Modélisation d'un langage synchrone dans un assistant de preuve

Martin Strecker, ACADIE
Séminaire vérification de Toulouse

21/04/2009

Plan

But de la sémantique

En général: Base de communication sur la signification des éléments du méta-modèle

Spécifiquement:

- Assurer la cohérence des données (typage, calcul d'horloge)
  $\rightsquigarrow$ "type soundness"
- Formaliser des transformations
  $\rightsquigarrow$ vérification de la préservation de la sémantique
- *Prospectif:* Formalisation et preuve de raffinements
- *Prospectif:* Formalisation et preuve d'un compilateur

Limitations et abstractions (1)

Limitations technologiques à l'état actuel:

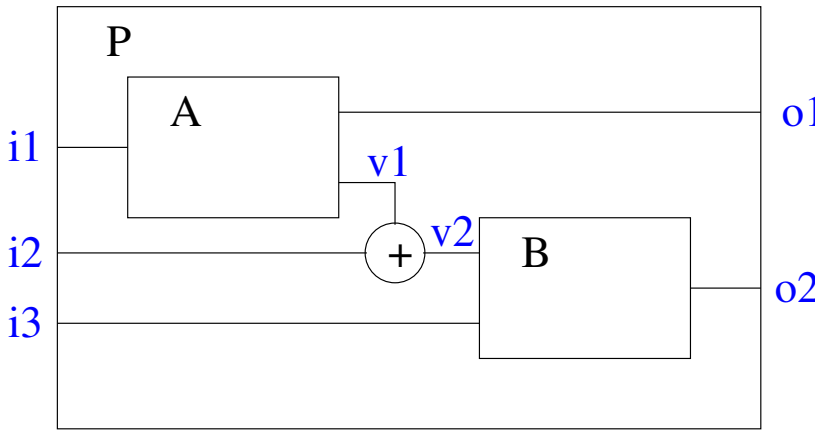
- Impossible de traiter des méta-modèles avec des centaines de classes
- Difficile d'accommoder des traits OO (sous-classes, héritage) de manière modulaire

Abstractions: Langage du style d'un "Signal modulaire":

- Essentiellement composition de blocs de flow de données
- . . . connectés par des fils

Limitations et abstractions (2)

Représentation graphique . . .



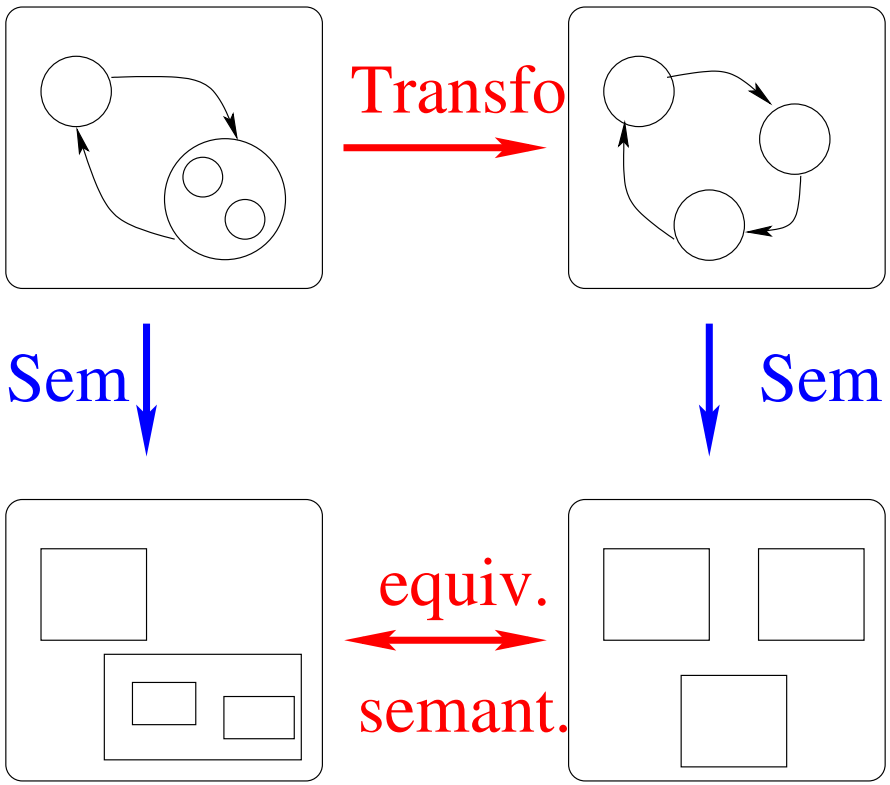
. . . vs. textuelle

```
proc P(i1,i2,i3, o1,o2) {
let v1, v2 in
    A(i1, o1, v1)
|| v2 = v1 + i2
|| B(v2, i3, o2)
}
```

- représentation “classique” comme type de donnée
- permet un raisonnement par induction

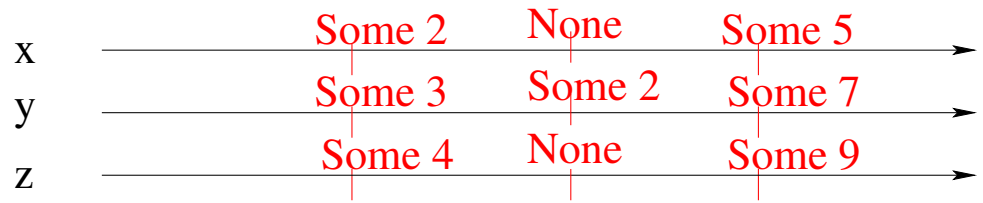
Extensions vers d'autres constructeurs

par exemple: codage d'automates



Plan

Signaux et traces




```
types ('t,'v) signal = 't ⇒ 'v option
```

```
types ('t,'n,'v) trace = 't ⇒ 'n ⇒  'v option
```

- `'t` domaine temporel
- `'v` domaine des valeurs
  `'v option`: Événement présent / absent
- `'n` domaine des noms / variables

Signal, le langage

```
process Sampler = {integer N}
     (? event reset, tick ! integer val; event alarm)
  (| val := Count(reset default alarm)
   | mod := (val = N-1)$1 init true
   | alarm := when mod
   | val ^= reset ^+ tick
   |) where boolean mod;
end;
```

Modélisé actuellement:

- Expressions combinatoires
- Statements ("affectation" `:=`, composition parallèle, `where`)
- Process (sans paramètres statiques)
- En partie: contraintes de synchronisation

Signal: Expressions

```
datatype ('n,'v) expr
  =
    (* nommage explicite des variables *)
    Var 'n
  | (* (e_0, e_1) *)
    PairE (('n, 'v) expr) (('n, 'v) expr)
  |  (* f (e)  *)
    Fun funid  (('n, 'v) expr)
  | (* e $1 init v *)
    Pre 'v  (('n, 'v) expr)
  | (* e_0  when e_1 *)
    When (('n, 'v) expr)  (('n, 'v) expr)
  | (* e_1 default e_1 *)
    Default (('n, 'v) expr) (('n, 'v) expr)
```

Signal: Statements

```
datatype ('n, 'v) stmt
  =
    EmptyStmt
  | (* y := e *)
    Eq 'n  (('n, 'v) expr)
  | (*  c_1 | c_2 *)
    Par (('n, 'v) stmt) (('n, 'v) stmt)
  | (* let x in c  *)
    Letv 'n  (('n, 'v) stmt)
  | (* P(i_1 .. i_m, o_1 .. o_n) *)
    PCall ('n cname) ('n list)
```

Procédures

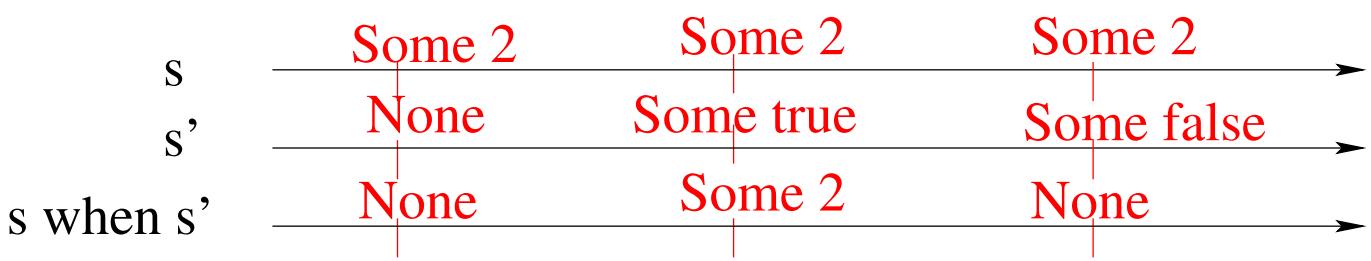
```
datatype ('n, 'v) proc
  = Proc (('n, proc_header) proc_interf)
         (('n, 'v) stmt)
```

Fonctions sur les signaux (1)

. . . pour préparer la sémantique des expressions.
Fonctions sans restriction du domaine temporel:

```
when :: ('t,'v) signal ⇒ ('t,'v) signal
        ⇒ ('t,'v) signal

when s s' == λ t.
             case (s' t) of
               None ⇒ None
             | Some v ⇒
               if (true_val v) then (s t) else None
```

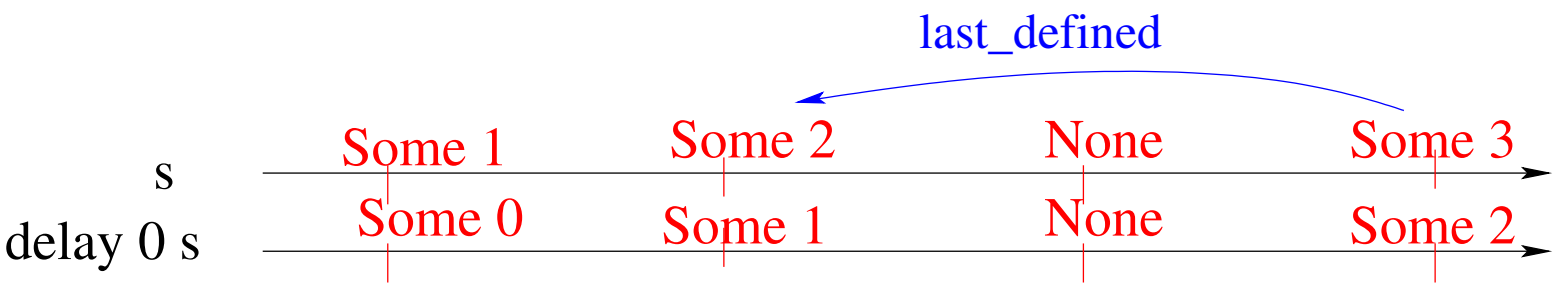


Fonctions sur les signaux (2)

Fonctions avec restriction du domaine temporel:

```
delay :: 'v ⇒ ('t::linorder, 'v) signal
      ⇒ ('t, 'v) signal

delay v s == λ t.
           case (s t) of
             None ⇒ None
           | Some v' ⇒ Some (last_defined v s t)
```



Signaux “raisonnables” ≡ ayant un domaine discret

Plan

Forme canonique (1)

But: Éviter des expressions imbriquées du style:

```
y := (x $1 init 0) when (b $1 init true)
```

Transformation en forme canonique:

```
(|
 | y := x' when b'
 | x' := (x $1 init 0)
 | b' := (b $1 init true)
 |) where { x', b' }
```

Forme canonique (2)

Expressions: Uniquement variables, paires, fonctions
Statements:

```
datatype ('n,'v,'tp) canon_stmt
= CanEmptyStmt
| CanCombin 'n  (('n,'v) canon_expr)
| CanPre 'n 'v 'n
| CanWhen 'n 'n 'n
| CanDefault 'n 'n 'n

| CanPar(('n,'v,'tp)canon_stmt) (('n,'v,'tp)canon_stmt
| CanPCall ('n cname) ('n list)
| CanLetv 'n 'tp (('n,'v,'tp) canon_stmt)
```

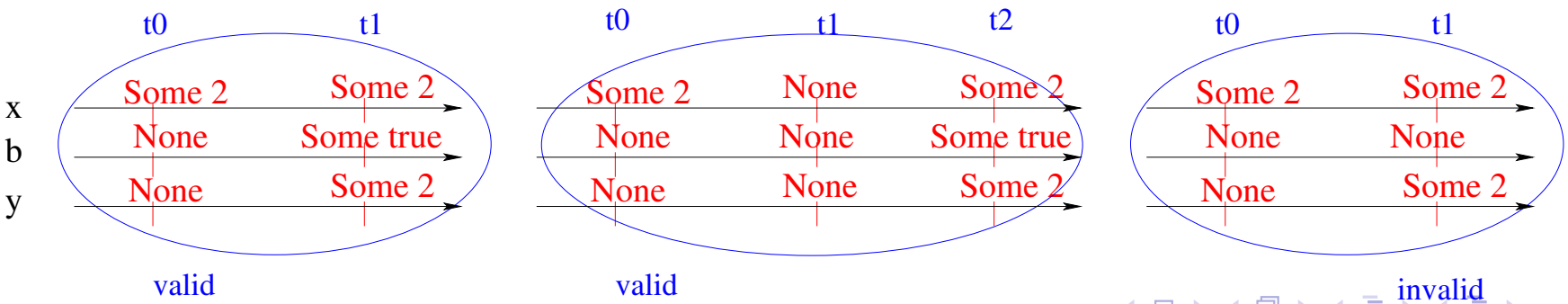
Annotations avec un type d'horloge $\rightsquigarrow$ plus tard

Sémantique: Les cas simples

Première approximation: “Trace *tr* est modèle du statement *c*”

```
inductive
  interp_canon_stmt ::
  ('t,'n,'v) trace ⇒ ('n, 'v, 'a) canon_stmt ⇒ bool
where
| interp_canonCanWhen:
    trace_proj tr y =
    when (trace_proj tr x) (trace_proj tr b)
  ⟹ interp_canon_stmt tr (CanWhen y x b)
```

Exemple: `y = x when b`



Sémantique avec erreurs (1)

Quel est le résultat pour (x, y) si x et y ne sont pas synchrones?

1. Sémantique sans erreurs:

```
signalpair :: ('t,'v) signal ⇒ ('t,'v) signal
      ⇒ ('t,'v) signal
signalpair s s' == λ t. lift_pair (s t) (s' t)
lift_pair (Some v)(Some v') = Some(pair_val (v,v'))
lift_pair _          _          = None
```

2. Sémantique avec erreurs:

```
datatype 'v error = Err | OK 'v
signalpair_err :: ('t,'v error) signal
  ⇒ ('t,'v error) signal ⇒ ('t,'v error) signal
lift_pair_err None None      = None
lift_pair_err(Some v) None =Some Err (& viceversa)
lift_pair_err (Some v) (Some v')
         = Some (lift_err v v')
```

Sémantique avec erreurs (2)

Extension: Deux variantes de sémantique d'expressions:

1. `interp_canon_expr error_fns_flat tr e`
2. `interp_canon_expr error_fns_lifted tr e`

Deuxième approximation: "Trace *tr* est modèle du statement *c*, prenant en compte / ne prenant pas en compte des erreurs"
Cas `y := e`, blocage en cas d'erreur

```
inductive
  interp_canon_stmt :: sync_error_handling ⇒
       ('t,'n,'v) trace ⇒
       ('n, 'v, 'a) canon_stmt ⇒ bool
where
| interp_canonCanCombin_bloc:
   s = (interp_canon_exp error_fns_lifted tr e) ∧
   error_free_signal s ∧
   trace_proj tr y = cancel_err_signal s
 ⟹ interp_canon_stmt Blocking tr (CanCombin y e)
```

Sémantique avec erreurs (3)

Cas `y := e`, non-blocage en cas d'erreur

```
| interp_canonCanCombin_nonbloc:
    s = (interp_canon_exp error_fns_lifted tr e) ∧
    trace_proj tr y = cancel_err_signal s
 ⟹ interp_canon_stmt Permissive tr (CanCombin y e)
```

Fonctions auxiliaires:

```
consts   cancel_err :: 'v error ⇒ 'v
primrec  cancel_err (OK v) = v

constdefs
  cancel_err_signal ::
      ('t, 'v error) signal ⇒ ('t, 'v) signal
  cancel_err_signal s == (option_map cancel_err) ∘ s
```

Plan

Horloges (1)

Horloge: ensemble d'instants de la présence d'un signal

- $cl(x)$: instants de la présence de x
- $[y > 5]$: instants quand $y > 5$ est présent et vrai

Calcul d'horloges: établit les conditions sous lesquelles un ensemble de signaux est bien synchronisé.

- Statement en signal: `z := x when (y > 5)`
- Condition d'horloge dérivée: $cl(z) = cl(x) \cap [y > 5]$

Horloges (2)

- *Problème:* Est-ce que `x when (y > 5)` et `x when (y > 2+3)` sont synchrones? (besoin: calcul effectif!)
- *Approche:* Introduire une abstraction d'expressions booléennes.

Comparaison avec *Lucid Synchrone:*

- Uniquement expressions `x when c`, où `c : clock`.
- Des horloges différentes sont incompatibles:

```
let node f x y =
    let clock cx = (y > 5) in
    let clock cy = (y > 2 + 3) in
    let z = (x when cx) + (y when cy) in  1
                          ^^^^^^^^^^
This expression has clock 'a on ?_cy0,
but is used with clock 'a on ?_cx1.
```

Calcul d'Horloges

Ce qui reste à faire:

- Définition de `clexpr` (correspondent aux `expr` de Signal).
  Sémantique: ensemble d'instants
- Définition de `clstmt` (correspondent aux `stmt` de Signal).
  Sémantique: booléen
  $\rightsquigarrow$ Calcul pour déterminer la validité d'un `clstmt`
- Extraction d'un `clstmt` à partir d'un `stmt`
- Montrer *type soundness*:
  Si `stmt` *c* a un `clstmt` valide
  et si *tr* est une trace qui est un modèle de *c*,
  alors *tr* est une trace sans erreurs

Donc: Pour tout `stmt` *c* avec un `clstmt` valide,
les sémantiques sans / avec erreurs coïncident.

Exemple (1)

- Soit le block B avec entrées a, b, c et sorties x, y, z, défini par:

```
proc B(a, b, c, x, y, z) {
    x := a when c
|| y := x $1 init 0
|| z := y when c }
```

- Soit la spécification "d'interface" des entrées:
  $cl(b) = cl(a) \cap [c]$
- On dérive les équations d'horloge:
  (1) $cl(x) = cl(a) \cap [c]$
  (2) $cl(y) = cl(x)$
  (3) $cl(z) = cl(y) \cap [c]$
- Insertion et simplification donne les contraintes sur les sorties:
  $cl(x) = cl(y) = cl(z) = cl(a) \cap [c]$

Exemple (2)

Traitement des variables locales - première approche:

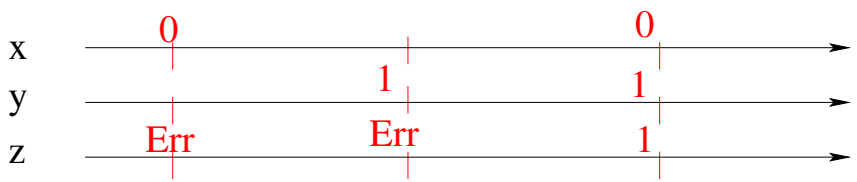
Programme:

```
let x, y in
    x := x $1 init 0
|| y := y $1 init 1
|| z = x + y
```

Contraintes:

$\exists x\ y.$
$cl(x) = cl(x) \wedge$
$cl(y) = cl(y) \wedge$
$cl(z) = cl(x) = cl(y)$
équivalent à *True*

Problème: Sans contraintes supplémentaires, le programme est éventuellement mal synchronisé.



Exemple (3)

Deuxième approche:

Programme:

```
let x, y
 with x ^= y ^= z
in
    x := x $1 init 0
|| y := y $1 init 1
|| z = x + y
```

Contraintes:

$\forall x\ y.$

$cl(x) = cl(y) = cl(z)$

$\longrightarrow$

$(cl(x) = cl(x) \wedge$

$cl(y) = cl(y) \wedge$

$cl(z) = cl(x) = cl(y))$

...aussi équivalent à *True*

Donc: Nécessaire de rajouter des contraintes de synchronisation pour des variables locales

Expressions d'Horloges

```
datatype ('n, 'a) clexpr
  = ClOf 'n          (* cl(x) *)
  | ClWhen 'a        (* [cond] *)
  | ClInter (('n, 'a) clexpr) (('n, 'a) clexpr)
  | ...
```

- `'n` domaine des noms / variables
- `'a` abstraction des expressions booléennes

et l'*interprétation* par rapport à une fonction de concrétisation `concr`:

```
interp_clexpr tr concr (ClOf x) = dom(trace_proj tr x)
interp_clexpr tr concr (ClWhen a) = (concr tr a)
interp_clexpr tr concr (ClInter ce ce') =
     (interp_clexpr tr concr ce) ∩
     (interp_clexpr tr concr ce')
```

Statements d'Horloges

```
datatype ('n, 'a) clstmt
  = ClEq (('n, 'a) clexpr) (('n, 'a) clexpr)
  | ClConj (('n, 'a) clstmt) (('n, 'a) clstmt)
  | ClForall 'n (('n,'a) clstmt)
  | ...
```

et l'*interprétation*:

```
interp_clstmt tr concr (ClEq e e') =
  (interp_clexpr tr concr e) =
  (interp_clexpr tr concr e')
interp_clstmt tr concr (ClConj c c') =
  interp_clstmt tr concr c ∧ interp_clstmt tr concr c'
interp_clstmt tr concr (ClForall v c) =
  ∀ s. interp_clstmt (trace_upd tr v s) concr c
```

Extraction de contraintes

```
consts clock_canon_stmt :: (('n, 'v) canon_expr ⇒ 'a)
      ⇒ ('n, 'v, 'a) canon_stmt  ⇒ ('n, 'a) clstmt
primrec
clock_canon_stmt abstr (CanCombin y e) =
  clock_combin y e
   (*  cl(y) = cl(x1) = ... = cl(xn),
         si fv(e) = {x1 .. xn} *)
clock_canon_stmt abstr (CanWhen y x b) =
  ClEq (ClOf y)
       (ClInter (ClOf x) (ClWhen (abstr (CanVar b))))
clock_canon_stmt abstr (CanPar c c') =
   ClConj (clock_canon_stmt abstr c)
          (clock_canon_stmt abstr c')
clock_canon_stmt abstr (CanLetv v clk c) =
   ClLet v clk (clock_canon_stmt abstr c)
```

Type soundness

Si une trace *tr* est un modèle (non-bloquant en cas d'erreur) pour un statement *c*
et si les contraintes d'horloge pour *c* sont satisfiables sous *tr*
alors *tr* est un modèle (bloquant en cas d'erreur) pour *c*

```
interp_clstmt tr concr (clock_canon_stmt abstr c)  ∧
tpi = (λ tr'. interp_clstmt tr' concr)
 ⟶
  interp_canon_stmt Blocking   tr cpenv tpi c
= interp_canon_stmt Permissive tr cpenv tpi c
```

et quelques conditions auxiliaires
Autrement dit: Si *c* admet un modèle *tr* et *c* est bien typé, alors *c* ne provoque pas d'erreur de synchronisation.

Plan

Résumé de la démarche (1)

Donné: un statement *c*
But: Montrer l'équivalence des deux sémantiques pour *c*:

```
  interp_canon_stmt Blocking   tr ... c
= interp_canon_stmt Permissive tr ... c
```

Précondition: Montrer

```
interp_clstmt tr concr (clock_canon_stmt abstr c)
```

Démarche: (pour `abstr` et `concr` donnés):

1. Extraire des contraintes d'horloge (`clock_canon_stmt`)
2. Déterminer la validité de `interp_clstmt`

pour n'importe quel `tr` ?!?

Résumé de la démarche (2)

Mieux: Remplacer "interprétation" par "solveur correct"
`solve_clstmt` avec:

```
solve_clstmt concr constr ⟶
∀ tr. interp_clstmt tr concr constr
```

Démarche:

1. Extraire des contraintes:
   `constr = (clock_canon_stmt abstr c)`
2. Vérifier que `solve_clstmt concr constr`
3. Conclure (correction du solveur, type soundness):

```
∀ tr.
    interp_canon_stmt Blocking   tr ... c
 = interp_canon_stmt Permissive tr ... c
```

Domaines d'application

- Horloges "classiques": `abstr` associe à une variable x les instants d'activation
- Intervalles d'activation: `abstr` associe à une variable x des intervalles $[l_x, u_x]$ d'activation
- Horloges périodiques

Travaux connexes:

- Horloges périodiques (Onera: J. Forget, F. Boniol, C. Pagetti)
- N-synchronous Kahn networks (M. Pouzet, C. Pagetti *et al*)

Horloges périodiques (1)

. . . pour des signaux activés avec une *période* et un *déphasage*

- $x : [|10t|]$ signifie: x est actif toutes les 10 unités de temps: 0, 10, 20, 30, . . .
- $y : [15t + 5]$ signifie: y est actif toutes les 15 unités de temps avec un déphasage de 5: 5, 20, 35, 50 . . .

Programmes bien / mal typés:

- *Incorrect* (`x` et `y` pas synchrones):

```
y = x
```

- *Correct:*

```
let z : [|30t + 20|] in
z = (x when [30t + 20]) + (y when [10t])
```

- *Correct* (sans horloges périodiques):

```
b1 = a1 when c ||  b2 = a2 when c  || r = b1 + b2
```

pourvu que $cl(a1) = cl(a2)$

Horloges périodiques (2)

Expressions d'horloge e de la forme:

- $cl(x)$, pour x une variable
- $[|at + b|]$, avec a, b des constantes de type `nat`
- union, intersection d'expressions

Statements d'horloge c de la forme:

- $e_1 = e_2$
- $c_1 \wedge c_2$
- quantification "qualifiée": $\forall v.cl(v) = e \longrightarrow c$

Extraction de contraintes

Etant donné:

- Un programme `p`
- la fonction `abstr` avec (essentiellement):
  - $x \mapsto cl(x)$
  - $[at + b] \mapsto [|at + b|]$

Extraction de statements d'horloge à l'aide de

`clock_canon_stmt abstr p`

Exemple:

- Programme:

```
let z : [|30t + 20|] in
z = (x when [30t + 20]) + (y when [10t])
```

- Contraintes:
  $\forall z.cl(z) = [|30t + 20|] \longrightarrow$
  $cl(z) = cl(x) \cap [|30t + 20|] = cl(y) \cap [|10t|]$

Résolution de contraintes

Synthèse:

1. Élimination des quantificateurs de la forme $\forall v.cl(v) = e \longrightarrow c$
   Exemple: $\forall z.cl(z) = [|30t + 20|] \longrightarrow$
   $cl(z) = cl(x) \cap [|30t + 20|] = cl(y) \cap [|10t|]$
   $\rightsquigarrow$
   $[|30t + 20|] = cl(x) \cap [|30t + 20|] = cl(y) \cap [|10t|]$
   Il reste: conjonction d'équations $e_1 = e_2$, avec
   e_i des variables "libres" $cl(x)$, des ens. $[|at + b|]$, intersection, union
2. Élimination des variables libres
3. Résolution d'équations avec des ensembles de la forme $[|at + b|]$

Résolution de contraintes: Variables libres (1)

Idée générale:

1. Mettre $e_1 = e_2$ en forme $e_1 \subseteq e_2 \wedge e_2 \subseteq e_1$
2. Éliminer $\cup$ à gauche et $\cap$ à droite:
   $X \subseteq Y \cap Z \rightsquigarrow X \subseteq Y \wedge X \subseteq Z$
   $X \cup Y \subseteq Z \rightsquigarrow X \subseteq Z \wedge Y \subseteq Z$
   Il reste: inclusions $\bigcap_i X_i \subseteq \bigcup_j Y_j$,
   où X_i, Y_j sont de la forme $cl(x)$ ou $[|at + b|]$
3. Pour éliminer $cl(v)$, regarder les cas:
   - $cl(v) \cap X \subseteq cl(v) \cap Y \rightsquigarrow True$
   - $\bigcap_i X_i \subseteq cl(v) \cap Y \rightsquigarrow \bigcap_i X_i \subseteq Y$
     si $cl(v) \notin \{X_i\}$
     Préservation de la *validité:* interpréter $cl(v)$ comme $\{\}$
   - $cl(v) \cap X \subseteq \bigcup_j Y_j \rightsquigarrow X \subseteq \bigcup_j Y_j$ si $cl(v) \notin \{Y_j\}$

Résolution de contraintes: Variables libres (2)

Détails à élaborer:

- Variables doivent être éliminées "successivement"
- Dépend d'un ordre partiel sur les variables (*acyclicité*)

Résolution de contraintes: Ensembles affins

(TER de Rémy Wyss)

But: Vérifier des contraintes de la forme $\bigcap_i [|a_i t + b_i|] \subseteq \bigcup_j [|a_j t + b_j|]$

Étapes:

1. Mettre contrainte sous la forme $\bigcap_i [|a_i t + b_i|] \cap -(\bigcup_j [|a_j t + b_j|]) = \{\}$
2. Appliquer des transformations ensemblistes pour obtenir une union d'intersections: $\bigcup_j \bigcap_i [|a_{i,j} t + b_{i,j}|] = \{\}$
3. Vérifier que chaque $\bigcap_i [|a_{i,j} t + b_{i,j}|]$ est vide

Calculer $[|a_0 t + b_0|] \cap [|a_1 t + b_1|]$:

- un ensemble de la forme $[|at + b|]$
  Exemple: $[|3t + 2|] \cap [|5t + 0|] = [|15t + 5|]$
- vide: $[|3t + 1|] \cap [|3t + 2|] = \{\}$

selon a_0, b_0, a_1, b_1

Horloges périodiques: Conclusion

Travail à faire:

- Comparaison formelle avec travaux connexes (Onera; Pouzet *et al*)
- *Inférence* d'horloges (et pas seulement vérification)

Extensions:

- Solveurs de contraintes modulaires et vérifiés
- Applications à d'autres domaines