2014-06-10. Question : c'est quoi faire de l'interprétation abstraite sur du langage synchrone ? Non résolu. Ubuntu Gnome c'est bien. Git clone KCG, tests, lecture document d'introduction.

Par exemple on aimerait pouvoir vérifier les assert suivants :

```
node updown() returns (x: int)
let
  automaton
    initial state UP
    unless when x >= 10 resume DOWN
      x = 0 -> pre x + 1

    state DOWN
    unless when x <= -10 resume UP
      x = 0 -> pre x - 1

  returns x;
  guarantee x >= -10 && x <= 10;
tel
```

Ou, reprenons le rate limiter d'A.Miné :

```
node limiter(x: int, d: int) returns (y: int)
  var s: int, r: int
  let
    assume x >= -128 && x <= 128;
    assume d >= 0 && d <= 16;
    s = 0 -> pre y;
    r = x - s;
    y = if r <= -d then s - d
        else if r >= d then s + d
        else x;
    guarantee y >= -128 && y <= 128;
  tel
```

2014-06-11. Subset minimal de langage à supporter :

```
decl := CONST id: type = expr;
      | NODE (var_def;*) RETURNS (var_def;*) VAR var_def;* body
var_def := (PROBE? id),* : type (last = expr)?
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 := [définition habituelle]
```

TODO :

- Parser un peu plus complet (ENUMs, merge, case, automates basiques)
- Analyse de noms (déroulage des instances ou noms avec des chemins...), analyse de typage basique, analyse de nombre d'étapes précédentes à garder, scheduling
- Trouver une interface générique, ou peut-être pas, pour interpréter concrètement (avec des entrées définies) et abstraitement (sur toutes les entrées possibles satisfiant certaines conditions). Problème : il faut avant tout faire le scheduling, mais celui-ci est fait sur du langage sans automate. Faut-il faire la traduction automates vers core dataflow ?

  Autre solution : supposer que l'on fait de petits exemples et faire le scheduling à la volée.

  Autre solution : s'arranger au niveau du déroulage, de l'instanciation et du renommage des variables, puis faire un scheduling au début (puisque l'ordre d'évaluation des variables ne dépend en théorie pas des états des automates). On enregistre juste la liste des noms de variables dans l'ordre où il faut les évaluer. Cette information sert pour l'interprétation concrète et abstraite.
- Interprète concret
- En profiter pour faire un mini-compilo ! P.S. : peut-on compiler les automates directement ? Avec des `enum` et des `structs` imbriquées...
- Abstraction simple, puis c'est parti pour de la vraie interprétation abstraite !

2014-06-12. Implémentons le parsing des automates. Abandon des structures clockées : when, merge, ... les automates feront l'affaire, et on fait de l'interprétation *sur des automates*.

Problème : supposons le code suivant :

```
automaton
  initial state A
  let x = 0 -> pre x + 1; tel
```

```
  until if x >= 10 resume B;
  state B
  let x = 0 -> pre x - 1; tel
  until if x <= -10 resume A;
returns x
```

Le `pre x` n'est pas partagé ! Chaque scope enregistre ses propre `pre`, en son temps propre. Dans le subset que nous avons choisi, puisqu'il n'y a pas de `when`, seuls les automates sont susceptibles d'introduire ce genre de problèmes.

Supposons que l'on fasse un interprète concret dans un premier temps :

- Il faut travailler de manière persistante : on garde quelque part l'état précédent, on spécifie les entrées, puis on "tire" sur les sorties : l'état suivant est calculé à la volée (ordonnancement dynamique).
- État initial : `init=true`, et tous les automates sont dans leur état par défaut.
- Imaginer une fonction `activate_scope` que l'on appelle quand on initialise un noeud/un scope. Appellé à deux moments : à l'initialisation, quand on active le scope racine, et quand un automate change d'état.
- Chaque scope a sa variable `init` !!

Voilà, c'est plus où moins fait pour les cas de base. Reste : les automates...

Info : parmi les fichiers du dossier `tests/`, le fichier `limiter.scade` ainsi que tous les `tests/test*.scade` sont validés. Le fichier `updown.scade` devrait être validé lorsque les automates seront implémentés. Le fichier `merge.scade` ne pourra pas être passé puisque j'ai choisi de ne pas implémenter les horloges. Le fichier `train.scade` utilise des transitions fortes donc ne passera pas, et de toute façon il est trop compliqué. Le fichier `count.scade` est inutile, c'est juste des tests pour moi.

2014-06-13. Réunion. Notes :

- Il faut impérativement être capable de décider dynamiquement quels blocs activer, en particulier en fonction de variables qui seront calculées à ce cycle-ci. En particulier cela permet d'implémenter les blocs activate. Cela pourrait aussi permettre d'implémenter les transitions fortes, mais il ne faut pas s'embêter avec le mélange faible/fort.
- Les blocs activate et les automates définissent certaines variables (clause `return`). Cette clause donne les mêmes informations que le LHS d'une affectation.
- En numérotant les points de programme (blocs, instances, ...) et en informant l'utilisateur de ces points de programme, on peut donner des informations de débug plus précises par exemple sur des cycles combinatoires.
- Construction `last` : on verra plus tard
- Envisager de faire des transitions resettantes pour les automates ! Du coup, il faut mettre à part la fonction reset pour pouvoir l'appeller à d'autres moments qu'à la phase d'initialisation de tout le système (essentiellement c'est la même fonction qu'il faut reprendre). On peut aussi envisager la construction `reset mon_noeud when un_signal`, c'est pas beaucoup plus compliqué.

La prochaine fois prendre une casquette pour aller manger !

Question : quelle structure de contrôle faut-il maintenant avoir pour gérer ça ? En gros des appels récursifs sur une structure environnement qui mémo(r)ise :

- Les variables dont on connait la valeur
- Les états pris par les automates ; les blocs sélectionnés dans les activate

2013-06-16. Nouvelle semaine. Implémentation des automates et des blocs activate dans l'interprète V2 (en fait la "bonne organisation" du code déjà écrit a rendu ça assez rapide). Remarques :

- La fonction reset scope n'effectue le reset que pour le cycle suivant. Pour implémenter des strong transitions, il faudrait pouvoir le faire aussi pour le cycle courant pour un bloc qui n'a pas encore été activé. Ça tombe bien, j'ai décidé de laisser tomber les strong transition.
- Pas grand chose d'autre... les transitions faibles sont prises à l'étape `do_weak_transitions`. On aurait sans doute pu les faire à la volée dans le `extract_st`.

Nous en revenons donc à la question originelle : comment faire de l'interprétation *abstraite* cette-fois ci ? Idées :

- Pour évacuer le problème des automates en parallelle et des automates hiérarchiques, on applatit tout en considérant le produit cartésien des ensembles d'états de tous les automates définis par le programme. Problème : il faut gérer quels automates sont actifs (et ont donc le droit de faire des transitions) à chaque instant.

  Le problème de base reste l'explosion du nombre de combinaisons. Pour des cas simples ce n'est pas grave. Les cas compliqués, on les traîtera plus tard. Ça serait néanmoins un achievement important que d'arriver à une bonne solution pour gérer ce problème.

- Pour chaque état de ce gros automate on va travailler sur un environnement abstrait qui représente la partie des états possibles du programme où l'automate est dans cet état. (merci le triple emploi du mot "état" !)
- Plus précisément, pour chaque état on va diviser cela en au moins deux domaines abstrait : S_q^{in} et S_q^{stay}. La partie in est celle qui grossit par toutes les transitions qui ammènent sur q, alors que la partie stay est celle qui grossit quand on est déjà sur q et qu'on n'emprunte aucune transition. (Attention : emprunter une transition qui rammène sur le même état q n'est pas considéré dans stay).

  Intérêt : dans S_q^{stay}, on sait que toutes les conditions de sortie (ie toutes les gardes de transitions sortantes depuis q) sont fausses.

- Pour chaque état q du système, on peut noter $f_q(i, S) = (o, S')$ le fait qu'être dans l'état q avec S comme état du système interne, et en recevant les entrées i, on peut passer à l'état S' interne en émettant les sorties o.
- Si l'état q a des transitions $q \xrightarrow{c_1} q_1$, $q \xrightarrow{c_2} q_2$, ..., $q \xrightarrow{c_n} q_n$, où les c_i sont les gardes, et que pour S une partie de S_q^{stay} ou de S_q^{new} on a $f_q(i, S) = (o, S')$ (où i représente n'importe quelle entrée, voir toutes les entrées possibles, conforme à une spécification (directives assume)), alors l'état S' peut être découpé en plusieurs parties : $c_1(S')$ est rajouté à $S_{q_1}^{\text{new}}$, $(c_2 \wedge \neg c_1)(S')$ est rajouté à $S_{q_2}^{\text{new}}$, etc jusqu'à $(\neg c_1 \wedge \cdots \wedge \neg c_n)(S')$ qui est rajouté à S_q^{stay}.

  On fait ainsi grossir les S_q^{stay} et S_q^{new} jusqu'à arriver à un point fixe.

- On peut même envisager un découpage plus fin de la partie S_q^{new}, puisque c'est sur elle que l'on va se retrouver à faire des unions qui peuvent perdre beaucoup d'information (si l'on vient d'autres états très variés). On peut par exemple isoler les parties de S_q^{new} en fonction de l'état d'où l'on vient, du fait que l'on a fait un reset ou non, etc.

- Il faut définir exactement ce que contient l'état interne S. Dans l'interprète concret : il ne contient que les valeurs des `pre`, des variables `init` et des états des automates. En pratique, il faut maintenant être capable de reconstituer les gardes des transitions, donc il vaut mieux y enregistrer toutes les variables du système.

- Le problème d'ordonnancement n'a pas disparu.

- Le langage dataflow synchrone déclare les variables en même temps, il est donc adapté de représenter une étape comme étant non pas une suite d'affectations (vision impérative) mais comme un ensemble de contraintes :

$$x = 0 \to \mathrm{pre}\, x + 1 \quad \Leftrightarrow \quad \begin{cases} \mathrm{init} \\ x = 0 \end{cases} \vee \begin{cases} \neg\mathrm{init} \\ x = x^{\mathrm{pre}} + 1 \end{cases}$$

$$\begin{array}{lll} \text{activate} & \text{if} & \mathrm{pre}\, y \geqslant 10 \\ & \text{then} & y = 0 \\ & \text{else} & y = 0 \to \mathrm{pre}\, y + 1 \end{array} \quad \Leftrightarrow \quad \begin{cases} y^{\mathrm{pre}} \geqslant 10 \\ y = 0 \end{cases} \vee \begin{cases} \mathrm{init} \\ y^{\mathrm{pre}} < 10 \\ y = 0 \end{cases} \vee \begin{cases} \neg\mathrm{init} \\ y^{\mathrm{pre}} < 10 \\ y = y^{\mathrm{pre}} + 1 \end{cases}$$

  Problème : des conditions un peu plus évoluées (notamment avec des $\wedge$ et des $\vee$) se traduisaient par des intersections/unions de plusieurs domaines, et n'étaient pas gérées directement dans l'AST des contraintes. En fait c'est probablement gérable ici exactement de la même façon...

  Problème : il faut faire des disjonctions pour tous les cas possibles (branches des activate, if dans les expressions, ...) ce qui finit par faire beaucoup de cas !

  Il pourrait être intéressant de prendre un programme, on considère que l'on sait dans quels états sont les automates, puis on transforme le programme en une grosse formule logique qui représente le lien entre l'état passé et l'état suivant. Considérer que l'on sait aussi quels scopes sont dans un état init ou pas. On rajoute dans les contraintes des équations du type $\mathrm{next}\,(\mathrm{pre}\, e) = e$ dans les scopes actifs et $\mathrm{next}(\mathrm{pre}\, e) = \mathrm{pre}\, e$ dans les scopes inactifs. (pour retrouver quels scopes sont actifs après résolution du domaine abstrait, en particulier dans le cas des blocs activate, on peut envisager d'introduire une variable pour chaque scope avec équation $\mathrm{act} = 1$ dans le cas actif et $\mathrm{act} = 0$ dans le cas inactif. on fait ensuite le meet avec la contrainte $\mathrm{act} = 1$ pour s'intéresser uniquement au cas où un scope est actif - par exemple dans le cas des automates, pour savoir si l'on fait les transitions ou pas. une variable numérique, dans le domaine des polyhèdres par exemple, est bien capable de faire la distinction entre $\{0\}$, $\{1\}$ et $\{0, 1\}$. on ne peut pas en faire de même pour les variables d'état des automates qui peuvent avoir plus de choix, sinon il suffirait de traduire tout le programme en une grosse condition que l'on itère)

  Ainsi le problème d'ordonnancement disparaît.

- Pour garder plus d'informations sur les liens temporels entre les différentes valeurs prises par une variable, on peut enregistrer des informations de type *last* à chaque fois. Mais attention aux problèmes d'horloge. Le problème étant, si à chaque tick on oublie tout sauf la valeur de $\mathrm{pre}\, x$, alors la relation linéaire entre x et $\mathrm{pre}\, x$ dans un programme comme $x = \mathrm{pre}\, x + 1$ est perdue.

- Plutôt que d'introduire une variable *init* et une variable *act* pour chaque scope, introduire une variable *time*, telle que :

$$\begin{array}{rcl} \mathrm{init} & \Leftrightarrow & \mathrm{time} = 0 \\ \mathrm{act} & \Leftrightarrow & \mathrm{next}\,\mathrm{time} = \mathrm{time} + 1 \\ \neg\mathrm{act} & \Leftrightarrow & \mathrm{next}\,\mathrm{time} = \mathrm{time} \end{array}$$

  Cette variable entière peut servir de référence pour découvrir des égalités linéaires selon le temps.

Faisons un exemple : le classique programme *updown*.

Pour $q = \text{UP}$:

```
((time = 0 and last_x = 0 and x = last_x + 1) or
(time > 0 and last_x = pre_x and x = last_x + 1))
and next_pre_x = x and next_time = time + 1
```

Pour $q = \text{DOWN}$:

```
((time = 0 and last_x = 0 and x = last_x - 1) or
(time > 0 and last_x = pre_x and x = last_x - 1))
and next_pre_x = x and next_time = time + 1
```

Pour la transition $\text{UP} \rightarrow \text{DOWN}$, on vérifie la condition $x \geqslant \text{bound}$. Pour le stay UP, on vérifie la condition $x < \text{bound}$. Pareil dans l'autre sens.

L'état initial est en gros :

`time = 0`

Du coup après application de la contrainte on a :

`time = 0 and last_x = 0 and x = last_x + 1 and next_pre_x = x and next_time = 1`

La contrainte pour passer à DOWN est fausse, donc il n'y a pas de division.

Après passage du cycle, on a :

`time = 1 and last_x = 1 and pre_x = 1`

puis :

`time = 2 and last_x = 2 and pre_x = 2`

Après widening, on pourait avoir un truc du genre :

`last_x = time and pre_x = time and time >= 1` (on est uniquement dans STAY)

Puis on restreint en appliquant la négation des conditions de sortie :

`last_x = time and pre_x = time and pre_x <= 6 and time >= 1`

On a atteint un point fixe petit (donc intéressant), on peut donc s'intéresser aux conditions sortantes. Après application de la fonction de cycle, on aurait quelque chose comme :

`time >= 1 and last_x = pre_x and x = last_x + 1 and pre_x <= 6`

En appliquant la condition de sortie, on a immédiatement :

`x = 7`

Ce qui est trop cool. On enrichit donc le NEW de DOWN par un état du style `time = last_x = pre_x = 7`. Il y a un certain nombre d'aller-retours qui vont être fait pour perdre la contrainte entre `x` et `time` puisque celle-ci n'est pas valable sur la durée. Dans tous les cas on doit pouvoir prouver sans trop de difficulté que x est borné.

Deuxième exemple : le programme *updown* buggué, où les `pre` sont à l'intérieur des automates (la suite produite diverge : 7 -7 8 -8 9 -9 etc.)

Pour $q = \text{UP}$:

```
((UP.time = 0 and x = 1) or
(UP.time >= 1 and x = UP.pre_x + 1))
```

```
and next_UP.pre_x = x and next_UP.time = UP.time + 1
and next_DOWN.pre_x = DOWN.pre_x and next_DOWN.time = DOWN.time
```

Pour $q = \text{DOWN}$:

```
((DOWN.time = 0 and x = -1) or
(DOWN.time >= 1 and x = DOWN.pre_x - 1))
and next_UP.pre_x = UP.pre_x and next_UP.time = UP.time
and next_DOWN.pre_x = x and next_DOWN.time = DOWN.time + 1
```

Il faudrait vérifier que l'analyse fait bien ce qu'on attend.

Maintenant, il faut coder pour voir si ça marche.

Remarque de Jean-Louis : pourquoi gérer les automates à part ? Ne peut-on pas travailler sur un domaine abstrait qui gère les énumérations (d'une manière ou d'une autre) et encoder tout le programme comme une grosse formule logique comme ci-dessus (on le fait bien pour les blocs activate).

En fait la grosse formule logique en question done une espèce de description du processus qui calcule un état suivant en fonction d'un état précédent. Elle peut donc peut-être représenter aussi les automates, les transitions, ...

Autre problème : arriver à écrire cette formule de manière factorisée pour limiter les explosions dues aux if et autres...

2014-06-17. Aujourd'hui, essentiellement du code.

On met les formules logiques sous une forme "normalisée" : liste de contrainte puis arbre de disjonctions. Comme ça on peut appliquer toutes les contraintes d'un coup pour restreindre au maximum le domaine et ce de manière cohérente. On traite ensuite les différentes disjoinctions une par une.

Code code code code code code....

2014-06-18. La partie interprétation abstraite *sans les automates* fonctionne. Remarques :

- On n'utilise pas les propriétés proposées par les `guarantee` pour restreindre notre domaine. En pratique, il faudrait rajouter un mécanisme permettant de le faire (même genre que pour les `asserts` lorsqu'on analyse mini-C : on fait des tours de boucle en supposant l'assert vrai, on cherche un point fixe. on fait ensuite un dernier tour de boucle sans supposer l'assert, et on regarde si celui-ci est violé. si oui, on râle. sinon on est content.)

  Il faudrait modifier les fonctions de transformation de programme de manière à sortir :

  - La fonction de boucle sans les guarantee
  - La fonction de boucle avec les guarantee
  - La fonction qui vérifie que les guarantee sont tous vérifiés (filtrer avec sa négation doit renvoyer $\bot$)

- Le domaine abstrait des intervalles est en fait assez puissant.

- Si on a une transition $S_n \xrightarrow{\forall i} S'_n, S_{n+1}$, alors $S'_n = f(S_n)$ et $S_{n+1} = \text{passCycle}(f(S_n)) = \text{cycle}(S_n)$. Ie si à partir d'un ensemble d'états de départ représenté par S_n, et pour n'importe quels entrées (conformes aux assume) on peut produire l'ensemble de sorties o_n, alors S'_n est l'ensemble de tous les états du système accessible concernant toutes les variables (entrées, internes, sorties), et S_{n+1} et l'ensemble d'états de départ pour le cycle suivant. (f est la fonction de boucle)

- Si S_0 représente l'état initial du système, sémantique implémentée :

$$\begin{aligned} \text{new} &= S_0 \\ \text{stay}_0 &= \text{cycle}(\text{new}) \\ \text{stay} &= \text{fix}\, \lambda I.(I \nabla g(I)) \\ g(I) &= \text{stay}_0 \sqcup \text{cycle}(I) \end{aligned}$$

- Après une après-midi de travail, ça y est les *guarantee* sont pris en compte. Pas mal de trucs sont prouvés, c'est bien.

- Il faudrait travailler sur un domaine abstrait qui sépare les variables de contrôles (booléennes et énums pour les états des automates) des variables numériques. Dans ce cas on pourrait essayer d'avoir un domaine relationnel-numérique (Apron) pour chaque valuation des variables de contrôle (la restriction aux variables de contrôle n'est pas approximée). On pourrait ensuite imaginer que l'on a une heuristique qui dit de "faire l'impasse" sur une certaine variable de contrôle - ou alors ponctuellement, oublier une variable de contrôle et faire la fusion de ce qui va ensemble pour eux.

  Niveau de granularité : peut-on facilement gérer des propriétés du type $x \neq E$ où x est une variable énumérée et E un élement de l'énumération ?

  En faisant ainsi une violente disjonction de cas on pourrait peut-être prouver la propriété sur les cartes de la suite de Gilbreath (le code a été rajouté aux exemples-tests du projet).

  Cela exige de faire une analyse de typage pour savoir quels sont les types des valeurs (numérique, contrôle, ou éventuellement liste de tout ça pour le cas où on a des valeurs multiples renvoyées par des noeuds et que l'on veut les garder dans des pre... pas supporté pour l'instant).

- Dans un premier temps on pourrait tenter un codage one-hot des automates (une variable booléenne par état).

  La disjonction de cas ne sera pas bonne, et de toute façon c'est mauvais de donner trop de variables à gérer à Apron (les équations sont horribles et les temps de calcul trop longs).

- Problème technique : à quel niveau les booléens seront transformés en énumérations ? On a souvent besoin d'utiliser des booléens en tant que formules logiques. Pour cela, faire du typage (par exemple dans un premier temps considérer que toutes les valeurs énumérées sont du type enum, et que le type bool est un sous-type du type enum).

- Du coup, introduire une variable énumérée pour l'init ? Avec une équation du type $(\text{init} \wedge \text{time} = 0) \vee (\neg\text{init} \wedge \text{time} \geqslant 1)$ pour chaque scope. Magie : il suffira de rajouter cette équation au moment où on fait la transformation programme vers formule, puis éventuellement de spécifier si on veut faire la disjonction ou non selon cette variable. Du coup, peut-être qu'on pourra enlever la variable *time* ? Ce sont des tests à faire.

- Résultats attendus : très bons !

- L'avantage d'utiliser un domaine abstrait qui gère les variables énuméréres c'est qu'on peut effectivement écrire *une seule formule* pour tout le programme. La disjonction de cas est gérée de manière uniforme par le domaine abstrait, et ce sans doute de manière plus simple.

2014-06-19.

- Refactoring du code avec une structure `rooted_prog` qui mémorise le root scope, la liste des variables, leurs types, ...

  Travailler là-dessus et faire une fonction de typage des expressions (à valeurs dans `typ list`, où `typ = | Int | Real | Enum of id list`).

- Modification des expressions et de la transformation pour prendre en compte deux types de variables : variables numériques et variables énumérées (discrètes)

- Implémentation du domaine machin.

2014-09-20.

- Ça marche pas terrible. En particulier, on manque de contrôle sur le widening. Mais tout est un peu buggué.

- Il faut être capable de gérer à plus haut niveau la disjonction de cas, pour pouvoir choisir quand on fait le widening (ou pas) pour tels et tels états.

- On pourait envisager de faire la disjonction NEW/STAY au niveau de chaque valuation des énumérés.

- Admettons que c'est un problème : il faut décider à un moment *fixe* de à quel moment on fait la disjonction de cas selon une variable. Ensuite, une fois qu'on a pris cette décision, il ne faut considérer plus que des états où cette variable est spécifiée. Ie : pour tous les cas de la disjonction, il faut que les variables précisées au niveau de la partie des énumérés soient les mêmes. Ou bien, réfléchir à des conditions un peu plus évoluées avec un peu de rigueur...

- Passes de simplification sur la formule : si on a $x = y$ au toplevel, donc vrai dans tous les cas, on peut faire le remplacement de y par x à chaque occurence. Attention, s'interdire de remplacer les variables `Nx` et les variables `x` telles que `Nx` existe.

- OU ALORS : apprendre à gérer des classes d'équivalence pour des variables. Par exemple, au départ toutes les variables appartiennent à une classe qui ne contient qu'elle et dont la valeur est inconnue. Lorsqu'on a $x = y$, fusionner les classes d'équivalence de x et y. Lorsqu'on a $x = v$, informer la classe d'équivalence de x qu'elle vaut v, et éventuellement si il y a une autre classe qui vaut ça, fusionner aussi.

  Ça devient plus dur de gérer les disjonctions ; on peut choisir de les gérer à part (garder une liste de contrainte $x \neq y$, et lorsqu'on a une valeur pour x et y, vérifier que c'est compatible. Si ça l'est, oublier la contrainte $x \neq y$. Si ça ne l'est pas, l'état devient $\bot$.)

- Ceci est en train de devenir un travail sur le model-checking et les problèmes de décision binaire/n-aire.

- Il faut définir un domaine abstrait pour les valeurs énumérées seules, qui soit capable de gérer des équations un peu évoluées (égalité, inégalité : c'est ce qu'on a dans nos formules). Ensuite, faire un domaine "produit" peut être imaginé à partir du cours d'A.Miné au MPRI sur les domaines disjonctifs (cardinal power abstraction : implications entre faits dans deux domaines abstraits). Il faut écrire formellement les définitions, les règles d'union, d'intersection, ... et le widening aussi, qui est aussi évoqué dans le cours d'A.Miné (partie sur les itérations chaotiques).

- Itérations chaotiques : ça marche bien quand on connait l'espace des états. Nous, on découvre l'espace des états au fur et à mesure. Pour le widening, on a un ensemble K_∇ d'états que l'on agrandit par widening au lieu de par union simple. La terminaison est garantie si tout cycle du graphe de contrôle contient au moins un état dans K_∇. En particulier, on peut proposer le fonctionnement suivant : lorsque l'on découvre de nouveaux états, ne pas les meettre dans K_∇. Lorsque les transitions sortantes d'un état que l'on analyse nous ramène dans un état que l'on connait déjà, mettre celui-ci dans K_∇ (ou alors, celui dont on vient ? cela permet de retarder le widening un peu...)

- Lundi, idées nouvelles (on espère).

2014-06-23.

Mettons au clair :

- Notons $\mathbb{X}$ l'ensemble des variables. $\mathbb{X} = \mathbb{X}_e \cup \mathbb{X}_n$ (union des variables de type énumération et des variables de type numérique)

- 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 :

$$I \;=\; \{s \in \mathbb{M} \,|\, s(\text{init}) = \text{tt}\}$$

  (rajouter comme contrainte que $s(\text{init}) = \text{tt}$ dans tous les scopes du programme, et que $s(\text{state}) = \text{initstate}$ pour tous les états d'automates)

- Entre deux cycles, les variables qui comptent réellement dans s sont les variables init pour les scopes, les variables d'état pour les automates, et les valeurs des *pre*.

- Déroulement d'un cycle : on prend l'état s, on y met les valeurs des entrées du système. On applique ensuite la fonction $f : \mathbb{M} \to \mathbb{M}$ qui calcule les valeurs de sorties et les valeurs suivantes des variables qui seront utiles entre deux cycles. On peut à ce moment récupérer les valeurs de sorties.

  La fonction de cycle $c : \mathbb{M} \to \mathcal{P}(\mathbb{M})$ 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 grâce à un ensemble C de couples de variables (x, y) telles que x vaudra au cycle suivant ce que y vaut au cycle courant. D'où la définition :

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

- Par facilité, on note $g = c \circ f$. La valeur qui nous intéresse est :

$$S \;=\; \text{lfp}_I\,(\lambda s. I \cup g(s))$$

  où I représente l'ensemble des états initiaux. S représente ici exactement l'ensemble de tous les états accessibles par le système.

- Exemple :

```
x = 0 -> pre x + 1;
```

  L'état initial s_0 : $s_0(\text{init}) = \text{tt}$

  Après application de f : $f(s_0)(\text{init}) = \text{tt}, f(s_0)(\text{next init}) = \text{ff}, f(s_0)(x) = 0, f(s_0)(\text{next pre}\, x) = 0$

  Après application de c, on a $s_1 = c(f(s_0))$. $s_1(\text{init}) = \text{ff}, s_1(\text{pre}\, x) = 0$

  Après application de f, on a $f(s_1)(\text{init}) = \text{ff}$, $f(s_1)(x) = 1$, $f(s_1)(\text{next init}) = \text{ff}$, $f(s_1)(\text{next pre}\, x) = 1$.

  En supposant une sémantique impérative naïve, la fonction f peut être donnée par :

$$f \;=\; [\![\text{next init} := \text{false}; x = (\text{init}?0 : \text{pre}\, x + 1); \text{next pre}\, x = x]\!]$$

La fonction c est définie par l'ensemble $C = \{(\mathrm{init}, \mathrm{next\,init}), (\mathrm{pre}\,x, \mathrm{next\,pre}\,x)\}$.

L'ensemble S des états accessibles est :

$$S = \{\rho \in \mathbb{M} \,|\, \rho(\mathrm{init}) = \mathrm{tt}\} \cup \{\rho \in \mathbb{M} | \rho(\mathrm{init}) = \mathrm{ff} \wedge \rho(\mathrm{pre}\,x) \in \mathbb{N}\}$$

En appliquant la fonction f, on obtient que x évolue dans $\mathbb{N}$.

- 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 et qui calcule les sorties o_n et l'état suivant s_{n+1} :

$$s_0 \xrightarrow{i_0} o_0, s_1 \xrightarrow{i_1} o_1, s_2 \xrightarrow{i_2} o_2, s_3 \rightarrow \cdots$$

  Avec nos notations : o_n est une partie de $f(s_n + i_n)$ et que $s_{n+1} = c(f(s_n + i_n))$, où $s_n + i_n$ correspond à définir les variables définies dans s_n et celles définies dans i_n.

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

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

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

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

  Cette fonction fait correspondre à une partie de $\mathbb{M}$ sa meilleure approximation dans le domaine abstrait (par exemple dans le cas des polyhèdres, 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 $\forall x \in D^{\#}$, $x \sqsubseteq \alpha(\gamma(x))$ 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 $D^{\#}$: les intervalles et les polyhèdres convexes. Ceux-ci sont considérés acquis pour la suite ; on les note $D^{\#}_{\mathrm{int}}$ et $D^{\#}_{\mathrm{poly}}$, avec les fonctions de concrétisation γ_{int} et γ_{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 $\mathbb{E}$: $x = 0, c = \mathrm{tt}, y \geqslant 5\,x - 2$.

  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 $D^{\#}$ et pour une expression $e \in \mathbb{E}$, on suppose que l'on a une fonction sémantique $[\![e]\!] : D^{\#} \rightarrow D^{\#}$ 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 comme l'application d'un certain nombre de contraintes de $\mathbb{E}$, ainsi que de disjonction de cas.

  Pour reprendre l'exemple du compteur, la fonction abstraite $f^{\#}$ peut être représentée par :

$$f^{\#}(s) \;=\; [\![\mathrm{next\,init} = \mathrm{ff}, \mathrm{next\,pre}\,x = x]\!]([\![x = 0]\!]([\![\mathrm{init} = \mathrm{tt}]\!](s)) \sqcup [\![x = \mathrm{pre}\,x + 1]\!]([\![\mathrm{init} = \mathrm{ff}]\!](s)))$$

  On remarque que l'aspect impératif disparait complètement, on n'a plus qu'un ensemble d'équations et de disjonctions.

- La fonction de cycle c correspond à un ensemble de paires de variables, signifiant : au cycle suivant, cette variable vaudra la valeur de cette autre variable au cycle courant. Notons C l'ensemble de telles paires de variables. Cette fonction peut être représentée dans l'abstrait par l'opérateur $c^{\#}$ défini par :

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

  Cela correspond à oublier un certain nombre de variables qui ne nous intéressent plus, et à renommer celles que l'on garde en le nom qu'elles auront dans le cycle suivant.

- Par facilité, on note $g^{\#} = c^{\#} \circ f^{\#}$. Étant donné qu'un programme est essentiellement une grosse boucle, la valeur qui nous intéresse est :

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

  Où s_0 est l'état initial du système et est défini par $I^{\#} = [\![i]\!](\top)$, où i est une équation du type init = tt, rajoutant éventuellement d'autres contraintes (états initiaux d'automates, scopes, ...). Par la suite, on appellera toujours i l'équation d'init du système.

- 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 $+, -, \times, \div, =, \geqslant, \neq$. 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 $D^{\#}_{\mathrm{int}}$ et $D^{\#}_{\mathrm{poly}}$ ne sont pas à même de représenter correctement les contraintes de $\mathbb{E}_e$. Généralement, on définit :

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

  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) :

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

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

- De plus, 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 D et surtout la fonction d'application d'une condition $[\![e]\!]$.

Un premier treillis : le treillis des disjonctions par valuation de variables fixées.

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 ammené 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 D_n capable de gérer aussi bien les contraintes numériques que les contraintes sur les variables énumérées. Le domaine D représente une abstraction de $\mathbb{M}_n = (\mathbb{X}_n \cup \mathbb{X}_e) \rightarrow \mathbb{V}$. On note $\bot_n$ 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 : $\mathbb{M}_d \cup \{\bot, \top\}$ forme un treillis plat : $\forall x, y \in \mathbb{M}_d, x \sqsubseteq y \Leftrightarrow x = y$, $\forall x \in \mathbb{M}_d, \bot \sqsubset x \sqsubset \top$ (l'utilité de cette structure n'apparait 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 $\{s \in \mathbb{M} | \forall x \in \mathbb{X}_d, s(x) = \mathbb{M}_d(x)\}$. De même on identifie $e \in \mathbb{M}_n$ à $\{s \in \mathbb{M} | \forall x \in \mathbb{X}_n \cup \mathbb{X}_e, s(x) = \mathbb{M}_n(x)\}$.

On construit maintenant le domaine abstrait disjonctif comme suit :

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

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

$$\begin{aligned} \bot_d &= \lambda d. \bot_n \\ \top_d &= \lambda d. \top_n \end{aligned}$$

On vérifie bien que $\gamma_d(\bot_d) = \varnothing$ 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 nombres d'opérateurs de restriction :

- $\forall x, y \in \mathbb{X}_d, \forall s \in D_d$,

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

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

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

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

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

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 accessible par le système est $S^\# = \mathrm{lfp}_{I^\#}(\lambda s.I^\# \sqcup g^\#(s))$, où $I^\# = [\![i]\!]_d(\top_d)$.

Définitions : pour $d_0 \in \mathbb{M}_d$, notons $r_{d_0} : D_d \to D_d$ 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 :

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

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 connait.

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 :

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

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

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

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

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

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.

Questions sur le mode d'itération :

- Peut-on se passer de K_∇ ? Il faudrait en avoir une intuition nette, puis une preuve.
- Est-on sûr que si l'on converge, on aura bien couvert tout les états possibles du système ? En principe oui, mais il faut une preuve.

Questions plus générales :

- Comment choisir l'ensemble $\mathbb{X}_d$ des variables servant à faire des disjonctions ? Première approximation, prendre :
  - Les variables d'init pour tous les scopes
  - Les variables d'état pour tous les automates
  - (?) Les variables représentant une *guarantee* que l'on cherche à prouver
- Peut-on dynamiquement modifier $\mathbb{X}_d$, ie l'ensemble des variables qui sert à faire des disjonctions ? Il semble qu'on peut facilement supprimer une variable de $\mathbb{X}_d$, il suffit de faire l'union abstraite pour chaque combinaison des autres variables. Pour en rajouter une, par contre, cela exige probablement de tout recalculer afin de refaire la disjonction.
- Peut-on faire la disjonction selon une certaine variable dans certains cas et pas dans d'autres ? Par exemple, si init = tt on ne fait aucune autre disjonction mais si init = ff alors on fait la disjonction selon les états d'un automate (juste une idée). Cela fait penser à utiliser un truc plus évolué que les valuations de $\mathbb{M}_d$, plutôt quelque chose comme un arbre de décision.
- Dans tous les cas, il semble pratique de faire en sorte que $\mathbb{X}_d$ soit inclus dans $\mathbb{X}_e$, c'est-à-dire que le domaine sous-jacent soit également informé des contraintes et informations sur les variables faisant les disjonctions. Si ce domaine est suffisament puissant, il peut arriver que l'on soit capable de retrouver un certain nombre de caractéristiques des disjonctions à certains moments (ceci est très vague).

Maintenant que c'est posé sur "papier", ça serait sympa de coder ça pour de vrai.

Deuxième treillis : le treillis des disjonctions dynamiques par arbre de disjonctions.

Ceci n'est qu'une idée : au lieu d'utiliser des valuations $\mathbb{M}_d$ pour un ensemble prédéfini de variables, utiliser un arbre de décision pour représenter les différentes possibilitées. Cela donne plus de flexibilité, mais laisse néanmoins ouverte la question : quand choisit-on de faire une disjonction ?

Lorsque l'on fait une disjonction, on peut imaginer que l'on met à $\perp$ les deux branches de la disjonction, pour être sûr d'en profiter au maximum. Il faut ensuite les repeupler grâce à tous leurs prédécesseurs potentiels : état initial, autres états encore mémorisés. Si il y a beaucoup d'états, on peut envisager de faire une itération à reculon pour déterminer les prédécesseurs potentiels (itération à reculon : sans doute assez facile à implémenter, c'est exactement la même chose mais avec une fonction de cycle inversée).

Ce domaine reste à définir formellement.

2014-06-24.

- Les valeurs mappées dans D_d sont des valeurs de $D_{\text{enum}} \times D_{\text{num}}$, et la partie enum répercute les informations de l'identifiant mappant à la valeurs dans le domaine disjonctif.

- En pratique on n'applique pas une formule sur un élément de D_d mais sur une valeur de $D_{\text{enum}} \times D_{\text{num}}$, ce qui donne une liste de valeurs dans cet ensemble (tout plein de disjonctions sont faites, et on a ensuite une étape d'intégration dans la valeur de D_d où des fusions peuvent être réalisées). $\rightarrow$ NON, ça ne peut pas trop marcher.

- Il y a deux ensembles de valeurs pour lesquels faire les disjonctions, en fait :

  - Un ensemble selon lequel on fait des disjonction par rapport aux valeurs qui comptent lors des transitions ; ceci permet de différencier les états sur lesquels on fait des itérations chaotiques.

  - Un ensemble selon lequel faire les disjonctions pendant qu'on fait le calcul, c'est-à-dire qu'on peut maintenant faire des disjonctions selon n'importe quelles variables.

Le domaine disjonctif (décrit en détails ci-dessus) est implémenté mais ne fait les disjonctions que selon un ensemble de variables, quel que soit le moment. On arrive néanmoins à checker des programmes relativement complexes (Gilbreath par exemple). Le U-turn par contre prend un temps inconnu mais très très long.

- Il faut *absolument* une structure disjonctive plus puissante, qui soit capable de comprendre que lorsque init = true, il est inutile de faire des disjonctions selon les valeurs des pre puisque ceux-ci par définition peuvent prendre n'importe quelle valeur dans l'état initial. Plus précisément, il faudrait un arbre de disjonction ; et quand on rajoute une valeur à notre arbre, alors on prend les valeurs dans l'ordre, et si la valeur est définie dans l'environnement alors on introduit un noeud de disjonction mais si elle n'est pas définie on ne fais pas de disjonction. Évidemment, une fois qu'on a choisi de ne pas faire une disjonction, c'est difficile de la rajouter, mais c'est faisable : il faut dupliquer tout le sous-arbre pour chaque branche de la disjonction.

- Pour l'ordre des variables dans l'arbre de disjonctions, au début on peut le prendre arbitraire, mais ça serait sympa de grouper les variables qui apparaissent dans des égalités dans les formules d'init ou de transition. Pour cela, utiliser un algorithme type *scope constriction algorithm* :

  Soient $x_1, ..., x_n$ des variables, et $(a_1, b_1), ..., (a_p, b_p)$ un ensemble de couples de variables reliées (ie apparaissant ensemble dans une équation). On veut choisir une permutation σ de $[\![1, ..., n]\!]$ qui minimise la somme :

  $$w(\sigma) \;=\; \sum_{i=1}^{p} 2\,|\sigma(a_i) - \sigma(b_i)|$$

(vérifier que les couples (a_i, b_i) sont deux à deux disjoints)

Pour cela, on peut utiliser un algorithme approché qui part de $\sigma = \text{id}$ et qui boucle : prendre une variable et la déplacer à la position qui minimise la somme (on fait des optimisations locales, en espérant que globalement ça soit efficace - en général ça peut l'être). On s'arrête quand on n'arrive plus à optimiser le truc en ne bougeant qu'une seule variable.

L'intérêt de minimiser $w(\sigma)$ apparait très clairement quand on va faire des BDD...

- Supposons que le domaine des énumérés soit implémenté par un BDD, il faut une façon de répercuter les relations enregistrées dans ce BDD dans le graphe de décision utilisé pour faire les disjonctions de cas. Par exemple, définir une structure de BDD qui soit indépendante du domaine abstrait, et pour chaque domaine abstrait implémenter une fonction qui l'exprime sous forme de BDD (pour les constantes, c'est aussi assez simple, on a un noeud de décision uniquement pour les variables définies).

  Évidemment, il faut utiliser le même ordre dans l'arbre des disjonctions d'états et dans les BDD représentant la partie énumérée.

  Je parle trop de BDD, il va finir par falloir les implémenter. Si on se fixe un ordre des variables au départ, ça ne semble pas être particulièrement difficile (beaucoup de calculs avec mémoisation).

- Si on implémente des BDD, a-t-on encore besoin d'enregistrer des informations sur les énumérés dans le domaine sous-jacent au domaine disjonctif ? Probablement qu'on peut s'en passer, on utilisera alors le seul domaine numérique en sous-jacent au domaine disjonctif. C'est la prochaine chose à implémenter.

Définition du domaine disjonctif par BDD.

On en revient à nos deux ensembles de variables : $\mathbb{X}_n$ les variables numériques et $\mathbb{X}_e$ les variables de type énuméré. On définit un ordre sur les variables de $\mathbb{X}_e$: $\mathbb{X}_e = \{x_1, x_2, ..., x_n\}$ (cf SCA ci-dessus pour le choix de cet ordre).

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

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

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 $\mathbb{X}_e$). Cette propriété implique automatiquement que l'arbre est fini, ce qui de toute façon était assez évident.

La concrétisation est définie de manière assez évidente :

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

Pour définir l'union, l'intersection, ... c'est assez simple.

Pour l'application d'une condition sur les énumérées, c'est simple : on ne garde que les branches qui correspondent (en fait, on fait le meet avec un arbre qui représente la contrainte, où le meet est un BDD représentant la contrainte (true = $\top$, false = $\bot$), c'est assez simple). Pour l'application d'une contrainte sur les numériques, on le fait sur toutes les feuilles de l'arbre (pour plus tard : penser à partager les feuilles si possible).

L'oubli d'une variable est simple à implémenter : on supprime les noeuds de décision qui la concernent en faisant le $\sqcup$ de toutes les branches.

Pour le passage de cycle, c'est complexe ! On risque d'avoir à réordonner les variables de l'arbre... Ou alors, peut-être qu'on peut oublier la nouvelle variable puis rajouter la contrainte nouvelle = ancienne. Ça peut sans doute marcher relativement bien, il faut tester. Bien sûr, ne pas oublier de rajouter les couples $(\text{nouveau}, \text{ancien})$ à l'ensemble des couples de variables devant être proches !

Pour les itérations chaotiques, il faut trouver un moyen d'identifier les branches de l'arbre et de se dire que chaque état correspond à une branche. Faire une étape d'itération, c'est prendre une branche récemment apparue ou récemment enrichie, puis faire comme précédemment. L'identification des branches est la seule chose qui pose problème ; et aussi éventuellement les problèmes de partage. Dans un premier temps, peut-être se contenter d'itérations non-chaotiques ; dans ce cas on pourrait enregistrer un compteur du nombre de joins qui ont été fait à chaque feuille, et utiliser ce compteur dans une opération de widening qui fait join/widening selon la valeur de ce compteur.

2014-06-25.

Avant de coder, il faut résoudre mieux que ça le problème des itérations sur un arbre de décision. Ie. savoir ce qu'on veut faire.

Je suis très *** : le même *pre* dans l'AST peut être instancié plusieurs fois si le noeud dans lequel il apparait est instancié plusieurs fois. La dénomination des *pre* doit donc se faire selon le chemin du noeud aussi ! Correction de ça immédiatement...

Début de code des EDD ; tentative d'appliquer au U-turn : c'est clairement pas aussi puissant que prévu ; il faut voir pour faire des optimisations dans les fonctions de join et de meet, pour éviter les temps exponentiels le plus possible (et comment choisir un ordre pour les variables, toujours !)

Pour simplifier la formule :

- Parmi chaque ensemble de variable tout le temps égales, n'en garder qu'une (sauf s'il y a des variables de type next, dans ce cas on évite) → c'est l'optim la plus grosse, certainement. En particulier, dès que l'on instancie un noeud dont les arguments sont directement des variables ou dont les valeurs de retour sont directement des variables, chacune de ces variables est dupliquée...

- Voir que un noeud a la même variable d'init que le scope qui l'a instancié → dans un programme sans activate et sans automates, il n'y a qu'une seule variable init, pourtant on en a actuellement une pour chaque node. Idée : quand on transforme, garder en mémoire le scope qui définit l'horloge du scope actuel (ce n'est pas le scope actuel dans le cas d'une instance de noeud).

Voir s'il y a peut-être des choses plus intéressantes sur lesquelles travailler... mais on peut se laisser encore jusqu'à vendredi pour travailler sur les EDD. Idées pour la suite : les tableaux ? Par exemple.

2014-06-26.

- Il faudrait réflechir à quelques nouveaux exemples d'application. Dixit B.Pagano : il faut pouvoir prouver qu'il n'y a pas de division par zéro, qu'il n'y a pas d'overflow, qu'un carré est positif, ... parce que ça pète.

- Exemples avec des automates :

  - ping-pong (en SCADE/KCG on est obligé d'introduire des délais pour des questions de causalité, ici ça n'est pas le cas)

  - automate qui cycle entre quelques états mais qui a la possibilité de s'en échapper définitivement (du coup il est intéressant de diviser l'ensemble des états en deux parties par exemple), ou le cas inverse, a plusieurs états d'initialisation avant de cycler parmi plusieurs états de régime permannent.

- Se pencher sur le support pour les nombres réels. Premièrement, faire le minimum de typage pour qu'Apron sache quand des expressions arithmétiques se font sur des réels et non des entiers. Secondément, adapter le domaine des intervalles pour prendre en compte des bornes rationnelles (il parait que c'est pas évident évident). Troisièmement, réfléchir à une prise en compte de la sémantique machine des nombres à virgule flottante (un des trucs les plus ennuyeux qui soient).

- Problème des transitions qui reset : comment les traduire ? Peut-on en profiter pour dire "tout ça, pas la peine de mémoriser" ? Du coup pour faire ça, il y a un peut d'analyse qui se fait directement sur le code SCADE.

- Nos résultats obtenus sur du code SCADE sont-ils toujours vrais sur le programme compilé en C ? (on voit venir la preuve de 500 pages !)

Idée d'autre optimisation possible pour améliorer l'analyse : factoriser les *pre* sur la même expression et selon la même horloge.

Nouveaux domaines abstraits pour les énumérés :

- Représenter un ensemble de valeurs acceptées pour chaque variables, ie. treillis des parties au lieu du treillis des constantes.

- Représenter ça plus un ensemble de valeurs acceptées pour des couples de variables (cela donne une traduction de toutes les contraintes). Ce domaine fonctionne mal, en particulier il ne voit pas toujours si les contraintes qu'il enregistre sont incohérentes et impliquent $\bot$.

Pas d'avancement sur les EDD. L'implémentation d'hier était trop lente car on s'appuie sur la comparaison de Caml. En vrai, il faut numéroter chaque noeud du graphe et utiliser ces numéros comme points d'identification (il n'y a pas moyen d'utiliser l'adresse physique en Caml). Modification de la structure de l'EDD pour faire ça, mais le code n'a pas été adapté (donc ça ne compile pas).

2014-06-27.

- Amélioration de l'implémentation des EDD : tous les noeuds du graphe sont numérotés, les comparaisons se font donc avec ces numéros comme clefs.

  Ça reste assez lents dès que l'on dépasse la centaine de noeuds... optimisations à trouver. Par exemple, vu qu'on représente $\bot$ à part, peut-on en faire de même pour $\top$? Cela accélèrerai bien les $\sqcup$, sans doute. Pour les $\sqcap$ il faut trouver autre chose (d'autant plus que c'est l'opération la plus utilisée des deux). Par exemple, on pourrait envisager de numéroter les noeuds et les feuilles de manière unique sur tous les EDD considérés, la copie devient donc triviale (pour penser à bien aller copier les feuilles : faire une fonction mémoisée qui parcours un sous-graphe et copie les feuilles). Du coup les $\sqcap$ avec $\top$ pourraient être bien améliorés, pareil pour les $\sqcup$ avec $\bot$.

- Remarque de JLCO : on peut gérer la mémoire de manière plus uniforme en introduisant *last* comme primitive et en exprimant tout à partir de là. Toutes les variables que l'on appelle dans un noeud *last* correspondent aux variables qu'on a besoin de perpétuer d'un cycle à l'autre, et qui forment notre mémoire. Du coup, trois étapes dans le cycle :

  - Copier les lasts dans un nouvel environnement, en les préfixant par *last_*

  - Calculer notre fonction

  - Oublier tout ce qui ne sera pas utile pour le cycle suivant, ie n'est jamais appellé dans un *last*.

Le résultat de cette troisième étape correspond à ce qu'on peut vouloir garder dans notre calcul de point fixe.

L'implémentation de la primitive *last 'x* pour un identifeur x reste néanmoins problématique (il faut perpétuer la mémoire, mais seulement si x est utilisée dans un *last*...)

2014-06-30.

- Amélioration (un peu) de l'implémentation des EDD, avec le $\top$ qui est aussi mis à part. Problème technique pour partager globalement les numéros des parties numériques des domaines : on ne sait jamais trop quand copier ou ne pas copier des machins dans la hashtbl, ...

- Le problème du widening n'est toujours pas résolu de manière satisfaisante... c'est l'objectif de la journée.

- On anote les feuilles par une étoile lorsqu'elles sont nouvelles / ont grandi depuis la dernière fois. L'identification d'une feuille se fait par la combinaison exacte qui y mène. Dès qu'on a une combinaison un peu différente lors d'une union-widen, on remet des étoiles partout.

- Structure des itérations :

  - Chercher une feuille étoilée

  - S'il n'y en a pas :

    - Itérer sur tout l'arbre (du coup on s'intéresse aussi aux cas $\top$)

    - Faire l'union avec init

    - Si on a un truc plus gros, marquer d'une étoile les cas nouveaux et refaire une itération

    - Sinon, stop.

  - S'il y en a une :

    - La prendre, lui enlever son étoile.

    - Considérer uniquement ce cas là (prendre l'ensemble des chemins vers la feuillle) ; considérer que l'on veut rester dans ce cas là ; trouver un point fixe par union + widening + itérations décroissantes

    - Faire une itération sans se restreindre

    - Rajouter ça à notre accumulateur, marquer d'une étoile les cas nouveaux

    - Refaire une itération

- Ça marche bien, mais un contre-exemple montre qu'on a aussi besoin de faire du widening parfois au niveau global. C'est l'exemple du *half*, où deux cas disjoints s'enrichissent mutuellement à l'infini (si on fait le point fixe sur un état, on ne progresse pas beaucoup ; sur les deux, il faut widen à un moment pour que ça termine). C'est pour demain.

2014-07-01. Premier juillet !

- Widening global sur les EDD $\rightarrow$ OK, du coup *half* termine

- Pour l'ordre des variables : [init ; state ; le reste] donne un truc "pas mal"... à étudier un jour : réordonnement dynamique des variables dans un EDD pour réduire sa taille. En tout cas, meilleures heuristiques pour l'ordre : on pourrait s'appuyer plus sur le texte du programme et sa sémantique.

- Traîter les `real` avec des rationnels en précision arbitraire (uniquement domaine Apron, pour les intervalles il faudrait faire trop de modifications - à voir)

- Les real : ok c'est donné à manger à Apron mais ça donne pas de très bon résultats. Que faire ?

- Pour la suite des opérations, discuter avec JLCO.

2014-07-02.

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).

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

Les EDD.

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

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

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

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

et 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 $[\![e]\!]$ par :

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

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

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

et symmétriquement lorsque $j > i$.

On peut ensuite poser :

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

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 :

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

Opérateur de widening.

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

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

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 :

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

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 selont 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 :

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

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

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

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

(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 :

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

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 :

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

  (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 disparaitre.

- 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 $s_{n+1} = s_n$.

2014-07-03.

Rien de bien intéressant. Pour implémenter les transitions *restart* :

- Pour chaque état introduire une variable qui dit si cet état doit être reset pour le prochain cycle
- Bien s'assurer de donner la bonne valeur à cette variable, tout le temps !
- Puis, dans la fonction de génération de machins, accumuler les valeurs qui peuvent provoquer un reset et les utiliser au bon endroit, lors de la propagation des init et des états.

(c'est compliqué ! on a déjà trop d'informations qui se balladent dans notre transformation...)

Petits essais sur le test DiffSystem issu des slides de Jean Bourderionnet ; impossible de prouver quoi que ce soit dessus juste comme ça (on ne trouve pas non plus d'informations intéressantes sur le domaine de variation de la variable de sortie...). Pour le NestedControl : si la condition C implique $\text{In} \neq 0$, on arrive à prouver qu'il n'y a pas de modulo par zéro, mais il faut introduire explicitement des variables qui font la disjonction de cas $\text{In} > 0$ et $\text{In} < 0$. Pour le dernier exemple, Overflows, il faut des automates à restart, on s'en occupera donc demain. Il faut aussi introduire la primitive `restart N every C` ; une fois que l'on aura implémenté le restart sur les automates ça doit être possible à faire (le restart...every introduit en fait un nouveau scope avec son time/init propre). Et il faut aussi le `fby(a; n; b)`, ça commence à devenir un peu compliqué.

Autre exemple à tester : ABRO, déjà pour tester que le restart est bien pris en charge.

2014-07-04.

Dans la branche git `e-last`, modification du code pour exprimer la mémoire à l'aide d'une primitive *last*, et non comme des valeurs suivantes (*next*) qu'il faut préparer à chaque cycle.

Les conditions de reset rstCond contiennent au moins une condition mustReset pour le noeud racine, cette condition étant donnée vraie pour la mémoire de l'état initial et fausse tout le reste du temps. → en théorie c'est beau, mais en pratique ça fait grossir la formule de manière assez dérésonnable. On extrait donc deux formules, une qui suppose que init global est vrai et une autre qui suppose qu'il est faux.

Du coup, les transformations sont du style :

$$
\begin{aligned}
\mathrm{init}_{\mathrm{scope}} &\rightsquigarrow \left(\bigvee_{c \in \mathrm{rstCond}_{\mathrm{scope}}} c \right) ?\, \mathrm{tt} : (\mathrm{act}_{\mathrm{scope}} \,?\, \mathrm{ff} : \mathrm{last}\ \mathrm{init}_{\mathrm{scope}}) \\
\mathrm{pre}\, e &\rightsquigarrow \mathrm{last}\, m; m = (\mathrm{act} \,?\, e : \mathrm{last}\, m) \\
e \to f &\rightsquigarrow (\mathrm{init} \,?\, e : f) \\
\mathrm{unless\ if}\ c\ \mathrm{then\ restart}\ S &\rightsquigarrow (c \,?\, \mathrm{nextSt} = S; \mathrm{mustResetS} = \mathrm{tt} : \ldots) \\
& (\mathrm{last\ mustResetS} \in \mathrm{rstCond}_S)
\end{aligned}
$$

Problème d'approximation : si une hypothèse est du type $x \neq c$ (c une constante), alors celle-ci peut difficilement être prise en compte automatiquement ! Il faut introduire des variables booléennes $b_1 = (x < c)$ et $b_2 = (x > c)$, et ensuite faire l'hypothèse $b_1 \vee b_2$. C'est gênant. (cf pour le code de l'exemple NestedControl)

TODO : adapter le premier domaine abstrait (disjonctions bourrin) pour le fonctionnement basé sur *last* et non sur *next*. Ok, c'est fait, on a aussi la possibilité (pas très paramétrable) d'utiliser deux ensembles de variables pour faire les disjonctions : un ensemble pour les disjonctions dans l'accumulateur (essentiellement, ça ne concerne que les variables de type last) et un ensemble pour les disjonctions au moment où l'on fait le calcul des infos pour un état. En pratique, ça marche pas mal, mais le prouveur montre assez vites ses limites (par exemple sur updown2, ça marche, mais c'est long !). La version de `master` est en l'occurence bien plus rapide sur updown2 lorsque l'on donne `--disj all`, mais avec la version de `e-next` on peut avoir de bons temps d'exécution en choisisant bien les variables selon lesquelles faire des disjonctions. Pour `updown2`, le bon choix est `--disj last+/aut2.next_state,/aut4.next_state`. Pour `counters.scade` (exemple de François), le bon choix est `--disj last+/a,/b,/z,/b1,/b2` (b_1 et b_2 sont optionnelles et ne servent qu'à prouver la propriété `bb`). Dans tous les cas, c'est un peu problématique.

Merge `e-last` dans `master` ; la branche `e-next` garde l'ancienne version qui utilise des next (qui n'est plus d'actualité !)

2014-07-07.

Envisager de faire une présentation à l'équipe. Pas trop technique, plus sur l'idée générale. Durée 30 à 45 mn.

Essayer de faire quelque chose de pertinent avec Astrée...

Astrée permet de grouper les variables en "packs", et sur chaque pack on applique un domaine abstrait approprié. La valeur totale représentée est l'intersection de toutes les contraintes représentées dans chacun de ces packs. En particulier, on peut avoir des packs qui utilisent les octagones ou les ellipsoides. Il existe aussi un pack booléen+numérique, où des variables booléennes servent à faire des disjonctions de cas et les variables numériques apparaissent sur les feuilles (c'est la même chose que le domaine que j'ai implémenté). Seulement, il semblerait que les variables numériques soient simplement représentées par un intervalle, alors que je permet la représentation de relations entre les variables numériques en plus.

Retour sur l'article de B.Jeannet, N.Halbwachs, P.Raymond : l'idée, c'est de partitionner dynamiquement notre espace d'états, pour essayer d'éliminer des cas. Si on splitte un état q en $q \wedge c$ et $q \wedge \neg c$, on cherche à avoir quelque chose du style $(q \wedge c) \nrightarrow q'$ alors qu'on avait $q \rightarrow q'$, par exemple. Idée de JLCo : on peut envisager de faire des itérations de raffinement *ad libitum*, puis un jour on les interromp avec un Ctrl+C. Normalement cela devrait déjà donner de bonnes informations sur le système - et le plus on attend, le meilleur elles seront ! Domaine abstrait utilisé : simple conjonction de formules logiques et de contraintes numériques. Peut-on profiter de quelque chose comme ça pour déterminer des relations linéaires entre des variables, sans avoir une propriété particulière à vérifier en vue ? Peut-on profiter d'un truc comme ça pour diviser une grosse formule logique à 2000 noeuds (dans la représentation en BDD) en trois formules à 500 noeuds, puis cinq formules à 100 noeuds, puis sept formules à 40 noeuds ? Peut-on imaginer, sur chaque état, faire une sur-approximation également dans le domaine des énumérés, par exemple en limitant violemment le nombre de noeuds dans le BDD ? Ou en faisant des "packs" ? Ou même, en ne représentant que des intervalles de valeurs pour les variables, c'est-à-dire avec un truc entièrement non-relationnel ? (garder les relations dans les définitions des états, ce qui permet que les relations booléennes, cette chose qui fait exploser les formules, soient également construites par le haut, par raffinements successifs - donc lorsqu'on fait un calcul de point fixe, les relations entre deux variables booléennes sont traitées de façon minimale ; on vérifie simplement leur accord avec la formule logique qui définit l'état étudié, et pas avec toutes les relations rencontrées jusqu'à présent)

Produit de domaines.

Si on a un domaine pour les énumérés D_e et un pour les valeurs numériques D_n, on construit leur produit simple $D_e \times D_n$ comme étant :

$$\gamma(s_e, s_n) \;=\; \gamma(s_e) \cap \gamma(s_n)$$

En particulier, les contraintes sur les énumérés sont répercutées sur la partie des énumérés, et les contraintes numériques le sont sur la partie numérique.

Domaines à puissance variable.

On dispose, en l'état, de deux domaines pour travailler sur les énumérés : le domaine des formules logiques (représentées par des EDD), qui ne fait pas d'approximation, et le domaine non-relationnel canonique (à chaque variable on associe un ensemble de valeurs possibles). Le premier est très puissant mais exponentiellement coûteux. Le second est peu puissant mais aussi très peu coûteux, et peut néanmoins stocker quelques informations d'intérêt.

Il est problématique d'utiliser le domaine exact dès le début car celui-ci a la particularité de stocker toutes les disjonctions possibles sans aucune approximation. On peut donc imaginer de l'utiliser pour partitionner dynamiquement notre système, en choisissant à chaque fois des raffinements qui vont bien et ne font pas trop exploser la taille des EDD. On peut auss imaginer que l'on fait les calculs avec les EDD de base, mais si l'on voit que ça explose alors on approxime violemment en rétrogradant sur le domaine simple.

Partitionnement.

Supposons que l'on définisse $Q \subset D_e \times D_n$.

On commence avec :

$$Q \;=\; \{[\![\mathrm{reset} = \mathrm{tt}]\!], [\![\mathrm{reset} = \mathrm{ff}]\!]\}$$

Pour chaque état $q \in Q$, on construit une sur-approximation $\mathrm{reach}(q)$ des états accessibles pour le système en considérant q vrai. Cela se fait par itérations chaotiques, comme on sait si bien le faire.

On cherche ensuite une condition c telle que $q' \rightarrow q \wedge [\![c]\!]$ mais $q' \nrightarrow q \wedge [\![\neg c]\!]$. On divise q en deux états $q \wedge [\![c]\!]$ et $q \wedge [\![\neg c]\!]$. Par exemple pour la première itération, q' sera l'état $[\![\mathrm{reset} = \mathrm{tt}]\!]$ et c sera n'importe quelle condition qui ne peut pas être vérifiée dès le premier cycle.

On refait un calcul de reach(q), et on a un système de transitions du type $q' \to q \wedge [\![c]\!] \to q \wedge [\![\neg c]\!]$, par exemple. On recommence. À chaque disjonction que l'on fait, non seulement on est certains qu'une certaine transition n'est pas possible, mais avec un peu de chance toutes les transitions vers l'état seront impossibles - en tout cas, c'est ce qu'on espère atteindre à un moment. Cela permet du coup d'éliminer des états non-accessibles.

Remarque : dans le papier il est question d'analyse arrière, parce qu'on cherche à se restreindre au co-accessible d'un état qui nous pose problème. Ici ça n'est pas le cas, on ne fait jamais que de l'analyse en avant.

Dans l'ordre des choses.

- Faire un domaine EDD simple pour coder des formules booléennes ; les feuilles ne sont plus des valeurs d'un domaine numérique.
- Faire un domaine produit simple.
- Faire une fonction qui prend un partitionnement du système et calcule des points fixes par itérations chaotiques.
- Faire une fonction qui rafine un partitionnement en divisant un état (donné) en fonction d'une condition
- Faire une fonction qui trouve un état et une condition qui peuvent mener à un meilleur partitionnement.
- Faire une boucle qui raffine, affiche des résultats, raffine, affiche des résultats, trouve un moment où ça semble inutile de raffiner plus (par exemple lorsqu'on ne trouve plus aucune condition qui mène de façon évidente à un meilleur raffinement)

Question : si on trouve un invariant numérique pour un état, est-ce que ça a un intérêt de le faire monter dans la définition de cet état ? Oui ; peut-être qu'au final, tout ce qu'on veut c'est que la définition de nos états soit un point fixe relativement raffiné de notre système, et que le contenu effectif de ce qui y est accessible ne soit pas particulièrement plus précis que les définitions elles-mêmes. C'est néanmoins intéressant de donner une définition fixe de ce qu'est un état au moment où on itère à la recherche du point fixe : cela donne quelque chose qui nous informe de quand on reste dans l'état et de quand on va dans d'autres états. Attention : il faut bien vérifier qu'on ne perd rien en route !

2014-07-08.

La formule est écrite à l'aide de conditions ternaires. Pour chaque état, écrire la formule qui s'applique quand on est dans cet état, ie en simplifiant une des conditions ternaires. Comment choisir la condition : regarder toutes les conditions disponnibles dans un état au toplevel, regarder dans l'ordre d'abord celles qui délimitent les blocs les plus gros de part et d'autre, tout en étant utiles (les deux motiés sont non $\perp$, et les transitions entrantes/sortantes sont différentes dans les deux moitiés).

PROBLÈMES SUR LA DÉFINITION DU RESET ET DE L'INIT DE VENDREDI DERNIER. Corrigé, mais au prix de l'introduction d'une variable "actif" pour chaque scope, et qui est aussi utilisée avec last donc doit être perpétuée dans la mémoire.

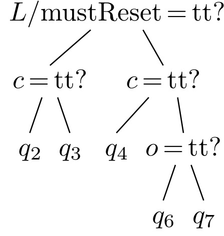
Combien de jours faudra-t-il encore attendre avant que tout ça soit assez clair pour coder ? Ou bien, la clarté viendra-t-elle en codant ? Je suis hésitant.

Demain en premier : simplification de formules (on enlève les $x \equiv y$ dans les énumérés, ça sert à rien).

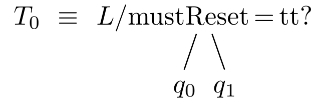
2014-07-09.

Il y a un peu un mélange entre les formules logiques et un environnement abstrait qui représente une sur-approximation de l'ensemble des environnements qui la satisfont.

Un découpage du système en états se fait avec un arbre de disjonction T :



L'arbre initial contient uniquement la disjonction `L/must_reset=tt` :



Pour un état $q_i \in T$, on pose $\mathrm{def}(q_i)$ la conjonction de toutes les conditions qui mènent à cet état dans l'arbre. On définit aussi un ensemble d'états initiaux, noté I.

On utilise un domaine abstrait D, supposé représenter les énumérés et les numériques avec une certaine précision (en principe on n'utilise pas un EDD dont les feuilles sont des valeurs numériques car la précision qu'apporte un tel domaine est sensée être remplacée par la précision de l'arbre de disjonctions, ceci dit on peut représenter la partie énumérés avec un EDD).

Pour une formule logique F, on note $\langle\!\langle F \rangle\!\rangle = [\![F]\!](\top)$.

Sur D on définit notre fonction $f^{\#}$, qui applique la formule du programme qui est vraie à tous les cycles. On définit aussi une fonction c qui correspond au passage du cycle (extraction de la mémoire), et on définit le passage inverse du cycle c^{-1} de la même façon. On pose $g^{\#} = f^{\#} \circ c$

L'algorithme itère la procédure suivante :

- On construit, pour chaque état q_i, l'abstraction $V(q_i)$ de l'ensemble des environnements possible pour le programme lorsque $\mathrm{def}(q_i)$ est vrai :

  1. Poser :

$$\begin{array}{rcl} V_0(q_i) & = & \begin{cases} f(\langle\!\langle \mathrm{def}(q_i) \rangle\!\rangle) & \text{si } q_i \in I \\ \bot & \text{sinon} \end{cases} \\ \delta_0 & = & I \\ K_{\nabla,0} & = & \varnothing \end{array}$$

  2. Itérer :

$$\begin{array}{rrcl} & s & \in & \delta_n \\ & V_{n+1}(s) & = & \mathrm{lfp}_{V_n(s)}(\lambda i. [\![\mathrm{def}(s)]\!](V_n(s) \sqcup g^{\#}(i))) \\ \forall q \in T \backslash \{s\}, & n_q & = & [\![\mathrm{def}(q)]\!](g^{\#}(V_{n+1}(s))) \\ \forall q \in T \backslash \{s\}, & V_{n+1}(q) & = & \begin{cases} V_n(q) \nabla n_q & \text{si } q \in K_{\nabla,n} \\ V_n(q) \sqcup n_q & \text{sinon} \end{cases} \\ & K_{\nabla,n+1} & = & \cdots \\ & \delta_{n+1} & = & (\delta_n \cup \{q \in T \mid V_{n+1}(q) \neq V_n(q)\}) \backslash \{s\} \end{array}$$

  On s'arrête quand $\delta_N = \varnothing$. On note $V = V_N$.

- On en déduit une relation de transition :

$$q \rightsquigarrow q' \;\; \Leftrightarrow \;\; [\![\mathrm{def}(q')]\!] \sqcap g^{\#}(V(q)) \neq \bot$$

  (cette relation de transition peut être calculée au fur et à mesure des itérations chaotiques)

  On en déduit donc éventuellement des états inaccessibles, que l'on retire de l'arbre.

- Pour chaque état $q \in T$, on suppose que l'on a une formule F_q qui est la formule du programme restreinte au cas $\mathrm{def}(q)$. On regarde parmi tous les états q s'il existerait une condition c qui apparait, ou dont la négation apparait, dans le texte de F_q et telle que :

$$\exists p \in T \;\; \Leftrightarrow \;\; \begin{cases} q \rightsquigarrow p \\ [\![\mathrm{def}(p)]\!](g^{\#}(V(q) \sqcap \langle\!\langle c \rangle\!\rangle)) = \bot \end{cases}$$

  Ou bien telle que :

$$\exists p \in T \;\; \Leftrightarrow \;\; \begin{cases} p \rightsquigarrow q \\ [\![\mathrm{def}(q)]\!](g^{\#}(V(p))) \sqcap \langle\!\langle c \rangle\!\rangle = \bot \end{cases}$$

  On peut à ce moment décider de splitter l'état q en deux états q' et q'' avec $\mathrm{def}(q') = \mathrm{def}(q) \wedge c$ et $\mathrm{def}(q'') = \mathrm{def}(q) \wedge \neg c$. (si $q \in I$, alors $q', q'' \in I'$)

- On peut dire qu'on s'arrête si on ne trouve pas d'état ou de condition qui convient.

2014-07-09. Code ; ça commence à marcher, mais les avantages sont encore peu visibles.

2014-07-10. C'est assez foireux...

Plutôt que de travailler sur des formules logiques, tenter de donner des définitions des différentes locations comme une valeur abstraite qui sera prise en conjonction avec le contenu. Les différentes locations ne sont plus totalement exclusives dans leurs définitions. Le fait de faire une interprétation abstraite sur les définitions permet de base de faire un certain nombre de simplifications.

2014-07-11. Test sur l'exemple du controleur CD : ça marche assez bien, en tout cas ça sait utiliser des conditions qu'on ne sait pas utiliser autrement puisqu'elles ne sont pas liées à une variable booléenne. À part ça c'est toujours très bof.

2014-07-15. Nouvelle phase : documentation et explications. En particulier :

- présentation pour l'équipe Core
- rapport de stage
- commenter le code, faire un README

2014-07-16. Rédaction.

2014-07-22. Pour le rapport : faire des ébauches de preuves !