Vérification avec Lustre/Lesar

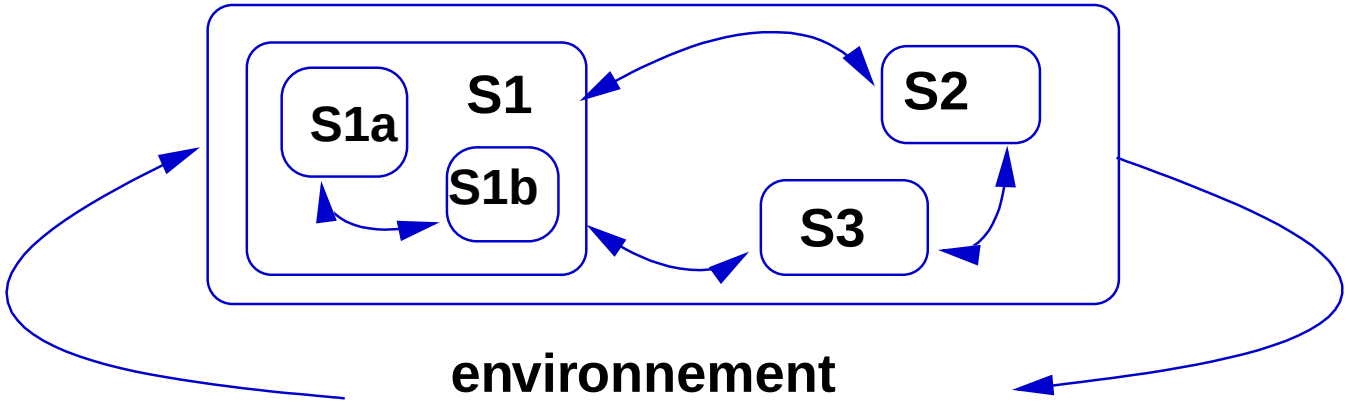
Pascal Raymond, VERIMAG

Plan

Approche synchrone 3
Le langage Lustre 7
Vérification de programme 11
Expression des propriétés 20
Algorithmique 27
Algorithmes énumératifs 32
Algorithmes symboliques 36
Conclusion et travaux en relation 43

Approche synchrone

Le domaine des systèmes réactifs



- **Domaine d'application : systèmes réactifs critiques (contrôle/commande, embarqué).**
- **Fonctionnement : séquence de réactions entrées/sorties.**
- **Conception parallèle, modulaire et hiérarchique.**

Approche classique

- **parallélisme de description $\longrightarrow$ parallélisme d'exécution,**
- **nécessité d'un d'exécutif complexe (OS temps-réel, primitive de communication/synchronisation),**
- **difficile de garantir une sémantique globale (indéterminisme).**
- **Vérification ?**

Approche synchrone

But : concilier conception hiérarchique et parallèle avec déterminisme.

- **Hypothèse de travail : on programme *comme si* communications et réactions se faisaient en *temps nul*.**
  On se concentre donc sur la *fonctionnalité* entrées/sorties.

- **Validité de l'hypothèse :**
  - ⋆ **la méthode garantit que les réactions se font en temps borné, évaluable pour une architecture donnée.**
  - ⋆ **l'hypothèse est valide si cette borne est inférieure à la dynamique de l'environnement.**

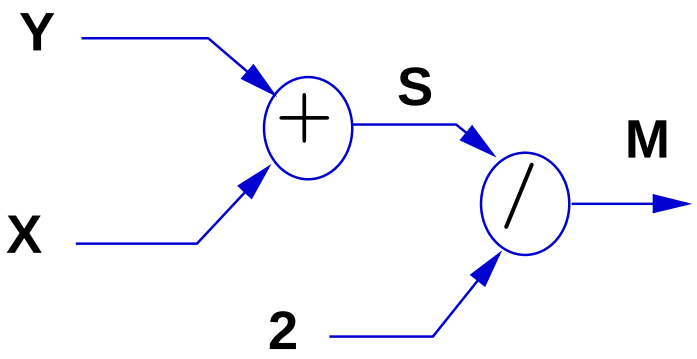
Langages synchrones

- **Un même principe général :**
  - ⋆ **Conception parallèle, hiérarchique.**
  - ⋆ **Compilation vers du code séquentiel simple (ordonnancement statique)**
  - ⋆ **(+ d'autres méthodes à ordonnancement maitrisé)**
- **Plusieurs styles :**
  - ⋆ **Lustre : style flot de données, transféré dans l'atelier SCADE.**
  - ⋆ **Signal : flots de données + structure d'horloges.**
  - ⋆ **Esterel : impératif à structures de contrôle "classiques". Version graphique à base d'automates hiérarchiques communiquants (SynchCharts).**

Le langage Lustre

Principes

- Modèle classique des graphes flot-de-données.




```
node Moyenne(X,Y:int)
returns(M:int);
var S:int;
let
   M = S/2;
   S = (X+Y);
tel
```

- Déclaratif : programme $=$ ensemble d'équations.
- Synchrone : les "fils" sont des fonctions d'une horloge discrète globale, assimilée à $\mathbb{N}$: $\forall t \in \mathbb{N} \quad M_t = (X_t + Y_t)/2$

Principes (suite)

- **Opérateurs temporels :**
  - ⋆ **Mémoire :** $\forall t \geq 1 \;\; (\texttt{pre X})_t = \texttt{X}_{t-1}$, **et** $(\texttt{pre X})_0 = \bot$.
  - ⋆ **Initialisation :** $(\texttt{X -> Y})_0 = \texttt{X}_0$ **et** $\forall t \geq 1 \;\; (\texttt{X -> Y})_t = \texttt{Y}_t$.
  - ⋆ **Exemple :** `N = 0 -> pre N + 1`
- **Modularité : tout programme (node) est réutilisable dans un autre programme.**
- **Langage dédié :**
  - ⋆ **Conçu pour la programmation des noyaux réactifs.**
  - ⋆ **Volontairement limité pour garantir un WCET (pas de récursion, pas d'allocation dynamique).**
  - ⋆ **Pas (peu) de support pour la définition/manipulation de données complexes ($\neq$ *general purpose language*).**

Exemple du compteur de balises

- **Un train circule sur une voie :**
  - ⋆ **il croise des balises, disposées de proche en proche sur la voie,**
  - ⋆ **il reçoit une signal seconde.**
  - ⋆ **vitesse idéale : 1 balise/seconde.**
- **Compteur de balises embarqué :**
  - ⋆ **décide si on est *à l'heure*, *en avance* ou *en retard*,**
  - ⋆ **utilise une hystérésis pour éviter les oscilations d'état (décalage des conditions de changement d'état).**

```
node compteur(sec,bal:  bool)
returns (alheure,retard,avance:  bool);
var diff :  int;
let
   diff = (0 -> pre diff) +
          (if bal then 1 else 0) +
          (if sec then -1 else 0);
   avance = (true -> pre alheure) and (diff > 3)
          or (false -> pre avance) and (diff > 1);
   retard = (true -> pre alheure) and (diff < -3)
          or (false -> pre retard) and (diff < -1);
   alheure = not (avance or retard);
tel
```

Vérification de programme

- **Les méthodes s'appliquent à tout langage/formalisme basé sur une notion de temps discret.**
- **$\Rightarrow$ Lustre et les autres langages synchrones, et bien d'autres.**

Notion de propriété temporelle

- **On s'intéresse aux propriétés *fonctionelles*.**
- **Le programme caclcule-t-il les bonnes sorties en fonction de ses entrées ?**
- **Une propriété $=$ une relation entre les séquences d'entrées et de sorties.**
- **On parle de *propriété temporelle*.**

Sûreté et vivacité

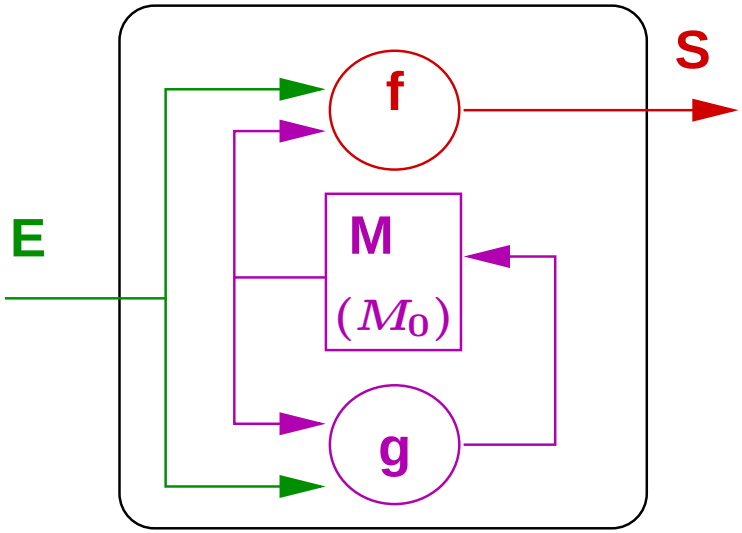
- **Il existe toute une théorie sur les propriétés (et les logiques) temporelles.**
- **On retiendra simplement une définition “intuitive” de la partition classique entre :**
  - ⋆ **sûreté (safety) : quelque chose (de mauvais) n’arrive jamais,**
  - ⋆ **vivacité (liveness) : quelque chose (de bon) peut ou doit arriver.**
- **Distinction importante car :**
  - ⋆ **Les safety sont de simples invariants, elles permettent de raisonner sur des séquences *arbitrairement longues*.**
  - ⋆ **Les liveness font référence à un futur non borné, elles nécessitent de raisonner sur des séquences *infinies*.**

Exemple du compteur de balises.

- **Quelques propriétés qu'on peut attendre de ce programme :**
  - ⋆ **On ne peut pas être en avance et en retard.**
  - ⋆ **On ne peut pas passer directement d'en retard à en avance (et réciproquement).**
  - ⋆ **On ne peut pas rester en retard pendant un seul instant.**
  - ⋆ **Si le train stoppe, il finira par être en retard.**
- **Les trois premières propriétés sont des *safety*, tandis que la dernière est clairement une *liveness*.**

Machine à mémoire

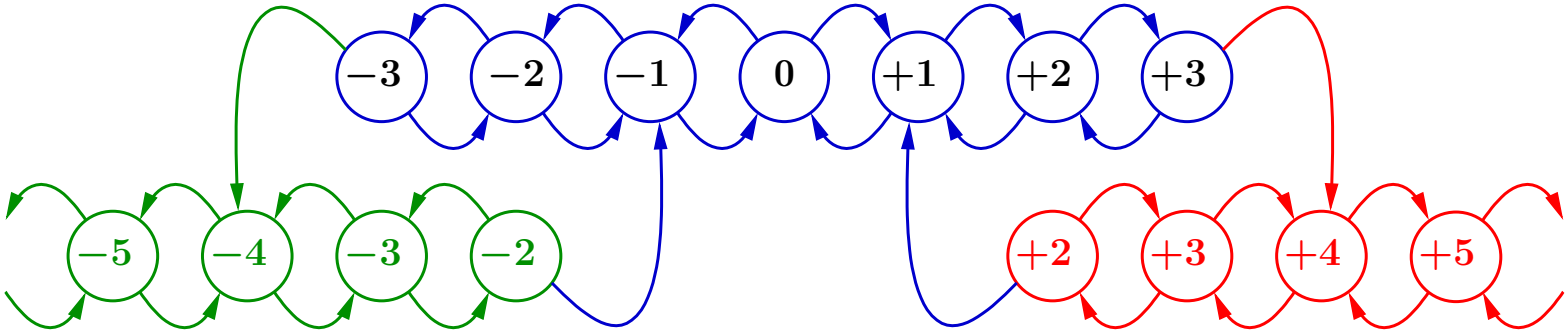
Une programme Lustre (plus généralement un programme synchrone) est une machine à mémoire :



- **l'état initial de M est parfaitement déterminé (M_0),**
- **sorties et mémoire suivante sont des fonctions (f, g) des entrées et de la mémoire courante.**

Automate explicite

- **Une machine à mémoire = un système de transition (automate) :**
  - ⋆ **un état = une configuration particulière de la mémoire,**
  - ⋆ **une transition = changement d'état + production des sorties en fonction des entrées.**
- **Compteur de balise :**



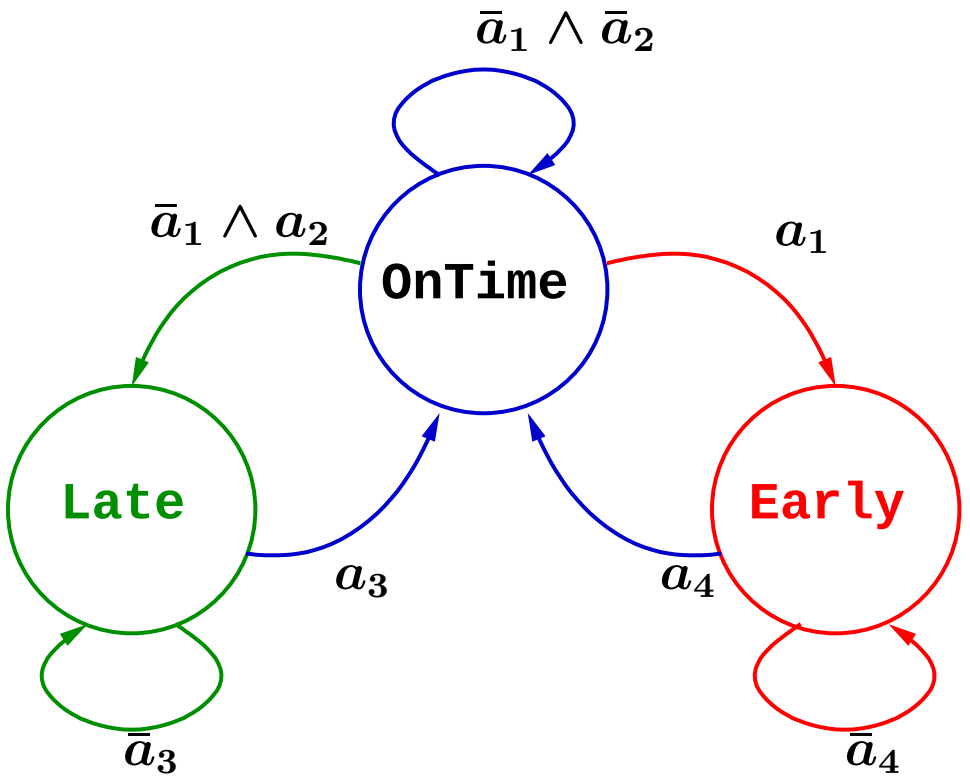
Principe du model-checking

- **L'automate explicite est un *modèle* des comportements possibles du programme.**
- **Explorer l'automate $=$ étudier les propriétés du programme.**
- **Problème : l'automate est généralement infini (ou énorme) $\Rightarrow$ impossible de l'explorer.**
- **Idée : travailler sur une *abstraction finie* (et pas trop grosse) de l'automate.**
- **Cette abstraction doit *conserver* certaines propriétés du programme.**

Exemple d'abstraction

- **L'abstraction booléenne ignore tout ce qui n'est pas strictement booléen.**
- **Exemple : ignorer `diff`, et remplacer les comparaisons entière par des *entrées* :**

  - ⋆ `diff > 3` $\longrightarrow a_1$
  - ⋆ `diff < -3` $\longrightarrow a_2$
  - ⋆ `diff < -1` $\longrightarrow a_3$
  - ⋆ `diff > 1` $\longrightarrow a_4$



- **L'abstraction reprisente une *sur-approximation* des comportements.**
- **Des propriétés sont conservées :**
  - ⋆ **On ne peut pas être en avance et en retard (sûreté).**
  - ⋆ **On ne peut pas passer directement d'en retard à en avance (sûreté).**
- **D'autres sont perdues :**
  - ⋆ **Si le train stoppe, il finira par être en retard (vivacité).**
  - ⋆ **On ne peut pas rester en retard pendant un seul instant (sûreté).**
- **Attention : la négation d'un propriété perdue est vraie sur l'abstraction mais pas sur le programme !**

Abstraction conservative et propriétés de sûreté

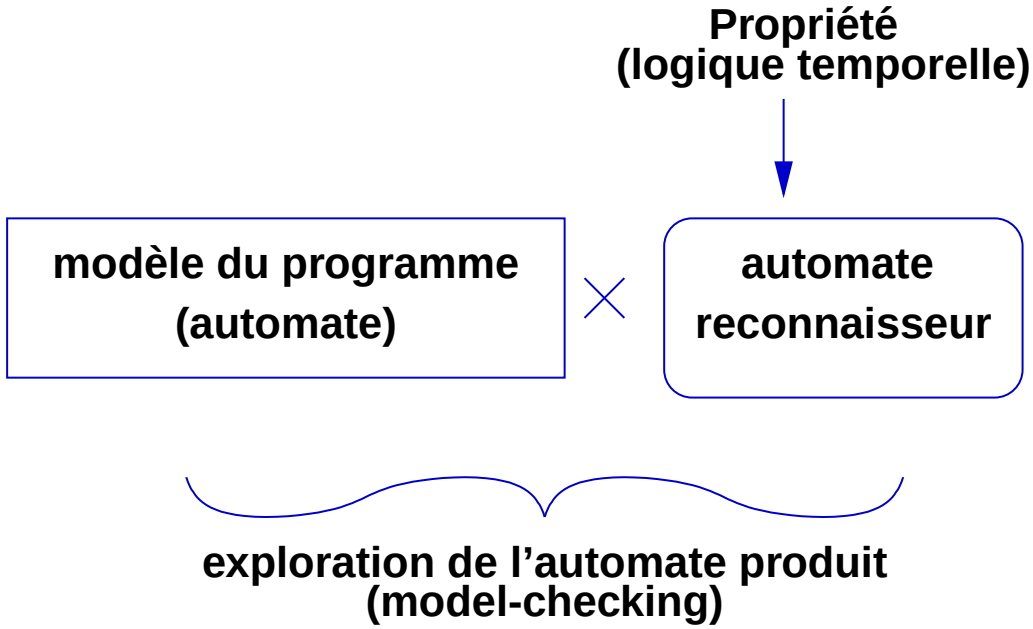
- **Abstraction booléenne = cas particulier de sur-approximation.**
- **Les sur-approximations conservent la classe des *safaty*,**
- **mais pas celle des *liveness*.**

En bref :

- **on peut faire de la preuve partielle de safety :**
  - ⋆ **la vérification aboutit, et la propriété est donc satisfaite,**
  - ⋆ **le vérification échoue et on ne peut (en général) pas conclure.**
- **on ne peut (généralement) rien faire sur les liveness.**

Expression des propriétés

Schéma classique du model-checking



- **Model-checking : vaste domaine !**
- **On va adapter à notre cas particulier.**

Adaptation aux programmes synchrones

- **Modèle du programme : le programme lui-même (sémantique formellement définie).**
- **Abstraction : automatique (réécriture du programme).**
- **Expression des propriétés :**
  - ⋆ **les logiques temporelles ne sont vraiment nécessaires que pour les liveness**
  - ⋆ **les safety (invariant) peuvent être *programmées* (observateurs).**
- **N.B. on peut souvent renforcer les liveness en safety :**
  - ⋆ **“le train s’arrète inévitablement” (liveness),**
  - ⋆ **“le train s’arrète inévitablement dans les 20 secondes” (safety),**

Observateurs

- **Observateur = un programme qui scrute les entrées/sorties et qui renvoie `ok` aussi longtemps que la propriété attendue est satisfaite.**
- **Peut être écrit dans le langage habituel de l'ingénieur.**
- **N.B. Programmer des observateur = ce qu'on fait avec le `assert` des langages C,CAML, etc.**

Exemples

Quelques propriétés en Lustre :

- On ne peut pas être en avance et en retard :

```
ok = not (avance and retard);
```

- On ne peut pas passer directement d'en retard à en avance :

```
ok = true -> not ((pre retard) and avance);
```

- On ne peut pas rester en retard pendant un seul instant :

```
ok = ((not PPretard) and Pretard) => retard;
```

avec :

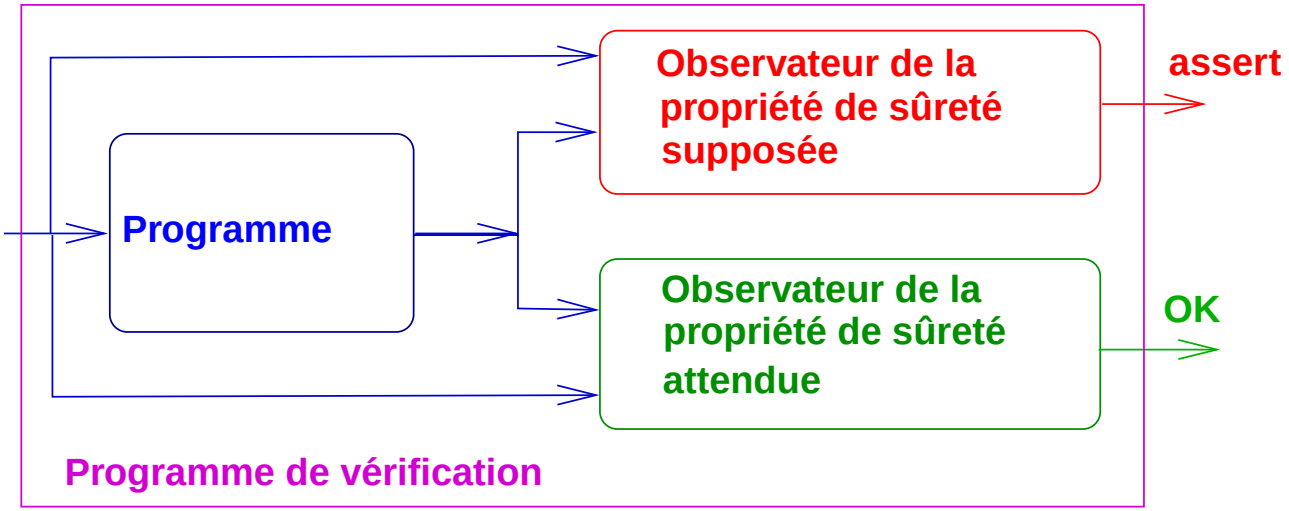
```
Pretard = false -> pre retard;
PPretard = false -> pre Pretard;
```

Hypothèses

- **En général, un programme n'est correct que dans un environnement particulier.**
- **Exemple classique :**
  - ⋆ **un programme contrôle une section critique avec des feux et des aiguillages,**
  - ⋆ **propriété attendue : pas de collision.**
  - ⋆ **MAIS : si les trains ne respectent pas les feux, aucune chance que ce soit vrai !**
  - ⋆ **D'où la "vraie" propriété :**
    "Si les trains respectent les feux, alors le programme garantit qu'il n'y aurra pas de collision"

Model-checking des programmes synchrones

- **Restriction : propriétés et hypothèses sont des invariants (donc programmable en observateurs).**
- **L'utilisateur fournit un *programme de vérification*, intégrant :**
  - ⋆ **le programme à valider proprement dit,**
  - ⋆ **l'observateur de la propriété attendue,**
  - ⋆ **l'observateur des hypothèses (*assert* en Lustre).**



- **Rôle du model-checker :**
  - ⋆ **extraire une abstraction finie (typiquement booléenne),**
  - ⋆ **explorer l'abstraction finie pour établir/réfuter : quelque soit une séquence d'entrée, `OK` reste vraie aussi longtemps que `assert` reste vraie**

- **En terme de logique temporelle :**
  P1 : ***(toujours `assert`)*** $\Rightarrow$ ***(toujours `OK`)***

N.B. P1 n'est pas une safety, mais :

- **elle fait partie d'un classe conservée par la sur-approximation,**
- **en pratique, elle est plus coûteuse à vérifier qu'un simple invariant,**
- **pour l'exposé, on sur-approxime un peu plus : P2 :**
  toujours(`assert` n'a jamais été faux $\Rightarrow$ ***`OK`)***

Algorithmique

Automate booléen

Un programme de vérification est caractérisé par :

- **ses variables libres (entrées)** V
- **ses variables d'états (mémoires)** S**,**
- **l'état initial** $Init$ **:** $\mathsf{B}^{|S|} \longrightarrow \mathsf{B}$
  (avec l'abstration on peut avoir plusieurs états initiaux),
- **ses fonctions de transitions** g_k **:** $\mathsf{B}^{|S|} \times \mathsf{B}^{|V|} \longrightarrow \mathsf{B}$**,**
  pour chaque $k = 1 \cdots |S|$**,**
- **l'hypothèse** H **:** $\mathsf{B}^{|S|} \times \mathsf{B}^{|V|} \longrightarrow \mathsf{B}$**,**
- **la propriété** Φ **:** $\mathsf{B}^{|S|} \times \mathsf{B}^{|V|} \longrightarrow \mathsf{B}$**.**

Automate explicite associé

L'automate booléen "code" un système à états/transitions explicite :

- $Q = \mathrm{B}^{|S|}$ **est l'espace d'états**
  (l'ensemble des valeurs potentielles de la mémoire),
- $Init \subseteq Q$ **est l'ensemble des états initiaux**
  (fonctions caractéristiques $\equiv$ ensembles),
- $R \subseteq Q \times \mathrm{B}^{|V|} \times Q$ **est la relation de transition :**

$$(q, v, q') \in R \;\Leftrightarrow\; q'_k = g_k(q, v) \quad k = 1 \cdots |S|$$

  on note $q \xrightarrow{v} q'$ pour $(q, v, q') \in R$.

- $H \subseteq T$ **est l'ensemble des transitions qui satisfont l'hypothèse,**
- $\Phi \subseteq T$ **est l'ensemble des transitions qui satisfont la propriété.**

Note sur le déterminisme

- **Dans notre cas, R est une fonction.**
- **Puisque le q' de $q \xrightarrow{v} q'$ est parfaitement déterminé, on appelera *transitions* les couples de $T = Q \times \mathrm{B}^{|V|}$.**

Fonctions *pre* et *post*

Pour tout état $q \in Q$, et tout ensemble d'états $X \subseteq Q$:

- $\mathit{post}_H(q) = \{q' \mid \exists v\ q \xrightarrow{v} q' \wedge H(q, v)\}$,
- $\mathit{POST}_H(X) = \bigcup_{q \in X} \mathit{post}_H(q)$
- $\mathit{pre}_H(q) = \{q' \mid \exists v\ q' \xrightarrow{v} q \wedge H(q', v)\}$,
- $\mathit{PRE}_H(X) = \bigcup_{q \in X} \mathit{pre}_H(q)$

Les états remarquables

- **Les états initiaux Init, et les état accessibles :**

$$\textbf{Acc} \quad = \quad \mu X \ \cdot \ (X = \textbf{Init} \cup \mathit{POST}_H(X))$$

on note $\textbf{Acc}_0 = \textbf{Init}$

- **Les états d'erreur (immédiate) :**

$$\textbf{Err} \quad = \quad \{q \mid \exists v \ H(q,v) \wedge \neg \Phi(q,v)\}$$

et plus généralement les "mauvais"états :

$$\textbf{Bad} \quad = \quad \mu X \ \cdot \ (X = \textbf{Err} \cup \mathit{PRE}_H(X))$$

on note $\textbf{Bad}_0 = \textbf{Err}$**.**

Principe de l'exploration

- **Le but est d'établir que Acc $\cap$ Bad $= \emptyset$, ou plus simplement :**
  - ⋆ **Acc $\cap$ Bad$_0$ $= \emptyset$ (méthode *en avant*)**
  - ⋆ **Bad $\cap$ Acc$_0$ $= \emptyset$, (méthode *en arrière*)**
- **Dans notre cas, il y a une dissymétrie :**
  - ⋆ **en avant (déterministe),**
  - ⋆ **en arrière (indéterministe)**

Algorithmes énumératifs

En avant

CurAcc := Init

Done := empty

while it exits q in CurAcc - Done do {

(* q ∈ CurAcc \ Done *)

for all q' in *post*(**q**) **do** {

if q' in Bad0 then EXIT(failed)

put q' in CurAcc

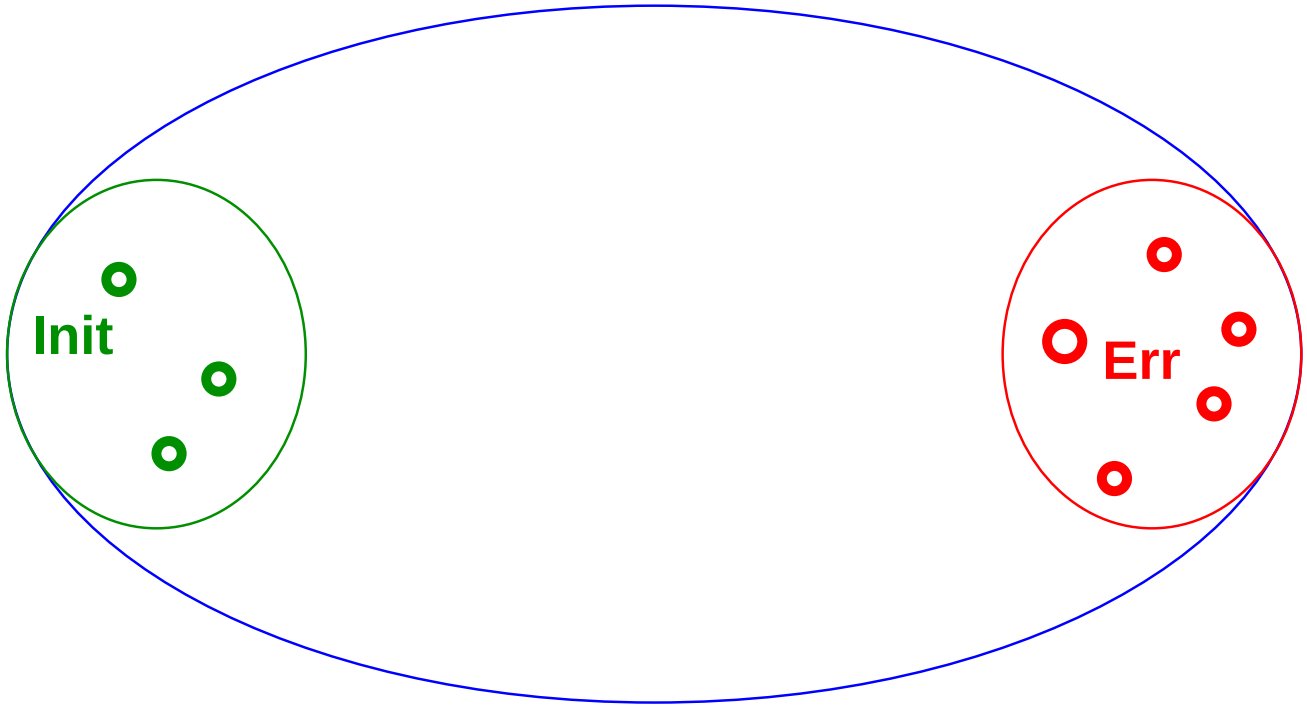
}

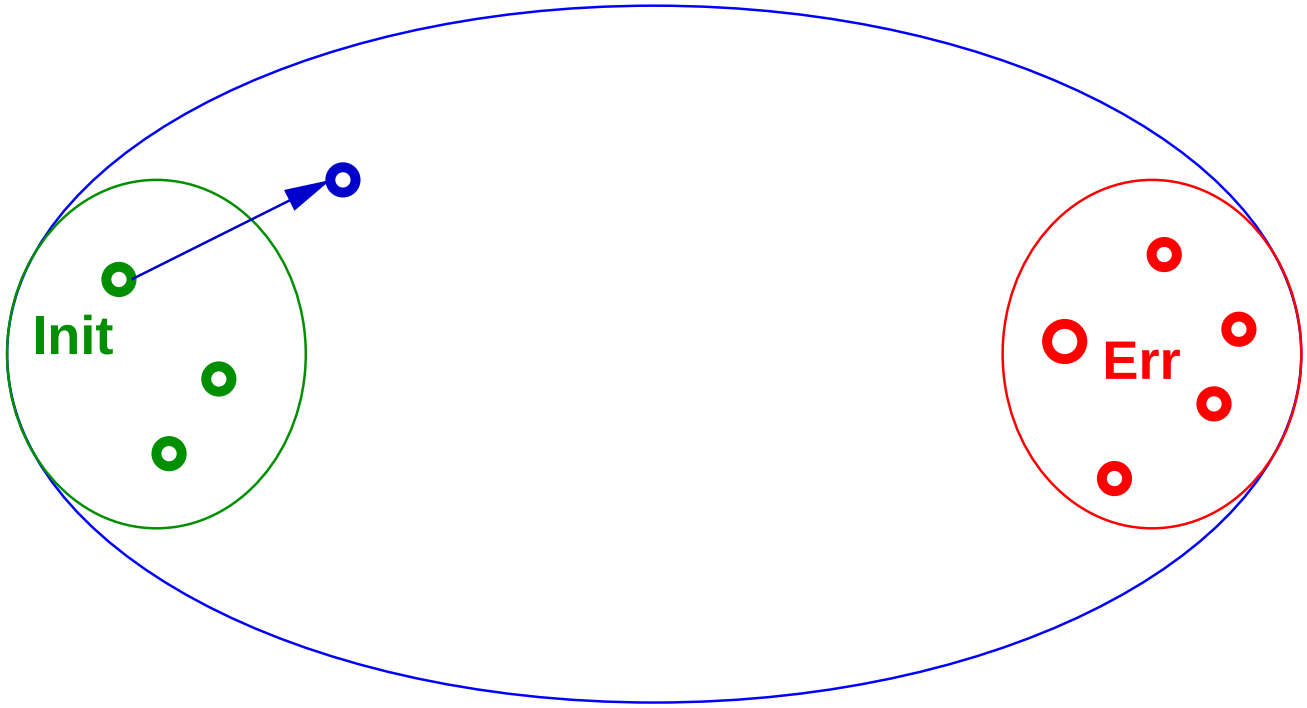
put q in Done

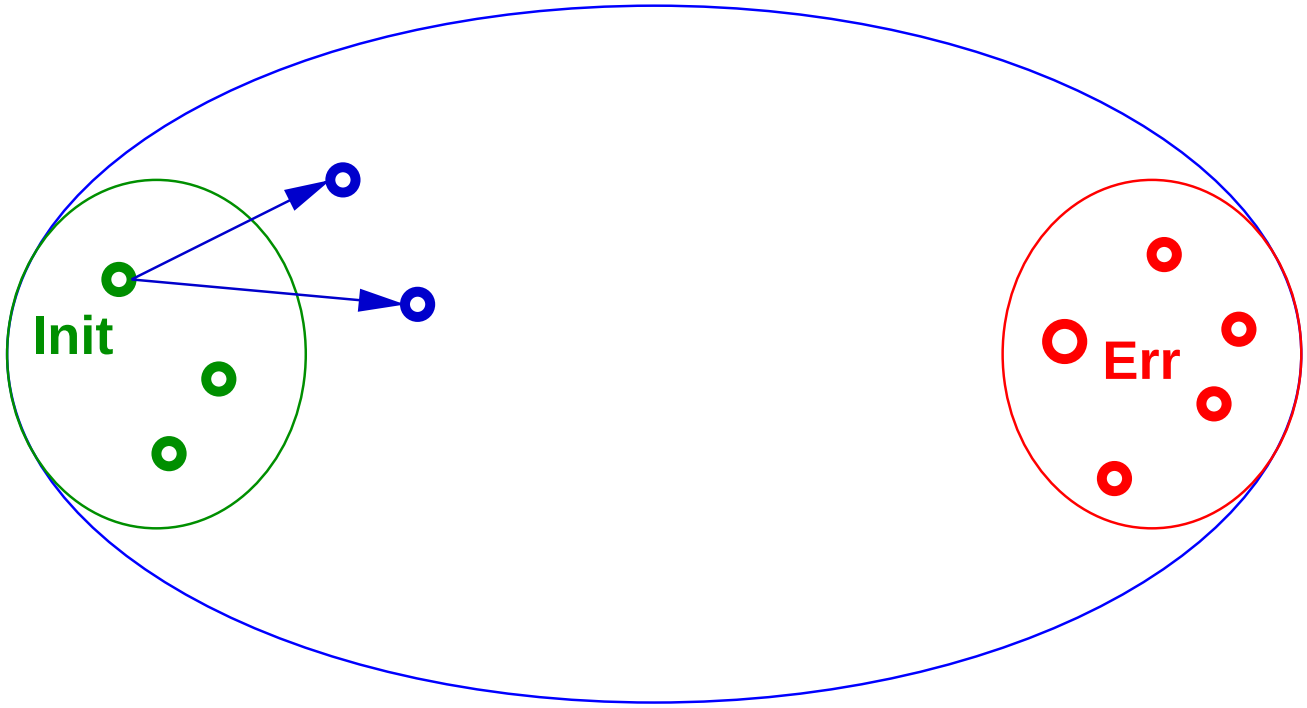
}

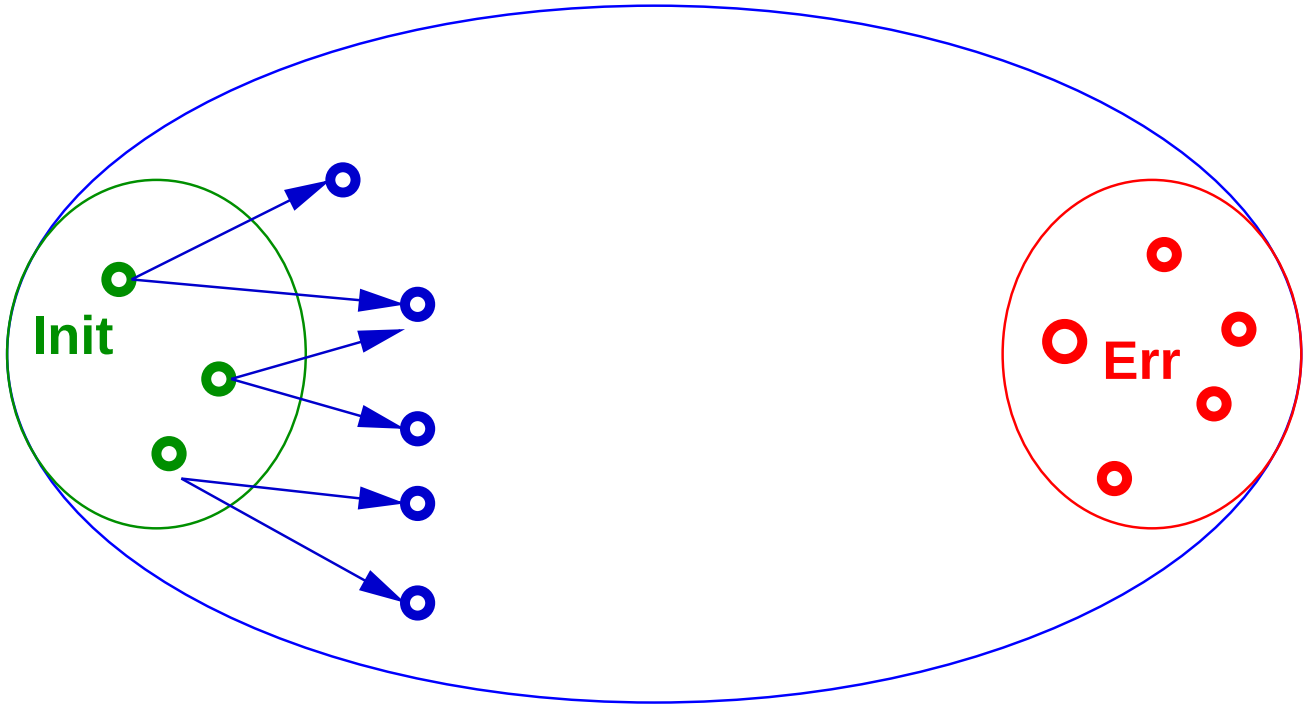
(* we have CurAcc = Done = Acc *)

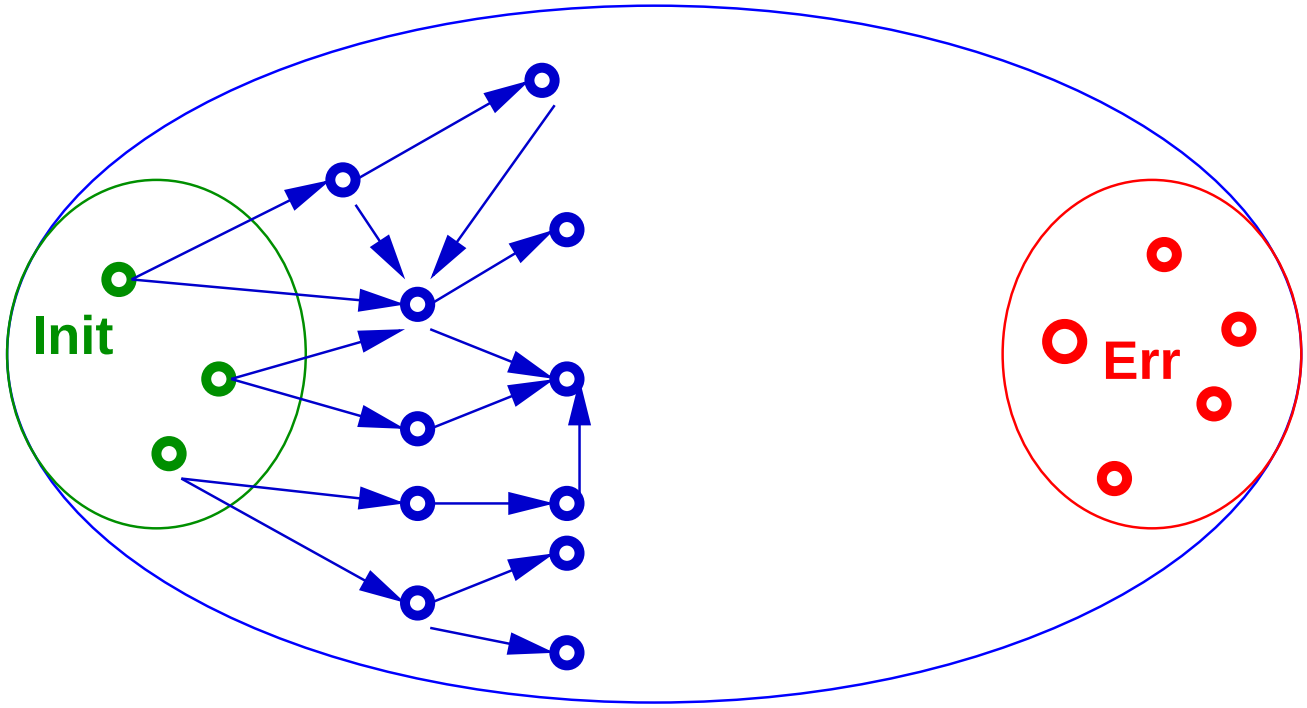
EXIT(succeed)

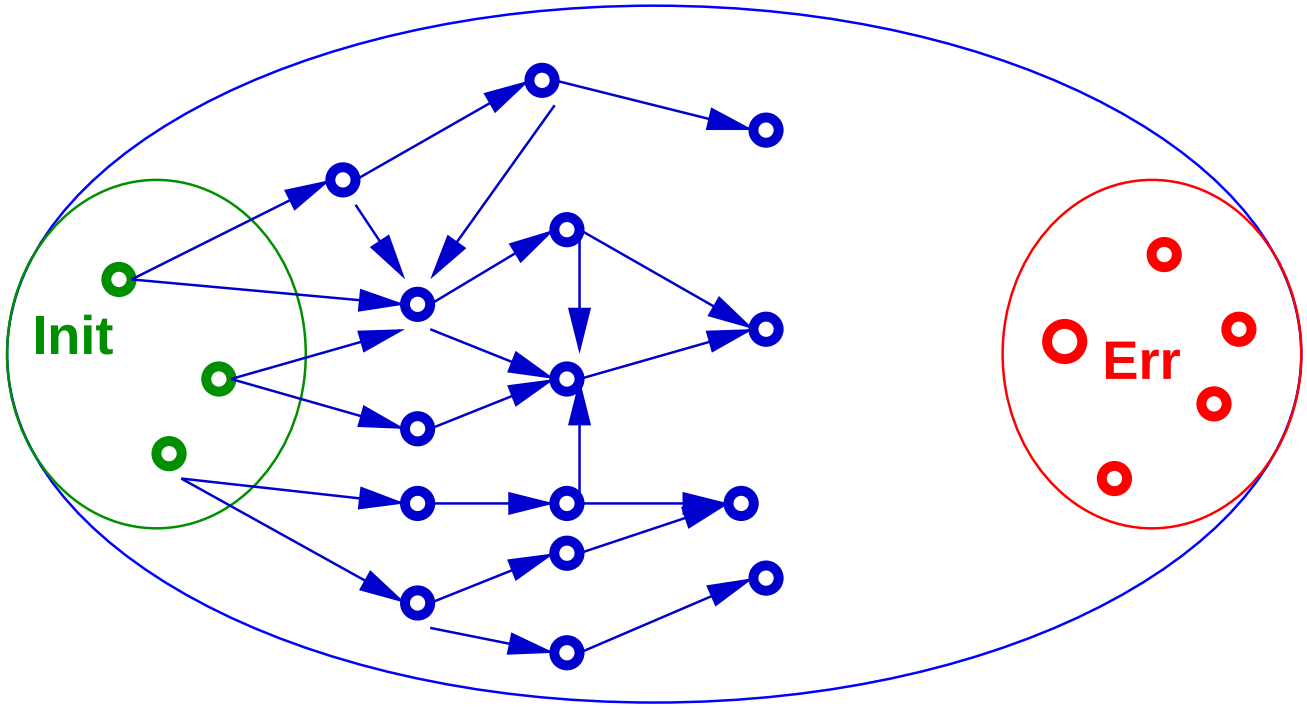

Init
Err

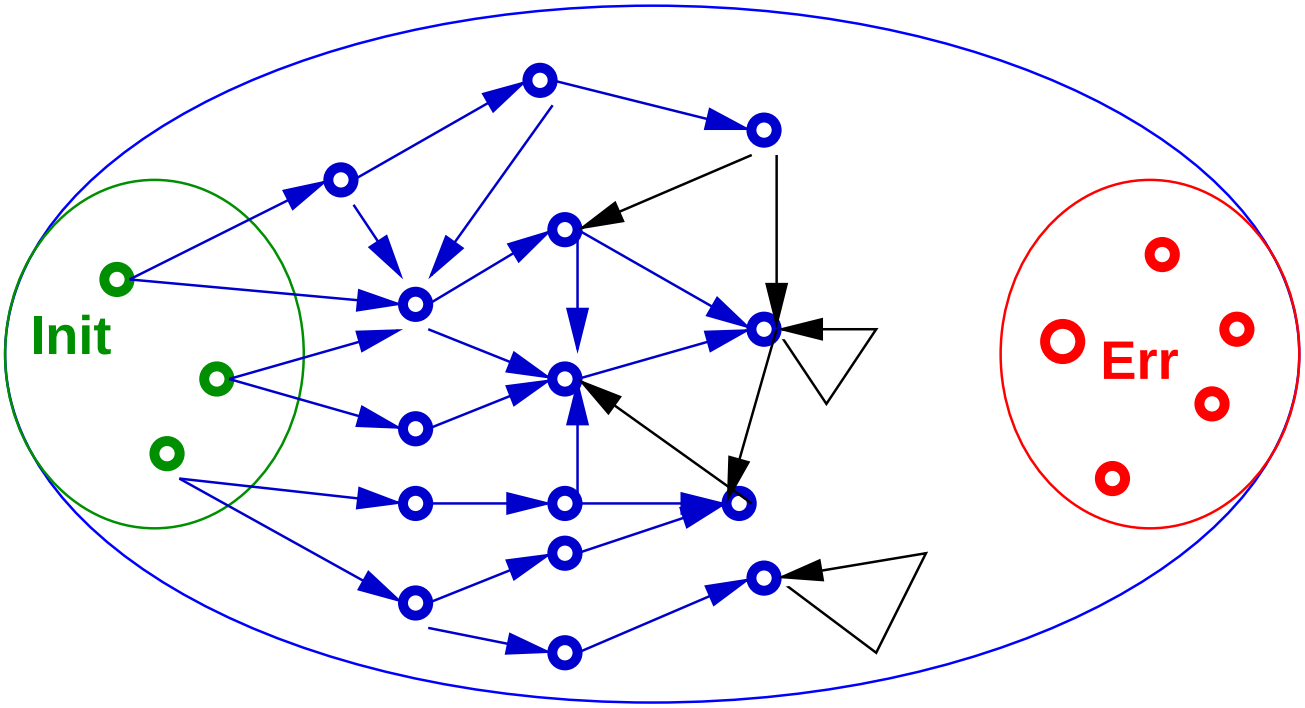

Init
Err


Init
Err

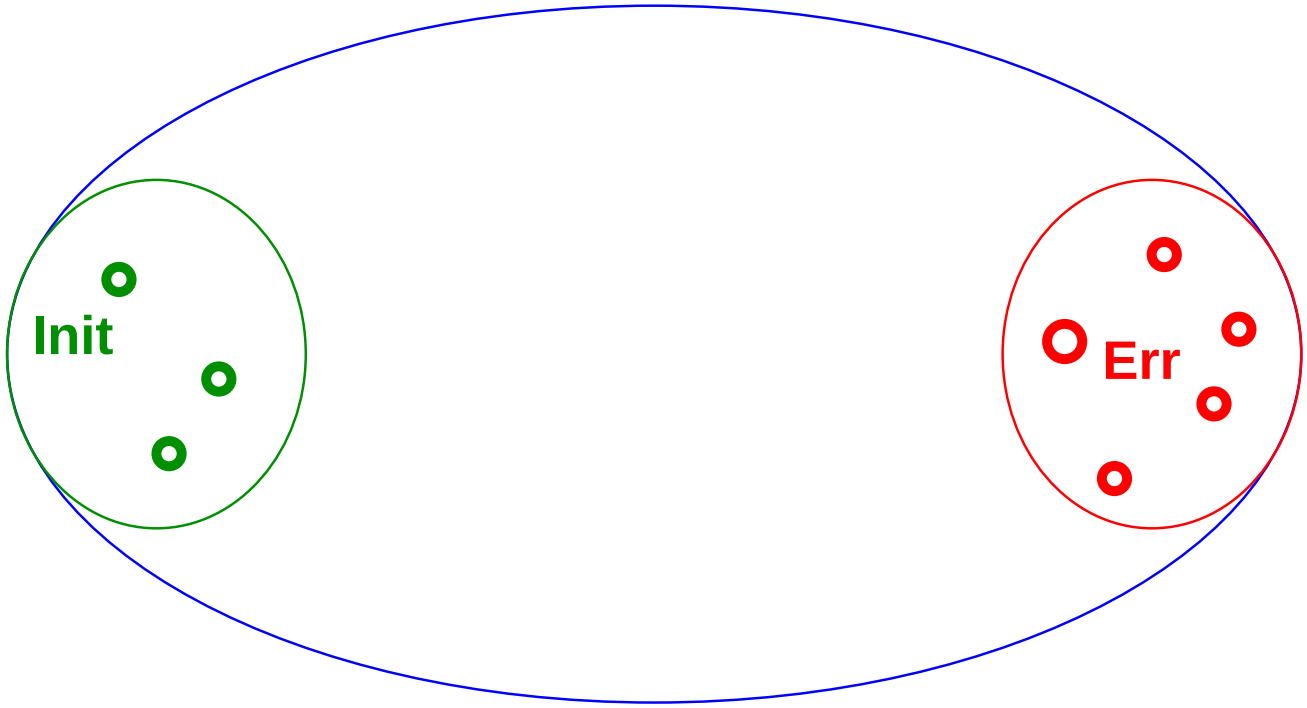

Init
Err

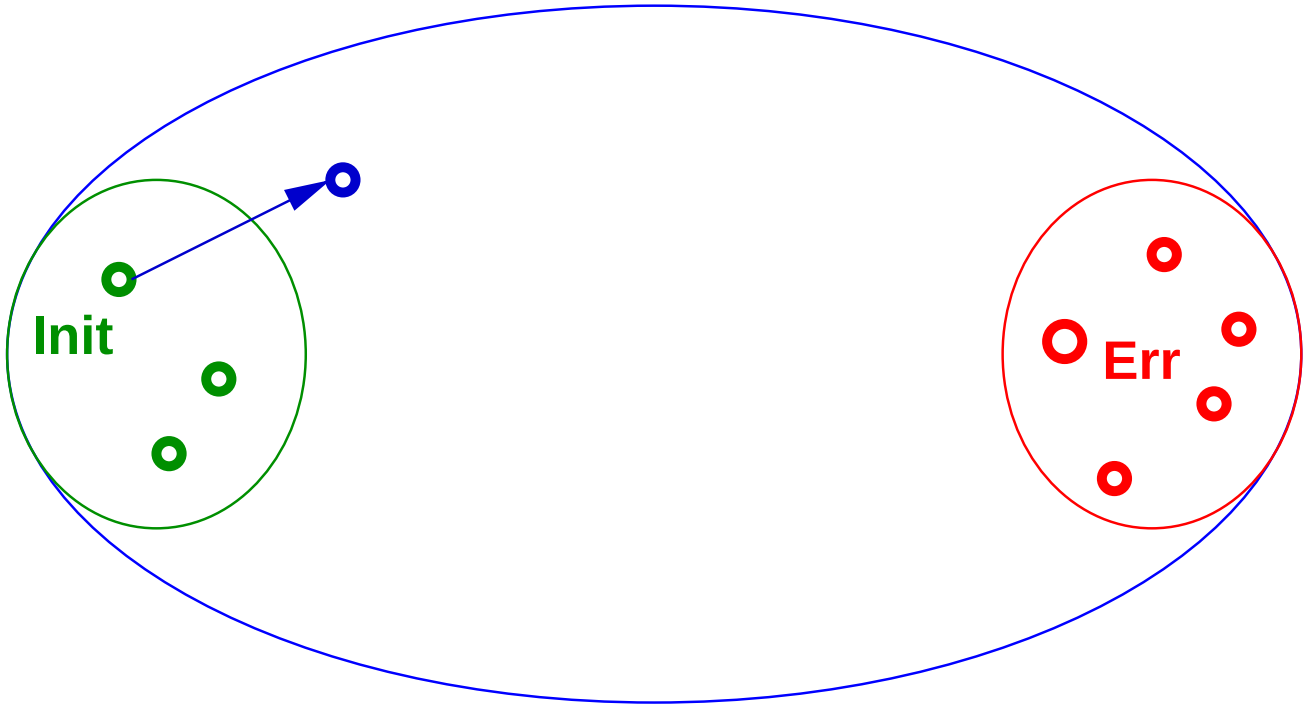

Init
Err

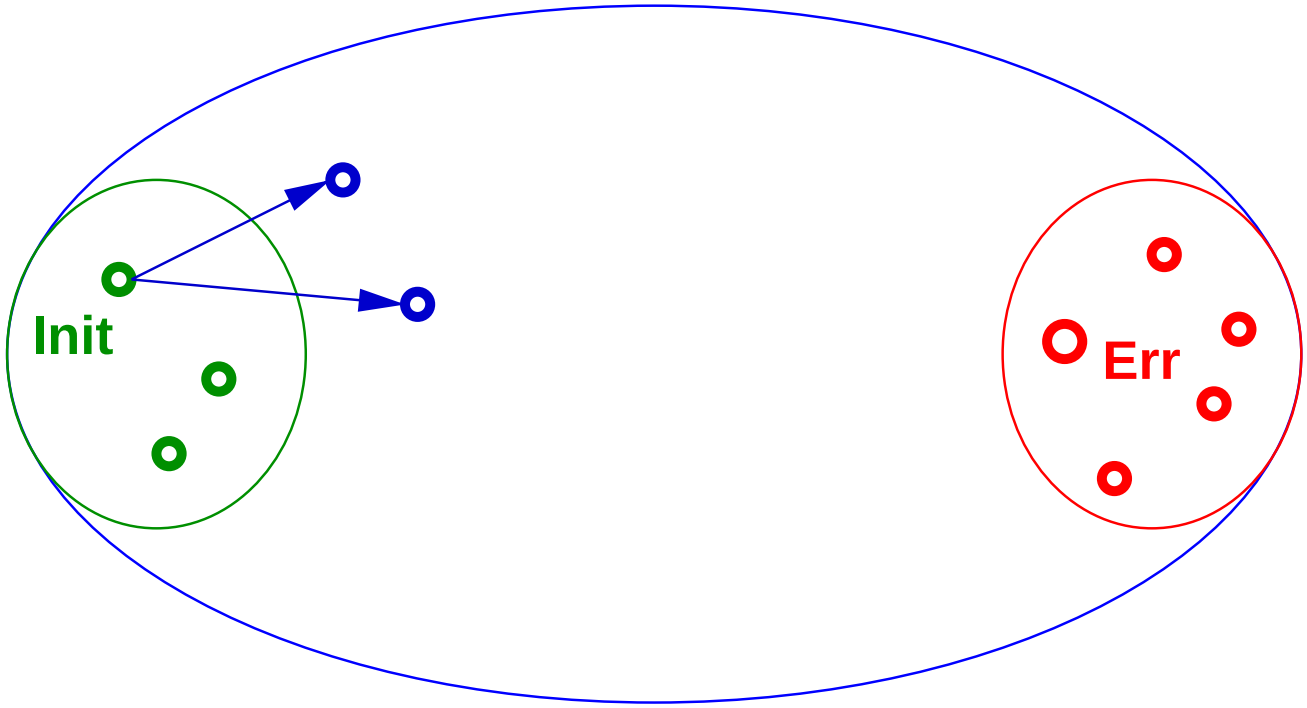

Init
Err

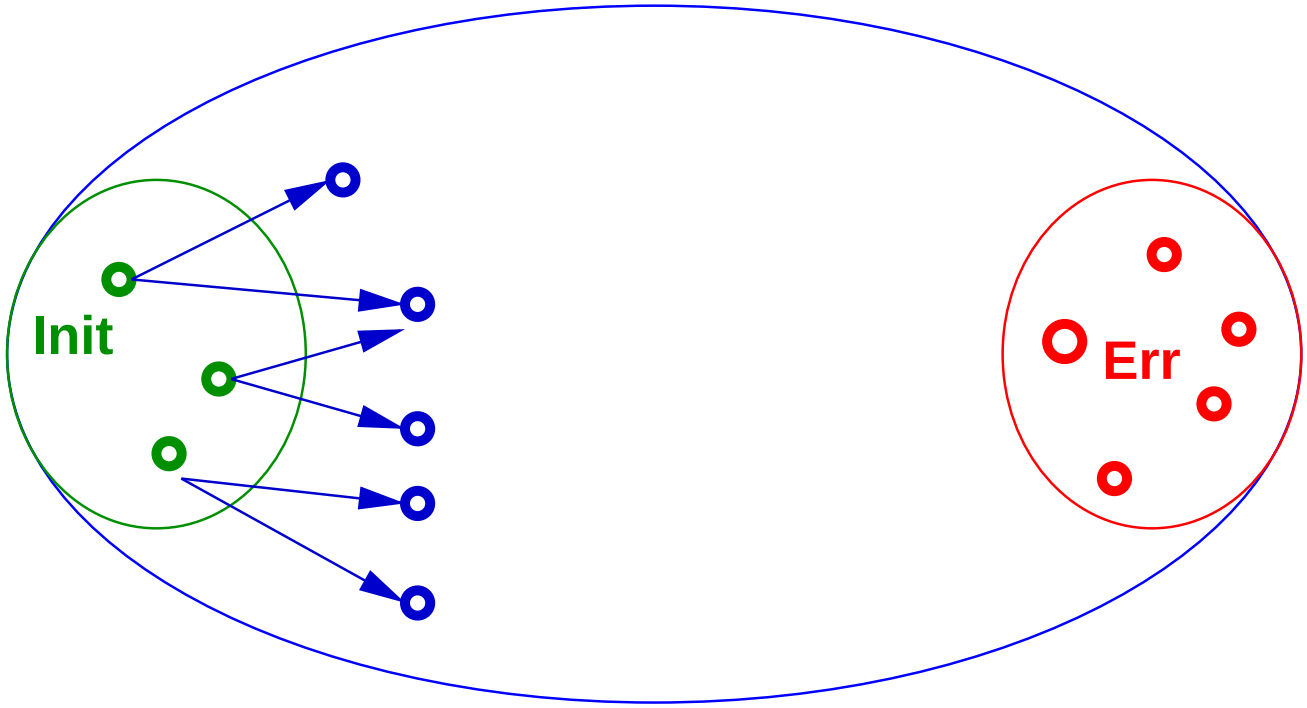


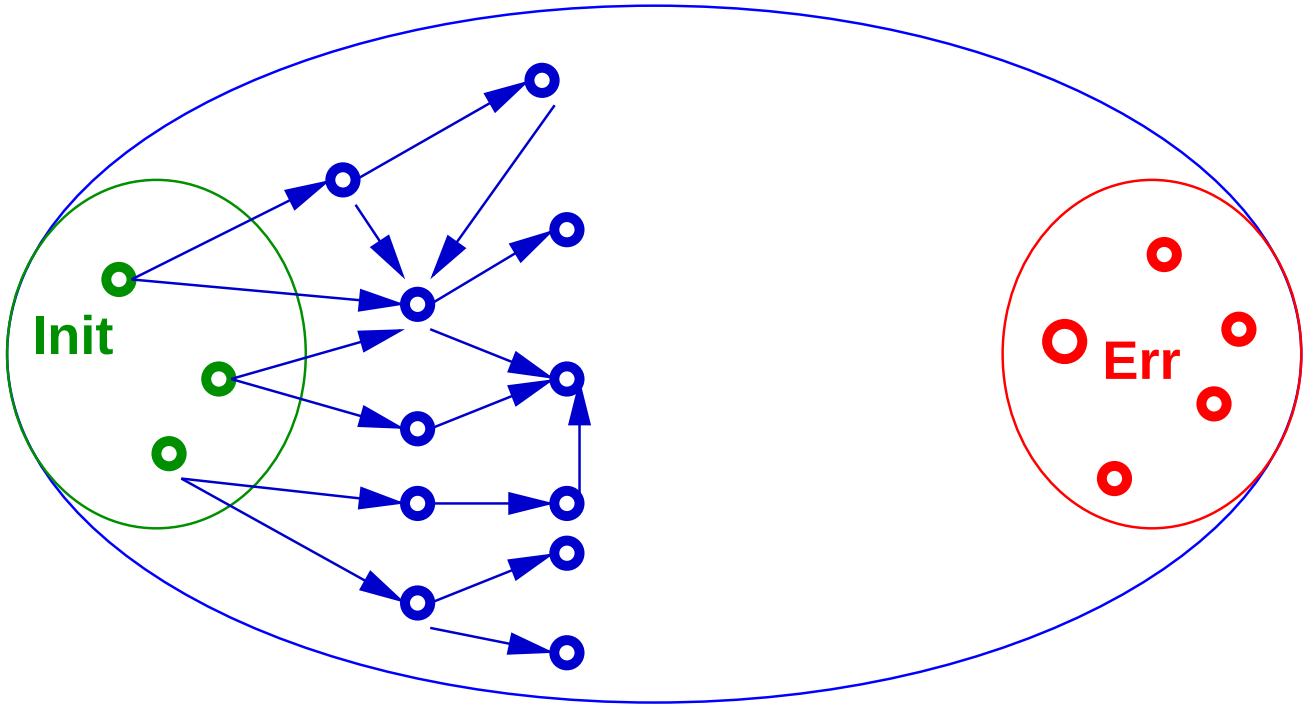
La preuve réussit

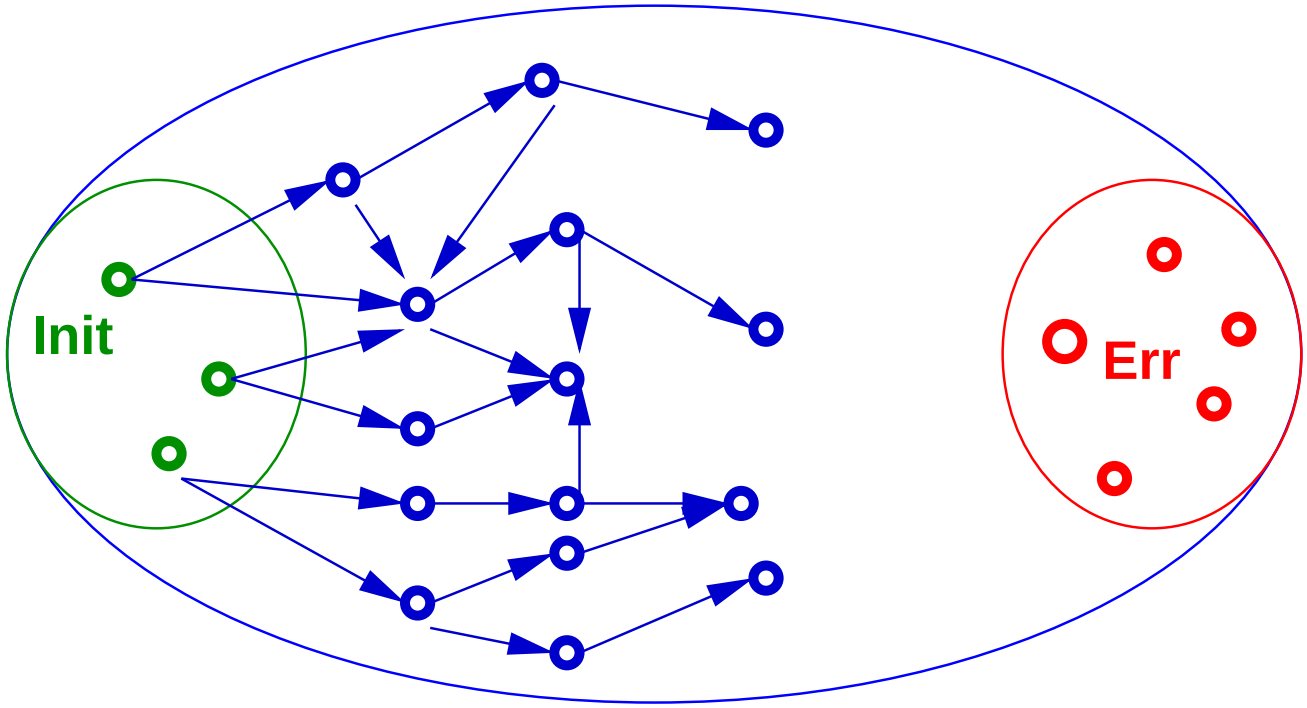

Init
Err

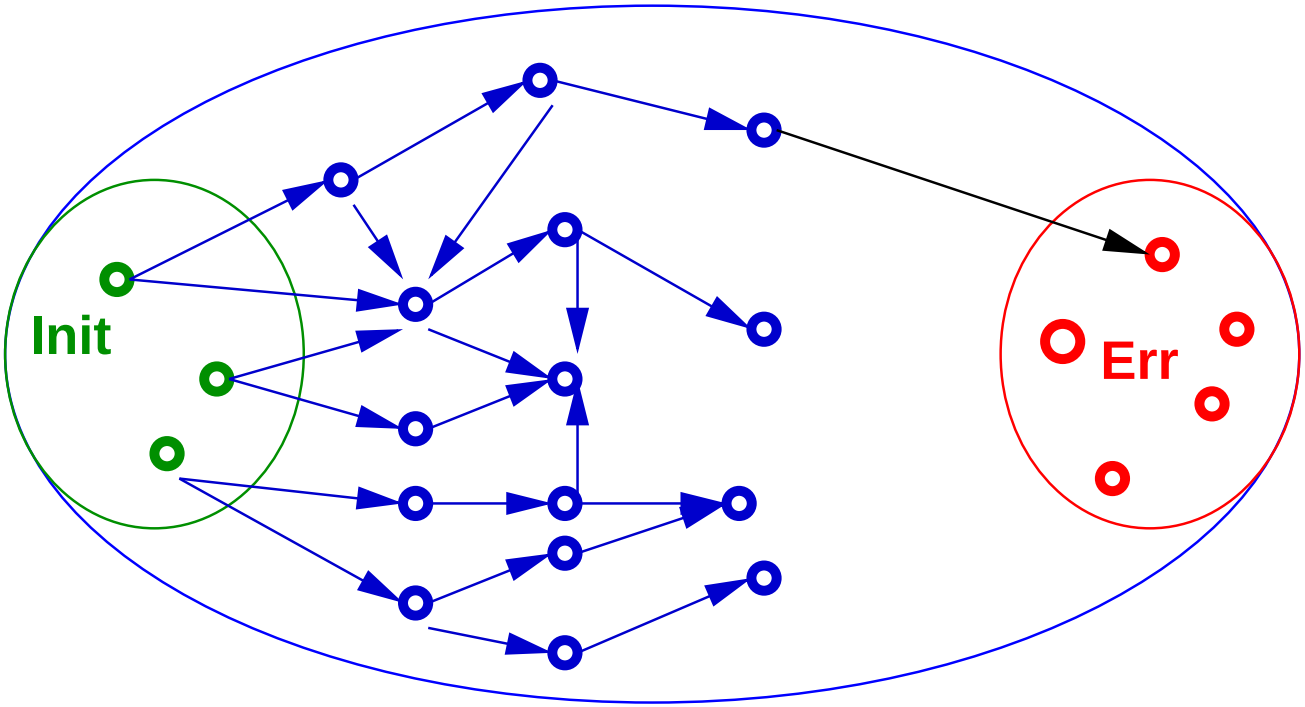

Init
Err


Init
Err


Init
Err


Init
Err


Init
Err



La preuve échoue !

Remarques

- Juste un schéma général : il y a beaucoup à faire pour que ce soit (un peu) efficace.
- De toute manière, la complexité est énorme, même pour des programmes moyens (2^S états, 2^{2S+V} transitions.
- Un algo énumératif en arrière n'a pas beacoup d'intérêt car il fait perdre l'avantage du déterminisme.

Algorithmes symboliques

Principes

- S'affranchir de la taille concrète des ensembles (d'états, de transitions) en se basant sur une représentation implicite.
  Par exemple, si x, y, z sont des variables d'états, le prédicat $(x \oplus y) \wedge \neg z$ "code" tous les états où z est faux, et où soit x, soit y est vrai.
- Rendu possible par la (re)découverte des techniques de graphes de décision (BDD).

Notes sur les BDDS

Inutile de savoir comment marchent les BDDS pour comprendre les algos, on retiendra simplement :

- **C'est une représentation (compacte) des fonctions booléennes (donc des ensembles)**
- **C'est une représentation *canonique*, i.e. l'égalité sémantique est "built-in"**
- **Les opérations usuelles sont disponibles : union (or), intersection (and), complément (not).**
- **Donc aussi les quantifications $\exists x\ f(x,y) = f(0,y) \vee f(1,y)$**

N.B. Le coût de la traduction d'une formule en BDDS est exponentiel (temps et mémoire) dans le pire des cas.

En avant

- **Manipule une variable BDD A (ensemble des états accessibles connus)**
- **La complexité est essentiellement dans le calcul de $POST_H$**

A := Init

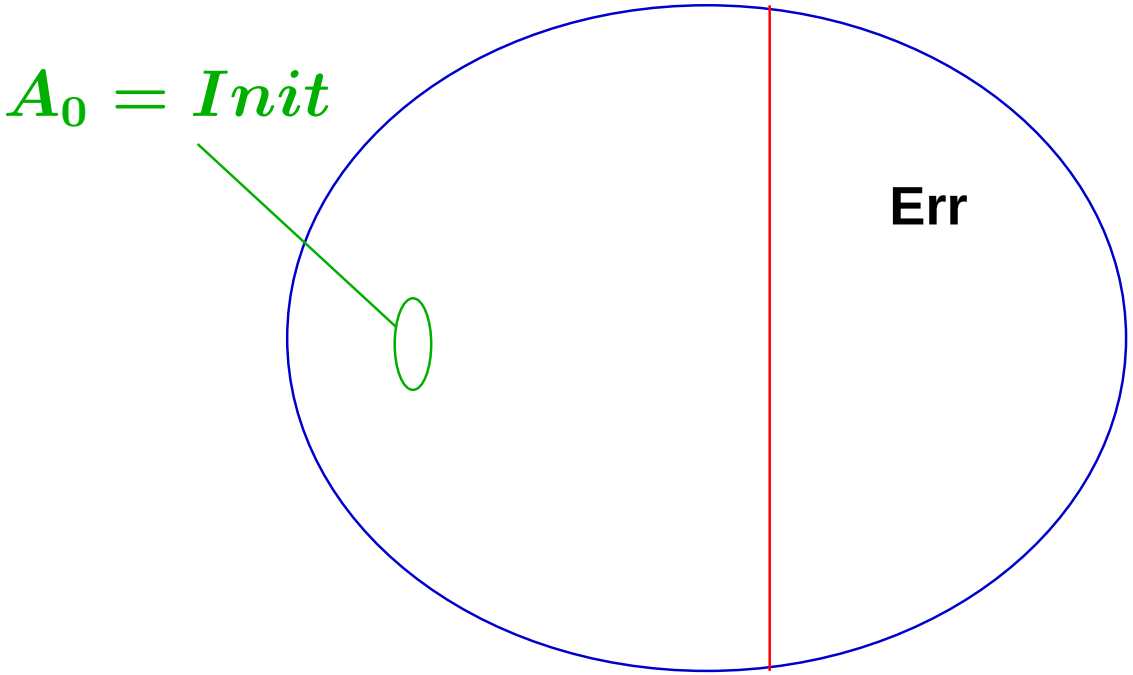
while true $\{$

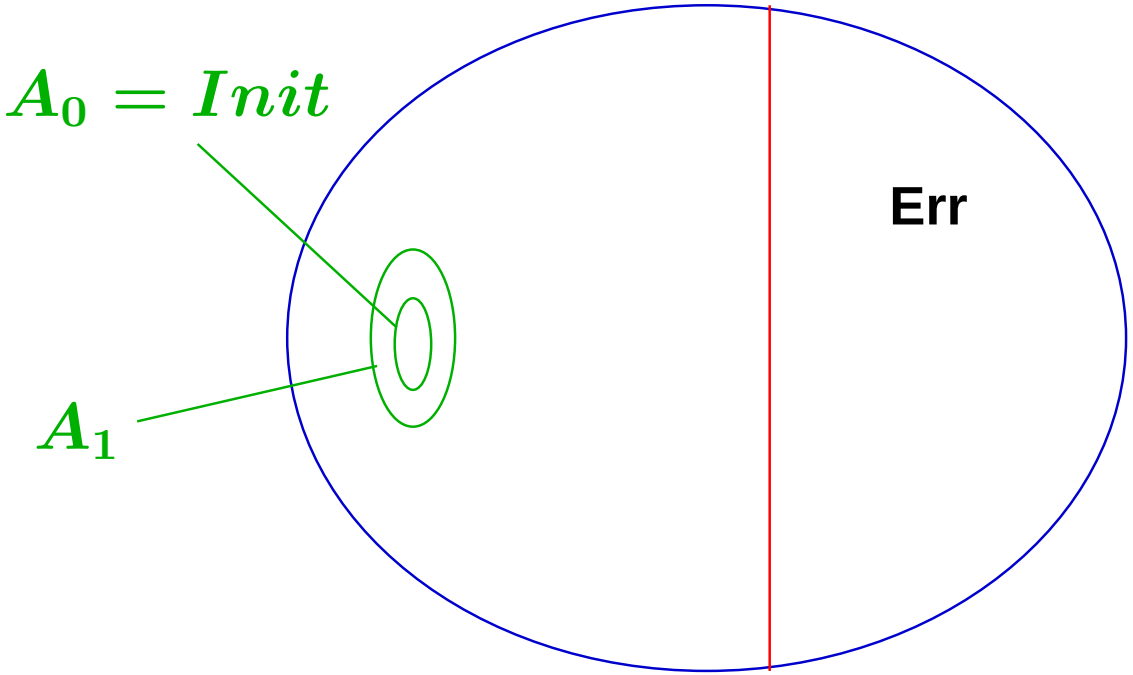
if $A \cap$ Err $\neq \emptyset$ then EXIT(failed)

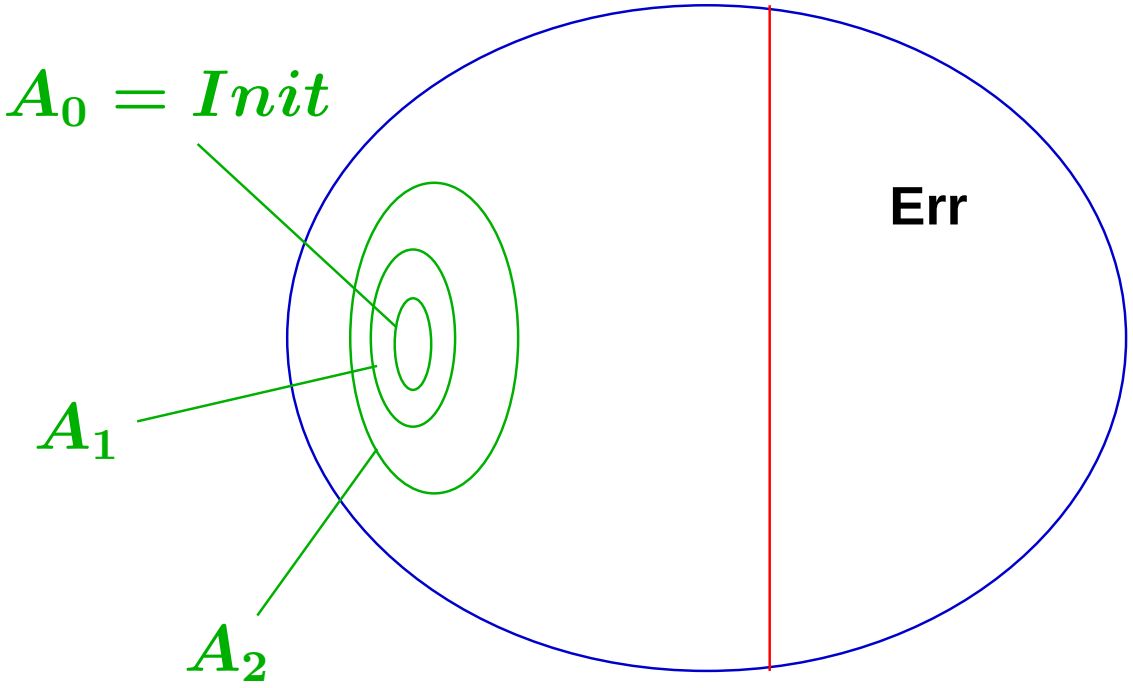
A' := $A \cup POST_H(A)$

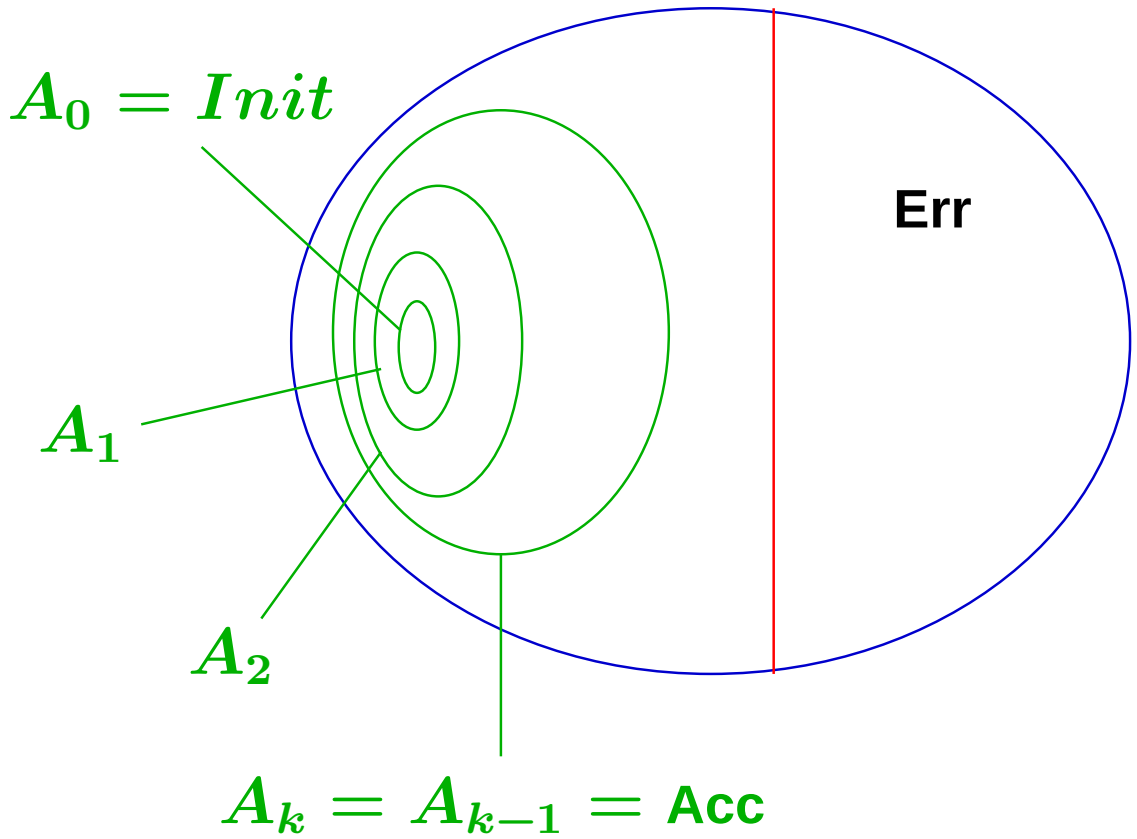
if A' == A then EXIT(succeed)

else A := A'

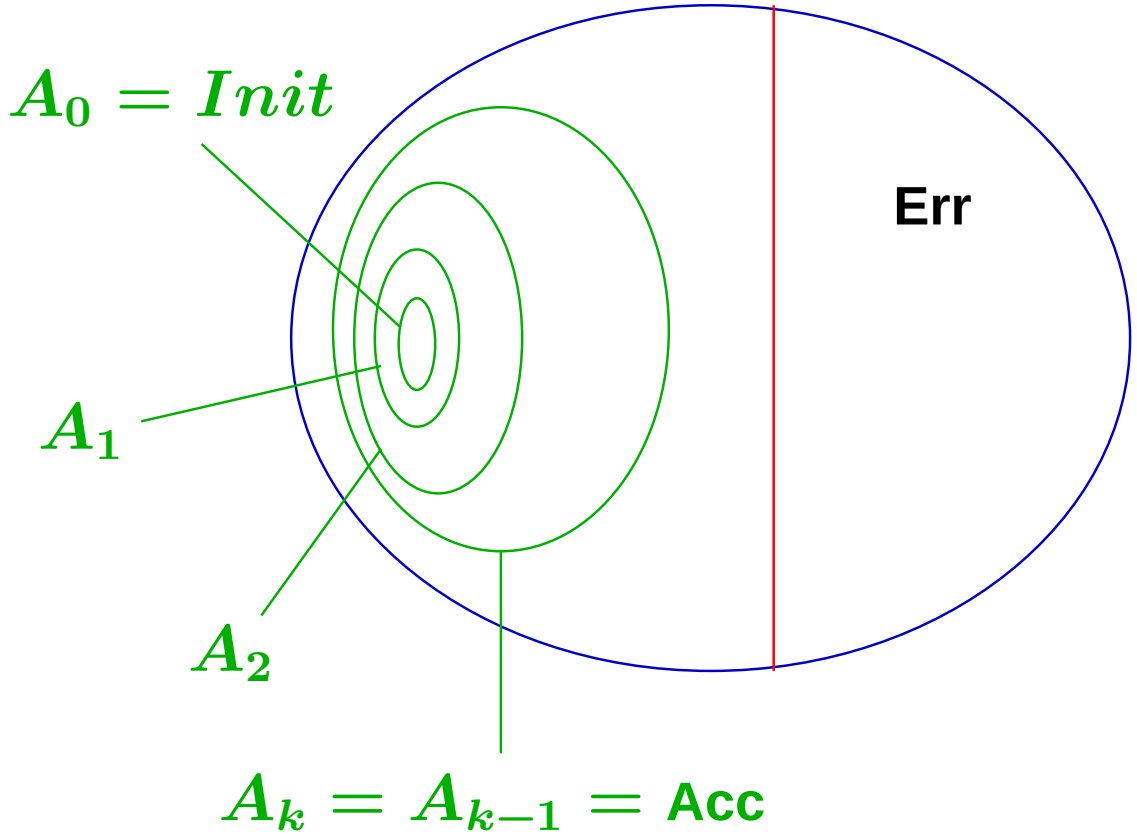

$A_0 = Init$
Err

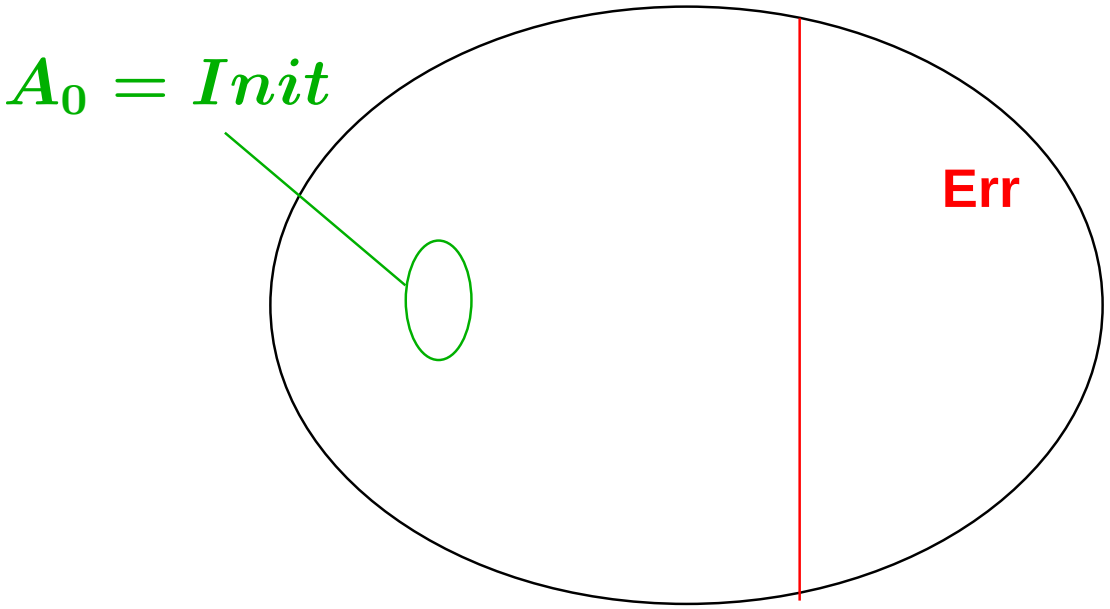

$A_0 = Init$
A_1
Err

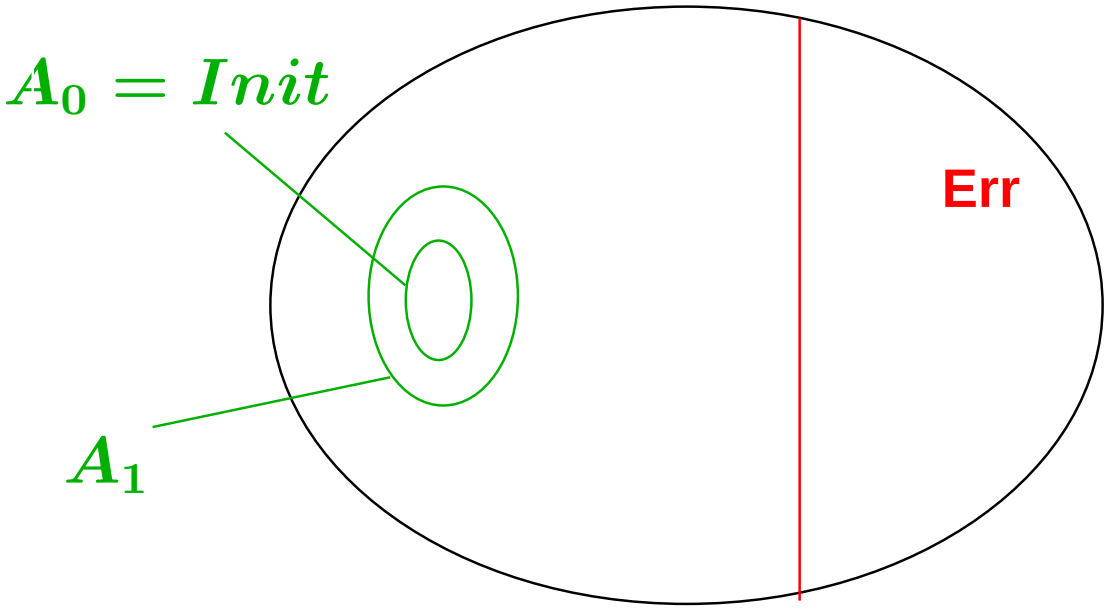

$A_0 = Init$
A_1
A_2
Err

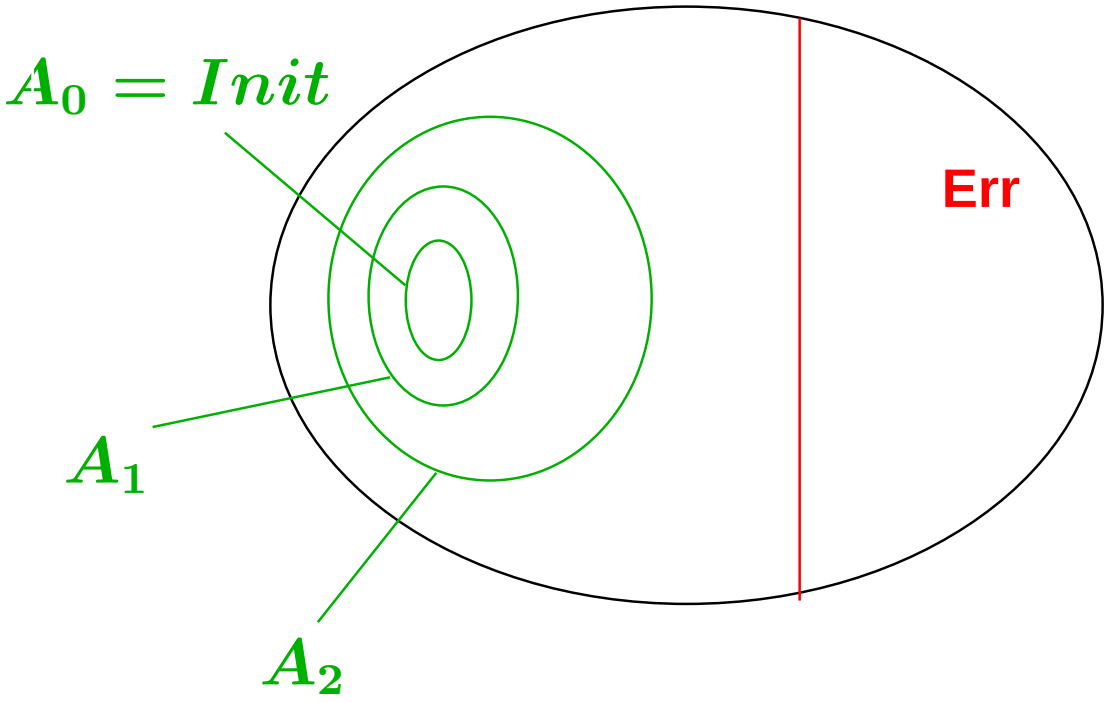

$A_0 = Init$
A_1
A_2
$A_k = A_{k-1} = \text{Acc}$
Err

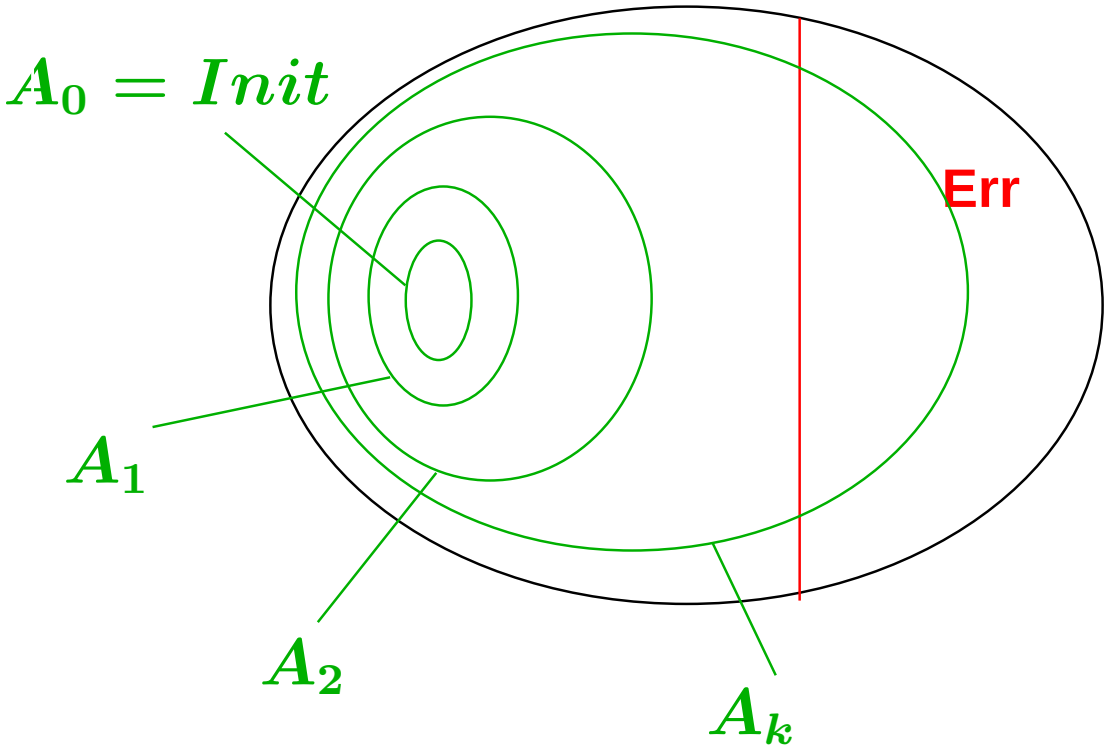
La preuve réussit

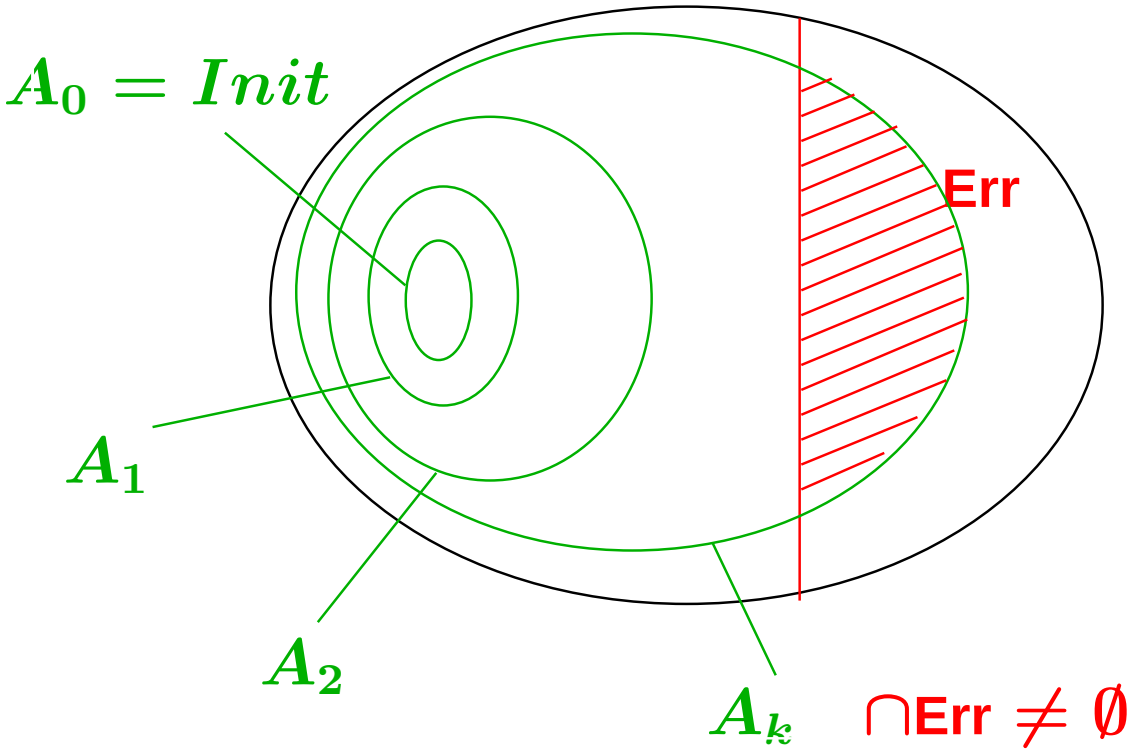



$A_0 = Init$
Err

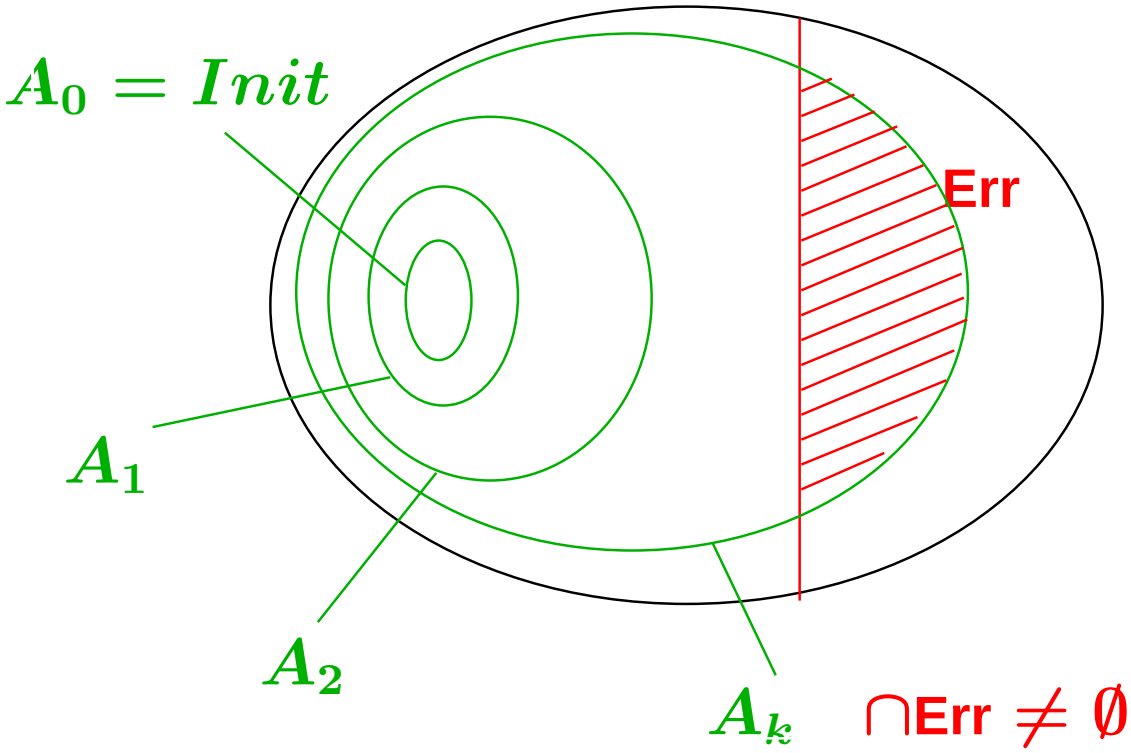

$A_0 = Init$
Err
A_1


$A_0 = Init$
Err
A_1
A_2


$A_0 = Init$
Err
A_1
A_2
A_k


$A_0 = Init$
Err
A_1
A_2
A_k
$\cap$Err $\neq \emptyset$

La preuve échoue



Calcul symbolique d'image

- **Le calcul de l'image ($POST_H$) est le extrêmement coûteux.**
- **Les "vrais" algos sont très sophistiqués.**
- **Juste le principe de faisabilité :**

Calcul symbolique d'image

- **Le calcul de l'image ($POST_H$) est le extrêmement coûteux.**
- **Les "vrais" algos sont très sophistiqués.**
- **Juste le principe de faisabilité :**

  $X(\vec{s})$

  $\longrightarrow$ $\vec{s}$ **est un état source**

Calcul symbolique d'image

- **Le calcul de l'image ($POST_H$) est le extrêmement coûteux.**
- **Les "vrais" algos sont très sophistiqués.**
- **Juste le principe de faisabilité :**

$$X(\vec{s}) \wedge H(\vec{s}, \vec{v})$$

$\longrightarrow$ $\vec{s}$ **est un état source**

$\longrightarrow$ $(\vec{s}, \vec{v})$ **satisfont les hypothèses**

Calcul symbolique d'image

- **Le calcul de l'image ($POST_H$) est le extrêmement coûteux.**
- **Les "vrais" algos sont très sophistiqués.**
- **Juste le principe de faisabilité :**

$$X(\vec{s}) \wedge H(\vec{s},\vec{v}) \wedge \bigwedge_{i=1}^{n} s'_i = g_i(\vec{s},\vec{v})$$

$\longrightarrow$ $\vec{s}$ **est un état source**

$\longrightarrow$ $(\vec{s},\vec{v})$ **satisfont les hypothèses**

$\longrightarrow$ **chaque** s'_i **est l'image du** g_i **correspondant**

Calcul symbolique d'image

- **Le calcul de l'image ($POST_H$) est le extrêmement coûteux.**
- **Les "vrais" algos sont très sophistiqués.**
- **Juste le principe de faisabilité :**

$$\exists \vec{s}, \vec{v}(\ X(\vec{s}) \wedge H(\vec{s}, \vec{v}) \wedge \bigwedge_{i=1}^{n} s'_i = g_i(\vec{s}, \vec{v}) \ )$$

$\longrightarrow$ $\vec{s}$ **est un état source**

$\longrightarrow$ $(\vec{s}, \vec{v})$ **satisfont les hypothèses**

$\longrightarrow$ **chaque** s'_i **est l'image du** g_i **correspondant**

$\longrightarrow$ **élimination des** s_i **et des** v_j

En arrière

- **Dual de en avant : on inverse les rôles des ensembles, et on prend PRE_H au lieu de $POST_H$**

B := Err

while true $\{$

if $B \cap$ Init $\neq \emptyset$ then EXIT(failed)

$B' := B \cup PRE_H(B)$

if $B' == B$ then EXIT(succeed)

else $A := A'$

- **PRE_H est “simple” à comprendre (composition de fonctions)**
- **mais pas forcément plus efficace...**
- **à l’usage, la méthode en avant marche *souvent* mieux que la méthode en arrière.**

Conclusion et travaux en relation

- La vérification des systèmes finis est théoriquement décidable,
- mais se heurte au problème de l'explosion combinatoire.
- Les BDDS et les méthodes symboliques ont cependant permis des progrès significatifs.
- Des techniques de décision alternatives somnt utilisées (méthodes SAT).
- Les sytèmes trop gros et/ou trop complexes peuvent être abordés avec les même techniques que dans le cas indécidable (interprétation abstraite).