Langage de nouvelle génération pour la modélisation et la vérification formelle de systèmes parallèles asynchrones

Frédéric Lang

LIG et Inria/CONVECS

CADP

(*Construction and Analysis of Distributed Processes*)

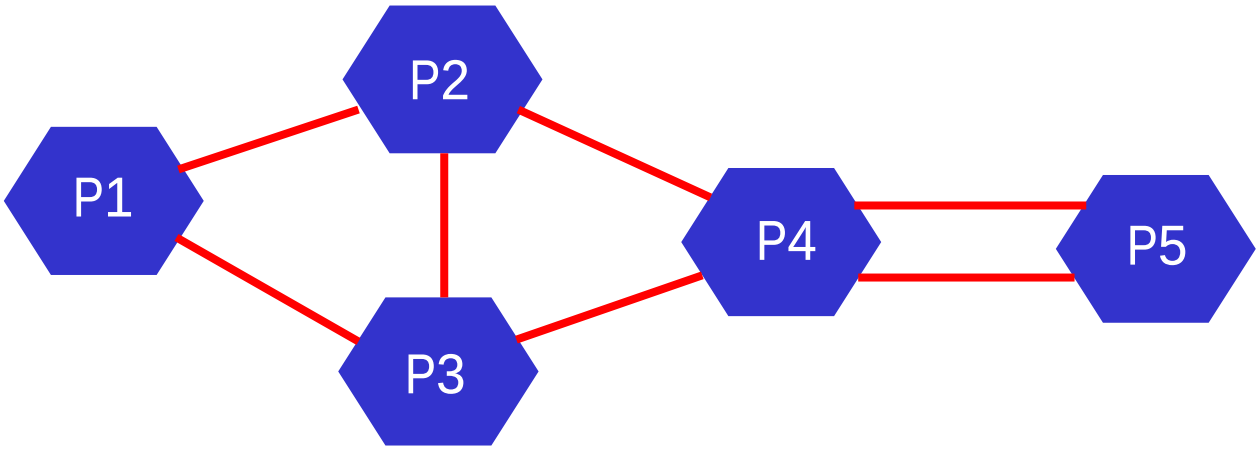
- Boîte à outils logiciels développée à l'Inria
  - Développement initié au milieu des années 80
- Nombreuses fonctionnalités (> 50 outils)
  - Modélisation formelle des systèmes parallèles asynchrones
  - Simulation, prototypage rapide, test, vérification, évaluation de performance
- Diffusion large
  - ≥ 441 contrats de licence académiques (gratuits) signés
  - CADP installé sur 1227 machines en 2012
  - Disponible sur de multiples architectures (processeurs, OS, compilateurs)

http://cadp.inria.fr

Modélisation formelle

- Exprimer mathématiquement le comportement des programmes parallèles critiques (pour en vérifier la correction)
- Les langages usuels (C, Java) ne conviennent pas
- **Bonne nouvelle** : on sait le faire (algèbres de processus, CSP, CCS, LOTOS)
  - expression correcte du parallélisme, non-déterminisme…
  - sémantique mathématique
- **Mauvaise nouvelle** : les industriels n'en veulent pas et inventent leurs propres langages
- **Défi** : combiner concepts mathématiques et notations usuelles

Systèmes parallèles asynchrones



- Différents processus (tâches, activités, agents, …)
- Qui s'exécutent en parallèle
- A des vitesses différentes (pas d'horloge centrale)
- Sans forcément de mémoire commune
- Communication par messages avec des délais variables

Domaines d'application

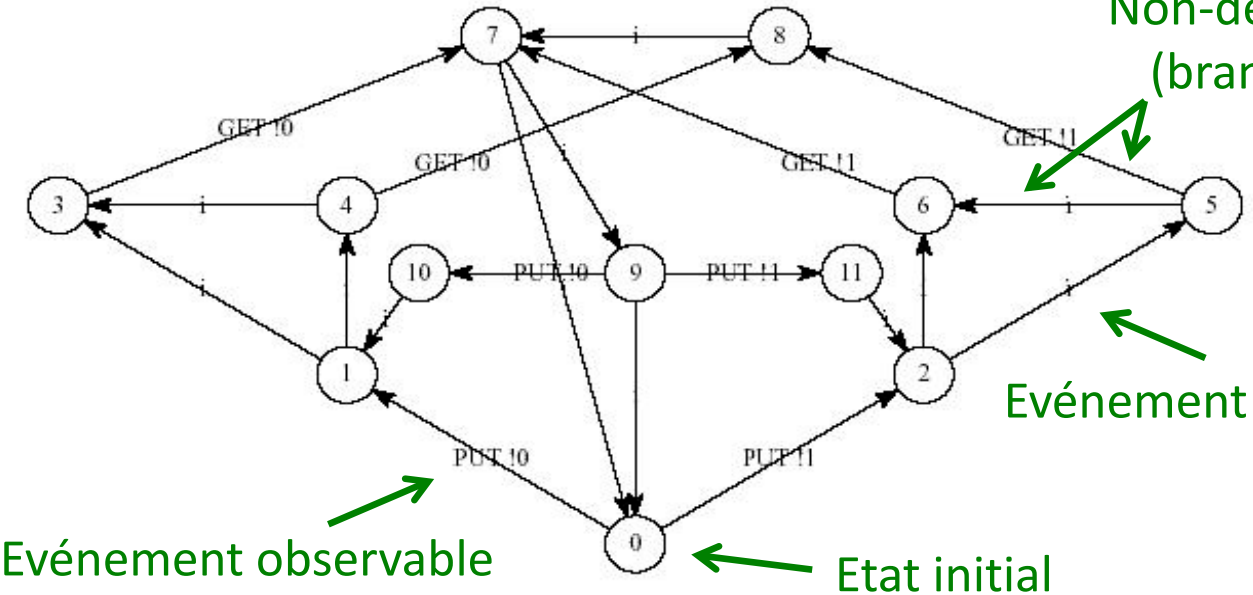
- Le parallélisme asynchrone ne se restreint pas à un domaine particulier
- 155 études de cas publiées dans les domaines suivants :

  avionics, bioinformatics, business processes, cognitive systems, communication protocols, component-based systems, constraint programming, control systems, coordination architectures, critical infrastructures, cryptography, database protocols, distributed algorithms, distributed systems, e-commerce, e-democracy, embedded software, grid services, hardware design, hardware/software co-design, healthcare, human-computer interaction, industrial manufacturing systems, middleware, mobile agents, model-driven engineering, networks, object-oriented languages, performance evaluation, planning, radiotherapy equipments, real-time systems, security, sensor networks, service-oriented computing, software adaptation, software architectures, stochastic systems, systems on chip, telephony, transport safety, Web services

  liste des études de cas : http://cadp.inria.fr/case-studies

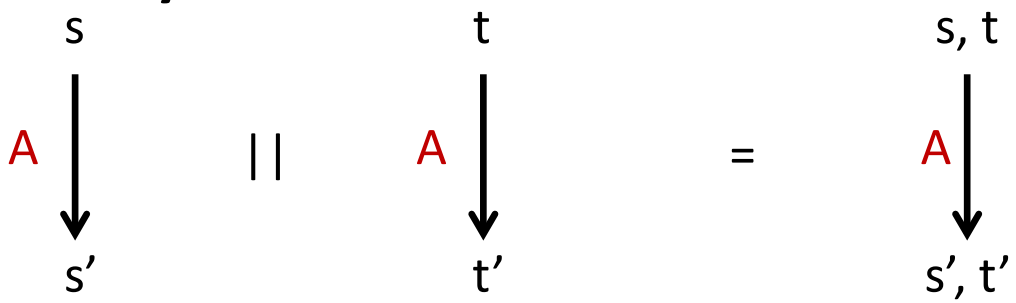
Modèle sémantique

- Le **comportement** d'un processus est caractérisé par un graphe : son STE (*Système de Transitions Etiquetées*)
  - Etats sans contenu visible, état initial
  - Transitions étiquetées par les **événements observables** (e.g., actions de communication) et **internes** (i)

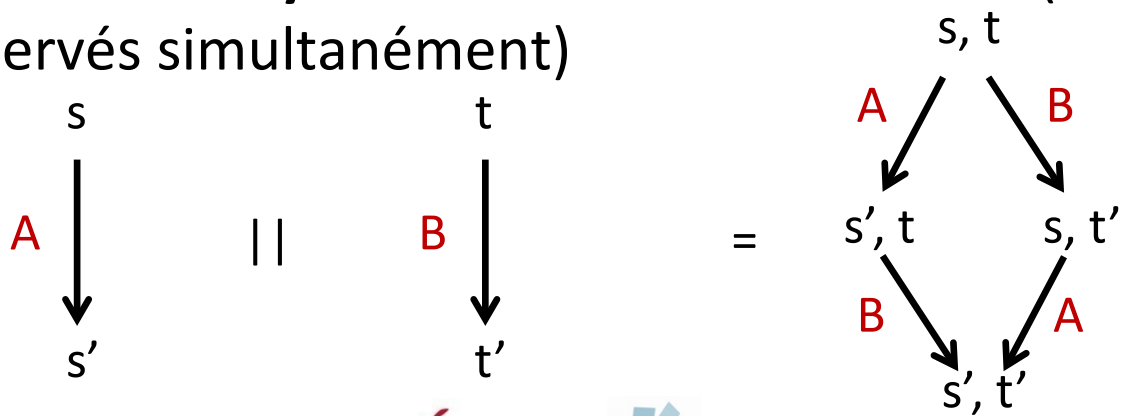


Sémantique du parallélisme asynchrone

- Possibilité de **synchroniser** des événements (rendez-vous)
- Sémantique de la composition parallèle = STE produit
  - **Evénements synchronisés** : exécution simultanée



  - **Evénements non-synchronisés** : entrelacement (ne peuvent pas être observés simultanément)



Vérification formelle

- **Objectif** : rechercher les erreurs éventuelles dans le STE
- Le **test** et la **simulation** sont utiles mais ne permettent pas de détecter certaines erreurs
- Utilisation de **méthodes énumératives** : exploration du STE
  - Pour **vérifier une propriété de logique temporelle** (e.g., l'évt X est obligatoirement suivi de l'évt Y) : ***model checking***
  - Pour **vérifier l'équivalence** avec un STE de référence : ***equivalence checking***
- Nécessité de lutter contre *l'explosion d'états* (mais c'est une autre histoire)

Langages formels pour la description du parallélisme asynchrone

- STE sont suffisamment expressifs pour décrire la plupart des systèmes et des modes de communication
  - Buffers, variables partagées, ...
- **Mais** de **trop bas niveau** pour décrire des systèmes de taille « réaliste »
- Besoin d'un **langage** combinant plusieurs caractéristiques
  - **Non-ambigü** : sémantique formelle sous forme de STE
  - **Riche** : données simples et complexes, non-déterminisme, parallélisme
  - **Structuré** : structures de contrôle, fonctions, processus, modules
  - **Accessible** sans trop d'effort à un programmeur non-spécialiste

→ Le langage LOTOS NT (abrégé en LNT)

Les racines : algèbres de processus

- Formalismes théoriques (*concurrency theory*) proposés à partir des années 80
- Initialement limitées aux concepts de base
  - récursion, parallélisme, synchronisation, non-déterminisme
  - pas de représentation des données
  - **Exemple** (CCS) : $P = a.P$; $Q = \grave{a}.b.Q$; $R = P \mid\mid Q$
- Puis LOTOS (norme internationale ISO 8807, 1989)
  - Combine algèbre de processus et types abstraits algébriques (ActOne) pour la représentation des données
  - Support dans CADP
  - Avancée importante **mais** langage ardu (2 en 1), apprentissage lent...

Exemple LOTOS : type énuméré

```
type AgId is Boolean
sorts AgId
opns
agent1 (*! constructor *),
agent2 (*! constructor *),
agent3 (*! constructor *): -> AgId
_eq_, _ne_, _lt_ :
        AgId, AgId -> Bool
```

```
eqns forall P1, P2 : AgId
ofsort Bool
   P1 eq P1 = true;
   P1 eq P2 = false;
ofsort Bool
   P1 ne P2 = not (P1 eq P2);
ofsort Bool
   agent1 lt agent2 = true;
   agent1 lt agent3 = true;
   agent2 lt agent3 = true;
   P1 lt P2 = false;
endtype
```

Types abstraits algébriques

Exemple LOTOS : type tableau

```
type AgArr is AgId, Natural
sorts AgArr
opns
AgArr (*! constructor *) :
   AgId, AgId, AgId -> AgArr
get : AgArr, Nat -> AgId
set : AgArr, Nat, AgId -> AgArr
```

```
eqns
forall a, a1, a2, a3 : AgArr
ofsort AgId
get (AgArr (a1, a2, a3), 1) = a1;
get (AgArr (a1, a2, a3), 2) = a2;
get (AgArr (a1, a2, a3), 3) = a3;
ofsort AgArr
set (AgArr (a1, a2, a3), 1, a) =
   AgArr (a, a2, a3);
set (AgArr (a1, a2, a3), 2, a) =
   AgArr (a1, a, a3);
set (AgArr (a1, a2, a3), 3, a) =
   AgArr (a1, a2, a);
endtype
```

Types abstraits algébriques

Exemple LOTOS : type ensemble

```
type AgSet is AgId, Boolean
sorts AgSet
opns
nil (*! constructor *) : -> AgSet
cons (*! constructor *) :
   AgId, AgSet -> AgSet
insert : AgId, AgSet -> AgSet
remove : AgId, AgSet -> AgSet
                eqns
                forall
                   a1, a2 : AgId,
                   s : AgSet
                ofsort AgSet
```

Types abstraits algébriques

```
insert (a1, nil) = cons (a1, nil);
a1 lt a2 =>
   insert (a1, cons (a2, s)) =
        cons (a1, cons (a2, s));
a1 eq a2 =>
   insert (a1, cons (a2, s)) =
        cons (a2, s);
insert (a1, cons (a2, s)) =
   cons (a2, insert (a1, s));
remove (a1, nil) = nil;
a1 lt a2 =>
   remove (a1, cons (a2, s)) =
        cons (a2, s);
a1 eq a2 =>
   insert (a1, cons (a2, s)) = s;
remove (a1, cons (a2, s)) =
   cons (a2, remove (a1, s));
```

Exemple LOTOS : fonction

```
opns
   Max_Ag : AgArr -> AgId
   Loop : AgId, Nat, AgArr
eqns
forall a : AgArr, n : Nat,
   result : AgId
```

```
ofsort AgId
   Max_Ag (a) =
        Loop (agent1, 1, a);
ofsort AgId
   n <= 3, get (a, n) > result =>
   Loop (result, n, a) =
        Loop (get (a, n), n+1, a);
   n <= 3 =>
   Loop (result, n, a) =
        Loop (result, n+1, a);
   Loop (result, n, a) = result;
```

Types abstraits algébriques

Exemple LOTOS : processus séquentiel

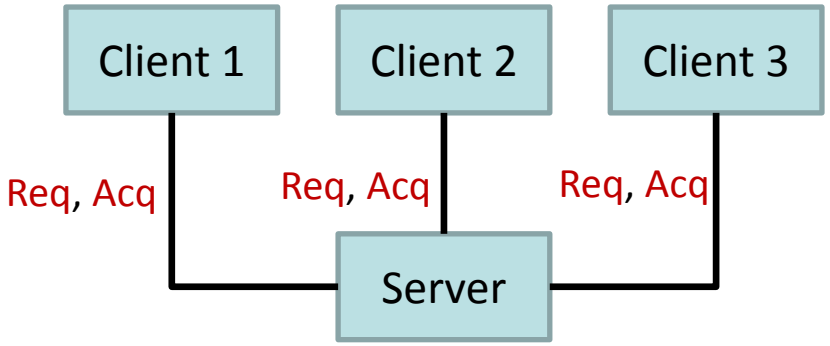
```
process Server [Req, Acq] : noexit :=
   Server_Aux [Req, Acq] (nil)
endproc

process Server_Aux [Req, Acq] (w : AgSet) : noexit :=
        Req (?a : AgId);
               Server_Aux [Req, Acq] (insert (a, w))
        []
        Acq (?a : AgId) [member (a, w)];
               Server_Aux [Req, Acq] (remove (a, w))
        endproc
```

Algèbre de processus

Exemple LOTOS : parallélisme

```
process Sys [Req, Acq] : noexit :=
   (
      Client [Req, Acq] (agent1)
      |||
      Client [Req, Acq] (agent2)
      |||
      Client [Req, Acq] (agent3)
   )
   |[Req, Acq]|
   Server [Req, Acq]
endproc
```

Algèbre de processus



Le langage LNT

- **Principe** : reprendre les (bonnes) idées de LOTOS

Mais

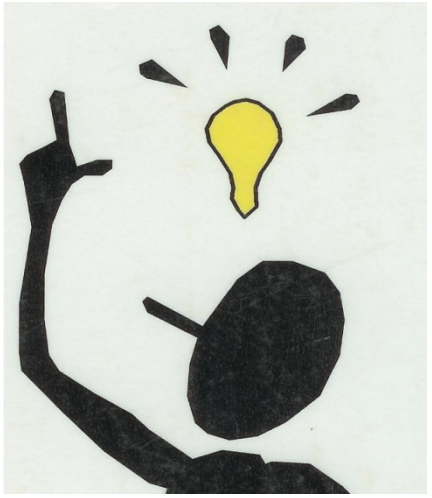
- En adoptant pour les concepts standards (types, fonctions, ...) une syntaxe et une sémantique **compréhensibles** et **usuelles**
- En intégrant de façon **homogène** les concepts qui n'ont pas d'équivalent dans les langages algorithmiques :
  - non-déterminisme
  - parallélisme
  - synchronisation, ...

Types LNT

- **Types prédéfinis** : booléens, entiers, caractères, chaînes
- **Types définis par l'utilisateur** :
  - Types définis par un ensemble de **constructeurs** avec **paramètres typés et nommés** (*types inductifs*)
  - **Cas particuliers** : types énumérés, enregistrements, unions, listes, arbres, etc.
  - **Notations abrégées** pour les tableaux, les listes et les ensembles (triés ou non)
  - **Sous-types** : intervalles et types à prédicats
  - Définition automatique de fonctions standard : "==", "<=", "<", ">=", ">" , sélecteurs de champs, etc.

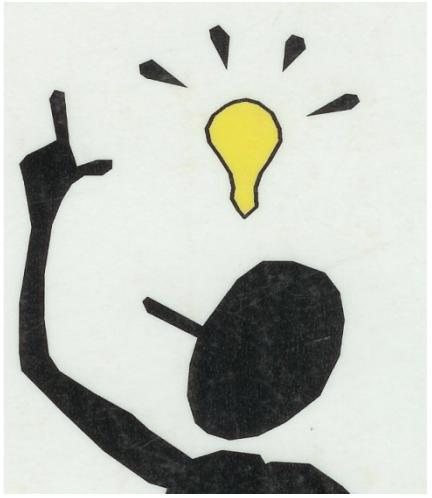
Exemple : type énuméré

```
type AgId is
    agent1, agent2, agent3
    with "==", "!=", "<", ">"
end type
```



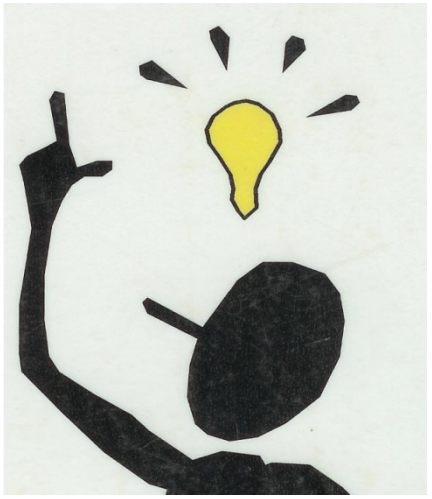
Exemple : type tableau

```
type AgArr is
  array [1..3] of AgId
end type
```



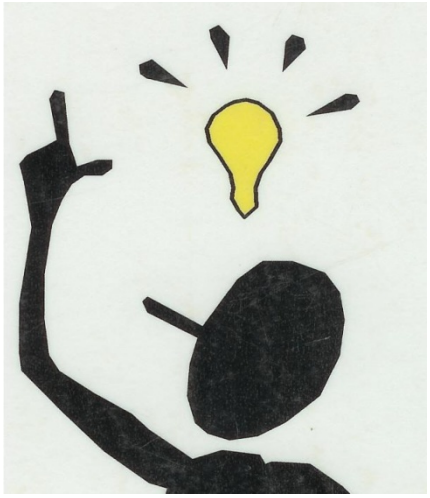
Exemple LNT : type ensemble

```
type AgSet is
  set of AgId
  with "remove", "member", "==", "!="
end type
```



Exemple : type arbre binaire

```
type Tree is
  leaf,
  node (left : Tree, right : Tree)
end type
```

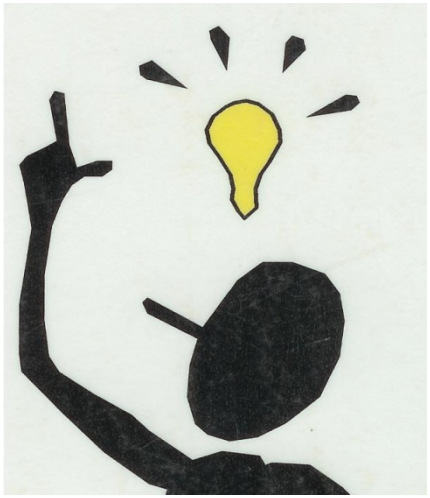


Fonctions LNT

- Fonctions **sans effets de bord** en syntaxe impérative
- **Exécution sûre** : typage fort, analyse d'initialisation
- Définition par des **instructions algorithmiques classiques**
  - Déclarations de variables locales : "**var**"
  - Affectations : ":="
  - Composition séquentielle : "**;**"
  - Boucles interuptibles : "**while**", "**for**"
  - Conditionnelles: "**if-then-else**"
  - Filtrage de motifs : "**case**"
  - Exceptions (non-rattrappables) : "**raise**"
- Trois modes de passage de paramètres :
  - "**in**" (appel par valeur)
  - "**out**" et "**inout**" (appel par référence)

Exemple : fonction

```
function Max_Ag (a : AgArr) : AgId is
   var
      result : AgId,
      n  : Nat
   in
      n := 1;
      result := agent1;
      while n <= 3 loop
         if a[n] > result then
            result := a[n]
         end if;
         n := n+1
      end loop;
      return result
   end var
end function
```



Processus LNT

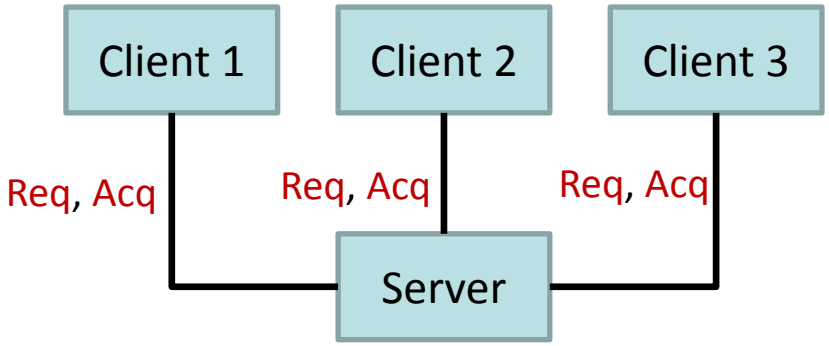
- Les processus sont un (quasi) **sur-ensemble des fonctions**
  - tous les opérateurs sauf return
- Opérateurs supplémentaires :
  - événements (rendez-vous avec passage de valeurs)
  - composition parallèle : "**par**"
  - masquage d'événements : "**hide**"
  - choix non-déterministe : "**select**"
  - "**disrupt**", affectation non-déterministe, etc.

Exemple : processus séquentiel

```
channel ClId is (AgId) end channel
process Server [Req, Acq : ClId] is
   var
      w : AgSet, a : AgId
   in
      w := nil;
      loop
         select
            Req (?a);
            w := insert (a, w)
         []
            Acq (?a) where member (a, w);
            w := remove (a, w)
         end select
      end loop
   end var
end process
```

Exemple : composition parallèle

```
process Sys [Req, Acq : ClId] is
   par Req, Acq in
      par
         Client [Req, Acq] (agent1)
      ||
         Client [Req, Acq] (agent2)
      ||
         Client [Req, Acq] (agent3)
      end par
   ||
      Server [Req, Acq]
   end par
end process
```



Conclusion

- LNT : un langage non-ambigü, riche, structuré, accessible pour modéliser les systèmes parallèles asynchrones
- Un héritage de 30 ans de recherche en théorie de la concurrence
- Intégré à CADP (http://cadp.inria.fr)
  - Simulation, test, prototypage rapide et vérification
  - Traduction vers LOTOS, dont les compilateurs sont développés et optimisés depuis 20 ans
- Des utilisateurs et des retours positifs
  - Etudiants (Ensimag, ...) et utilisateurs académiques
  - Utilisateurs industriels (financement initial)