Rapport de stage
Estimation de WCET : analyse haut-niveau avec Interprétation abstraite et Programmation par contraintes

Aurèle Barrière

Résumé

Nous nous intéressons au problème du WCET qui consiste à calculer ou estimer le pire temps d'exécution d'un programme. Nous proposons une méthode qui mélange analyse statique et programmation par contraintes. L'originalité de la méthode est de traduire différentes informations venant d'analyses du programme en problèmes de satisfaction de contraintes dont il faut compter les solutions. La méthode proposée adapte un solveur de contraintes, utilise un interprète abstrait et reformule les contraintes pour résoudre le problème. La méthode a été implémentée avec l'analyseur statique SawjaCard et le solveur de contraintes AbSolute avec des résultats prometteurs.

Mots-clés Interprétation abstraite ; Programmation par contraintes ; Estimation WCET

1 Introduction

Alors que de nombreuses tâches critiques sont confiées à des ordinateurs, il est nécessaire d'avoir des garanties sur les programmes exécutés. Ainsi, un système embarqué doit pouvoir garantir que son programme s'exécutera en un temps limité : par exemple, un programme qui choisit de déclencher un airbag doit s'exécuter en moins de 150 millisecondes.

Dans le cadre de ce rapport, nous nous intéressons à la majoration du pire temps d'exécution d'un programme. Différentes approches existent pour estimer le WCET. Les méthodes reposant sur le test mesurent les temps de plusieurs exécutions, mais ne peuvent pas garantir une majoration sûre du pire temps d'exécution.

Pour en trouver une majoration sûre, les méthodes issues de l'analyse statique consistent à analyser un programme sans l'exécuter. Une approximation du comportement du programme y est réalisée, de telle sorte qu'on soit garanti d'englober tous les comportements possibles. Il est alors possible de prouver des propriétés sur les nombre d'exécutions de chaque arête par exemple.

Dans ce travail, nous nous intéressons à déterminer une estimation sûre du pire temps d'exécution d'un programme, en majorant le nombre d'exécutions de chaque instruction. Nous considérons des programmes qui terminent, sans paramètres. Nous utilisons ainsi l'analyse statique, mais nous proposons également d'utiliser la programmation par contraintes dont la richesse des langages de contraintes nous permet une meilleure précision que les méthodes précédentes.

Dans un premier temps, nous présentons un état de l'art non exhaustif des trois domaines impliqués dans ce travail : interprétation abstraite, programmation par contraintes et estimation de pire temps d'exécution (section 2). Ensuite, nous présentons la méthode introduite pour obtenir une estimation sûre du pire temps d'exécution (section 3). Après une section (4) consacrée aux définitions, notations et premières propriétés, nous détaillons dans les sections 5, 6 et 7 plusieurs étapes de la méthode. Enfin, nous présentons son implémentation (section 8).

2 État de l'art

Nous présentons dans cette section un état de l'art non exhaustif de l'interprétation abstraite, de la programmation par contraintes et de l'estimation de WCET. Pour des présentations plus détaillées, voir [4], [13], [14].

2.1 Interprétation abstraite

Alors qu'il est indécidable de connaître dans le cas général une propriété non triviale sur des programmes, l'analyse statique consiste à analyser un programme sans l'exécuter dans le but de vérifier des propriétés : il peut s'agir de déterminer si une certaine partie du code peut s'exécuter avec certains paramètres, s'il y a pas d'*overflow* d'une variable, de division par 0, etc.

Dans l'analyse statique, nous nous intéressons à l'**interprétation abstraite**, une théorie d'approximation de la sémantique d'un programme introduite par Cousot et Cousot dans [4]. L'interprétation abstraite associe à chaque point de programme un élément de domaine abstrait. Nous nous intéressons particulièrement aux domaines abstraits qui représentent un ensemble d'**états de la mémoire** atteignables. Un interprète abstrait calcule, étant donné un certain domaine abstrait, un élément de ce domaine par point de programme, de telle sorte qu'on ait une sur-approximation **correcte** de sa sémantique, c'est à dire que chaque fois que le programme passe par ce point, l'état de la mémoire est bien inclus dans l'élément de domaine abstrait associé. Par exemple, dans le cas d'un programme où une variable x est toujours comprise entre 0 et 10, si notre interprète abstrait trouve un domaine où x est compris entre -2 et 12, il s'agit bien d'une interprétation abstraite correcte.

Le domaine des intervalles est un exemple de domaine abstrait numérique. Pour un programme utilisant n variables, un élément du domaine abstrait des intervalles est un produit cartésien de n intervalles. Ainsi, l'interprétation abstraite correcte la plus précise avec le domaine des intervalles entiers du programme FIGURE 1 associe au point de programme `P` (indiqué par un commentaire) l'élément de domaine abstrait $[\![0, 22]\!] \times [\![1, 485]\!]$.

```
START
x = 0;
y = 0;
while (x < 22) do
   while (y < x*x) do
   y = y+1; // P
   done;
x = x+1;
done;
END
```

FIGURE 1 – Exemple de programme avec une relation non linéaire entre deux variables

Mais il existe également des domaines abstraits qui tiennent compte des relations entre les variables. Par exemple le domaine des polyèdres, introduit dans [6], exprime toutes les relations linéaires entre variables. L'ensemble des valeurs de (x, y) telles que $x \in [0, 5]$, $y \in [2, 7]$ et $2 \times x < y$ est un polyèdre. Ces domaines sont appelés **domaines relationnels**.

Il existe de nombreux domaines abstraits pour représenter différents types de contraintes [5] : relations entre variables linéaires ou de congruence, bornes sur les variables, etc. Le choix du domaine abstrait est une question d'importance en interprétation abstraite. Un domaine abstrait doit pouvoir représenter le plus d'informations possibles tout en disposant des outils nécessaires (union de plusieurs éléments, outils pour sur-approximer les boucles qu'on ne peut pas exécuter, etc) en un temps raisonnable. Par exemple, le domaine des intervalles ne porte pas beaucoup d'informations, le résultat d'une interprétation abstraite avec le domaine des intervalles n'est donc pas très précis. Cependant, elle est plus rapide qu'avec le domaine abstrait des octogones [10], qui permet d'exprimer toutes les informations de la forme $\pm x \pm y \leq c$ où x et y sont des variables du programme.

L'interprétation abstraite peut être faite de manière paramétrique, c'est à dire que le programme a en entrée des paramètres et le résultat de l'interprétation abstraite dépend de ces paramètres.

L'interprétation abstraite est notamment utilisée pour la vérification de code critique. Par exemple, son utilisation a permis de prouver que les logiciels de commande de vol des avions Airbus A340 et A380 ne pouvaient pas avoir d'erreurs à l'exécution [1].

2.2 Programmation par contraintes

Le but de la **programmation par contrainte** est de modéliser et résoudre des problèmes donnés sous une forme déclarative. Le problème est décrit avec des variables, inconnues du problème, sur lesquelles portent des contraintes, relations de la logique du premier ordre portant sur les variables. Les solveurs de contraintes implémentent des outils génériques permettant de résoudre efficacement de tels problèmes.

On appelle **CSP (constraint satisfaction problem)** un problème sous la forme d'un triplet (X_{CSP}, D, C) : X_{CSP} un ensemble de variables, D un ensemble de domaines (un pour chaque variable), et C un ensemble de contraintes sur ces variables. Un **COP (constraint optimization problem)** est un quadruplet (X_{CSP}, D, C, O) où O est la fonction objectif (fonction des variables de X_{CSP}). Il faut alors trouver la valuation des variables de X_{CSP} dans les domaines de D qui vérifient les contraintes de C et qui minimise (ou maximise) O.

Par exemple, $X = \{x_1, x_2\}$, $D = \{[\![0, 2]\!], [\![0, 4]\!]\}$ et $C = \{x_1 = 0, x_1 + x_2 = 5\}$ est un CSP sans solution. Par contre, si $D = \{[\![0, 2]\!], [\![0, 7]\!]\}$, alors on a la solution $x_1 = 0, x_2 = 5$.

Les domaines peuvent changer complètement les méthodes de résolution. Par exemple, on n'utilise pas les mêmes méthodes sur les variables entières ou réelles (même flottantes).

L'objectif d'un solveur de contraintes peut être différent selon le problème et le solveur [13] :

trouver une solution au problème, trouver toutes les solutions du problème, ou les compter, ou trouver une valuation des variables qui vérifie les contraintes tout en minimisant une certaine fonction (pour un COP).

Pour cela, la programmation par contrainte propose des méthodes qui rendent la résolution la plus rapide possible. En effet, pour tous ces problèmes il serait possible d'énumérer toutes les valuations possibles et de regarder lesquelles vérifient les contraintes, mais ce serait au prix d'une complexité temporelle exponentielle. Les méthodes de propagation consistent à raisonner automatiquement sur les contraintes pour réduire le nombre de cas à énumérer. On définit ainsi des algoithmes de consistance qui réduisent l'espace de recherche tout en garantissant de conserver toutes les solutions.

Il est aussi important de noter que la programmation par contrainte n'est pas paramétrée : tous les solveurs génériques existant à ce jour à notre connaissance ne donnent comme solution que des valuations explicites de chaque variable. Dans le cas de certains langages de contraintes (par exemple, avec seulement des contraintes linéaires), on peut trouver des solutions qui dépendent d'un paramètre, mais toujours au prix d'une restriction des problèmes, là où la programmation par contrainte se propose idéalement de résoudre des problèmes sur un langage de contraintes le plus large possible.

2.3 Estimation de WCET

Le WCET (*worst case execution time*) désigne le pire temps d'exécution d'un programme. Il existe essentiellement deux types de méthodes pour estimer le WCET [14] : faire des mesures sur plusieurs exécutions du programme et prendre empiriquement le plus long temps d'exécution au risque d'oublier certains cas, ou par analyse statique. L'analyse statique permet d'avoir une majoration **sûre** du WCET, c'est à dire qu'on ne pourra jamais dépasser cette estimation, quitte à avoir une valeur trop grande. C'est ce type de méthodes que nous étudierons dans ce rapport.

Une estimation statique de WCET comporte plusieurs étapes :

- Une analyse bas-niveau, spécifique au matériel utilisé, pour trouver une majoration des pire temps d'exécution de chaque instruction.
- Une analyse haut-niveau, qui représente une exécution de programme comme le parcours d'un graphe et estime alors le nombre d'exécution de chaque point de programme.
- Une analyse des anomalies temporelles. Ces anomalies sont par exemple introduites par les caches, les *pipelines* ou la prédiction de branchement, et influencent beaucoup le temps d'exécution.

Dans ce travail, nous nous intéresserons à l'analyse haut-niveau : il s'agit de trouver une sur-approximation, la plus précise possible, du nombre d'exécution de chaque instruction du programme.

2.4 Interprétation abstraite pour l'estimation de WCET haut-niveau

Nous fondons notre méthode sur un résultat qui permet d'utiliser l'interprétation abstraite pour estimer le WCET : Pour un programme déterministe qui termine, chaque point de programme ne peut pas être exécuté plus de fois que le cardinal de l'ensemble des états mémoires atteignables en ce point de programme. Un énoncé plus formel et une intuition de la démonstration sont donnés section 4.3.

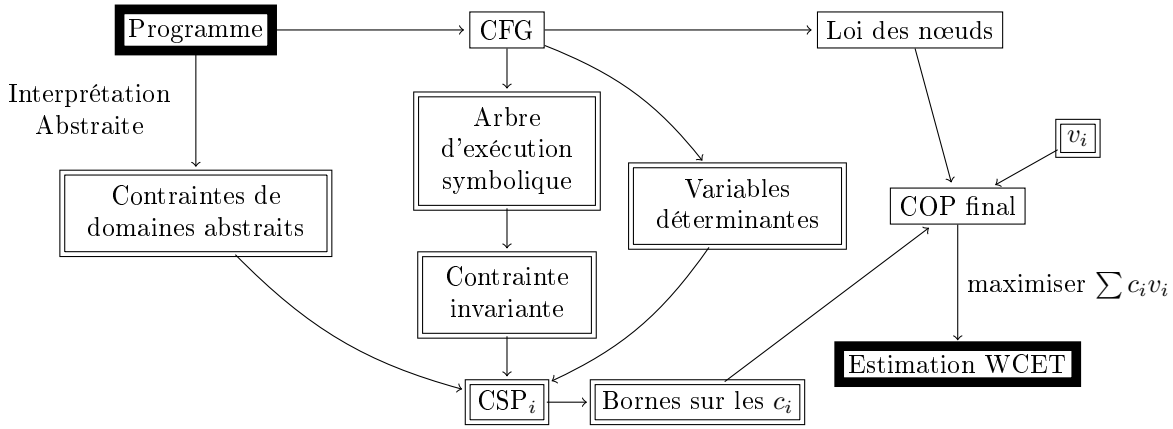


FIGURE 2 – Méthode proposée pour l'estimation de WCET non paramétrique
Un nœud est doublement encadré quand on doit en avoir un par point de programme

Ainsi, comme une interprétation abstraite correcte donne, en chaque point de programme, une sur-approximation de l'ensemble des états mémoires atteignables, on peut se servir de l'interprétation abstraite pour avoir une majoration du nombre d'états de mémoire atteignables, et donc une estimation sûre de WCET. C'est la méthode utilisée par exemple dans [3].

Pour affiner cette première majoration, on peut déduire des contraintes linéaires du graphe de flot de contrôle (défini section 4.1) sur les nombres d'exécutions de chaque arête. Dans [3], Bygde, Ermedahl et Lisper proposent la construction d'un arbre pour exprimer certaines de ces contraintes. Ces contraintes seront alors résolues de manière paramétrique. De cette manière, ils obtiennent une estimation sûre du WCET paramétrique.

3 Résumé de la contribution

Nous proposons dans ce rapport la méthode résumée FIGURE 2.

Cette méthode est une analyse statique haut-niveau sûre de WCET. Le but est donc de trouver une estimation sûre du temps d'exécution en majorant le nombre d'exécutions de chaque instruction. Notre contribution repose sur une interprétation abstraite qui donne une sur-approximation des états mémoires atteignables par le programme, dont nous utilisons le résultat comme des contraintes.

Dans un premier temps, nous construisons un problème CSP_i pour chaque arête i du CFG, de telle sorte que les états mémoires accessibles en cette arête soient des solutions du CSP_i. Les variables de ce CSP sont les différentes versions des variables du programme. La signification de ces différentes versions sera détaillée dans la section 5.

Les contraintes du CSP_i viennent du résultat d'une interprétation abstraite à l'arête i, et de la création, à partir du CFG, d'un arbre d'exécution symbolique qui donne une formule invariante (section 5.3). Ces contraintes diminuent le nombre de solutions du problème.

Nous proposons également un ensemble de variables déterminantes qui permet de modifier le calcul de solutions pour obtenir une borne plus précise sur le nombre d'exécution de chaque arête (section 6). Enfin, le CFG nous donne de nouvelles contraintes avec la loi des nœuds, qui définissent un dernier COP (section 7). Il suffit alors de résoudre ce problème en maximisant une bonne fonction objectif pour obtenir l'estimation de WCET.

L'interprétation abstraite servait déjà de point de départ de l'estimation. Cependant, si le

CFG a déjà été exploité pour de l'estimation de WCET paramétrique [9], ce n'était pas sous forme de contraintes d'un problème CSP. De même, si des méthodes existaient pour récupérer tous les invariants possible sous forme de contraintes linéaires [8], la construction d'arbre d'exécution symbolique présentée dans ce rapport est la seule à notre connaissance qui exprime des contraintes non linéaires pour en déduire une estimation de WCET.

Nous ne cherchons pas ici à obtenir une expression paramétrique comme dans les travaux de Bygde, Ermedhal et Lisper [3], ce qui va nous permettre d'utiliser la programmation par contraintes.

4 Préliminaires techniques

Notation Dans tout le rapport, nous noterons $\#A$ le cardinal, fini ou infini, de l'ensemble A.

Dans cette section, nous décrivons la syntaxe des langages utilisés, puis la sémantique d'une exécution d'un programme.

4.1 Syntaxe

Nous utilisons dans la suite des programmes écrits dans une syntaxe simplifiée. Soit X un ensemble de variables. Soit *Val* un ensemble de valeurs. Dans toute la suite, $Val = \mathbb{Z}$.

Langages utilisés Pour nos exemples et notre implémentation, nous utiliserons les langages d'expressions et de tests $\mathcal{E}, \mathcal{T}$ définis par les grammaires :

$$\mathcal{E} : \qquad \texttt{expr} := (\texttt{expr} \lozenge \texttt{expr}) \mid -\texttt{expr} \mid x \mid c$$

où $x \in X$, $c \in Val$ et $\lozenge \in \{+, -, \times, \%\}$;

$$\mathcal{T} : \qquad \texttt{test} := (\texttt{expr} \bowtie \texttt{expr}) \mid \textbf{not}\ \texttt{test} \mid (\texttt{test}\ \textbf{and}\ \texttt{test}) \mid (\texttt{test}\ \textbf{or}\ \texttt{test})$$

où $\bowtie \in \{=, \neq, <, \leq, \geq, >\}$.

Définition: Variable utilisée On appelle variable utilisée par un test ou une expression toute variable qui apparaît dans sa formule syntaxique. Par exemple, $x - x = 0$ utilise la variable x.

Dans la suite, nous représentons les programmes par leur graphe de flot de contrôle.

Définition: CFG Un CFG (*control flow graph*) est un graphe orienté $G = (V, E)$, tel que chaque nœud $v \in V$ est de la forme : `START(i)` ou `Assign(x,expr,i,j)` ou `Test(test,i,j,k)` ou `Join(i,j,k)` ou `End(i)`, avec $i, j, k \in E$, $x \in X$, `expr` $\in \mathcal{E}$, `test` $\in \mathcal{T}$. Une représentation de chaque nœud est donnée FIGURE 3. De plus, V contient un unique nœud `START` et un unique nœud `END`. On a `expr` $\in \mathcal{E}$ et `test`$\in \mathcal{T}$.

Remarques

- Dans un tel graphe, chaque arête va d'un nœud vers un autre. Chaque arête a donc exactement un nœud d'origine et un nœud de destination.
- Dans la suite, par convention, on suppose les arêtes numérotées de 1 à n. L'arête 1 est celle qui sort du nœud `START`, et l'arête n celle qui entre dans le nœud `END`.

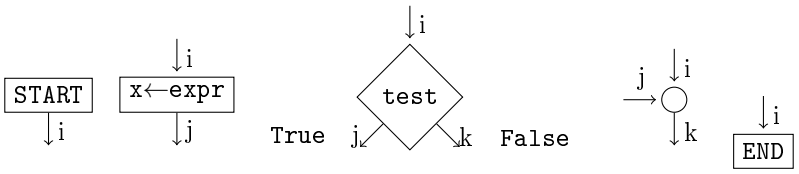


FIGURE 3 – Les différents éléments des CFG de nos programmes
`START(i)`, `Assign(x,expr,i,j)`, `Test(test,i,j,k)`, `Join(i,j,k)`, `End(i)`

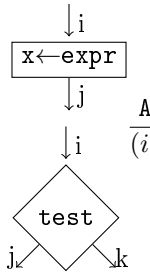
4.2 Sémantique opérationnelle

Nous présentons dans cette section la formalisation d'une exécution d'un CFG et d'un état mémoire.

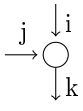
Définition: État mémoire On appelle état mémoire une fonction $\sigma : X \to (Val \cup \{unassigned\})$.

On note Σ l'ensemble des états mémoires, et σ_0 la fonction constante $\forall x \in X, \sigma_0(x) = unassigned$.

Définition: Configuration On appelle configuration γ un couple (e, σ) où $e \in E$ et $\sigma \in \Sigma$. On note Γ l'ensemble des configurations. On note $[\![]\!] : \mathcal{E} \to Val$ la sémantique d'une expression. On note $[\![]\!] : \mathcal{T} \to \{true, false\}$ la sémantique d'un test. On définit une relation $\mapsto$ sur Γ :



$$\frac{\texttt{Assign(x,expr,i,j)} \in G}{(i,\sigma) \mapsto (j, \sigma[x \leftarrow [\![\texttt{expr}]\!]_\sigma])}$$



$$\frac{\texttt{Join(i,j,k)} \in G}{(i,\sigma) \mapsto (k,\sigma)} \qquad \frac{\texttt{Join(i,j,k)} \in G}{(j,\sigma) \mapsto (k,\sigma)}$$

$$\frac{\texttt{Test(t,i,j,k)} \in G \text{ et } [\![\texttt{test}]\!]_\sigma = true}{(i,\sigma) \mapsto (j,\sigma)} \qquad \frac{\texttt{Test(t,i,j,k)} \in G \text{ et } [\![\texttt{test}]\!]_\sigma = false}{(i,\sigma) \mapsto (k,\sigma)}$$

Définition: Exécution Soit G un CFG. Une exécution de G est une suite de configurations (finie ou non) $(\gamma_k)_{k\in\mathbb{N}}$ telle que $\gamma_0 = (1, \sigma_0)$ et $\forall k \in \mathbb{N}, \gamma_k \mapsto \gamma_{k+1}$.

Définition: Exécution maximale μ, exécution finie se terminant sur γ, est une exécution maximale s'il n'existe pas de configuration γ' telle que $\gamma \mapsto \gamma'$. Une exécution infinie est maximale.

Hypothèses

- H_1 Déterminisme : On considère une sémantique des expressions et des tests déterministe.
- H_2 Terminaison : On considère les CFG dont toute exécution maximale est finie, et termine sur une configuration (n, σ).
- On note $H = H_1 \wedge H_2$.

Proposition: Un CFG qui satisfait H n'a qu'une seule exécution maximale.

Idée de démonstration : Par induction, comme chaque exécution est finie (hypothèse de terminaison) et comme la sémantique des expressions et tests est déterministe, on n'a pas le choix de la prochaine configuration dans l'exécution à chaque étape.

4.3 Estimation du nombre d'exécutions

Nous nous intéressons dans cette section aux compteurs d'exécutions de chaque arête, et plus généralement aux notations utilisées pour l'analyse haut-niveau de WCET.

Définition: Compteur d'exécution de l'arête i (noté c_i) Soit G un CFG qui satisfait H. Soit $\mu = (\gamma_k)_{k\in\mathbb{N}}$ son exécution. c_i représente le nombre de fois qu'est exécutée l'arête i. $c_i = \#\{k \in \mathbb{N} \mid \exists \sigma, \gamma_k = (i, \sigma)\}$.

Définition: $E(i)$ Soit G un CFG qui satisfait H. Soit $\mu = (\gamma_k)_{k\in\mathbb{N}}$ son exécution. $E(i)$ représente l'ensemble des états mémoires accessibles à l'arête i. $E(i) = \{\sigma_k \mid (i, \sigma_k) \in \mu\}$.

Proposition: Soit G un CFG qui satisfait H, avec n arêtes. Alors $\forall i \in \{1 \dots n\}$, $c_i = \#E(i)$.

Idée de la démonstration : comme le programme est déterministe, la prochaine étape d'une exécution ne dépend que de la configuration courante. Ainsi, on ne peut pas passer deux fois dans un même arête avec le même état mémoire sinon l'exécution comporterait cette configuration une infinité de fois et ne terminerait pas. Alors, pour une arête i, tous les $\gamma_k = (e_k, \sigma_k) \in \mu$ tels que $e_k = i$ ont des σ_k tous différents. Alors $c_i = \#\{k \in \mathbb{N} \mid \exists \sigma, \gamma_k = (i, \sigma)\} = \#\{\sigma_k \mid (i, \sigma_k) \in \mu\} = \#E(i)$.

Définition: $AI(i)$ Soit G un CFG qui satisfait H et $\mathcal{D}^\#$ un domaine abstrait. On appelle $AI(i)$ l'élément de domaine abstrait de $\mathcal{D}^\#$ associé à l'arête i après avoir effectué une interprétation abstraite sur G.

Correction de l'interprétation abstraite Soit G un CFG qui satisfait H, avec n arêtes. On dit qu'une interprétation abstraite est correcte si $\forall i \in \{1 \dots n\}$, $E(i) \subseteq AI(i)$.

Proposition: Soit G un CFG qui satisfait H, avec n arêtes. $\forall i \in \{1 \dots n\}$, soit v_i une majoration sûre du pire temps d'exécution de l'instruction exécutée avant l'arête i, obtenue par analyse bas-niveau. Alors $\sum_i c_i \times v_i$ est une majoration sûre du WCET.

Intuition de la démonstration v_i est une majoration sûre du pire temps d'exécution t_i. Donc $v_i \geq t_i$. Comme $\forall i, c_i \geq 0, t_i \geq 0$, alors $\sum_i c_i \times v_i \geq \sum_i c_i \times t_i \geq WCET$.

4.4 Exemple

Le CFG FIGURE 4 est un CFG vérifiant les hypothèses H. Les arêtes sont numérotées de 1 à 11. Ici, un état mémoire est une fonction $\{x, y\} \to \mathbb{Z}$. Son unique exécution contient chacune des arêtes une ou plusieurs fois, et termine par les configurations $(8, (x = 22, y = 22)) \mapsto (11, (x = 22, y = 22))$. On a $c_3 = 23$. $E(4) = \{(x = k, y = 0) \mid k \in \{0 \dots 22\}\}$. Avec le domaine abstrait des intervalles et une interprétation abstraite correcte, on pourra avoir au mieux $AI(5) = [0, 22] \times [0, 22]$ (de cardinal 529), alors que $E(5) = \{(x = k, y = l) \mid k \leq 22 \text{ et } l \leq k\}$ (de cardinal $276 = c_5$).

5 Construction des CSP$_i$

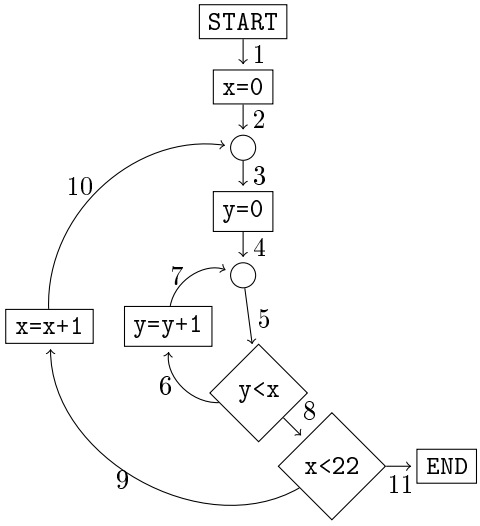


FIGURE 4 – Un exemple de CFG vérifiant les hypothèses

Soit G un CFG qui satisfait H, avec n arêtes et X comme ensemble de variables. Nous construisons ici, pour chaque arête i, un CSP$_i$ dont l'ensemble de solutions doit contenir $E(i)$, c'est à dire que chaque état mémoire accessible doit être solution du CSP.

5.1 Les variables des CSP$_i$

On appelle **variable de programme** une des variables du programme donné en entrée. On note X l'ensemble des variables de programme.

Définition: Version de variable Un couple (x, k) où $x \in X$ et $k \in \mathbb{N}$. Ce sont les variables des CSP$_i$. Dans le CSP$_i$, $(x, 0)$ correspond à x dans l'arête i. Dans la suite, on identifie $x \in X$ et $(x, 0) \in X_{CSP}$.

Cette notion de version n'est pas sans rappeler le SSA [7]. Cependant, il ne s'agit pas ici de considérer une version de variable par arête du graphe, nous ne créerons de nouvelle version que lorsque la valeur de la variable change et dépend de son ancienne version (cela sera expliqué dans la section 5.3). De plus, la numérotation des versions n'est pas globale, elle dépend de l'arête i.

Définition: Sol Soit C un CSP. On note Sol(C) l'ensemble des solutions de C. Un élément de Sol(C) est une valuation de chaque variable du CSP qui satisfait les contraintes.

On crée ainsi pour chaque arête i du CFG un CSP$_i$. Ses variables sont toutes les versions de variables utilisées dans l'arbre d'exécution symbolique (voir section 5.3). Ses contraintes sont celles issues de l'interprétation abstraite (section 5.2) et de l'arbre d'exécution symbolique (section 5.3).

5.2 Contraintes issues de l'interprétation abstraite

Pour un programme qui satisfait nos hypothèses de terminaison et déterminisme, on sait que $c_i = \#E(i)$. Et dans le cas d'une interprétation abstraite correcte, $E(i) \subseteq AI(i)$. Nous construisons dans un premier temps un ensemble de contraintes $\mathcal{C}_{\mathrm{AI}(i)}$, de telle sorte que l'ensemble de solutions de $\mathcal{C}_{\mathrm{AI}(i)}$ soit $AI(i)$.

La première étape de notre méthode consiste à faire une interprétation abstraite sur le programme. L'interprétation abstraite nous donne donc, pour chaque arête, un élément de domaine abstrait. Dans les domaines abstraits considérés (intervalles, octogones, polyèdres), il s'agit de contraintes sur les variables de programme[1] : si à une certaine arête, l'interprétation abstraite affirme que x appartient à l'intervalle $[0, 2]$, alors on a $(x, 0) \geq 0$ et $(x, 0) \leq 2$.

1. Chaque domaine abstrait est défini à l'aide d'une fonction de concrétisation γ qui associe à un élément d'un domaine abstrait un ensemble d'éléments du domaine concret. Dans les domaines considérés dans ce travail, on peut voir cette fonction comme des contraintes sur les variables du programme.

$\mathcal{C}_{\text{AI}(i)}$est ainsi un ensemble de contraintes sur tous les couples $(x, 0)$ où $x \in X$, et traduit les contraintes de l'interprétation abstraite.

Exemple Dans la FIGURE 4, après une interprétation abstraite correcte dans le domaine des intervalles ayant donné la boîte $[0, 22] \times [0, 22]$, $\mathcal{C}_{\text{AI}(5)} = \{x \geq 0, x \leq 22, y \geq 0, y \geq 22\}$

5.3 Arbres d'exécutions symboliques

Cependant, l'interprétation abstraite nous donne toujours des éléments de domaines abstraits, ce qui peut être une source d'imprécision. Ainsi dans l'exemple FIGURE 4, $E(5) = \{(x = k, y = l) \mid k \leq 22 \text{ et } l \leq k\}$, mais le domaine des intervalles donnera au mieux $AI(5) = [0, 22] \times [0, 22]$. En réalité, l'arête est exécutée 276 fois, mais ici on trouvera seulement $c_i \leq 529$.

Mais les problèmes de programmation par contraintes peuvent avoir des ensembles de solutions de n'importe quelle forme. Le but des arbres d'exécution symbolique est d'ajouter aux différents CSP_i des contraintes sur les versions de variables qui soient toujours vraies quand l'exécution passe par l'arrête i. Ainsi, l'ensemble de solutions est plus petit qu'avec l'interprétation abstraite seule puisqu'on a rajouté des contraintes, et donc notre approximation des c_i est plus précise. Mais on a toujours $E(i) \subseteq \text{Sol}(\text{CSP}_i)$ si on identifie les éléments de X avec leur version de X_{CSP} d'indice 0.

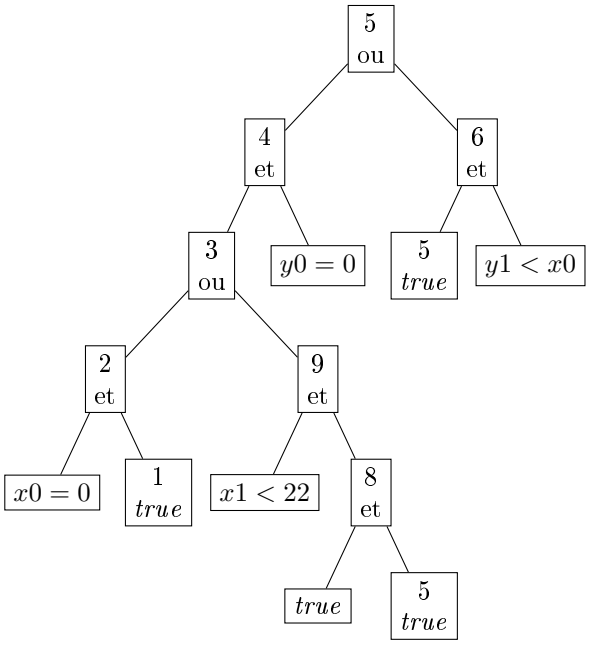


FIGURE 5 – L'arbre d'exécution symbolique de l'arête 5 du CFG FIGURE 4

5.3.1 Construction de l'arbre

Soit G un CFG vérifiant H, avec n arêtes. Soit X l'ensemble des variables utilisées. Soit i l'arête considérée. Soient $\mathcal{E}, \mathcal{T}$ les langages d'expressions et de tests utilisées. Soit $[\![]\!]$ une sémantique de tests et d'expressions déterministe.

Définition: Arbre syntaxique de contrainte Un arbre syntaxique de contrainte est soit une contrainte écrite sur $\mathcal{VT} \cup \{true\}$, soit un nœud OU avec deux fils, soit un nœud ET avec deux fils. Les fils sont eux-mêmes des arbres syntaxiques de formule. *true* est une contrainte toujours vérifiée.

Pour construire l'arbre d'exécution symbolique en une arête i, on regarde le nœud dont vient i. Si c'est une affectation $x \leftarrow 3$ vers laquelle arrive une arête j, alors on sait qu'en i, la contrainte $(x, 0) = 3$ est vraie, ainsi que toute contrainte qui était vraie en j, sauf celles sur x puisque la variable a été affectée entre l'arête i et j. L'arbre sera donc un nœud ET, de la formule $x = 3$ et de l'arbre d'exécution symbolique de j où on retient que x est déjà affectée et que l'arête i est déjà vue. De même, si l'arête i venait d'un nœud `Join(j,k,i)`, alors l'exécution est passée par l'arête j ou k. On aura donc un nœud OU, avec le arbres de j et k. Lorsqu'on a des affectations où une variable dépend de son ancienne valeur (par exemple $x \leftarrow x + 1$), alors on se sert des

versions de variables pour conserver les contraintes sur l'ancienne valeur de x. Ainsi, on aura la contrainte $(x,0) = (x,1) + 1$, puis dans le reste de l'arbre, on écrira les contraintes sur $(x,1)$. Enfin, pour que la construction termine, on ne considèrera pas deux fois la même arête dans la même branche. Pendant la construction, il faudra retenir, pour chaque branche, un ensemble de variables déjà affectées (initialisé avec $\emptyset$), un ensemble d'arêtes déjà visitées (initialisé avec $\emptyset$), et une fonction qui à chaque variable donne la version à utiliser (initialisé avec la fonction constante égale à 0). De plus, il faudra écrire en dehors de l'arbre les contraintes de relations entre versions (par exemple, $(x,10) = (x,22) + (y,3) - 5$).

Définition: Langage de version On appelle langage de version les langages d'expressions $\mathcal{VE}$ et de tests $\mathcal{VT}$ dont les variables sont écrites avec une certaine version. Étant donné $v : X \to \mathbb{N}$ une fonction qui à chaque variable associe l'indice à utiliser, on a la fonction $\{.\}_v : \mathcal{T} \to \mathcal{VT}$ qui remplace les variables de l'argument par la version spécifiée par v. Par exemple, $\{(x+1) < y\}_{(x\mapsto 0, y\mapsto 1)} = ((x,0)+1) < (y,1)$. Les éléments de $\mathcal{VT}$ sont donc des contraintes sur X_{CSP}.

Définition: Arbre d'exécution symbolique L'arbre d'exécution symbolique de i dans G est un arbre syntaxique de contrainte fini défini ainsi :

$$SEtree(i) = \begin{cases} true & \text{si } i \text{ vient du nœud } \texttt{START} \\ true & \text{si } i \text{ a déjà été vue dans cette branche de l'arbre} \\ OU(SEtree(a), SEtree(b)) & \text{si } i \text{ vient d'un nœud } \texttt{Join(a,b,i)} \\ ET(\{\texttt{test}\}_v, SEtree(a)) & \text{si } i \text{ vient d'un nœud } \texttt{Test(test,a,i,b)} \text{ et aucune} \\ & \text{des variables de } \texttt{test}n\text{'a été affectée dans cette branche} \\ ET(\{not(\texttt{test})\}_v, SEtree(a)) & \text{si } i \text{ vient d'un nœud } \texttt{Test(test,a,b,i)} \text{ et aucune} \\ & \text{des variables de } \texttt{test}n\text{'a été affectée dans cette branche} \\ SEtree(a) & \text{si } i \text{ vient d'un nœud } \texttt{Test(test,a,b,i)} \text{ ou } \texttt{Test(test,a,i,b)} \\ & \text{et des variables de } \texttt{test} \text{ont été affectées dans cette branche} \\ ET(\{x = \texttt{expr}\}_v, SEtree(a)) & \text{si } i \text{ vient d'un nœud } \texttt{Assign(x,expr,a,i)} \\ & x \text{ n'appartient pas aux variables utilisées par } \texttt{expr} \text{ et ni } x \\ & \text{ni les variables d'} \texttt{expr} \text{ ont été affectées dans cette branche} \\ & x \text{ est alors une variable affectée pour } SEtree(a) \\ SEtree(a) & \text{si } i \text{ vient d'un nœud } \texttt{Assign(x,expr,a,i)} \\ & \text{et une variable de } \texttt{expr} \text{ou } x \text{ est affectée dans cette branche} \\ SEtree(a), & \\ \{x\}_v = \{\texttt{expr}\}_{v(x\mapsto new(x))} & \text{si } i \text{ vient d'un nœud } \texttt{Assign(x,expr,a,i)} \\ & \text{et } x \text{ est une variable utilisée par } \texttt{expr} \\ & \text{on remplace alors } v \text{ par } v(x \mapsto new(x)) \text{ dans cette branche} \end{cases}$$

La contrainte de relation entre versions $\{x\}_v = \{\texttt{expr}\}_{v(x\mapsto new(x))}$ n'est pas écrite dans l'arbre, mais comme une contrainte supplémentaire en conjonction avec la contrainte décrite par l'arbre.

Lors de la construction, on conserve, pour chaque variable, un compteur global du plus grand indice déjà utilisé pour cette variable. La fonction $new : X \to \mathbb{N}$ renvoie alors le prochain indice disponible en mettant à jour le compteur de la variable concernée.

v est initialement la fonction qui à chaque variable associe 0. Lorsqu'on construit l'arbre, il faut retenir pour chaque branche l'ensemble des variables affectées, l'ensemble des arêtes déjà visitées et une fonction qui à chaque variable associe l'indice courant.

La construction termine : dans une même branche, on ne peut pas rencontrer une même arête plus de deux fois.

Proposition: Correction de l'arbre. $E(i) \subseteq \text{Sol}(\text{CSP}_i)$ en identifiant les variables de programme avec leur version d'indice 0. On a alors $c_i \leq \#\text{Sol}(\text{CSP}_i)$.

Démonstration: en Annexes, section 10.1.

5.4 Exemple de CSP_i

Considérons le CFG FIGURE 4 et l'arbre d'exécution symbolique de l'arête 5 (FIGURE 5).

Si le domaine abstrait renvoyé par le domaine des intervalles était $AI(5) = [0, 22] \times [0, 22]$, alors nous obtenons le CSP suivant :

```
CSP 5
init{
  int x0 = [-1000;1000];
  int y0 = [-1000;1000];
  int x1 = [-1000;1000];
  int y1 = [-1000;1000];
}
constraints {
  // AI constraints
  x0 >= 0;
  x0 <= 22;
  y0 >= 0;
  y0 <= 22;
  // relations between versions
  x0 = x1+1;
  y0 = y1+1;
  // symbolic execution tree
  (y0=0 && ((x0=0 && true) || x1<22 && (true && true)))
  || (y1 < x0 && true);
}
```

Les deux contraintes de relations entre versions de variables proviennent des passages par les arêtes 7 et 10.

Dans cet exemple, $E(5) = \text{Sol}(CSP_5)$ et nous obtenons donc la valeur exacte de c_5. Dans le cas général, on ne peut que garantir $E(i) \subseteq \text{Sol}(\text{CSP}_i)$ (comme vu section 5.3.1). Dans ce cas, l'utilisation d'un domaine plus précis que les intervalles (par exemple les octogones) aurait aussi permis d'avoir $E(5) = AI(5)$. Mais dans un exemple où l'arbre d'exécution symbolique exprime des contraintes non linéaires, alors le CSP permettra bien une estimation plus précise, comme dans le programme FIGURE 1. Si il est toujours possible de créer un nouveau domaine abstrait qui exprimera les contraintes vérifiées par un programme, il ne s'agit pas d'un processus facile. La programmation par contraintes permet alors d'obtenir un gain de précision à moindre effort : il suffit d'avoir un langage de contraintes qui puisse exprimer les relations utilisées dans la syntaxe du programme.

6 Comptage de solutions

À partir de la section 5, nous obtenons, pour chaque point de programme i un CSP_i, dont l'ensemble des solutions contient l'ensemble des états mémoires atteignables à ce point de pro-

gramme : $E(i) \subseteq \text{Sol}(\text{CSP}_i)$. En résolvant ce CSP, il est possible d'obtenir le nombre de solutions $\#\text{Sol}(\text{CSP}_i)$. D'après la section 3, ce nombre est une majoration de c_i.

Cependant, il est parfois suffisant de compter seulement le nombre de solutions qui diffèrent sur un certain ensemble de variables pour obtenir une sur-approximation des c_i. Par exemple, prenons le programme FIGURE 6.

L'interprétation abstraite avec le domaine des intervalles donne la boîte $[0, 22] \times [0, 231]$: 5336 solutions possibles pour l'arête 7.

Dans cette boucle, les variables x et y sont reliées par la relation $y = \frac{x(x-1)}{2}$. Cependant, cette relation n'apparaît syntaxiquement ni dans une garde, ni dans une affectation. Donc, cette relation n'apparaît pas dans l'arbre d'exécution symbolique. Le CSP_7 a donc autant de solutions que : (nombre de valeurs possibles(x)) $\times$ (nombre de valeurs possibles(y)).

Pourtant, à chaque valeur de x correspond une unique valeur de y. Il suffit de calculer le nombre de valeurs possibles pour x pour majorer c_7.

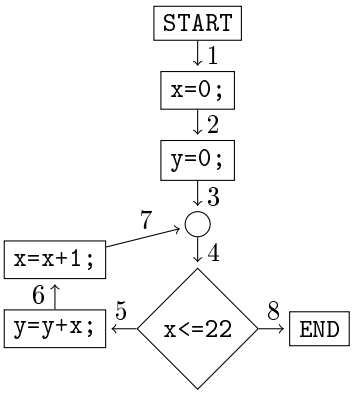


FIGURE 6 – Un CFG où une variable n'est pas déterminante

6.1 Variables déterminantes

Dans cette section, nous introduisons la notion de variables déterminantes. Soit G un CFG qui satisfait H, avec X l'ensemble de variables. Soit i une arête de G. Soit CSP_i un CSP obtenu comme dans la section 5, telle que $E(i) \subseteq \text{Sol}(\text{CSP}_i)$.

Définition: Sol^*_Y Soit $\mathcal{P} = (X_{CSP}, D, C)$ un CSP. Soit $Y \subseteq X_{CSP}$ un ensemble de variables. On note $\text{Sol}^*_Y(\mathcal{P})$ le nombre de solutions de $\mathcal{P}$ qui ne diffèrent que sur Y.

Exemple Soit $\mathcal{P}$ le CSP avec $X_{CSP} = \{x, y\}$, $D = \{\mathbb{Z}, \mathbb{Z}\}$, et $C = \{x = 0; y \geq 2; y + x \leq 11\}$. $\text{Sol}(\mathcal{P}) = 10$, $\text{Sol}^*_{\{x\}}(\mathcal{P}) = 1$.

Définition: Variables déterminantes Un ensemble $\mathcal{D} \subseteq X$ est un ensemble de variables déterminantes en i si $c_i \leq \text{Sol}^*_{\mathcal{D}}(\text{CSP}_i)$.

Notre objectif est donc de trouver pour chaque CSP_i un ensemble minimal de variables déterminantes, pour avoir la majoration la plus précise sur les c_i. Dans [2], Bygde prouve que les variables de contrôle forment un ensemble de variables déterminantes car il n'existe pas deux états mémoires différents accessibles en i qui coïncident sur cet ensemble.

Définition: Variables de contrôle d'une arête i Si une arête i est à l'intérieur d'une ou plusieurs boucles, une variable de contrôle de i est une variables utilisées dans les tests de ces boucles ou qui peut modifier indirectement (via des affectations) des variables impliquées dans ces tests.

Par exemple, dans le programme donné FIGURE 6, la variable y n'est pas une variable de contrôle de l'arête 7. On va en déduire qu'il suffit de compter les solutions qui ont une valeur différente sur x : il y en a 23, et non 5336.

6.2 Modifier le comptage

Une fois l'ensemble de variables de contrôle $\mathcal{D}$ trouvées pour chaque arête, il faut donc compter les solutions de chaque CSP qui diffèrent sur $\mathcal{D}$. On peut proposer deux solutions :

- On utilise un solveur de contraintes pour récupérer toutes les solutions possibles. Puis on les parcourt en ne comptant que le nombre de solutions différentes sur $\mathcal{D}$.
- On peut modifier la méthode de résolution du solveur. Dès qu'il trouve une solution au problème, on lui interdit de chercher parmi les valuations qui coïncident sur $\mathcal{D}$ avec cette solution. On a ainsi moins de valuations à vérifier et on peut espérer un temps de résolution plus court.

Pour l'instant, seule la première solution a été implémentée.

7 Loi des nœuds et COP final

Les sections précédentes nous permettent donc d'obtenir une majoration sûre de chaque c_i. De plus, le CFG permet de trouver de nouvelles contraintes sur les c_i.

7.1 Loi des nœuds

On peut déduire du CFG d'autres contraintes sur les c_i. Ces contraintes forment l'ensemble K.

Proposition :

- Pour chaque nœud `Assign(x,expr,i,j)`, $c_i = c_j$.
- Pour chaque nœud `Test(t,i,j,k)`, $c_i = c_j + c_k$.
- Pour chaque nœud `Join(i,j,k)`, $c_i + c_j = c_k$.
- L'arête issue du nœud `START` ne s'exécute qu'une fois : $c_1 = 1$.
- L'arête entrant dans le nœuds `END` ne s'exécute qu'une fois : $c_n = 1$.

Démonstrations : en Annexes, section 10.2.

7.2 Construction du COP final

Soit $G = (V, E)$ un CFG qui satisfait H. Pour tout arête i, soit CSP_i le CSP construit comme dans la section 5. Soit $b_i = \mathrm{Sol}^*_{\mathcal{D}_i}(\mathrm{CSP}_i)$ où $\mathcal{D}_i$ est l'ensemble de variables de contrôles de l'arête i comme vu dans la section 6.

Définition: COP final Le COP final du CFG G est le quadruplet

- $X_{CSP} = \{c_i \mid i \in E\}$
- $\forall x \in X_{CSP}, D_x = \mathbb{N}$
- $C = \{c_i \leq b_i\} \cup K$
- $O = \sum_i c_i \times v_i$

Où O est la fonction à maximiser. Les v_i sont donnés par une analyse bas-niveau. K est l'ensemble de contraintes supplémentaire données par le CFG décrites section 7.1.

Proposition: Les c_i sont une solution des contraintes du COP final.

Démonstration: $\forall i \in E$, on a bien $c_i \in \mathbb{N}$, $c_i \leq b_i$ d'après la section 6. Et les c_i vérifient les contraintes de K d'après la section 10.2.

7.3 L'estimation de WCET

Le résultat du problème de maximisation est donc notre estimation sûre du WCET.

En effet, $c_1 \dots c_n$ est une solution des contraintes du COP final. Et on sait que $\sum_i c_i \times v_i = O(c_i)$ est une majoration du WCET. Alors $max_{(s_1 \dots s_n) \in \text{Sol}(CSOPfinal)} O(s_i) \geq O(c_i)$, donc $max_{(s_1 \dots s_n) \in \text{Sol}(CSOPfinal)} O(s_i) \geq$ WCET.

Alors résoudre le COP final en maximisant O donne bien une majoration sûre du WCET.

8 Implémentation et résultats

AbSolute [12] est le solveur par contraintes que nous nous proposons d'utiliser dans ce travail. C'est un solveur qui peut travailler sur des variables entières ou flottantes, pour traiter des contraintes arithémtiques génériques. La particularité d'AbSolute est que ce solveur est déjà issu de la coopération entre la programmation par contraintes et l'interprétation abstraite, puisqu'il utilise des domaines abstraits dans sa méthode de résolution [11]. Les solutions proposées seront ainsi des unions de domaines abstraits. AbSolute est capable de résoudre des CSPs en donnant les solutions, de compter les solutions d'un CSP et de minimiser une fonction objectif en donnant la valuation qui minimise cette fonction tout en satisfaisant les contraintes.

Cette section sera remplie plus tard.

9 Conclusion

La méthode proposée permet bien de récupérer des contraintes plus précises que l'interprétation abstraite seule, ce qui alors donne une approximation plus précise du nombre d'exécution de chaque c_i. L'avantage d'utiliser la programmation par contraintes en complément de l'interprétation abstraite réside dans la richesse de ses langages de contraintes : il est possible de traiter, dans le même problème, des contraintes linéaires, non linéaires, de congruence, etc.

De plus, chacune des techniques utilisées permet bien d'affiner l'approximation dans des cas où les autres ne le peuvent pas. En effet, dans bien des cas, seule l'interprétation abstraite permet de récupérer des bornes sur les variables qui évitent un nombre de solutions infini. Dès que le programme a dans ses tests ou ses affectations des contraintes non linéaires, seuls les arbres d'exécutions symboliques permettent de les exprimer pour réduire la taille de l'ensemble de solutions. Si il y a des variables non déterminantes, sans qu'une relation explicite soit écrite dans la syntaxe du programme, seule la détection de variables déterminantes permet de réduire grandement la majoration du nombre d'exécution d'une arête. Enfin, la loi des nœuds permet de pallier à d'éventuels manque de précision en un point de programme lorsque les points de programme précédents ou suivants ont été estimés plus précisément.

Bien sûr, ces résultats ne tiennent qu'au prix d'hypothèses fortes : les langages considérés sont déterministes, terminent et ne prennent pas d'entrée en paramètres, sur un langage restreint.

Ainsi, pour continuer ce travail, il serait possible de chercher à agrandir le langage utilisé (pour ajouter des pointeurs, des fonctions, des tableaux par exemple). De plus, si des solveurs par contraintes pouvaient résoudre des problèmes paramétrés, alors il serait possible d'adapter la méthode pour obtenir un résultat paramétré. Il est déjà possible d'adapter la méthode pour

que le programme prenne en entrée des paramètres, mais le résultat n'est pas paramétré. Enfin, l'ensemble de variables déterminantes proposé (les variables de contrôle) n'est pas toujours l'ensemble minimal de variables déterminantes. Il serait alors possible de s'intéresser à d'autres méthodes pour trouver de meilleurs ensembles de variables déterminantes.

Références

[1] http ://www.astree.ens.fr.

[2] S. Bygde. Static WCET Analysis based on Abstract Interpretation and Counting of Elements. PhD thesis, School of Innovation, Design and Engineering, Malardalen University, Dec. 2009.

[3] S. Bygde, A. Ermedahl, and B. Lisper. An efficient algorithm for parametric WCET calculation. Journal of Systems Architecture - Embedded Systems Design, 57(6) :614–624, 2011.

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

[5] P. Cousot and R. Cousot. Abstract interpretation : past, present and future. In Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014, pages 2 :1–2 :10, 2014.

[6] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst., 13(4) :451–490, 1991.

[8] T. M. Gawlitza and D. Monniaux. Invariant generation through strategy iteration in succinctly represented control flow graphs. Logical Methods in Computer Science, 8(3), 2012.

[9] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference, DAC '95, pages 456–461, New York, NY, USA, 1995. ACM.

[10] A. Miné. Domaines numériques abstraits faiblement relationnels. PhD thesis, Ecole Polytechnique, Dec. 2004.

[11] M. Pelleau. Domaines abstraits en programmation par contraintes. PhD thesis, Université de Nantes, 2012.

[12] M. Pelleau, A. Miné, C. Truchet, and F. Benhamou. A constraint solver based on abstract domains. In Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings, pages 434–454, 2013.

[13] F. Rossi, P. van Beek, and T. Walsh, editors. Handbook of Constraint Programming, volume 2 of Foundations of Artificial Intelligence. Elsevier, 2006.

[14] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem : Overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst., 7(3) :36 :1–36 :53, May 2008.