Résolution de contraintes sur les flottants dédiée à la vérification de programmes

Mohammed Belaid

UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS

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

T H È S E

pour obtenir le titre de

Docteur en Sciences

de l'Université de Nice - Sophia Antipolis
Mention : Informatique

Présentée et soutenue par

Mohammed Said Belaid

Résolution de contraintes sur les flottants dédiée à la vérification de programmes

Thèse dirigée par Michel Rueher et Claude Michel
préparée à l'I3S (UNSA/CNRS).
soutenue le 4 décembre 2013

Jury :

Yves Deville	Professeur à l'université Catholique de Louvain, Belgique	Rapporteur
Eric Goubault	Professeur à l'école Polytechnique, France	Rapporteur
Michel Rueher	Professeur à l'université de Nice Sophia Antipolis, France	Directeur
Claude Michel	Ingénieur de recherche à l'I3S, France	Co-Directeur
Arnaud Gotlieb	Simula Research lab, Norvège	Examinateur
Roberto Bagnara	Professeur à l'université de Parma, Italie	Examinateur
Yahia Lebbah	Professeur à l'université d'Oran, Algérie	Invité

À toi Abaty, mon père...

Remerciements

À faire en dernier...

Table des matières

Introduction

Contexte

La vérification et le test de programmes sont des étapes très importantes dans un processus de développement de logiciel, plus particulièrement lorsqu'il s'agit de logiciels critiques [D'Silva 2008]. Certains de ces logiciels effectuent des calculs basés sur l'arithmétique des nombres à virgule flottante, dans l'aéronautique par exemple. Or l'utilisation sans précaution de cette arithmétique peut causer de graves dégâts. Citons, par exemple, le crash de la fusée Ariane V lié à un problème de conversion d'un nombre à virgule flottante vers un entier 16 bits [Lions 1996]. L'échec de la mission de l'anti-missile Patriot est un exemple qui montre bien qu'une erreur, même mineure, peut avoir de graves conséquences. Une imprécision dans le calcul du temps (de l'ordre de 10^{-7}) n'a pas permis à l'anti-missile de déterminer correctement sa position. Le cumul de l'erreur était d'autant plus grand que ce programme était lancé bien avant le départ de l'anti-missile. Il n'a donc pas pu réaliser sa mission, laissant le missile Scud atteindre sa destination et causer la mort de 28 personnes et en blesser 100 autres. Ces événements montrent bien que la vérification et le test sont encore plus cruciaux pour les programmes avec des nombres à virgule flottante.

Problématiques

Plusieurs approches de vérification de programmes avec du calcul sur les nombres flottants ont été développées. On peut notamment citer les approches basées sur l'interprétation abstraite [Miné 2004a, Goubault 2006] qui, si elles permettent de garantir pour certains programmes la correction des opérations arithmétiques, rejettent malheureusement de nombreux programmes corrects. D'autres approches de la vérification de programmes avec des nombres à virgule flottante se basent sur la programmation par contraintes. Les contraintes ont été utilisées pour le test de programmes [Bardin 2009], ou encore, la vérification de la conformité des programmes vis-à-vis de leurs spécifications [Collavizza 2006]. Les outils conçus pour ces approches se limitent souvent au traitement de l'arithmétique des entiers comme, par exemple, Euclide [Gotlieb 2009] dédié au test de programmes et CPBPV [Collavizza 2010] dédié à la vérification de conformité d'un programme vis-à-vis de sa spécification. Il existe cependant peu d'exemples d'outils de vérification de programmes avec des nombres à virgule flottante basés sur les contraintes, en particulier, à cause de l'absence d'un solveur capable de traiter efficacement des contraintes sur les nombres à virgule flottante. En effet, les méthodes disponibles de résolution de contraintes sur les nombres à virgule flottante disponibles [Michel 2001, Michel 2002, Botella 2006], qui s'appuient sur une adaptation aux flottants d'algorithmes de filtrage sur les réels, peinent à passer à l'échelle.

Contributions

Nous nous intéressons dans cette thèse à la conception et la réalisation d'un solveur de contraintes sur les nombres à virgule flottante dédié à la vérification des programmes.

Afin de pallier les limites actuelles des solveurs sur les flottants, nous introduisons des sur-approximations des contraintes sur les flottants par des contraintes sur les réels [Belaid 2010b, Belaid 2010a]. Ces sur-approximations doivent être conservatives des solutions sur les flottants. Les contraintes obtenues par la sur-approximation sont alors traitées par un solveur sur les réels.

En se basant sur cette sur-approximation de contraintes sur les flottants, nous proposons un nouvel algorithme de filtrage de domaines sur les nombres flottants [Belaid 2012a, Belaid 2012b]. Cet algorithme utilise principalement les techniques de la programmation linéaire. L'algorithme de filtrage proposé est combiné avec les techniques existantes de résolution de contraintes sur les flottants. Une telle combinaison a permis d'améliorer l'étape de filtrage à la fois au niveau de la précision et des performances.

Un des principaux objectifs du traitement des contraintes sur les flottants est la génération de cas de test ou de contre-exemples. Nous proposons une nouvelle méthode de recherche de solutions sur les nombres flottants. Cette méthode est basée sur les solveurs SMT sur les réels et sur des techniques d'heuristique pour la recherche de solution sur les nombres flottants.

La méthode introduite offre aussi la possibilité de combiner les contraintes sur les réels et les contraintes sur les flottants. Ceci permet de détecter des différences entre un programme et sa spécification sur les réels. En particulier, dans certains cas nous pouvons produire un contre-exemple qui met en évidence la différence de comportement entre un programmes sur les flottants et son implémentation idéale sur les réels.

Les méthodes proposées ont été implémentées et expérimentées sur des programmes académiques afin de valider l'approche.

Organisation du manuscrit

Cette thèse est divisée en trois parties, chaque partie contient trois chapitres. La première partie présente l'état de l'art du domaine. Le premier chapitre introduit les bases des nombres à virgule flottante, ainsi que les problèmes majeurs du calcul sur les nombres flottants. Le deuxième chapitre décrit des méthodes existantes de vérification de programmes avec des calculs sur les flottants. Le troisième chapitre présente un état de l'art sur la programmation par contraintes sur les nombres flottants.

La deuxième partie est consacrée à nos contributions dans la résolution des contraintes sur les flottants. Le quatrième chapitre introduit le principe de base de notre proposition, i.e. la sur-approximation des contraintes sur les flottants. Dans le cinquième chapitre nous détaillons l'étape de filtrage des domaines des variables. Le

chapitre 6 est consacré à la recherche de solutions pour un système de contraintes sur les nombres flottants.

Dans la troisième partie, nous présentons une première évaluation expérimentale des méthodes présentées. Le septième chapitre décrit l'implémentation de notre solveur de contraintes sur les flottants. Dans le huitième chapitre, nous présentons les expérimentations réalisées sur un ensemble de programmes avec des nombres flottants.

Le dernier chapitre clôt ce manuscrit avec les conclusions et les perspectives de la programmation par contraintes pour la vérification des programmes numériques.

Première partie

État de l'art

Dans cette partie, nous rappelons à la fois quelques concepts de base et ensuite les différents travaux connexes aux nôtres. Dans le chapitre 1, nous rappelons les notions de base sur les nombres flottants utiles pour la compréhension du reste du manuscrit. Dans le chapitre 2, nous présentons un bref état de l'art des méthodes de vérification de programmes avec des calculs sur les nombres flottants. Dans le chapitre 3, nous nous focalisons sur les méthodes basées sur la programmation par contraintes.

Chapitre 1

Les nombres à virgule flottante

Sommaire

1.1 Introduction

De nombreux domaines de l'informatique nécessitent une manipulation efficace des nombres réels. Or l'ensemble des nombres réels est à la fois infini et continu. Une représentation complète de l'ensemble des nombres réels en machine est donc impossible.

Pour approximer les nombres réels, plusieurs approches ont été utilisées, comme la représentation en virgule fixe, les nombres rationnels... . La représentation en virgule flottante reste la plus utilisée pour approximer les nombres réels. Nous introduisons dans ce chapitre les notions de base de cette représentation.

1.2 Définition

Les nombres à virgule flottante –souvent appelés nombres flottants, ou encore flottants– sont une représentation d'un sous-ensemble fini des nombres réels. Ils sont utilisés dans les programmes informatiques pour approcher des valeurs de type réel.

Ils sont représentés par le triplet [Goldberg 1991] : signe s (0 ou 1), mantisse m et exposant e. La valeur d'un flottant est donnée par :

$$(-1)^s m \times \beta^e$$

où β est la base de calcul (2 dans le cas le plus courant).
L'exposant est un entier signé délimité par $e_{min} \leq e \leq e_{max}$.
La mantisse m est représentée par un nombre fixé de chiffres.

$$d_0.d_1 d_2 \cdots d_{p-1} \text{ avec } (0 \leq d_i < \beta)$$

où p est le nombre de chiffres qui composent la mantisse. Notons qu'il existe une relation directe entre p et la précision du calcul.

Le bit de poids fort d_0 est un bit implicite dont la valeur se déduit de celle de l'exposant e (voir 1.4.2).

1.3 Arrondi et erreur d'arrondi

La plupart du temps, le résultat d'une opération sur les flottants n'est pas un nombre flottant. Aussi, afin de n'avoir que des nombres flottants à manipuler, ce résultat doit être arrondi vers l'un des deux flottants les plus proches du résultat réel, en fonction du mode d'arrondi choisi. Les modes d'arrondi de la norme IEEE-754 seront présentés dans la sous-section 1.4.3.

L'opération d'arrondi introduit donc une erreur d'arrondi. L'erreur introduite dans les calculs sur les nombres flottants peut être mesurée de trois façons différentes : erreur absolue, erreur relative et erreur en *ulp*.

On note par $\circ$ la fonction d'arrondi :

$$\hat{x} = \circ(x)$$

où x est un nombre de type réel et $\hat{x}$ son approximation dans les nombres flottants.

L'erreur absolue

L'erreur absolue (Δ) est la distance absolue entre le résultat flottant et la valeur réelle. Cette mesure peut être utilisée pour estimer l'erreur d'arrondi d'une formule lorsqu'on connaît une majoration pour les valeurs des variables.

$$\Delta(x) = |x - \circ(x)|$$

L'erreur relative

L'erreur relative (ε) est le rapport entre l'erreur absolue et la valeur réelle.

$$\varepsilon(x) = \frac{\Delta(x)}{|x|} = \left| \frac{x - \circ(x)}{x} \right|$$

L'erreur relative est utilisée pour estimer l'erreur indépendamment de l'échelle de grandeur de la variable.

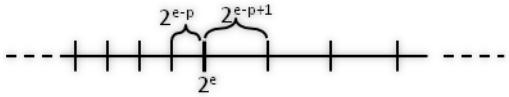


FIGURE 1.1 – Ambiguïté de la valeur de l'*ulp* au voisinage des puissances de la base β, 2 dans ce cas.

L'erreur en *ulp*

L'*ulp* (*Unit in the Last Place*) est la distance qui sépare deux flottants consécutifs [Goldberg 1991, Muller 2005] et donc, l'erreur maximale qui peut être faite lors d'un arrondi. Plus formellement, la valeur de l'*ulp* pour un nombre flottant x est définie par :

$$ulp(x) = \begin{cases} x^+ - x & \text{si} \quad x \geq 0 \\ x - x^- & \text{si} \quad x < 0 \end{cases}$$

où x^- et x^+ sont respectivement les flottants prédécesseur et successeur de x.

Cette définition de l'*ulp* reste ambiguë au voisinage des puissances de la base β (figure 1.1). Il existe plusieurs définitions différentes pour éviter cette ambiguïté (Kahan, Harrison, Goldberg...). Dans le reste du document, nous allons utiliser la définition de Goldberg étendue aux réels. Si x est un nombre réel appartenant à $[\beta^e, \beta^{e+1}[$ alors

$$ulp(x) = \beta^e \times \beta^{-p+1} = \beta^{e-p+1}$$

où β est la base de calcul (2 en général) et p désigne le nombre de chiffres de la mantisse.

1.4 Le standard IEEE-754

En 1985, l'IEEE normalise l'arithmétique des nombres flottants binaires [IEEE 1985]. En 1987, la norme IEEE-854 introduit les nombres flottants en base 10 [IEEE 1987]. En 2008, la norme a été révisée [IEEE 2008] pour lever certaines ambiguïtés, notamment dans les choix d'implémentation. La nouvelle version a été étendue à d'autres types plus précis pour les nombres flottants binaires et décimaux.

La plupart des processeurs modernes implémentent une unité de calcul sur les flottants conforme à la norme IEEE-754.

1.4.1 Formats

La norme IEEE-754 définit, principalement, les nombres à virgule flottante en base 2. Elle définit deux formats principaux pour représenter les nombres à virgule flottante :

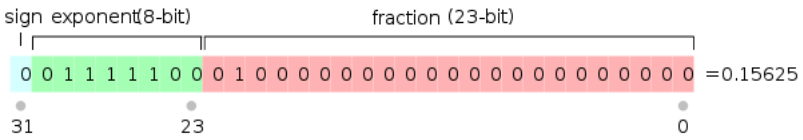


FIGURE 1.2 – Format IEEE-754 simple précision

Le format simple précision

Dans ce format, le nombre flottant est représenté par une chaîne de 32 bits (voir figure 1.2), dont 1 bit pour le signe, 24 pour la mantisse (dont le bit implicite), et 8 pour l'exposant. La valeur de l'exposant est donc comprise entre -126 et $+127$. Voir figure 1.2.

Le format double précision

Dans ce format, le nombre flottant est représenté par une chaîne de 64 bits, dont 1 bit pour le signe, 53 pour la mantisse (bit implicite inclus), et 11 pour l'exposant. La valeur de l'exposant est donc comprise entre -1022 et $+1023$.

Lors de la révision de la norme, deux autres représentations ont été introduites : le double étendu et le quadruple, ainsi qu'une représentation décimale des nombres flottants.

1.4.2 Les nombres

La norme IEEE-754 distingue les nombres normalisés des nombres dénormalisés :

Les nombres normalisés

Les nombres normalisés ont une mantisse où la partie entière est égale à 1, i.e. $x = m_x 2^{e_x}$ est normalisé avec $m_x = 1.x_1 \cdots x_{p-1}$ et $e_{max} < e_x < e_{min}$. Cette normalisation des mantisses garantit l'unicité de la représentation.

Les nombres dénormalisés

Il existe un grand vide entre 0 et le plus petit nombre normalisé (voir figure 1.3). La distance entre les deux est supérieure à l'ulp minimale. Les nombres dénormalisés ont été introduits afin de remédier à ce problème et de conserver certaines propriétés sur les nombres flottants.

Un nombre $x = m_x 2^{e_x}$ est dénormalisé si la partie entière de la mantisse est nulle, i.e., $m_x = 0.x_1 \cdots x_{p-1}$, et l'exposant est le plus petit exposant possible, i.e., $e_x = e_{min}$.

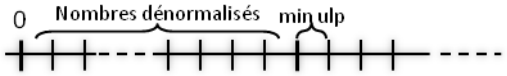


FIGURE 1.3 – Les nombres dénormalisés

Nombres	Exposant	Mantisse	Valeur
Zéros	-126	$00\cdots00$	± 0
Dénormalisés	-126	$[0.00\cdots01, 0.11\cdots11]$	$\pm[1.45\cdot10^{-45}, 1.17\cdot10^{-38}]$
Normalisés	$[-125, 126]$	$[1.00\cdots00, 1.11\cdots11]$	$\pm[1.17\cdot10^{-38}, 3.40\cdot10^{+38}]$
Infinies	$+127$	$00\cdots00$	$\pm\infty$
NaN	$+127$	$[00\cdots01, 11\cdots11]$	NaN

TABLE 1.1 – Les nombres en format simple précision

Les valeurs symboliques

La norme introduit également des valeurs symboliques telles que les infinis ($-\infty$, $+\infty$) et les NaNs (*Not a Number*).

Les infinis représentent des dépassements de capacité de calcul alors que les **NaN** représentent le résultat d'une opération impossible comme une division de zéro par lui même, ou encore, la racine carrée d'un nombre négatif. Notons aussi que le **zéro** est signé ($+0$ et -0).

Le tableau 1.1 illustre la représentation de ces nombres dans le format simple précision.

1.4.3 Les modes d'arrondi

La norme IEEE-754 définit quatre modes d'arrondi : l'arrondi vers 0, vers $-\infty$, vers $+\infty$ et au plus proche. Avec le mode d'arrondi vers $-\infty$, le réel x est arrondi vers le plus grand flottant inférieur à x, alors qu'avec le mode d'arrondi à $+\infty$, il est arrondi vers le plus petit flottant supérieur à x. Le mode d'arrondi vers 0 se comporte comme un arrondi vers $-\infty$ si x est positif, et comme un arrondi vers $+\infty$ dans les autres cas. Pour le mode d'arrondi au plus proche, le nombre réel est arrondi au flottant le plus proche de x. Lorsque x est à équidistance de deux flottants, celui avec une mantisse paire est choisi (i.e. celle dont le bit le moins significatif est à 0).

La version révisée de la norme [IEEE 2008] définit une variante du mode d'arrondi au plus proche appelée *Round to nearest away* qui, dans le cas où x est à équidistance de deux flottants, choisit celui dont la valeur absolue est la plus grande.

1.4.4 Les opérations

La norme définit cinq opérations de base : les quatre opérations arithmétiques ($+$, $-$, $\times$, $/$), et la racine carrée ($\sqrt{}$). La norme impose à ces opérations d'être **correctement arrondies**, i.e. le résultat de ces opérations sur les flottants

doit être égal à l'arrondi du résultat obtenu en effectuant l'opération équivalente sur les réels. Par exemple, pour l'addition, si $\circ$ est la fonction d'arrondi, $+$ l'addition sur les réels et $\oplus$ l'addition sur les flottants, alors

$$x \oplus y = \circ(x + y)$$

La révision de la norme IEEE-754 en 2008 est plus exigeante pour le calcul des fonctions transcendantales de base (*cos*, *sin*...). Elle suggère que ces fonctions soient correctement arrondies.

1.4.5 Les exceptions

La norme IEEE-754 spécifie cinq types d'exceptions arithmétiques sur les opérations. Ces exceptions peuvent être ignorées, une réponse silencieuse étant alors retournée. Les différents types d'exception sont expliqués par la suite et récapitulés dans le tableau 1.2.

Dépassement de capacité (*overflow*)

Cette exception est provoquée lorsque le résultat est trop grand en valeur absolue pour être représenté. Le résultat d'une opération qui provoque un *overflow* est soit un nombre infini soit la valeur maximale dans le format choisi (selon le mode d'arrondi).

Exemple : $10^{20} \otimes 10^{20}$ dans un format simple précision, provoque un dépassement de capacité et retourne la valeur symbolique $+\infty$.

Soupassement de capacité (*underflow*)

Cette exception est provoquée lorsque le résultat réel est trop petit en valeur absolue pour être représenté dans le format choisi. Le résultat d'une telle exception est le plus petit nombre dénormalisé ou 0 en fonction du mode d'arrondi.

Exemple : $10^{-45} \oslash 3$ dans un format simple précision, provoque un soupassement de capacité et retourne $+0$.

Division par zéro (*divide-by-zero*)

Cette exception est provoquée lors d'une opération de division par zéro. Une telle opération retourne un infini.

Exemple : $x \oslash 0$, avec x différent de 0 et NaN, provoque une exception de division par zéro et retourne $+\infty$ comme résultat.

Opération invalide (*invalid*)

Cette exception est provoquée lorsque l'opération est invalide. La réponse silencieuse de cette exception est la valeur symbolique NaN.

Exemples : $(-\infty) \oplus (+\infty)$, $0 \oslash 0$, $\sqrt{-1}$.

Type d'exception	La cause de l'exception	Résponse silencieuse	Exemple
Opération invalide	L'opérande est invalide pour l'opération	NaN	$(-\infty) \oplus (+\infty)$ $0 \oslash 0$, $0 \otimes \infty$
Division par zéro	Le diviseur est nul, le dividende est finis	$\pm\infty$	$x \oslash 0$, $x \neq 0$
Dépassement de capacité	Le résultat est trop grand pour être représenté	$\pm\infty$ $\pm Max_{\mathbb{F}}$	$10^{20} \otimes 10^{20}$, dans un format simple précision
Souspassement de capacité	Le résultat est trop petit pour être représenté	± 0 $\pm min_{\mathbb{F}}$	$10^{-45} \oslash 3$, dans un format simple précision
Inexact	La valeur exacte ne peut être représenté	Arrondi	$1 \oslash 3$

TABLE 1.2 – Les exceptions du calcul sur les nombres flottants

Résultat inexact (*inexact*)

Cette exception est provoquée lorsque le résultat est arrondi, i.e. la valeur flottante et le résultat mathématique exact sont différents.

Exemple : $1 \oslash 10$ ne peut être représenté exactement dans un format binaire.

1.5 Problèmes du calcul sur les flottants

Les erreurs d'arrondi introduites dans le calcul flottant rendent l'arithmétique des flottants différente de celle des réels. Nous détaillons ici certains problèmes communs avec l'arithmétique des nombres flottants.

1.5.1 Les propriétés de l'arithmétique flottante

La plupart des propriétés de l'arithmétique des réels ne sont pas préservées dans l'arithmétique flottante. Si les opérations d'addition et de multiplication restent commutatives, elles ne sont pas associatives à l'arithmétique flottante.

$$\begin{array}{rcl} x \odot y & = & y \odot x \\ (x \odot y) \odot z & \neq & x \odot (y \odot z) \\ \odot & \in & \{\oplus, \otimes\} \end{array}$$

avec x, y et z des nombres flottants, $\odot$ une opération d'addition ou de multiplication sur les flottants.

Le contre-exemple suivant montre bien que l'addition n'est pas associative avec les flottants (en simple précision).

$$\begin{array}{rcl} (-10^8 \oplus 10^8) \oplus 1 & = & 1 \\ -10^8 \oplus (10^8 \oplus 1) & = & 0 \end{array}$$

De même, la multiplication de nombres flottants n'est pas distributive par rapport à l'addition et la soustraction.

$$\begin{array}{ccc} x \otimes (y \odot z) & \neq & x \otimes y \odot x \otimes z \\ \odot & \in & \{\oplus, \ominus\} \end{array}$$

En revanche, certaines opérations sur les flottants sont exactes et n'ont aucune erreur d'arrondi. Par exemple la multiplication par les puissances de la base 2, à condition que l'opération ne provoque pas de dépassement de capacité :

$$2^n \otimes x = 2^n x$$

avec n un entier signé, x un nombre à virgule flottante.

La soustraction entre deux nombres flottants x et y est exacte dans le cas où $y/2 \leq x \leq 2y$ [Sterbenz 1974].

$$x \ominus y = x - y, \quad \text{avec} \quad y/2 \leq x \leq 2y$$

1.5.2 Absorption

Ce phénomène apparaît dans l'addition de deux nombres flottants très différents en valeur absolue. Le résultat est alors arrondi vers le plus grand en valeur absolue. Par exemple, dans un format simple précision et un mode d'arrondi au plus proche :

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

De ce fait, la série $(1 \oplus 1 \oplus 1 \oplus \cdots)$ va être convergente à partir d'une certaine valeur.

1.5.3 Cancellation

L'un des problèmes classiques de l'arithmétique flottante est l'élimination souvent appelée cancellation [Goldberg 1991]. Ce problème se produit lors d'une soustraction de deux nombres flottants très proches. Le résultat est très petit par rapport aux opérandes, et le nombre de chiffres significatif diminue. La cancellation devient plus grave si les opérandes contiennent des erreurs d'arrondi.

Ce phénomène est bien illustré par le polynôme de Rump [Rump 1988].

$$R(x, y) = \frac{1335y^6}{4} + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + \frac{11y^8}{2} + \frac{x}{2y}$$

Le signe de ce polynôme ne peut être déterminé pour $x = 77617$ et $y = 33096$. Il faut au moins 122 bits de précision pour obtenir le signe correct du polynôme. Le résultat obtenu avec le format simple ou double précision est autour de 1.172603. Alors que le résultat du polynôme sur les réels est autour de $-0.873960599...$.

1.5.4 Conversion des nombres décimaux

Dans les programmes informatiques, certaines constantes décimales ne peuvent être représentées exactement dans le format flottant binaire. Cela peut générer des erreurs d'arrondi dans le calcul flottant.

L'exemple typique de ce problème est l'échec de la mission de l'anti-missile Patriot [Office 1992], où le temps s'incrémentait avec une unité de 0.1 seconde. Cette constante ne peut être représentée correctement dans un format binaire. La conversion introduisait une erreur de l'ordre de 10^{-7}. Cette erreur s'est accumulée et a causé de grave dégâts !

1.5.5 Problèmes liés à l'implémentation

En plus des problèmes de l'arithmétique des flottants, il existe d'autres problèmes liés à l'implémentation de cette arithmétique, que ça soit au niveau du processeur ou dans la phase de compilation [Monniaux 2008].

Par exemple, certains processeurs Intel utilisent des registres de 80 bits pour stocker les résultats intermédiaires. De ce fait, le calcul d'une formule dépend de la façon dont le compilateur va allouer les registres. Par exemple, la formule suivante devrait générer une exception d'*overflow* pour $v = 10^{308}$ (format double précision).

$$x = (v \otimes v) \oslash v$$

Avec un compilateur *gcc* et un processeur Intel x86 sur 32 bits, cette formule donne comme résultat $x = 10^{308}$ et ne provoque aucun dépassement de capacité. Or le résultat intermédiaire de la multiplication $(v * v)$ devrait provoquer un *overflow*. Il ne se produit pas puisque ce résultat est enregistré dans un format double étendu sur 80 bits.

L'utilisation de ces registres peut aussi provoquer un autre phénomène : celui du double arrondi. Dans certaines situations, lorsque le mode d'arrondi est au plus proche, le résultat flottant est arrondi à deux reprises dans deux directions différentes. Cela peut donner un résultat différent par rapport à l'arrondi direct.

De plus, les compilateurs peuvent faire des optimisations de code. Un même programme peut retourner des résultats différents selon le niveau d'optimisation du compilateur.

Dans le reste du document, nous allons faire abstraction des problèmes liés à l'architecture ou la compilation des programmes avec du calcul flottant.

1.5.6 Comportement bizarre des nombres flottants

L'histoire de l'informatique présente de nombreux bugs liés aux nombres flottants. Citons par exemple :

- Le crash d'Ariane 5 qui est dû à une conversion d'un nombre flottant en 64 bits à un entier de 16 bits.
- Le problème de l'anti-missile Patriot, dû a une erreur d'arrondi de l'ordre de 10^{-7}.

- Dans les processeurs Pentium, l'opération de division n'était pas correctement arrondie pour certains cas !
- Le tableur Excel 2007 affichait la valeur 100000 pour un résultat autour de 65535.
- Dans la version 6.0 de Maple, le logiciel affiche une chaîne de caractères bizarres pour quelques nombres flottants.
- Le compilateur de Java pour les versions antérieures à la 1.6.22 rentre dans une boucle infinie si le programme contient certaines constantes sur les nombres flottants.

Les exemples suivants montrent bien que même si l'algorithme est bien conçu, le programme sur les flottants peut générer des résultats totalement différents à ceux calculés sur les nombres réels.

Calcul des intérêts

Une banque propose une nouvelle offre à ses clients : (1) le premier dépôt est fixé à 1.71828182845 $, (2) chaque année on multiplie le compte courant par le nombre d'année, (3) on prend 1 $ de frais chaque année, et cela pour 25 années [Muller 2010].

Un client décide de simuler l'offre sur sa machine. Il implémente alors le programme 1. En utilisant sa machine, il obtient un résultat de l'ordre de 10^9 $. Après 25 ans, il réalise qu'il n'a seulement que 0.03 $ dans son compte !

Toute la différence est dans l'arithmétique utilisée. Dans l'arithmétique des réels la série implémentée converge vers zéro si le solde initial est égal à $e - 1$ (avec e la base du logarithme népérien). À cause du cumul des erreurs d'arrondi, la valeur calculée avec les nombres flottants diverge vers $+\infty$.

Programme 1 Calcul des intérêts !

```
float chaotic_bank()
{
    int i;
    int N = 25;
    float x = 1.71828182845;
    for (i = 0; i < N; i++)
    {
        x = x * i - 1;
    }
    return x;
}
```

Convergence de série

Le programme 2 implémente la suite suivante :

$$u_n = 111 - \frac{1130}{u_{n-1}} + \frac{3000}{u_{n-1} * u_{n-2}}$$

avec $u_0 = 2$ et $u_1 = -4$.

Cette suite converge vers 6 en utilisant l'arithmétique des réels. Par contre, la valeur de la suite calculée avec les nombres flottants converge vers 100 !

Cet exemple et l'exemple précédent montrent bien que, même si le programme est bien implémenté, son comportement peut être totalement différent par rapport aux attentes de l'utilisateur.

Programme 2 Convergence s'une série mathématique.

```
void convergenve()
{
    float u = 2;
    float v = -4;
    for (i = 0; i < N; i++)
    {
        w = 111. - 1130./v + 3000./(v*u);
        u = v;
        v = w;
        cout << "w = " << w;
    }
}
```

1.6 Conclusion

Nous avons présenté dans ce chapitre les notions de base de l'arithmétique des nombres flottants. Nous avons également présenté les principaux problèmes liés au calcul sur les flottants et la différence avec l'arithmétique des nombres réels. Nous présentons par la suite quelques méthodes de vérification de programmes avec du calcul sur les flottants.

Chapitre 2

Vérification de programmes avec des calculs sur les flottants

Sommaire

2.1 Introduction

Nous avons présenté dans le chapitre 1 les spécificités de l'arithmétique des nombres flottants et les problèmes classiques de cette arithmétique. Ces problèmes rendent difficile l'analyse des programmes avec des calculs sur les flottants. C'est pour cela que la plupart des méthodes de vérification et validation de programmes se restreignent à des modèles simplifiés des langages de programmation. Ces modèles font généralement abstraction de l'arithmétique des nombres flottants. Or dans la réalité, les programmes réels contiennent de plus en plus de calculs avec des flottants, y compris s'agissant de programmes critiques, dans l'aéronautique par exemple.

Nous présentons dans ce chapitre quelques méthodes existantes de vérification de programmes avec des calculs sur les flottants. La première section introduit une approche statique de l'estimation de l'erreur d'arrondi. La deuxième section présente une autre approche basée sur l'interprétation abstraite pour la détection des erreurs d'exécution. Nous présentons également une approche de preuve semi-automatique de programmes avec du calcul sur les flottants. Pour chacune de ces méthodes, nous

présentons le principe de base, l'outil qui l'implémente ainsi que les apports et les limites.

Les méthodes de vérification basées sur la programmation par contraintes, qui ont un lien direct avec notre sujet, sont discutées dans le chapitre 3. Il existe également des méthodes de vérification de l'unité du calcul flottant dans le processeur [O'Leary 1999, Harrison 2000]. L'objectif principal de ces méthodes est de vérifier la conformité de l'unité avec le standard IEEE-754. Ces méthodes ne font pas l'objet de notre champ d'étude et nous ne les abordons pas dans le présent manuscrit.

2.2 Estimation de l'erreur d'arrondi

L'estimation de l'erreur d'arrondi maximale dans un programme est une problématique essentielle dans l'analyse des programmes avec du calcul sur les flottants. Nous présentons dans cette section une méthode d'analyse statique pour estimer les erreurs d'arrondi. Cette méthode a été initiée par Goubault et al [Goubault 2002, Putot 2004, Goubault 2006]. Il existe d'autres méthodes basées principalement sur l'analyse dynamique de programmes. Une des premières approches basée sur l'analyse dynamique est l'approche CESTAC [Brunet 1986]. Cette approche utilise l'arithmétique stochastique pour analyser la stabilité des programmes. L'outil CADNA [Jézéquel 2008], basé sur cette approche, est utilisé pour estimer dynamiquement les erreurs d'arrondi et pour analyser l'instabilité des programmes. La méthode CESTAC a été largement critiquée pour sa démarche probabiliste, notamment par W.Kahan qui est l'un des principaux auteurs du standard IEEE-754 [Kahan 1996].

2.2.1 Principe de la méthode d'analyse statique

La méthode initiée par Goubault et al [Goubault 2002, Putot 2004, Goubault 2006] est une méthode d'analyse statique par interprétation abstraite. Le but principal de cette méthode consiste à déterminer les erreurs d'imprécision dans le calcul sur les flottants. Cela est fait en calculant pour chaque variable un encadrement garanti de l'ensemble des valeurs atteignables ainsi que les erreurs commises lors du calcul de ses valeurs.

La méthode repose sur une abstraction du calcul basée sur l'arithmétique affine [Goubault 2005]. Chaque variable dans le programme est sur-approximée par la formule suivante :

$$x = f^x + \Sigma\, \omega_i^x \varepsilon_i + \omega_{ho}^x \varepsilon_{ho}$$

avec f^x, le nombre flottant calculé, ω_i^x, l'erreur commise dans le calcul de x au point i dans le programme, et ε_i, une variable formelle associée au point i dans le programme. $\omega_{ho}^x \varepsilon_{ho}$ regroupe les erreurs d'arrondi d'ordre supérieur à 1. Les erreurs d'ordre supérieur à 1 sont produites par les opérations de multiplication dans le programme.

Ces erreurs d'arrondi sont propagées dans le programme. Chaque opération arithmétique introduit une erreur. Cette erreur est associée au label de l'opération. Pour l'opération d'addition par exemple, nous avons l'abstraction suivante :

$$z = x +^l y = f^z + \Sigma\left(\omega_i^x \varepsilon_i + \omega_i^y \varepsilon_i\right) + \omega_{ho}^x \varepsilon_{ho} + \omega_{ho}^y \varepsilon_{ho} + err(f^x + f^y)\varepsilon_l$$

où $err(f^x + f^y)\varepsilon_l$ représente l'erreur introduite dans le calcul de $x+y$ au point l dans le programme. L'opération de multiplication introduit en plus des erreurs d'ordre supérieur à 1, qui sont regroupées dans un seul terme ε_{ho}.

Goubault et al. proposent aussi une autre abstraction basée sur les zonotopes [Goubault 2009, Ghorbal 2010, Goubault 2012]. Les zonotopes sont des ensembles de formes affines. Ils permettent de préserver la corrélation linéaire entre les variables. Chaque variable d'entrée $x \in [a, b]$ est alors exprimée par :

$$x = \frac{a+b}{2} + \frac{b-a}{2}\varepsilon_i$$

avec ε_i une variable de type réel appartenant au domaine $[-1, 1]$ qui symbolise le bruit au point i dans le programme.

2.2.2 Fluctuat

Fluctuat [1] [Goubault 2002, Delmas 2009] est un outil d'analyse statique développé au CEA [2]. Ce logiciel met en œuvre la méthode présentée précédemment. Il permet d'estimer les imprécisions de calcul : plus précisément, il permet de comparer une sur-approximation du calcul sur les flottants avec une sur-approximation du calcul sur les réels. La figure 2.1 montre l'interface graphique de cet outil.

Fluctuat prend en entrée un programme écrit en C ou Ada (avec quelques restrictions) et un ensemble de valeurs possibles de ces entrées. Il détermine pour chaque variable dans le programme :

- un encadrement des valeurs possibles sur les flottants.
- un encadrement des valeurs possibles sur les réels. Les bornes sont calculées avec une précision paramétrable à l'aide de la bibliothèque MPFR [3] [Zimmermann 2000, Fousse 2007].
- un encadrement de l'erreur d'arrondi entre la valeur sur les réels et sur les flottants.
- la contribution de chaque instruction à l'erreur.

L'outil effectue principalement trois types d'analyse :

- Une étude d'un programme pour une valeur d'entrée précise, basée sur l'exécution symbolique.
- Une analyse statique basée sur les intervalles en tant que domaine abstrait non relationnel.
- Une analyse statique plus précise basée sur les zonotope en tant que domaine abstrait faiblement relationnel.

1. http ://www.lix.polytechnique.fr/Labo/Sylvie.Putot/fluctuat.html
2. Commissariat à l'énergie atomique et aux énergies alternatives
3. Multiple Precision Floating-Point Reliably

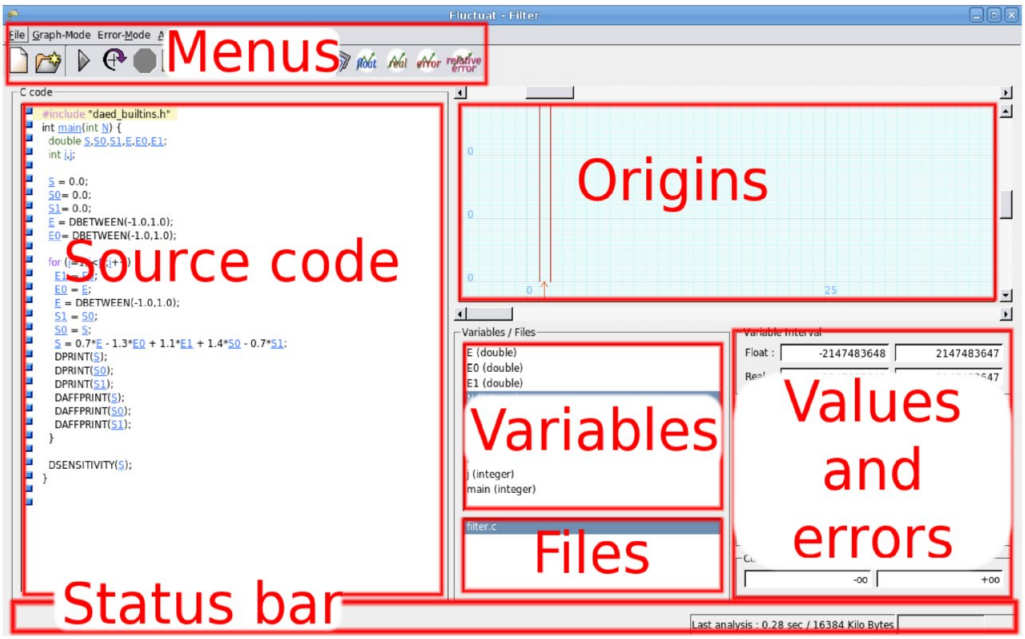


FIGURE 2.1 – L'interface graphique du logiciel Fluctuat. [Putot 2012]

Afin d'améliorer la précision d'analyse, Fluctuat permet de régler la précision arbitraire de la bibliothèque MPFR. Il permet aussi de subdiviser le domaine d'au plus deux variables d'entrée.

2.2.3 Apports et limites

Fluctuat permet d'estimer les erreurs d'arrondi dans un programme en un temps très raisonnable. Il permet aussi de comparer les domaines des valeurs calculées sur les flottants avec les valeurs correspondantes sur les réels. Contrairement aux méthodes probabilistes, la méthode d'analyse statique est complète. Comme toutes les approches basées sur l'interprétation abstraite, Fluctuat offre un bon compromis entre le temps d'analyse et la précision du calcul. Par contre, il peut générer de fausses alarmes. En effet, les sur-approximations calculées par Fluctuat sont parfois très grossières. Notamment pour les expressions non-linéaires et les instructions conditionnelles. De plus les encadrements des erreurs d'arrondi peuvent être plus grossières lorsque les domaines des variables d'entrée sont très grands.

Des travaux récents étudient l'utilisation des techniques de programmation par contraintes pour améliorer l'analyse statique de Fluctuat [Ponsini 2011, Ponsini 2012]. Ces travaux montrent qu'une coopération entre les techniques basées sur l'interprétation abstraite et celles basées sur la programmation par contraintes peut améliorer efficacement l'analyse statique des programmes.

2.3 Preuve d'absence d'erreur d'exécution

Le but principal de ces méthodes est de prouver l'absence des erreurs d'exécution dans les programmes, comme la division par zéro, ou le dépassement de capacité.

Dans cette section nous présentons une méthode implémentée dans l'outil Astrée pour détecter les erreurs d'exécution d'un calcul sur les flottants. Cette méthode a été initiée par Miné dans ses travaux de thèse [Miné 2004b].

2.3.1 Principe de la méthode

Cette méthode est basée sur l'interprétation abstraite. Elle a pour but la détection statique d'erreurs d'exécution potentielles comme la division par zéro, ou le dépassement de capacité. Le principe de base est de construire des abstractions pour les calculs dans les programmes numériques. Ces abstractions sont basées sur les intervalles ou sur les domaines relationnels. Avec ces abstractions, on peut s'assurer, par exemple, qu'il n'y a aucun dépassement de capacité, ou bien que la racine carrée ne s'applique que sur des nombres positifs.

Ces abstractions ne peuvent être appliquées directement sur les nombres flottants ni même sur les entiers machine (qui utilisent l'arithmétique modulaire).

Antoine Miné propose d'étendre l'interprétation abstraite aux nombres flottants et aux entiers machines [Miné 2004a, Miné 2004b]. Pour les flottants, la méthode consiste à sur-approximer le calcul sur les nombres flottant. Pour l'abstraction par intervalle, des arrondis extérieurs des bornes d'intervalle sont necessaires, i.e. les bornes inférieures sont arrondies vers $-\infty$, alors que les bornes supérieures vers $+\infty$.

Pour les domaines relationnels, Miné propose des approximations des calculs numériques. Ces approximations introduisent les erreurs d'arrondi dans les expressions arithmétiques. Les erreurs d'arrondis introduites sont non-linéaires, Miné propose de les linéariser ou de revenir, le cas échéant, à l'abstraction par intervalle.

D'autres domaines d'abstraction ont été proposés, telles que les octogones. Les octogones ont une précision intermédiaire entre les intervalles et les polyèdres. Il en va de même pour le coût d'analyse [Miné 2006].

2.3.2 Astrée

Astrée [4][Cousot 2005, Kästner 2010] est un analyseur statique basé sur l'interprétation abstraite. Il permet d'analyser des programmes écrits dans un sous-ensemble du langage C. Il retourne en sortie un sur-ensemble des erreurs d'exécution.

Astrée permet d'assurer qu'un programme donné ne contient aucune erreur d'exécution. Ces erreurs d'exécution peuvent être arithmétiques, comme les divisions par zéro, les dépassements de capacité, ou bien les expressions invalides (la racine carrée d'un nombre négatif par exemple). Il permet aussi de garantir l'absence d'erreurs à l'exécution due à des accès mémoire incorrectes, comme les indices

4. http ://www.astree.ens.fr/

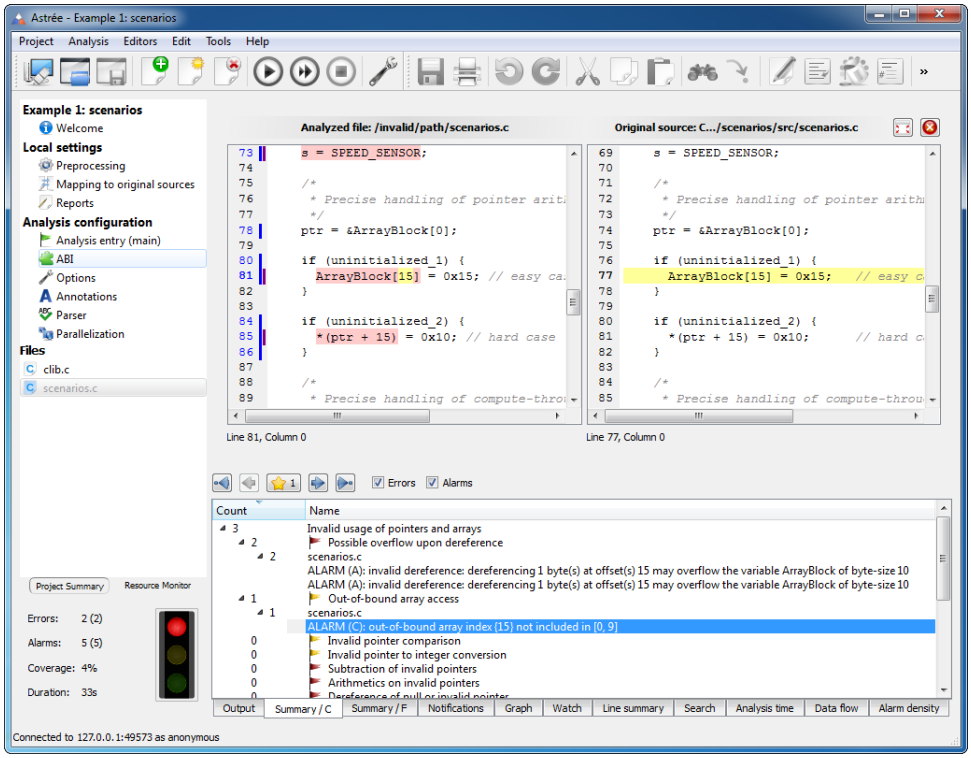


FIGURE 2.2 – L'interface graphique du logiciel Astrée.

invalides de tableau ou encore les pointeurs nuls ou invalides. Astrée offre aussi la possibilité de vérifier des assertions définies par l'utilisateur.

Astrée a été utilisé dans de nombreuses applications industrielles, notamment pour analyser certains programmes critiques d'Airbus.

2.3.3 Apports et limites

La méthode proposée par Miné permet d'analyser correctement les programmes numériques en prenant en compte les spécificités du calcul sur les nombres flottants.

Comme toutes les approches basées sur l'interprétation abstraite, cette méthode peut rejeter des programmes justes. Les sur-approximations introduites peuvent être très grossières. Du coup, la méthode peut générer de nombreuses fausses alarmes.

Cette méthode peut seulement prouver l'absence d'erreurs d'exécution. Elle ne peut détecter ni la source du problème, ni retourner un jeu de données capable de provoquer l'erreur d'exécution.

La coopération entre cette méthode et la programmation par contraintes peut apporter un plus dans les deux sens. D'un coté, les techniques de programmation par contraintes peuvent améliorer les domaines des variables calculés par Astrée. C'est l'objet des travaux de D'silva et al. dans [D'Silva 2012]. Une comparaison a été faite avec l'outil Astrée. Les premiers résultats montrent que la programmation par contraintes peut apporter des améliorations aux techniques d'interprétation

abstraite.

D'un autre coté, les méthodes d'abstraction introduites par Miné peuvent être bénéfiques pour accélérer les méthodes de filtrage des contraintes, notamment les abstractions basées sur les octogones [Pelleau 2013].

2.4 Preuve formelle

Les méthodes classiques de preuve de programme ne peuvent être appliquées directement aux nombres flottants. De nombreuses méthodes de preuve spécifiques aux flottants ont été réalisées. On distingue principalement trois types de méthode :

- Des méthodes de description formelle de l'arithmétique des nombres flottants pour les assistants de preuve [Barrett 1989, Carreño 1995].
- Des méthodes de preuve de la conformité des micro-circuits implémentant les opérations arithmétiques avec le standard IEEE-754 [O'Leary 1999, Russinoff 2000, Kaivola 2003].
- Des méthodes de formalisation pour les langages de haut niveau, notamment pour les assistants de preuve HOL et Coq.[Harrison 1999, Harrison 2000, Daumas 2001, Boldo 2006]

Parmi ces méthodes nous présentons dans la section suivante : Gappa, un des outils de preuve de programme sur les flottants.

2.4.1 Gappa

Gappa [5] est un assistant de preuve de programmes contenant du calcul sur les nombres flottants [Boldo 2009]. Afin de traiter le cas spécial des flottants, Gappa associe à chaque variable de type flottant :

- la valeur de la variable sur les flottants,
- la valeur de la variable sur les réels en utilisant MPFR,
- et l'erreur d'arrondi commise entre les deux.

Gappa offre la possibilité d'interactions avec l'assistant de preuve Coq pour prendre en compte les spécificités des nombres flottants.

2.4.2 Apports et limites

Bien que les méthodes de preuve aient été utilisées pour valider les unités de calcul sur les flottants, les recherches sur les méthodes de preuve de programmes numériques n'en sont encore qu'au début. Elles ne sont de ce fait pas encore utilisées pour prouver les programmes réels.

De plus, les méthodes de preuve de programme ne sont pas automatiques : elles nécessitent l'intervention de l'utilisateur pour guider la preuve. Ces méthodes requièrent donc une bonne expertise sur les outils de preuve.

5. Génération Automatique de Preuves de Propriétés Arithmétiques

2.5 Conclusion

Nous avons présenté dans ce chapitre quelques méthodes de vérification de programmes avec du calcul sur les nombres flottants. Nous avons également présenté les limites de ces méthodes. Nous proposons par la suite nos contributions qui essaient de remédier à certaines limites de ces méthodes, notamment la génération de contre-exemples ou la réfutation des fausses alarmes. Dans le chapitre suivant, nous nous concentrons sur les méthodes de vérification de programme basées sur la programmation par contraintes.

Chapitre 3

Contraintes sur les nombres flottants

Sommaire

3.1 Introduction

La programmation par contraintes a été utilisée dans de nombreux domaines, dont la vérification de programmes. Nous rappelons dans ce chapitre les principes de la programmation par contraintes et ses applications dans la vérification et le test des programmes. Nous montrons également la spécificité des contraintes sur les flottants par rapport aux contraintes sur les réels. Enfin, nous présentons les travaux existants sur la programmation par contraintes sur les flottants.

3.2 La programmation par contraintes

Avant d'introduire la contribution de la programmation par contraintes dans la vérification de programmes, nous rappelons les notions de base de la programmation par contraintes (PPC).

3.2.1 Définition

La programmation par contraintes est un concept de résolution de problème apparue dans les années 70 [Mackworth 1977, Jaffar 1987, Van Hentenryck 1989, Colmerauer 1985, Colmerauer 1990]. Elle consiste à poser le problème sous forme de contraintes sans se soucier de la façon de le résoudre.

Un programme à contraintes $\mathcal{CSP}$ [1] est défini par le triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ où :

- $\mathcal{X} = \{x_1, \ldots, x_n\}$ est un ensemble de variables ;
- $\mathcal{D} = \{D_{x_1}, \ldots, D_{x_n}\}$ est un ensemble de domaines (D_{x_i} est le domaine contenant toutes les valeurs potentielles de la variable x_i) ;
- $\mathcal{C} = \{c_1, \ldots, c_m\}$ est un ensemble de contraintes.

L'objectif de la programmation par contraintes est de trouver des solutions au problème. On peut chercher soit une solution, soit toutes les solutions possibles du problème.

3.2.2 Résolution de problèmes à contraintes

La résolution d'un problème repose sur deux techniques : le filtrage qui réduit l'espace de recherche et la recherche de solutions. Nous détaillons chaque étape à part par la suite.

Filtrage et propagation

Cette étape consiste à réduire les domaines des variables. On cherche d'abord à supprimer les valeurs pour lesquelles une contrainte est trivialement insatisfaite. On propage ensuite les nouveaux domaines obtenus par le filtrage aux autres contraintes. Ce processus est répété jusqu'à ce que les domaines des variables ne peuvent plus être réduits.

Le filtrage est généralement associé au concept de consistance locale telle que la consistance d'arc. Un algorithme de filtrage réalise la consistance d'arc si toute les valeurs inconsistantes sont supprimées pour une contrainte donnée [Mackworth 1977].

Recherche de solution

La deuxième étape dans le processus de résolution de contraintes consiste à trouver des solutions au problème $\mathcal{CSP}$, i.e. une instanciation des variables qui satisfait toute les contraintes du problème.

1. Constraint satisfaction problem

Une recherche consiste à instancier des variables, ou à décomposer les domaines des variables lorsqu'ils sont très grands, c'est-à-dire diviser le problème en sous-problèmes moins difficiles. L'ordre d'instanciation des variables, ainsi que le choix des valeurs d'instanciation ou du point de coupe dépendent des critères de la recherche. En général, on effectue un filtrage après chaque instanciation de variable ou décomposition du problème.

3.2.3 La programmation par contraintes sur les domaines continus

La programmation par contraintes sur les domaines continus se base sur l'arithmétique des intervalles [Sunaga 1958, Moore 1966, Benhamou 1994, Older 1993, Lebbah 2005, Lebbah 2007] Les techniques de filtrage des domaines continus sont généralement basées sur des consistances locales. Nous présentons dans cette section les consistances de base dans le traitement des contraintes sur les domaines continus.

2B-Consistance

La contrainte c est 2B-Consistante pour la variable x, de domaine $D_x = [a, b]$, s'il existe des valeurs dans les domaines de toutes les autres variables de c qui satisfont c lorsque x est instancié à $[a, a^+]$ et lorsque x est instancié avec $[b^-, b]$ [Lhomme 1993]. Prenons l'exemple de la contrainte suivante :

$$z = x + y$$

avec $z \in [-5, 3]$, $x \in [1, 3]$ et $y \in [0, 3]$.

On peut déduire de cette contrainte que $z \in [1, 3]$ en utilisant l'arithmétique des intervalles. On parle alors de fonction de projection directe. De même pour x et y, on peut déduire que $y \in [0, 2]$ alors que le domaine de x demeure inchangé. Les domaines des opérandes sont déduits par les fonctions de projection inverse. Par exemple, le domaine de y est calculé de la façon suivante (où X, Y et Z représentent les domaines des variables x, y et z) :

$$Y \leftarrow Y \cap (Z - X)$$

$$Y \leftarrow [0, 3] \cap ([1, 3] - [1, 3])$$

$$Y \leftarrow [0, 3] \cap [-2, 2]$$

$$Y \leftarrow [0, 2]$$

La différence entre la 2B-Consistance et la consistante d'arc est que cette dernière impose une condition sur tous les éléments des domaines, tandis que la 2B-Consistance n'impose une condition qu'aux seules bornes de l'intervalle.

Un des problèmes connus de la consistance d'intervalle est le problème des occurrences multiples d'une même variable qui sont traitées comme des variables différentes (mais de même domaine) par la 2B-Consistance. Les kB-Consistances ont été proposées pour remédier à ce problème [Lhomme 1993].

Par exemple, la 3B-consistance vérifie si le système de contraintes ne devient pas trivialement inconsistant lorsqu'une variable est instanciée à la valeur d'une des bornes de l'intervalle qui lui est associée.

Box-Consistance

La Box-Consistance est une approximation plus grossière de la consistance d'arc que la 2B-Consistance [Collavizza 1999]. Elle repose sur l'arithmétique des intervalles. Formellement, une contrainte c est Box-Consistante si, pour tout x_i dans $\{x_1, \ldots, x_k\}$ tel que $D_{x_i} = [a, b]$, les relations ci-dessous sont vérifiées [Benhamon 1994].

$$C(D_{x_1}, \ldots, D_{x_{i-1}}, [a, a^+[, D_{x_{i+1}}, \ldots, D_{x_k}),$$

$$C(D_{x_1}, \ldots, D_{x_{i-1}},]b^-, b], D_{x_{i+1}}, \ldots, D_{x_k}).$$

Le filtrage par la Box-Consistance est beaucoup plus efficace que la 2B si une seule variable a des occurrences multiples.

3.3 Techniques de vérification de programmes basées sur les contraintes

Dans cette section nous présentons les principales contributions de la programmation par contraintes dans le domaine de vérification de programmes. On distingue deux principales applications : la génération de cas de test, et la vérification de propriétés dans un programme.

3.3.1 Test basé sur la programmation par contraintes

Nous parlerons essentiellement des méthodes de test structurel. Le processus du test basé sur la programmation par contraintes essaie de générer des cas de test pour des chemins dans le programme [Edvardsson 1999, DeMilli 1991, Gotlieb 1998, Gotlieb 2000].

Le choix de ces chemins dépend des critères de sélection des données de test. Ces critères peuvent être par exemple :

- la couverture de tous les chemins possibles,
- la couverture de toutes les conditions dans le programme,
- la couverture de toutes les définitions de variables,
- la couverture de toutes les utilisations de variables.

Plusieurs outils implémentent les méthodes de génération de test basée sur la programmation par contraintes [Marre 2000, Sy 2001, Denmat 2007, Charreteur 2010, Albert 2012, Gotlieb 2012] ou encore PathCrawler [Williams 2005].

Certaines méthodes combinent le test basé sur la programmation par contraintes avec la génération aléatoire de donnée de test. Pour gagner en efficacité, on génère d'abord des cas de test aléatoirement. La programmation par contraintes intervient

ensuite pour trouver des cas de test pour les chemins non couverts par la génération aléatoire. C'est le cas du Framework Pex [Tillmann 2008].

Les contraintes ensemblistes ont aussi été utilisées pour le test fonctionnel ou l'animation de modèle [Bouquet 2004, Frédéric Dadeau 2011]. Les techniques mises en œuvre sont assez éloignées des nôtres et nous ne détaillerons pas ces approches.

3.3.2 Vérification basée sur la programmation par contraintes

La programmation par contraintes a été également utilisée pour la vérification de propriétés dans les programmes et pour la génération de contre-exemples. Le principe de cette méthode est simple : il suffit de trouver une solution pour un chemin dans le programme combiné avec la négation de sa post condition. Cette solution servira comme contre-exemple.

Si aucune solution n'est retournée pour tous les chemins possibles, alors le programme est correct vis-à-vis sa spécification.

L'un des outils qui implémente cette méthode est CPBPV [2] [Collavizza 2010, Collavizza 2011].

3.4 Spécificité des contraintes sur les flottants

Dans cette section, on essaie de répondre à la question clef suivante : pourquoi avons nous besoin d'un solveur spécifique aux contraintes sur les nombres flottants ? Pourquoi ne pas utiliser un solveur de contraintes sur les réels ?

La réponse est évidente : comme montré dans le chapitre 1, l'arithmétique des réels est différente de celle des nombres flottants. Pour illustrer cette différence sur les contraintes, prenons l'exemple suivant :

$$x + 1000 = 1000$$

Il est évident que $(x = 0)$ est la solution unique de cette contrainte si elle est interprétée sur les réels.

Si cette même contrainte est interprétée sur les nombres flottants :

$$x \oplus 1000 == 1000$$

où x un nombre flottant en simple précision, il existe de nombreuses solutions [3] dans l'intervalle $[-5.9 \cdot 10^{-5}, 5.9 \cdot 10^{-5}]$.

Pour montrer l'impact de cette différence dans les programmes numériques, prenons l'exemple du programme 3 [Botella 2006]. Dans ce programme le chemin $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ correspond à l'ensemble de contraintes suivant :

$$\begin{cases} x <= 10000 \\ x + 10^{12} > 10^{12} \end{cases}$$

2. A Constraint-Programming framework for Bounded Program Verification
3. Pour le mode d'arrondi au plus proche

Interprété sur les nombres réels, cet ensemble de contraintes est faisable et a de nombreuses solutions $x \in]0, 10000]$. Cependant, il n'existe aucune solution sur les nombres flottants qui correspond au chemin en question !

Programme 3 Programme qui montre la différence du traitement des contraintes sur les flottants par rapport aux réels

```
void foo(float x)
{
    float y = 1.0e12;
    float z;
1.  if (x <= 10000.0)
    {
2.      z = x + y;
        ...
    }
3.  if (z > y)
4.      ...
}
```

Un solveur de contraintes spécifique aux flottants est donc nécessaire pour pouvoir bénéficier des méthodes de vérification basées sur la programmation par contraintes. Nous présentons dans la section suivante les principaux travaux existants sur la programmation par contraintes sur les nombres flottants.

3.5 Travaux sur la résolution de contraintes sur les nombres flottants

A notre connaissance, Michel et al furent les premiers à s'intéresser à la programmation par contraintes sur les nombres flottants [Michel 2001, Michel 2002, Botella 2006]. Nous présentons d'abord les contributions de ces travaux dans le traitement des contraintes sur les nombres flottants.

3.5.1 box-consistance

La première méthode de résolution de contrainte sur les flottants propose d'étendre le concept de box-consistance aux nombres flottants [Michel 2001]. L'algorithme de filtrage basé sur la box-consistance réduit les domaines des variables par la technique dite du shaving.

Ainsi le domaine de $x \in [\underline{x}, \overline{x}]$ est réduit à $[\underline{x}', \overline{x}]$, si et seulement s'il n'existe aucune solutions dans le domaine $[\underline{x}, \underline{x}'[$.

Pour prouver qu'il n'existe pas de solution, Michel et al [Michel 2001] utilise l'arithmétique des intervalles qui reste conservative des solutions lorsqu'elle est utilisée dans le sens direct du calcul.

Bien que cette méthode est conservative de solutions, elle nécessite beaucoup de temps de calcul pour réduire les domaines des variables.

3.5.2 2b-consistance

La deuxième méthode dans [Michel 2002, Botella 2006] propose d'étendre le concept de la 2b-consistance aux nombres flottants. Cette méthode se base sur des fonctions de projections adaptées à l'arithmétique des nombres flottants. La fonction de projection directe ne requière qu'une petite modification de l'arithmétique des intervalles, alors que les fonctions de projections inverses demandent plus d'attention pour prendre en compte les spécificités des flottants.

Prenons l'exemple de l'opération d'addition $z = x \oplus y$. La 2b-consistance utilise les trois fonctions de projection suivantes :

$$\begin{cases} Z & \leftarrow & Z \cap \Pi_z(x, y) \\ X & \leftarrow & X \cap \Pi_x(z, y) \\ Y & \leftarrow & Y \cap \Pi_y(z, x) \end{cases}$$

où, x, y et z sont des variables de type flottant, $X = [\underline{x}, \overline{x}]$, $Y = [\underline{y}, \overline{y}]$ et $Z = [\underline{z}, \overline{z}]$ sont les domaines associés aux variables x, y et z respectivement. Π_z est la fonction de projection directe. Elle est calculée directement au moyen de l'arithmétique des intervalles. Par exemple, la fonction de projection directe de l'opération d'addition est la suivante :

$$\Pi_z(x, y) = [\underline{x} \oplus \underline{y}, \overline{x} \oplus \overline{y}]$$

Le calcul des bornes est fait sur les flottants avec le mode d'arrondi associé à l'opération en question.

Les fonctions de projections inverses doivent être conservatives de solutions sur les flottants. Une simple application de l'arithmétique des intervalles ne suffit pas. Prenons le cas de la fonction de projection inverse sur x dans le cas où le mode d'arrondi est vers $-\infty$. Dans ce cas précis la fonction de projection inverse est exprimée par :

$$\Pi_x(z, y) = [(\underline{z}^- \ominus_{+\infty} \overline{y})^+, \overline{z} \ominus_{-\infty} \underline{y}]$$

Plus de détail sur ces méthodes peuvent être retrouvés dans [Michel 2002, Botella 2006].

3.5.3 Amélioration des fonctions de projection

Michel et Marre [Marre 2010] ont proposé une amélioration de la fonction de projection inverse pour les opérations d'addition et de soustraction. Cette méthode utilise l'espace qui sépare deux flottants consécutifs (ulp) pour réduire la taille des domaines des opérandes.

Prenons le cas de la contrainte suivante :

$$\begin{cases} z = x \oplus_{-\infty} y \\ z = 2^- \end{cases}$$

où x, y et z sont des nombres flottants. Les domaines des variables x et y sont fixés à $[-100, 100]$.

L'application de la fonction de projection directe donne

$$z \leftarrow z \cap [-200, 200] = [2^-, 2^-]$$

L'intervalle résultat est dégénéré est ne peut donc être réduit.

Pour les fonctions de projections inverses, on obtient :

$$\begin{cases} x \leftarrow x \cap [-98^-, 102^-] = [-98^-, 100] \\ y \leftarrow y \cap [-98^-, 100^-] = [-98^-, 100^-] \end{cases}$$

Si on répète ce processus plusieurs fois et après plus de quatre heures d'exécution on finit par retrouver les domaines suivants (Plus de 10^{20} d'itérations sont requises pour atteindre ce point fixe) :

$$\begin{cases} x = [-2^-, 4^-] \\ y = [-2^-, 4^-] \\ z = [2^-, 2^-] \end{cases}$$

Michel et Marre ont proposé une nouvelle fonction de projection inverse pour l'addition et la soustraction qui donne directement ce résultat. La méthode repose sur le filtrage par l'ulp maximal. Ces fonctions de projection sont détaillées dans [Marre 2010].

Des travaux plus récents [Carlier 2011, Bagnara 2013b, Bagnara 2013a] s'intéressent à la fonction de projection inverse plus fine pour les opérations de multiplication et de division.

3.5.4 Autres méthodes

Il existe d'autres travaux sur la résolution de contraintes sur les nombres flottants. Certains travaux se basent sur les vecteurs de bits. Brillout et al [Brillout 2009] proposent une technique d'abstraction pour l'outil du model checking CBMC [4]. L'abstraction est basée sur une sous et une sur-approximation des vecteurs de bit de la mantisse. Cette méthode reste très lente.

Récemment D'Silva et al ont développé CDFL un nouvel outil d'analyse de programme. Cet outil est basé sur les techniques d'apprentissage dirigé par les conflits des solveurs SAT [D'Silva 2012] et n'utilise pas les contraintes sur les flottants.

On peut également citer Flopsy [Lakhotia 2010] une extension de l'outil Pex pour généré des cas de test. L'outil est basé sur des algorithmes génétiques d'apprentissage pour la génération de cas de test sur les nombres flottants.

4. Bounded Model Checking for ANSI-C

3.5.5 Outils

FPCS [5]

FPCS est un solveur de contraintes sur les flottants conçu au sein de l'équipe CeP [6] [Blanc 2006]. Il est basé sur les méthodes de consistance d'intervalle. Il implémente principalement l'algorithme de filtrage qui repose sur la 2B-Consistance. FPCS implémente également des consistances plus fortes comme les kb-Consistances pour remédier au problème des occurrences multiples.

FPCS supporte les opérations de base ainsi que la racine carrée. D'autres fonctions ont été implémentées comme le cosinus et le sinus. Cependant ces fonctions ne sont pas correctement arrondies. Leurs fonctions de projections en tiennent compte. Malheureusement, il n'est, dans ce cas, pas possible de garantir qu'aucune solution n'est perdue.

Pour énumérer les solutions sur les nombres flottants FPCS implémente une stratégie de recherche classique qui repose sur la bissection des domaines.

FPSE [7]

FPSE [8] est un outil d'exécution symbolique pour les calculs flottants écrits en langage C. Le solveur FPSE a été développé par Arnaud Gotlieb et Al [Botella 2005]. Ce solveur est basé sur les méthodes de consistances d'intervalle précédemment présentées. Les améliorations des fonctions de projection inverses ont été également implémenté dans cette outil [Carlier 2011, Bagnara 2013b, Bagnara 2013a].

3.6 Apports et limites des méthodes existantes

3.6.1 Le filtrage des domaines

Les algorithmes de filtrage basés sur les consistances réussissent à réduire les domaines des variables sans perdre de solutions sur les nombres flottants.

Toutefois, un des problèmes des algorithmes de filtrage basés sur les consistances locales est le problème des occurrences multiples. Illustrons ce phénomène sur un exemple :

$$z = x + y - x$$

où x, y et z sont des variables de type flottant sur 32 bits. Sur les nombres réels, cette expression peut évidemment être simplifiée à $z = y$. Ce n'est pas le cas pour les nombres flottants. Par exemple, sur les flottants et avec un mode d'arrondi au plus proche $10.0 + 10^{-8} - 10.0$ n'est pas égal à 10^{-8} mais à 0. Ce phénomène d'absorption montre bien pourquoi les expressions sur les flottants ne peuvent pas être simplifiées de la même manière que les expressions sur les réels.

5. Floating-point Constraints Solver
6. http ://www.i3s.unice.fr/∼ponsini/html/cep_tools.html
7. Floating-point Symbolique Execution
8. http ://www.irisa.fr/celtique/carlier/fpse.html

Supposons que $x \in [0.0, 10.0]$, $y \in [0.0, 10.0]$ et $z \in [0.0, 10.0^8]$. FP2B, un algorithme de 2B-consistance adapté aux flottants [Botella 2006], propage les domaines de x et y vers le domaine de z en utilisant l'arithmétique des intervalles. La propagation inverse n'étant d'aucune utilité ici, le processus de filtrage donne :

$$x \in [0.0, 10.0],\ y \in [0.0, 10.0],\ z \in [0.0, 20.0]$$

Ce résultat souligne les difficultés des algorithmes de filtrages classiques à traiter les occurrences multiples. Une consistance plus forte comme la 3B-consistance [Lhomme 1993] peut réduire le domaine de z à $[0.0, 10.01835250854492188]$. Par contre, la 3B-consistance ne réussit pas à réduire le domaine de z lorsque x et y ont tous les deux plus de deux occurrences comme dans la contrainte $z = x + y - x - y + x + y - x$.

Dans la partie suivante, nous présentons une nouvelle méthode qui permet de remédier à ce problème.

3.6.2 La recherche de solutions

Les méthodes présentées dans ce chapitre utilisent un filtrage local des domaines sur les nombres flottants. La recherche de solutions sur les nombres flottants en utilisant ces filtrages peut être très couteuse en temps.

La recherche de solutions se fait principalement par la bissection des domaines des variables. On relance donc le filtrage sur les sous-domaines obtenus jusqu'à trouver des instanciations sur les flottants qui satisfont les contraintes. Cette méthode peine à passer à l'échelle pour trouver des solutions sur les flottants, surtout lorsqu'il s'agit de grands systèmes de contraintes.

3.7 Conclusion

Nous avons rappelé dans ce chapitre les notions de la programmation par contraintes. Nous avons présenté les travaux principaux sur la résolution de contraintes sur les nombres flottants, ainsi que les limites de ces méthodes. Nous présentons dans la partie suivante nos contributions pour l'amélioration de ces techniques, à la fois au niveau du filtrage et de la recherche de solution.

Deuxième partie

Résolution de contraintes sur les flottants

Cette partie est consacrée à nos principales contributions dans la résolution des contraintes sur les flottants. Le chapitre 4 introduit le principe de base de notre proposition à savoir la sur-approximation sur les réels des contraintes sur les flottants. Dans les chapitres 5 et 6, nous présentons respectivement l'application de la sur-approximation au filtrage des domaines et l'énumération de solutions sur les nombres flottants.

Chapitre 4

Approximation des contraintes sur les flottants

> ... si on peut trouver moins cher, c'est que rien vaut déjà quelque chose !
> On peut acheter quelque chose avec rien ! En le multipliant !
> Une fois rien... c'est rien !
> Deux fois rien... c'est pas beaucoup !
> Pour trois fois rien on peut déjà acheter quelque chose !... Et pour pas cher !
>
> Raymond Devos

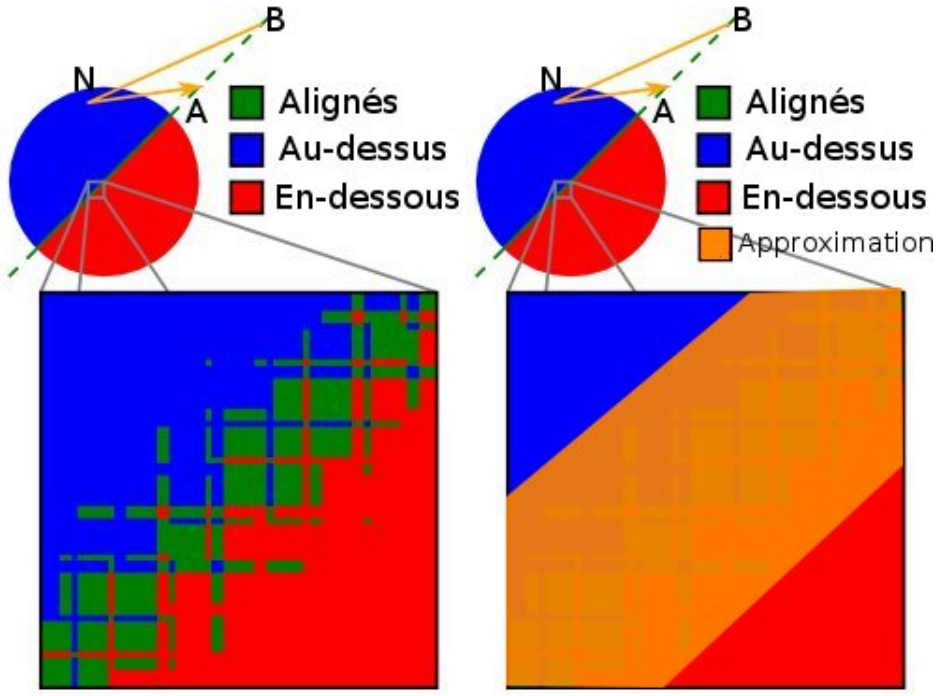
Sommaire

4.1 Introduction

Nous présentons dans ce chapitre le principe de base de notre contribution : l'approximation des contraintes sur les flottants par des contraintes sur les réels. Nous détaillons d'abord le principe de base de notre approche. Puis nous illustrons notre proposition sur un exemple simplifié. Dans les sections suivantes, nous présentons le processus de construction de ces approximations.

4.2 Principe de base

Le principe de base de notre méthode consiste à approximer sur les réels les contraintes sur les nombres flottants. Ces approximations doivent conserver le maximum d'information des contraintes originales, et ne doivent éliminer aucune solution sur les flottants.



(a) La position du point N en utilisant le calcul sur les flottants.

(b) Sur-approximation des solutions sur les flottants par des contraintes sur les réels.

FIGURE 4.1 – La position des points du plan par rapport à une droite. *Nous cherchons à déterminer la position du point N par rapport à la droite (AB). Cela est fait en calculant un déterminant d'une matrice 2×2 sur les nombre flottants.*

Afin d'illustrer l'intuition de notre proposition, prenons l'exemple dans la figure 4.1 [1]. Cette figure schématise la position des points N par rapport à la droite (AB), les points A et B étant des points fixés, N est un point variable.

La couleur verte représente les points alignés avec A et B. Les autres couleurs représentent les points non-alignés avec A et B, la couleur rouge, les points au-dessus de la droite (AB) et la couleur bleu, les points est en-dessous de cette droite. Pour déterminer la position du point N il suffit d'étudier le signe du déterminant des deux vecteurs $\overrightarrow{NA}$ et $\overrightarrow{NB}$.

1. http ://toccata.lri.fr/fp-orient.png

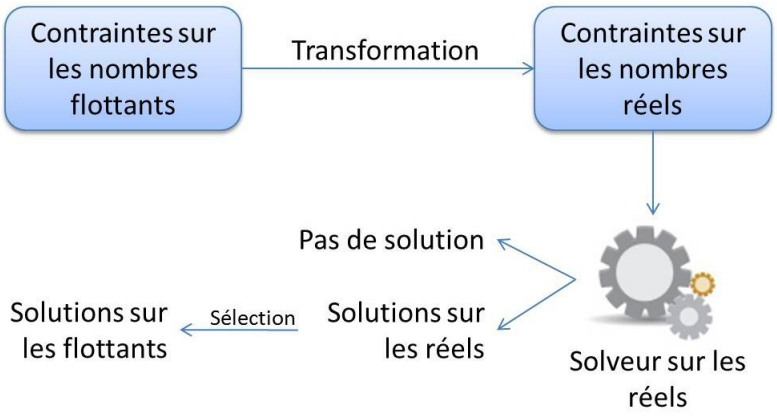


FIGURE 4.2 – Étapes de la méthode proposée : *la première étape approxime les contraintes sur les flottants par des contraintes sur les réels et les flottants ; la deuxième consiste à résoudre les contraintes obtenus, puis Les solutions flottantes sont choisies.*

La figure 4.1a montre les résultats obtenus avec le calcul du déterminant sur les nombres flottants. Elle montre bien la différence avec le calcul sur les réels.

La figure 4.1b montre l'intuition de base de notre approche. L'approche consiste à trouver un moyen pour englober toute les solutions possibles sur les flottants. Bien évidement, cette approximation peut inclure des points non-solutions (qui doivent être le moins nombreux possible).

L'intuition de notre approche est donc de construire une sur-approximation sur les réels des contraintes sur les flottants. Ensuite, les contraintes obtenues sur les réels peuvent être résolues par les solveurs existants sur les réels.

La démarche peut être illustrée par l'exemple suivant :

$$x \oplus 10.0 == 10.0$$

Interprétée sur $\mathbb{R}$, cette contrainte a une solution unique $x == 0$. Par contre, elle en a plusieurs dans l'ensemble des nombres flottants. Supposons que l'opération est faite avec un mode d'arrondi vers le plus proche. $x \oplus 10$ est égal à 10 si et seulement si le résultat réel de l'opération $x+10$ est compris entre $10-\frac{1}{2}ulp(10)$ et $10+\frac{1}{2}ulp(10)$, i.e.

$$10 - \frac{1}{2}ulp(10) \leq x + 10 \leq 10 + \frac{1}{2}ulp(10)$$

qui est une contrainte sur les réels. Pour cette contrainte, l'ensemble des flottants $x \in [-\frac{1}{2}ulp(10), \frac{1}{2}ulp(10)]$ sont solutions de la contrainte initiale sur $\mathbb{F}$. Cependant, dans le cas général, il n'est pas possible d'obtenir une approximation aussi fine des solutions.

En se basant sur ce principe, nous avons proposé une nouvelle méthode dont les étapes principales sont illustrées dans la figure 4.2. Les contraintes sur les flottants

sont d'abord approximées par des contraintes sur les réels. Les détails de cette transformation sont donnés dans les sections suivantes. Les contraintes obtenues sont ensuite traitées par des algorithmes de filtrage issus d'un solveur sur les réels. S'il n'existe aucune solution pour le système de contraintes sur les réels, alors les contraintes initiales sur les nombres à virgule flottante n'ont pas de solutions. Dans le cas contraire, une recherche combinée avec un processus d'énumération valide sur les flottants, ou la connexion avec un solveur de contraintes sur les flottants est nécessaire.

4.3 Exemple illustratif

Afin d'illustrer les cas plus généraux, considérons un type particulier de nombres flottants binaires représentés sur 4 bits, dont deux bits pour la mantisse ($p = 2$) et deux pour l'exposant. Pour des raisons de simplicité, ces flottants ne permettent que de représenter des nombres positifs (d'où l'absence de bit de signe). Les valeurs réelles des nombres représentables dans ce format sont

$$\{0, 0.25, 0.5, 0.75, 1.0, 1.5, 2.0, 3.0, 4.0, 6.0\}$$

Par exemple, 0.75 est représenté par $1.1_{(2)} \times 2^{-1}$ et 4.0 par $1.0_{(2)} \times 2^{2}$. Le mode d'arrondi considéré étant l'arrondi vers $-\infty$, nous avons, par exemple :

$$1 \oplus 0.75 = \circ(1 + 0.75) = \circ(1.75) = 1.5$$

.

Considérons le système de contraintes sur les nombres flottants suivant :

$$\mathcal{CSP}_{\mathbb{F}} = \begin{cases} x & \geq 0 \\ y & \geq 0 \\ (x \oplus y) \oplus y & \leq 4 \\ x \ominus y & \leq 2 \end{cases}$$

où $x \in \mathbb{F}$ et $y \in \mathbb{F}$.

Les solutions flottantes de ces contraintes sont représentées par les points de couleur noire dans la figure 4.3. À partir du système de contraintes précédent, il est possible de construire un système équivalent sur les réels en interprétant directement chaque opération comme étant une opération sur les réels. nous obtenons alors le système suivant :

$$\mathcal{CSP}_{\mathbb{R}} = \begin{cases} x & \geq 0 \\ y & \geq 0 \\ x + 2 \times y & \leq 4 \\ x - y & \leq 2 \end{cases}$$

où $x \in \mathbb{R}$ et $y \in \mathbb{R}$. La zone bleu de la figure 4.3 représente les solutions réelles de $\mathcal{CSP}_{\mathbb{R}}$. Notez que toutes les solutions flottantes de $\mathcal{CSP}_{\mathbb{F}}$ ne sont pas incluses dans l'espace des solutions réelles de $\mathcal{CSP}_{\mathbb{R}}$. Par exemple, le point $(3, 0.75)$ satisfait les

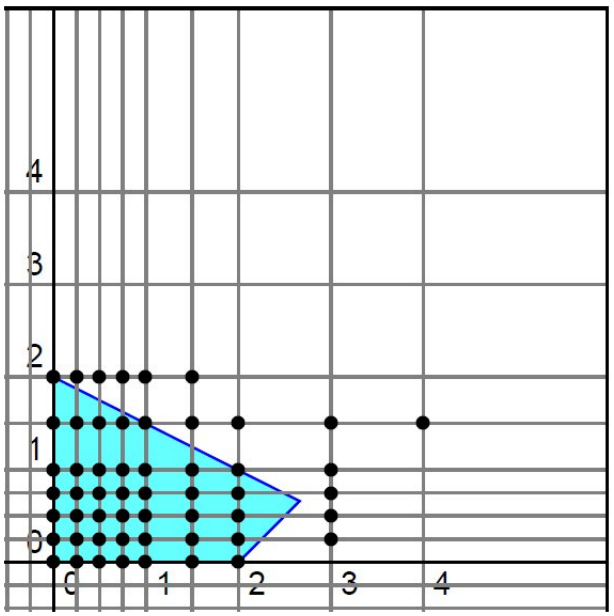


FIGURE 4.3 – Solutions réelles vs solutions flottantes. *L'espace en bleu représente les solutions des contraintes sur les réels. Les points en noir représentent les solutions sur les flottants.*

contraintes sur les flottants, puisque $(3 \oplus 0.75) \oplus 0.75 = 3 \leq 4$ et $(3 \ominus .75) = 2 \leq 2$, alors qu'il ne satisfait pas les contraintes sur les réels, puisque $(3 + 0.75) + 0.75 = 4.5 > 4$ et $(3 - .75) = 2.25 > 2$. Ce comportement met en évidence la nécessité d'utiliser une approche spécifique pour résoudre correctement les contraintes sur les flottants.

Bien qu'une interprétation directe des opérations sur les flottants en opérations sur les réels n'est pas conservative des solutions sur les flottants, il est possible d'utiliser des algorithmes de filtrage sur les réels tout en s'assurant de ne perdre aucune solution des contraintes sur les flottants grâce à des approximations correctes des contraintes sur les flottants. La figure 4.4 présente la première approximation que nous avons définie dans ce but.

Comme souvent pour une approximation correcte, elle englobe non seulement toutes les solutions des contraintes sur les flottants mais aussi des éléments non-solutions représentés ici par des points rouges. Une meilleure approximation est cependant possible comme l'illustre la figure 4.5.

Notez qu'avec cette seconde approximation, le nombre de points non-solutions a été réduit. Si le nombre d'éléments éliminés semble faible, il peut être plus significatif pour des systèmes de taille importante.

De telles approximation peuvent être traitées directement à l'aide d'algorithmes de filtrage sur les réels et permettre ainsi une réduction notable des domaines des variables. En combinant ce processus avec un solveur de contraintes sur les flottants,

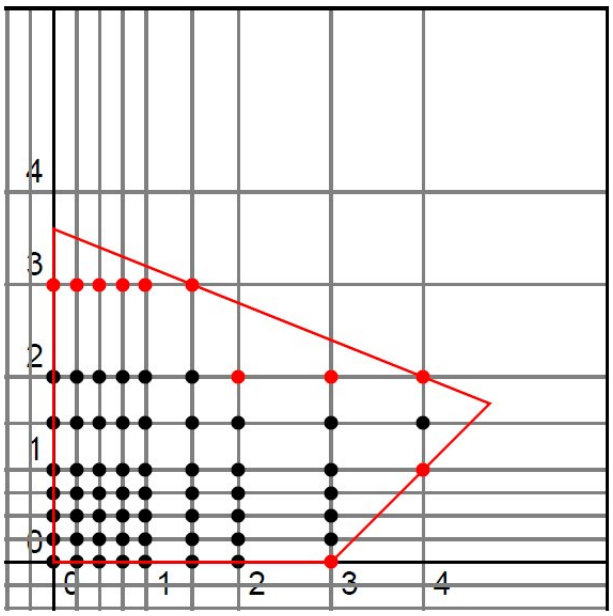


FIGURE 4.4 – Premier niveau d'approximation. *Les points en noirs représentent les solutions des contraintes sur les flottants. Les points en rouge représentent les éléments non-solution des contraintes sur les flottants et qui appartiennent à l'approximation.*

il devient possible d'obtenir, plus rapidement, des solutions correctes du système de contraintes sur les flottants.

4.4 Approximation par l'erreur relative

Cette section introduit les approximations sur les réels des contraintes sur les flottants en utilisant l'erreur relative. Ces approximations sont la pierre angulaire de notre contribution. Elles doivent non seulement être *correctes*, i.e., préserver l'ensemble des solutions du problème initial, mais aussi *fines*, i.e., inclure le moins possible de flottants non solution.

La construction de ces approximations repose sur deux techniques : l'*erreur relative* et les opérations *correctement arrondies*. La première de ces techniques est communément utilisée pour analyser la précision du calcul. La seconde propriété est garantie par toute implémentation conforme au standard IEEE 754 de l'arithmétique des flottants : une opération correctement arrondie est une opération dont le résultat sur les flottants est égal à l'arrondi du résultat de l'opération équivalente sur les réels. En d'autre terme : soient x et y deux nombres flottants, $\odot$ et $\cdot$, respectivement, une opération sur les flottants et son équivalent sur les réels, si $\odot$ est correctement arrondie alors, $x \odot y = \circ(x \cdot y)$.

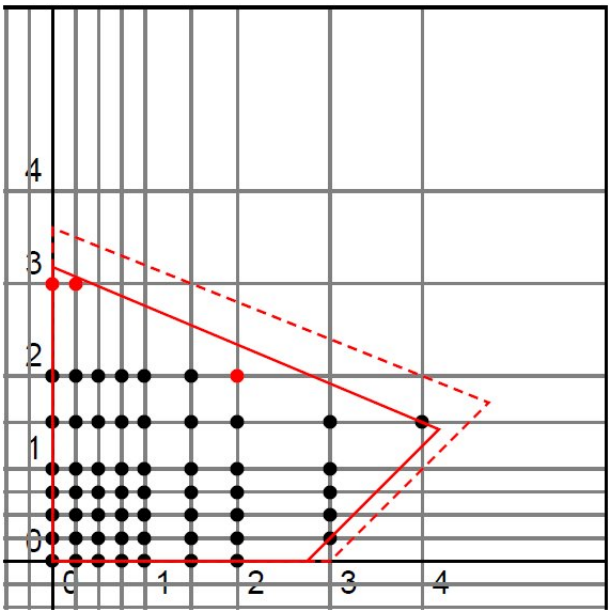


FIGURE 4.5 – Approximation plus fine *Les points noirs représentent les solutions des contraintes sur les flottants. Les points en rouge représentent les éléments non-solution des contraintes sur les flottants et qui appartiennent à l'approximation.*

La suite de cette section détaille d'abord la construction de ces approximations pour un cas particulier avant de la généraliser aux autres cas.

4.4.1 Un cas particulier

Nous présentons ici une méthode pour approximer sur les réels les opérations de base sur les flottants. Sans perte de généralité, nous nous limiterons à l'addition $\oplus$ avec un mode d'arrondi vers $-\infty$. Le même raisonnement peut être suivi pour les autres opérations et modes d'arrondi. Nous nous limitons également aux nombres flottants positifs normalisés.

Cette première approximation s'appuie sur le fait que les opérations sont correctement arrondies i.e. le résultat flottant doit être égal à l'arrondi du résultat réel. Pour que $(x + y)$ soit arrondi à $(x \oplus y)$ pour un mode d'arrondi vers $-\infty$, il doit être compris entre $(x \oplus y)$ et son successeur, i.e., $(x \oplus y)^+$ (voir figure 4.6).

Nous avons donc

$$x \oplus y \leq x + y < (x \oplus y)^+$$

avec $(x \oplus y)^+ = (x \oplus y) + ulp(x \oplus y)$. De cette relation nous pouvons tirer une approximation sur $\mathbb{R}$ pour $(x \oplus y)$:

$$(x + y) - ulp(x \oplus y) < x \oplus y \leq x + y$$

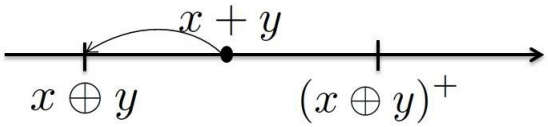


FIGURE 4.6 – La valeur réelle est comprise entre la valeur flottante de l'arrondi et sa valeur suivante.

Cependant, dans cette relation, la valeur d'$ulp(x \oplus y)$ dépend des valeurs de x et de y. S'il est possible de trouver une borne supérieure de cet ulp, une approximation plus fine peut cependant être construite. Elle est donnée par la propriété suivante :

Proposition 1. *Soient x et y, des nombres flottants dont la mantisse possède p bits. Supposons que le mode d'arrondi soit fixé à $-\infty$ et que le résultat de $x \oplus y$ soit un nombre strictement positif normalisé et strictement inférieur à $Max_{\mathbb{F}}$, le plus grand des flottants, alors on a*

$$\alpha \times (x + y) < x \oplus y \leq x + y \leq Max_{\mathbb{F}}$$

avec $\alpha = \dfrac{1}{1 + 2^{-p+1}}$ et où p est le nombre de chiffres de la mantisse.

Démonstration. Pour obtenir cette relation, nous devons d'abord calculer la borne supérieure et la borne inférieure de l'erreur relative ε :

$$\varepsilon = \left| \frac{valeur_{relle} - valeur_{flottante}}{valeur_{relle}} \right|$$

$$\varepsilon = \left| \frac{(x + y) - (x \oplus y)}{(x + y)} \right|$$

Sachant que, pour un mode d'arrondi fixé à $-\infty$, la valeur réelle de $(x + y)$ est toujours supérieure au résultat flottant de $(x \oplus y)$, et que le résultat réel est toujours positif puisque x et y sont positifs, ε est toujours positif. Notons aussi que le cas $x + y = 0$ correspond à $x \oplus y = 0$, i.e., à une erreur absolue nulle et donc une erreur relative que l'on peut considérer comme nulle. Par la suite, on suppose que $x + y > 0$, x et y étant des nombres flottants positifs normalisés. On a donc :

$$\varepsilon = \frac{(x + y) - (x \oplus y)}{(x + y)} \geq 0$$

$$= 1 - \frac{x \oplus y}{x + y}$$

Il nous faut maintenant trouver une borne supérieure pour ε.

L'addition étant correctement arrondie, on a :

$$x \oplus y \leq x + y < x \oplus y + ulp(x \oplus y)$$

donc

$$\frac{1}{x \oplus y + ulp(x \oplus y)} < \frac{1}{x + y}$$

En multipliant chaque membre de la relation par $(x \oplus y)$, on obtient :

$$\frac{x \oplus y}{x \oplus y + ulp(x \oplus y)} < \frac{x \oplus y}{x + y}$$

Ce qui, multiplié par -1 et en ajoutant 1 donne :

$$1 - \frac{x \oplus y}{x + y} < 1 - \frac{x \oplus y}{x \oplus y + ulp(x \oplus y)}$$

donc

$$0 \leq \varepsilon < \frac{ulp(x \oplus y)}{x \oplus y + ulp(x \oplus y)}$$

Posons $z = x \oplus y = m_z 2^{e_z}$ où m_z est la mantisse de z (elle est normalisée puisque les deux opérandes sont normalisées) et e_z l'exposant de z. On a alors :

$$0 \leq \varepsilon < \frac{ulp(x \oplus y)}{x \oplus y + ulp(x \oplus y)}$$

$$0 \leq \varepsilon < \frac{ulp(z)}{z + ulp(z)}$$

La valeur de l'$ulp(z)$ est donnée par $ulp(z) = 2^{-p+1}2^{e_z}$. En remplaçant $ulp(z)$ par cette formule, on obtient :

$$0 \leq \varepsilon < \frac{ulp(z)}{z + ulp(z)}$$

$$0 \leq \varepsilon < \frac{2^{-p+1}2^{e_z}}{m_z 2^{e_z} + 2^{-p+1}2^{e_z}}$$

$$0 \leq \varepsilon < \frac{2^{-p+1}}{m_z + 2^{-p+1}}$$

Donc il suffit de trouver une borne supérieure pour $2^{-p+1}/(m_z + 2^{-p+1})$ qui est maximal lorsque $m_z \in [1.0, 2.0[$ est minimal i.e. $m_z = 1.0$.

$$0 \leq \varepsilon < \frac{2^{-p+1}}{m_z + 2^{-p+1}} \leq \frac{2^{-p+1}}{1 + 2^{-p+1}}$$

donc

$$0 \leq \varepsilon < \frac{2^{-p+1}}{1 + 2^{-p+1}}$$

Il ne reste qu'à trouver une approximation pour $(x \oplus y)$. On a :

$$\begin{aligned} 0 \leq & \quad \varepsilon & < \frac{2^{-p+1}}{1+2^{-p+1}} \\ 0 \leq & \quad \frac{(x+y)-(x\oplus y)}{(x+y)} & < \frac{2^{-p+1}}{1+2^{-p+1}} \\ 0 \leq & \quad (x+y)-(x\oplus y) & < (x+y)\frac{2^{-p+1}}{1+2^{-p+1}} \end{aligned}$$

On obtient une approximation pour l'opération d'addition

$$(x+y)\frac{1}{1+2^{-p+1}} < x \oplus y \leq x+y$$

□

Donc la valeur de $(x \oplus y)$ peut être approximée par l'intervalle $]\alpha(x+y), x+y]$, avec $\alpha = 1/(1+2^{-p+1})$.

Noter que pour le cas où $x + y > Max_{\mathbb{F}}$ la propriété n'est pas valide. Prenons l'exemple où

$$x = y = Max_{\mathbb{F}}$$

Le résultat flottant de l'addition $(x \oplus y)$ pour un mode d'arrondi vers $-\infty$ est égale à la même valeur.

$$x \oplus y = Max_{\mathbb{F}}$$

L'erreur relative correspondante est égale à 0.5 :

$$\begin{aligned} \varepsilon &= \left|\frac{(x+y)-(x\oplus y)}{(x+y)}\right| \\ &= \frac{2\times Max_{\mathbb{F}} - Max_{\mathbb{F}}}{2\times Max_{\mathbb{F}}} \\ &= \frac{1}{2} \end{aligned}$$

De ce fait, la proposition ne s'applique pas lorsque $x + y > Max_{\mathbb{F}}$.

Approximation plus fine

L'approximation précédente peut encore être affinée. En effet, le résultat précédent est basé sur la relation suivante :

$$x \oplus y \leq x + y < x \oplus y + ulp(x \oplus y)$$

Or, $(x + y)$ n'atteindra jamais $x \oplus y + ulp(x \oplus y)$. En recherchant le pire cas qui peut réellement être atteint, il est possible de construire une approximation avec une inégalité large.

Comme dans la méthode précédente, nous recherchons une délimitation de l'erreur relative, mais, cette fois, dont les bornes sont atteignables par ε. La propriété suivante donne ces bornes :

Proposition 2. *Soient x et y, des nombres flottants dont la mantisse possède p bits. Supposons que le mode d'arrondi soit fixé à $-\infty$ et que le résultat de $x \oplus y$ soit un nombre strictement positif normalisé et strictement inférieur à $Max_{\mathbb{F}}$, le plus grand des flottants, alors on a*

$$0 \leq \varepsilon \leq \frac{y}{x+y}$$

De plus, l'erreur relative maximal correspond à $(x \oplus y = x)$.

Démonstration. Pour un x donné, nous avons $x \leq x \oplus y$ puisque y est positif (l'addition sur les flottants étant commutative, x et y sont ici interchangeables). Par ailleurs, le résultat réel est toujours supérieur ou égal à $x \oplus y$ pour un mode d'arrondi vers $-\infty$. On a donc :

$$x \leq x \oplus y \leq x + y$$

À partir de ces relations, on obtient :

$$\begin{aligned} x \leq \quad & x \oplus y \quad && \leq x + y \\ \frac{x}{x+y} \leq \quad & \frac{x \oplus y}{x+y} \quad && \leq 1 \\ 0 \leq \quad & 1 - \frac{x \oplus y}{x+y} \quad && \leq \frac{y}{x+y} \end{aligned}$$

Donc l'erreur relative est comprise entre

$$0 \leq \varepsilon \leq \frac{y}{x+y}$$

□

Remarquons que $y/(x+y)$ est atteignable par ε. En effet, lorsque $(x \oplus y = x)$, i.e., lorsque y est absorbé par x lors de l'opération d'arrondi, l'erreur relative est alors égale à

$$\varepsilon = \frac{(x+y) - (x \oplus y)}{(x+y)} = \frac{y}{x+y}$$

Cet encadrement plus fin de l'erreur relative nous permet de construire une approximation plus fine de l'addition sur les flottants donnée par la propriété suivante :

Proposition 3. *Soient x et y, des nombres flottants dont la mantisse possède p bits. Supposons que le mode d'arrondi soit fixé à $-\infty$ et que le résultat de $x \oplus y$ soit un nombre strictement positif normalisé et strictement inférieur à $Max_{\mathbb{F}}$, le plus grand des flottants, alors on a*

$$\gamma \times (x+y) \leq x \oplus y \leq x + y$$

avec $\gamma = \dfrac{1}{1 + (2^{-p+1} - 2^{-2p+1})}$.

Démonstration. À la fin de la démonstration de la propriété précédente, nous avons établi que l'erreur relative atteint sa borne supérieure pour

$$x \oplus y = x$$

Nous savons aussi que, pour que l'addition soit correctement arrondie, il faut que :

$$x \oplus y \leq \quad x + y \quad < (x \oplus y) + ulp(x \oplus y)$$

En combinant ces deux résultats, nous obtenons que l'erreur est maximale pour :

$$\begin{array}{rcl} x \leq & x + y & < x + ulp(x) \\ 0 \leq & y & < ulp(x) \end{array}$$

Donc, pour que $x \oplus y = x$, $y \in [0, ulp(x)[$.
Revenons à l'erreur relative ε

$$0 \leq \varepsilon \leq \frac{y}{x+y}$$

Nous devons trouver une limite supérieure pour la fonction $y/(x+y)$. La fonction $y/(x+y)$ est croissante par rapport à y. Donc elle est maximale lorsque y est maximal. Sachant que dans ce cas $y \in [0, ulp(x)[$ et $y \in \mathbb{F}$, la plus grande valeur flottante de y est le prédécesseur de $ulp(x)$ i.e. $(ulp(x))^-$. Posons $x = m_x 2^{e_x}$, l'$ulp(x)$ est donc

$$ulp(x) = 2^{e_x} 2^{-p+1} = 1.0 \times 2^{e_x - p + 1}$$

et son prédécesseur a pour valeur

$$\begin{array}{rcl} (ulp(x))^- & = & (1.0 \times 2^{e_x - p + 1})^- \\ & = & 1.11 \cdots 11 \times 2^{e_x - p} \\ & = & (2 - 2^{-p+1}) \times 2^{e_x - p} \\ & = & 2^{e_x - p + 1} - 2^{e_x - 2p + 1} \\ & = & 2^{e_x}(2^{-p+1} - 2^{-2p+1}) \end{array}$$

Remplaçons, dans la propriété 2, y par la valeur obtenue. On a

$$\begin{array}{rcl} 0 \leq \quad \varepsilon \quad & \leq & \dfrac{y}{x+y} \\ 0 \leq \quad \varepsilon \quad & \leq & \dfrac{2^{e_x}(2^{-p+1} - 2^{-2p+1})}{m_x 2^{e_x} + 2^{e_x}(2^{-p+1} - 2^{-2p+1})} \\ 0 \leq \quad \varepsilon \quad & \leq & \dfrac{2^{-p+1} - 2^{-2p+1}}{m_x + (2^{-p+1} - 2^{-2p+1})} \end{array}$$

Il suffit donc de trouver une borne supérieure à $2^{-p+1} - 2^{-2p+1}/(m_x + (2^{-p+1} - 2^{-2p+1}))$. Cette fonction est maximale lorsque m_x est minimal i.e. $m_x = 1.0$. On a donc

$$0 \leq \quad \varepsilon \quad \leq \frac{2^{-p+1} - 2^{-2p+1}}{1 + (2^{-p+1} - 2^{-2p+1})}$$

Mode d' arrondi	Négatif normalisé	Négatif dénormalisé	Positif dénormalisé	Positif normalisé
vers $-\infty$	$[(1+2^{-p+1})z_r, z_r]$	$[z_r - min_{\mathbb{F}}, z_r]$	$[z_r - min_{\mathbb{F}}, z_r]$	$[\frac{1}{(1+2^{-p+1})}z_r, z_r]$
vers $+\infty$	$[z_r, \frac{1}{(1+2^{-p+1})}z_r]$	$[z_r, z_r + min_{\mathbb{F}}]$	$[z_r, z_r + min_{\mathbb{F}}]$	$[z_r, (1+2^{-p+1})z_r]$
vers 0	$\left[z_r, \frac{1}{(1+2^{-p+1})}z_r\right]$	$[z_r - min_{\mathbb{F}}, z_r]$	$[z_r, z_r + min_{\mathbb{F}}]$	$[\frac{1}{(1+2^{-p+1})}z_r, z_r]$
au plus proche	$[(1+\frac{2^{-p}}{(1+2^{-p})})z_r,$ $(1-\frac{2^{-p}}{(1-2^{-p})})z_r]$	$[z_r - \frac{min_{\mathbb{F}}}{2},$ $z_r + \frac{min_{\mathbb{F}}}{2}]$	$[z_r - \frac{min_{\mathbb{F}}}{2}$ $z_r + \frac{min_{\mathbb{F}}}{2}]$	$[(1-\frac{2^{-p}}{(1-2^{-p})})z_r,$ $(1+\frac{2^{-p}}{(1+2^{-p})})z_r]$

TABLE 4.1 – Sur-approximations de $x \odot y$ pour chaque mode d'arrondi, avec $z_r = x \cdot y$.

En reprenant le même raisonnement que celui de la fin de la preuve de la première propriété, nous pouvons maintenant trouver l'approximation qui correspond à cette erreur relative :

$$\gamma \times (x+y) \leq x \oplus y \leq x + y$$
$$\text{avec} \gamma = \frac{1}{1 + (2^{-p+1} - 2^{-2p+1})}$$

□

Notons que $1 + 2^{-p+1} - 2^{-2p+1} < 1 + 2^{-p+1}$ et que donc $\gamma > \alpha$. Cette seconde approximation est donc meilleure que la première.

Un résultat identique peut être obtenu pour la soustraction lorsque $x \geq y$. Cependant, ce n'est pas possible pour la multiplication et la division pour lesquelles la première approximation reste vraie.

4.4.2 Généralisation

Avec le même raisonnement que celui de la proposition 1, un résultat similaire est obtenu pour les autres opérations, toujours avec un mode d'arrondi vers $-\infty$:

Proposition 4. *Soient x et y, des nombres flottants dont la mantisse possède p bits. Supposons que le mode d'arrondi soit fixé à $-\infty$ et que le résultat de $x \odot y$ soit un nombre strictement positif normalisé et strictement inférieur à max_{float}, le plus grand des flottants, alors on a*

$$\frac{1}{1+2^{-p+1}}(x \cdot y) < x \odot y \leq (x \cdot y)$$

où $\odot$ une opération basique sur les nombres flottants et $\cdot$ est l'opération équivalente sur les nombres réels.

Le tableau 4.1 donne les approximations pour les différents modes d'arrondi et les différents cas, i.e., les nombres flottants positifs ou négatifs, normalisés ou non. À chaque cas est associée une approximation correcte et fine obtenue de façon similaire à celle détaillée dans la section précédente.

Comme expliqué précédemment, certains cas particuliers permettent d'améliorer la précision des relaxations. Par exemple, l'addition avec un mode d'arrondi vers $\pm\infty$ peut être légèrement améliorée [Belaid 2010b]. La structure du problème offre aussi des possibilités d'amélioration des approximations. Par exemple, $2 \otimes x$ étant calculé exactement [2], cette expression peut directement être évaluée sur les réels.

Notez que pour chaque opération nous avons quatre cas de sur-approximations. Nous verrons par la suite que cela peut nécessiter la résolution d'un problème combinatoire.

4.5 Approximation par l'erreur absolue

Dans la méthode d'approximation basée sur l'erreur relative : pour chaque opération on a au maximum quatre sur-approximations possibles (cela dépend des domaines des opérandes), ce qui peut nécessiter la résolution d'un problème combinatoire.

Nous présentons dans cette section une autre méthode de sur-approximation basée sur la l'erreur d'arrondi absolue. La méthode consiste à sur-approximer les calculs sur les flottants par l'erreur absolue maximale.

4.5.1 Un cas particulier

Pour des raisons de simplicité, prenons le cas de l'opération d'addition avec un mode d'arrondi au plus proche. Dans ce cas, on définit la sur-approximation en utilisant l'erreur absolue par la propriété suivante :

Proposition 5. *Soient $x \in [\underline{x}, \overline{x}]$ et $y \in [\underline{y}, \overline{y}]$, des nombres flottants dont la mantisse possède p bits. Supposons que le mode d'arrondi soit fixé à $-\infty$ et que le résultat de $x \oplus y$ soit un nombre strictement positif normalisé et strictement inférieur à $Max_{\mathbb{F}}$, le plus grand des flottants, alors on a*

$$x \oplus y == x + y + err$$

$$err \in [-ulp(\overline{|x+y|}), 0]$$

avec $\overline{|x+y|}$ est le maximum en valeur absolue de l'opération $x + y$.

$$\overline{|x+y|} = max(|\underline{x} + \underline{y}|, |\overline{x} + \overline{y}|)$$

Démonstration. Pour que $(x+y)$ soit arrondi à $(x \oplus y)$ pour un mode d'arrondi vers $-\infty$, il doit être compris entre $(x \oplus y)$ et son successeur, i.e., $(x \oplus y)^+$. On a donc

$$x \oplus y \leq x + y < (x \oplus y)^+$$

avec $(x \oplus y)^+ = (|x \oplus y|) + ulp(x \oplus y)$. De cette relation on peut tirer une approximation sur $\mathbb{R}$ pour $(x \oplus y)$:

$$(x + y) - ulp(x \oplus y) < x \oplus y \leq x + y$$

2. S'il n'y a pas de dépassement de capacité.

Mode d'arrondi	l'intervalle de l'erreur err
Vers $+\infty$	$\left[0, ulp(\overline{\lvert x \cdot y \rvert})\right]$
Vers $-\infty$	$\left[-ulp(\overline{\lvert x \cdot y \rvert}), 0\right]$
Vers 0	$\left[-ulp(\overline{\lvert x \cdot y \rvert}), ulp(\overline{\lvert x \cdot y \rvert})\right]$
Au plus proche	$\left[-\frac{1}{2}ulp(\overline{\lvert x \cdot y \rvert}), \frac{1}{2}ulp(\overline{\lvert x \cdot y \rvert})\right]$

TABLE 4.2 – L'intervalle de l'erreur absolue pour chaque mode d'arrondi.

La valeur de $ulp(x \oplus y)$ est variable et dépend des valeurs de x et y. Nous cherchons donc a trouvé la valeur maximale de l'*ulp*. Cette borne correspond à la valeur maximale en valeur absolue de l'opération d'addition, notée $ulp(\overline{|x+y|})$.

$$ulp(x \oplus y) \leq ulp(\overline{|x+y|})$$

La valeur maximale de l'opération d'addition peut être obtenue par l'arithmétique des intervalles :

$$\overline{|x+y|} = max(|\underline{x}+\underline{y}|, |\overline{x}+\overline{y}|)$$

Donc, l'erreur d'arrondi err commise sur l'opération d'addition peut être délimitée par :

$$-ulp(\overline{|x+y|}) \leq err \leq 0$$

Nous pouvons ainsi réécrire l'opération d'addition sur les flottants en fonction des opération sur les réels :

$$x \oplus y == x + y + err$$

avec $err \in [-ulp(\overline{|x+y|}), 0]$

□

4.5.2 Généralisation

La propriété présentée sur l'opération d'addition peut être généralisée sur les autres opérations.

Proposition 6. *Soient $x \in [\underline{x}, \overline{x}]$ et $y \in [\underline{y}, \overline{y}]$, des nombres flottants dont la mantisse possède p bits. Supposons que le mode d'arrondi soit fixé à $-\infty$ et que le résultat de $x \oplus y$ soit un nombre strictement positif et normalisé strictement inférieur à $Max_{\mathbb{F}}$, le plus grand des flottants, alors on a*

$$x \odot y == x \cdot y + err$$

$$err \in [-ulp(\overline{|x \cdot y|}), 0]$$

avec $\overline{|x \cdot y|}$ est le maximum en valeur absolue de l'opération $x \cdot y$.

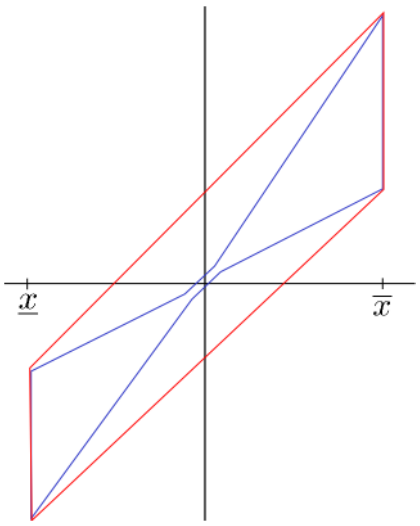


FIGURE 4.7 – Comparaison des différents types de sur-approximation. *La sur-approximation par l'erreur absolue (en rouge) est plus grossière que à la sur-approximation par l'erreur relative (en bleu). Tandis que cette dernière n'est pas convexe, contrairement à l'approximation par l'erreur absolue.*

Le tableau 4.2 résume les approximations par l'erreur absolue pour chaque mode d'arrondi. Nous donnons pour chaque mode d'arrondi l'intervalle de l'erreur d'arrondi correspondante. La valeur maximale de l'opération $\overline{|x \cdot y|}$ peut être obtenu en utilisant l'arithmétique des intervalles.

4.6 Comparaison

Le problème principal de la sur-approximation par l'erreur relative est le nombre de cas à considérer pour chaque opération. Si nous avons une contrainte avec n opérations, nous aurons au pire des cas 4^n systèmes de contraintes. Alors qu'avec l'approximation par l'erreur absolue nous avons qu'une sur-approximation par opération.

L'approximation avec l'erreur absolue dépend fortement des bornes initiales des variables, puisqu'on calcul l'erreur absolue maximale. La méthode avec l'erreur relative en est indépendante. De ce fait l'approximation par l'erreur absolue est plus grossière et peut inclure de nombreux de points non-solutions (voir figure 4.7).

Par exemple, si on a la contrainte suivante avec un mode d'arrondi au le plus proche : $x \oplus 100 == 100$, et $x \in [0, 10^{10}]$, la sur-approximation avec l'erreur absolue nous donne : $x + 100 + e = 100$, avec $e \in [-298.02, +298.02]$.

Plus de détails sur l'utilisation de ces types de sur-approximations peuvent être trouvés dans les chapitre suivants.

4.7 Conclusion

Nous avons présenté dans ce chapitre le principe de base de notre proposition. Nous présentons dans les chapitres suivant l'application de ce principe sur la résolution de contraintes sur les flottant à la fois dans l'étape du filtrage et celle de la recherche de solution.

Chapitre 5

Filtrage des domaines

Ne méprise aucune bonne action, ne serait-ce qu'en rencontrant ton Frère avec un visage souriant.

Mohamed

Sommaire

5.1 Introduction

Nous présentons dans ce chapitre comment se servir des approximations détaillées dans le chapitre 4 pour le filtrage des domaines sur les nombres flottants. Nous présentons d'abord l'algorithme de filtrage basé sur l'approximation par l'erreur relative et comment remédier au problème de l'explosion combinatoire. Nous verrons ensuite l'algorithme de filtrage basé sur les sur-approximations par l'erreur absolue et comment améliorer les bornes de ces sur-approximations.

5.2 La méthode du filtrage

Comme nous l'avons présenté dans le chapitre 3, les méthodes existantes de filtrage de contraintes sur les flottants se basent sur des consistances locales. L'inconvénient de ces méthodes vient du fait que les contraintes sont gérées indépendamment.

Nous présentons dans ce chapitre une méthode de filtrage globale des contraintes sur les flottants [Belaid 2012a, Belaid 2012b]. Cette méthode est inspirée des travaux de Lebbah et al [Lebbah 2005, Lebbah 2007]. Elle se base principalement sur la

linéarisation des contraintes non-linéaires et l'utilisation de l'algorithme du simplexe pour réduire les domaines des variables.

Les étapes principales de cette méthode sont :

- L'approximation des contraintes sur les flottants en utilisant l'une des méthodes présentées dans le chapitre 4.
- La linéarisation des termes non-linéaires en utilisant la méthode de Mc Cormick et al [McCormick 1976].
- La maximisation et minimisation de chaque variable en utilisant la programmation linéaire.

Ce processus est répété jusqu'à ce que les domaines des variables ne puissent plus être réduits.

5.3 Filtrage basé sur l'approximation par l'erreur relative

Comme présentée dans le chapitre 4, l'approximation par l'erreur relative introduit différents cas pour chaque opération du système de contraintes. Le principal problème de ces approximations est que le processus de résolution doit traiter les différents cas. Ainsi, pour n opérations élémentaires, le solveur devra potentiellement traiter 4^n combinaisons d'approximations.

Afin de diminuer substantiellement cette complexité, nous décrivons ici une combinaison des quatre cas liés à chaque arrondi en une unique sur-approximation.

5.3.1 Simplification des approximations

Pour des raisons de simplicité, considérons d'abord le cas où le mode d'arrondi est fixé à $-\infty$:

Proposition 7. *Soient x et y deux nombres flottants dont la taille de la mantisse est p. Supposons que le mode d'arrondi est fixé $-\infty$ et que le résultat de $x \odot y$ est tel que $-Max_{\mathbb{F}} < x \odot y < Max_{\mathbb{F}}$. Alors, on a :*

$$z_r - 2^{-p+1}|z_r| - min_{\mathbb{F}} \leq x \odot y \leq z_r$$

où $min_{\mathbb{F}}$ est le plus petit nombre flottant positif, $\odot$ et $\cdot$ sont, respectivement, une opération arithmétique de base sur les flottants et son équivalent sur les réels, et $z_r = x \cdot y$.

Démonstration. La première étape consiste à combiner les approximations des nombres normalisés et dénormalisés. Si $z_r > 0$ alors $\frac{1}{1+2^{-p+1}} z_r < z_r$ (voir tableau 4.1). Donc,

$$\frac{1}{1+2^{-p+1}} z_r - min_{\mathbb{F}} < z_r - min_{\mathbb{F}}$$

et

$$\frac{1}{1+2^{-p+1}} z_r - min_{\mathbb{F}} < \frac{1}{1+2^{-p+1}} z_r$$

Mode d'arrondi	Approximation de $x \odot y$
vers $-\infty$	$[z_r - 2^{-p+1}\lvert z_r\rvert - min_{\mathbb{F}}, z_r]$
vers $+\infty$	$[z_r, z_r + 2^{-p+1}\lvert z_r\rvert + min_{\mathbb{F}}]$
vers 0	$[z_r - 2^{-p+1}\lvert z_r\rvert - min_{\mathbb{F}},$ $z_r + 2^{-p+1}\lvert z_r\rvert + min_{\mathbb{F}}]$
au plus proche	$[z_r - \frac{2^{-p}}{(1-2^{-p})}\lvert z_r\rvert - \frac{min_{\mathbb{F}}}{2},$ $z_r + \frac{2^{-p}}{(1-2^{-p})}\lvert z_r\rvert + \frac{min_f}{2}]$

TABLE 5.1 – Les sur-approximations simplifiées de $x \odot y$ pour chaque mode d'arrondi (avec $z_r = x \cdot y$).

Il s'en suit que :

$$\frac{1}{1+2^{-p+1}} z_r - min_{\mathbb{F}} < x \odot y \leq z_r,\ z_r \geq 0$$

De la même manière, lorsque $z_r \leq 0$ (voir tableau 4.1), on a :

$$(1+2^{-p+1}) z_r - min_{\mathbb{F}} < x \odot y \leq z_r,\ z_r \leq 0$$

Ces deux approximations peuvent être réécrites comme suit :

$$\begin{cases} z_r - \frac{2^{-p+1}}{1+2^{-p+1}} z_r - min_{\mathbb{F}} < x \odot y \leq z_r, & z_r \geq 0 \\ z_r + 2^{-p+1} z_r - min_{\mathbb{F}} < x \odot y \leq z_r, & z_r \leq 0 \end{cases}$$

Pour combiner les approximations des cas positifs et négatifs, on utilise la valeur absolue :

$$\begin{cases} z_r - \frac{2^{-p+1}}{1+2^{-p+1}} \lvert z_r\rvert - min_{\mathbb{F}} < x \odot y \leq z_r, & z_r \geq 0 \\ z_r - 2^{-p+1} \lvert z_r\rvert - min_{\mathbb{F}} < x \odot y \leq z_r, & z_r \leq 0 \end{cases}$$

Puisque $max\{\frac{2^{-p+1}}{1+2^{-p+1}}, 2^{-p+1}\} = 2^{-p+1}$, on a

$$z_r - 2^{-p+1}\lvert z_r\rvert - min_{\mathbb{F}} \leq x \odot y \leq z_r$$

□

Le même raisonnement s'applique aux autres cas. Le tableau 5.1 donne les approximations simplifiées pour chaque mode d'arrondi.

5.3.2 Linéarisation des approximations

Les sur-approximations introduites dans la section précédente contiennent des termes non-linéaires qui ne peuvent pas être directement manipulés par un solveur linéaire. Cette section décrit comment ces termes sont approximés par un ensemble de contraintes linéaires.

Linéarisation de la valeur absolue

Les approximations simplifiées précédentes contiennent des valeurs absolues qui ne peuvent être traitées par des solveurs linéaires. Elles peuvent être soit grossièrement approximées par trois inégalités ou implémentées grâce à une décomposition classique plus fine basée sur une réécriture utilisant des "big M" :

$$\begin{cases} z = z_p - z_n \\ |z| = z_p + z_n \\ 0 \leq z_p \leq M \times b \\ 0 \leq z_n \leq M \times (1 - b) \end{cases}$$

où b est une variable booléenne, z_p et z_n sont des variables réelles positives, et M est un grand nombre flottant tel que $M \geq max\{|\underline{z}|, |\overline{z}|\}$. La méthode sépare z_p, les valeurs positives de z, de z_n, ses valeurs négatives. Lorsque $b = 1$, $z_n = 0$ et z prend ses valeurs positives avec $z = z_p = |z|$. Si $b = 0$, $z_p = 0$ et z prend ses valeurs négatives avec $z = -z_n$ et $|z| = z_n$.

Lorsque le solveur linéaire permet l'utilisation de contraintes d'indicateur, les deux dernières contraintes peuvent alors être remplacé par :

$$\begin{cases} b = 0 \rightarrow z_p = 0 \\ b = 1 \rightarrow z_n = 0 \end{cases}$$

Cette réécriture évite l'utilisation de la constante M.

Linéarisation des opérations non-linéaires

La linéarisation du produit, du carrée, et de la division sont basées sur les techniques standard proposées par Mc Cormick et al [McCormick 1976]. Elles ont aussi été utilisées dans l'algorithme de la Quad [Lebbah 2005] pour résoudre des contraintes sur les réels. $x \times y$ est linéarisé selon Mc Cormick [McCormick 1976] comme suit :

Supposons que $x \in [\underline{x}, \overline{x}]$ et $y \in [\underline{y}, \overline{y}]$, alors

$$\begin{cases} L1 : z - \underline{x}\underline{y} - \underline{y}x + \underline{x}\underline{y} \geq 0 \\ L2 : -z + \underline{x}\underline{y} + \overline{y}x - \underline{x}\overline{y} \geq 0 \\ L3 : -z + \overline{x}\overline{y} + \underline{y}x - \overline{x}\underline{y} \geq 0 \\ L4 : z - \overline{x}\overline{y} - \overline{y}x + \overline{x}\overline{y} \geq 0 \end{cases}$$

Ces linéarisations ont été prouvées optimales par Al-Khayyal et Falk [Al-Khayyal 1983]. La figure 5.1 schématise la linéarisation du produit $x \times y$ pour $z = 5$.

Lorsque $x = y$, i.e. dans le cas où $z = x \otimes x$, la linéarisation peut être améliorée : l'espace convexe de x^2 est sous-estimé par l'ensemble des tangentes à la courbe de x^2 entre $\underline{x}$ et $\overline{x}$, et sur-estimé par la ligne qui joint $(\underline{x}, \underline{x}^2)$ à $(\overline{x}, \overline{x}^2)$ (voir figure 5.2.

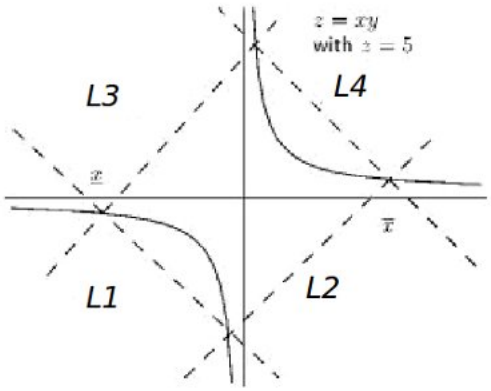


FIGURE 5.1 – La linéarisation du produit $x \times y$.

Un bon compromis est obtenu en ne prenant que les deux tangentes correspondant aux bornes de x. La linéarisation de $z = x^2$ donne donc :

$$\begin{cases} L1 : z + \underline{x}^2 - 2\underline{x}x \geq 0 \\ L2 : z + \overline{x}^2 - 2\overline{x}x \geq 0 \\ L3 : (\underline{x} + \overline{x})x - z - \underline{x}\overline{x} \geq 0 \\ L4 : z \geq 0 \end{cases}$$

La division peut bénéficier des propriétés de l'arithmétique sur les réels : observons simplement que $z = x/y$ est équivalent à $x = z \times y$ si $y \neq= 0$. Les linéarisations de Mc Cormick [McCormick 1976] s'appliquent donc aussi à ce cas. Elles nécessitent cependant les bornes de z qui sont calculables à l'aide de l'arithmétique par intervalles :

$$[\underline{z}, \overline{z}] = [\nabla(min(\underline{x}/\underline{y}, \underline{x}/\overline{y}, \overline{x}/\underline{y}, \overline{x}/\overline{y})), \\ \Delta(max(\underline{x}/\underline{y}, \underline{x}/\overline{y}, \overline{x}/\underline{y}, \overline{x}/\overline{y}))]$$

où ∇ et Δ sont respectivement les modes d'arrondis vers $-\infty$ et $+\infty$. Le domaine de la variable y ne doit pas contenir 0 ($0 \notin [\underline{y}, \overline{y}]$).

5.3.3 L'algorithme de filtrage

L'algorithme de filtrage proposé s'appuie sur les linéarisations des approximations par l'erreur relative. Les domaines des variables sont ensuite réduits au moyen d'un solveur linéaire.

Les approximations par l'erreur relative introduisent, en plus des variables initiales, des variables binaires. Cela nécessite l'utilisation d'un solveur de programmation linéaire mixte (MILP [1]). Sachant que les solveurs MILP sont implémentés

1. Mixed Integer Linear Programming

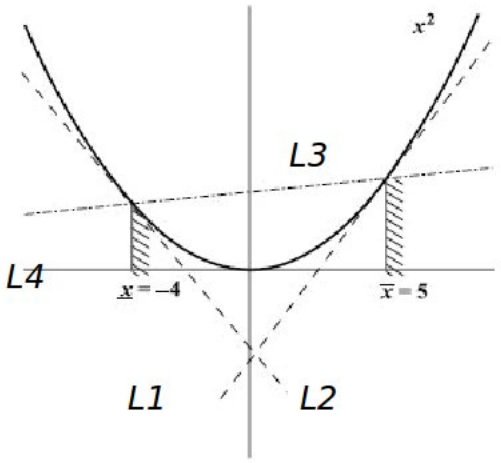


FIGURE 5.2 – La linéarisation du carré x^2.

généralement sur les nombres flottants. Il faut s'assurer que les optimisations calculées par ces solveurs sont correctes ! (voir chapitre 7 pour plus de détails).

L'algorithme 1 détaille les étapes de ce processus de filtrage. Initialement, les contraintes sur les flottants sont approximées par des contraintes non-linéaires sur les réels (la fonction **ApproximateRelative**). Puis, les termes non-linéaires de ces approximations sont linéarisés afin d'obtenir un programme linéaire mixte (la fonction **Linearise**).

Après cela, le solveur MILP est utilisé afin d'améliorer les bornes des domaines de chacune des variables. Cela est fait en minimisant et maximisant chaque variable. La fonction **safeMin** prend en entrée la fonction objectif et les contraintes linéaires. Elle retourne en sortie un minimum rigoureux de la fonction objectif. Idem pour **safeMax**.

Une fois les domaines réduits, les coefficients des linéarisations sont mis à jours (la fonction **UpdateLinearisation**). Seuls les coefficients affectés par les changements de bornes sont pris en compte.

Ce processus est répété jusqu'à ce que le pourcentage de réduction des domaines des variables soit inférieur à un ε donné.

5.4 Filtrage basé sur l'approximation par l'erreur absolue

Comme nous l'avons présenté dans le chapitre 4, l'utilisation de l'approximation par l'erreur absolue ne pose pas de problème de convexité. Par contre, les approximations introduites sont plus grossières, surtout lorsque les domaines initiaux des variables sont très grands.

Algorithm 1 Filtrage des domaines en utilisant l'approximation par l'erreur relative.

1: **Function** $FilterRelative(\mathcal{V}, \mathcal{D}, \mathcal{C}, \varepsilon)$
2: % $\mathcal{V}$: Variables flottantes
3: % $\mathcal{D}$: Domaines des variables
4: % $\mathcal{C}$: Contraintes sur les flottants
5: % ε : Réduction minimale entre deux itérations
6: $\mathcal{C}' \leftarrow$ **ApproximateRelative** $(\mathcal{C})$;
7: $\mathcal{C}'' \leftarrow$ **Linearise** $(\mathcal{C}', \mathcal{D})$;
8: $reduction \leftarrow 0$;
9: **repeat**
10: $oldReduction \leftarrow reduction$;
11: **for all** $x \in \mathcal{V}$ **do**
12: $[\underline{x}_{\mathcal{D}'}, \overline{x}_{\mathcal{D}'}] \leftarrow [$**safeMin**$(x, \mathcal{C}'')$, **safeMax**$(x, \mathcal{C}'')]$;
13: **end for**
14: $\mathcal{C}'' \leftarrow$ **UpdateLinearisations**$(\mathcal{C}'', \mathcal{D}')$;
15: $reduction \leftarrow \sum_{x \in \mathcal{V}} ((\overline{x}_{\mathcal{D}} - \underline{x}_{\mathcal{D}}) - (\overline{x}_{\mathcal{D}'} - \underline{x}_{\mathcal{D}'}))$;
16: $\mathcal{D} \leftarrow \mathcal{D}'$
17: **until** $reduction \leq \varepsilon \times oldReduction$;
18: **return** $\mathcal{D}$;

Une façon de remédier à ce problème et d'améliorer les sur-approximations de façon itérative, i.e. le processus de sur-approximation est répété jusqu'à ce que les domaines des variables ne peuvent plus être plus réduits.

Prenons l'exemple de la contrainte suivante

$$x \oplus 100 == 100$$

avec x un nombre flottant en simple précision qui prend ses valeurs dans l'intervalle $[0, 10^{10}]$.

Avec la méthode de sur-approximation par l'erreur absolue nous obtenons la contrainte sur les réels suivante :

$$x + 100 + e == 100$$

avec $e \in [-298.02, +298.02]$.

En utilisant un solveur linéaire, le domaine de x peut être réduit à $[-298.02, +298.02]$ et celui de e à $[-6 \times 10^{-6}, 6 \times 10^{-6}]$. Or avec la méthode de sur-approximation par l'erreur relative nous obtenons pour x le domaine $[-6 \times 10^{-6}, 6 \times 10^{-6}]$.

Pour améliorer les bornes obtenues par l'approximation utilisant l'erreur absolue, le processus peut être répété plusieurs fois afin d'atteindre un point fixe.

Algorithm 2 Filtrage des domaines en utilisant l'approximation par l'erreur absolue

1: **Function** $FilterAbsolute(\mathcal{V}, \mathcal{D}, \mathcal{C}, \varepsilon)$
2: % $\mathcal{V}$: Variables flottantes
3: % $\mathcal{D}$: Domaines des variables
4: % $\mathcal{C}$: Contraintes sur les flottants
5: % ε : Réduction minimale entre deux itérations
6: $\mathcal{C}' \leftarrow$ **ApproximateAbsolute** $(\mathcal{C})$;
7: $\mathcal{C}'' \leftarrow$ **Linearise** $(\mathcal{C}', \mathcal{D})$;
8: $reduction \leftarrow 0$;
9: **repeat**
10: $oldReduction \leftarrow reduction$;
11: **for all** $x \in \mathcal{V}$ **do**
12: $[\underline{x}_{\mathcal{D}'}, \overline{x}_{\mathcal{D}'}] \leftarrow [$**safeMin**$(x, \mathcal{C}'')$, **safeMax**$(x, \mathcal{C}'')]$;
13: **end for**
14: $\mathcal{C}'' \leftarrow$ **UpdateApproximation**$(\mathcal{C}'', \mathcal{D}')$;
15: $\mathcal{C}'' \leftarrow$ **UpdateLinearisations**$(\mathcal{C}'', \mathcal{D}')$;
16: $reduction \leftarrow \sum_{x \in \mathcal{V}} ((\overline{x}_{\mathcal{D}} - \underline{x}_{\mathcal{D}}) - (\overline{x}_{\mathcal{D}'} - \underline{x}_{\mathcal{D}'}))$;
17: $\mathcal{D} \leftarrow \mathcal{D}'$
18: **until** $reduction \leq \varepsilon \times oldReduction$;
19: **return** $\mathcal{D}$;

L'algorithme 2 implémente ce processus. D'abord les contraintes sur les flottants sont approximées en utilisant la méthode d'approximation par l'erreur absolue (**ApproximateAbsolute**). On obtient alors de nouvelles contraintes sur les réels. De même que dans l'algorithme précédent, les termes non-linéaires sont linéarisés (**Linearise**) en utilisant la méthode de Mc Cormick et al.

Les principales différences avec l'algorithme précédent sont :

– l'utilisation du solveur LP au lieu du MILP (puisqu'on a pas de variables binaires),
– après chaque réduction de domaines la sur-approximation par l'erreur est refaite.

Ce processus est répété jusqu'à ce que la proportion de réduction des domaines des variables soit inférieur à un ε donné.

5.5 Méthodes hybrides

Les méthodes présentées dans ce chapitre peuvent être utilisées conjointement avec les méthodes de consistances locales présentées dans le chapitre 3. Cette coopération peut nettement améliorer les performances de l'algorithme de filtrage [Belaid 2012a, Belaid 2012b].

Un simple appel à l'algorithme de la 2B sur les flottants permet d'améliorer

les bornes des variables et de les propager aux variables intermédiaires. Le coût du filtrage utilisant la 2B est relativement faible. Le chapitre 8présente plus de détail sur l'impact de la coopération avec la 2B.

L'algorithme 3 présente l'amélioration de l'algorithme de filtrage utilisant l'approximation par l'erreur relative. La fonction **FP2B** fait appel à l'algorithme de filtrage basé sur la 2B-consistance sur les contraintes originales sur les flottants.

Algorithm 3 Filtrage hybride utilisant l'approximation par l'erreur relative et la 2B-Consistance.

Function $FilterRelative2B(\mathcal{V}, \mathcal{D}, \mathcal{C}, \varepsilon)$
% $\mathcal{V}$: Variables flottantes
% $\mathcal{D}$: Domaines des variables
% $\mathcal{C}$: Contraintes sur les flottants
% ε : Réduction minimale entre deux itérations
$\mathcal{C}' \leftarrow$ **ApproximateRelative** $(\mathcal{C})$;
$\mathcal{C}'' \leftarrow$ **Linearise** $(\mathcal{C}', \mathcal{D})$;
$reduction \leftarrow 0$;
repeat
  $\mathcal{D}' \leftarrow$ **FP2B** $(\mathcal{V}, \mathcal{D}, \mathcal{C}, \varepsilon)$;
  $\mathcal{C}'' \leftarrow$ **UpdateLinearisations**$(\mathcal{C}'', \mathcal{D}')$;
  $oldReduction \leftarrow reduction$;
  for all $x \in \mathcal{V}$ **do**
    $[\underline{x}_{\mathcal{D}'}, \overline{x}_{\mathcal{D}'}] \leftarrow [$**safeMin**$(x, \mathcal{C}''),$
      $-$**safeMin**$(-x, \mathcal{C}'')]$;
  end for
  $reduction \leftarrow \sum_{x \in \mathcal{V}} ((\overline{x}_{\mathcal{D}} - \underline{x}_{\mathcal{D}}) - (\overline{x}_{\mathcal{D}'} - \underline{x}_{\mathcal{D}'}))$;
  $\mathcal{D} \leftarrow \mathcal{D}'$
until $reduction \leq \varepsilon \times oldReduction$;
return $\mathcal{D}$;

5.6 Conclusion

Nous avons présenté dans ce chapitre nos contributions dans l'étape de filtrage des domaines pour les contraintes sur les nombres flottants. Ces méthodes utilisent les sur-approximations des contraintes sur les flottants avec une méthode de filtrage globale. L'apport de ces méthodes à la fois au niveau de la performance et de la précision du filtrage est présenté dans le chapitre 8.

Dans ce chapitre, nous n'avons présenté que l'étape du filtrage des domaines. Nous présentons dans le chapitre suivant une nouvelle approche pour la recherche de solutions sur les flottants.

Chapitre 6

Recherche de solution

> En vérité, ce qui est licite est clair, et ce qui est illicite est clair ; et entre les deux se trouvent des choses douteuses ; peu sont les gens qui connaissent si elles relèvent du licite ou de l'illicite...
>
> Mohamed

Sommaire

6.1 Introduction

Nous avons présenté dans le chapitre précédent une nouvelle méthode de filtrage des contraintes sur les flottants. Cette méthode a pour but la réduction des domaines des variables. Nous présentons dans ce chapitre une nouvelle méthode pour la recherche de solutions sur les nombres flottants. Cette méthode est capable de fournir des solutions exactes sur les flottants. Elle est basée principalement sur l'utilisation de solveurs SMT et des heuristiques de recherche. Nous présentons aussi une nouvelle méthode pour comparer le calcul sur les nombres flottants avec les nombres réels.

6.2 La méthode proposée

Dans la programmation par contraintes, une des méthodes classiques de recherche de solutions consiste à : (1) subdiviser les domaines des variables, (2) effectuer un filtrage sur les sous-domaines obtenus. Ce processus est répété jusqu'à

trouver une solution. Pour les nombres flottants, cette méthode est couteuse en temps.

La méthode que nous proposons utilise les solveurs sur les réels pour résoudre les approximations des contraintes sur les flottants. Notre méthode peut être résumée par les étapes suivantes.

- Filtrage des domaines des variables. Ainsi, l'espace de recherche de solutions est réduit.
- Approximation des contraintes en utilisant la méthode basée sur l'erreur absolue.
- Recherche de solutions pour la sur-approximation à l'aide d'un solveur sur les réels. Les solutions obtenues ne sont pas forcément solutions pour du système de contraintes sur les flottants.
- Tester si les solutions obtenues sont effectivement des solutions sur les flottants. Si ce n'est pas le cas, une recherche basée sur des heuristiques est lancée afin de trouver dans le voisinage une solution exacte sur les flottants.

Cette dernière étape est une heuristique qui nous a semblé la plus pertinente dans le domaine des flottants. On peut naturellement aussi utiliser d'autres heuristiques, par exemple le retour à l'étape précédente pour la recherche d'une autre solution sur les réels assez "éloignée" de la précédente après un ou plusieurs échecs de la recherche sur les flottants dans le voisinage de la première solution sur les réels. Les sections suivantes donnent plus de détails sur ces étapes.

6.3 Recherche de solutions potentielles

Avant de commencer l'étape de recherche de solutions, il faut d'abord réduire les domaines des variables en utilisant une des méthodes de filtrage.

Les contraintes sur les flottants sont ensuite approximées par des contraintes sur les réels. Nous utilisons pour cela la méthode basée sur l'erreur absolue. Comme nous l'avons présenté dans le chapitre 4, cette méthode d'approximation dépend des bornes des variables.

Une fois les contraintes sur les réels obtenues, nous pouvons utiliser un solveur sur les réels pour trouver des solutions potentielles sur les flottants. Pour cela, nous avons opté pour les solveurs SMT. Ce type de solveur peut prouver l'infaisabilité d'un système de contraintes ou trouver une solution dans le cas de faisabilité. Notre choix s'est porté sur le solveur Z3. Ce choix est discuté dans le chapitre 7.

En utilisant un solveur SMT nous pouvons donc obtenir des solutions pour l'approximation sur les réels. En revanche, elles ne sont pas forcément solution des contraintes. Nous présentons dans la section suivante une méthode basée sur des heuristiques pour trouver une solution exacte sur les flottants.

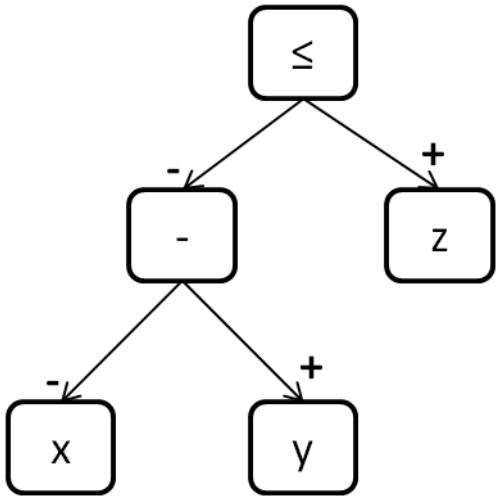


FIGURE 6.1 – Marquage de l'arbre syntaxique de l'expression $x - y \leq z$.

6.4 Recherche de solutions exactes

Bien que les solutions fournies par le solveurs sur les réels soient solutions de l'approximation, elles ne sont pas forcément solutions des contraintes sur les flottants. C'est pour cette raison que nous devons d'abord tester si les solutions obtenues sur les réels sont solutions pour les flottants. Si c'est le cas, les solutions sont alors retournées. Si ce n'est pas le cas, une recherche de solutions sur les flottants au voisinage de solutions sur les réels est lancée.

Le processus du test fait juste une évaluation de chaque contraintes sur les flottants. Le but de cette étape est d'essayer de satisfaire les contraintes violées. Pour chaque contrainte violée, nous passons principalement par les étapes suivantes :

- Détecter les modifications qui peuvent potentiellement satisfaire les contraintes violées.
- Choisir les variables à modifier en premier selon une heuristique donnée.
- Changer les valeurs de ces variables.

Ces étapes sont détaillées par la suite.

6.4.1 Détection des modifications potentielles

Le but de cette étape est de détecter quels sont les changements à effectuer au niveau des valeurs des variables pour lever la violation de la contrainte. Par exemple, si la contrainte $x \geq y$ n'est pas satisfaite. Il est inutile de diminuer la valeur de x ou d'augmenter celle de y, puisque la contrainte restera toujours violée. Les seules modifications qui peuvent potentiellement satisfaire la contrainte sont l'augmentation de la valeur de x ou la diminution de celle de y.

La détection des modifications potentielles est faite en analysant l'expression arithmétique de la contrainte. Nous procédons au marquage des modifications à effectuer dans l'arbre syntaxique de la contrainte. Prenons l'exemple de la contrainte

Opération	Marquage de x	Marquage de y
$x > y$, $x \geq y$	$+$	$-$
$x < y$, $x \leq y$	$-$	$+$
$x == y$	$signe(y - x)$	$signe(x - y)$
$x! = y$	$+, -$	$+, -$

TABLE 6.1 – Marquage des contraintes.

Opération	Marquage de l'opération	Marquage de x	Marquage de y
$x + y$	$+$	$+$	$-$
$x + y$	$-$	$+$	$-$
$x - y$	$+$	$+$	$-$
$x - y$	$-$	$-$	$+$
$x * y$	$+$	$signe(y)$	$signe(x)$
$x * y$	$-$	$-signe(y)$	$-signe(x)$
x/y	$+$	$signe(y)$	$-signe(x)$
x/y	$-$	$-signe(y)$	$signe(x)$

TABLE 6.2 – Marquage des opérations arithmétiques.

$x - y \leq z$, la figure 6.1 représente le marquage de l'arbre syntaxique de cette contrainte. Cette contrainte peut être satisfaite si la partie droite augmente (représentée par le signe $+$) ou la partie gauche diminue (représentée par le signe $-$). Pour la soustraction elle diminue si la partie droite augmente ou la partie gauche diminue. Donc, la contrainte $x - y \leq z$ peut être satisfaite en augmentant les valeurs de y ou z, ou bien en diminuant la valeur de x.

Le marquage de l'arbre syntaxique de la contrainte se fait en se basant sur les règles de marquage dans les tableaux 6.1 et 6.2. Le tableau 6.1 présente les règles de marquage des contraintes. Tandis que le tableau 6.2 présente les règles de marquage des opérations arithmétiques de base. La première colonne représente l'opération. La deuxième : le marquage de cette opération. Les autres colonnes contiennent les signes de marquage à effectuer sur les opérandes.

Une fois les modifications potentielles relevées. Nous devons changer les valeurs des variables qui correspondent à ces modifications jusqu'à trouver une solution sur les flottants.

6.4.2 Heuristique de choix de variables

L'étape précédente nous fournit une liste exhaustive des modifications à effectuer pour satisfaire les contraintes. Pour l'exemple de la contrainte de la contrainte $x - y \leq z$ nous aurons l'ensemble suivant des modifications $\{(x, -), (y, +), (z, +)\}$.

Le choix des variables à modifier en premier est fait selon des heuristiques sur les variables. Pour cela nous avons utilisé les méthodes statiques d'heuristique (STO [1]) [Rice 2008]. Ces méthodes maintiennent le même ordre de variable durant tout le processus de recherche. L'ordre de choix de variable peut se baser sur les critères suivants :

- L'ordre de déclaration.
- La taille des domaines (en terme de nombre de flottants).
- La distance entre la valeur courante et la borne de l'intervalle.
- Le degré des variables, i.e. le nombre de contraintes liées à une variable donnée.

6.4.3 Heuristique de choix de valeurs

Une fois la variable choisie, nous devons changer sa valeur en prenant en compte la direction de modification. La valeur est choisie selon une heuristique. Par exemple, si la valeur courante de la variable flottante x est val et le sens de modification est vers $+\infty$, on peut appliquer l'une des heuristiques suivantes :

- Le flottant suivant, i.e. $x \leftarrow val + ulp(val)$.
- De façon exponentielle, i.e. $x \leftarrow val + 2^n ulp(val)$, avec n le nombre d'itérations.
- De façon dichotomique, i.e. $x \leftarrow \frac{val+\overline{x}}{2}$ si on doit augmenter la valeur.

Ce processus est répété jusqu'à satisfaire l'ensemble des contraintes ou jusqu'à atteindre la borne du domaine. En cas de convergence lente, on peut faire un saut vers l'extrémité du domaine.

6.5 Systèmes de contraintes hybrides (flottants et réels)

La méthode que nous avons présentée offre la possibilité d'utiliser dans le même système à contraintes à la fois des contraintes sur les flottants et des contraintes sur les réels. Après l'étape de sur-approximation des contraintes sur les flottants, nous obtenons un système de contraintes sur les réels. Cela permet de rajouter d'autres contraintes sur les réels à la sur-approximation obtenue. Le tout est alors traité par un système sur les réels.

Cette possibilité peut être bénéfique pour la vérification ou le test des programmes. Ceci permet de comparer le comportement d'un programme sur les flottants avec son comportement en utilisant l'arithmétique sur les réels. Nous pouvons également vérifier des propriétés sur les réels pour des programmes sur les nombres flottants. Dans le chapitre 9 nous donnons quelques exemples de cette possibilité.

6.6 Conclusion

Nous avons présenté dans ce chapitre une nouvelle méthode pour la recherche de solutions sur les nombres flottants. Cette méthode est basée sur l'utilisation des solveur SMT sur les réels. Nous avons également utilisé des heuristiques pour la

1. Static Variable Ordering

recherche de solutions. La méthode que nous avons proposée donne la possibilité de comparer le calcul sur les flottants avec celui sur les réels. Plus de détails sur l'implémentation de cette méthode et les expérimentations effectuées peuvent être trouvés dans la partie suivante.

Troisième partie

Validation expérimentale

Dans cette partie, nous présentons les différentes expérimentations pour valider les approches proposées. Dans le chapitre 7 nous présentons le prototype FPSolveur qui implémente les méthodes que nous avons présentées. Ensuite nous présentons les expérimentations réalisées sur l'étape de filtrage dans le chapitre 8 et les experimentations sur l'étape de recherche de solutions dans le chapitre 9.

Chapitre 7

Implémentation

Sommaire

7.1 Introduction

Nous présentons dans ce chapitre le prototype que nous avons implémenté pour évaluer la pertinence des approches que nous avons proposées. Nous présentons d'abord le prototype, puis les solveurs que nous avons utilisés pour implémenter les méthodes présentées.

7.2 Le prototype FPSolver

FPSolver[1] est un solveur sur les nombres flottants. Ce solveur implémente les méthodes que nous avons présentées dans les chapitres précédents.

L'outil implémente les deux méthodes de sur-approximation des contraintes sur les flottants. Nous avons implémenté les différentes techniques de filtrage présentées dans le chapitre 5. Nous avons interfacé notre solveur avec le solveur FPCS pour les méthodes de filtrage hybrides qui utilisent la 2B-Consistance. L'outil implémente également l'étape de recherche de solutions basée sur les heuristiques.

FPSolveur est implémenté en C++. Ce langage facilite la manipulation des nombres flottants. Le code du solveur fait environ 3000 lignes de code. L'outil a été implémenté dans un environnement linux en utilisant le compilateur *gcc*.

La section suivante discute des choix des solveurs utilisés dans les étapes de filtrage des domaines et de recherche de solutions.

1. Floating-Point Solver

7.3 Choix des solveurs

7.3.1 Solveur linéaire

L'étape de filtrage que nous avons présenté dans le chapitre 5 se base sur l'utilisation d'un solveur linéaire pour la réduction des domaines. Dans le prototype FPSolver nous avons opté pour le solveur linéaire CPLEX [2].

Basé sur l'algorithme du simplexe, CPLEX est très performant dans la résolution de problèmes d'optimisation linéaire et d'optimisation linéaire mixte. Le choix de CPLEX a été aussi motivé par la possibilité de modifier certaines parties du système de contraintes sans que cela nécessite de refaire un nouvel appel au présolve. Cette étape est, généralement, celle qui prend plus de temps dans le processus d'optimisation. Les performances de notre méthode sont ainsi améliorées puisque nous faisons plusieurs appels au solveur linéaire avec de légères modifications dans le système. CPLEX propose aussi les contraintes indicateurs, et nous évite donc l'utilisation de la réécriture "big M" pour la linéarisation de la valeur absolue dans l'approximation basée sur l'erreur relative.

Un des problèmes rencontrés avec CPLEX est qu'il est lui même implémenté sur les flottants. Le processus d'optimisation peut donc inclure des erreurs d'arrondi et fournir des résultats erronés. Pour remédier à ce problème, nous avons utilisé la méthode de correction du simplexe basée sur les travaux de Neumaier et Shcherbina [Neumaier 2004]. Cette méthode permet de calculer un minimum prouvé correct en utilisant l'arithmétique des intervalles et des modes d'arrondi bien choisis.

7.3.2 Solveur SMT

Z3 [3] [De Moura 2008] est un solveur SMT développé à Microsoft Research. Ce solveur cible les problèmes de vérification de programmes. Il a été utilisé dans de nombreux outils de test et de vérification de programmes, comme l'outil Pex [Tillmann 2008].

Z3 est capable de vérifié la faisabilité d'un système de contraintes dans un temps raisonnable. En cas de faisabilité Z3 retourne une instanciation des variables qui satisfait les contraintes. Z3 prend en compte les contraintes linéaires sur les réels et sur les entiers. Z3 prend aussi en compte les contraintes non-linéaires, Bien qu'il ne soit pas très performant là dessus. Dans l'étape de recherche de solutions, nous avons utilisé Z3 pour la recherche de solutions potentielles.

7.4 Modélisation des contraintes

La figure 7.1 illustre un exemple de modélisation de contraintes avec notre solveur. Initialement, nous déclarons le modèle avec l'environnement de travail (type

2. ILOG IBM CPLEX Optimizer : http ://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/index.html
3. http ://z3.codeplex.com/

```
void unstable3() {
  // Déclaration du modèle et des variables
  FPModel fp_model = new FPModel("unstable3");
  Variable a(fp_model, "a", 0, 1);
  Variable b(fp_model, "b", 1, 1e5);

  // Déclaration des contraintes
  fp_model->post(a+b <= 1e5);// Contrainte sur les flottants
  fp_model->post_real(a+b > 1e5); //Contrainte sur les réels

  // Processus de résolution
  // Filtrage basé sur l'approximation avec l'erreur absolue
  fp_model->filter_abs();
  // Recherche de solutions sur les flottants
  fp_model->solve_z3();

  fp_model->display_solutions();
}
```

FIGURE 7.1 – Un exemple de modélisation de contraintes sur les flottants avec FPSolveur.

et mode d'arrondi). Le type par défaut est le type flottant simple précision. Le mode d'arrondi par défaut est au plus proche.

Nous déclarons ensuite les variables et leurs domaines. Les contraintes sont déclarées en utilisant la surcharge des opérateurs. Le langage C++ permet de définir des opérateurs sur les classes définies par l'utilisateur. La fonction *post* permet d'ajouter au modèle des contraintes sur les flottants.

Nous pouvons également ajouter des contraintes interprétées sur réels, grâce à la fonction *post_real*.

Une fois le modèle défini, nous avons la possibilité d'appeler différents types de filtrage. Dans l'exemple de la figure 7.1, nous avons utilisé la méthode basée sur l'approximation avec l'erreur absolue, *filter_abs*(). Nous pouvons aussi rechercher des solutions concrètes sur les flottants utilisant le solveur Z3 et les heuristiques de recherche à l'aide de la fonction *solve_z3*().

Finalement nous pouvons afficher les solutions obtenues grâce à la fonction *display_solutions*(). Nous pouvons également afficher les domaines des variables après l'étape de filtrage.

7.5 Limitations

Le prototype FPSolveur présente quelques limitations d'implémentation. Par exemple, il ne traite que le type flottant simple précision. Le type double précision n'est pas encore pris en compte. Ou encore, le mode d'arrondi des opérations est statique. On ne peut donc avoir différents modes d'arrondi dans le même système

de contraintes.

Le prototype FPSolveur ne prend pas en compte les fonctions transcendantales comme le cosinus, le sinus... La difficulté réside dans l'approximation de ces fonctions de façon conservative.

7.6 Conclusion

Nous avons présenté dans ce chapitre le prototype FPSolveur qui implémente nos contributions. Nous avons également décrit l'architecture de ce solveur, et les outils utilisés dans le processus de résolution de contraintes sur les nombres flottants. Les chapitres suivants expérimentent les différentes méthodes présentées dans ce manuscrit.

Chapitre 8

Expérimentations du filtrage

Sommaire

8.1 Introduction

Dans ce chapitre, nous présentons les premiers résultats de différentes techniques de filtrage de contraintes sur les nombres flottants. Nous avons expérimenté le filtrage sur des exemples académiques afin de valider l'approche proposée. Les expérimentations ont été faites sur un PC portable équipé d'un processeur Intel Duo Core 2.8Ghz, de 4Go de mémoire et dans un environnement Linux.

8.2 Les programmes expérimentés

Cette section décrit les programmes sur lesquels nous avons expérimenté l'étape de filtrage.

8.2.1 Absorption

Le programme 4 est un programme qui détecte le phénomène d'absorption sur une opération d'addition sur les nombres flottants. `Absorb 1` est le chemin qui détecte si x absorbe y lors d'une simple addition ; alors que `Absorb 2` vérifie si y absorbe x.

Le système de contraintes pour `Absorb 1` est le suivant :

$$\begin{cases} x \in [0, 1e10] \\ y \in [100, 1e5] \\ x \oplus y == x \end{cases}$$

Programme 4 Détecter le phénomène d'absorption dans une addition.

```
/*@
 * requires 0 < x <= 1e10
 * requires 100 <= y <= 1e5
 */
void absorb(float & x, float & y) {
  if (x + y == x)
    cout << ("x absorbs y");
  else if (x + y == y)
         cout << ("y absorbs x");
       else cout << ("No significant absorption");
}
```

Pour le deuxième chemin `Absorb 2`, nous avons le système suivant :

$$\begin{cases} x \in [0, 1e10] \\ y \in [100, 1e5] \\ x \oplus y == y \end{cases}$$

8.2.2 Fluctuat

Le programme 5 est extrait d'un article qui présente l'outil Fluctuat [Ghorbal 2010]. Le premier chemin `Fluctuat1` qui vérifie la condition $y \geq 0$ génère les contraintes suivantes.

$$\begin{cases} x \in [0, 10] \\ y = x \otimes x \ominus x \\ y \geq 0 \\ y1 = x \oslash 10 \end{cases}$$

Programme 5 Programme flucuat [Ghorbal 2010].

```
/*@
 * requires 0 < x <= 10
 */
void fluctuat(float x) {
  float y = x * x - x;
  if (y >= 0)
     y = x / 10;
  else
     y = x * x + 2;
    //....;
}
```

`Fluctuat2` est un autre chemin du même programme.

$$\begin{cases} x \in [0, 10] \\ y = x \otimes x \ominus x \\ y < 0 \\ y1 = x \otimes x \oplus 2 \end{cases}$$

8.2.3 Zéro d'une fonction

Le programme 6 est un programme qui retourne vrai si un intervalle donné contient un zéro pour une fonction f. Le système de contraintes généré pour ce programme est le suivant :

$$\begin{cases} x1 \in [2, 10] \\ x2 \in [2, 10] \\ f1 == x1 \otimes x1 \ominus 4 \otimes x1 \ominus 3 \\ f2 == x2 \otimes x2 \ominus 4 \otimes x2 \ominus 3 \\ f1 \geq 0 \\ f2 \leq 0 \end{cases}$$

Programme 6 Tester si une fonction f a un point d'annulation.

```
/*@
 * requires 2 <= x <= 10
 * requires 2 <= y <= 10
 */
boolean MeanValue(float x, float y) {
  float f1 = f(x); // f(x) = x*x - 4*x -3
  float f2 = f(y);
  if ((f1 >= 0) && (f2 <= 0)) return true;
  else return false;
}
```

8.2.4 Cosinus

`Cosine` est un programme qui calcule la fonction $cos()$ avec une formule de Taylor. Le programme 7 qui l'implémente génère les contraintes suivantes :

$$\begin{cases} x \in [0, 3.14/2] \\ x2 == x \otimes x \\ x4 == x2 \otimes x2 \\ x6 == x4 \otimes x2 \\ tmp == 1 \ominus x2 \oslash 2 \oplus x4 \oslash 24 \ominus x6 \oslash 720 \end{cases}$$

Programme 7 Calcul du cosinus en utilisant les développements limités de Taylor.

```
/*@
 * requires 0 <= x <= 3.14/2.0
 */
float Cosine(float x) {
  float x2, x4, x6, tmp;
  x2 = x * x;
  x4 = x2 * x2;
  x6 = x4 * x2;
  tmp =  1 - x2/2.0 + x4/24.0 - x6/720.0;
  return tmp;
}
```

8.2.5 Calcul de la racine carrée

Nous avons expérimenté deux programmes pour le calcul de la racine carrée. Le premier `SqrtV1` calcule la racine carrée d'un nombre dans l'intervalle $[0.5, 2.5]$ en utilisant une méthode itérative à deux variables (voir programme 8). Le deuxième `SqrtV2` calcule la racine carrée en utilisant les séries de Taylor de façon itérative.

En fixant le nombre d'itérations à 5, nous obtenons pour `SqrtV1` le système de contraintes suivant :

$$\begin{cases} x \in [0.5, 1.5] \\ a0 = x \\ c0 = x \ominus 1 \\ a1 = a0 \ominus a0 \otimes c0/2 \\ c1 = c0 \otimes c0 \otimes c0/4 \ominus c0 \otimes c0 \otimes 0.75 \\ a2 = a1 \ominus a1 \otimes c1/2 \\ c2 = c1 \otimes c1 \otimes c1/4 \ominus c1 \otimes c1 \otimes 0.75 \\ a3 = a2 \ominus a2 \otimes c2/2 \\ c3 = c2 \otimes c2 \otimes c2/4 \ominus c2 \otimes c2 \otimes 0.75 \\ a4 = a3 \ominus a3 \otimes c3/2 \\ c4 = c3 \otimes c3 \otimes c3/4 \ominus c3 \otimes c3 \otimes 0.75 \end{cases}$$

8.3 Résultats expérimentaux

Le tableau 8.1 présente les résultats des expérimentations faites avec les méthodes de filtrage suivantes :

- FP2B, une adaptation de l'algorithme de la 2B-consistance aux contraintes sur les flottants. Nous avons utilisé pour cela le solveurs FPCS.
- FP3B, une adaptation de l'algorithme de la 3B-consistance aux contraintes sur les flottants,

Programme 8 Calcul de la racine carrée d'un nombre au voisinage de 1.

```
/*@
 * requires 0.5 <= x <= 1.5
 */
float SqrtV1(float x) {
  int i;
  float an, an1, cn, cn1;
  an1 = x;
  cn1 = x-1;
  while (an1-an > 0.01)
  {
    an = an1;
    cn = cn1;
    an1 = an - an*cn/2;
    cn1 = cn*cn*cn/4 - cn*cn*3/4;
    i=i+1;
  }
  return an1;
}
```

- FPLP(without 2B), une implémentation de l'algorithme 1 sans appel intermédiaire à FP2B, et,
- FPLP, une implémentation de l'algorithme 3.

La première colonne du tableau 8.1 donne le nom du programme, la deuxième, le nombre de variables du problème initial, et la troisième, le nombre de variables intermédiaires utilisées pour décomposer les contraintes initiales complexes en opérations élémentaires. La quatrième colonne donne le nombre de variables binaires utilisées par FPLP. Pour chaque algorithme de filtrage, le tableau 8.1 donne le temps nécessaire au filtrage des contraintes (colonnes $t(ms)$). Pour tous les algorithmes de filtrage – à l'exception de FP2B – le tableau 8.1 donne le pourcentage de réduction obtenu par rapport à celle obtenue en utilisant la FP2B (colonne %(FP2B)). Notez qu'une limite de 2 minutes pour terminer le filtrage est imposée dans ces expérimentations.

Les résultats obtenus dans le tableau 8.1 montrent que FPLP effectue de meilleures réductions des domaines en moins de temps que les méthodes de filtrage locale. Ce phénomène est particulièrement accentué sur les exemples `Absorb1` et `SqrtV1`. Dans ces deux exemples, FP2B souffre du problème des occurrences multiples de variables. FPLP présente de manière consistante de meilleures performances que FP3B : il fournit presque toujours des domaines plus petits et nécessite toujours moins de temps.

Une comparaison entre FPLP sans et avec appel à FP2B montre l'intérêt d'une coopération entre ces deux méthodes de filtrage qui peut réduire considérablement

				FP2B	FP3B		FPLP (sans 2B)		FPLP	
Program	n	n_T	n_B	$t(ms)$	$t(ms)$	%FP2B	$t(ms)$	%FP2B	$t(ms)$	%FP2B
`Absorb1`	2	1	1	TO	TO	-	3	98.91	5	98.91
`Absorb2`	2	1	1	1	24	0.00	3	100.00	4	100.00
`Fluctuat1`	3	12	2	4	156	99.00	264	99.00	172	99.00
`Fluctuat2`	3	10	2	1	4	0.00	29	0.00	21	0.00
`MeanValue`	4	28	6	3	82	97.45	530	97.46	78	97.46
`Cosine`	5	33	7	5	153	33.60	104	33.61	43	33.61
`SqrtV1`	11	140	29	9	27198	99.63	1924	100.00	1187	100.00
`SqrtV2`	21	80	17	7	TO	-	6081	100.00	1321	100.00

TABLE 8.1 – L'expérimentation de l'étape de filtrage.

le temps de calcul sans modifier les capacités de filtrage.

8.4 Conclusion

Nous avons présenté dans ce chapitre l'expérimentation de l'étape du filtrage sur quelques exemples de programmes. Nous avons également montré l'intérêt d'une coopération entre la méthode de filtrage basée sur la sur-approximation et le filtrage basé sur la 2B-consistance.

Chapitre 9

Expérimentations de la recherche de solutions

Sommaire

9.1 Introduction

Ce chapitre est consacré à l'expérimentation de l'étape de recherche de solutions. Nous commençons d'abord par présenter les résultats généraux. Puis nous détaillons chaque exemple à part. Les exemples choisis sont des programmes académiques qui montrent bien la différence entre flottants et réels. Nous avons également montré l'intérêt des contraintes hybrides sur les flottants et les réels.

Les expérimentations ont été faites sur un PC portable équipé d'un processeur Intel Duo Core 2.8Ghz, de 4Go de mémoire et dans un environnement Linux.

9.2 Les résultats des expérimentations

Le tableau 9.1 résume le temps des différents filtrages et de recherche de solutions. La première colonne contient les noms des programmes à vérifier. Nous présentons par la suite les détails de chaque programme. Les autres colonnes montrent -dans l'ordre- le temps d'exécution en millisecondes des méthodes suivante :

- Filtrage avec la consistance 2B pour les flottants en utilisant FPCS. Le pourcentage de réduction de la 2B est de 0.05.
- Filtrage avec la consistance 3B pour les flottants en utilisant FPCS. Le pourcentage de réduction de la 3B est de 0.05.

Programme	2B	3B	App	2B+App	search
`Permutation`	1	3	3	1	1
`Beal`	1	3	1	1	7
`Absorption`	1	2	1	1	1
`Patriot`	1	11	3	4	10
`Banque chaotique`	1	43	5	3	15

TABLE 9.1 – L'expérimentation de l'étape de recherche de solutions.

– Filtrage avec la méthode de sur-approximation en utilisant la méthode de l'erreur absolue.
– La combinaison de la 2B avec la sur-approximation avec l'erreur absolue.
– La recherche de solutions exactes sur les flottants.

Noter que le filtrage basé sur les consistances locales n'est appliqué qu'aux contraintes sur les flottants. Nous ne pouvons pas construire des systèmes de contraintes hybrides avec des méthodes.

Pour l'étape de recherche de solutions, nous avons utilisé le solveurs Z3 pour localiser une solution potentielle. On utilise l'heuristique de changement de valeurs est de façon exponentielle (voir chapitre 6). Quant au tri des variables, il correspond à l'ordre de déclaration.

Nous détaillons par la suite chaque exemple à part.

9.3 Les programmes expérimentés

Les exemples que nous avons choisis sont des programmes qui montrent bien la différence entre le comportement des nombres flottants et les nombres réels.

Nous détaillons ici chaque exemple à part. Nous présentons pour chaque programme :

– Le code avec la propriété à vérifier
– Le système de contraintes généré
– Les résultats obtenus pour le filtrage et la recherche de solutions.

9.3.1 Permutation

Le programme 9 essaye de faire une permutation entre deux nombres flottants sans variable intermédiaire. La propriété à vérifier est que la permutation est bien faite à la fin du programme, i.e le résultat réel est égal au résultat flottant. À cause de l'arrondi, cette propriété n'est pas vérifiée.

Les contraintes générées pour ce programmes sont :

$$\begin{cases} x & = x0 \oplus y0 \\ y & = x1 \ominus y0 \\ x2 & = x1 \ominus y0 \\ x2 & \neq y0 \end{cases}$$

Avec les méthodes de filtrage existantes, nous n'avons pas pu prouver que le programme ne fait pas exactement une permutation. Par contre, avec notre outil nous avons réussi à prouver (avec un contre-exemple) que ce programme ne vérifie pas la post-condition.

Programme 9 Permutation de variables utilisant seulement deux variables.

```
/*@
 * ensure x2 == y0
 */
void permutation(float & x, float & y) {
  x = x + y; // x1 = x0 + y0;
  y = x - y; // y1 = x1 - y0;
  x = x - y; // x2 = x1 - y1;
}
```

Le contre-exemple fourni par notre outil FPSolveur est le suivant :

$$\begin{cases} x0 & == 3.5527136788005009293556213378906 25e-15 \\ y0 & == 1.4012984643248170709237295832899 16..e-45 \\ x1 & == 3.5527136788005009293556213378906 25e-15 \\ y1 & == 3.5527136788005009293556213378906 25e-15 \\ x2 & == 0 \end{cases}$$

Cet contre-exemple montre bien que le programme précédent peut ne pas faire de permutations correctes.

9.3.2 Absorption

Dans le programme 10, nous essayons d'assurer qu'après un test sur les flottants de $x \oplus y <= y$, la résultat sur les réels de $x + y$ est aussi inférieur à y.

Le but d'une telle vérification est de comparer le comportement du programme avec l'arithmétique des réels.

Programme 10 Convergence s'une série mathématique.

```
/*@
 * requires 0 <= y <= 1000
 */
bool absorption(float x, float y) {
  if (x + y <= y)
    cout << "x + y <= y";
    // assert Real(x + y) <= y
}
```

Nous obtenons les contraintes suivantes pour le programme 10.

$$\begin{cases} x \oplus y <= & y \\ x + y > & y \end{cases}$$

Le prototype FPSolveur nous fournit le contre-exemple suivant :

$$\begin{cases} x & == 1.1920928955078125e-07 \\ y & == 2.50000095367431640625 \end{cases}$$

Cette solution prouve qu'on peut avoir $x + y <= y$ sur les flottants alors que nous l'avons pas sur les réels.

9.3.3 Beal

Cet exemple provient de [Lakhotia 2010] (voir programme 11). Dans cet exemple, on essaye de prouver que l'équation de Beal peut être satisfaite sur les flottants et pas sur les réels.

Programme 11 Calcul de l'équation Beal [Lakhotia 2010].

```
/*@
 * ensure \result => Real(x1 * (x2 - 2) - 1.5) == 0
 */
bool beal(float x1, float x2) {
  float t1 = x2 - 2;
  float t2 = x1 * t1;
  float z = t2 - 1.5;
  return z == 0;
}
```

Les contraintes générées pour le programme 11 :

$$\begin{cases} (x1 \otimes (x2 \ominus 2) \ominus 1.5) & == 0 \\ (x1 \times (x2 - 2) - 1.5) & \neq 0 \end{cases}$$

Pour la partie sur les flottants, nous obtenons le même résultat de filtrage avec la 3B et la méthode d'approximation.

$$x1 \in [1.5, 2] \text{ et } x2 \in [2.75, 3]$$

Les heuristiques de recherche de solutions nous fournissent le contre-exemple suivant :

$$\begin{cases} x1 == & 1.74999988079071044921875 \\ x2 == & 2.857142925262451171875 \end{cases}$$

Nous avons donc vérifié qu'il existe des cas pour lesquels l'équation de Beal est satisfaite sur les flottants et pas sur les réels.

9.3.4 Patriot

Le programme 12 est une simple incrémentation de la variable x par un pas de type flottant. La post-condition vérifie si cette incrémentation a été bien faite, i.e. la somme des pas sur les flottants est égale exactement à $N \times pas$.

Ce programme a été source de problème dans l'antimissile Patriot lorsqu'on incrémentait le temps avec une unité de 0.1.

Programme 12 Incrémentation d'une variable.

```
#define N 15
/*@
 * requires account > 0
 * ensure \result == N * pas
 */
float patriot(float pas)
{
    int i;
    float x = pas;
    for (i = 0; i < N; i++)
    {
        x = x + pas;
    }
    return x;
}
```

Les contraintes générées pour ce programme :

$$\begin{cases} x0 & = pas \\ x1 & = x0 \oplus pas \\ & \dots \\ (x14 & = x13 \oplus pas) \\ (x14 & \neq pas \times 15) \end{cases}$$

Notre outil fournit le contre-exemple suivant :

$$\begin{cases} cst == 1.0000001192092895507812 5 \\ \dots \\ x14 == 15.000000953674316406 25 \end{cases}$$

9.3.5 Banque chaotique

Le programme 13 implémente une procédure de calcul des intérêts d'une banque. Avec un compte initial *account*. Chaque année on multiplie le compte courant par le nombre d'année, et on ne prend qu'un euro de charge. Le programme provient du livre Handbook of floating-point numbers [Muller 2010].

Pour certaines valeurs, ce programme fournit des résultats sur les flottants qui ont une grande différence avec le résultat avec l'arithmétique des réels. Ces valeurs sont autour de $e - 1$, avec e la base du logarithme népérien.

La post-condition vérifie qu'après 15 ans, la différence entre le résultat flottant et réel ne dépasse pas 10 euros.

Programme 13 Calcul des intérêts.

```
#define N 15
/*@
 * requires account > 0
 * ensure \result - Real(\result) < 10
 */
float chaotic_bank(float account)
{
    int i;
    float x = account;
    for (i = 0; i < N; i++)
    {
        x = x * i - 1;
    }
    return x;
}
```

Pour 15 itération de la boucle du programme 13, nous obtenons le système de contraintes suivant :

$$\begin{cases} x0 = account \ominus 1 \\ r0 = account - 1 \\ x1 = x0 \otimes 2 \ominus 1 \\ r1 = r0 \times 2 - 1 \\ x2 = x1 \otimes 3 \ominus 1 \\ r2 = r1 \times 3 - 1 \\ ... \\ x14 = x13 \otimes 15 - 1 \\ r14 = r13 \times 15 - 1 \\ x14 - r14 > 10 \end{cases}$$

Nous obtenons avec notre solveur la solution suivante :

$$\begin{cases} account == 1.71828186511993408203125 \\ ... \\ x14 == 47940.57421875 \\ r14 == 0 \end{cases}$$

La solution obtenue montre bien que la différence entre le résultat réel et le résultat flottant peut dépasser 10. Dans notre exemple nous avons obtenu un contre-

exemple qui retourne 47940.57... dans le programme, tandis que le résultat avec l'arithmétique des nombres réels converge vers 0.

9.4 Conclusion

Nous avons présenté dans ce chapitre quelques exemples d'applications de notre méthode à la vérification de programmes. Les exemples choisis essayent de comparer le comportement des programmes sur les flottants par rapport à une spécification sur les réels. Dans les exemples présentés, nous avons pu trouver un contre-exemple.

Conclusion

Bilan

Le but principal de cette thèse était la conception et la réalisation d'un solveur de contraintes sur les flottants dédié à la vérification et au test de programmes. Durant nos travaux de thèse, nous avons proposé différentes contributions dans le domaine de la résolution des contraintes sur les flottants.

Nous avons tout d'abord défini une nouvelle méthode de résolution d'un système de contraintes sur les nombres flottants. Cette méthode consiste à approximer les contraintes sur les flottants par des contraintes sur les réels. Nous avons proposé différents types d'approximations. Ces approximations sont conservatives des solutions sur les flottants. L'objectif initial était de pouvoir utiliser n'importe quel solveur sur les réels pour résoudre ces approximations.

Nous avons introduit différentes méthodes pour sur-approximer les contraintes sur les flottants. La première méthode que nous avons proposée est basée sur l'erreur relative d'arrondi. Cette méthode soulève un problème de complexité. Pour remédier à ce problème, nous avons utilisé des techniques de linéarisations de contraintes. La seconde méthode d'approximation est basée sur l'erreur absolue d'arrondi. Cette approximation dépend des bornes initiales des variables et peut être très grossière.

En se basant sur ces sur-approximations, nous avons conçu une nouvelle méthode de filtrage des contraintes sur les flottants. Les techniques existantes de filtrage se basent sur les consistances locales adaptées aux flottants. Ces techniques souffrent du problème des occurrences multiples de variables. La méthode que nous avons proposée est une méthode de filtrage plus globale. Elle repose sur des techniques issues de la programmation linéaire et utilise des linéarisations de contraintes sur les flottants.

Nous avons expérimenté notre proposition sur un ensemble de programmes sur les flottants. Les premiers résultats montrent l'apport de notre méthode dans l'étape de filtrage des domaines à la fois en termes de précision et de performances. Nous avons également montré qu'une coopération avec les techniques de filtrage local peut encore améliorer les performances.

Une autre contribution de notre thèse réside dans l'étape de recherche de solutions sur les nombres flottants. Cette étape est primordiale pour les outils de test et de vérification basée sur la programmation par contraintes. Les méthodes existantes de recherche de solutions sur les flottants se basent sur les approches classiques de la programmation par contraintes. Ces approches consistent à utiliser le filtrage et la propagation pour trouver des solutions concrètes. Les méthodes de résolution de contraintes basées sur ces approches peinent à passer à l'échelle.

La méthode que nous avons proposée repose sur l'utilisation de solveurs sur les réels pour identifier des solutions aux sur-approximations des contraintes sur les flottants. Les solutions obtenues ne sont pas forcément des solutions pour le système

de contraintes initial sur les flottants. Nous avons donc proposé des heuristiques de recherche de solutions au voisinage des solutions obtenues par le solveur sur les réels. Ces heuristiques sont dédiées aux nombres flottants.

Nous avons également proposé une nouvelle méthode de vérification de programmes sur les flottants. Cette méthode consiste à comparer le comportement du programme qui utilise l'arithmétique des flottants avec l'arithmétique des réels. Cela est fait en utilisant des systèmes de contraintes hybrides qui peuvent traiter à la fois des contraintes sur les flottants et des contraintes sur les réels. L'hybridation de ces deux type de contraintes donne la possibilité de s'assurer que des propriétés sur les réels sont vérifiées par un programme sur les flottants.

Cette méthode a été expérimentée sur des programmes qui montrent bien la différence entre les programmes sur les flottants et leurs interprétations sur les réels. Nous avons réussi à générer des contre-exemples pour ces programmes dans un temps raisonnable.

Les différentes contributions ont été implémentées dans un nouveau prototype que nous avons nommé FPSolveur. Les différentes expérimentations ont été faites avec ce prototype.

Perspectives

Les méthodes que nous avons proposées ouvrent de nouvelles perspectives et sujets de recherche. Ces perspectives tournent autour de la résolution de contraintes sur les flottants et de la vérification de programmes avec du calcul sur les flottants.

Approximation des contraintes sur les flottants

Les méthodes d'approximation présentées dans ce manuscrit se limitent aux opérations arithmétiques de base. Les fonctions transcendantes ne sont pas prises en compte. Or les programmes avec du calcul sur les flottants peuvent contenir de nombreuses fonctions transcendantes (cos, sin, exp...).

Les différentes méthodes d'approximations peuvent être utilisées conjointement pour améliorer les performances du solveur. Une des méthodes consiste à identifier le meilleur moyen de sur-approximer une opération donnée. Nous obtenons ainsi un système de contraintes qui peut contenir les deux types de sur-approximations.

Filtrage

Les approximations que nous avons proposées peuvent être utilisées avec n'importe quel type de solveur correctement implémenté sur les réels. Jusqu'à présent, la méthode filtrage de notre solveur repose sur les solveurs linéaires. Nous pouvons expérimenter d'autres types de solveur sur les réels.

Recherche de solutions

La méthode de recherche de solutions que nous avons présentée repose sur des heuristiques pour trouver des solutions exactes sur les flottants. L'étape de recherche de solutions peut être améliorée en explorant d'autres heuristiques qui tiennent compte des particularités des nombres flottants et de la nature des contraintes.

Une autre amélioration de l'étape de recherche des solutions peut être faite en coopération avec la méthode de filtrage. Nous pouvons par exemple instancier partiellement les variables puis relancer le filtrage afin de réduire l'espace de recherche.

Une des limitations de notre implémentation est que le solveur Z3 n'est pas très efficace sur les contraintes non-linéaires. Nous envisageons donc une méthode de linéarisation de ces contraintes pour gagner en performance.

Vérification de programmes

Le solveur que nous avons implémenté n'est pas encore interfacé avec les outils de test et de vérification de programmes. Pour faciliter la vérification de programmes, il serait bénéfique d'interfacer le solveur que nous avons implémenté avec quelques outils de vérification et de test basés sur la programmation par contraintes.

Nous pouvons également utiliser les systèmes de contraintes hybrides entre flottants et réels pour générer des cas de test donnant des résultats différents entre flottants et réels.

Une des pistes d'application de la résolution de contraintes sur les nombres flottants est la détection des erreurs d'exécutions comme la division par zéro. Cela peut être fait en cherchant les zéros pour l'un des dénominateurs dans le programme. Ainsi, nous pouvons générer des cas de test qui provoque des erreurs d'exécution.

Expérimentations

Les expérimentations que nous avons réalisées se limitent à des programmes académiques. Afin de valider davantage les approches proposées, elles doivent être expérimentées sur des programmes concrets venant du monde de l'industrie. De nombreux domaines critiques utilisent le calcul sur les nombres flottant. Il est essentiel de trouver des programmes réels afin tester nos approches.

Table des figures

Liste des tableaux

List of Algorithms

Bibliographie

[Al-Khayyal 1983] F.A. Al-Khayyal et J.E. Falk. *Jointly Constrained Biconvex Programming*. Mathematics of Operations Research, pages 8 :2 :273–286, 1983. (Cité en page 60.)

[Albert 2012] Elvira Albert, Puri Arenas et Miguel Gómez-Zamalloa. *Towards Testing Concurrent Objects in CLP*. In Agostino Dovier et Vítor Santos Costa, editeurs, ICLP (Technical Communications), volume 17 of *LIPIcs*, pages 98–108. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012. (Cité en page 30.)

[Bagnara 2013a] Roberto Bagnara, Matthieu Carlier, Roberta Gori et Arnaud Gotlieb. *Filtering floating-point constraints by maximum ULP*. arXiv preprint arXiv :1308.3847, 2013. (Cité en pages 34 et 35.)

[Bagnara 2013b] Roberto Bagnara, Matthieu Carlier, Roberta Gori, Arnaud Gotlieb*et al.* *Symbolic Path-Oriented Test Data Generation for Floating-Point Programs*. In Proc. of the 6th IEEE Int. Conf. on Software Testing, Verification and Validation (ICST'13), 2013. (Cité en pages 34 et 35.)

[Bardin 2009] Sebastian Bardin, Bernard Botella, Frédéric Dadeau, Florence Charreteur, Arnaud Gotlieb, Bruno Marre, Claude Michel, Michel Rueher et Nicky Williams. *Constraint-based software testing*. Journée du GDR-GPL, vol. 9, 2009. (Cité en page 1.)

[Barrett 1989] Geoff Barrett. *Formal methods applied to a floating-point number system*. Software Engineering, IEEE Transactions on, vol. 15, no. 5, pages 611–621, 1989. (Cité en page 25.)

[Belaid 2010a] Mohammed Said Belaid, Claude Michel et Michel Rueher. *Approximating floating-point operations to verify numerical programs*. In 14th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN'10), 2010. (Cité en page 2.)

[Belaid 2010b] Mohammed Said Belaid, Claude Michel, Michel Rueher*et al.* *Résolution de contraintes sur les nombres à virgule flottante par une approximation sur les nombres réels*. In JFPC 2010-Sixièmes Journées Francophones de Programmation par Contraintes, pages 51–60, 2010. (Cité en pages 2 et 52.)

[Belaid 2012a] Mohammed Said Belaid, Claude Michel et Michel Rueher. *Boosting local consistency algorithms over floating-point numbers*. In Principles and Practice of Constraint Programming, pages 127–140. Springer, 2012. (Cité en pages 2, 57 et 64.)

[Belaid 2012b] Mohammed Said Belaid, Claude Michel, Michel Rueher*et al.* *Un nouvel algorithme de consistance locale sur les nombres flottants*. In Huitièmes Journées Francophones de Programmation par Contraintes-JFPC 2012, 2012. (Cité en pages 2, 57 et 64.)

[Benhamon 1994] F Benhamon, D McAllester et P Van Hentenryck. *CLP (Intervals) revisited*. Rapport technique, Citeseer, 1994. (Cité en page 30.)

[Benhamou 1994] Frédéric Benhamou, David A. McAllester et Pascal Van Hentenryck. *CLP(Intervals) Revisited*. In SLP, pages 124–138, 1994. (Cité en page 29.)

[Blanc 2006] Benjamin Blanc, Fabrice Bouquet, Arnaud Gotlieb, Bertrand Jeannet, Thierry Jéron, Bruno Legeard, Bruno Marre, Claude Michel et Michel Rueher. *The V3F project*. CSTVA'06, 2006. (Cité en page 35.)

[Boldo 2006] Sylvie Boldo. *Pitfalls of a full floating-point proof : example on the formal proof of the Veltkamp/Dekker algorithms*. In Automated Reasoning, pages 52–66. Springer, 2006. (Cité en page 25.)

[Boldo 2009] Sylvie Boldo, Jean-Christophe Filliâtre et Guillaume Melquiond. *Combining Coq and Gappa for certifying floating-point programs*. In Intelligent Computer Mathematics, pages 59–74. Springer, 2009. (Cité en page 25.)

[Botella 2005] B Botella et A Gotlieb. *FPSE : Floating-Point Symbolic Execution*. INRIA/IRISA, Rennes, 2005. (Cité en page 35.)

[Botella 2006] Bernard Botella, Arnaud Gotlieb et Claude Michel. *Symbolic execution of floating-point computations*. Softw. Test., Verif. Reliab., vol. 16, no. 2, pages 97–121, 2006. (Cité en pages 1, 31, 32, 33 et 36.)

[Bouquet 2004] Fabrice Bouquet, Bruno Legeard et Fabien Peureux. *CLPS-B - A constraint solver to animate a B specification*. STTT, vol. 6, no. 2, pages 143–157, 2004. (Cité en page 31.)

[Brillout 2009] Angelo Brillout, Daniel Kroening et Thomas Wahl. *Mixed abstractions for floating-point arithmetic*. In Formal Methods in Computer-Aided Design, 2009. FMCAD 2009, pages 69–76. IEEE, 2009. (Cité en page 34.)

[Brunet 1986] Marie-Christine Brunet et Françoise Chatelin. Cestac, a tool for a stochastic round-off error analysis in scientific computing. IBM France Scientific Center, 1986. (Cité en page 20.)

[Carlier 2011] Matthieu Carlier et Arnaud Gotlieb. *Filtering by ULP maximum*. In Tools with Artificial Intelligence (ICTAI), 2011 23rd IEEE International Conference on, pages 209–214. IEEE, 2011. (Cité en pages 34 et 35.)

[Carreño 1995] Victor A Carreño et Paul S Miner. *Specification of the IEEE-854 floating-point standard in HOL and PVS*. High Order Logic Theorem Proving and Its Applications, 1995. (Cité en page 25.)

[Charreteur 2010] Florence Charreteur et Arnaud Gotlieb. *Constraint-Based Test Input Generation for Java Bytecode*. In ISSRE, pages 131–140. IEEE Computer Society, 2010. (Cité en page 30.)

[Collavizza 1999] Hélene Collavizza, François Delobel et Michel Rueher. *Comparing partial consistencies*. Reliable computing, vol. 5, no. 3, pages 213–228, 1999. (Cité en page 30.)

[Collavizza 2006] Hélene Collavizza et Michel Rueher. *Exploration of the capabilities of constraint programming for software verification.* In Tools and Algorithms for the Construction and Analysis of Systems, pages 182–196. Springer, 2006. (Cité en page 1.)

[Collavizza 2010] Hélène Collavizza, Michel Rueher et Pascal Hentenryck. *CPBPV : a constraint-programming framework for bounded program verification.* Constraints, vol. 15, no. 2, pages 238–264, 2010. (Cité en pages 1 et 31.)

[Collavizza 2011] Hélène Collavizza, Nguyen Le Vinh, Michel Rueher, Samuel Devulder et Thierry Gueguen. *A dynamic constraint-based bmc strategy for generating counterexamples.* In Proceedings of the 2011 ACM Symposium on Applied Computing, pages 1633–1638. ACM, 2011. (Cité en page 31.)

[Colmerauer 1985] Alain Colmerauer. *Prolog in 10 Figures.* Commun. ACM, vol. 28, no. 12, pages 1296–1310, 1985. (Cité en page 28.)

[Colmerauer 1990] Alain Colmerauer. *An Introduction to Prolog III.* Commun. ACM, vol. 33, no. 7, pages 69–90, 1990. (Cité en page 28.)

[Cousot 2005] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux et X. Rival. *The Astrée analyzer.* In Proc. of the European Symposium on Programming (ESOP'05), volume 3444 of *Lecture Notes in Computer Science (LNCS)*, pages 21–30. Springer, Apr. 2005. (Cité en page 23.)

[Daumas 2001] Marc Daumas, Laurence Rideau et Laurent Théry. *A generic library for floating-point numbers and its application to exact computing.* In Theorem Proving in Higher Order Logics, pages 169–184. Springer, 2001. (Cité en page 25.)

[De Moura 2008] Leonardo De Moura et Nikolaj Bjørner. *Z3 : An efficient SMT solver.* In Tools and Algorithms for the Construction and Analysis of Systems, pages 337–340. Springer, 2008. (Cité en page 76.)

[Delmas 2009] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal et Franck Védrine. *Towards an industrial use of FLUCTUAT on safety-critical avionics software.* In Formal Methods for Industrial Critical Systems, pages 53–69. Springer, 2009. (Cité en page 21.)

[DeMilli 1991] RA DeMilli et A. Jefferson Offutt. *Constraint-based automatic test data generation.* Software Engineering, IEEE Transactions on, vol. 17, no. 9, pages 900–910, 1991. (Cité en page 30.)

[Denmat 2007] Tristan Denmat, Arnaud Gotlieb et Mireille Ducassé. *Improving Constraint-Based Testing with Dynamic Linear Relaxations.* In ISSRE, pages 181–190. IEEE Computer Society, 2007. (Cité en page 30.)

[D'Silva 2008] Vijay D'Silva, Daniel Kroening et Georg Weissenbacher. *A survey of automated techniques for formal software verification.* Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 27, no. 7, pages 1165–1178, 2008. (Cité en page 1.)

[D'Silva 2012] Vijay D'Silva, Leopold Haller, Daniel Kroening et Michael Tautschnig. *Numeric bounds analysis with conflict-driven learning.* In Tools and Algorithms for the Construction and Analysis of Systems, pages 48–63. Springer, 2012. (Cité en pages 24 et 34.)

[Edvardsson 1999] Jon Edvardsson. *A survey on automatic test data generation.* In Proceedings of the 2nd Conference on Computer Science and Engineering, pages 21–28, 1999. (Cité en page 30.)

[Fousse 2007] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier et Paul Zimmermann. *MPFR : A multiple-precision binary floating-point library with correct rounding.* ACM Transactions on Mathematical Software (TOMS), vol. 33, no. 2, page 13, 2007. (Cité en page 21.)

[Frédéric Dadeau 2011] Frédéric Dadeau, Fabien Peureux, Bruno Legeard, Régis Tissot, Jacques Julliand, Pierre-Alain Masson et Fabrice Bouquet. *Test Generation Using Symbolic Animation of Models.* In Justyna Zander, Ina Schieferdecker et Pieter J. Mosterman, editeurs, Model-based testing for embedded systems, Computational analysis, synthesis, and design of dynamic systems. CRC Press, Boca Raton, 2011. (Cité en page 31.)

[Ghorbal 2010] K. Ghorbal, E. Goubault et S. Putot. *A logical product approach to zonotope intersection.* In Computer Aided Verification, pages 212–226. Springer, 2010. (Cité en pages 21 et 80.)

[Goldberg 1991] David Goldberg. *What Every Computer Scientist Should Know About Floating-Point Arithmetic.* ACM Computing Surveys, vol. 23, no. 1, pages 5–48, 1991. (Cité en pages 8, 9 et 14.)

[Gotlieb 1998] Arnaud Gotlieb, Bernard Botella et Michel Rueher. *Automatic Test Data Generation Using Constraint Solving Techniques.* In ISSTA, pages 53–62, 1998. (Cité en page 30.)

[Gotlieb 2000] Arnaud Gotlieb, Bernard Botella et Michel Rueher. *A CLP Framework for Computing Structural Test Data.* In Computational Logic, pages 399–413, 2000. (Cité en page 30.)

[Gotlieb 2009] Arnaud Gotlieb. *Euclide : A constraint-based testing framework for critical c programs.* In Software Testing Verification and Validation, 2009. ICST'09. International Conference on, pages 151–160. IEEE, 2009. (Cité en page 1.)

[Gotlieb 2012] Arnaud Gotlieb. *TCAS software verification using constraint programming.* Knowledge Eng. Review, vol. 27, no. 3, pages 343–360, 2012. (Cité en page 30.)

[Goubault 2002] Eric Goubault, Matthieu Martel et Sylvie Putot. *Asserting the precision of floating-point computations : a simple abstract interpreter.* In Programming Languages and Systems, pages 209–212. Springer, 2002. (Cité en pages 20 et 21.)

[Goubault 2005] Eric Goubault et Sylvie Putot. *Weakly relational domains for floating-point computation analysis*. In First International Workshop on Numerical and Symbolic Abstract Domains. Citeseer, 2005. (Cité en page 20.)

[Goubault 2006] Eric Goubault et Sylvie Putot. *Static analysis of numerical algorithms*. In Static Analysis, pages 18–34. Springer, 2006. (Cité en pages 1 et 20.)

[Goubault 2009] Eric Goubault et Sylvie Putot. *A zonotopic framework for functional abstractions*. arXiv preprint arXiv :0910.1763, 2009. (Cité en page 21.)

[Goubault 2012] Eric Goubault, Sylvie Putot et Franck Védrine. *Modular static analysis with zonotopes*. In Static Analysis, pages 24–40. Springer, 2012. (Cité en page 21.)

[Harrison 1999] John Harrison. *A machine-checked theory of floating point arithmetic*. In Theorem Proving in Higher Order Logics, pages 113–130. Springer, 1999. (Cité en page 25.)

[Harrison 2000] John Harrison. *Formal verification of floating point trigonometric functions*. In Formal Methods in Computer-Aided Design, pages 254–270. Springer, 2000. (Cité en pages 20 et 25.)

[IEEE 1985] IEEE. *IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Std 754-1985, 1985. (Cité en page 9.)

[IEEE 1987] IEEE. *ANSI/IEEE Std 854-1987 : IEEE Standard for Radix-Independent Floating-Point Arithmetic*, 1987. (Cité en page 9.)

[IEEE 2008] IEEE. Ieee 754-2008, standard for floating-point arithmetic. IEEE, New York, NY, USA, 2008. (Cité en pages 9 et 11.)

[Jaffar 1987] Joxan Jaffar et Jean-Louis Lassez. *Constraint Logic Programming*. In 14th Annual ACM Symposium on Principles of Programming Languages, pages 111–119, 1987. (Cité en page 28.)

[Jézéquel 2008] Fabienne Jézéquel et Jean-Marie Chesneaux. *CADNA : a library for estimating round-off error propagation*. Computer Physics Communications, vol. 178, no. 12, pages 933–955, 2008. (Cité en page 20.)

[Kahan 1996] William Kahan. *The improbability of probabilistic error analyses for numerical computations*. In UCB Statistics Colloquium, evans hall edition, 1996. (Cité en page 20.)

[Kaivola 2003] Roope Kaivola et Katherine Kohatsu. *Proof engineering in the large : formal verification of Pentium® 4 floating-point divider*. International Journal on Software Tools for Technology Transfer, vol. 4, no. 3, pages 323–334, 2003. (Cité en page 25.)

[Kästner 2010] D. Kästner, S. Wilhelm, S. Nenova, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné et X. Rival. *Astrée : Proving the absence of runtime errors*. In Proc. of Embedded Real Time Software and Systems (ERTS2 2010), page 9, May 2010. (Cité en page 23.)

[Lakhotia 2010] Kiran Lakhotia, Nikolai Tillmann, Mark Harman et Jonathan De Halleux. *Flopsy-search-based floating point constraint solving for symbolic execution*. In Testing Software and Systems, pages 142–157. Springer, 2010. (Cité en pages 34 et 88.)

[Lebbah 2005] Yahia Lebbah, Claude Michel, Michel Rueher, David Daney et Jean-Pierre Merlet. *Efficient and Safe Global Constraints for Handling Numerical Constraint Systems*. SIAM J. Numer. Anal, vol. 42, pages 2076–2097, 2005. (Cité en pages 29, 57 et 60.)

[Lebbah 2007] Yahia Lebbah, Claude Michel et Michel Rueher. *An efficient and safe framework for solving optimization problems*. J. Comput. Appl. Math., vol. 199, pages 372–377, February 2007. (Cité en pages 29 et 57.)

[Lhomme 1993] Olivier Lhomme. *Consistency techniques for numeric CSPs*. In IJCAI, volume 93, pages 232–238. Citeseer, 1993. (Cité en pages 29 et 36.)

[Lions 1996] Jacques-Louis Lions*et al.* *Ariane 5 flight 501 failure*, 1996. (Cité en page 1.)

[Mackworth 1977] Alan K. Mackworth. *Consistency in networks of relations*. Artificial Intelligence, vol. 8, no. 1, pages 99–118, 1977. (Cité en page 28.)

[Marre 2000] Bruno Marre et Agnes Arnould. *Test Sequences Generation from LUSTRE Descriptions : GATEL*. In ASE : Proceedings of the 15th IEEE international conference on Automated software engineering, page 229, Washington and DC and USA, 2000. IEEE Computer Society. (Cité en page 30.)

[Marre 2010] Bruno Marre et Claude Michel. *Improving the floating point addition and subtraction constraints*. In Principles and Practice of Constraint Programming–CP 2010, pages 360–367. Springer, 2010. (Cité en pages 33 et 34.)

[McCormick 1976] G.P. McCormick. *Computability of global solutions to factorable nonconvex programs – Part I – Convex underestimating problems*. Mathematical Programming, vol. 10, pages 147–175, 1976. (Cité en pages 58, 60 et 61.)

[Michel 2001] Claude Michel, Michel Rueher et Yahia Lebbah. *Solving Constraints over Floating-Point Numbers*. In CP, volume 2239 of *Lecture Notes in Computer Science*, pages 524–538. Springer, 2001. (Cité en pages 1 et 32.)

[Michel 2002] Claude Michel. *Exact Projection Functions for Floating Point Number Constraints*. In AMAI, 2002. (Cité en pages 1, 32 et 33.)

[Miné 2004a] A. Miné. *Relational abstract domains for the detection of floating-point run-time errors*. In Proc. of the European Symposium on Programming (ESOP'04), volume 2986 of *Lecture Notes in Computer Science (LNCS)*, pages 3–17. Springer, Mar. 2004. (Cité en pages 1 et 23.)

[Miné 2004b] Atoine Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Palaiseau, France, December 2004. (Cité en page 23.)

[Miné 2006] A. Miné. *The octagon abstract domain.* Higher-Order and Symbolic Computation (HOSC), vol. 19, no. 1, pages 31–100, 2006. http://www.di.ens.fr/~mine/publi/article-mine-HOSC06.pdf. (Cité en page 23.)

[Monniaux 2008] David Monniaux. *The pitfalls of verifying floating-point computations.* ACM Trans. Program. Lang. Syst., vol. 30, no. 3, pages 1–41, 2008. (Cité en page 15.)

[Moore 1966] Ramon E Moore. Interval analysis, volume 2. Prentice-Hall Englewood Cliffs, 1966. (Cité en page 29.)

[Muller 2005] Jean-Michel Muller. *On the definition of ulp.* 2005. (Cité en page 9.)

[Muller 2010] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé et Serge Torres. Handbook of floating-point arithmetic. Birkhäuser Boston, 2010. ISBN 978-0-8176-4704-9. (Cité en pages 16 et 89.)

[Neumaier 2004] Arnold Neumaier et Oleg Shcherbina. *Safe bounds in linear and mixed-integer linear programming.* Mathematical Programming, vol. 99, no. 2, pages 283–296, 2004. (Cité en page 76.)

[Office 1992] US General Accounting Office. *Patriot Missile Defense : Software Problem Led to System Failure at Dhahran, Saudi Arabia.* 1992. (Cité en page 15.)

[Older 1993] William J. Older et Frédéric Benhamou. *Programming in CLP(BNR).* In PPCP, pages 228–238, 1993. (Cité en page 29.)

[O'Leary 1999] John O'Leary, Xudong Zhao, Rob Gerth et Carl-Johan H Seger. *Formally verifying IEEE compliance of floating-point hardware.* Intel Technology Journal, vol. 3, no. 1, pages 1–14, 1999. (Cité en pages 20 et 25.)

[Pelleau 2013] Marie Pelleau, Antoine Miné, Charlotte Truchet et Frédéric Benhamou. *A constraint solver based on abstract domains.* In Verification, Model Checking, and Abstract Interpretation, pages 434–454. Springer, 2013. (Cité en page 25.)

[Ponsini 2011] Olivier Ponsini, Claude Michel et Michel Rueher. *Refining abstract interpretation-based approximations with a floating-point constraint solver.* In Fourth International Workshop on Numerical Software Verification, 2011. (Cité en page 22.)

[Ponsini 2012] Olivier Ponsini, Claude Michel et Michel Rueher. *Refining abstract interpretation based value analysis with constraint programming techniques.* In Principles and Practice of Constraint Programming, pages 593–607. Springer, 2012. (Cité en page 22.)

[Putot 2004] Sylvie Putot, Eric Goubault et Matthieu Martel. *Static analysis-based validation of floating-point computations.* In Numerical software with result verification, pages 306–313. Springer, 2004. (Cité en page 20.)

[Putot 2012] Sylvie Putot. *Static analysis of numerical programs and systems.* PhD thesis, habilitation thesis, university paris sud, 2012. (Cité en pages 22 et 97.)

[Rice 2008] Michael Rice et Sanjay Kulhari. *A survey of static variable ordering heuristics for efficient bdd/mdd construction*. University of California, Tech. Rep, 2008. (Cité en page 71.)

[Rump 1988] Siegfried M Rump. *Algorithms for verified inclusions*. Reliability in computing, vol. 19, pages 109–126, 1988. (Cité en page 14.)

[Russinoff 2000] David M Russinoff. *A Case Study in Formal Verification of Register-Transfer Logic with ACL2 : The Floating Point Adder of the AMD Athlon TM Processor*. In Formal Methods in Computer-Aided Design, pages 22–55. Springer, 2000. (Cité en page 25.)

[Sterbenz 1974] Pat H Sterbenz. Floating-point computation, volume 26. Prentice-Hall Englewood Cliffs, NJ, 1974. (Cité en page 14.)

[Sunaga 1958] Teruo Sunaga. *Theory of an interval algebra and its application to numerical analysis*. page 547–564, 1958. (Cité en page 29.)

[Sy 2001] Nguyen Tran Sy et Yves Deville. *Automatic Test Data Generation for Programs with Integer and Float Variables*. In 16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA, pages 13–21, 2001. (Cité en page 30.)

[Tillmann 2008] Nikolai Tillmann et Jonathan De Halleux. *Pex–white box test generation for. net*. In Tests and Proofs, pages 134–153. Springer, 2008. (Cité en pages 31 et 76.)

[Van Hentenryck 1989] Pascal Van Hentenryck. Constraint satisfaction in logic programming. MIT Press, 1989. (Cité en page 28.)

[Williams 2005] Nicky Williams, Bruno Marre, Patricia Mouy et Muriel Roger. *Pathcrawler : Automatic generation of path tests by combining static and dynamic analysis*. In Dependable Computing-EDCC 5, pages 281–292. Springer, 2005. (Cité en page 30.)

[Zimmermann 2000] Paul Zimmermann et G Hanrot. *MPFR : a library for multi-precision floating-point arithmetic with exact rounding*. In 4th Conference on Real Numbers and Computers, 2000, Dagstuhl, 89, volume 90, 2000. (Cité en page 21.)

Résumé : La vérification de programmes avec des calculs sur les nombres à virgule flottante est une étape très importante dans le développement de logiciels critiques. Les calculs sur les nombres flottants sont généralement imprécis, et peuvent dans certains cas diverger par rapport au résultat attendu sur les nombres réels.
L'objectif de cette thèse est de concevoir un solveur de contraintes sur les nombres à virgule flottante dédié à la vérification de programmes. Nous présentons dans ce manuscrit une nouvelle méthode de résolution de contraintes sur les flottants. Cette méthode se base principalement sur la sur-approximation des contraintes sur les flottants par des contraintes sur les réels. Cette sur-approximation doit être conservative des solutions sur les flottants. Les contraintes obtenues sont ensuite résolues par un solveur de contraintes sur les réels. Nous avons proposé un algorithme de filtrage des domaines sur les flottants basé sur le concept de la sur-approximation qui utilise des techniques de programmation linéaire. Nous avons aussi proposé une méthode de recherche de solutions basée sur des heuristiques. Cette méthode offre aussi la possibilité de comparer le comportement des programmes par rapport à une spécification sur les réels. Ces méthodes ont été implémentées et expérimentées sur un ensemble de programmes avec du calcul sur les nombres flottants.

Mots clés : Vérification de programmes, programmation par contraintes, nombres à virgule flottante

Constraint solver over floating-point numbers designed for program verification

Abstract : The verification of programs with floating-point numbers computation is an important issue in the development of critical software systems. Computations over floating-point numbers are not accurate, and the results may be very different from the expected results over real numbers.
The aim of this thesis is to design a constraint solver over floating-point numbers for program verification purposes. We introduce a new method for solving constraints over floating-point numbers. This method is based on an over-approximation of floating-point constraints using constraints over real numbers. This over-approximation is safe, that's to say it doesn't loose any solution over the floats. The generated constraints are then solved with a constraint solver over real numbers. We propose a new filtering algorithm using linear programming techniques, which takes advantage of these over-approximations of floating-point constraints. We introduce also new search methods and heuristics to find floating-point solutions of these constraints. Using our implementation, we show on a set of counter-examples the difference of the execution of programs over the floats with the specification over real numbers.

Keywords : Program verification, constraints programming, floating-point numbers