

IRIT - EQUIPE TRACES

RAPPORT DE STAGE - M2R IT SpÉ. SRLC

01 MARS 2014 - 03 AOÛT 2014

---

# Détermination de propriétés de flot de données pour l'amélioration du temps d'exécution pire-cas

---

*Auteur :*  
Jordy RUIZ

*Encadrant :*  
Hugues CASSÉ

Année 2013 - 2014



Université  
de Toulouse

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	WCET, les enjeux . . . . .	2
1.2	La recherche de chemins infaisables . . . . .	2
1.3	Langages supportés . . . . .	3
1.4	Restrictions . . . . .	5
<b>2</b>	<b>Contexte</b>	<b>6</b>
2.1	Choix du solveur SMT . . . . .	6
2.2	Représentation en graphe de flot de contrôle . . . . .	7
2.3	Les instructions sémantiques d'OTAWA . . . . .	8
2.4	Interprétation abstraite . . . . .	8
<b>3</b>	<b>Algorithme de recherche de chemins infaisables</b>	<b>8</b>
3.1	Représentation des prédicats . . . . .	8
3.2	Analyse d'un graphe de flot de contrôle . . . . .	13
3.3	Analyse des instructions sémantiques d'un bloc de base . . . . .	13
<b>4</b>	<b>Identification de chemins infaisables minimaux</b>	<b>17</b>
4.1	Algorithme complet naïf (en $\theta(n!)$ ) . . . . .	18
4.2	Algorithme incomplet (en $\theta(n)$ ) . . . . .	18
4.3	Algorithme complet (entre $\theta(n)$ et $\theta(2^n)$ ) - TODO . . . . .	19
<b>5</b>	<b>Traduction en contraintes ILP</b>	<b>19</b>
5.1	Cas avec deux if en séquence . . . . .	19
5.2	Cas avec trois if en séquence . . . . .	20
5.3	Généralisation . . . . .	22
5.4	Limitations des contraintes ILP . . . . .	22

# 1 Introduction

## 1.1 WCET, les enjeux

À Toulouse même, les besoins des industriels en termes de vérification de logiciel sont présents. Le calcul du pire temps d'exécution, ou WCET en anglais (Worst Case Execution Time) est un de ces multiples processus de vérification. La grande majorité des logiciels développés n'ont pas besoin de tels procédés de vérification, les bugs y sont gênants mais la plupart du temps un simple redémarrage suffit. Pour les systèmes dits **temps-réel critiques** (algorithme dans un satellite, frein d'une voiture, pilotage d'une rame de métro...), une erreur ou un échec peuvent entraîner de lourdes pertes économiques ou humaines : le plantage du logiciel d'un satellite nécessite des interventions très coûteuses, le temps de réponse exceptionnellement long d'un frein de voiture peut entraîner un accident...

Le problème est simple : il peut exister dans ces systèmes temps-réel critiques des cas extrêmement rares, indétectables par une batterie de tests, où le temps d'exécution explose anormalement. On pourrait par exemple imaginer un algorithme qui sauvegarde (disons hebdomadairement) le contenu de ses données sur un disque externe. Si tout est géré de manière synchrone et qu'un ordre critique arrive à ce moment-là, on pourrait se retrouver avec un temps d'exécution très élevé. On peut imaginer plein d'autres exemples (une plage de données très fragmentée qui entraînerait des défauts de page en masse, etc...).

Dans le cadre de mon étude, je fais abstraction de certains problèmes très bas niveau comme la gestion du cache ou le fonctionnement du pipeline et des prédictions de branchement. Ces problèmes ne sont pas ignorés, ils sont simplement traités par d'autres travaux, d'autres outils que les miens.

Le calcul du WCET consiste en une **surestimation** : on ne donne certes pas une mesure exacte du pire temps d'exécution, mais on garantit que cette mesure est en tout cas supérieure au pire temps réel. Le but de mon étude est d'améliorer, d'affiner, donc de **réduire** la surestimation du WCET calculée par notre outil OTAWA, tout en restant correct. On cherche à avoir le plus petit majorant possible.

La stratégie que je vais mettre en oeuvre pour la réduction de notre évaluation du WCET est la recherche de chemins infaisables.

## 1.2 La recherche de chemins infaisables

L'idée est la suivante : pour (sur)estimer le WCET, notre outil OTAWA fonctionne par énumération implicite des "chemins" du programme, ou IPET en anglais (Implicit Path Enumeration Technique), il prendra ensuite le temps d'exécution du chemin le plus coûteux du programme pour le calcul du WCET. Les programmes étudiés contiennent souvent de nombreux chemins infaisables (une étude avait observé 99.9% de chemins infaisables dans un programme de gestion des vitres d'une voiture [4]). Prenons par exemple le programme suivant :

```

Données : n
Résultat : k
si  $n > 10$  alors
|   k = 0 ; // (1)
|   sauvegarder();
sinon
|   k = 1 ; // (2)
fin
si  $n > 0$  alors
|   k = k + 1 ; // (3)
sinon
|   k = k - 1 ; // (4)
|   sauvegarder();
fin

```

**Algorithme 1 :** Exemple d'un programme avec chemin infaisable

Ce programme contient 4 chemins :

- Un chemin  $c_1$  qui passe par (1) et (3) ;
- Un chemin  $c_2$  qui passe par (1) et (4) ;
- Un chemin  $c_3$  qui passe par (2) et (3) ;
- Un chemin  $c_4$  qui passe par (2) et (4).

Le chemin  $c_2$ , qui passe par (1) et (4) est **infaisable**, puisqu'il implique  $n > 10$  et  $n \not> 0$ .

On pourrait maintenant imaginer que la fonction sauvegarder fait une opération relativement coûteuse, comme par exemple une écriture sur un fichier du système. Dans ce cas, le WCET calculé correspondra au temps d'exécution du chemin  $c_2$ , qui vaut à peu près à deux fois le temps d'exécution de la fonction sauvegarder.

Or, ce chemin est infaisable, c'est-à-dire qu'il ne sera jamais emprunté par le programme. Dans la réalité, ce programme n'exécuterait au plus qu'une fois cette fonction de sauvegarde, nous avons donc estimé le WCET (à peu près) au double du WCET réel !

En détectant ce chemin infaisable  $c_2$ , on pourrait indiquer à OTAWA de ne pas considérer ce chemin, et ainsi réduire effectivement l'estimation du WCET obtenue de moitié.

Le but de mon étude est donc de détecter ces chemins infaisables et de transmettre cette information à l'outil avant qu'il ne démarre son analyse pour l'estimation du WCET.

### 1.3 Langages supportés

Notre analyse portera sur des programmes écrits en **langage assembleur**, qui est le langage de programmation le plus proche de la machine. Le travail sur le code source a effectivement de multiples inconvénients : il faut prouver

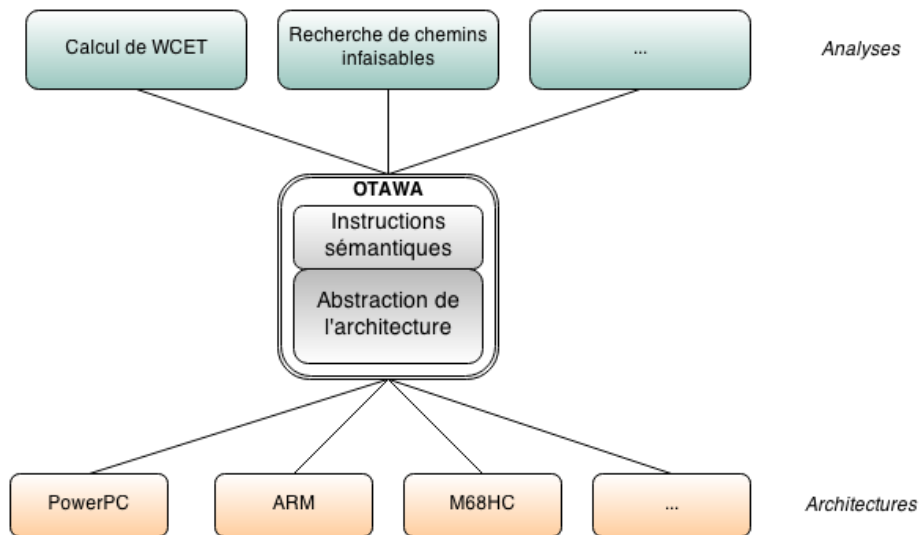


FIGURE 1 – OTAWA permet aux analyses de s’abstraire de l’architecture

le(s) compilateur(s) qui viennent avec (un travail très conséquent), et cela nous restreint aussi aux programmes dont nous disposons des sources complètes (certaines bibliothèques ne sont pas open-source).

Le langage assembleur, aussi appelé langage machine, est en revanche bien sûr beaucoup moins facile à analyser : les registres ne sont en général pas typés, et il n’y a pas réellement d’instructions `if`, `then`, `else` ou `while`, `for` (celle-ci est particulièrement dure à détecter dans le code assembleur) : il s’agit seulement d’instructions de comparaison et de branchements conditionnels.

Il n’y a pas non plus d’appels de fonctions avec des paramètres et des valeurs de retours explicites : il s’agit là encore de branchements (BL, Branch with Link dans le cas du langage d’assembleur ARM [7]) et on utilise un mélange de registres et de valeurs empilées pour les entrées et sorties de valeurs.

Un autre problème des langages machines est qu’il sont spécifiques à une architecture, les exécutables compilés à partir d’un même langage source seront différents selon qu’on est sur une architecture ARM ou PowerPC... Fort heureusement, notre outil OTAWA nous permet de nous **abstraire de l’architecture** en transformant les instructions du langage assembleur en **instructions sémantiques** génériques, universelles à tous les langages assembleur.

Ainsi, l’analyse que je développerai avec OTAWA supportera toutes les architectures supportées par OTAWA, et pour supporter une nouvelle architecture, il suffit de rajouter celle-ci dans OTAWA. OTAWA fonctionne comme une interface entre l’analyse et le code (écrit dans un langage spécifique à une architecture). Cette interface est schématisée sur la figure 1.

Voici, sans donner plus de détails pour le moment, à quoi ressemble la traduction de code assembleur (celui-ci a été généré avec gcc) en instructions

sémantiques. Les `tX` sont des variables temporaires utilisées pour la traduction en instructions sémantiques et les `?X` sont les registres. Pour chaque instruction assembleur, on trouve dessous une ou plusieurs instructions sémantiques équivalentes, en commentaire :

---

```
ldr r0, [pc, #20]
@ seti ?15, 0x8310
@ seti t2, 0x14
@ add t1, ?15, t2
@ load ?0, t1, uint32

mov r1, #0
@ seti ?1, 0x0

mov r2, r1
@ set t1, ?1
@ set ?2, t1

bl 8574
@ seti t1, 0x8574
@ seti ?14, 0x8318
@ branch t1
```

---

De plus amples explications à ce sujet seront données dans la suite de ce mémoire.

## 1.4 Restrictions

Nous travaillons dans le cadre de cette étude exclusivement sur des programmes sans boucles. Le problème de la gestion des boucles est un problème complexe qui ne pouvait pas se traiter en l'espace de quelques mois. Cette hypothèse n'est pas complètement irréaliste puisqu'il existe des programmes, voire même des langages sans boucles. Plus tard quand ce travail sera continué en thèse, il faudra les traiter mais nous ne savons pas encore à quel point ce traitement sera limité en efficacité.

Un autre vaste problème se pose, celui de la gestion de la mémoire. Il s'agit de se représenter dans le programme l'état de la mémoire (ou du moins les quelques parties sur lesquelles on sait quelque chose) et de déduire des propriétés sur la valeur du registre de destination d'un LOAD en mémoire. Le traitement des instructions mémoire est à l'heure que j'écris ces lignes quasiment terminé mais il reste assez limité.

Notre programme ne traite pour l'instant pas les instructions sémantiques d'OTAWA spécifiques aux "unsigned" (entiers représentés non signés). En effet, pour une même valeur binaire dans un registre, selon qu'on l'interprète comme un entier signé ou non, son interprétation peut être différente. Un exemple sur 4 bits est donné sur la figure 2.

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Unsigned	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
signed	0	1	2	3	4	5	6	7	-8	-7	-6	-5	-4	-3	-2	-1

FIGURE 2 – Différentes interprétations de la valeur d’un registre

Toutefois, l’interprétation de la valeur d’un registre est identique tant que le bit de poids fort est à 0, c’est à dire tant que la valeur interprétée appartient à l’intervalle  $\llbracket 0, 2^{n-1} - 1 \rrbracket$ , où  $n$  est le nombre de bits du registre.

Et bien sûr, toutes les instructions qui ne sont pas supportées par OTAWA (impossibles à traduire en expression sémantiques) ne sont pas non plus supportées par notre programme, même si ce type d’erreur est géré de manière à ne pas être critique : on va perdre de l’information et donc peut-être détecter moins de chemins infaisables, mais l’analyse n’échoue pas.

## 2 Contexte

### 2.1 Choix du solveur SMT

Pour le choix de notre solveur SMT, nous avons dressé un petit état de l’art en nous basant principalement sur des résultats de la SMT-COMP [8], une compétition entre solveurs, en éliminant d’office les solveurs qui ne présentaient pas d’API utilisable en C++ :

- Z3 [9]
  - Gagnant de la SMT-COMP 2011
  - Apparemment encore le plus performant en termes de temps d’exécution
  - Développé par Microsoft Research
  - Distribué sous la licence *Microsoft Research License Agreement Non-Commercial Use Only*
- CVC4 [10]
  - CVC3 a l’air insuffisant et limité mais le solveur a été globalement réécrit pour CVC4, bien qu’il semble y avoir une certaine continuité dans les *features* proposées
  - CVC4 a tourné sur de multiples benchmarks et a toujours eu de bons ou très bons résultats
  - Open source, sans restriction pour un usage commercial ou à fins de recherches
- MathSAT 5 [11]
  - API disponible seulement en C
  - Bons résultats pour MathSAT5-smtcomp12
  - “An SMT solver for Formal Verification”, ouverture possible sur la

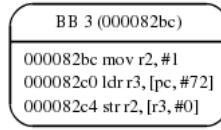


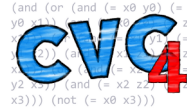
FIGURE 3 – Un exemple de bloc de base en langage machine ARM

vérification formelle

- Boolector [12]
  - API disponible seulement en C
  - Quelques très bons résultats à la SMT-COMP mais qui ne semblent pas porter sur des critères intéressants pour notre usage.
- SONOLAR [13]
  - Modeste dans ses performances
  - N’a été testé que sur benchmarks notés “BV”, restrictions aux booléens ?
- MISTRAL [14]
  - Pas évalué à la SMT-COMP 2012
- VeriT [15]
  - Pas évalué à la SMT-COMP 2012
- Barcelogic [16]
  - Pas évalué à la SMT-COMP 2012

Notre choix s’est finalement porté sur **CVC4**, pour ses bonnes performances, son API riche et bien documentée, et sa licence très libre.

Notre application utilise toutefois ses propres structures pour manipuler les prédicats, et le module qui fait l’interface avec le solveur SMT devrait être suffisamment indépendant du reste pour permettre de changer de solveur SMT si besoin sans trop de difficultés.



## 2.2 Représentation en graphe de flot de contrôle

Lors de notre analyse, nous considérerons le programme sous la forme d’un graphe connexe enraciné, appelé **graphe de flot de contrôle, ou CFG** (Control Flow Graph). Dans le cas des programmes sans boucles que nous traitons, il s’agit aussi d’un graphe acyclique.

Un CFG est donc un graphe dont les noeuds sont appelés des **blocs de bases**, et qui sont constitués d’une suite d’instructions exécutées *séquentiellement*. Il ne peut donc y avoir d’appels de fonctions, de conditions, de boucles, ou tout autre branchement à l’intérieur d’un bloc de base (excepté en dernière instruction). La figure 3 en montre un exemple.

Les arêtes représentent les chemins d’exécution du programme, elles correspondent à des branchements où à l’exécution séquentielle du programme. Il y a deux types de branchements : les branchements conditionnels et les branche-



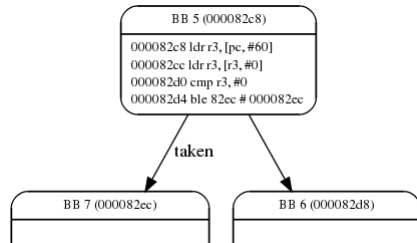


FIGURE 4 – Exemples de branchements conditionnels

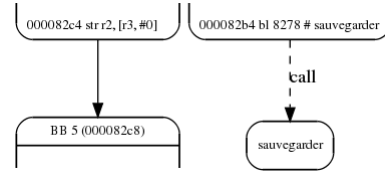


FIGURE 5 – Exemples de branchements inconditionnels

ments inconditionnels.

Parmi les **branchements conditionnels**, ceux qui correspondent à l’arc “pris”, c’est-à-dire au cas où la condition évoquée est vérifiée, sont annotés sur le CFG avec l’étiquette “taken”, l’arc dit “non pris” n’a conventionnellement pas d’étiquette.

Parmi les **branchements inconditionnels**, il peut s’agir de l’exécution séquentielle du programme ou d’un branchement non conditionnel (ce qui revient en assembleur à changer le registre pointeur d’instruction pc ou r15 pour l’ARM) auquel cas l’arc n’est pas annoté, ou il peut s’agir d’un appel de fonction (*Branch with Link* en assembleur), auquel cas l’arc est dessiné en pointillés et annoté “call”.

Les figures 4 et 5 montrent des exemples de ces branchements.

Prenons pour exemple l’algorithme 1 que nous avons présenté plus haut. Le CFG en pseudo-code correspondant est exhibé sur la figure 6. On y voit bien apparaître les 4 chemins cités précédemment.

La figure 7 montre le CFG généré par OTAWA à partir du code ARM correspondant à l’algorithme 1 (compilé à partir d’un fichier C). On y observe à peu près la même structure que sur le CFG en pseudo-code, avec des calls à la fonction sauvegarder et des instructions ARM plus volumineuses à l’intérieur des blocs.

## 2.3 Les instructions sémantiques d’OTAWA

## 2.4 Interprétation abstraite

# 3 Algorithme de recherche de chemins infaisables

## 3.1 Représentation des prédicats

Nous utilisons nos propres structures de prédicats, appropriées à l’usage qui leur est destiné. En voici la grammaire abstraite :

Un prédicat est composé de deux opérandes et d’un opérateur de comparaison :

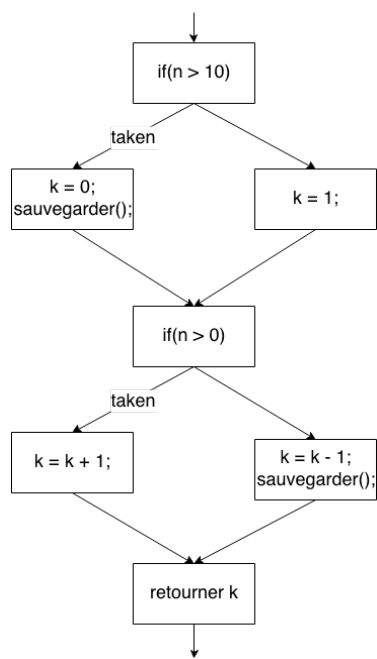


FIGURE 6 – CFG de l'algorithme 1

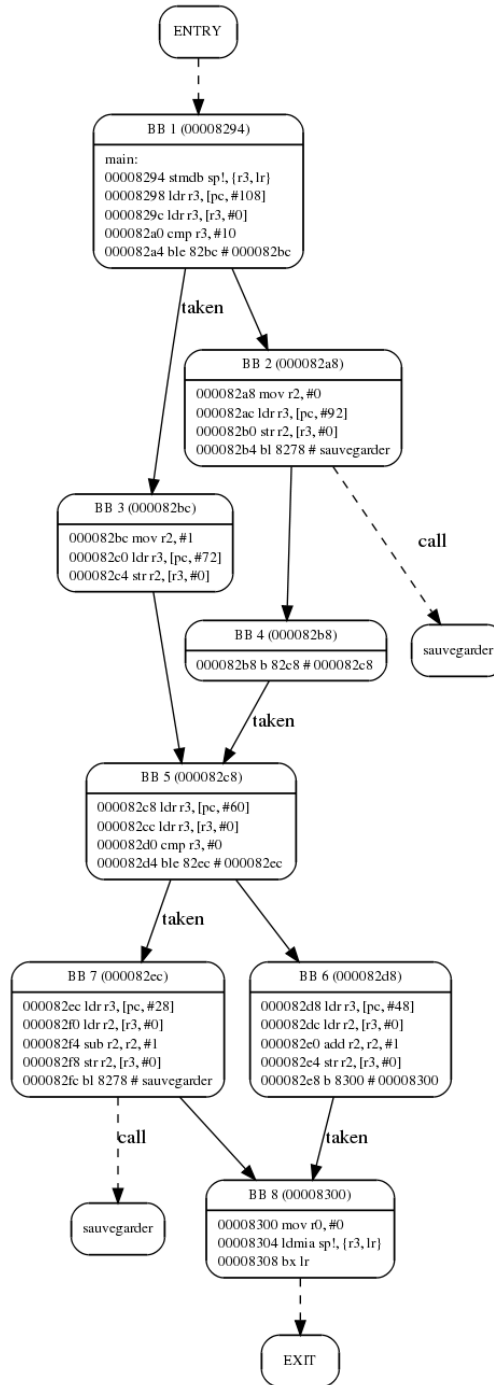


FIGURE 7 – CFG généré par OTAWA

Instruction	Sémantique
NOP	(rien)
BRANCH TRAP CONT	Indicateurs du flot du programme
IF cond sr jump	si la condition cond sur le registre sr est vraie, continuer, sinon sauter jump instructions
LOAD reg addr type	$\text{reg} \leftarrow \text{MEM}_{\text{type}}$
STORE reg addr type	$\text{MEM}_{\text{type}} \leftarrow \text{reg}$
SCRATCH d	$d \leftarrow \top$ (invalidation)
SET d a	$d \leftarrow a$
SETI d cst	$d \leftarrow \text{cst}$
SETP d cst	$\text{page}(d) \leftarrow \text{cst}$
CMP d a b	$d \leftarrow a \sim b$
CMPU d a b	$d \leftarrow a \sim_{\text{unsigned}} b$
ADD d a b	$d \leftarrow a + b$
SUB d a b	$d \leftarrow a - b$
SHL d a b	$d \leftarrow \text{unsigned}(a) \ll b$
SHR d a b	$d \leftarrow \text{unsigned}(a) \gg b$
ASR d a b	$d \leftarrow a \gg b$
NEG d a	$d \leftarrow -a$
NOT d a	$d \leftarrow \neg a$
AND d a b	$d \leftarrow a \& b$
OR d a b	$d \leftarrow a   b$
XOR d a b	$d \leftarrow a \oplus b$
MUL d a b	$d \leftarrow a \times b$
MULU d a b	$d \leftarrow \text{unsigned}(a) \times \text{unsigned}(b)$
DIV d a b	$d \leftarrow a / b$
DIVU d a b	$d \leftarrow \text{unsigned}(a) / \text{unsigned}(b)$
MOD d a b	$d \leftarrow a \% b$
MODU d a b	$d \leftarrow \text{unsigned}(a) \% \text{unsigned}(b)$
SPEC	(instruction spéciale non supportée par OTAWA)

*En gris : les instructions qui ne sont pas (encore) traitées par notre analyse.*

FIGURE 8 – Liste des instructions sémantiques d’OTAWA

**Prédicat :** Expression  $\times$  Comparateur  $\times$  Expression

**Comparateur :**

- | EQ, // =
- | NE, //  $\neq$
- | LE, //  $\leq$
- | LT // <

Pour limiter le nombre de cas à traiter, nous n'avons pas inclus les opérateurs  $>$  et  $\geq$  (on peut les retrouver en inversant l'ordre des opérandes avec  $<$  et  $\leq$ ). Ce genre d'astuce est aussi utilisé dans l'API fournie par le solveur SMT CVC4, à cela près qu'ils préfèrent exclure  $\leq$  et  $<$ .

Une expression peut être une constante, une variable, une valeur en mémoire, ou une expression arithmétique composée d'un opérateur arithmétique et de deux opérandes, elles-mêmes des expressions :

**Expression :**

- | Const( $k : \mathbb{Z}$ )
- | Var( $id : \text{Variable}$ )
- | Mem( $addr : \text{Adresse}$ )
- | - Expression
- | Expression + Expression
- | Expression - Expression
- | Expression  $\times$  Expression
- | Expression / Expression
- | Expression *mod* Expression

Nous ne supportons pour le moment que les opérations sur les entiers signés ( $\mathbb{Z}$ ), donc pas d'opérateurs "non signés" (sur  $\mathbb{N}$ ), ou d'opérateurs logiques (sur  $\mathbb{Z}/2\mathbb{Z}$ ).

Une variable peut être un registre temporaire (un entier strictement positif) ou un registre (un entier positif). La quantité de registres est généralement très limitée mais comme nous travaillons indépendamment de l'architecture, nous ne pouvons pas connaître cette limitation (et nous n'en avons pas besoin).

**Variable :**

- | Temporaire( $k : \mathbb{N}^*$ )
- | Registre( $k : \mathbb{N}$ )

Une adresse peut être absolue (par exemple, `0x8014`), ou relative à une variable (par exemple `r13 + 4`).

**Adresse :**

- | Absolu( $k : \mathbb{Z}$ )
- | Relatif( $reg : \text{Registre}, \text{offset} : \mathbb{Z}$ )

A chaque prédicat nous lui associerons un arc du CFG, qui correspondra, sur le chemin emprunté, à l'arc immédiatement après le bloc de base qui contenait l'instruction qui a généré le prédicat en question.

Nous ne transportons donc pas exactement une liste de prédicats (qui correspond à une conjonction de prédicats), mais plutôt une liste de couples (Prédicat, Edge) qui contiennent des informations sur l'origine de leur prédicats. Cela nous permettra ensuite, une fois que nous aurons trouvé une insatisfiabilité, de remonter jusqu'aux arêtes responsables du CFG et d'identifier le chemin infaisable. Par exemple, à partir de la liste :

$$\{(x = 0)^{1 \rightarrow 2}, (y = x)^{3 \rightarrow 5}, (y > 10)^{3 \rightarrow 5}\}$$

on pourra en déduire que le chemin qui passe par les arcs  $\{1 \rightarrow 2, 3 \rightarrow 5\}$  est infaisable.

### 3.2 Analyse d'un graphe de flot de contrôle

Comme nous travaillons sans boucles, l'algorithme ci-dessous n'est pas correct, ou du moins ne terminera pas si on le fait fonctionner sur un CFG avec boucles. Nous initialisons d'abord l'algorithme de parcours de graphe de contrôle avec le premier bloc de base du CFG, qui nous est donné par OTAWA.

Voici donc l'algorithme en pseudo-code, qui prend en paramètre un bloc de base :

```

Données : bb : BasicBlock, lpreds : LabelledPredicate list
Résultat : Chemins infaisables trouvés
si bb est un bloc de sortie (EXIT) alors
  | retourner [] // Fin de l'analyse de ce chemin
fin
Appeler le solveur SMT avec lpreds ;
si on a trouvé une insatisfiabilité alors
  | Extraire la liste des chemins infaisables ;
  | retourner cette liste // Fin de l'analyse de ce chemin
fin
preds = parseBasicBlock(bb) ; // Analyse linéaire des instructions
sémantiques
pour tous les arcs sortants edge de bb faire
  | Labeller les prédicats preds avec l'arc edge comme étiquette ;
  | Ajouter ces prédicats labellés à lpreds ;
  | Appel récursif de cette fonction avec pour paramètres (
    | le bloc de base vers lequel edge pointe,
    | la nouvelle liste lpreds
  | );
fin

```

### 3.3 Analyse des instructions sémantiques d'un bloc de base

Le programme parcourt les blocs de base linéairement en mettant à jour la liste de prédicats.

Nous définissons d’abord une fonction d’invalidation d’une variable dans la liste de prédicats, qui consiste à supprimer tous les prédicats qui utilisent cette variable. En effet, nous allons rencontrer des instructions “destructrices”, c’est-à-dire des instructions qui vont rendre invalides toutes les propriétés que nous avons sur la variable en question. Par exemple, si nous rencontrons une instruction qui met  $r0$  à 0, alors il faudra supprimer les prédicats précédemment créés  $r0 = r1$  et  $r0 = 2$ .

---

```
invalidate var p =
  {predicate ∈ p | var ∉ predicate}
```

---

Les seules fois où l’invalidation ne sera pas nécessaire pour une instruction qui modifie une variable seront celles où la modification en question est **bijective**. Par exemple, une instruction qui incrémente une variable de 1 (abstraite par la fonction  $ajouter_1$  ici) est bijective :

$$ajouter_1 : \mathbb{Z} \rightarrow \mathbb{Z}$$

$$x \mapsto x + 1$$

$$ajouter_1^{-1} : \mathbb{Z} \rightarrow \mathbb{Z}$$

$$x \mapsto x - 1$$

Si par exemple nous rencontrons une instruction qui nous demande d’incrémenter de 1 la variable  $r0$ , nous pourrions simplement remplacer  $r0$  par  $r0 - 1$  dans les prédicats  $r0 = r1$  et  $r0 = 2$ . Nous obtenons les nouveaux prédicats  $r0 - 1 = r1$  et  $r0 - 1 = 2$  (c’est-à-dire  $r0 = 3$ ).

Nous avons également implémenté des stratégies pour essayer de préserver l’information autant que possible lors de l’invalidation d’une variable. Par exemple, si on a un prédicat  $r0 = r1$  et que l’on doit invalider  $r0$ , on se contentera de supprimer ce premier prédicat et remplacer toutes les occurrences de  $r0$  par  $r1$ .

Il s’agit maintenant de définir l’effet de la lecture de chaque instruction sur la liste de prédicats. Exemple sur un premier cas trivial :

---

```
t [NOP] p =
  p (* rien ne change *)
```

---

La fonction de traduction  $t$  opère sur une instruction (ici NOP) et la liste de prédicats  $p$ .

Regardons maintenant comment est traitée l’instruction d’assignation d’une variable à une autre variable (SET) :

---

```
t [SET d a] p =
  (d = a) @ (invalidate d p)
```

---

Un nouveau prédicat  $d = a$  est généré après qu'on ait invalidé  $d$ , c'est-à-dire qu'on ait supprimé tous les prédicats qui contiennent  $t$ . De même pour l'instruction SETI qui assigne une constante à une variable, **puisque pour tout  $n$ , la fonction  $assigner_n$  définie ci-dessous n'est pas bijective :**

$$assigner_n : \mathbb{Z} \rightarrow \mathbb{Z}$$

$$x \mapsto n$$

**On ne peut donc pas en extraire une fonction inverse définie sur tout  $\mathbb{Z}$ .**

---

```
t [SETI d cst] p =
  (d = a) @ (invalidate d p)
```

```
t [CMP d a b] p =
  (d = a ~ b) @ (invalidate d p)
```

---

Cet opérateur  $\sim$  bien spécifique aux langages d'assembleur sert à se souvenir que  $d$  contient des informations sur la comparaison entre  $a$  et  $b$ .

Pour la suite nous allons avoir besoin de la fonction `update` qui sert à remplacer toutes les occurrences d'une variable `var` par l'expression `expr` dans la liste de prédicats `p` :

---

```
update var expr p =
  {predicate[expr / var] | predicate ∈ p}
```

---

où la syntaxe `predicate[expr / var]` dénote ici le prédicat où le terme `var` est remplacé par l'expression `expr`.

**Nous distinguerons toujours dans les instructions ci-dessous plusieurs cas, selon que  $d$  soit une variable distincte de  $a$  et  $b$  ou non. Nous allons ensuite modifier les prédicats qui contiennent  $d$  en fonction des résultats de ces tests :**

- Dans le cas où  $d$  n'est pas la même variable que  $a$  ou  $b$ , on invalide simplement les prédicats qui contiennent  $d$  et on rajoute un nouveau prédicat exprimant ce qu'on sait sur le nouveau  $d$ .
- Dans l'autre cas, il faut mettre à jour les prédicats qui contiennent  $d$  pour prendre en compte la modification appliquée sur  $d$ .

---

```
t [ADD d a b] p = % TODO: on a oublié un cas, a changer dans le code aussi !!!
  if (d = a) then (* d <- d+b *)
    (update d (d - b) p)
  else if (d = b) (* d <- a+d *)
    (update d (d - a) p)
  else
    (d = a + b) @ (invalidate d p)

t [SUB d a b] p =
  if (d = a) then
```



```

    if (d = b) then (* d <- d-d *)
      (d = 0) @ (invalidate d p) % TODO a changer dans le vrai code !!!
    else (* d <- d-b *)
      (update d (d+b) p)
  else
    if (d = b) then (* d <- a-d *)
      (update d (a-d) p)
    else (* d <- a-b *)
      (d = a - b) @ (invalidate d p)

t [MUL d a b] p =
  if (d = a) then
    if (d = b) then (* d <- d*d *)
      (* impossible de remplacer d par  $\sqrt{d}$ , on invalide *)
      (0 <= d) @ (invalidate d p)
    else (* d <- d*b *)
      (* on rajoute un predicat pour indiquer que d est
         divisible par b *)
      (d % b = 0) @ (update d (d/b) p)
  else
    if (d = b) then (* d <- a*d *)
      (d % a = 0) @ (update d (d/a) p)
    else (* d <- a*b *)
      (d = a * b) @ (invalidate d p)

t [DIV d a b] p =
  if (d = a) then
    if (d = b) then (* d <- d/d *)
      (d = 1) @ (invalidate d p)
    else (* d <- d/b *)
      (* impossible de remplacer d par (d*b),
         on a perdu de l'information ! *)
      (invalidate d p)
  else
    if (d = b) then (* d <- a/d *)
      (invalidate d p)
    else (* d <- a/b *)
      (d = a / b) @ (invalidate d p)

```

---

Pour illustrer le problème de l'instruction DIV, prenons le cas où l'on a {t1 = 7, t2 = 3} et une instruction [DIV t1 t1 t2]. En remplaçant t1 par t1 \* t2, on obtiendrait {(t1 \* t2 = 7), (t2 = 3)}, c'est-à-dire (t1 \* 3 = 7), ce qui est impossible puisqu'on travaille sur des entiers! **On fait donc le choix d'invalider t1.**

---

```

t [MOD d a b] p =

```

```

if (d = a or d = b)
  (invalidate d p)
else
  (d = a % b) @ (invalidate d p)

```

---

Nous utiliserons dans la suite une fonction `eval` qui cherche la valeur **constante** pour une variable. La fonction `eval` parcourt donc la liste des prédicats à la recherche de prédicats du type `(var = 2)` permettant d'identifier la valeur de `var`.

Les scénarios où on ne peut pas identifier une valeur constante pour le nombre de bits à décaler dans le cas d'une instruction de décalage logique sont rares. Nous n'avons de toutes façons pas la possibilité de représenter des prédicats du type  $\dots = 2^{var}$  dans notre structure, c'est pourquoi nous cherchons une valeur constante pour `b` dans l'instruction ASR.

---

```

t [ASR d a b] p =
  let b_val = eval b in
  if b_val = undefined then
    (invalidate d p)
  else
    let factor = 2 ** b_val in
    if (d = a) then (* d <- d>>b *)
      (update d (d / factor) p)
    else (* d <- a>>b *)
      (d = a / factor) @ (invalidate d p)

t [NEG d a] =
  if (a = d) (* d <- -d *)
    (update d (-d) p)
  else (* d <- -a *)
    (d = -a) @ (invalidate d p)

```

---

## 4 Identification de chemins infaisables minimaux

Une fois qu'un appel au solveur SMT avec une conjonction de prédicats en paramètre `a` renvoyé une insatisfiabilité (code "UNSAT"), le travail pour trouver le chemin infaisable en question n'est pas encore terminé. En effet, nous n'avons certes pas la liste complète de tous les arcs du chemin que nous avons parcouru, et seulement la liste des *arcs qui ont généré un prédicat appartenant à la liste de prédicats que le SMT a détectée insatisfiable*, mais c'est loin d'être minimal.

Nous aurons souvent une liste d'une dizaine de prédicats qui ont peu de rapport entre eux et deux ou trois prédicats qui à eux seuls ont causé l'insatisfiabilité. Le but de cet étape de l'algorithme est d'**identifier les prédicats responsables de l'insatisfiabilité** pour inclure le moins d'arcs possibles dans le chemin insatisfiable renvoyé.

L'intérêt de faire cette réduction est que, comme nous le verrons plus loin, l'exploitabilité des chemins infaisables trouvés (pour réduire l'estimation du WCET) diminue plus le chemin infaisable contient d'arcs.

Pour cela, nous présentons trois algorithmes :

#### 4.1 Algorithme complet naïf (en $\theta(n!)$ )

Cet algorithme naïf va tout simplement, à partir d'une liste de prédicats  $\{p_1^{c_1}, p_2^{c_2}, \dots, p_n^{c_n}\}$ , tester la satisfiabilité des  $n!$  combinaisons possibles (par exemple,  $\{p_1, p_3\}$ ).

On relève ensuite la liste des conjonctions de prédicats qui ont donné lieu à des insatisfiabilités, par exemple pour  $n = 4$  :

- $\{p_1^{c_1}, p_2^{c_2}, p_3^{c_3}\}$
- $\{p_1^{c_1}, p_2^{c_2}\}$
- $\{p_1^{c_1}, p_4^{c_4}\}$

On simplifie ensuite (à l'aide d'une disjonction globale) pour obtenir un nombre minimal de conjonctions de prédicats. Pour l'exemple précédent, on obtiendrait :

- $\{p_1^{c_1}, p_2^{c_2}\}$
- $\{p_1^{c_1}, p_4^{c_4}\}$

On a donc obtenu l'information " $\{c_1, c_2\}$  et  $\{c_1, c_4\}$ " sont des chemins infaisables, au lieu de " $\{c_1, c_2, c_3, c_4\}$ " est un chemin infaisable. C'est beaucoup mieux !

Le problème de cet algorithme est bien sûr sa complexité exponentielle qui fait qu'il explose sur des grosses listes : il fait 3,6 millions d'appels au solveur SMT pour une liste de 10 prédicats !

#### 4.2 Algorithme incomplet (en $\theta(n)$ )

En partant d'une liste de prédicats  $\{p_1^{c_1}, p_2^{c_2}, \dots, p_n^{c_n}\}$ , on parcourt une fois la liste en testant pour chaque prédicat  $p_i$  :

- Si, en supprimant  $p_i$  de la liste des prédicats, la liste reste insatisfiable, alors on supprime  $p_i$  définitivement.
- Si, en supprimant  $p_i$  de la liste des prédicats, la liste devient insatisfiable, alors on laisse  $p_i$  dans la liste.

Après une seule passe, on s'arrête et on obtient un **unique** chemin infaisable. L'inconvénient de cette méthode très rapide est que l'on peut manquer des chemins infaisables que l'on pourrait ne pas retrouver ailleurs. Le chemin retenu est arbitraire et décidé par l'ordre d'analyse. Par exemple, nous avons la liste de prédicats suivantes :

$$\{x = 0^{c_1}, x > 1^{c_2}, x > 2^{c_3}\}$$

L'algorithme essaye de supprimer  $x = 0^{c_1}$ , mais  $\{x > 1^{c_2}, x > 2^{c_3}\}$  est satisfiable, donc le premier prédicat est conservé.

L'algorithme essaye de supprimer le deuxième prédicat  $x > 1^{c_2}$ , et obtient  $\{x = 0^{c_1}, x > 2^{c_3}\}$ , qui est insatisfiable. Par conséquent, il supprime définitivement ce prédicat.

La troisième passe ne donne rien, vu que  $x = 0^{c_1}$  tout seul est satisfiable.

On obtient ainsi un seul chemin infaisable,  $\{c_1, c_3\}$ , alors qu'il y en a en réalité deux,  $\{c_1, c_3\}$  et  $\{c_1, c_2\}$  !

La complexité est en revanche très bonne,  $\theta(n)$ . L'algorithme suivant a pour but de résoudre ce problème tout en conservant une complexité acceptable.

### 4.3 Algorithme complet (entre $\theta(n)$ et $\theta(2^n)$ ) - TODO

dire que même si on a  $\theta(2^n)$  pire cas c'est très bien en cas moyen

## 5 Traduction en contraintes ILP

Une fois les chemins infaisables trouvés, il faut générer des contraintes ILP sur le CFG utilisables par OTAWA pour affiner (du moins on l'espère) l'estimation du WCET.

### 5.1 Cas avec deux if en séquence

Commençons avec un exemple simple de deux `if... then... else ...` en séquence (figure 9). La première condition est notée  $x$  et la deuxième  $y$ , nous avons étiquetés les 4 arcs correspondants aux branchements conditionnels :  $x$ ,  $\neg x$ ,  $y$ ,  $\neg y$ .

Nous notons  $n_x$ ,  $n_{\neg x}$ ,  $n_y$  et  $n_{\neg y}$  les variables qui représentent le nombre de fois que sont exécutés les arcs respectifs  $x$ ,  $\neg x$ ,  $y$ , et  $\neg y$ .

Le problème est le suivant : nous savons que le chemin  $x$ - $y$  (je note ainsi le seul chemin qui passe par les arcs notés  $x$  et  $y$ ) est infaisable. Quelles contraintes ILP peut-on en déduire ?

Attention, nous ne cherchons pas à trouver une équivalence (chemin infaisable  $\Leftrightarrow$  contraintes ILP) mais simplement une implication (chemin infaisable  $\Rightarrow$  contraintes ILP). Nous verrons plus loin que la perte d'information lors du passage aux contraintes ILP est irréversible et qu'il est parfois impossible d'avoir une équivalence.

Pour un tel chemin infaisable  $x$ - $y$ , l'unique contrainte ILP que nous générons est la suivante :

$$n_x \leq n_{\neg y}$$

Intuitivement, cette contrainte est valide (c'est-à-dire qu'elle est toujours vraie si  $x$ - $y$  est un chemin infaisable) :

En effet, si  $x-y$  est un chemin infaisable, alors tous les chemins qui passent par  $x$  passeront forcément par  $\neg y$ . Donc le nombre de fois que l'arc  $\neg y$  est exécuté, noté  $n_{\neg y}$ , est nécessairement *supérieur ou égal* au nombre de fois que l'arc  $x$  est exécuté, c'est-à-dire  $n_x$  (c'est supérieur s'il existe des chemins  $\neg x-y$ ). Cela se traduit bien par

$$n_x \leq n_{\neg y}$$

On pourrait observer que de même, puisque  $x-y$  est un chemin infaisable, alors tous les chemins qui passent par  $y$  passeront forcément par  $\neg x$ , et donc que :

$$n_y \leq n_{\neg x}$$

Nous allons maintenant montrer que  $n_x \leq n_{\neg y} \Leftrightarrow n_y \leq n_{\neg x}$  et que par conséquent générer ces deux contraintes au lieu d'une seule est *superflu*. Pour cela nous allons utiliser la propriété de flots de graphes qui dit que

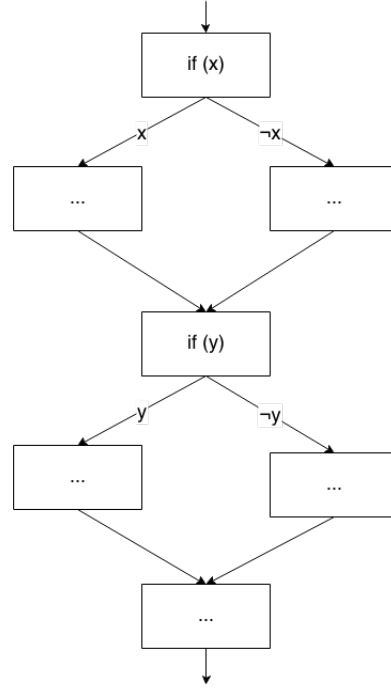


FIGURE 9 – Premier exemple : deux if en séquence

où  $N$  est une constante (qui représente le nombre total d'exécutions du programme). Nous avons ainsi :

$$\begin{aligned}
 & n_x \leq n_{\neg y} \\
 \Leftrightarrow & n_x + (n_{\neg x} - n_{\neg x}) \leq n_{\neg y} + (n_y - n_y) \\
 \Leftrightarrow & (n_x + n_{\neg x}) - n_{\neg x} \leq (n_{\neg y} + n_y) - n_y \\
 \Leftrightarrow & N - n_{\neg x} \leq N - n_y \\
 \Leftrightarrow & -n_{\neg x} \leq -n_y \\
 \Leftrightarrow & n_y \leq n_{\neg x}
 \end{aligned}$$

■

## 5.2 Cas avec trois if en séquence

Voyons maintenant ce qui se passe lorsque l'on rajoute un autre if... then... else ... en séquence, ainsi que deux arcs notés  $z$  et  $\neg z$ .

Supposons que  $x-y-z$  soit un chemin infaisable. Intuitivement, on en déduit que si un chemin passe par  $x$ , alors il passera par  $\neg y$  ou par  $\neg z$ . C'est-à-dire, exprimé en termes de nombre d'exécution des arcs :

$$n_x \leq n_{\neg y} + n_{\neg z}$$

De manière similaire, on trouve aussi que “x-y-z chemin infaisable” entraîne :

$$n_y \leq n_{\neg x} + n_{\neg z}$$

$$n_z \leq n_{\neg x} + n_{\neg y}$$

Une fois de plus, nous allons prouver que ces trois assertions sont équivalentes. Nous avons toujours la propriété de flots de graphes (cette sorte de loi des noeuds) qui dit que :

$$n_x + n_{\neg x} = n_y + n_{\neg y} = n_z + n_{\neg z} = N$$

Montrons alors ces équivalences :

$$\boxed{n_x \leq n_{\neg y} + n_{\neg z}}$$

$$\Leftrightarrow n_x \leq n_{\neg y} + (n_y - n_y) + n_{\neg z}$$

$$\Leftrightarrow n_x \leq (n_{\neg y} + n_y) - n_y + n_{\neg z}$$

$$\Leftrightarrow n_x \leq N - n_y + n_{\neg z}$$

$$\Leftrightarrow n_x + n_y \leq N + n_{\neg z}$$

$$\Leftrightarrow n_y \leq N - n_x + n_{\neg z}$$

$$\Leftrightarrow n_y \leq (n_x + n_{\neg x}) - n_x + n_{\neg z}$$

$$\Leftrightarrow \boxed{n_y \leq n_{\neg x} + n_{\neg z}}$$

$$\Leftrightarrow n_y \leq n_{\neg x} + n_{\neg z}$$

$$\Leftrightarrow n_y \leq n_{\neg x} + (N - n_z)$$

$$\Leftrightarrow n_z \leq n_{\neg x} + N - n_y$$

$$\Leftrightarrow \boxed{n_z \leq n_{\neg x} + n_{\neg y}}$$

■

Générer une seule de ces trois contraintes est donc suffisant. On peut remarquer que cela entraîne également la propriété suivante :

$$\left. \begin{array}{l} n_x \leq n_{\neg y} + n_{\neg z} \\ n_y \leq n_{\neg x} + n_{\neg z} \\ n_z \leq n_{\neg x} + n_{\neg y} \end{array} \right\} \Rightarrow n_x + n_y + n_z \leq 2(n_{\neg x} + n_{\neg y} + n_{\neg z})$$

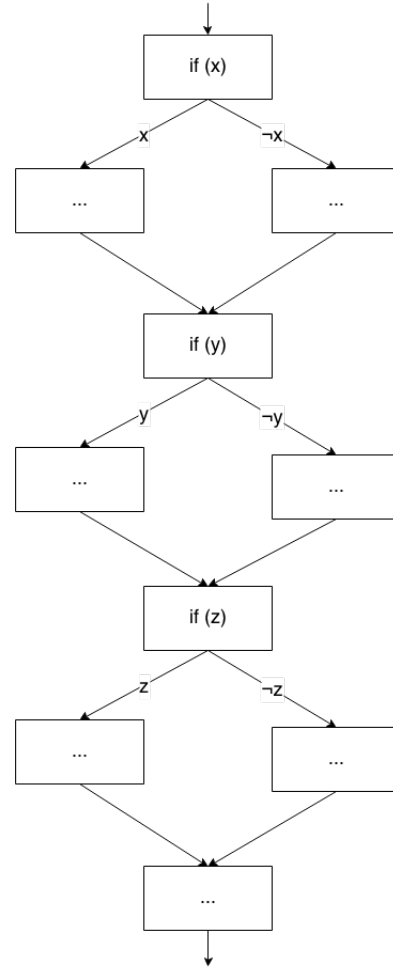


FIGURE 10 – Deuxième exemple : trois if en séquence

Nous voyons apparaître un motif qui se répète, que nous allons tenter de généraliser dans la prochaine section.

### 5.3 Généralisation

**Theorème 1** *Pour un CFG composé de  $k$  `if... then... else...` en séquence, où les arcs pris (`then`) de chacune de ses conditions sont notés  $x_1, x_2, \dots, x_k$  et les arcs non pris (`else`) sont notés  $\neg x_1, \neg x_2, \dots, \neg x_k$ , on note le nombre de fois qu'un arc  $x_i$  (resp.  $\neg x_i$ ) est emprunté  $n_{x_i}$  (resp.  $n_{\neg x_i}$ ).*

*Si  $x_1 - x_2 - \dots - x_k$  est un chemin infaisable, alors on en déduit les  $k$  contraintes ILP suivantes :*

$$\forall i \in \llbracket 1, k \rrbracket, n_{x_i} \leq \sum_{j \in \llbracket 1, k \rrbracket \setminus \{i\}} n_{\neg x_j}$$

*et chacune de ces contraintes est équivalente. De plus, il est immédiat que :*

$$\sum_{i \in \llbracket 1, k \rrbracket} n_{x_i} \leq (k - 1) \sum_{i \in \llbracket 1, k \rrbracket} n_{\neg x_i}$$

On pourra remarquer que les CFG contenant des conditions du type `if... then...` sans `else...` ne diffèrent pas du cas étudié (avec `else`) puisqu'il y a toujours deux arcs sortant du bloc de base qui calcule la condition, la différence étant que l'arc non pris pointe sur le bloc de fin de condition au lieu de pointer sur un bloc qui lui-même pointerait vers ce bloc de fin de condition.

### 5.4 Limitations des contraintes ILP

Les contraintes ILP manquent toutefois d'expressivité pour représenter toute l'information contenue dans la notion de "chemin infaisable", comme expliqué dans un article du WCET 2005 workshop :

"This is because the usual ILP formulation introduces formal variables for the execution counts of the nodes and edges in the

Control Flow Graph (CFG) of the program. Since the variables denote aggregate execution counts of basic blocks, it is not possible to express certain infeasible path patterns as constraints on these variables.” [4]

En effet, si on considère l’exemple de la figure 11, où les valeurs entre parenthèses sur chaque arc représentent le nombre de fois que l’arc est exécuté dans le scénario considéré, il est impossible de déterminer si l’on a un chemin infaisable (en tout cas un chemin qui n’est jamais pris) ou non à partir des seules variables  $n_x$ ,  $n_{\neg x}$ ,  $n_y$ ,  $n_{\neg y}$  !

Il se pourrait, par exemple, que les 20 exécutions du CFG consistent en 10 fois le chemin  $x \rightarrow \neg y$  et 10 fois le chemin  $\neg x \rightarrow y$ . Dans ce cas, le chemin  $x \rightarrow y$  n’est effectivement jamais pris.

Mais il se pourrait aussi, que l’on ait 10 exécutions qui passent par le chemin  $x \rightarrow y$  !

Dans ces conditions, on ne peut pas signifier à OTAWA si ce scénario (avec ces comptes d’exécution) est viable ou non, et il sera donc inclus dans le calcul du WCET. Les contraintes ILP ne permettent pas d’exprimer tout ce que l’on sait à partir de l’hypothèse “chemin infaisable”.

Il faut prendre conscience de cette limitation à l’étape “utilisation des chemins infaisables trouvés”, de cette perte d’information, mais cela ne signifie pas nécessairement que le choix de la génération de contraintes ILP est mauvais. L’autre alternative serait de faire de la réécriture de graphe pour faire apparaître (ou plutôt disparaître en l’occurrence) les chemins infaisables sur le CFG, mais c’est bien plus coûteux, et aussi plus compliqué.

Étant donné qu’un membre de l’équipe TRACES travaille justement sur la réécriture de CFG, une collaboration sur ce sujet est envisageable pour ma thèse.

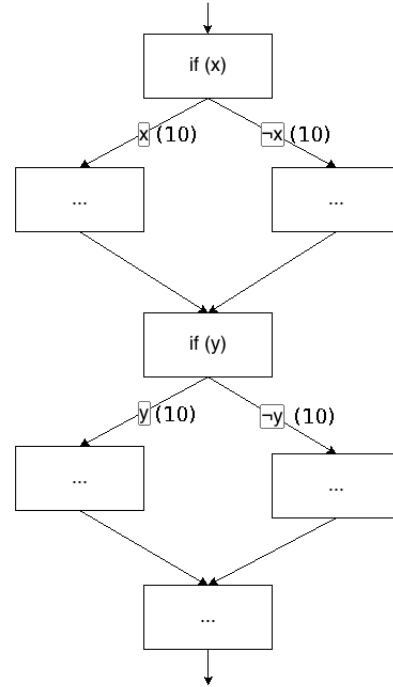


FIGURE 11 – Exemple du manque d’expressivité des contraintes ILP



## Conclusion

Petit récap...

Tout ce qu'il y a à faire en thèse !

Parler un peu de la théorie, analyse statique...

## Références

- [1] Patrick Cousot and Rahida Cousot. 1992. *Abstract interpretation and application to logic programs*. J. Log. Program. 13, 2-3 (July 1992), 103-179. [http://dx.doi.org/10.1016/0743-1066\(92\)90030-7](http://dx.doi.org/10.1016/0743-1066(92)90030-7)
- [2] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.
- [3] *Electronic Notes in Theoretical Computer Science (Entcs)*. Elsevier Sci. Pub. B. V., Amsterdam, The Netherlands, The Netherlands. <http://dx.doi.org/10.1016/j.entcs.2010.09.014>
- [4] T. Chen, T. Mitra, A. Roychoudhury, and V. Suhendra. *Exploiting branch constraints without exhaustive path enumeration*. In Int'l Workshop on WCET Analysis, 2005. <http://www.comp.nus.edu.sg/~tulika/wcet05.pdf>
- [5] S. Andalam, P. Roop, and A. Girault. *Pruning Infeasible Paths for Tight WCRT Analysis of Synchronous Programs*. In Intl' Conf. on Design, Automation and Test in Europe (DATE), 2011.
- [6] Leonardo De Moura and Nikolaj Bjørner. 2011. *Satisfiability modulo theories : introduction and applications*. Commun. ACM 54, 9 (September 2011), 69-77. <http://doi.acm.org/10.1145/1995376.1995394>
- [7] ARM Instruction Set Quick Reference Card. [http://infocenter.arm.com/help/topic/com.arm.doc.qrc00011/QRC0001\\_UAL.pdf](http://infocenter.arm.com/help/topic/com.arm.doc.qrc00011/QRC0001_UAL.pdf)
- [8] SMT-COMP : Satisfiability Modulo Theories Competition. Édition 2012. <http://smtcomp.sourceforge.net/2012/>
- [9] SMT solver Z3. <http://z3.codeplex.com/>
- [10] SMT solver CVC4. <http://cvc4.cs.nyu.edu/web/>
- [11] SMT solver MathSAT 5. <http://mathsat.fbk.eu/documentation.html>
- [12] SMT solver Boolector. <http://fmv.jku.at/boolector/>
- [13] SMT solver SONOLAR. <http://www.informatik.uni-bremen.de/~florian/sonolar/>
- [14] SMT solver MISTRAL. <http://www.cs.wm.edu/~tdillig/mistral/index.html>
- [15] SMT solver VeriT. <http://www.verit-solver.org/veriT-download.php>
- [16] SMT solver Barcelogic. <http://www.lsi.upc.edu/~oliveras/bclt-main.html>