

# 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)

## 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
          | 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  $s: \mathbb{X} \rightarrow \mathbb{V}$  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  $s_0, s_1, \dots, s_n, \dots$  conforme à la spécification du programme et à une série d'entrées. On note :

$$s_0 \xrightarrow{i_1} s_1 \xrightarrow{i_2} s_2 \rightarrow \dots$$

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.

Pour traduire l'init et le reset, on introduit dans chaque scope  $\Sigma$  plusieurs variables :

- $\text{nreset}_\Sigma$  : indique que le scope devra être reset lors du prochain cycle
- $\text{init}_\Sigma$  : indique que le scope est à l'état initial dans ce cycle
- $\text{active}_\Sigma$  : 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 :

- $\text{state}_A$ , qui définit l'état de l'automate à ce cycle
- $\text{nstate}_A$ , 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  $s_0(\text{reset}_/) = \text{true}$ , 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, et pour chaque  $\text{pre}_i e$ , on introduit la variable  $m_i$  qui enregistre la valeur de  $e$  dans l'état et qui sert de mémoire pour le **pre**.

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  $\mathbb{X} \rightarrow \mathbb{V} \cup \{\top\}$ , la valeur  $\top$  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  $\forall x \in \mathbb{X}, s(x) \neq \top$ .

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  $l, s \models e \rightarrow_{\Sigma}^v v$  signifient "avec la mémoire  $l$  et l'état partiellement calculé  $s$ , l'expression  $e$  calculée dans le scope  $\Sigma$  prend la valeur  $v$ ".

### 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 :

$$\frac{l \Rightarrow s}{l \Rightarrow s[\text{act}_{\text{r}} := t][\text{nreset}_{\text{r}} := f]}$$

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

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

$$\begin{aligned} & \frac{l \Rightarrow s \quad l, s \models r_{\Sigma}}{l \Rightarrow s[\text{init}_{\Sigma} := t]} \\ & \frac{l \Rightarrow s \quad l, s \models \neg r_{\Sigma} \quad l(\text{act}_{\Sigma}) = t}{l \Rightarrow s[\text{init}_{\Sigma} := f]} \\ & \frac{l \Rightarrow s \quad l, s \models \neg r_{\Sigma} \quad l(\text{act}_{\Sigma}) = f}{l \Rightarrow s[\text{init}_{\Sigma} := l(\text{init}_{\Sigma})]} \end{aligned}$$

En particulier, pour le scope racine on instancie ces règles avec  $(l, s \models r_{\text{r}}) \Leftrightarrow l(\text{nreset}_{\text{r}}) = t$

### 2.3.3 Règles de réduction d'expressions

Ces règles permettent d'exprimer le calcul d'une expression. On note  $\Sigma$  le scope dans lequel l'expression est évaluée.

$$\begin{aligned} & \overline{l, s \models c \rightarrow_{\Sigma}^v c}, c \in \mathbb{V} \\ & \frac{s(x) \neq \top}{l, s \models x \rightarrow_{\Sigma}^v s(x)}, x \in \mathbb{X} \\ & \overline{l, s \models \text{pre}_i e \rightarrow_{\Sigma}^v l(m_i)} \\ & \frac{l, s \models e_1 \rightarrow_{\Sigma}^v v_1 \quad l, s \models e_2 \rightarrow_{\Sigma}^v v_2}{l, s \models e_1 \odot e_2 \rightarrow_{\Sigma}^v v_1 \odot v_2} \\ & \frac{s(\text{init}_{\Sigma}) = t \quad l, s \models e_1 \rightarrow_{\Sigma}^v v_1}{l, s \models (e_1 \rightarrow e_2) \rightarrow_{\Sigma}^v v_1} \\ & \frac{s(\text{init}_{\Sigma}) = f \quad l, s \models e_2 \rightarrow_{\Sigma}^v v_2}{l, s \models (e_1 \rightarrow e_2) \rightarrow_{\Sigma}^v v_2} \\ & \text{etc...} \end{aligned}$$

### 2.3.4 Règles de réduction pour les pre

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

$$\begin{aligned} & \frac{l \Rightarrow s \quad s(\text{act}_{\Sigma}) = t \quad l, s \models e \rightarrow_{\Sigma}^v v}{l \Rightarrow s[m_i := v]} \\ & \frac{l \Rightarrow s \quad s(\text{act}_{\Sigma}) = f}{l \Rightarrow s[m_i := l(m_i)]} \end{aligned}$$

### 2.3.5 Règles de réduction pour les affectations

Pour toute affectation  $x = e$  apparaissant dans le scope  $\Sigma$ , on donne la règle suivante :

$$\frac{l \Rightarrow s \quad s(\text{act}_\Sigma) = t \quad l, s \models e \rightarrow_\Sigma^v v}{l \Rightarrow s[x := v]}$$

### 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  $\Sigma$ , on crée deux nouveaux scopes  $\Sigma_1$  et  $\Sigma_2$  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  $\Sigma_1$  et  $\Sigma_2$  :

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

$$\frac{l \Rightarrow s \quad s(\text{act}_\Sigma) = t \quad l, s \models c \rightarrow_\Sigma^v t}{l \Rightarrow s[\text{act}_{\Sigma_1} := t][\text{act}_{\Sigma_2} := f]}$$

$$\frac{l \Rightarrow s \quad s(\text{act}_\Sigma) = t \quad l, s \models c \rightarrow_\Sigma^v f}{l \Rightarrow s[\text{act}_{\Sigma_1} := f][\text{act}_{\Sigma_2} := t]}$$

Les deux scopes  $\Sigma_1$  et  $\Sigma_2$  héritent de la condition de reset du scope  $\Sigma$ .

### 2.3.7 Règles de réduction pour les automates

On note  $A$  l'automate,  $s_0$  son état initial. Les règles d'activation des différents scopes des états se font en fonction de la variable  $\text{state}_A$  définie pour l'automate  $A$  comme suit :

$$\frac{l \Rightarrow s \quad l, s \models r_\Sigma}{l \Rightarrow s[\text{state}_A = s_0]}$$

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

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

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

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

Dans tous les scopes  $\Sigma_i$ , 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  $\text{nreset}_{\Sigma_i}$  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 :

$$\frac{l \Rightarrow s \quad s(\text{state}_A) = s_i \quad l, s \models c \rightarrow_\Sigma^v t}{l \Rightarrow s[\text{nstate}_A := s_j]}$$

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  $\text{nreset}_{\Sigma_i}$ .

**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 :

$$\frac{l \Rightarrow s}{l \Rightarrow s[\text{act}_\ell := t][\text{nreset}_\ell := f]}$$

$$\frac{l \Rightarrow s \quad l(\text{nreset}_\ell) = t}{l \Rightarrow s[\text{init}_\Sigma := t]}$$

$$\frac{l \Rightarrow s \quad l(\text{nreset}_\ell) = f \quad l(\text{act}_\ell) = t}{l \Rightarrow s[\text{init}_\ell := f]}$$

$$\frac{l \Rightarrow s \quad l(\text{nreset}_\ell) = f \quad l(\text{act}_\ell) = f}{l \Rightarrow s[\text{init}_\ell := l(\text{init}_\ell)]}$$

Affectation  $c = a - b$  :

$$\frac{l \Rightarrow s \quad s(\text{act}_\ell) = t \quad l, s \models a - b \rightarrow_\ell^v v}{l \Rightarrow s[c := v]}$$

Affectations de  $la$  et  $lb$  :

$$\frac{l \Rightarrow s \quad s(\text{act}_\ell) = t \quad l, s \models (0 \rightarrow \text{pre}_1) a \rightarrow_\ell^v v}{l \Rightarrow s[la := v]}$$

$$\frac{l \Rightarrow s \quad s(\text{act}_\ell) = t \quad l, s \models (0 \rightarrow \text{pre}_2) b \rightarrow_\ell^v v}{l \Rightarrow s[lb := v]}$$

Affectation de  $half$  :

$$\frac{l \Rightarrow s \quad s(\text{act}_\ell) = t \quad l, s \models (\text{true} \rightarrow \text{not pre}_3 \text{ half}) \rightarrow_\ell^v v}{l \Rightarrow s[\text{half} := v]}$$

Mémorisation des valeurs pour  $\text{pre } a$ ,  $\text{pre } b$  et  $\text{pre } half$  :

$$\frac{l \Rightarrow s \quad s(\text{act}_\ell) = t \quad l, s \models a \rightarrow_\ell^v v}{l \Rightarrow s[m_1 := v]}$$

$$\frac{l \Rightarrow s \quad s(\text{act}_\ell) = t \quad l, s \models b \rightarrow_\ell^v v}{l \Rightarrow s[m_2 := v]}$$

$$\frac{l \Rightarrow s \quad s(\text{act}_\ell) = t \quad l, s \models \text{half} \rightarrow_\ell^v v}{l \Rightarrow s[m_3 := v]}$$

$$\frac{l \Rightarrow s \quad s(\text{act}_\ell) = f}{l \Rightarrow s[m_1 := l(m_1)]}$$

$$\frac{l \Rightarrow s \quad s(\text{act}_\ell) = f}{l \Rightarrow s[m_2 := l(m_2)]}$$

$$\frac{l \Rightarrow s \quad s(\text{act}_\ell) = f}{l \Rightarrow s[m_3 := l(m_3)]}$$

Activation des deux moitiés du activate :

$$\frac{l \Rightarrow s \quad s(\text{act}_/) = t \quad l, s \models \text{half} \rightarrow t}{l \Rightarrow s[\text{act}_/1 := t][\text{act}_/2 := f]}$$

$$\frac{l \Rightarrow s \quad s(\text{act}_/) = t \quad l, s \models \text{half} \rightarrow f}{l \Rightarrow s[\text{act}_/1 := f][\text{act}_/2 := t]}$$

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

Affectation de **a** et **b** dans la première moitié :

$$\frac{l \Rightarrow s \quad s(\text{act}_/1) = t \quad l, s \models \text{la} + 1 \rightarrow_{/1}^v v}{l \Rightarrow s[a := v]}$$

$$\frac{l \Rightarrow s \quad s(\text{act}_/1) = t \quad l, s \models \text{lb} \rightarrow_{/1}^v v}{l \Rightarrow s[b := v]}$$

Et dans la deuxième moitié :

$$\frac{l \Rightarrow s \quad s(\text{act}_/2) = t \quad l, s \models \text{la} \rightarrow_{/2}^v v}{l \Rightarrow s[a := v]}$$

$$\frac{l \Rightarrow s \quad s(\text{act}_/2) = t \quad l, s \models \text{lb} + 1 \rightarrow_{/2}^v v}{l \Rightarrow s[b := v]}$$

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  $\square$  correspond à la fonction  $e \mapsto e$ , une formule  $a \wedge x = \square$  correspond à la fonction  $e \mapsto a \wedge x = e$ , etc. L'argument d'une fonction trou peut aussi être un couple d'expressions, ainsi  $\square_1 + \square_2$  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  $\Sigma$  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 généralement être une formule booléenne.

On note  $\odot$  n'importe quel opérateur binaire :  $+$ ,  $-$ ,  $\times$ ,  $/$ ,  $\text{mod}$ ,  $<$ ,  $>$ ,  $\leq$ ,  $\geq$ ,  $=$ ,  $\neq$ ,  $\wedge$ ,  $\vee$ .

Les règles sont les suivantes :

$$\begin{aligned} T(\Sigma, (e_1, e_2, \dots, e_n), w) &= w[T(\Sigma, e_1, \square), \dots, T(\Sigma, e_n, \square)] \\ \forall c \in \mathbb{V}, \quad T(\Sigma, c, w) &= w[c] \\ \forall x \in \mathbb{X}, \quad T(\Sigma, x, w) &= w[s(x)] \\ T(\Sigma, \text{pre}_i e, w) &= w[l(m_i)] \\ T(\Sigma, e_1 \rightarrow e_2, w) &= (s(\text{init}_\Sigma) \wedge T(\Sigma, e_1, w)) \vee (\neg s(\text{init}_\Sigma) \wedge T(\Sigma, e_2, w)) \\ T(\Sigma, e_1 \odot e_2, w) &= T_n(\Sigma, (e_1, e_2), w[\square_1 \odot \square_2]) \\ T(\Sigma, \text{if } c \text{ then } e_1 \text{ else } e_2, w) &= (T(\Sigma, c, \square) \wedge T(\Sigma, e_1, w)) \vee (\neg T(\Sigma, c, \square) \wedge T(\Sigma, e_2, w)) \\ T(\Sigma, -e, w) &= T(\Sigma, e, w[\neg \square]) \\ T(\Sigma, \neg e, w) &= T(\Sigma, e, w[\neg \square]) \end{aligned}$$

Par la suite, une affectation  $x = e$  sera traduite par la formule booléenne  $T(\Sigma, e, x = \square)$ .

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

$$\begin{aligned} F &= T(\Sigma, \text{if } y \geq 0 \text{ then } y \text{ else } -y, x = \square) \\ &= (T(\Sigma, y \geq 0, \square) \wedge T(\Sigma, y, x = \square)) \vee (\neg T(\Sigma, y \geq 0, \square) \wedge T(\Sigma, -y, x = \square)) \\ &= (y \geq 0 \wedge x = y) \vee (\neg(y \geq 0) \wedge x = -y) \end{aligned}$$

**Exemple 3.** Effectuons la traduction de  $x = 0 \rightarrow \text{pre}_1 x + 1$ .

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

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  $(c \wedge T_{\text{active}}(b_1) \wedge T_{\text{inactive}}(b_2)) \vee (\neg c \wedge T_{\text{active}}(b_2) \wedge T_{\text{inactive}}(b_1))$ .

Tout cela 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 :

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

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 définie plus haut.

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 affectation  $x = e$ , rajouter dans  $s$  pour la variable  $x$  la fonction qui calcule  $e$  et rajoute la valeur trouvée dans  $s$ .
- Pour un bloc active, rajouter dans  $s$  pour toutes les variables retournées 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 retournées 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.

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

/!\ travaux travaux travaux /!\

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 approximations 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  $\mathbb{X} = \mathbb{X}_e \cup \mathbb{X}_n$  : c'est l'union des variables de type énumération et des variables de type numérique.

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

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

$$I = \{s \in \mathbb{M} \mid s(\text{nreset}) = t\}$$

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 valeurs des *pre*.

Vision habituelle : on a une suite d'états  $s_0, s_1, \dots$  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  $s_{n+1}$  :

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

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 *last*, c'est-à-dire qu'on a leur valeur du cycle précédent.

La fonction de cycle  $c: \mathbb{M} \rightarrow \mathcal{P}(\mathbb{M})$  s'occupe de faire passer les valeurs suivantes aux valeurs actuelles (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 ensemble de variables  $C$ , qui sont les variables qui nous intéresseront lors du calcul de la transition :

$$c(s) = \{s' \in \mathbb{M} \mid \forall x \in C, s'(Lx) = s(x)\}$$

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  $f: \mathbb{M} \rightarrow \mathbb{M}$  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 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 SCADÉ quel 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  $q: \mathbb{M} \rightarrow \{t, f\}$  qui nous dit si un environnement est conforme à la spécification.

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

$$\begin{aligned} s_0 &= \{s \in \mathbb{M} \mid s(\text{nreset}) = t\} \\ s_{n+1} &= g(s_n) \\ g(x) &= \bigcup_{s \in x} \{f(a), a \in c(s), q(f(a)) = t\} \end{aligned}$$

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 :

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

$S$  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 :

$$\gamma_{\mathbb{M}} : \mathcal{D}^\# \rightarrow \mathcal{P}(\mathbb{M})$$

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

$$\alpha_{\mathbb{M}} : \mathcal{P}(\mathbb{M}) \rightarrow \mathcal{D}^\#$$

Cette fonction fait correspondre à une partie de  $\mathbb{M}$  sa meilleure approximation dans le domaine abstrait (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).

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  $\gamma_{\text{int}}$  et  $\gamma_{\text{poly}}$  associées.

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

Pour  $s \in \mathbb{M}$  et  $e \in \mathbb{E}$ , on note  $s \models e$  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  $\llbracket e \rrbracket : \mathcal{D}^\# \rightarrow \mathcal{D}^\#$  qui restreint l'abstraction  $s^\#$  en une sur-approximation (la meilleure possible) de  $\alpha(\{s \in \gamma(s^\#) \mid s \models e\})$  :

$$\{s \in \gamma(s^\#) \mid s \models e\} \sqsubseteq \gamma(\llbracket e \rrbracket(s^\#))$$

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 :

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

(cette définition n'est pas constructive car on n'implémente jamais  $\gamma$  directement)

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

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  $s_0$  est l'état initial du système et est défini par  $I^\# = \llbracket i \rrbracket(\top)$ , où  $i$  est une équation du type  $\text{Lreset}_j = \text{tt}$ .

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  $+, -, \times, \div, =, \geq, \neq$ . 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 des types des variables, les opérateurs sont  $\equiv, \neq$ . 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 :

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

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

$$\begin{aligned}\forall x \in \mathbb{X}_b, \llbracket x = \text{tt} \rrbracket &= \llbracket x = 1 \rrbracket_{\text{poly}} \\ \forall x \in \mathbb{X}_b, \llbracket x = \text{ff} \rrbracket &= \llbracket x = 0 \rrbracket_{\text{poly}} \\ \forall x, y \in \mathbb{X}_b, \llbracket x = y \rrbracket &= \llbracket x = y \rrbracket_{\text{poly}} \\ \forall x, y \in \mathbb{X}_b, \llbracket x \neq y \rrbracket &= \llbracket x = 1 - y \rrbracket_{\text{poly}}\end{aligned}$$

Les résultats sont généralement plus que médiocres. De plus, on ne peut pas représenter ainsi les valeurs d'énumérations ayant plus de deux éléments.

## 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 deux 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  $Q = \{\text{up}, \text{down}, \text{left}, \text{right}, \text{stay}\}$ , 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  $\llbracket e \rrbracket$ .

### 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  $\mathcal{D}^\#$  capable de gérer les contraintes sur les variables numériques et sur les variables énumérées, mais sans relation entre les deux. Le domaine  $\mathcal{D}^\#$  représente une abstraction de  $\mathbb{M}_n = (\mathbb{X}_n \cup \mathbb{X}_e) \rightarrow \mathbb{V}$ . On note  $\perp_n$  et  $\top_n$  les éléments bottom et top de ce treillis,  $\sqcup_d$  et  $\sqcap_d$  les bornes inf et sup de ce treillis.

La particularité des variables d'état est que l'on ne réalise pas d'abstraction sur celles-ci : on représente directement un état par une valuation de ces variables, dans  $\mathbb{X}_d \rightarrow \mathbb{V}_d = \mathbb{M}_d$ . On peut le munir d'une structure de treillis :  $\mathbb{M}_d \cup \{\perp, \top\}$  forme un treillis plat :  $\forall x, y \in \mathbb{M}_d, x \sqsubseteq y \Leftrightarrow x = y$ ,  $\forall x \in \mathbb{M}_d, \perp \sqsubseteq x \sqsubseteq \top$  (l'utilité de cette structure n'apparaît pas.)

On appelle toujours  $\mathbb{M} = \mathbb{X} \rightarrow \mathbb{V}$ , 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  $\{s \in \mathbb{M} \mid \forall x \in \mathbb{X}_d, s(x) = \mathbb{M}_d(x)\}$ . De même on identifie  $e \in \mathbb{M}_n$  à  $\{s \in \mathbb{M} \mid \forall x \in \mathbb{X}_n \cup \mathbb{X}_e, s(x) = \mathbb{M}_n(x)\}$ .

On construit maintenant le domaine abstrait disjonctif comme suit :

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

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

$$\begin{aligned}\perp_d &= \lambda d. \perp_n \\ \top_d &= \lambda d. \top_n\end{aligned}$$

On vérifie bien que  $\gamma_d(\perp_d) = \emptyset$  et  $\gamma_d(\top_d) = \mathbb{M}$ .

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

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

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

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

$$\begin{aligned}\llbracket x = y \rrbracket_d(s) &= \lambda d. \begin{cases} s(d) & \text{si } d(x) = d(y) \\ \perp_n & \text{sinon} \end{cases} \\ \llbracket x \neq y \rrbracket_d(s) &= \lambda d. \begin{cases} s(d) & \text{si } d(x) \neq d(y) \\ \perp_n & \text{sinon} \end{cases}\end{aligned}$$

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

$$\begin{aligned}\llbracket x = v \rrbracket_d(s) &= \lambda d. \begin{cases} s(d) & \text{si } d(x) = v \\ \perp_n & \text{sinon} \end{cases} \\ \llbracket x \neq v \rrbracket_d(s) &= \lambda d. \begin{cases} s(d) & \text{si } d(x) \neq v \\ \perp_n & \text{sinon} \end{cases}\end{aligned}$$

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

$$\forall e \in \mathbb{E}_n \cup \mathbb{E}_e, \llbracket e \rrbracket_d(s) = \lambda d. \llbracket e \rrbracket(s(d))$$

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

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

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 accessibles par le système est  $S^\# = \text{lfp}_{I^\#}(\lambda s. I^\# \sqcup g^\#(s))$ , où  $I^\# = \llbracket i \rrbracket_d(\top_d)$ .

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

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

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

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

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

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 :

$$\begin{aligned}
s_0 &= I^\# \\
\delta_0 &= \{d \in \mathbb{M}_d \mid s_0(d) \neq \perp_n\} \\
K_{\nabla,0} &= \emptyset \\
a_{n+1} &\in \delta_n \text{ (choisi arbitrairement)} \\
D_{n+1} &= g^\#(r_{a_{n+1}}(s_n)) \\
s_{n+1} &= \lambda d. \begin{cases} s_n(d) \nabla D_{n+1}(d) & \text{si } d \in K_{\nabla,n} \\ s_n(d) \sqcup D_{n+1}(d) & \text{sinon} \end{cases} \\
K_{\nabla,n+1} &= K_{\nabla,n} \cup \{d \in \mathbb{M}_d \mid s_n(d) \neq \perp \wedge s_{n+1}(d) \neq s_n\} \\
\delta_{n+1} &= (\delta_n \setminus \{a_{n+1}\}) \cup \{d \in \mathbb{M}_d \mid s_{n+1}(d) \neq s_n(d)\}
\end{aligned}$$

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. Cherchons une autre forme d'itération, par exemple :

$$\begin{aligned}
s_0 &= I^\# \\
\delta_0 &= \{d \in \mathbb{M}_d \mid s_0(d) \neq \perp_n\} \\
K_{\nabla,0} &= \emptyset \\
a_{n+1} &\in \delta_n \text{ (choisi arbitrairement)} \\
D_{n+1}^0 &= \text{lfp}_{r_{a_{n+1}}(s_n)}(\lambda i. r_{a_{n+1}}(s_n \sqcup g^\#(i))) \\
D_{n+1} &= D_{n+1}^0 \sqcup g^\#(D_{n+1}^0) \\
s_{n+1} &= \lambda d. \begin{cases} s_n(d) \nabla D_{n+1}(d) & \text{si } d \in K_{\nabla,n} \text{ et } d \neq a_{n+1} \\ s_n(d) \sqcup D_{n+1}(d) & \text{sinon} \end{cases} \\
K_{\nabla,n+1} &= K_{\nabla,n} \cup \{d \in \mathbb{M}_d \mid s_n(d) \neq \perp \wedge s_{n+1}(d) \neq s_n\} \\
\delta_{n+1} &= (\delta_n \setminus \{a_{n+1}\}) \cup \{d \in \mathbb{M}_d \mid s_{n+1}(d) \neq s_n(d)\}
\end{aligned}$$

Où le point fixe  $D_{n+1}^0$  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 arbre de disjonctions

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 mémoïsation 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 mémoïsation et du partage).

**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).

**Domaine numérique.** On note  $D_n$  le domaine des valeurs numériques et  $\sqcup_n, \sqcap_n, \llbracket \cdot \rrbracket_n, \perp_n, \top_n, \sqsubseteq_n, \gamma_n, \nabla_n$  les éléments correspondants dans ce domaine. On considère que  $\gamma_n : D_n \rightarrow \mathcal{P}(\mathbb{X}_e \cup \mathbb{X}_n \rightarrow \mathbb{V})$  donne toutes les valuations possibles pour les variables de  $\mathbb{X}_e$ .

**Les EDD.** On définit un ordre sur les variables de  $\mathbb{X}_e$  :  $\mathbb{X}_e = \{x_1, x_2, \dots, x_n\}$  (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 :

$$\begin{aligned} s &:= V(t \in D_{\text{num}}) \\ &\mid C(x_i, v_1 \rightarrow s_1, v_2 \rightarrow s_2, \dots, v_p \rightarrow s_p) \end{aligned}$$

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

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

$$\begin{aligned} \delta(C(x_i, v_1 \rightarrow s_1, \dots, v_p \rightarrow s_p)) &= i \\ \delta(V(t)) &= \infty \end{aligned}$$

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

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

La concrétisation est définie comme suit :

$$\begin{aligned} \gamma(V(t)) &= \gamma_n(t) \\ \gamma(C(x_i, v_1 \rightarrow s_1, \dots, v_p \rightarrow s_p)) &= \bigcup_{j=1}^p \{s \in \gamma(s_j) \mid s(x_i) = v_j\} \end{aligned}$$

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

$$\begin{aligned} \top &= V(\top_n) \\ \perp &= V(\perp_n) \end{aligned}$$

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

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

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

$$\begin{aligned} V(t) \sqcap V(t') &= V(t \sqcap_n t') \\ C(x_i, v_1 \rightarrow s_1, \dots, v_p \rightarrow s_p) \sqcap C(x_i, v_1 \rightarrow s'_1, \dots, v_p \rightarrow s'_p) &= r(x_i, v_1 \rightarrow s_1 \sqcap s'_1, \dots, v_p \rightarrow s_p \sqcap s'_p) \\ C(x_i, v_1 \rightarrow s_1, \dots, v_p \rightarrow s_p) \sqcap s' &\stackrel{\text{lorsque } i < \delta(s')}{=} r \left( x_i, \begin{matrix} v_1 \rightarrow s_1 \sqcap s' \\ \vdots \\ v_p \rightarrow s_p \sqcap s' \end{matrix} \right) \end{aligned}$$

et symmétriquement 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  $\llbracket e \rrbracket$  par :

$$\begin{aligned} \llbracket e \rrbracket(V(t)) &= V(\llbracket e \rrbracket_n(t)) \\ \llbracket e \rrbracket(C(x_i, v_1 \rightarrow s_1, \dots, v_p \rightarrow s_p)) &= r(x_i, v_1 \rightarrow \llbracket e \rrbracket(s_1), \dots, v_p \rightarrow \llbracket e \rrbracket(s_p)) \end{aligned}$$

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

$$\begin{aligned} c(x \equiv v) &= C(x, v \rightarrow V(\top), v' \rightarrow V(\perp), v' \in \mathbb{V}_e \setminus \{v\}) \\ c(x \not\equiv v) &= C(x, v \rightarrow V(\perp), v' \rightarrow V(\top), v' \in \mathbb{V}_e \setminus \{v\}) \\ c(x_i \equiv x_j) &\stackrel{\text{lorsque } i < j}{=} C(x_i, v_1 \rightarrow c(x_j \equiv v_1), \dots, v_p \rightarrow c(x_j \equiv v_p)) \\ c(x_i \not\equiv x_j) &\stackrel{\text{lorsque } i < j}{=} C(x_i, v_1 \rightarrow c(x_j \not\equiv v_1), \dots, v_p \rightarrow c(x_j \not\equiv v_p)) \end{aligned}$$

et symmétriquement lorsque  $j > i$ .

On peut ensuite poser :

$$\llbracket e \rrbracket(s) = c(e) \sqcap s$$

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 :

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

**Opérateur de widening.** Sur nos EDD, on définit une opération  $\rho: D_n \times D \rightarrow D$  comme suit :

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

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 :

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

Les autres cas sont définis exactement pareil (cf définition de  $\sqcup$ , plus on passe  $a$  et  $b$  à notre fonction  $f_{\nabla}$ ). 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  $b$ .

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

$$\begin{array}{l}
 s := V(t)_i \\
 \quad | \quad V(t)_i^* \\
 \quad | \quad C(x_i, v_1 \rightarrow s_1, v_2 \rightarrow s_2, \dots, v_p \rightarrow s_p)
 \end{array}$$

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  $n$  itérations", où le  $n$  permet d'implémenter un délai de widening.

On définit maintenant une fonction d'accumulation  $\diamond$  comme suit ( $\tau$  correspond à un délai de widening) :

$$\begin{aligned}
 a \diamond b &= f_{\diamond}(a, b, a, b) \\
 f_{\diamond}(a, b, V(\perp_n), V(t)) & \stackrel{\text{lorsque } t \neq \perp_n}{=} V(t)_0 \\
 f_{\diamond}(a, b, V(t)_i^{\nu}, V(\perp_n)) &= V(t)_i^{\nu} \\
 f_{\diamond}(a, b, V(t)_i^{\nu}, V(t')) &= \begin{cases} V(t \nabla_n t')_{i+1}^{\nu} & \text{si } \rho(t, a) = \rho(t', b) \wedge i \geq \tau \\ V(t \sqcup_n t')_{i+1}^{\nu} & \text{sinon} \end{cases} \\
 f_{\diamond}(a, b, s, C(x_i, v_1 \rightarrow s_1, \dots, v_p \rightarrow s_p)) & \stackrel{\text{lorsque } \delta(s) > i}{=} r \left( x_i, \begin{matrix} v_1 \rightarrow f_{\diamond}(a, b, s, s_1) \\ \vdots \\ v_p \rightarrow f_{\diamond}(a, b, s, s_p) \end{matrix} \right)
 \end{aligned}$$

(où  $\nu$  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 :

$$\begin{aligned}
 \circledast_{s_0}(s) &= f_{\circledast}(s_0, s, s) \\
 f_{\circledast}(s_0, s, V(t)_i) &= \begin{cases} V(t)_i^* & \text{si } (\rho(t, s) \sqcap s) \not\sqsubseteq s_0 \\ V(t)_i & \text{sinon} \end{cases} \\
 f_{\circledast}(s_0, s, V(t)_i^*) &= V(t)_i^* \\
 f_{\circledast}(s_0, s, C(x_i, v_1 \rightarrow s_1, \dots, v_p \rightarrow s_p)) &= C(x_i, v_1 \rightarrow f_{\circledast}(s_0, s, s_1), \dots, v_p \rightarrow f_{\circledast}(s_0, s, s_p))
 \end{aligned}$$

On commence avec :

$$s_0 = \circledast_{\perp} I^{\#}$$

(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 :

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

(où le point fixe  $D_{n+1}^0$  est fait en faisant appel à  $\sqcup$  et  $\nabla$  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  $s_{n+1} = s_n$ .