scade-analyzer
note de référence

1 Introduction

Le projet consiste en la réalisation d'un analyseur statique pour des programmes SCADE, utilisant l'interprétation abstraite comme base de travail. Les objectifs attendus sont :

- Preuve de propriétés de sûreté sur des programmes
- Étude d'intervalles de variation pour des variables

L'expérience a été menée sur un sous-ensemble très restreint du langage SCADE, comportant notamment :

- noyau dataflow (opérations arithmétiques élémentaires, opérateurs `->` et `pre`), pas de prise en charge des horloges explicites (primitives `when`, `merge`, ...)
- blocs `activate`
- automates simples, à transitions faibles seulement, sans actions sur les transitions (les transitions de type `restart` sont prises en compte)

Dans ce document nous mettrons au clair les points suivants :

1. Spécification du sous-ensemble de SCADE considéré
2. Explication du fonctionnement de l'interprète pour la sémantique concrète
3. Explication de la traduction d'un programme SCADE en une formule logique représentant le cycle du programme
4. Explications sur l'interprétation abstraite en général, sur les domaines numériques, sur la recherche de points fixe
5. Explications sur notre adaptation de ces principes à l'analyse du langage SCADE, en particulier :
   - itérations chaotiques
   - domaines capables de faire des disjonctions de cas (graphes de décision)

Par la suite, nous ne considérons que des programmes SCADE bien typés.

2 Spécification

2.1 Grammaire

```
decl     := CONST id: type = expr;
          | NODE (var_def;*) RETURNS (var_def;*) VAR var_def;* body
var_def  := (PROBE? id),* : type
body     := LET eqn;* TEL
          | eqn;
type     := INT | BOOL | REAL | (TYPE,+)
eqn      := id = expr
          | GUARANTEE id : expr
          | ASSUME id : expr
          | AUTOMATON state* RETURNS id,*
          | ACTIVATE scope_if RETURNS id,*
scope_if := (VAR var_def;*)? body
```

```
          | IF expr THEN scope_if ELSE scope_if
state     := INITIAL? STATE id
             body
             until*
until     := UNTIL IF expr (RESUME|RESTART) id
expr      := int_const | real_const | TRUE | FALSE | id
           | unary_op expr | expr bin_op expr
           | IF expr THEN expr ELSE expr
           | id ( expr,* )
unary_op := - | PRE | NOT
bin_op    := + | - | * | / | -> | AND | OR | MOD
           | = | <> | < | > | <= | >=
```

2.2 Sémantique concrète

Pour simplifier, on considère dans cette section que tous les noeuds ont été inlinés.

On note $\mathbb{X}$ l'ensemble des variables définies dans le texte du programme, et on note $\mathbb{V}$ l'ensemble des valeurs qui peuvent être prises.

Un environnement de valeurs est une fonction $s\colon \mathbb{X} \to \mathbb{V}$ qui décrit un état de la mémoire du programme. On définit aussi un sous-ensemble $\mathbb{V}_i \subset \mathbb{V}$ de variables dont les valeurs sont des entrées du système.

L'exécution d'un programme est une séquence d'états mémoire $s_0, s_1, ..., s_n, ...$ conforme à la spécification du programme et à une série d'entrées. On note :

$$s_0 \xrightarrow{i_1} s_1 \xrightarrow{i_2} s_2 \to \cdots$$

Dans la sémantique par traduction, la sémantique d'un programme est définie par traduction du programme P en une formule logique F telle que :

$$s_{n-1} \xrightarrow{i_n} s_n \;\Leftrightarrow\; \begin{cases} \forall x \in \mathbb{V}_i, s_n(x) = i_n(x) \\ F(s_{n-1}, s_n) \end{cases}$$

Dans la sémantique par réduction, la sémantique d'un programme est définie par un ensemble de règles de réduction qui aboutissent à

$$s_{n-1} \xrightarrow{i_n} s_n \;\Leftrightarrow\; \frac{s_{n-1} \Rightarrow i_n}{s_{n-1} \Rightarrow s_n}$$

Pour un programme bien typé, étant donné s_{n-1} et i_n, il existe un unique état s_n qui remplit la condition.

Un scope Σ correspond à un ensemble de définitions (de variables, d'automates, ou de blocs activate), identifiés par un chemin. Chaque scope dispose d'une horloge propre. Pour traduire l'init et le reset, on introduit dans chaque scope Σ plusieurs variables :

- nreset_Σ : indique que le scope devra être reset lors du prochain cycle
- init_Σ : indique que le scope est à l'état initial dans ce cycle
- act_Σ : indique qu'un scope est actif dans ce cycle

Un nouveau scope est introduit dans chaque body d'un bloc activate ou d'un état d'automate.

De même, les automates introduisent deux variables :

- state_A, qui définit l'état de l'automate à ce cycle
- nstate_A, qui définit l'état de l'automate au cycle suivant, en supposant qu'il n'y a pas de reset du scope où l'automate est défini

L'état s_0 est défini par $s_0(\text{reset}_/) = \text{tt}$, où / est le scope racine englobant tout le programme ; toutes les autres variables de $\mathbb{V}$ pouvant prendre n'importe quelle valeur.

On suppose que chaque instance de `pre` est numérotée. Pour chaque $\text{pre}_i\ e$, on introduit la variable m_i qui enregistre la valeur de e dans le cycle courant et qui sert de mémoire pour le pre lors du cycle suivant.

Par la suite, que ce soit dans l'étude de la sémantique par réduction ou par traduction, on notera toujours l la mémoire (c'est-à-dire s_{n-1}) et s l'état courant (c'est-à-dire s_n, sur lequel on travaille).

2.3 Sémantique par réduction

On définit notre ensemble d'environnements comme étant $\mathbb{X} \to \mathbb{V} \cup \{\epsilon\}$, la valeur ϵ signifiant qu'une variable n'a pas encore pris sa valeur.

Pour chaque élément de programme, on va introduire un certain nombre de règles de réduction. On applique ces règles jusqu'à en déduire $l \Rightarrow s$, avec $\forall x \in \mathbb{X}, s(x) \neq \epsilon$.

Les déductions de la forme $l \Rightarrow s$ signifient « avec la mémoire l, on peut déduire du système l'état (partiel) s ». Les déductions de la forme $l, s \vDash e \to^v_\Sigma v$ signifient « avec la mémoire l et l'état partiellement calculé s, l'expression e calculée dans le scope Σ prend la valeur v » (la flèche $\to^v_\Sigma$ correspond à une réduction par valeur dans le scope Σ).

On notera $\Sigma \Vdash x = e$ pour signifier « dans le scope Σ on a la définition $x = e$ », de même pour les définitions de blocs activate et d'automates. On notera $\Sigma \Vdash \mathrm{pre}_i\, e$ pour signifier que $\mathrm{pre}_i\, e$ apparaît dans le scope Σ.

2.3.1 Règles de réduction pour l'activation du scope racine

On note i les entrées du système à ce cycle ; on fait par définition l'hypothèse $l \Rightarrow i$, c'est-à-dire qu'on peut obtenir l'environnement où seules les variables d'entrée sont définies. On introduit ensuite la règle suivante, qui dit que le scope racine est toujours actif et jamais reset par la suite :

$$\frac{l \Rightarrow s}{l \Rightarrow s[\mathrm{act}_/ := \mathrm{tt}][\mathrm{nreset}_/ := \mathrm{ff}]}$$

2.3.2 Règles de réduction pour l'init dans tous les scopes

Pour tout scope Σ défini dans le programme, qui est reset si et seulement si la condition $l, s \vDash r$ est vraie, on rajoute les règles de réduction suivantes qui permettent de déterminer si le scope est init ou pas :

$$\frac{l \Rightarrow s \quad l, s \vDash r_\Sigma}{l \Rightarrow s[\mathrm{init}_\Sigma := \mathrm{tt}]}$$

$$\frac{l \Rightarrow s \quad l, s \vDash \neg r_\Sigma \quad l(\mathrm{act}_\Sigma) = \mathrm{tt}}{l \Rightarrow s[\mathrm{init}_\Sigma := \mathrm{ff}]}$$

$$\frac{l \Rightarrow s \quad l, s \vDash \neg r_\Sigma \quad l(\mathrm{act}_\Sigma) = \mathrm{ff}}{l \Rightarrow s[\mathrm{init}_\Sigma := l(\mathrm{init}_\Sigma)]}$$

En particulier, pour le scope racine on instancie ces règles avec $(l, s \vDash r_/) \Leftrightarrow l(\mathrm{nreset}_/) = \mathrm{tt}$

2.3.3 Règles de réduction d'expressions

Ces règles permettent d'exprimer le calcul d'une expression. On note Σ le scope dans lequel l'expression est évaluée. On note $\odot$ n'importe quel opérateur binaire : $+, -, \times, /, \mathrm{mod}, <, >, \leqslant, \geqslant, =, \neq, \wedge, \vee$.

$$\frac{}{l, s \vDash c \to^v_\Sigma c}, c \in \mathbb{V} \qquad \frac{s(x) \neq \epsilon}{l, s \vDash x \to^v_\Sigma s(x)}, x \in \mathbb{X}$$

$$\frac{}{l, s \vDash \mathrm{pre}_i\, e \to^v_\Sigma l(m_i)}$$

$$\frac{l, s \vDash e_1 \to^v_\Sigma v_1 \quad l, s \vDash e_2 \to^v_\Sigma v_2}{l, s \vDash e_1 \odot e_2 \to^v_\Sigma v_1 \odot v_2}$$

$$\frac{s(\mathrm{init}_\Sigma) = \mathrm{tt} \quad l, s \vDash e_1 \to^v_\Sigma v_1}{l, s \vDash (e_1 \to e_2) \to^v_\Sigma v_1} \qquad \frac{s(\mathrm{init}_\Sigma) = \mathrm{ff} \quad l, s \vDash e_2 \to^v_\Sigma v_2}{l, s \vDash (e_1 \to e_2) \to^v_\Sigma v_2}$$

etc...

2.3.4 Règles de réduction pour les pre

Pour chaque expression $\text{pre}_i\, e$ introduite dans le scope Σ, on donne les deux règles suivante :

$$\frac{l \Rightarrow s \quad s(\text{act}_\Sigma) = \text{tt} \quad l, s \vDash e \rightarrow^v_\Sigma v}{l \Rightarrow s[m_i := v]}, \Sigma \Vdash \text{pre}_i\, e$$

$$\frac{l \Rightarrow s \quad s(\text{act}_\Sigma) = \text{ff}}{l \Rightarrow s[m_i := l(m_i)]}, \Sigma \Vdash \text{pre}_i\, e$$

2.3.5 Règles de réduction pour les définitions de variables

Pour toute définition $x = e$ apparaissant dans le scope Σ, on donne la règle suivante :

$$\frac{l \Rightarrow s \quad s(\text{act}_\Sigma) = \text{tt} \quad l, s \vDash e \rightarrow^v_\Sigma v}{l \Rightarrow s[x := v]}, \Sigma \Vdash x = e$$

2.3.6 Règles de réduction pour les blocs activate

Pour tout bloc activate if c then b_1 else b_2 apparaissant dans le scope Σ, on crée deux nouveaux scopes Σ_1 et Σ_2 dans lesquels on rajoute les règles de réductions pour b_1 et b_2 respectivement, et on rajoute les règles suivantes qui régissent l'activation des deux scopes Σ_1 et Σ_2 :

$$\frac{l \Rightarrow s \quad s(\text{act}_\Sigma) = \text{ff}}{l \Rightarrow s[\text{act}_{\Sigma_1} := \text{ff}][\text{act}_{\Sigma_2} := \text{ff}]}$$

$$\frac{l \Rightarrow s \quad s(\text{act}_\Sigma) = \text{tt} \quad l, s \vDash c \rightarrow^v_\Sigma \text{tt}}{l \Rightarrow s[\text{act}_{\Sigma_1} := \text{tt}][\text{act}_{\Sigma_2} := \text{ff}]} \qquad \frac{l \Rightarrow s \quad s(\text{act}_\Sigma) = \text{tt} \quad l, s \vDash c \rightarrow^v_\Sigma \text{ff}}{l \Rightarrow s[\text{act}_{\Sigma_1} := \text{ff}][\text{act}_{\Sigma_2} := \text{tt}]}$$

(implicitement sur toutes les règles : $\Sigma \Vdash \text{activate if } c \text{ then } b_1 \text{ else } b_2$)

Les deux scopes Σ_1 et Σ_2 héritent de la condition de reset du scope Σ.

2.3.7 Règles de réduction pour les automates

On se place dans le cadre $\Sigma \Vdash A$. On note s_0 l'état initial de A. Les règles d'activation des différents scopes des états se font en fonction de la variable state_A définie pour l'automate A comme suit :

$$\frac{l \Rightarrow s \quad l, s \vDash r_\Sigma}{l \Rightarrow s[\text{state}_A = s_0]} \qquad \frac{l \Rightarrow s \quad l, s \vDash \neg r_\Sigma}{l \Rightarrow s[\text{state}_A = l(\text{nstate}_A)]}$$

Puis pour chaque état s_i, on définit Σ_i son scope dans lequel on traduit son corps, puis on rajoute la règle d'activation suivante :

$$\frac{l \Rightarrow s \quad s(\text{state}_A) = s_i}{l \Rightarrow s[\text{act}_{\Sigma_i} = \text{tt}]} \qquad \frac{l \Rightarrow s \quad s(\text{state}_A) \neq s_i}{l \Rightarrow s[\text{act}_{\Sigma_i} = \text{ff}]}$$

Dans tous les scopes Σ_i, la condition de reset peut être éventuellement augmentée d'un $\vee$ avec une condition du type $l(\text{nreset}_{\Sigma_i}) = t$, où la variable nreset_{Σ_i} est mise à *true* dès que l'on emprunte une transition qui reset, et à *false* le reste du temps.

La règle pour une transition $s_i \xrightarrow{c} s_j$ sont du style :

$$\frac{l \Rightarrow s \quad s(\text{state}_A) = s_i \quad l, s \vDash c \rightarrow^v_\Sigma \text{tt}}{l \Rightarrow s[\text{nstate}_A := s_j]}$$

À cela, il faut rajouter les conditions qui disent que l'on emprunte exactement une transition à chaque cycle, ainsi que la partie qui définit les variables nreset_{Σ_i}.

Exemple 1. On va écrire les règles de réduction qui définissent le programme suivant :

```
node half() returns(c: int)
var half: bool;
    a, b: int;
    la, lb: int;
```

```
let
  half = true -> not pre half;
  activate
    if half then let
        a = la + 1;
        b = lb;
    tel else let
        a = la;
        b = lb + 1;
    tel
  returns a, b;
  la = 0 -> pre a;
  lb = 0 -> pre b;
  c = a - b;
tel
```

Les règles de calcul des expressions sont tout le temps les mêmes. Les règles déduites de la structure du programme sont les suivantes :

Initialisation :

$$\frac{l \Rightarrow s}{l \Rightarrow s[\mathrm{act}_{/} := \mathrm{tt}][\mathrm{nreset}_{/} := \mathrm{ff}]} \qquad \frac{l \Rightarrow s \quad l(\mathrm{nreset}_{/}) = \mathrm{tt}}{l \Rightarrow s[\mathrm{init}_{/} := \mathrm{tt}]}$$

$$\frac{l \Rightarrow s \quad l(\mathrm{nreset}_{/}) = \mathrm{ff} \quad l(\mathrm{act}_{/}) = \mathrm{tt}}{l \Rightarrow s[\mathrm{init}_{/} := \mathrm{ff}]} \qquad \frac{l \Rightarrow s \quad l(\mathrm{nreset}_{/}) = \mathrm{ff} \quad l(\mathrm{act}_{/}) = \mathrm{ff}}{l \Rightarrow s[\mathrm{init}_{/} := l(\mathrm{init}_{/})]}$$

Définition `c = a - b` (la réduction par valeur de $a - b$ en une valeur v se fait selon les règles données ci-dessus) :

$$\frac{l \Rightarrow s \quad s(\mathrm{act}_{/}) = \mathrm{tt} \quad l, s \vDash a - b \to^{v}_{/} v}{l \Rightarrow s[c := v]}$$

Définitions de `la` et `lb` :

$$\frac{l \Rightarrow s \quad s(\mathrm{act}_{/}) = \mathrm{tt} \quad l, s \vDash (0 \to \mathrm{pre}_1)\, a \to^{v}_{/} v}{l \Rightarrow s[\mathrm{la} := v]}$$

$$\frac{l \Rightarrow s \quad s(\mathrm{act}_{/}) = \mathrm{tt} \quad l, s \vDash (0 \to \mathrm{pre}_2\, b) \to^{v}_{/} v}{l \Rightarrow s[\mathrm{lb} := v]}$$

Définition de `half` :

$$\frac{l \Rightarrow s \quad s(\mathrm{act}_{/}) = \mathrm{tt} \quad l, s \vDash (\mathrm{true} \to \mathrm{not}\, \mathrm{pre}_3\, \mathrm{half}) \to^{v}_{/} v}{l \Rightarrow s[\mathrm{half} := v]}$$

Mémorisation des valeurs pour `pre a`, `pre b` et `pre half` (on écrit aussi les règles pour le cas où le scope est inactif à titre d'exemple simplement ; en pratique celles-ci sont éliminées puisque le scope racine est toujours actif) :

$$\frac{l \Rightarrow s \quad s(\mathrm{act}_{/}) = \mathrm{tt} \quad l, s \vDash a \to^{v}_{/} v}{l \Rightarrow s[m_1 := v]} \qquad \frac{l \Rightarrow s \quad s(\mathrm{act}_{/}) = \mathrm{ff}}{l \Rightarrow s[m_1 := l(m_1)]}$$

$$\frac{l \Rightarrow s \quad s(\mathrm{act}_{/}) = \mathrm{tt} \quad l, s \vDash b \to^{v}_{/} v}{l \Rightarrow s[m_2 := v]} \qquad \frac{l \Rightarrow s \quad s(\mathrm{act}_{/}) = \mathrm{ff}}{l \Rightarrow s[m_2 := l(m_2)]}$$

$$\frac{l \Rightarrow s \quad s(\mathrm{act}_{/}) = \mathrm{tt} \quad l, s \vDash \mathrm{half} \to^{v}_{/} v}{l \Rightarrow s[m_3 := v]} \qquad \frac{l \Rightarrow s \quad s(\mathrm{act}_{/}) = \mathrm{ff}}{l \Rightarrow s[m_3 := l(m_3)]}$$

Activation des deux moitiés du activate :

$$\frac{l \Rightarrow s \quad s(\mathrm{act}_{/}) = \mathrm{tt} \quad l, s \vDash \mathrm{half} \to \mathrm{tt}}{l \Rightarrow s[\mathrm{act}_{/1} := \mathrm{tt}][\mathrm{act}_{/2} := \mathrm{ff}]} \qquad \frac{l \Rightarrow s \quad s(\mathrm{act}_{/}) = \mathrm{tt} \quad l, s \vDash \mathrm{half} \to \mathrm{ff}}{l \Rightarrow s[\mathrm{act}_{/1} := \mathrm{ff}][\mathrm{act}_{/2} := \mathrm{tt}]}$$

$$\frac{l \Rightarrow s \quad s(\mathrm{act}_{/}) = \mathrm{ff}}{l \Rightarrow s[\mathrm{act}_{/1} := \mathrm{ff}][\mathrm{act}_{/2} := \mathrm{ff}]}$$

Définition de `a` et `b` dans la première moitié :

$$\frac{l \Rightarrow s \quad s(\mathrm{act}_{/1}) = \mathrm{tt} \quad l, s \vDash \mathrm{la} + 1 \to^{v}_{/1} v}{l \Rightarrow s[a := v]} \qquad \frac{l \Rightarrow s \quad s(\mathrm{act}_{/1}) = \mathrm{tt} \quad l, s \vDash \mathrm{lb} \to^{v}_{/1} v}{l \Rightarrow s[b := v]}$$

Et dans la deuxième moitié :

$$\frac{l \Rightarrow s \quad s(\mathrm{act}_{/2}) = \mathrm{tt} \quad l, s \vDash \mathrm{la} \to^{v}_{/2} v}{l \Rightarrow s[a := v]} \qquad \frac{l \Rightarrow s \quad s(\mathrm{act}_{/2}) = \mathrm{tt} \quad l, s \vDash \mathrm{lb} + 1 \to^{v}_{/2} v}{l \Rightarrow s[b := v]}$$

Et c'est tout.

(il faudrait faire un exemple avec des automates, mais ça risque d'être encore plus long !)

2.4 Sémantique par traduction, règles de traduction

On définit la traduction de P en une formule $F(l, s)$ comme suit.

2.4.1 Traduction des expressions numériques et booléennes

On utilise des formules "à trous" pour faire la traduction. Un trou $\Box$ correspond à la fonction $e \mapsto e$, une formule $a \wedge x = \Box$ correspond à la fonction $e \mapsto a \wedge x = e$, etc. L'argument d'une formule à trou peut aussi être un couple d'expressions, ainsi $\Box_1 + \Box_2$ correspond à la fonction $(e, f) \mapsto e + f$.

On définit la fonction $T(\Sigma, e, w)$ comme étant la traduction de l'expression e considérée dans le scope Σ et devant être placée dans le trou w. En pratique, on divise T en deux fonctions, une pour les expressions booléennes et une pour les expressions numériques. Le résultat d'une traduction doit être une formule booléenne.

Les règles sont les suivantes :

$$\begin{aligned}
T(\Sigma, (e_1, e_2, ..., e_n), w) &= T(\Sigma, e_1, \lambda f_1 . T(\Sigma, (e_2, ..., e_n), w[f_1, \Box_1, ..., \Box_{n-1}])) \\
\forall c \in \mathbb{V}, \quad T(\Sigma, c, w) &= w[c] \\
\forall x \in \mathbb{X}, \quad T(\Sigma, x, w) &= w[s(x)] \\
T(\Sigma, \mathrm{pre}_i\, e, w) &= w[l(m_i)] \\
T(\Sigma, e_1 \to e_2, w) &= (s(\mathrm{init}_\Sigma) \wedge T(\Sigma, e_1, w)) \vee (\neg s(\mathrm{init}_\Sigma) \wedge T(\Sigma, e_2, w)) \\
T(\Sigma, e_1 \odot e_2, w) &= T(\Sigma, (e_1, e_2), w[\Box_1 \odot \Box_2]) \\
T(\Sigma, \mathrm{if}\, c\, \mathrm{then}\, e_1\, \mathrm{else}\, e_2, w) &= (T(\Sigma, c, \Box) \wedge T(\Sigma, e_1, w)) \vee (\neg T(\Sigma, c, \Box) \wedge T(\Sigma, e_2, w)) \\
T(\Sigma, -e, w) &= T(\Sigma, e, w[-\Box]) \\
T(\Sigma, \neg e, w) &= T(\Sigma, e, w[\neg \Box])
\end{aligned}$$

Par la suite, on définira la fonction de traduction d'une définition par :

$$T_{\mathrm{def}}(\Sigma, x = e) = T(\Sigma, e, s(x) = \Box)$$

Dans le cas d'une multi-affectation $x_1, ..., x_n = e$, on utilisera la traduction suivante :

$$T_{\mathrm{def}}(\Sigma, (x_1, ..., x_n = e)) = T(\Sigma, e, s(x_1) = \Box_1 \wedge \cdots \wedge s(x_n) = \Box_n)$$

Dans le cas où l'on doit traduire une instanciation de noeud $n_i(v_1, ..., v_m)$ (c'est le cas dans notre implémentation puisqu'on ne fait pas d'inlining), on nomme $r_1, ..., r_n$ les valeurs renvoyées par le noeud et on utilise la règle $T(\Sigma, n_i(v_1, ..., v_m), w) = w[s(r_1), ..., s(r_n)]$. Il faut par ailleurs générer la traduction des définitions données dans le noeud et s'occuper de passer les arguments (nommés $\mathrm{arg}_1, ..., \mathrm{arg}_m$), en introduisant des équations du type $T(\Sigma, v_i, s(\mathrm{arg}_i) = \Box)$.

Exemple 2. Effectuons par exemple la traduction de $x = \mathrm{if}\ y \geqslant 0\ \mathrm{then}\ y\ \mathrm{else} - y$:

$$\begin{aligned}
F &= T(\Sigma, \mathrm{if}\ y \geqslant 0\ \mathrm{then}\ y\ \mathrm{else} - y, s(x) = \Box) \\
&= (T(\Sigma, y \geqslant 0, \Box) \wedge T(\Sigma, y, s(x) = \Box)) \vee (\neg T(\Sigma, y \geqslant 0, \Box) \wedge T(\Sigma, -y, s(x) = \Box)) \\
&= (s(y) \geqslant 0 \wedge s(x) = s(y)) \vee (\neg(s(y) \geqslant 0) \wedge s(x) = -s(y))
\end{aligned}$$

Exemple 3. Effectuons la traduction de $x = 0 \to \mathrm{pre}_1\, x + 1$.

$$\begin{aligned} F &= T(\Sigma, 0 \to \mathrm{pre}_1 x + 1, s(x) = \square) \\ &= (s(\mathrm{init}_\Sigma) \wedge T(\Sigma, 0, s(x) = \square)) \vee (\neg s(\mathrm{init}_\Sigma) \wedge T(\Sigma, \mathrm{pre}_1 x + 1, s(x) = \square)) \\ &= (s(\mathrm{init}_\Sigma) \wedge s(x) = 0) \vee (\neg s(\mathrm{init}_\Sigma) \wedge T(\Sigma, (\mathrm{pre}_1 x, 1), s(x) = \square_1 + \square_2)) \\ &= (s(\mathrm{init}_\Sigma) \wedge s(x) = 0) \vee (\neg s(\mathrm{init}_\Sigma) \wedge T(\Sigma, 1, s(x) = l(m_1) + \square))) \\ &= (s(\mathrm{init}_\Sigma) \wedge s(x) = 0) \vee (\neg s(\mathrm{init}_\Sigma) \wedge s(x) = l(m_1) + 1) \end{aligned}$$

Remarque : dans une étape à part, il faut penser à mémoriser une valeur pour m_1, c'est-à-dire à introduire l'équation $s(m_1) = s(x)$.

2.4.2 Traduction de scopes et de programmes

On définit une traduction complète pour les programmes, en pensant à introduire les équations de persistance de la mémoire pour les pre. On introduit aussi des équations qui déterminent si un scope est actif ou inactif, si il est init ou pas, si il doit être reset ou pas, en fonction des divers paramètres du programme (états d'automates, conditions de blocs activate). De manière générale, un scope a deux traductions qui sont produites : une pour le cas où ce scope est actif, et une pour le cas où il est inactif (dans le cas inactif, il s'agit simplement de perpétuer les mémoires). Ainsi la traduction d'un bloc « activate if c then b_1 else b_2 » sera du style $(c \wedge T_{\mathrm{active}}(b_1) \wedge T_{\mathrm{inactive}}(b_2)) \vee (\neg c \wedge T_{\mathrm{active}}(b_2) \wedge T_{\mathrm{inactive}}(b_1))$.

Tout ceci est long à écrire, aussi nous nous contenterons de donner un exemples.

Exemple 4. Traduction du programme :

```
node test() returns(x: int)
var lx: int;
let
  lx = 0 -> pre x;
  x = if lx >= 5 then 0 else lx + 1;
tel
```

On obtient la formule :

$$\begin{aligned} & \left(\begin{array}{l} \quad l(\mathrm{nreset}_/) \wedge s(\mathrm{init}_/) \\ \vee \ \neg l(\mathrm{nreset}_/) \wedge \left(\begin{array}{l} \quad l(\mathrm{act}_/) \wedge \neg s(\mathrm{init}_/) \\ \vee \ \neg l(\mathrm{act}_/) \wedge s(\mathrm{init}_/) = l(\mathrm{init}_/) \end{array} \right) \end{array} \right) \\ \wedge \ & s(\mathrm{act}_/) \\ \wedge \ & \neg s(\mathrm{nreset}_/) \\ \wedge \ & (s(\mathrm{init}_/) \wedge s(\mathrm{lx}) = 0) \vee (\neg s(\mathrm{init}_/) \wedge s(\mathrm{lx}) = l(m_1)) \\ \wedge \ & (s(\mathrm{lx}) \geqslant 5 \wedge s(x) = 0) \vee (s(\mathrm{lx}) < 5 \wedge s(x) = s(\mathrm{lx}) + 1) \\ \wedge \ & s(m_1) = s(x) \end{aligned}$$

Pour d'autres traductions, voir les sorties produites par `scade-analyzer`.

3 Interprète pour sémantique concrète

Une première façon de faire un interprète pour la sémantique concrète serait de faire un interprète de formules logiques, et d'appliquer la formule obtenue par traduction répétitivement jusqu'à obtenir un point fixe.

Ce n'est cependant pas cette solution que nous avons choisi de mettre en place, notre solution est plus proche de la sémantique par réduction.

Nous avons choisi de représenter un état s du programme en cours de calcul comme une valeur mutables, où les variables sont calculées au fur et à mesure qu'on les demande. La structure s peut donc contenir à un instant donné pour une certaine variable soit une valeur, soit une fonction à appeler pour que cette valeur soit calculée et rajoutée à s.

La procédure de calcul consiste à activer le scope racine, puis à appeler les fonctions de calcul sur les variables de sortie jusqu'à ce qu'on obtienne des valeurs. On enregistre ensuite la portion d'état qui nous intéresse pour le cycle suivant.

L'activation d'un scope se fait selon la procédure suivante :

- Pour une définition de variable $x = e$, rajouter dans s pour la variable x la fonction qui calcule e et rajoute la valeur trouvée dans s.
- Pour un bloc activate, rajouter dans s pour toutes les variables renvoyées par le bloc, la fonction qui choisit la branche à activer en calculant les conditions et active le scope correspondant.
- Pour un automate, rajouter dans s pour toutes les variables renvoyées par l'automate, la fonction qui choisit l'état à activer en se basant sur l'état enregistré au cycle précédent, et active le scope correspondant.

Le calcul de la valeur prise par une expression e, utilisée dans une condition ou pour une définition de variable, peut faire appel à d'autres variables du programme. À ce moment si une valeur a été mémorisée on l'utilise, sinon on appelle la fonction de calcul pour cette variable. Lorsque le calcul d'une variable est « en cours », ce statut est enregistré dans s, ce qui permet de détecter les cycles de dépendances.

L'étape d'enregistrement des variables d'intérêt pour le cycle suivant comporte notamment une phase de calcul des transitions faibles empruntées par les automates du programme, pour qu'à l'étape suivante le calcul puisse reprendre directement. Les restart sont aussi traités à ce moment là, avec une fonction pour reset un scope qui remet toutes les variables init à true et tous les états à l'état initial, récursivement.

4 Interprétation abstraite

Le but de l'interprétation abstraite est de prouver certaines propriétés sur un programme. Pour cela, nous passons par une première approximation, la sémantique collectrice, que nous approximons une seconde fois en la passant dans un domaine de représentation abstraite.

4.1 Sémantique collectrice

4.1.1 Nouvelle notation : fonction de transition

Notons $\mathbb{X}$ l'ensemble des variables. On note $\mathbb{X}_e$ l'ensemble des variables de type énumération et $\mathbb{X}_n$ les variables de type numérique, de sorte que $\mathbb{X} = \mathbb{X}_e \cup \mathbb{X}_n$.

Un état du système est une fonction $\mathbb{X} \to \mathbb{V}$, où $\mathbb{V}$ représente l'ensemble des valeurs (numériques ou énumération). On note $\mathbb{M} = \mathbb{X} \to \mathbb{V}$ l'ensemble des états du système.

Avant le premier cycle, le système peut être dans n'importe quel état de I :

$$I \;=\; \{s \in \mathbb{M} \,|\, s(\mathrm{nreset}_{/}) = \mathrm{tt}\}$$

Entre deux cycles, les variables qui comptent réellement dans s sont les variables actif et reset pour les scopes, les variables d'état pour les automates, et les mémoires des *pre*.

Vision habituelle : on a une suite d'états $s_0, s_1, \ldots$ qui représentent la mémoire entre deux cycles. s_0 est défini. On a une relation de transition qui prend s_n et les entrées i_{n+1} et qui calcule les sorties o_{n+1} et l'état suivant s_{n+1} :

$$s_0 \xrightarrow{i_1} o_1, s_1 \xrightarrow{i_2} o_2, s_2 \xrightarrow{i_3} o_3, s_3 \to \cdots$$

Nous introduisons ici une seconde notation pour ce fonctionnement.

Les variables de l'état précédent s_{n-1}, au lieu d'être considérées comme un lieu à part, sont partiellement copiées dans l'état s_n par une fonction que l'on appellera fonction de cycle. Cette fonction copie uniquement les variables dont les valeurs précédentes sont utiles pour le calcul de la transition, et préfixe leur noms d'un préfixe standard, « L », qui indique que ce sont des variables de type *last*, c'est-à-dire des copies de valeurs du cycle précédent.

On note cette fonction de cycle $c\colon \mathbb{M} \to \mathcal{P}(\mathbb{M})$; nous écrivons cette fonction comme non-déterministe car après cette fonction un certain nombre de variables sont oubliées et on considère que leurs valeurs n'ont pas d'importance. Elle peut être définie à partir d'un ensembles de variables C, qui sont les variables qui nous intéresseront lors du calcul de la transition :

$$c(s) \;=\; \{s' \in \mathbb{M} \,|\, \forall x \in C, s'(L\,x) = s(x)\}$$

Déroulement d'un cycle : on prend l'état s après passage par la fonction de cycle, on y met les valeurs des entrées du système. On applique ensuite la fonction $f\colon \mathbb{M} \to \mathbb{M}$ qui calcule toutes les variables du système (elle peut être définie comme la saturation de la sémantique par réduction, comme le point fixe de l'application répétée d'une formule, ou plus classiquement comme l'application d'une série d'instructions en style impératif, résultat de la compilation du programme). On peut à ce moment récupérer les valeurs de sorties.

Avec nos notations : o_{n+1} est la restriction de $f(c(s_n) + i_{n+1})$ aux variables de sortie, où $c(s_n) + i_{n+1}$ correspond à définir les variables de $c(s_n)$ et de i_{n+1}.

Remarque 5. En principe, quel que soit $x \in c(s_n)$, $f(x + i_{n+1})$ ne peut être qu'un seul environnement, car le programme est déterministe. C'est pourquoi on se permet l'abus de notation $f(c(s_n) + i_{n+1})$.

4.1.2 Suppression des entrées, sémantique collectrice

On s'intéresse maintenant à l'exécution d'un programme SCADE quelles que soient ses entrées i_n. On peut faire des hypothèses sur ces entrées en utilisant la directive `assume` du langage. On suppose que l'on dispose d'une fonction $q\colon \mathbb{M} \to \{\mathrm{tt}, \mathrm{ff}\}$ qui nous dit si un environnement est conforme à la spécification donnée par les directives `assume`.

On s'intéresse maintenant à la sémantique non déterministe suivante :

$$\begin{aligned} s_0 &= \{s \in \mathbb{M} \,|\, s(\mathrm{nreset}_/) = \mathrm{tt}\} \\ s_{n+1} &= g(s_n) \\ g(x) &= \bigcup_{s \in x} \{f(a), a \in c(s), q(f(a)) = \mathrm{tt}\} \end{aligned}$$

La valeur s_n contient tous les environnements possibles pour le système à la n-ème étape, quelles que soient les entrées jusque là.

On définit maintenant la sémantique collectrice du programme comme étant la valeur :

$$\begin{aligned} S &= \mathrm{lfp}_{s_0}\,(\lambda s.s_0 \cup g(s)) \\ S &\in \mathcal{P}(\mathbb{M}) \end{aligned}$$

S représente ici exactement l'ensemble de tous les états accessibles par le système, à tout moment, quelles que soient les valeurs en entrée.

Toute la suite de notre travail consistera à construire une approximation la meilleure possible de S.

4.2 Généralités sur l'interprétation abstraite

Une abstraction est définie par une correspondance de Galois entre $\mathcal{P}(\mathbb{M})$ et $\mathcal{D}^\#$, représentation abstraite d'une partie de $\mathbb{M}$. L'abstraction peut être caractérisée par sa fonction de concrétisation :

$$\gamma \;:\; \mathcal{D}^\# \to \mathcal{P}(\mathbb{M})$$

On peut aussi généralement s'appuyer sur l'existence d'une fonction d'abstraction :

$$\alpha \;:\; \mathcal{P}(\mathbb{M}) \to \mathcal{D}^\#$$

Cette fonction fait correspondre à une partie de $\mathbb{M}$ sa meilleure approximation dans le domaine abstrait, lorsque celle-ci existe (par exemple dans le cas des polyèdres, l'abstraction d'un ensemble fini de points est leur enveloppe convexe, mais un cercle n'a pas de meilleure abstraction). En pratique la fonction d'abstraction n'est pas très utilisée.

Dans tous les cas, on s'attend à ce que $\forall x \in \mathcal{D}^\#$, $x \sqsubseteq \alpha(\gamma(x))$ d'une part et $\forall y \in \mathcal{P}(\mathbb{M})$, $y \subseteq \gamma(\alpha(y))$ d'autre part.

De base ici, nous avons deux choix simples pour $\mathcal{D}^\#$: les intervalles et les polyèdres convexes. Ceux-ci sont considérés acquis pour la suite ; on les note $\mathcal{D}^\#_{\text{int}}$ et $\mathcal{D}^\#_{\text{poly}}$, avec les fonctions de concrétisation γ_{int} et γ_{poly} associées.

On note $\mathbb{E}$ l'ensemble des équations (égalités et inégalités) sur des variables de $\mathbb{X}$. Par exemple les éléments suivants sont des équations de $\mathbb{E}$: $x = 0, c = \text{tt}, y \geqslant 5\,x - 2$.

Pour $s \in \mathbb{M}$ et $e \in \mathbb{E}$, on note $s \vDash e$ si l'expression e est vraie dans l'état s.

Pour un domaine abstrait $\mathcal{D}^\#$ et pour une expression $e \in \mathbb{E}$, on suppose que l'on a une fonction sémantique $[\![e]\!] : \mathcal{D}^\# \to \mathcal{D}^\#$ qui restreint l'abstraction $s^\#$ en une sur-approximation (la meilleure possible) de $\alpha(\{s \in \gamma(s^\#) \,|\, s \vDash e\})$:

$$\{s \in \gamma(s^\#) \,|\, s \vDash e\} \;\sqsubseteq\; \gamma([\![e]\!](s^\#))$$

La fonction de transition f est représentée dans l'abstrait par une fonction $f^\#$ qui correspond à l'application d'un certain nombre de contraintes de $\mathbb{E}$, ainsi que de disjonction de cas. On remarque que l'aspect impératif disparaît complètement, on n'a plus qu'un ensemble d'équations et de disjonctions. La traduction du programme SCADE en formule logique donne directement une formule de ce style que l'on peut appliquer sur un environnement abstrait.

La fonction de cycle c correspond à conserver un certain nombre de variables en tant que « mémoires », en préfixant leur noms d'un « L » pour *last*. Notons C l'ensemble des variables à conserver. Cette fonction peut être représentée dans l'abstrait par l'opérateur $c^\#$ dont une définition est :

$$c^\#(s) \;=\; \alpha(\{\rho \in \mathbb{M} \,|\, \forall x \in C, \exists \rho' \in \gamma(s) \,|\, \rho(Lx) = \rho'(x)\})$$

(cette définition n'est pas constructive car on n'implémente jamais γ directement)

Cela correspond à oublier un certain nombre de variables qui ne nous intéressent plus, et à renommer celles que l'on garde.

Exemple 6. Soit le programme suivant :

```
node counter() returns(x: int)
  x = 0 -> (if pre x = 5 then 0 else pre x + 1)
```

Celui-ci est traduit par une formule du style :

$$\begin{array}{ll} & \text{init}_/ \equiv \text{Lnreset}_/ \\ \wedge & \text{nreset}_/ \equiv \text{ff} \\ \wedge & \left(\begin{array}{ll} & (\text{init}_/ \equiv \text{tt} \wedge x = 0) \\ \vee & \left(\text{init}_/ \equiv \text{ff} \wedge \left(\begin{array}{ll} & (\text{Lx} = 5 \wedge x = 0) \\ \vee & (\text{Lx} \neq 5 \wedge x = \text{Lx} + 1) \end{array} \right) \right) \end{array} \right) \end{array}$$

Les deux variables qui ont besoin d'être perpétuées d'un cycle au suivant sont $\text{nreset}_/$ et x, la fonction de cycle $c^\#$ est donc définie à partir de $C = \{\text{nreset}_/, x\}$.

La fonction $f^\#$, quant à elle, reflète directement la structure de la formule :

$$f^\#(s) \;=\; [\![\text{init}_/ \equiv \text{Lnreset}_/]\!] \circ [\![\text{nreset}_/ \equiv \text{ff}]\!] \left(\begin{array}{ll} & [\![\text{init}_/ \equiv \text{tt}]\!] \circ [\![x = 0]\!](s) \\ \sqcup & [\![\text{init}_/ \equiv \text{ff}]\!] \left(\begin{array}{ll} & [\![\text{Lx} = 5]\!] \circ [\![x = 0]\!](s) \\ \sqcup & [\![\text{Lx} \neq 5]\!] \circ [\![x = \text{Lx} + 1]\!](s) \end{array} \right) \end{array} \right)$$

Par facilité, on note $g^\# = c^\# \circ f^\#$. Étant donné qu'un programme est essentiellement une grosse boucle, la valeur qui nous intéresse est l'abstraction de S donnée par :

$$S^\# \;=\; \text{lfp}_{I^\#}(\lambda i.I^\# \sqcup g^\#(i))$$

Où $I^\#$ est l'état initial du système et est défini par $I^\# = [\![i]\!](\top)$, où i est une équation du type $\text{Lnreset}_/ = \text{tt}$.

Nous en venons donc à chercher des domaines abstraits les mieux à même de représenter les différentes contraintes exprimables dans $\mathbb{E}$. Dans notre cas, celles-ci se divisent essentiellement en deux catégories :

- Contraintes numériques : les variables sont dans $\mathbb{X}_n$, les constantes dans $\mathbb{N}$ (ou $\mathbb{Q}$), les opérateurs sont $+, -, \times, \div, \mathrm{mod}, =, \geqslant, \neq$. On note $\mathbb{E}_n$ l'ensemble de telles contraintes.
- Contraintes énumérées : les variables sont dans $\mathbb{X}_e$, les constantes dans un ensemble fini qui dépend du types des variables, les opérateurs sont $\equiv, \not\equiv$. On note $\mathbb{E}_e$ l'ensemble de telles contraintes.

Les domaines numériques $\mathcal{D}^{\#}_{\text{int}}$ et $\mathcal{D}^{\#}_{\text{poly}}$ ne sont pas à même de représenter correctement les contraintes de $\mathbb{E}_e$. Généralement, on définit :

$$\begin{aligned} \forall e \in \mathbb{E}_e, [\![e]\!]_{\text{int}} &= \mathrm{id}_{\mathcal{D}^{\#}_{\text{int}}} \\ \forall e \in \mathbb{E}_e, [\![e]\!]_{\text{poly}} &= \mathrm{id}_{\mathcal{D}^{\#}_{\text{poly}}} \end{aligned}$$

Les variables booléennes peuvent être représentées par 0 et 1, par exemple on peut introduire les transformations suivantes (en notant $\mathbb{X}_b$ l'ensemble des variables à valeurs booléennes) :

$$\begin{aligned} \forall x \in \mathbb{X}_b, [\![x = \mathrm{tt}]\!] &= [\![x = 1]\!]_{\text{poly}} \\ \forall x \in \mathbb{X}_b, [\![x = \mathrm{ff}]\!] &= [\![x = 0]\!]_{\text{poly}} \\ \forall x, y \in \mathbb{X}_b, [\![x = y]\!] &= [\![x = y]\!]_{\text{poly}} \\ \forall x, y \in \mathbb{X}_b, [\![x \neq y]\!] &= [\![x = 1 - y]\!]_{\text{poly}} \end{aligned}$$

Les résultats sont généralement plus que médiocres. De plus, on ne peut pas représenter ainsi de façon exacte les valeurs d'énumérations ayant plus de deux éléments (puisqu'on se restreint à une enveloppe convexe).

5 Domaines abstraits et disjonction de cas

Pour l'analyse de programmes SCADE, l'analyse de l'ensemble de la boucle de contrôle comme une seule valeur dans un environnement abstrait numérique est insuffisante. Il nous a donc été crucial de développer des domaines abstraits capables de faire des disjonctions de cas afin de traiter de manière plus fine l'évolution du programme.

Nous souhaitons pouvoir faire des disjonctions de cas selon les valeurs des variables de $\mathbb{X}_e$. Par exemple si on a un automate A dont la variable d'état s'appelle q et évolue dans l'ensemble $Q = \{\mathrm{up}, \mathrm{down}, \mathrm{left}, \mathrm{right}, \mathrm{stay}\}$, on voudrait pouvoir isoler cette variable des autres, ne plus l'inclure dans le domaine abstrait et l'utiliser pour différencier plusieurs valeurs abstraites. Il nous faut donc redéfinir le domaine abstrait $\mathcal{D}$ et surtout la fonction d'application d'une condition $[\![e]\!]$ avec $e \in \mathbb{E}_e$.

5.1 Domaine à disjonction simple

Supposons que l'on ait maintenant trois ensembles de variables :

- $\mathbb{X}_n$: variables numériques
- $\mathbb{X}_e$: variables énumérées non considérées comme variables de disjonction
- $\mathbb{X}_d$: variables de disjonction, prenant leurs valeurs dans $\mathbb{V}_d$ un ensemble fini (pour être précis, il faudrait noter $\forall x \in \mathbb{X}_d$, $\mathbb{V}_d(x)$ l'ensemble des valeurs possibles pour la valeur x, qui peut être différent selon la variable - on est amené à faire un peu de typage, il faut en particulier s'assurer que les contraintes que l'on donne sont entre deux variables pouvant prendre les mêmes valeurs).

On considère dans cette section que l'on a un domaine abstrait $\mathcal{D}^{\#}_0$ capable de gérer les contraintes sur les variables numériques et sur les variables énumérées, mais sans relation entre les deux. Le domaine $\mathcal{D}^{\#}_0$ représente une abstraction de $\mathbb{M}_0 = (\mathbb{X}_n \cup \mathbb{X}_e) \to \mathbb{V}$. On note $\bot_0$ et $\top_0$ les éléments bottom et top de ce treillis, $\sqcup_0$ et $\sqcap_0$ les bornes inf et sup de ce treillis, ainsi que ∇_0 son opérateur de widening. On note $[\![\cdot]\!]_0$ la fonction de restriction par une contrainte.

La particularité des variables de disjonction est que l'on ne réalise pas d'abstraction sur celles-ci : on représente directement un état par une valuation de ces variables, dans $\mathbb{X}_d \to \mathbb{V}_d = \mathbb{M}_d$.

On appelle toujours $\mathbb{M} = \mathbb{X} \to \mathbb{V}$, où $\mathbb{X} = \mathbb{X}_n \cup \mathbb{X}_e \cup \mathbb{X}_d$. On a une injection évidente de $\mathbb{M}_d$ dans $\mathcal{P}(\mathbb{M})$, on identifie donc $d \in \mathbb{M}_d$ à $\{s \in \mathbb{M} | \forall x \in \mathbb{X}_d, s(x) = d(x)\}$. De même on identifie $e \in \mathbb{M}_0$ à $\{s \in \mathbb{M} | \forall x \in \mathbb{X}_n \cup \mathbb{X}_e, s(x) = e(x)\}$.

On construit maintenant le domaine abstrait disjonctif comme suit :

$$\begin{aligned} \mathcal{D}^\# &= \mathbb{M}_d \to \mathcal{D}_0^\# \\ \gamma(s) &= \bigcup_{d \in \mathbb{M}_d} d \cap \gamma(s(d)) \end{aligned}$$

Les éléments $\top$ et $\bot$ sont définis comme suit :

$$\begin{aligned} \bot &= \lambda d. \bot_0 \\ \top &= \lambda d. \top_0 \end{aligned}$$

On vérifie bien que $\gamma(\bot) = \varnothing$ et $\gamma(\top) = \mathbb{M}$.

On peut aussi définir les opérations $\sqcup$ et $\sqcap$:

$$\begin{aligned} s \sqcup t &= \lambda d. (s(d) \sqcup_0 t(d)) \\ s \sqcap t &= \lambda d. (s(d) \sqcap_0 t(d)) \end{aligned}$$

Enfin, la partie intéressante : on peut définir un certain nombres d'opérateurs de restriction :

- $\forall x, y \in \mathbb{X}_d, \forall s \in \mathcal{D}^\#$,

$$\begin{aligned} [\![x = y]\!](s) &= \lambda d. \begin{cases} s(d) & \text{si } d(x) = d(y) \\ \bot_0 & \text{sinon} \end{cases} \\ [\![x \neq y]\!](s) &= \lambda d. \begin{cases} s(d) & \text{si } d(x) \neq d(y) \\ \bot_0 & \text{sinon} \end{cases} \end{aligned}$$

- $\forall x \in \mathbb{X}_d, \forall v \in \mathbb{V}_d$,

$$\begin{aligned} [\![x = v]\!](s) &= \lambda d. \begin{cases} s(d) & \text{si } d(x) = v \\ \bot_0 & \text{sinon} \end{cases} \\ [\![x \neq v]\!](s) &= \lambda d. \begin{cases} s(d) & \text{si } d(x) \neq v \\ \bot_0 & \text{sinon} \end{cases} \end{aligned}$$

$[\![e]\!]$ est donc défini correctement pour tout $e \in \mathbb{E}_d$, où $\mathbb{E}_d$ est l'ensemble des conditions sur variables de disjonction de $\mathbb{V}_d$. Pour toute expression $e \in \mathbb{E}_n$ ou $e \in \mathbb{E}_e$, le domaine $\mathcal{D}_0^\#$ est sensé savoir les prendre en compte de manière satisfaisante, on définit donc :

$$\forall e \in \mathbb{E}_n \cup \mathbb{E}_e, [\![e]\!](s) = \lambda d. [\![e]\!]_0(s(d))$$

L'opérateur de widening reste problématique. On peut définir un opérateur de widening point par point :

$$s \nabla t = \lambda d. s(d) \nabla_0 t(d)$$

Mais celui-ci est peu satisfaisant car chaque état $d \in \mathbb{M}_d$ représente potentiellement un état d'un système de transitions, pouvant déboucher sur lui-même ou sur un autre état, et il faut savoir prendre en compte ces disjonctions à un niveau plus fin. Il faut donc plutôt voir le tout comme un système de transitions.

Étant donné notre système représenté par une fonction de transition $f^\#$ et une fonction de cycle $c^\#$ (par facilité, on notera $g^\# = c^\# \circ f^\#$), l'ensemble des états accessible par le système est $S^\# = \mathrm{lfp}_{I^\#}(\lambda s. I^\# \sqcup g^\#(s))$, où $I^\# = [\![i]\!](\top)$.

Pour $d_0 \in \mathbb{M}_d$, notons $r_{d_0} : \mathcal{D}^\# \to \mathcal{D}^\#$ tel que $r_{d_0}(s) = \lambda d. \begin{cases} \bot_0 & \text{si } d \neq d_0 \\ s(d) & \text{si } d = d_0 \end{cases}$

Le principe des itérations chaotiques peut s'écrire comme suit :

- Poser :

$$\begin{aligned} s_0 &= I^\# \\ \delta_0 &= \{d \in \mathbb{M}_d | s_0(d) \neq \bot_0\} \end{aligned}$$

- Tant que $\delta_n \neq \varnothing$, on répète le processus suivant :

$$\begin{aligned} a_{n+1} &\in \delta_n \text{ (choisi arbitrairement)} \\ D_{n+1} &= g^{\#}(r_{a_{n+1}}(s_n)) \\ s_{n+1} &= s_n \sqcup D_{n+1} \\ \delta_{n+1} &= (\delta_n \backslash \{a_{n+1}\}) \cup \{d \in \mathbb{M}_d | s_{n+1}(d) \neq s_n(d)\} \end{aligned}$$

Intuitivement : $\mathbb{M}_d$ représente l'ensemble des états possibles pour notre système de transition. À chaque itération, on choisit un état qui a grossi depuis la dernière fois. On calcule ses successeurs et on met à jour l'ensemble des états que l'on connaît.

Problème : ici on ne fait pas de widening, et on peut être à peu près sûr que l'analyse ne terminera pas (sauf cas simples). Pour cela, on introduit un ensemble $K_\nabla \subset \mathbb{M}_d$ qui représente l'ensemble des états que l'on devra faire grossir par widening et non par union simple dans le futur. La définition devient alors :

$$\begin{aligned} s_0 &= I^{\#} \\ \delta_0 &= \{d \in \mathbb{M}_d | \, s_0(d) \neq \bot_0\} \\ K_{\nabla,0} &= \varnothing \end{aligned}$$

$$\begin{aligned} a_{n+1} &\in \delta_n \text{ (choisi arbitrairement)} \\ D_{n+1} &= g^{\#}(r_{a_{n+1}}(s_n)) \\ s_{n+1} &= \lambda d. \begin{cases} s_n(d) \nabla_0 D_{n+1}(d) & \text{si } d \in K_{\nabla,n} \\ s_n(d) \sqcup_0 D_{n+1}(d) & \text{sinon} \end{cases} \\ K_{\nabla,n+1} &= K_{\nabla,n} \cup \{d \in \mathbb{M}_d | \, s_n(d) \neq \bot_0 \wedge s_{n+1}(d) \neq s_n\} \\ \delta_{n+1} &= (\delta_n \backslash \{a_{n+1}\}) \cup \{d \in \mathbb{M}_d | s_{n+1}(d) \neq s_n(d)\} \end{aligned}$$

Ie : si un état est apparu à une étape, et si à une étape ultérieure il grossit, alors lors de toutes les étapes suivantes on le fera grossir non pas par union simple mais par élargissement.

Reste une question : comment prendre en compte les conditions de boucle qui permettent de réduire le domaine abstrait ? La définition précédente n'est peut-être pas la bonne, car elle risque d'appliquer des élargissements que l'on ne sait plus ensuite comment rétrécir pour refaire apparaître les bonnes conditions. Nous proposons comme forme final le processus d'itération suivant :

$$\begin{aligned} s_0 &= I^{\#} \\ \delta_0 &= \{d \in \mathbb{M}_d | \, s_0(d) \neq \bot_0\} \\ K_{\nabla,0} &= \varnothing \end{aligned}$$

$$\begin{aligned} a_{n+1} &\in \delta_n \text{ (choisi arbitrairement)} \\ D^0_{n+1} &= \mathrm{lfp}_{r_{a_{n+1}}(s_n)}(\lambda i. r_{a_{n+1}}(s_n \sqcup g^{\#}(i))) \\ D_{n+1} &= D^0_{n+1} \sqcup g^{\#}(D^0_{n+1}) \\ s_{n+1} &= \lambda d. \begin{cases} s_n(d) \nabla_0 D_{n+1}(d) & \text{si } d \in K_{\nabla,n} \text{ et } d \neq a_{n+1} \\ s_n(d) \sqcup_0 D_{n+1}(d) & \text{sinon} \end{cases} \\ K_{\nabla,n+1} &= K_{\nabla,n} \cup \{d \in \mathbb{M}_d | \, s_n(d) \neq \bot_0 \wedge s_{n+1}(d) \neq s_n\} \\ \delta_{n+1} &= (\delta_n \backslash \{a_{n+1}\}) \cup \{d \in \mathbb{M}_d | s_{n+1}(d) \neq s_n(d)\} \end{aligned}$$

Où le point fixe D^0_{n+1} est calculé avec appel au widening au besoin, et en faisant une ou des itérations décroissantes à la fin. Intuitivement : on fait grossir un état au maximum, en cherchant son point fixe en boucle sur lui-même. Ensuite seulement on s'occupe de savoir ce qu'il peut propager aux autres états.

5.2 Domaine à graphe de décision

Nous proposons dans ce paragraphe un second domaine abstrait capable de faire des disjonctions de cas, et qui permet de mieux traîter des problèmes ayant un nombre important de variables de type énuméré reliées entre elles par des relations complexes.

Définition du domaines abstraits avec graphes de décision : on va écrire ici une définition mathématique des opérateurs que l'on a implémenté. On fait abstraction des problématiques de mémoisation et de partage des sous-graphes, qui font tout l'intérêt de la technique d'un point de vue pratique mais qui peuvent être considérés comme un traitement à part (ce n'est rien de plus que de la mémoisation et du partage).

5.2.1 Variables et contraintes

Il y a deux domaines de variables, $\mathbb{X}_e$ pour les énumérés et $\mathbb{X}_n$ pour les variables numériques. Il y a deux domaines pour les contraintes, $\mathbb{E}_e$ les contraintes sur les énumérés (de la forme $x \equiv y$ ou $x \equiv v, v \in \mathbb{V}_e$) et $\mathbb{E}_n$ les contraintes sur les variables numériques (égalités ou inégalités).

5.2.2 Domaine numérique

On note D_n le domaine des valeurs numériques et $\sqcup_n$, $\sqcap_n$, $[\![\cdot]\!]_n$, $\bot_n$, $\top_n$, $\sqsubseteq_n$, γ_n, ∇_n les éléments correspondants dans ce domaine. On considère que $\gamma_n : D_n \to \mathcal{P}(\mathbb{M})$ (avec $\mathbb{M} = \mathbb{X}_e \cup \mathbb{X}_n \to \mathbb{V}$) donne toutes les valuations possibles pour les variables de $\mathbb{X}_e$.

5.2.3 Les EDD

On définit un ordre sur les variables de $\mathbb{X}_e$: $\mathbb{X}_e = \{x_1, x_2, ..., x_n\}$ (bien choisi pour réduire la taille du graphe).

On définit ensuite une valeur du domaine disjonctif, ie un EDD, par un type somme comme suit :

$$\begin{array}{rcl} s & := & V(t \in D_{\text{num}}) \\ & | & C(x_i, v_1 \to s_1, v_2 \to s_2, ..., v_p \to s_p) \end{array}$$

Si on voit ça comme un arbre, alors il faut que si un noeud $C(x_i, ...)$ est ancêtre d'un noeud $C(x_j, ...)$, alors $i < j$ (par rapport à l'ordre donné sur $\mathbb{X}_e$).

Pour faciliter les notations, on introduit le rang d'un noeud :

$$\begin{array}{rcl} \delta(C(x_i, v_1 \to s_1, ..., v_p \to s_p)) & = & i \\ \delta(V(t)) & = & \infty \end{array}$$

La contrainte se traduit par, pour tout noeud $C(x_i, v_1 \to s_1, ..., v_p \to s_p)$, on a $\forall j, i < \delta(s_j)$.

On définit aussi la contrainte suivante : on n'a pas le droit d'avoir de noeud $C(x_i, v_1 \to s_1, v_2 \to s_2, ..., v_p \to s_p)$ si $s_1 = s_2 = \cdots = s_p$. Cela implique l'unicité de l'arbre qui représente un environnement donné.

La concrétisation est définie comme suit :

$$\begin{array}{rcl} \gamma(V(t)) & = & \gamma_n(t) \\ \gamma(C(x_i, v_1 \to s_1, ..., v_p \to s_p)) & = & \bigcup_{j=1}^{p} \{s \in \gamma(s_j) \,|\, s(x_i) = v_j\} \end{array}$$

Les éléments $\top$ et $\bot$ sont définis comme suit :

$$\begin{array}{rcl} \top & = & V(\top_n) \\ \bot & = & V(\bot_n) \end{array}$$

Pour assurer l'unicité lors des transformations, on définit la fonction de réduction r :

$$r(x_i, v_1 \to s_1, ..., v_p \to s_p) = \begin{cases} s_1 & \text{si } s_1 = s_2 = \cdots s_p \\ C(x_i, v_1 \to s_1, ..., v_p \to s_p) & \text{sinon} \end{cases}$$

L'opération $\sqcap$ est définie comme suit :

$$\begin{array}{rcl} V(t) \sqcap V(t') & = & V(t \sqcap_n t') \\ C(x_i, v_1 \to s_1, ..., v_p \to s_p) \sqcap C(x_i, v_1 \to s'_1, ..., v_p \to s'_p) & = & r(x_i, v_1 \to s_1 \sqcap s'_1, ..., v_p \to s_p \sqcap s'_p) \\ C(x_i, v_1 \to s_1, ..., v_p \to s_p) \sqcap s' & \overset{\text{lorsque } i < \delta(s')}{=} & r\begin{pmatrix} & v_1 \to s_1 \sqcap s' \\ x_i, & \vdots \\ & v_p \to s_p \sqcap s' \end{pmatrix} \end{array}$$

et symétriquement lorsque $\delta(s) > \delta(s')$ (le noeud le plus haut est celui correspondant à la variable d'indice le plus faible, pour respecter l'ordre).

L'opération $\sqcup$ est définie pareil.

Si $e \in \mathbb{E}_n$, on définit $[\![e]\!]$ par :

$$\begin{aligned} [\![e]\!](V(t)) &= V([\![e]\!]_n(t)) \\ [\![e]\!](C(x_i, v_1 \to s_1, ..., v_p \to s_p)) &= r(x_i, v_1 \to [\![e]\!](s_1), ..., v_p \to [\![e]\!](s_p)) \end{aligned}$$

Pour les conditions sur les énumérés, on définit d'abord :

$$\begin{aligned} c(x \equiv v) &= C(x, v \to \top, v' \to \bot, v' \in \mathbb{V}_e \backslash \{v\}) \\ c(x \not\equiv v) &= C(x, v \to \bot, v' \to \top, v' \in \mathbb{V}_e \backslash \{v\}) \\ c(x_i \equiv x_j) &\overset{\text{lorsque } i<j}{=} C(x_i, v_1 \to c(x_j \equiv v_1), ..., v_p \to c(x_j \equiv v_p)) \\ c(x_i \not\equiv x_j) &\overset{\text{lorsque } i<j}{=} C(x_i, v_1 \to c(x_j \not\equiv v_1), ..., v_p \to c(x_j \not\equiv v_p)) \end{aligned}$$

et symétriquement lorsque $j > i$.

On peut ensuite poser, pour $e \in \mathbb{E}_e$:

$$[\![e]\!](s) = c(e) \sqcap s$$

L'égalité entre les valeurs représentées par deux EDD correspond à l'égalité de ces deux EDD (c'est une CNS).

L'inclusion est également définie par induction :

$$\begin{aligned} V(t) \sqsubseteq V(t') &\equiv t \sqsubseteq_n t' \\ C(x_i, v_1 \to s_1, ..., v_p \to s_p) \sqsubseteq C(x_i, v_1 \to s'_1, ..., v_p \to s'_p) &\equiv \bigwedge_{i=1}^{p} s_i \sqsubseteq s'_i \\ s \sqsubseteq C(x_i, v_1 \to s_1, v_2 \to s_2, ..., v_p \to s_p) &\overset{\text{lorsque } \delta(s)>i}{\equiv} \bigwedge_{i=1}^{p} s \sqsubseteq s_i \\ C(x_i, v_1 \to s_1, v_2 \to s_2, ..., v_p \to s_p) \sqsubseteq s' &\overset{\text{lorsque } i<\delta(s')}{\equiv} \bigwedge_{i=1}^{p} s_i \sqsubseteq s' \end{aligned}$$

5.2.4 Opérateur de widening

Sur nos EDD, on définit une opération $\rho : D_n \times D \to D$ comme suit :

$$\begin{aligned} \rho(t_0, V(t)) &= \begin{cases} \top & \text{si } t = t_0 \\ \bot & \text{sinon} \end{cases} \\ \rho(t_0, C(x_i, v_1 \to s_1, ..., v_p \to s_p)) &= r(x_i, v_1 \to \rho(t_0, s_1), ..., v_p \to \rho(t_0, s_p)) \end{aligned}$$

Explication : cette fonction extrait d'un EDD la fonction booléenne qui mène vers exactement une certaine valeur abstraite des numériques.

On introduit maintenant un opérateur de widening sur nos arbres :

$$\begin{aligned} a \nabla b &= f_\nabla(a, b, a, b) \\ f_\nabla(a, b, V(t), V(t')) &= \begin{cases} V(t \nabla_n t') & \text{si } \rho(t, a) = \rho(t', b) \\ V(t \sqcup_n t') & \text{sinon} \end{cases} \\ f_\nabla(a, b, s, C(x_i, v_1 \to s_1, ..., v_p \to s_p)) &\overset{\text{lorsque } i<\delta(s)}{=} r\begin{pmatrix} x_i, & \begin{matrix} v_1 \to f_\nabla(a, b, s, s_1) \\ \vdots \\ v_p \to f_\nabla(a, b, s, s_p) \end{matrix} \end{pmatrix} \end{aligned}$$

Les autres cas sont définis exactement pareil (cf définition de $\sqcup$, plus on passe a et b à notre fonction f_∇). Explication : lorsque l'on doit faire l'union de deux feuilles, on fait un widening si et seulement si les deux feuilles sont accessibles selon exactement la même formule booléenne sur les énumérés dans a et b.

5.2.5 Itérations chaotiques.

On enrichit un peu notre arbre au niveau des feuilles pour enregistrer quelques informations supplémentaires :

$$\begin{array}{rcl} s & := & V(t)_i \\ & | & V(t)_i^* \\ & | & C(x_i, v_1 \to s_1, v_2 \to s_2, ..., v_p \to s_p) \end{array}$$

L'étoile correspondra à : « cette feuille est nouvelle, il faut l'analyse comme nouveau cas », et l'indice $i \in \mathbb{N}$ correspond à : « cette feuille est là depuis k itérations », où le k permet d'implémenter un délai de widening.

On se donne τ un délai de widening, paramètre de l'analyse. On définit maintenant une fonction d'accumulation $\diamond$ comme suit :

$$\begin{array}{rcl} a \diamond b & = & f_\diamond(a, b, a, b) \\ f_\diamond(a, b, \bot, V(t)) & \overset{\text{lorsque } t \neq \bot_n}{=} & V(t)_0 \\ f_\diamond(a, b, V(t)_i^\nu, \bot) & = & V(t)_i^\nu \\ f_\diamond(a, b, V(t)_i^\nu, V(t')) & = & \left\{ \begin{array}{ll} V(t \nabla_n t')_{i+1}^\nu & \text{si } \rho(t, a) = \rho(t', b) \text{ et } i \geqslant \tau \\ V(t \sqcup_n t')_{i+1}^\nu & \text{sinon} \end{array} \right. \\ f_\diamond(a, b, s, C(x_i, v_1 \to s_1, ..., v_p \to s_p)) & \overset{\text{lorsque } \delta(s) > i}{=} & r\left(x_i, \begin{array}{l} v_1 \to f_\diamond(a, b, s, s_1) \\ \vdots \\ v_p \to f_\diamond(a, b, s, s_p) \end{array} \right) \end{array}$$

(où ν correspond à soit une étoile soit pas d'étoile)

(les autres cas se font par appel récursif encore une fois comme dans le cas de l'union)

Puis une fonction de détection des cas nouveaux par rapport à une valeur précédente :

$$\begin{array}{rcl} \circledast_{s_0}(s) & = & f_\circledast(s_0, s, s) \\ f_\circledast(s_0, s, V(t)_i) & = & \left\{ \begin{array}{ll} V(t)_i^* & \text{si } (\rho(t, s) \sqcap s) \not\sqsubseteq s_0 \\ V(t)_i & \text{sinon} \end{array} \right. \\ f_\circledast(s_0, s, V(t)_i^*) & = & V(t)_i^* \\ f_\circledast(s_0, s, C(x_i, v_1 \to s_1, ..., v_p \to s_p)) & = & C(x_i, v_1 \to f_\circledast(s_0, s, s_1), ..., v_p \to f_\circledast(s_0, s, s_p)) \end{array}$$

Nous sommes maintenant en mesure de décrire le processus d'itérations chaotiques à proprement parler. On commence avec :

$$s_0 \ = \ \circledast_\bot I^\#$$

(appliquer $\circledast$ de la sorte permet de faire que toutes les feuilles soient étoilées)

Puis pour les itérations, deux cas :

- Si il existe $V(t_0)_i^*$ une feuille étoilée dans s_n : on marque s'_n l'arbre s_n où toutes les feuilles $V(t_0)$ sont dé-étoilées, puis :

$$\begin{array}{rcl} c_n & = & \rho(t_0, s_n) \\ D_{n+1}^0 & = & \text{lfp}_{c_n \sqcap s_n}(\lambda i. c_n \sqcap (s_n \sqcup g^\#(i))) \\ D_{n+1} & = & D_{n+1}^0 \sqcup g^\#(D_{n+1}^0) \\ s_{n+1} & = & \circledast_{s'_n}(s'_n \diamond D_{n+1}) \end{array}$$

  (où le point fixe D_{n+1}^0 est fait en faisant appel à $\sqcup$ et ∇ définis précédemment, avec un délai de widening convenable)

  Dans tous les cas, on refait une itération. Les étoiles finiront bien par disparaître.

- Si il n'existe pas de telle feuille étoilée dans s_n :

$$s_{n+1} \ = \ \circledast_{s_n}(s_n \diamond g^\#(s_n))$$

  Dans ce cas, on s'arrête si $s_{n+1} = s_n$.

5.3 Partitionnement dynamique

Une autre approche à base de partitionnement dynamique a été tentée (cf *Dynamic Partitionning in Analyses of Numerical Properties*), mais elle n'a pas donné de très bons résultats par manque d'une heuristique sur les partitionnements à effectuer.

L'idée de base consiste à se donner une liste de conditions $d_1, d_2, ..., d_n$ qui définissent n parties de $S^\#$ (en principe disjointes, mais les approximations mènent parfois à des recoupenents). On étudie les parties (ou *lieux*) $S_i^\# = [\![d_i]\!](S^\#)$ comme un système de transitions pour lequel on calcule un point fixe par itérations chaotiques.

Les conditions d_i sont données par une heuristique qui découpe à chaque rafinement un des lieux $S_i^\#$ en plusieurs sous-lieux.

Découpage de base. Le découpage de base est généralement : init ; non init et propriété vérifiée ; non init et propriété non vérifiée.

Rafinement. On note $S_i^\# \to S_j^\#$ si dans notre sur-approximation on peut être à un cycle dans $S_i^\#$ et dans le suivant dans $S_j^\#$.

On recherche un lieu $S_i^\#$ et une condition c apparaissant dans le programme (ou la négation d'une telle condition) tel qu'il existe un état $S_j^\#$ ayant les propriétés suivantes :

$$\begin{array}{rcl} S_j^\# & \to & S_i^\# \\ S_j^\# & \not\to & [\![c]\!]S_i^\# \end{array}$$

On peut aussi le faire dans l'autre sens :

$$\begin{array}{rcl} S_i^\# & \to & S_j^\# \\ [\![c]\!]S_i^\# & \not\to & S_j^\# \end{array}$$

Si une telle condition est détectée, alors on peut diviser le lieu $S_i^\#$ en deux parties, $[\![c]\!]S_i^\#$ et $[\![\neg c]\!]S_i^\#$, c'est-à-dire qu'on pose comme nouvelles définitions $d_1, ..., d_{i-1}, c \wedge d_i, \neg c \wedge d_i, d_{i+1}, ..., d_n$. On refait ensuite un point fixe. Notre analyse est en principe plus fine, donc on espère trouver des lieux inacessibles.

Observations. En pratique ça ne marche pas très bien car les rafinements ne sont pas effectués dans le bon ordre. Dans le papier sus-cité, une double analyse en avant et en arrière était utilisée pour trouver l'intersection des valeurs accessibles depuis l'état initial et co-accessibles jusqu'à un état qui provoque une erreur (ie dont la définition est : la propriété est violée), mais nous n'avons pas tenté d'implémenter une telle analyse, qui serait sans doute bien plus précise. Remarquons que l'implémentation de B.Jeannet pour LUSTRE est disponnible sur sa page web (c'est l'outil NBac).

6 Implémentation

Le projet scade-analyzer propose une implémentation simple des composants suivants :

- parser, lexer, typeur pour notre sous-ensemble de SCADE (source des fichiers dans `frontend/`)
- interprète pour la sémantique cocnrète (`interpret/interpret.ml`)
- implémentation de la transformation d'un programme en formule logique ; quelques simplifications sur les formules logiques (`abstract/formula.ml`, `abstract/transform.ml`).
- trois domaines numériques basés sur Apron : intervalles, polyèdres et octogones (`abstract/apron_domain.ml`)
- vestige d'une implémentation « maison » d'un domaine non-relationnel à base d'intervalles (`abstract/num_domain.ml`, `abstract/nonrelational.ml`, `abstract/value_domain.ml`, `abstract/intervals_domain.ml`)
- deux domaines abstraits à disjonctions et procédures d'analyse statique correspondantes (`abstract/abs_interp.ml`, `abstract/abs_interp_edd.ml`)

- procédure d'analyse par partitionnement dynamique (`abstract/abs_interp_dynpart.ml`)

En nous basant sur les options de la ligne de commande, nous allons maintenant décrire les différentes fonctionnalités.

6.1 Parsing et affichage de programmes SCADE

- `--dump` : parse un fichier SCADE et le ré-affiche en sortie
- `--dump-rn` : parse un fichier SCADE et le ré-affiche en sortie, après une étape de renommage qui consiste à rendre les noms unique au sein d'un noeud. Cette passe est implémentée dans `frontend/rename.ml`.

6.2 Interprète SCADE

- `--test` : execute le programme SCADE donné en argument, avec l'interprète `interpret/interpret.ml` basé sur la sémantique à réduction. Le programme doit satisfaire la spécification suivante : avoir un noeud `test` qui servira de racine, ce noeud devant prendre une unique entrée, `i`, qui est un compteur incrémenté à chaque cycle, et renvoyant trois entiers, `a`, `b` et `c` (qui seront affichés en sortie), ainsi qu'un booleen `exit` qui indiquera que l'interprète doit terminer. Cf fichiers dans `tests/source/*.scade` pour des exemples.
- `--vtest` : pareil mais affiche plus de détails (tout le contenu de la mémoire est affiché à chaque cycle).

6.3 Analyse statique par interprétation abstraite

6.3.1 Domaine à disjonctions simples

Cette analyse est implémentée dans `abstract/abs_interp.ml`.

Modes d'analyse

- `--ai-itv` : fait une passe d'analyse statique par interprétation abstraite utilisant le domaine à disjonctions simples, et en s'appuyant sur le domaine non-relationnel à base d'intervalles pour la partie numérique
- `--ai-poly` : de même mais utilise le domaine abstrait relationnel basé sur les polyèdres d'Apron pour la partie numérique
- `--ai-oct` : de même mais utilise les octogones

Options de l'analyse

- `--root <noeud>` : spécifie le noeud racine dont on veut faire l'analyse (par défaut : `test`)
- `--ai-vci` : affiche des détails sur le contenu de l'accumulateur à chaque itération
- `--ai-vvci` : affiche encore plus de détails
- `--ai-wd <n>` : définit un délai pour les opérations de widening (par défaut 5). Ce délai est utilisé à deux endroits différents de l'analyse : pour le point fixe local de chaque itération chaotique, et pour le point fixe global des itérations.
- `--disj <vars>` : variables à utiliser comme variables de disjonction (par défaut : aucune). Donner comme argument `all` permet d'utiliser toutes les variables énumérées pour les disjonctions. Donner comme argument `last` permet d'utilsier toutes les variables énumérées qui sont utilisées dans une mémoire. On peut aussi utiliser la syntaxe `last+v1,v2,v3` pour rajouter des variables aux variables mémoire.
- `--no-time <scopes>` : donne un certain nombre de scopes pour lesquels ne pas introduire de variable `time` (par défaut `all`, c'est-à-dire que `time` n'est jamais introduite). Lorsqu'une variable `time` est introduite, on génère les équations qui font en sorte que `time` soit égal au numéro du cycle depuis le début de l'exécution du programme.

- `--init <scopes>` : donne un certain nombre de scopes pour lesquels introduire une variable `init` (par défaut `all`). Il est envisageable de remplacer la variable `init` de chaque scope par une variable `time`, les disjonctions de cas init/non init se feront alors selon la condition time $= 0$ ou time $\geqslant 1$. En ne générant ni variable `time` ni variable `init`, la disjonction n'est pas faite.

6.3.2 Domaine à disjonction par graphe de décision

Cette analyse est implémentée dans `abstract/abs_interp_edd.ml`.

Modes d'analyse

- `--ai-itv-edd` : fait une passe d'analyse statique utilisant le domaine à base d'EDD et en utilisant les intervalles comme domaine de valeurs numériques
- `--ai-poly-edd` : de même mais utilise le domaine abstrait relationnel basé sur les polyèdres d'Apron pour la partie numérique
- `--ai-oct-edd` : de même mais utilise les octogones

Options de l'analyse

- `--root`
- `--ai-vci`, `--ai-vvci`
- `--ai-wd`
- `--no-time`, `--init`
- Non implémenté : paramètre `--disj` pouvant intervenir sur le choix des variables à considérer dans le graphe de décision (actuellement toutes sont nécessairement considérées).

6.3.3 Analyse par partitionnement dynamique

Cette analyse est implémentée dans `abstract/abs_interp_dynpart.ml`.

Modes d'analyse

- `--ai-s-itv-dp` : analyse par partitionnement dynamique, utilise un domaine simple non-relationnel pour les valeurs énumérées et les intervalles pour la partie numérique
- `--ai-edd-itv-dp` : analyse par partitionnement dynamique, utilise des graphes de décision pour représenter les conditions sur les énumérées et les intervalles pour la partie numérique
- `--ai-s-rel-dp` : utilise les polyèdres d'Apron
- `--ai-edd-rel-dp`

Paramètres de l'analyse

- `--root`
- `--ai-wd`
- `--ai-vci`, `--ai-vvci`
- `--no-time`, `--init`
- `--ai-max-dp-depth` : profondeur maximale de partitionnement
- `--ai-max-dp-width` : largeur maximale de partitionnement (ie nombre maximal de parties à considérer)

7 Prolongements envisageables

7.1 Analyse des propriétés des nombres flottants

- Implémenter un domaine d'intervalles permettant de gérer les rationnels en précision arbitraire, ou se brancher sur le module Box d'Apron

- Utiliser un opérateur de widening aproprié (cf Astrée)
- Implémenter la sémantique des nombres flottants machine (ce qui n'est pas simple)

7.2 Analyse de programmes « taille réelle »

- Modification du domaine EDD pour pouvoir ne prendre en compte qu'une partie des variables énumérées (pour l'instant elles sont toutes considérées, ce n'est pas paramétrable). Permettre que les feuilles contiennent également des informations non-relationnelles sur les variables de type énuméré.
- Analyse de base avec des intervalles, plus des « packs » de variables traités avec des domaines plus puissants (octagones, polyhèdres, domaines à disjonctions)
- Analyse par contrats : abstraire certains noeuds par les garanties que vérifient les sorties, plutôt que de considérer le noeud en entrée. En échange, il faut que l'on vérifie que les propriétés en entrée (assume) sont bien vraies.
- L'ordre des équations dans le programme semble avoir un impact sur l'analyse : si celles-ci sont écrites dans l'ordre où elles seront effectivement exécutées (ie triées par ordre de dépendance), l'analyse semble gagner en précision. Implémenter une passe de scheduling (approximatif : on ne veut pas diviser les automates, les blocs activate, ...) que l'on exécuterait comme pré-processing sur le programme.

Références

- N.HALBWACHS, *About synchronous programming and abstract interpretation*, International Symposium on Static Analysis, SAS'94.
- N.HALBWACHS, F.LAGNIER and C.RATEL, *Programming and verifying real-time systems by means of the synchronous data-flow programming language Lustre*, IEEE Transactions on Software Engineering, Spcial Issue on the Specification and Analysis of Real-Time Systems, Sept. 1992
- B.JEANNET, N.HALBWACHS and P.RAYMOND, *Dynamic Partitioning in Analyses of Numerical Properties*, Static Analysis Symposium, SAS'99.