

1 Brouillon

1.1 Analyse d'un graphe de flot de contrôle

Nous travaillons sans boucles, cet algorithme n'est pas correct (il 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

1.2 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.

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

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.

```
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`.

```
t [ADD d a b] p =
  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 *)
      (update d 0 p)
    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 !

```

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`.

```

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)

```

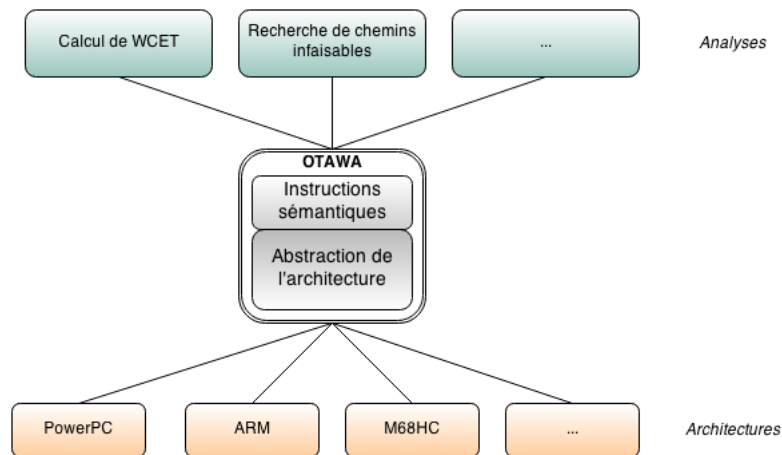


Figure 1: OTAWA permet aux analyses de s'abstraire de l'architecture

2 Objectif

2.1 WCET, enjeux (systèmes critiques etc)

Améliorer l'estimation de WCET

3 Solution

Explication : calcul de WCET par IPET, max d'un système ILP

Solution = Recherche de chemins infaisables pour améliorer l'estimation du WCET

Parler des outils choisis (OTAWA + CVC4 ?)

4 Structures de données utilisées

La représentation des prédicats (associés à un Edge)

=> représentation également de la mémoire

Traduction des prédicats dans le solveur SMT

5 Structure de l'algorithme

CFG : Représentation du programme sous la forme de graphe

On parcourt tous les chemins

On les représente sous la forme de prédicats (on y associe un arc du CFG)

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 2: Liste des instructions sémantiques d'OTAWA

On fait des appels au SMT pour vérifier la satisfiabilité
Retourne la liste de prédicats
On génère les contraintes ILP