F}})\}$$

La formule 7 à la page 84 explicite le calcul de $absorb(y_{\mathbb{F}}, x_{\mathbb{F}}, c_i)$.

7.2.2 Stratégie de choix de sous-domaines : `dom_splitAbs`

Cette stratégie génère au plus trois sous-domaines pour chaque variable du couple $< x_{\mathbb{F}}, y_{\mathbb{F}} >$. Les sous-domaines de $x_{\mathbb{F}}$ générés dans le cas où $abs(\underline{x}_{\mathbb{F}}) < abs(\overline{x}_{\mathbb{F}})$ avec abs étant la valeur absolue sont :

1. $[2^{e_{\overline{x}_{\mathbb{F}}}}, \overline{x}_{\mathbb{F}}]$
2. $[\underline{x}_{\mathbb{F}}, 2^{e_{\overline{x}_{\mathbb{F}}}-}]$

Les sous-domaines sont considérés dans cet ordre. Le premier capture les valeurs de $x_{\mathbb{F}}$ les plus absorbantes, e.g., les valeurs de $x_{\mathbb{F}}$ absorbants le plus de valeurs de $y_{\mathbb{F}}$. Lorsque $abs(\underline{x}_{\mathbb{F}}) > abs(\overline{x}_{\mathbb{F}})$, les sous-domaines générés s'obtiennent facilement par symétrie. Si $\overline{x}_{\mathbb{F}}$ est égale à $\underline{x}_{\mathbb{F}}$, les deux sous-domaines positif et négatif absorbant les valeurs de $y_{\mathbb{F}}$ sont générés, ainsi que le sous-domaine n'absorbant aucune valeur de $y_{\mathbb{F}}$.
De la même manière, les sous-domaines générés lorsque $abs(\underline{y}_{\mathbb{F}}) \leq abs(\overline{y}_{\mathbb{F}})$ sont :

1. $y \cap [-2^{e_{\overline{y}}-p-1}, 2^{e_{\overline{y}}-p-1}]$
2. $y \cap [(2^{e_{\overline{y}}-p-1})^+, \overline{x}]$
3. $y \cap [\underline{x}_{\mathbb{F}}, (2^{e_{\overline{y}}-p-1})^-]$

Les sous-domaines générés lorsque $abs(\underline{y}_{\mathbb{F}}) > abs(\overline{y}_{\mathbb{F}})$ s'obtiennent en remplaçant $\overline{y}_{\mathbb{F}}$ par $\underline{y}_{\mathbb{F}}$. La Figure 7.4 illustre les sous-domaines générés dans le cas où les domaines initiaux sont positifs.

FIGURE 7.4 – Sous-domaines génerés par *dom_SplitAbs* ($\underline{y} > 0$ et $\underline{y} > 0$)

Exemple de programme avec une absorption Considerons le programme suivant :

```
void foo(float x, float y){
   float z = x + 2 * y;
   if(z != x)
     systemOK();
   else
     systemNOK();
}
```

Supposons également que les entrées proviennent de capteurs, et que leurs domaines sont $[0.0, 1e{+}04]$ pour $\mathbf{x}$, et $[-16.0, 4.0]$ pour $\mathbf{y}$. La branche `else` correspond à un état instable du système. Déterminer si cet état est accessible et à partir de quelles valeurs d'entrée est une question légitime.

Ce problème se réduit à identifier si la variable $z_{\mathbb{F}}$ peut être égale à $x_{\mathbb{F}}$, ce qui correspond à une absorption (sauf lorsque $y = 0$). La figure ci-dessous montre les sous-domaines résultants. Cette stratégie met l'accent sur le sous-domaine $[8.19200098e{+}03, 1.00000000e{+}04]$ pour $\mathbf{x}$ et le sous-domaine $[-4.88281221e{-}04, 4.88281220e{-}04]$ pour $\mathbf{y}$, sous domaines de x et y pour lesquels toute valeur de x absorbe toute valeur de y. Pour la variable $y_{\mathbb{F}}$, il n'existe aucune solution impliquant une valeur appartenant à un autre sous-domaine. Pour la variable $x_{\mathbb{F}}$, des valeurs de magnitude plus faible que celles de ce sous-domaine, peuvent aussi menée à une absorption, mais elle implique un sous-domaine de $y_{\mathbb{F}}$ plus petit. Une parties de ces valeurs seront considérées une fois le sous-domaine $[8192, 10000]$ complétement réfuté.

Le filtrage initial n'a pas d'impact sur ces domaines. L'état instable est accessible avec, par exemple, les valeurs $1e{+}04$ pour la variable $x_{\mathbb{F}}$ et $2,44140625e{-}04$ pour $y_{\mathbb{F}}$.

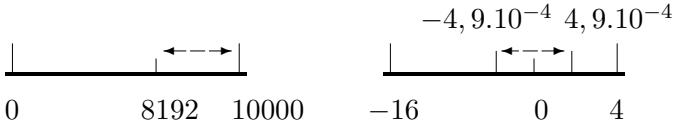


FIGURE 7.5 – Sous-domaines générées par `dom_SplitAbs`

Cette stratégie peut également être combinée avec une autre stratégie, qui est appelée chaque fois que la variable $x_{\mathbb{F}}$ n'a pas de valeur qui absorbe $y_{\mathbb{F}}$.

7.2.2.1 Absorptions multiples

Dans cette section, nous utilisons à nouveau le CSP $\mathcal{A}$ présentée dans le chapitre précédent à la page 98. Cet exemple simple contient une double absorption. Toutes les solutions de ce CSP contiennent une absorption déterminante pour la résolution de ce problème.

CSP $\mathcal{A}$:
$x \in [1e3, 1e11]$, $y \in [1.0, 40000.0]$, $z \in [1.0, 40000.0]$, $t \in [-\infty, +\infty]$
$w \in [-5000.0, 40000.0]$, $a \in [-5000.0, 40000.0]$
$t = x + y + z$
$t == x$
$w \geq a + y$

Pour cet exemple, la stratégie de choix de variable `var_MaxAbs*` combinée avec la stratégie `dom_splitAbs` permet de résoudre cet exemple en 8 points de choix, alors qu'une bissection nécessite 115 points de choix. La stratégie `dom_splitAbs` est également plus rapide d'un ordre de grandeur.

7.3 Stratégie de choix de sous-domaines : `dom_3BSplit`

La prochaine stratégie de choix de sous-domaines, appelée `dom_3BSplit`, est inspirée par une consistance plus forte nommée 3B-consistance [Lhomme, 1993]. La 3B-Consistance est une relaxation sur de la consistance de chemin sur les domaines continus, une extension d'ordre supérieur à la consistance d'arc. Pour résumer, la 3B-consistance vérifie si le filtrage par 2B-consistance du système de contraintes ne génère pas système inconsistant (i.e., qui contient au moins un domaine vide) lorsque le domaine d'une des variables du CSP est réduit à la valeur d'une de ses bornes [Collavizza et al., 1999]. La 3B-consistance est en pratique très efficace sur des problèmes avec des occurrences multiples de variables. Cependant, assurer une telle consistance peut être coûteux : l'algorithme 3B peut tenter plusieurs fois de réfuter sans succès des sous-domaines aux bornes du domaine initial au moyen d'une 2B-consistance.

La stratégie de choix de sous-domaines `dom_3BSplit` tente également de renforcer la consistance aux bornes du domaine sur une seule variable. Une observation clé est que si un petit sous-domaine aux bornes du domaine initial, par exemple, $sd = [\underline{x}_{\mathbb{F}}, \underline{x}_{\mathbb{F}} + \delta]$, est immédiatement réfuté dans le noeud de recherche suivant, alors le domaine résultant, $[\underline{x}_{\mathbb{F}} + \delta, \overline{x}_{\mathbb{F}}]$, offre une meilleure borne inférieure que celle du domaine initial. De plus, il est probable qu'il soit encore possible d'améliorer cette borne si la même procédure est réitérée en utilisant un sous-domaine plus large comme $[\underline{x}_{\mathbb{F}}, \underline{x}_{\mathbb{F}} + 2\delta]$ (où $\underline{x}_{\mathbb{F}}$ est la nouvelle borne inférieure de $x_{\mathbb{F}}$). En effet, un tel processus est similaire à un *shaving* appliqué au domaine initial mais sans nécessiter une gestion supplémentaire de l'état du domaine : dans le cas d'un `dom_3BSplit`, la possibilité de revenir à l'état initial est naturellement mise à disposition par l'algorithme de recherche.

Ce sous-domaine peut ne pas être réfuté immédiatement dans le nœud de recherche suivant. Il peut être réfuté soit après avoir exploré un sous-arbre de recherche profond, soit il peut fournir une ou plusieurs solutions. Dans les deux cas, il est difficile d'améliorer la borne. L'étape suivante de la stratégie passe donc à la borne suivante ou, si les deux bornes ont été vérifiées, à un autre nœud de recherche ou à une autre stratégie. La Figure7.6 illustre le comportement de `dom_3BSplit`.

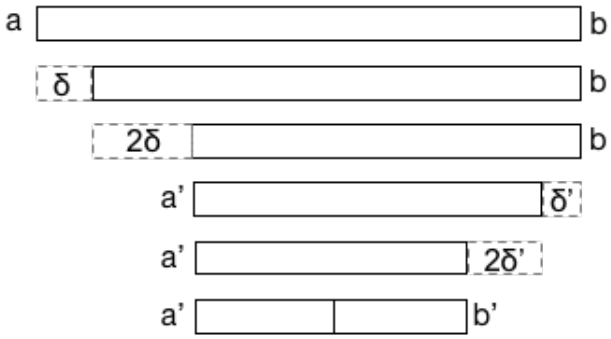


FIGURE 7.6 – Stratégie de choix de sous-domaines `dom_3BSplit`

En résumé, `dom_3BSplit` exploite l'information sur le sous-arbre pour décider si renforcer la borne courante a une chance de se faire sans effort ou s'il serait plus sage de passer à l'étape suivante. En conséquence, cette stratégie divise dynamiquement le domaine courant en fonction du comportement de la recherche dans les sous-arbres. Notez également que le choix d'un petit sous-domaine initial aux bornes du domaine est similaire à la stratégie `dom_delta-N` (section 7.1.1), c'est-à-dire que 3Bsplit offre également des possibilités de trouver des solutions dans le voisinage des bornes courantes.

7.4 Expérimentations

Les expérimentations ont été effectuées dans les mêmes conditions et sur les mêmes benchmarks que pour celles sur les stratégies de choix de variables.

7.4.1 Benchmarks avec solutions

La Table 7.1 présente les résultats des différentes stratégies de choix de sous-domaines présentées dans ce chapitre sur l'ensemble de benchmarks avec solutions.

Les stratégies considérant quelques flottants sur les bords des domaines (e.g., `dom_Delta-2`) montrent de bien meilleurs résultats qu'une `bissection` classique sur les benchmarks avec solutions sur la stratégie de choix de variables `var_MaxAbs`. Ces résultats sont en général encore meilleurs lorsqu'ils sont combinés avec un filtrage fort comme une 3B-consistance. L'intuition est qu'un filtrage fort permet d'avoir des solutions locales aux bornes des intervalles, énumérer quelques-unes des valeurs sur ces bornes permet donc

choix de var.		Filt.	TO	$\sum$ t
strat.	sel.			(s)
dom_Delta-2	n	3B(5)	6	340
dom_Enum-2	n	3B(5)	6	391
dom_Enum-1	n	3B(5)	6	392
dom_Enum-1	y	3B(5)	7	456
dom_Enum-10	n	3B(5)	7	457
dom_6split	n	3B(5)	7	457
dom_Delta-10	n	3B(5)	7	468
dom_3Bsplit	n	2B(5)	7	472
dom_Delta-10	y	3B(5)	7	479
dom_6split	y	3B(5)	7	498
dom_Enum-2	y	3B(5)	8	515
dom_Delta-2	n	2B(5)	9	558
dom_Enum-10	y	3B(5)	8	562
dom_6split	n	2B(5)	9	568
dom_Delta-10	n	2B(5)	11	690
dom_SplitAbs	y	2B(5)	11	690
dom_3Bsplit	y	3B(5)	9	715
dom_Enum-10	n	2B(5)	12	736
dom_Enum-1	n	2B(5)	12	739
dom_Enum-2	n	2B(5)	12	744
dom_Delta-10	y	2B(5)	13	788
dom_Enum-10	y	2B(5)	13	791
dom_3Bsplit	y	2B(5)	13	800
dom_3Bsplit	n	3B(5)	10	805
dom_SplitAbs	y	3B(5)	9	806
dom_Delta-2	y	3B(5)	6	810
dom_Delta-2	y	2B(5)	13	823
dom_6split	y	2B(5)	13	832
dom_Enum-1	y	2B(5)	14	870
dom_Enum-2	y	2B(5)	14	874
dom_SplitAbs	n	2B(5)	15	922
bissection	n	3B(5)	12	949
dom_SplitAbs	n	3B(5)	20	1328

TABLE 7.1 – Benchmarks SAT avec la stratégie de choix de variable `var_MaxAbs`

de trouver plus rapidement une solution ou donne de nouvelles opportunités aux filtrages de réduire de nouveau les domaines.

La stratégie `dom_SplitAbs` a des résultats assez moyens sur ces benchmarks, ceci s'explique par le fait que tous les benchmarks ne conduisent pas à des solutions avec des absorptions. Chercher à exploiter ces absorptions lorsque aucunes solutions n'en contient entraine une perte de temps inutile. Enfin, la stratégie `dom_3Bsplit` montre des résultats corrects.

7.4.2 Benchmarks avec solutions et absorption

La Table 7.2 présente les résultats des différentes stratégies de choix de sous-domaines sur l'ensemble des benchmarks avec solutions restreints aux benchmarks pouvant conduire à une solution avec une absorption.

choix de var.		Filt.	TO	$\sum$ t
strat.	sel.			(s)
dom_SplitAbs	y	2B(5)	3	198
dom_3Bsplit	y	2B(5)	3	199
dom_Enum-2	n	3B(5)	3	208
dom_Delta-2	n	3B(5)	3	209
dom_Enum-1	n	3B(5)	3	209
dom_6Split	n	3B(5)	3	217
dom_Delta-10	n	3B(5)	3	219
dom_SplitAbs	y	2B(5)	4	259
dom_Enum-10	n	3B(5)	4	268
dom_Delta-2	n	3B(5)	4	277
dom_Delta-10	n	3B(5)	4	290
dom_6Split	n	3B(5)	4	312
dom_Enum-2	y	2B(5)	5	328
dom_Enum-10	n	3B(5)	5	328
dom_Enum-1	n	3B(5)	5	329
dom_3Bsplit	n	3B(5)	4	393
dom_SplitAbs	n	3B(5)	4	422
dom_Delta-2	y	2B(5)	7	436
bissection	n	3B(5)	4	442
dom_6Split	y	2B(5)	7	447
dom_Delta-10	y	2B(5)	7	450
dom_3Bsplit	n	3B(5)	6	526
dom_Enum-1	y	2B(5)	9	558
dom_Enum-10	y	2B(5)	10	606
dom_Enum-2	n	3B(5)	8	615
dom_3Bsplit	y	2B(5)	10	618
dom_Enum-2	y	2B(5)	10	623
dom_Delta-2	y	2B(5)	10	630
dom_Enum-1	y	2B(5)	10	630
dom_Enum-10	y	2B(5)	11	666
dom_Delta-10	y	2B(5)	11	667
dom_SplitAbs	n	3B(5)	10	676
dom_6Split	y	2B(5)	11	711

TABLE 7.2 – Benchmarks SAT avec absorption pour la stratégie de choix de variable `var_MaxAbs`

Dans les benchmarks où des solutions avec une absorption existent, la stratégie dédiée `dom_SplitAbs` a des résultats clairement meilleurs. La stratégie `dom_3Bsplit` a également de bon résultat sur ces benchmarks. Cette stratégie cherche des solutions dans un petit morceau aux bords des domaines. Or les variables qui absorbent ont forcément des supports sur ces bords. Si une absorption est effective, il y a des chances que le filtrage réduise le domaine de la variable absorbée au voisinage de zéro. Dans ces benchmarks, toutes les variables ne sont pas impliquées dans une absorption : renforcer la consistance aux bornes de variables non impliquées dans une absorption permet d'éviter certains échecs auxquels conduit la stratégie `dom_SplitAbs`. Contrairement aux stratégies `dom_Enum-N` ou `dom_Delta-N`, cette stratégie est aussi plus raisonnable dans le sens où elle considère un intervalle ni trop grand ni constitué seulement de quelques flottants. Cela permet potentiellement d'avoir plus de chance de tomber sur une solution après les réductions opérées par le filtrage. La stratégie `dom_3Bsplit` semble être un bon compromis pour traiter les benchmarks avec quelques absorptions.

7.4.3 Benchmarks sans solution

La Table 7.3 présente les résultats des différentes stratégies de choix de sous-domaines sur l'ensemble des benchmarks sans solutions.

choix de var.		Filt.	TO	$\sum$ t
strat.	sel.			(s)
dom_SplitAbs	n	2B(5)	4	326
bissection	n	3B(5)	4	413
dom_SplitAbs	n	3B(5)	4	466
dom_Delta-10	n	3B(5)	5	473
dom_Delta-2	n	3B(5)	5	474
dom_6split	n	3B(5)	5	480
dom_6split	y	3B(5)	5	489
dom_SplitAbs	y	3B(5)	5	526
dom_SplitAbs	y	2B(5)	10	659
dom_Delta-2	y	3B(5)	9	681
dom_3Bsplit	y	3B(5)	9	695
dom_3Bsplit	n	3B(5)	9	697
dom_Delta-2	n	2B(5)	11	762
dom_Delta-10	n	2B(5)	11	765
dom_6split	n	2B(5)	11	782
dom_Enum-10	y	3B(5)	14	899
dom_6split	y	2B(5)	13	912
dom_Delta-10	y	3B(5)	14	925
dom_Enum-10	n	3B(5)	13	926
dom_Enum-1	n	3B(5)	13	937
dom_Enum-1	y	3B(5)	15	972
dom_Enum-2	y	3B(5)	15	987
dom_Enum-2	n	3B(5)	15	1003
dom_3Bsplit	n	2B(5)	19	1161
dom_Delta-2	y	2B(5)	19	1184
dom_Enum-1	n	2B(5)	20	1239
dom_3Bsplit	y	2B(5)	21	1282
dom_Enum-2	n	2B(5)	21	1344
dom_Enum-2	y	2B(5)	21	1390
dom_Enum-10	n	2B(5)	23	1418
dom_Enum-10	y	2B(5)	24	1475
dom_Delta-10	y	2B(5)	24	1476
dom_Enum-1	y	2B(5)	22	1492

TABLE 7.3 – Benchmarks UNSAT avec la stratégie de choix de variable `var_MaxAbs`

La stratégie `dom_SplitAbs` avec une 2B(5)-consistance et sans resélection réussit étonnamment bien sur les benchmarks sans solution. Ces résultats sont légèrement meilleurs qu'une `bissection`. Cela s'explique peut-être par le fait que cette stratégie cible des sous-domaines où il est clair qu'il n'y à aucune solution, ce qui permettrait de supprimer une partie trivialement inconsistante de l'espace de recherche.

Les stratégies qui considèrent des sous-domaines ont des résultats logiquement meilleurs que les stratégies qui perdent du temps à énumérer. En général, un filtrage fort et donc plus précis permet de réfuter plus rapidement les domaines.

7.5 Conclusion

Le chapitre précédent proposait essentiellement un ensemble de stratégies de choix de variables utilisant les spécificités des flottants pour guider la recherche. Ces stratégies de sélection de variables améliorent la recherche d'un contre-exemple illustrant la violation de propriétés dans un programme à vérifier, mais ne sont pas suffisantes pour passer à l'échelle lorsque les benchmarks sont plus difficiles et plus réalistes. Des stratégies de choix de sous-domaines dédiés pour les flottants sont nécessaires. Les contributions présentées dans ce chapitre sont un ensemble de stratégies de choix de sous-domaines sur des flottants. La première, exploite les phénomènes d'absorption, la seconde embrasse les idées dérivées de consistance forte et les deux dernières étendent les stratégies introduites dans [Zitoun et al., 2017].
Les stratégies présentées, fonctionnent bien et obtiennent de bien meilleurs résultats que la stratégie standard utilisée pour résoudre ce genre de problème.

Chapitre 8

Conclusion

Bilan

Dans cette thèse, nous nous sommes intéressés aux stratégies de recherches dans les approches BMC basées sur la programmation par contrainte dans le cadre de la vérification de programmes sur les flottants. Ces stratégies de recherche ont pour objectif de permettre de trouver plus rapidement des contre-exemples dans les programmes à vérifier. En programmation par contraintes, les stratégies de recherche reposent à la fois sur un choix de variables et un choix de sous-domaines.

D'abord, nous avons introduit un ensemble de propriétés permettant de choisir une variable pour guider la recherche lors de la résolution d'un système de contraintes sur les nombres à virgule flottante. Ces propriétés capturent des spécificités des domaines et des contraintes sur les flottants dans un CSP. Ces propriétés sont à la base d'heuristique de choix de variables. Ces heuristiques de choix de variables que nous avons développé maximisent ou minimisent ces propriétés. Les expérimentations que nous avons effectuées sur un ensemble de 48 benchmarks avec solutions, et 47 benchmarks sans solution, ont montré que tirer profit des spécificités des nombres flottants (e.g., densité) et de l'arithmétique qui leur est propre (e.g., absorption) pour guider la recherche lors de la résolution permet de trouver des contre-exemples plus rapidement. Par exemple, la stratégie de choix de variables qui maximise l'absorption permet de résoudre la totalité des benchmarks avec solutions, 30 secondes plus vite qu'un ordre lexicographique. Ces stratégies de choix de variables améliorent la recherche d'un contre-exemple illustrant la violation de propriétés dans un programme à vérifier, mais peinent à passer à l'échelle lorsque les benchmarks sont plus difficiles et plus réalistes. Des stratégies de

choix de sous-domaines dédiés aux flottants sont nécessaires. Nous avons donc introduit un ensemble de stratégies de choix de sous-domaines sur les flottants. La première exploite les phénomènes d'absorption, la seconde embrasse les idées dérivées de consistance forte, et les dernières mixent énumération et bissection. Les évaluations que nous avons effectuées ont montré l'intérêt d'utiliser des stratégies de choix de sous-domaines dédiées pour chercher des contre-exemples dans les programmes avec du calcul sur les nombres flottants. Plus précisément, la stratégie qui cible les sous-domaines impliqués dans une absorption permet, sur l'ensemble des 17 benchmarks avec une solution contenant une absorption, de résoudre la totalité de ces benchmarks plus rapidement que les autres stratégies de recherche (e.g., un facteur 2 est gagné par rapport à une bissection). Dans le cas général, les expérimentations ont aussi mis en évidence que la stratégie basée sur des consistances fortes est un bon compromis pour résoudre les benchmarks avec ou sans absorption. Ces stratégies permettent également de prouver rapidement l'absence de solution sur les benchmarks sans solution.

Perspectives

Les stratégies de recherche que nous avons introduites ouvrent de nouvelles perspectives. Dans la suite, nous présentons l'ensemble de ces perspectives.

Amélioration de la stratégie `var_MaxAbs`*

La stratégie `var_MaxAbs`* a pour objectif de tirer profit des absorptions multiples. Pour ce faire, elle considère toutes les variables x_i qui peuvent mener à une absorption. Ces variables sont des variables supplémentaires introduites lors de la décomposition du système de contraintes initiales en contraintes élémentaires.

Une amélioration de cette stratégie consisterait à ne choisir que les sous-domaines impliqués dans l'absorption lorsqu'il s'agit des variables x_i. Dans la version actuelle, ces variables sont considérées comme des variables du modèle. Une fois le sous-domaine impliquant une absorption réfuté, le reste du domaine est donc exploré ce qui à moins d'intérêt. Fixer un seuil d'absorption minimum pour considérer ces variables peut également avoir un intérêt.

Stratégie alternative `var_MaxAbs_Alt`

La stratégie `var_MaxAbs` privilégie une variable qui absorbe globalement le plus à une variable qui absorbe le plus localement. En d'autres termes, cette stratégie peut choisir une variable qui absorbe peu sur beaucoup de contraintes, plutôt qu'une variable qui absorbe beaucoup sur une seule contrainte. Cette seconde alternative mériterait aussi d'être explorée.

Stratégie basée sur la cancellation

La stratégie basée sur la cancellation que nous avons développée utilise un critère un peu trop grossier. Par exemple, ce critère ne permet pas de différencier les cancellations bénignes de celles qui sont catastrophiques. Déterminer le type de la cancellation est un point clef. Une cancellation bénigne n'a, en général, pas d'impact direct sur la déviation du calcul flottant vis-a-vis de son interprétation sur les réels. À contrario, la cancellation catastrophique peut avoir un impact fort sur la déviation du calcul. Affiner cette propriété est donc essentiel pour avoir une stratégie qui tire au mieux profit de la cancellation.

Comme pour l'absorption, les domaines des variables impliquées dans une cancellation sont grands. Une heuristique de choix de sous-domaines dédié permettrait de se focaliser sur les sous-domaines réellement impliqués dans une cancellation. Une première piste pour cette stratégie de choix de sous-domaine est de poser des contraintes sur les erreurs des opérandes et sur l'erreur du résultat. Les opérandes doivent avoir une erreur non nulle pour que la cancellation soit catastrophique. Plus l'erreur du résultat est importante, plus la cancellation est catastrophique, d'où l'intérêt de contraindre aussi l'erreur du résultat.

Stratégie basée sur la propriété de la dérivée

Une des propriétés que nous avons introduite, mais pas développée, est la propriété de dérivée. Dans une contrainte monovariée, 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. Il serait donc intéressant d'implanter et d'évaluer ces stratégies.

Stratégies multi-propriétés

Dans cette thèse, nous avons introduit un ensemble de propriétés. Les stratégies que nous avons proposées tirent profit dans le meilleur des cas de deux de ces dernières.

Il serait intéressant de définir une stratégie combinant un ensemble plus important de propriétés. La combinaison de propriétés pertinentes doit s'accompagner d'un ordre d'application de ces propriétés. Cet ordre commencerait par exemple par les propriétés spécifiques et ponctuelles (e.g., l'absorption ou la cancellation), avant de passer à des propriétés moins éphémères (e.g., la densité), mais couplées à des propriétés plus classiques pour discriminer les candidats de même rang (e.g., de même densité).

Stratégie dynamique

La stratégie générale combinant plusieurs propriétés proposées dans la section précédente est intéressante, mais ce n'est pas toujours une bonne idée d'appliquer la même stratégie *ad vitam aeternam* jusqu'à qu'elle ne soit plus applicable. Par exemple, les stratégies basées sur l'absorption sont intéressantes, mais une fois que la variable qui maximise l'absorption est instanciée est-il toujours pertinent de rester sur cette stratégie ? Une stratégie plus dynamique qui tire profit de choix locaux avant d'appliquer d'autres propriétés est une piste qui reste à développer.

Stratégie basée sur les erreurs

Dans notre équipe, Rémy Garcia et Claude Michel travaillent sur l'incorporation d'un nouveau domaine sur les erreurs. Ce domaine fournit une borne des erreurs des variables dans la partie flottante du solveur Objective-CP. Une piste intéressante serait de développé des stratégies basées sur ces domaines d'erreurs. Par exemple, définir une stratégie de choix de variables qui choisit la variable maximisant l'erreur est simple à mettre en oeuvre et pertinente. Pour le choix de sous-domaine, une première approche peut être d'échantillonner le domaine de la variable choisie et de choisir le sous-domaine avec la borne inférieure de l'erreur la plus grande.

Une stratégie basée sur l'erreur devrait s'inscrire avantageusement dans un schéma de stratégie dynamique.

Expérimentations

La dernière perspective, et non des moindres, consiste à augmenter le nombre et la taille des benchmarks traités. C'est un point clef pour conforter nos résultats expérimentaux grâce à une variabilité plus importante de problème.

Chapitre 9

Annexes

La Table 9.1 présente l'ensemble des benchmarks avec solutions avec le nombres de variables et contraintes du modèle correspondant. Parmi ces benchmarks ceux dont une solution existe avec une absorption "Y" et ceux sans absorption "N" sont explicités.

La Table 9.2 présente l'ensemble des benchmarks sans solutions avec le nombres de variables et contraintes du modèle correspondant.

La Table 9.3 reprends les résultats présentées à [Zitoun et al., 2017] des différentes stratégies de choix de variables, pour résoudre un ensemble limité de benchmarks.

benchmarks	Nombre de variables	Nombre de contraintes élémentaires	solutions avec absorption
PID_diff_opt	62	156	Y
sine_1	2	22	N
sine_2	2	22	N
sine_3	2	22	N
PID_diff_opt_5	74	191	Y
PID_diff_opt_6	86	226	Y
PID_diff_opt_7	98	261	Y
PID_diff_opt_8	110	296	Y
PID_diff_opt_9	122	331	Y
PID_diff_opt_10	134	366	Y
square_1	2	19	N
square_2	2	18	N
square_3	2	19	N
square_4	2	19	N
square_5	2	19	N
square_6	2	19	N
square_7	2	19	N
square_8	2	19	N
newton_1_4	4	39	N
newton_1_5	4	39	N
newton_1_6	4	39	N
newton_1_7	4	39	N
newton_1_8	4	39	N
newton_2_6	7	77	N
newton_2_7	7	77	N
newton_2_8	7	77	N
newton_3_6	10	115	N
newton_3_7	10	115	N
newton_3_8	10	115	N
slope26+10	7	14	N
heron156	5	23	N
heron	5	23	N
slope26-1	7	14	N
solve_quadratic	4	7	N
solve_cubic	11	31	N
MullerKahan	302	702	N
optimized_heron	4	25	N
heron10-8	5	24	N
Odometrie_1	27	123	Y
Odometrie_10	153	1014	Y
Odometrie_50	713	4974	Y
Odometrie_150	2113	14874	Y
Odometrie_200	2813	19824	Y
runge_kutta_1_1	7	45	Y
runge_kutta_1_2	11	82	Y
runge_kutta_1_3	15	119	Y
runge_kutta_1_4	19	156	Y
runge_kutta_1_5	23	193	Y

TABLE 9.1 – Descriptif des benchmarks avec solutions

benchmarks	Nombre de variables	Nombre de contraintes élémentaires
e1_2	2	4
e1_1	2	3
kepler0	7	17
kepler1	5	26
kepler2	7	37
optimized_heron156	4	24
add_01_1_1	9	23
add_01_1_2	9	23
add_01_1_3	9	23
add_01_1_4	9	23
add_01_10_1	9	23
add_01_10_2	9	23
add_01_10_3	9	23
add_01_10_4	9	23
add_01_100_1	9	23
add_01_100_2	9	23
add_01_100_3	9	23
add_01_100_4	9	23
add_01_1000_1	9	23
add_01_1000_2	9	23
add_01_1000_3	9	23
add_01_1000_4	9	23
sine_4_true-unreach-call	2	21
sine_5_true-unreach-call	2	21
sine_6_true-unreach-call	2	21
sine_7_true-unreach-call	2	21
sine_8_true-unreach-call	2	21
PID_diff_opt_5_unsat	74	163
PID_diff_opt_6_unsat	86	163
PID_diff_opt_7_unsat	98	163
PID_diff_opt_8_unsat	110	163
PID_diff_opt_9_unsat	122	163
PID_diff_opt_10_unsat	134	163
newton_1_1_true-unreach-call	4	38
newton_1_2_true-unreach-call	4	38
newton_1_3_true-unreach-call	4	38
newton_2_1_true-unreach-call	7	76
newton_2_2_true-unreach-call	7	76
newton_2_3_true-unreach-call	7	76
newton_2_4_true-unreach-call	7	76
newton_2_5_true-unreach-call	7	76
newton_3_1_true-unreach-call	10	114
newton_3_2_true-unreach-call	10	114
newton_3_3_true-unreach-call	10	114
newton_3_4_true-unreach-call	10	114
newton_3_5_true-unreach-call	10	114
slope26-10	7	14

TABLE 9.2 – Descriptif des benchmarks sans solutions

variable choice	all		with solution		without solution	
strat.	$\sum$ t (ms)	#OUT	$\sum$ t (ms)	#OUT	$\sum$ t (ms)	#OUT
maxWidth	4330019	21	2962680	14	1367339	7
minWidth	3762938	19	3470297	18	292641	1
maxCard	3231581	16	1962573	9	1269008	7
minCard	4315427	25	4023103	24	292324	1
maxDens	2573614	13	2323093	12	250521	1
minDens	4936905	27	3316894	18	1620011	9
maxMagn	4881081	24	3261049	15	1620011	9
minMagn	3722681	19	2761916	14	960765	5
maxDegree	3413676	17	1793656	8	1620020	9
minDegree	5259904	27	3639886	18	1620018	9
maxOcc	3360986	17	3071415	16	289571	1
minOcc	5259433	27	3639415	18	1620018	9
maxAbs	2728996	15	2521099	14	207897	1
minAbs	3784212	20	3492984	19	291228	1
maxCanc	3360698	17	3071986	16	288712	1
minCanc	3356934	17	3068356	16	288578	1

TABLE 9.3 – Temps totals des stratégies de choix de variables pour résoudre une petit ensemble de benchmarks

Liste des tableaux

LISTE DES TABLEAUX

Table des figures

TABLE DES FIGURES

Bibliographie

[Aït-Kaci et al., 2007] Aït-Kaci, H., Berstel, B., Junker, U., Leconte, M., and Podelski, A. (2007). Satisfiability modulo structures as constraint satisfaction : An introduction.

[Barrett et al., 2016] Barrett, C., Fontaine, P., and Tinelli, C. (2016). The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`.

[Bartk, 1970] Bartk, R. (1970). Constraint programming : In pursuit of the holy grail.

[Batnini et al., 2005] Batnini, H., Michel, C., and Rueher, M. (2005). Mind the gaps : A new splitting strategy for consistency techniques. In van Beek, P., editor, *Principles and Practice of Constraint Programming - CP 2005*, pages 77–91, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Belaid et al., 2012] Belaid, M. S., Michel, C., and Rueher, M. (2012). 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.

[Benhamou et al., 1995] Benhamou, F., Mcallester, D., and Van Hentenryck, P. (1995). Clp(intervals) revisited.

[Benz et al., 2012] Benz, F., Hildebrandt, A., and Hack, S. (2012). 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.

[Bessiere and Régin, 1996] Bessiere, C. and Régin, J.-C. (1996). Mac and combined heuristics : Two reasons to forsake fc (and cbj ?) on hard problems. pages 61–75.

[Bessière, 1994] Bessière, C. (1994). Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1) :179 – 190.

[Bessiére et al., 1999] Bessiére, C., Freuder, E. C., and Regin, J.-C. (1999). Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107(1) :125 – 148.

[Biere et al., 1999] Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. (1999). Symbolic model checking without bdds. In Cleaveland, W. R., editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Botella et al., 2006] Botella, B., Gotlieb, A., and Michel, C. (2006). Symbolic execution of floating-point computations : Research articles. *Softw. Test. Verif. Reliab.*, 16(2) :97–121.

[Boussemart et al., 2004] Boussemart, F., Hemery, F., Lecoutre, C., and Sais, L. (2004). Boosting systematic search by weighting constraints. *ECAI'04*, pages 146–150.

[Brélaz, 1979] Brélaz, D. (1979). New methods to color the vertices of a graph. *Commun. ACM*, 22(4) :251–256.

[Bruttomesso et al., 2008] Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., and Sebastiani, R. (2008). The mathsat 4 smt solver. In Gupta, A. and Malik, S., editors, *Computer Aided Verification*, pages 299–303, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Clarke et al., 2004] Clarke, E., Kroening, D., and Lerda, F. (2004). A tool for checking ansi-c programs. In Jensen, K. and Podelski, A., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176.

[Collavizza et al., 1999] Collavizza, H., Delobel, F., and Rueher, M. (1999). Comparing partial consistencies. *Reliable Computing*, 5(3) :213–228.

[Collavizza et al., 2016] Collavizza, H., Michel, C., and Rueher, M. (2016). Searching critical values for floating-point programs. In *Testing Software and Systems - 28th IFIP WG 6.1 International Conference, ICTSS 2016, Graz, Austria, October 17-19, 2016, Proceedings*, pages 209–217.

[Collavizza and Rueher, 2006] Collavizza, H. and Rueher, M. (2006). Exploration of the capabilities of constraint programming for software verification. pages 182–196.

[Collavizza et al., 2010] Collavizza, H., Rueher, M., and Van Hentenryck, P. (2010). Cpbpv : A constraint-programming framework for bounded program verification. *Constraints*, 15(2) :238–264.

[Collavizza et al., 2011] Collavizza, H., Vinh, N. L., Rueher, M., Devulder, S., and Gueguen, T. (2011). A dynamic constraint-based BMC strategy

for generating counterexamples. *26th ACM Symposium On Applied Computing*.

[Coq development team, 2009] Coq development team, T. (2009). *The Coq proof assistant reference manual*. LogiCal Project.

[Cordeiro et al., 2012] Cordeiro, L., Morse, J., Nicole, D., and Fischer, B. (2012). Context-bounded model checking with esbmc 1.17. In Flanagan, C. and König, B., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 534–537, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Cousot and Cousot, 1977a] Cousot, P. and Cousot, R. (1977a). Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA. ACM.

[Cousot and Cousot, 1977b] Cousot, P. and Cousot, R. (1977b). Static determination of dynamic properties of generalized type unions. *SIGPLAN Not.*, 12(3) :77–94.

[Cousot and Cousot, 2010] Cousot, P. and Cousot, R. (2010). *A gentle introduction to formal verification of computer systems by abstract interpretation*, pages 1–29. NATO Science Series III : Computer and Systems Sciences. IOS Press.

[Cousot et al., 2005] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Mine, A., Monniaux, D., and Rival, X. (2005). The astrée static analyzer.

[Damouche et al., 2017] Damouche, N., Martel, M., Panchekha, P., Qiu, C., Sanchez-Stern, A., and Tatlock, Z. (2017). Toward a standard benchmark format and suite for floating-point analysis. In Bogomolov, S., Martel, M., and Prabhakar, P., editors, *Numerical Software Verification*, pages 63–77, Cham. Springer International Publishing.

[de Moura and Bjørner, 2007] de Moura, L. and Bjørner, N. (2007). Efficient e-matching for smt solvers. In Pfenning, F., editor, *Automated Deduction – CADE-21*, pages 183–198, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Defense, 1992] Defense, P. M. (1992). Software problem led to system failure at dhahran, saudi arabia. *US GAO Reports, report no. GAO/IMTEC-92-26*.

[Denmat et al., 2007] Denmat, T., Gotlieb, A., and Ducassé, M. (2007). An abstract interpretation based combinator for modelling while loops in

constraint programming. In *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, pages 241–255.

[D'Silva et al., 2012a] D'Silva, V., Haller, L., Kroening, D., and Tautschnig, M. (2012a). Numeric bounds analysis with conflict-driven learning. In Flanagan, C. and König, B., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 48–63.

[D'Silva et al., 2012b] D'Silva, V., Haller, L., Kroening, D., and Tautschnig, M. (2012b). Numeric bounds analysis with conflict-driven learning. In Flanagan, C. and König, B., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 48–63, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Eén and Sörensson, 2004] Eén, N. and Sörensson, N. (2004). An extensible sat-solver. In Giunchiglia, E. and Tacchella, A., editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Einarsson, 2005] Einarsson, B. (2005). *1. What Can Go Wrong in Scientific Computing ?*, pages 3–12.

[Freuder, 1978] Freuder, E. C. (1978). Synthesizing constraint expressions. *Commun. ACM*, 21(11) :958–966.

[Gay et al., 2015] Gay, S., Hartert, R., Lecoutre, C., and Schaus, P. (2015). 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.

[Ghorbal et al., 2009] Ghorbal, K., Goubault, E., and Putot, S. (2009). The zonotope abstract domain taylor1+. In Bouajjani, A. and Maler, O., editors, *Computer Aided Verification*, pages 627–633, Berlin, Heidelberg. Springer Berlin Heidelberg.

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

[Gotlieb, 2012] Gotlieb, A. (2012). Tcas software verification using constraint programming. *The Knowledge Engineering Review*, 27(3) :343–360.

[Gotlieb et al., 2000] Gotlieb, A., Botella, B., and Rueher, M. (2000). A clp framework for computing structural test data. pages 399–413.

[Goubault and Putot, 2006] Goubault, E. and Putot, S. (2006). Static analysis of numerical algorithms. pages 18–34.

[Haralick and Elliott, 1980] Haralick, R. M. and Elliott, G. L. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3) :263 – 313.

[Harrison, 1999] Harrison, J. (1999). A machine-checked theory of floating point arithmetic. In Bertot, Y., Dowek, G., Théry, L., Hirschowitz, A., and Paulin, C., editors, *Theorem Proving in Higher Order Logics*, pages 113–130, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Hauser, 1996] Hauser, J. R. (1996). Handling floating-point exceptions in numeric programs. *ACM Trans. Program. Lang. Syst.*, 18(2) :139–174.

[Hentenryck et al., 1992] Hentenryck, P. V., Deville, Y., and Teng, C.-M. (1992). A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2) :291 – 321.

[Hoare, 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580.

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

[Jussien and Lhomme, 1998] Jussien, N. and Lhomme, O. (1998). Dynamic domain splitting for numeric csps. *ECAI*, pages 224–228.

[Kabi et al., 2016] Kabi, B., Goubault, E., and Putot, S. (2016). A concoction of zonotope abstraction and constraint programming for finding an invariant.

[Kästner et al., 2010] Kästner, D., Wilhelm, S., Nenova, S., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., and Rival, X. (2010). Astree : Proving the Absence of Runtime Errors. In Laprie, J., editor, *Embedded real time software and systems - ERTS2 2010*, Toulouse, France. AAAF, SEE, SIA.

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

[Kovásznai et al., 2016] Kovásznai, G., Fröhlich, A., and Biere, A. (2016). Complexity of fixed-size bit-vector logics. *Theory of Computing Systems*, 59(2) :323–376.

[Lebbah et al., 2002] Lebbah, Y., Rueher, M., and Michel, C. (2002). A global filtering algorithm for handling systems of quadratic equations and inequations. In Van Hentenryck, P., editor, *Principles and Practice of Constraint Programming - CP 2002*, pages 109–123, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Lhomme, 1993] Lhomme, O. (1993). 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. Morgan Kaufmann Publishers Inc.

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

[Mackworth, 1977] Mackworth, A. K. (1977). Consistency in networks of relations. *Artificial Intelligence*, 8(1) :99 – 118.

[Mackworth and Freuder, 1985] Mackworth, A. K. and Freuder, E. C. (1985). The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25(1) :65 – 74.

[Marciszewski, 1994] Marciszewski, W. (1994). *A Jaśkowski-Style System of Computer-Assisted Reasoning*, pages 85–101. Springer Netherlands, Dordrecht.

[Marre and Michel, 2010] Marre, B. and Michel, C. (2010). Improving the floating point addition and subtraction constraints. In Cohen, D., editor, *Principles and Practice of Constraint Programming – CP 2010*, pages 360–367, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Michel, 2002] Michel, C. (2002). Exact projection functions for floating point number constraints.

[Michel et al., 2003] Michel, C., Lebbah, Y., and Rueher, M. (2003). Safe embedding of the simplex algorithm in a csp framework.

[Michel et al., 2001] Michel, C., Rueher, M., and Lebbah, Y. (2001). 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.

[Michel et al., 2018] Michel, L., Pierre, S., and Pascal, V. H. (2018). Minicp : A light open-source solver for constraint programming - journal en préparation.

[Michel and Van Hentenryck, 2012] Michel, L. and Van Hentenryck, P. (2012). Activity-based search for black-box constraint programming solvers. In Beldiceanu, N., Jussien, N., and Pinson, É., 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. Springer Berlin Heidelberg.

[Miné, 2004a] Miné, A. (2004a). Relational abstract domains for the detection of floating-point run-time errors. In Schmidt, D., editor, *Programming Languages and Systems*, pages 3–17, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Miné, 2004b] Miné, A. (2004b). *Weakly Relational Numerical Abstract Domains*. Theses, Ecole Polytechnique X.

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

[Mohr and Henderson, 1986] Mohr, R. and Henderson, T. C. (1986). Arc and path consistency revisited. *Artificial Intelligence*, 28(2) :225 – 233.

[Moore, 1966] Moore, R. E. (1966). *Interval analysis [by] Ramon E. Moore*. Prentice-Hall Englewood Cliffs, N.J.

[Muller, 2005] Muller, J.-M. (2005). *Elementary Functions : Algorithms and Implementation*. Birkhauser.

[Nipkow et al., 2002] Nipkow, T., Wenzel, M., and Paulson, L. C. (2002). *Isabelle/HOL : A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg.

[Pavlotskaya, 1973] Pavlotskaya, L. M. (1973). Solvability of the halting problem for certain classes of turing machines. *Mathematical notes of the Academy of Sciences of the USSR*, 13(6) :537–541.

[Pelleau et al., 2014] Pelleau, M., Truchet, C., and Benhamou, F. (2014). The octagon abstract domain for continuous constraints. *Constraints*, 19(3) :309–337.

[Perlin, 1992] Perlin, M. (1992). Arc consistency for factorable relations. *Artificial Intelligence*, 53(2) :329 – 342.

[Podelski, 2000] Podelski, A. (2000). Model checking as constraint solving. pages 22–37.

[Ponsini et al., 2010] Ponsini, O., Collavizza, H., Fédèle, C., Michel, C., and Rueher, M. (2010). Automatic verification of loop invariants. In *2010 IEEE International Conference on Software Maintenance*, pages 1–5.

[Ponsini et al., 2012] Ponsini, O., Michel, C., and Rueher, M. (2012). Refining abstract interpretation based value analysis with constraint programming techniques. In Milano, M., editor, *Principles and Practice of Constraint Programming*, pages 593–607, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Ponsini et al., 2016] Ponsini, O., Michel, C., and Rueher, M. (2016). Verifying floating-point programs with constraint programming and abstract

interpretation techniques. *Automated Software Engineering*, 23(2) :191–217.

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

[Régin, 1994] Régin, J.-C. (1994). A filtering algorithm for constraints of difference in csps.

[Rice, 1953] Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2) :358–366.

[Rump, 1988] Rump, S. M. (1988). Reliability in computing : The role of interval methods in scientific computing. chapter Algorithms for Verified Inclusions&Mdash ;Theory and Practice, pages 109–126. Academic Press Professional, Inc., San Diego, CA, USA.

[Singh et al., 2017] Singh, G., Püschel, M., and Vechev, M. (2017). Fast polyhedra abstract domain. *SIGPLAN Not.*, 52(1) :46–59.

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

[Titolo et al., 2018a] Titolo, L., A. Muñoz, C., Feliu, M. A., and Moscato, M. (2018a). Eliminating unstable tests in floating-point programs.

[Titolo et al., 2018b] Titolo, L., Feliú, M. A., Moscato, M., and Muñoz, C. A. (2018b). An abstract interpretation framework for the round-off error analysis of floating-point programs. In Dillig, I. and Palsberg, J., editors, *Verification, Model Checking, and Abstract Interpretation*, pages 516–537, Cham. Springer International Publishing.

[Tucker and Noonan, 2007] Tucker, A. B. and Noonan, R. (2007). *Programming Languages*. McGraw-Hill, Inc., New York, NY, USA, 2 edition.

[Zitoun et al., 2017] Zitoun, H., Michel, C., Rueher, M., and Michel, L. (2017). Search strategies for floating point constraint systems. In *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, pages 707–722.