EMSoC Recherche, octobre 2008

Lutin : Programmation de systèmes réactifs indéterministes

Pascal Raymond

Erwan Jahier

Yvan Roux

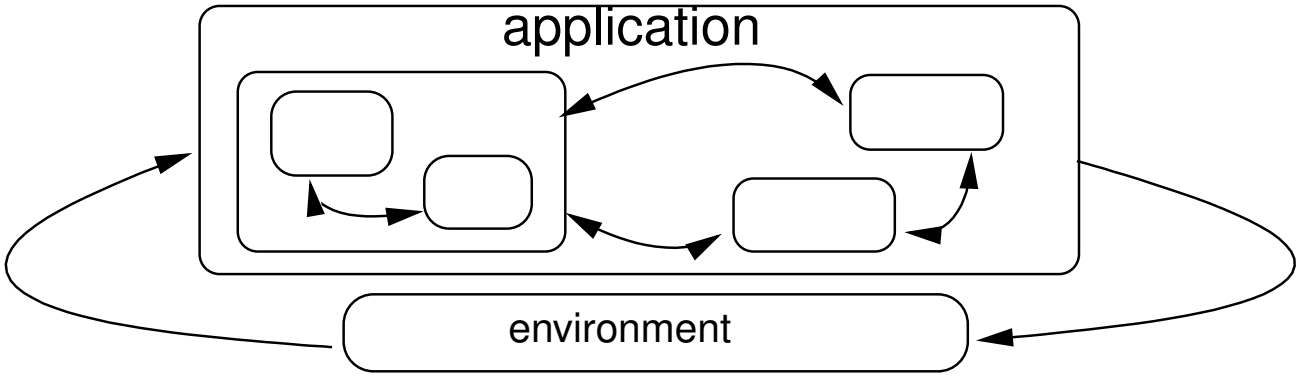
VERIMAG, Grenoble

Sommaire

Introduction

Domaine : les systèmes réactifs

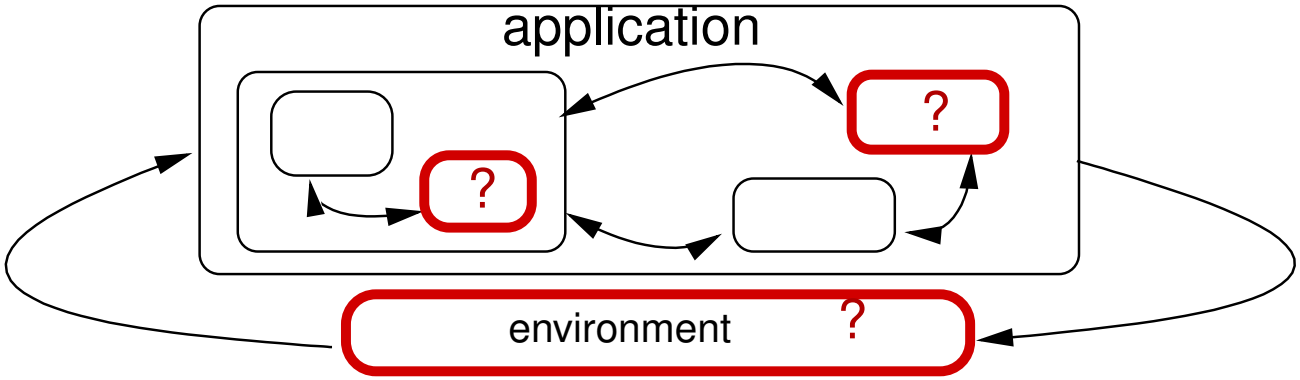
- **Séquence infinie de réactions entrées/sorties, systèmes parallèles et hiérarchiques typiques**
- **Systèmes critiques, déterminisme souhaité**



Introduction

Domaine : les systèmes réactifs

- Séquence infinie de réactions entrées/sorties, systèmes parallèles et hiérarchiques typiques
- Systèmes critiques, déterminisme souhaité



Cependant, l'indéterminisme peut être utile :

- pour le test (simulation de l'environnement),
- prototypage (abstraction des modules indisponibles).

But : décrire et simuler

- **Applications visées : test, simulation/prototypage ...**
- **Définir un langage adéquat pour décrire l'indéterminisme**

But : décrire et simuler

- **Applications visées : test, simulation/prototypage ...**
- **Définir un langage adéquat pour décrire l'indéterminisme**
  - ⋆ **indéterminisme $\neq$ chaotique : comportements réalistes/intéressants, conforme à des propriétés temporelles connues**

But : décrire et simuler

- **Applications visées : test, simulation/prototypage ...**
- **Définir un langage adéquat pour décrire l'indéterminisme**
  - ⋆ **indéterminisme $\neq$ chaotique : comportements réalistes/intéressants, conforme à des propriétés temporelles connues**
  - ⋆ **l'indéterminisme ne suffit pas : simuler nécessite d'exprimer des « probabilités » (ex. un comportement est plus probable qu'un autre)**
- **Fournir des outils pour animer/simuler les descriptions**
  - ⋆ **Sémantique claire (simple ?) : l'indéterminisme est voulu et maîtrisé, il n'est pas la conséquence d'un vide sémantique.**
  - ⋆ **Un seul comportement = des centaines de réactions, le simulation doit être aussi efficace que possible.**

Quel est le « bon » style de programmation ?

- **Style déclaratif : naturel pour décrire les contraintes (qui sont des relations entre entrées/sorties).**
- **Impératif/séquentiel : naturel pour décrire des scénarios (séquence, boucle, branchements ...).**

Quel est le « bon » style de programmation ?

- **Style déclaratif : naturel pour décrire les contraintes (qui sont des relations entre entrées/sorties).**
- **Impératif/séquentiel : naturel pour décrire des scénarios (séquence, boucle, branchements ...).**

Notre solution, deux niveaux de langage :

- **déclaratif, à la Lustre, pour les contraintes atomiques**
- **structures de contrôle pour construire les scénarios :**
  - ⋆ **inspirées par les expressions régulières,**
  - ⋆ **enrichies avec des structures plus complexes, à la Esterel.**

Le langage

Système réactif

- Un programme Lutin décrit le comportement d'un système réactif indéterministe. L'entête d'un programme est :

```
system foo (x : int; y : bool)
returns (a,b :bool; c :int) =   trace-exp
```

- Le corps *trace-exp* décrit les comportements possibles comme un « langage formel » dont les symboles de base sont des ***contraintes*** sur la valeur des entrées/sorties (appelées les ***variables du support***).

Réactions

- **Un comportement atomique est une réaction indéterministe.**
- **Une telle réaction est décrite par une contrainte entre les valeurs courantes et précédentes des variables du support (`x` et `pre x`) :**

  data-exp **::= expression logique**

Réactions

- **Un comportement atomique est une réaction indéterministe.**
- **Une telle réaction est décrite par une contrainte entre les valeurs courantes et précédentes des variables du support (`x` et `pre x`) :**

  data-exp **::= expression logique**
- **Les variables peuvent être :**
  - ⋆ **contrôlable (sorties),**
  - ⋆ **incontrôlable (entrées, `pre`s).**

Réactions

- **Un comportement atomique est une réaction indéterministe.**
- **Une telle réaction est décrite par une contrainte entre les valeurs courantes et précédentes des variables du support (`x` et `pre x`) :**
  data-exp **::= expression logique**
- **Les variables peuvent être :**
  - ⋆ **contrôlable (sorties),**
  - ⋆ **incontrôlable (entrées, `pre`s).**
- **Faire une réaction, pour des valeurs données des variables incontrôlables, consiste à générer aléatoirement des valeurs pour les contrôlables qui satisfassent la contrainte.**
- **N.B. S'il n'existe aucune solution, une exception « deadlock » est levée**

Exemple

Soit `x` une entrée booléenne, `c` une sortie réelle, et soit la contrainte :

```
(x and (c <= 10.0) and (c > pre c))
```

Exemple

Soit `x` une entrée booléenne, `c` une sortie réelle, et soit la contrainte :

```
(x and (c <= 10.0) and (c > pre c))
```

- **si `x` est vraie, et `pre c` < 10, alors la contrainte admet des solutions :**
  - ⋆ **une valeur est choisie aléatoirement dans** $]$`pre c`$, 10.0]$

Exemple

Soit `x` une entrée booléenne, `c` une sortie réelle, et soit la contrainte :

```
(x and (c <= 10.0) and (c > pre c))
```

- **si `x` est vraie, et `pre c` < 10, alors la contrainte admet des solutions :**
  - ⋆ **une valeur est choisie aléatoirement dans $]$`pre c`$, 10.0]$**
- **sinon, la contrainte n'a pas de solution,**
  - ⋆ **on dit qu'elle lève un « deadlock »**

Exemple

Soit `x` une entrée booléenne, `c` une sortie réelle, et soit la contrainte :

```
(x and (c <= 10.0) and (c > pre c))
```

- **si `x` est vraie, et `pre c` < 10, alors la contrainte admet des solutions :**
  - ⋆ **une valeur est choisie aléatoirement dans $]$`pre c`$, 10.0]$**
- **sinon, la contrainte n'a pas de solution,**
  - ⋆ **on dit qu'elle lève un « deadlock »**
- **cette exception, locale, ne conduit pas forcément à une erreur globale : les structures de contrôle, qu'on va voir maintenant, sont justement conçues pour gérer ces blocages locaux.**

Le flot de contrôle de base

- **Les réactions atomiques sont combinées dans des instructions impératives (appelées *expressions de traces*),**
- **les instructions de base sont inspirées des expressions régulières :**

trace-exp ::= *data-exp* **(doit être booléenne)**

| *trace-exp* `fby` *trace-exp* **(séquence)**

| `loop` *trace-exp* **(boucle non bornée)**

| `{` *trace-exp* `|` ... `|` *trace-exp* `}` **(choix indéterministe)**

Exemple :

```
((x > 0) and (x <= 10)) fby
loop {
    (on and (x > pre x))
|   (x = pre x)
}
```

Indéterminisme et deadlock

La sémantique des instructions indéterministes est liée à l'exception deadlock, en première approximation :

- **une** *trace-exp* **est *démarrable* si elle ne lève pas immédiatement un deadlock.**

Indéterminisme et deadlock

La sémantique des instructions indéterministes est liée à l'exception deadlock, en première approximation :

- **une** *trace-exp* **est *démarrable* si elle ne lève pas immédiatement un deadlock.**
- « { *t1* | ... | *tn* } » **est démarrable si au moins une des branches l'est :**
  - ⋆ **si c'est le cas, une branche démarrable est sélectionnée et on lui passe le contrôle,**
  - ⋆ **sinon le choix lève un deadlock.**

Indéterminisme et deadlock

La sémantique des instructions indéterministes est liée à l'exception deadlock, en première approximation :

- **une** *trace-exp* **est *démarrable* si elle ne lève pas immédiatement un deadlock.**
- « { *t1* | ... | *tn* } » **est démarrable si au moins une des branches l'est :**
  - ⋆ **si c'est le cas, une branche démarrable est sélectionnée et on lui passe le contrôle,**
  - ⋆ **sinon le choix lève un deadlock.**
- « `loop` *t* » **se comporte :**
  - ⋆ **comme** « *t* `fby loop` *t* »**si** *t* **est démarrable,**
  - ⋆ **est « effacée » sinon (i.e. la boucle s'arrête)**

Indéterminisme et deadlock

La sémantique des instructions indéterministes est liée à l'exception deadlock, en première approximation :

- une *trace-exp* **est *démarrable* si elle ne lève pas immédiatement un deadlock.**
- « { *t1* | ... | *tn* } » **est démarrable si au moins une des branches l'est :**
  - ⋆ **si c'est le cas, une branche démarrable est sélectionnée et on lui passe le contrôle,**
  - ⋆ **sinon le choix lève un deadlock.**
- « `loop` *t* » **se comporte :**
  - ⋆ **comme** « *t* `fby loop` *t* »**si** *t* **est démarrable,**
  - ⋆ **est « effacée » sinon (i.e. la boucle s'arrête)**
  - ⋆ **sémantique intuitive : boucle tant que possible**

Autres instructions utiles

- `assert` *data-exp* `in` *trace-exp*

  distribue la contrainte *data-exp* **(expression booléenne) tout au long du comportement** *trace-exp*

- `local` *ident* `:` *type* `in` *trace-exp*

  déclare une variable de support locale (similaire à une sortie cachée)

- `try` *trace-exp* `do` *trace-exp*

  si et quand la première *trace-exp* **lève un deadlock, le contrôle passe à la deuxième.**

Contrôler l'indéterminisme

- **poids relatifs (`1` par défaut) :**

  `{` *trace-exp* `weight` *w1* `|` **...** `|` *trace-exp* `weight` *wn* `}`

  où les *wi* **sont des expressions entières** ***incontrôlables***

- **itérations limitées dans un intervalle :**

  `loop [`*min*`,` *max*`]` *trace-exp*

  où *min* **et** *max* **sont des** ***constantes*** **entières**

- **itérations suivant une loi normale :**

  `loop ~`*av* `:` *sd trace-exp*

  où *av* **et** *sd* **sont des** ***constantes*** **entières**

Boucles instantanées

- **Peut-on accepter des itérations « instantanées » ?**
- **Avec la sémantique intuitive : « `loop loop` *c* » boucle indéfiniment sans rien faire si *c* est insatisfiable**

Principe des boucles bien fondées

Précise la sémantique des boucles : si on décide de faire une itération, alors cette itération doit générer au moins une réaction.

En terme de langages réguliers : `loop` $t = (t \setminus \varepsilon)^*$

Indéterminisme, deadlock et probabilités

- **Principe de réactivité : un choix ne peut bloquer que si toutes les possibilités bloquent.**
- **N.B. la réactivité est prioritaire sur les poids :**

  `{` *t1* `weight` **1000000** `|` *t2* `weight` **1** `}`

  malgré son poids relatif infime, *t2* **sera choisi si** *t1* **n'est pas démarrable.**

- **Exemple :** `{X weight 3 | Y weight 5 | Z }`

Les probabilité effectives sont :

indémarrables	X	Y	Z	deadlock
$\emptyset$	3/9	5/9	1/9	non
{X}	0	5/6	1/6	non
{Y}	3/4	0	1/4	non
...	...	...	...	...
{X,Y}	0	0	1	non
...	...	...	...	...
{X,Y,Z}	0	0	0	oui

Indéterminisme, deadlock et priorité

On a parfois besoin d'un choix *parfaitement déterministe*, qui ne peut être atteint avec le mécanisme des poids : même avec un poids infime, une branche démarrable a quand même une probabilité théorique d'être choisie à la place d'une branche « prioritaire ».

Indéterminisme, deadlock et priorité

On a parfois besoin d'un choix *parfaitement déterministe*, qui ne peut être atteint avec le mécanisme des poids : même avec un poids infime, une branche démarrable a quand même une probabilité théorique d'être choisie à la place d'une branche « prioritaire ».

D'où la définition d'une instruction *choix prioritaire* :

{ *t1* |> *t2* |> ... |> *tn* }

- **Sémantique intuitive *ou sinon ...***
- **Exemple typique :** { *optimal* |> *degraded* |> *rescue* |> *lost* }

Concurrence

- **Syntaxe :**

  { *trace-exp* **&>...** **&>** *trace-exp* }

- **Tout au long de l'exécution concurrente, chaque branche produit sa propre contrainte, la conjonction de toute les contraintes donne la contrainte globale.**
- **L'instruction termine si et quand toutes les branches ont terminé (cf. Esterel)**

  i.e. une branche qui termine devient « neutre »
- **Si (au moins) une branche lève un deadlock, le tout lève un deadlock.**

Concurrence et probabilités

Sont des notions peu compatibles, par exemple :

```
{ {X weight 1000 | Y } &> {A weight 1000 | B } }
```

Si X et A sont démarrable indépendamment, mais pas leur conjonction :

- **le comportement le plus probable peut être Y`&>`A, ce qui est injuste pour la première branche,**
- **ou alors X`&>`B, ce qui est injuste pour la seconde.**

Concurrence et probabilités

Sont des notions peu compatibles, par exemple :

`{ {`*X* `weight` 1000 `|` *Y* `} &> {`*A* `weight` 1000 `|` *B* `} }`

Si *X* **et** *A* **sont démarrable indépendamment, mais pas leur conjonction :**

- **le comportement le plus probable peut être** *Y*`&>`*A*, **ce qui est injuste pour la première branche,**
- **ou alors** *X*`&>`*B*, **ce qui est injuste pour la seconde.**

Il faut faire un choix de conception :

- **la première branche « joue » la première, la seconde devra faire avec, etc.**
- **i.e. on rend « l'inévitable injustice » claire et systématique (de gauche à droite).**
- **N.B. La syntaxe souligne cette non-commutativité.**

Exceptions programmées

Elles permettent de détourner le flot de contrôle normal (essentiellement séquentiel). Similaires à ce qu'on trouve dans les langages classiques (caml, Java, les trap d'Esterel etc.)

- **Déclaration/portée :**

  `exception` *ident* **-- global**

  `exception` *ident* `in` *trace-exp* **-- local**

- **Levée :** `raise` *ident*

- **Rattrapage :**

  `catch` *ident* `in` *t1* `do` *t2*

  si *ident* **est levée au cours de l'exécution de** *t1*, **le contrôle passe immédiatement à** *t2*.

Exceptions programmées (suite)

- **Raccourcis :**
  - ⋆ `trap` *x* `in` *t1* `do` *t2*

    pour : `exception` *x* `in` `catch` *x* `in` *t1* `do` *t2*
  - ⋆ `try` *t1* `do` *t2* **, déjà vu est en fait un raccourci pour :**

    `catch` `DeadLock` `in` *t1* `do` *t2*

- **Exception et concurrence, choix de conception :**
  - ⋆ **il n'y a jamais de « levées multiples »,**
  - ⋆ **tout comme les poids, les** `raise` **sont traités en séquence de gauche à droite.**
  - ⋆ **e.g.** `{raise` **E** `&>`*X* `}` $\Leftrightarrow$ `raise` **E**

Modularité

Le langage propose une « couche fonctionnelle simple » qui permet de :

- **partager des définitions,**
- **définir et réutiliser des « combinateurs » opérant sur les données mais aussi les comportements.**
- **Un type abstrait `trace` est défini, pour permettre le typage des combinateurs de comportements et de leurs paramètres.**
- **La sémantique est définie en terme de substitution (ce sont plus des macros que de « vraies » fonctions).**

Modularité (suite)

- **Une macro peut être globale (définie en dehors d'un `system`) :**

  `let` *ident* `(`*params*`)` `:` *type* `=` *exp*

  exp **est une** *trace-exp* **ou une** *data-exp*, **selon selon** *type*.

- **ou locale à une** *trace-exp* **:**

  `let` *ident* `(`*params*`)` `:` *type* `=` *exp* `in` *exp*

  auquel, les règles classiques de portée s'appliquent.

- **Les** *params* **d'entrée, et le** *type* **de sortie sont optionnels.**

- **Attention à ne pas confondre variables de support et macros sans paramètre (alias)**

Exemples

- **Combinateur de données : la relation « intervalle »**

```
let within(x, min, max : real) : bool =
            (min <= x) and (x <= max)
```

- **Combinateur de traces : contrainte initiale**

```
let assert_init(init : bool; t : trace) : trace =
trap Stop in {
    init -- cast implicite bool=trace atomique
&>
    t fby raise Stop
}
```

Exemples (suite)

- Parallèle qui termine dès que la deuxième branche termine :

```
let as_long_as(X, Y : trace) : trace =
    trap Stop in
        X &> {Y fby raise Stop}
    }
```

- Ou dès qu'une des deux termine :

```
let racing(X, Y : trace) : trace =
    trap Stop in
      {X fby raise Stop} &> {Y fby raise Stop}
    }
```

Paramètres et variables de support

- Le type `trace` est très abstrait : quid des variables support ?
- En fait, ça n'a pas d'importance en général :

  les opérateurs de traces (prédéfinis ou programmés) sont en général polymorphes

Paramètres et variables de support

- **Le type `trace` est très abstrait : quid des variables support ?**
- **En fait, ça n'a pas d'importance en général :**
  les opérateurs de traces (prédéfinis ou programmés) sont en général polymorphes
- **Le seul cas ou ce n'est pas vrai, c'est quand un combinateur nécessite un paramètre qui doit impérativement être une variable support, typiquement, si l'opérateur utilise le `pre` de ce paramètre.**
- **C'est pourquoi on peut sur-spécifier le type d'un paramètre :**
  x `:` *type* `ref`
- **Le système de type du compilateur fait le reste :**

```
let foo (x : bool) = ... pre x ... -- TYPE ERROR
let foo (x : bool ref) = ... pre x ... -- OK
```

Exemple

Relation « filtre du premier ordre » :

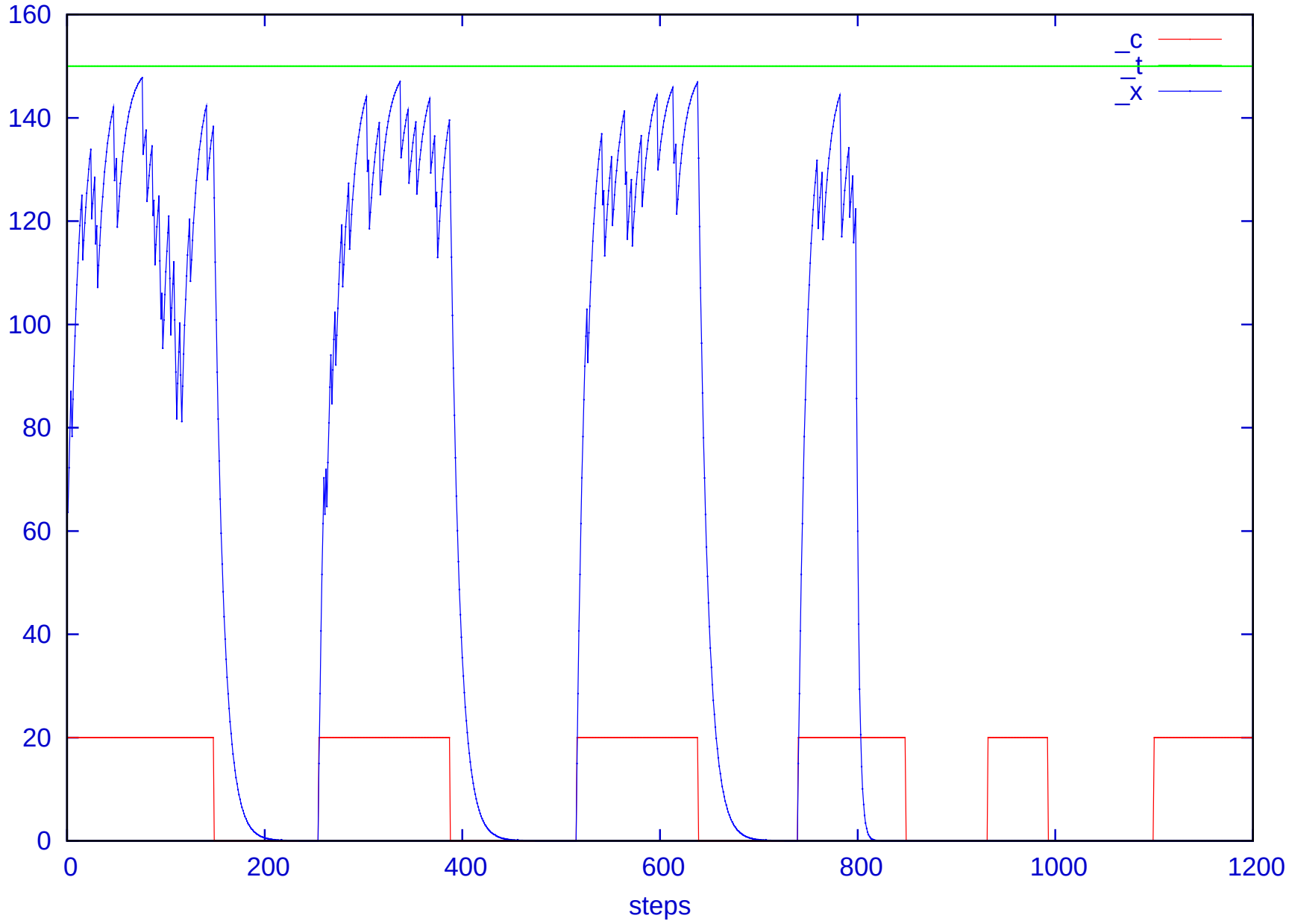
```
let fof (y : real ref; x, gain : real) : bool =
        (y = gain*(pre y) + (1.0-gain)*x)
```

Exemple

Un système complet :

- **La sortie `x` tend vers l'entrée `t` quand l'entrée `c` est vraie, sinon elle tend vers 0.**
- **Le système fonctionne à peu près correctement pendant à peu près 1000 réactions : il peut « rater » une commande `c` (à peu près une fois sur 10).**
- **Puis il tombe en panne et `x` tend rapidement vers 0.**

```
system foo(c : bool; t :real) returns (x : real) =
       within(x, -100.0, 100.0) fby
       local a : real in
       let gen_gain() : trace = loop {
           within(a, 0.8, 0.9)
           fby loop[30,40] (a = pre a)
       } in
       as_long_as (
            gen_gain(),
            loop~1000 :100 {
                 (c and fof(x, t, a)) weight 9
            |    fof(x, 0.0, a)
            }
       ) fby loop fof(x, 0.0, 0.7)
```


160
140
120
100
80
60
40
20
0
0
200
400
600
800
1000
1200
steps
_c
_t
_x

Conclusion

- Un langage qui mélange contraintes relationnelles et structures de contrôle.
- Une sémantique formellement définie (cf. papier Eurasip).
- Les programmes peuvent être :
  - ⋆ interprétés,
  - ⋆ compilés dans un format d'automate ad hoc

Notes sur les solveurs de contraintes

- Le solveur actuel traite la logique et les contraintes linéaires (à base de BDD et de polyèdres convexes).
- C'est la partie « coûteuse » des outils ...
- ... mais il peut être adapté/remplacé selon les besoins, e.g. dans la plupart des cas, un solveur d'intervalles suffit.