Model Checking Modulo Theories (MCMT)

Sylvain Conchon

Cours 6 / 2 avril 2014

MCMT avec  Cubicle

`http://cubicle.lri.fr`

Comment . . .

- modéliser la sémantique d'un programme concurrent ?
- décrire un système avec un nombre arbitraire de processus ?
- ça marche ?
- développer un *model checker* efficace ?

Comment modéliser la sémantique d'un programme concurrent ?

Modélisation par systèmes de transition

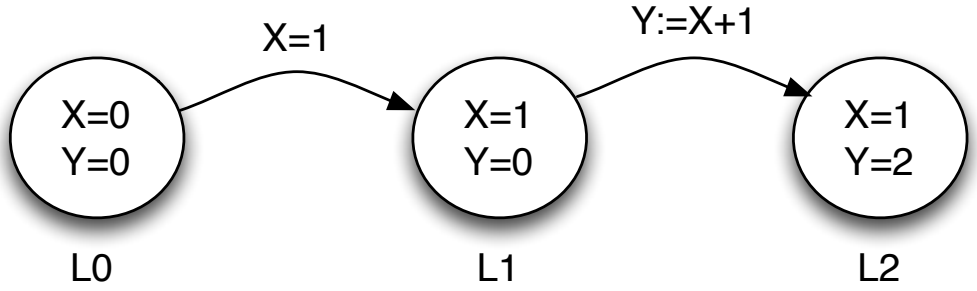
Un programme concurrent est vu comme un ensemble de *threads* séquentiels qui s'exécutent "en même temps"

Chaque *thread* est décrit par un système de transition

- états du système = ensemble des variables (partagées et locales)
- transitions = actions atomiques effectuées par la machine

Exemple 1 : un programme séquentiel

```
L0:
    X := 1;
L1:
    Y := X + 1;
L2:
```




```
type loc = L0 | L1 | L2
var X : int
var Y : int
var PC : loc

init () { X = 0 && Y = 0 && PC = L0 }

transition t1 ()
requires { PC = L0 }
{ X := 1; PC := L1  }

transition t2 ()
requires { PC = L1 }
{ Y := X + 1; PC := L2  }
```

Granularité des opérations

```
L0:
    X := 1;
L1:
    Y := X + 1;
L2:
```

Granularité des opérations

```
L0:
    X := 1;
L1:
    EAX := X;
L2:
    EAX := EAX + 1;
L3:
    Y := EAX;
```

Granularité des opérations

```
L0:
    X := 1;
L1:
    EAX := X;
L2:
    EAX := EAX + 1;
L3:
    Y := EAX;
L4:
```

```
type loc = L0 | L1 | L2 | L3 | L4
var X : int
var Y : int
var EAX : int
var PC : loc

init () { X = 0 && Y = 0 && PC = L0 }

transition t1 ()
requires { PC = L0 }
{ X := 1; PC := L1  }
transition t2 ()
requires { PC = L1 }
{ EAX := X; PC := L2  }
transition t3 ()
requires { PC = L2 }
{ EAX := EAX + 1; PC := L3  }
transition t4 ()
requires { PC = L3 }
{ Y := EAX; PC := L4  }
```

Sémantique par entrelacement

- construire des systèmes concurents à partir de plusieurs composants séquentiels
- mélanger / entrelacer les actions de ces composants
- aucune hypothèse n'est faite sur l'ordre dans lequel les processus s'exécutent
- explosion combinatoire (tous les entrelacements possibles) !

Sémantique par entrelacement

- construire des systèmes concurents à partir de plusieurs composants séquentiels
- mélanger / entrelacer les actions de ces composants
- aucune hypothèse n'est faite sur l'ordre dans lequel les processus s'exécutent
- explosion combinatoire (tous les entrelacements possibles) !
- nombre infini d'entrelacements si le système ne termine pas

Exemple 2 : programme multi-thread

Code assembleur x86

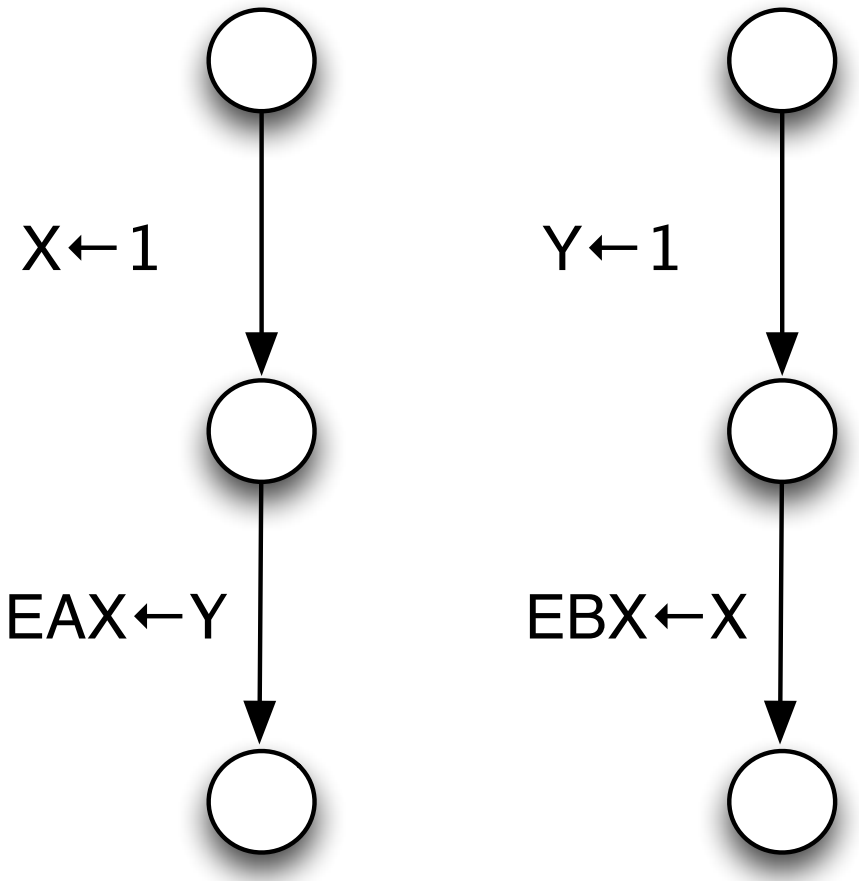
Mémoire partagée : `X` et `Y` valent initiallement `0`

Registres par processeur : `EAX` et `EBX`

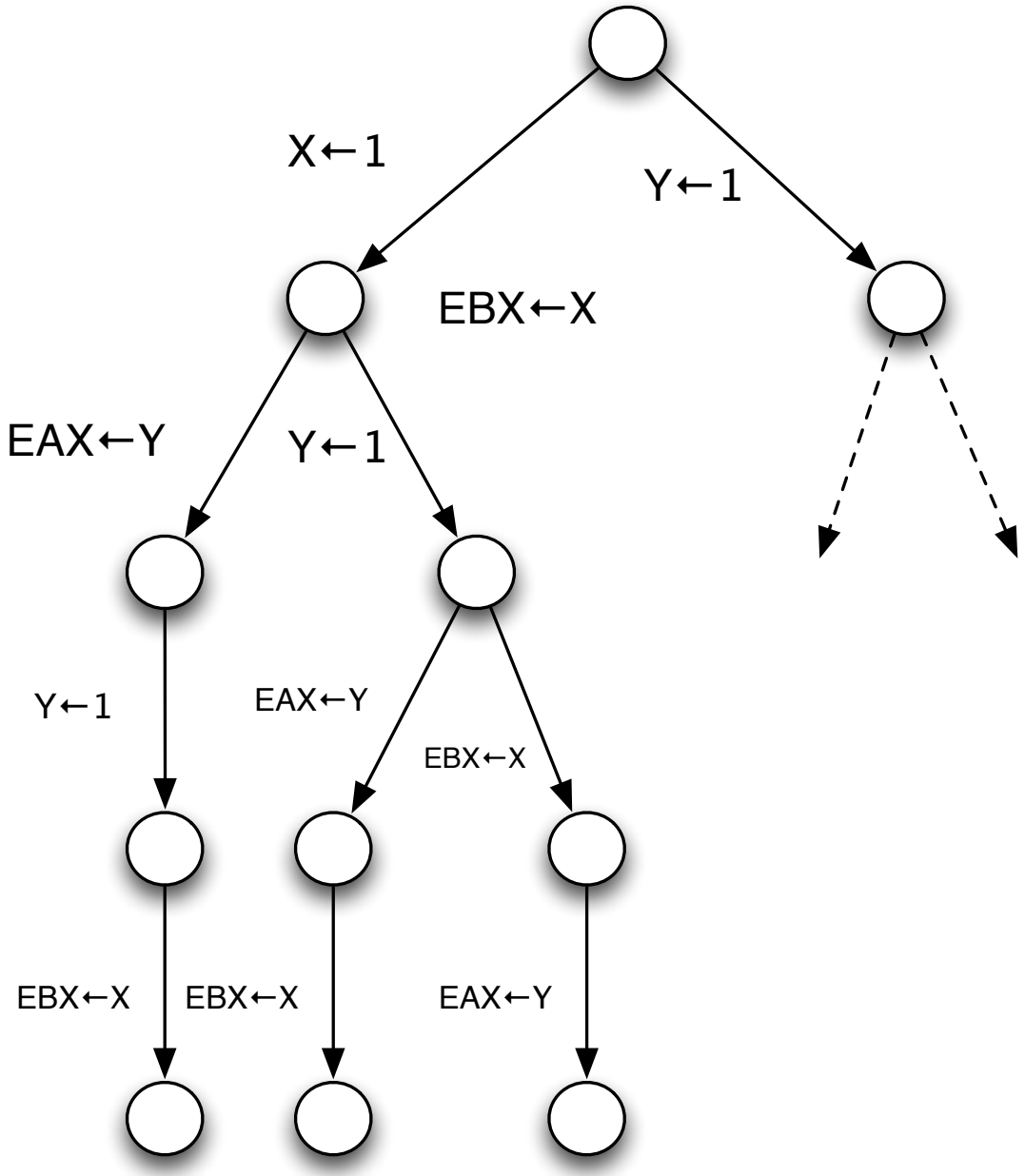
Thread 1	Thread 2
`MOV X , 1`	`MOV Y , 1`
`MOV EAX , Y`	`MOV EBX , X`

Valeurs des registres `EAX` et `EBX` à la fin de l'exécution du programme ?

Exemple : entrelacements



Exemple : entrelacements



Propriétés de sûreté

- Quels sont les états potentiellement mauvais du système ?
- Ces états sont-ils atteignables ?

Exemple 2 : programme multi-thread

Code assembleur x86

Mémoire partagée : `X` et `Y` valent initiallement `0`

Registres par processeur : `EAX` et `EBX`

Thread 1	Thread 2
`MOV X , 1`	`MOV Y , 1`
`MOV EAX , Y`	`MOV EBX , X`

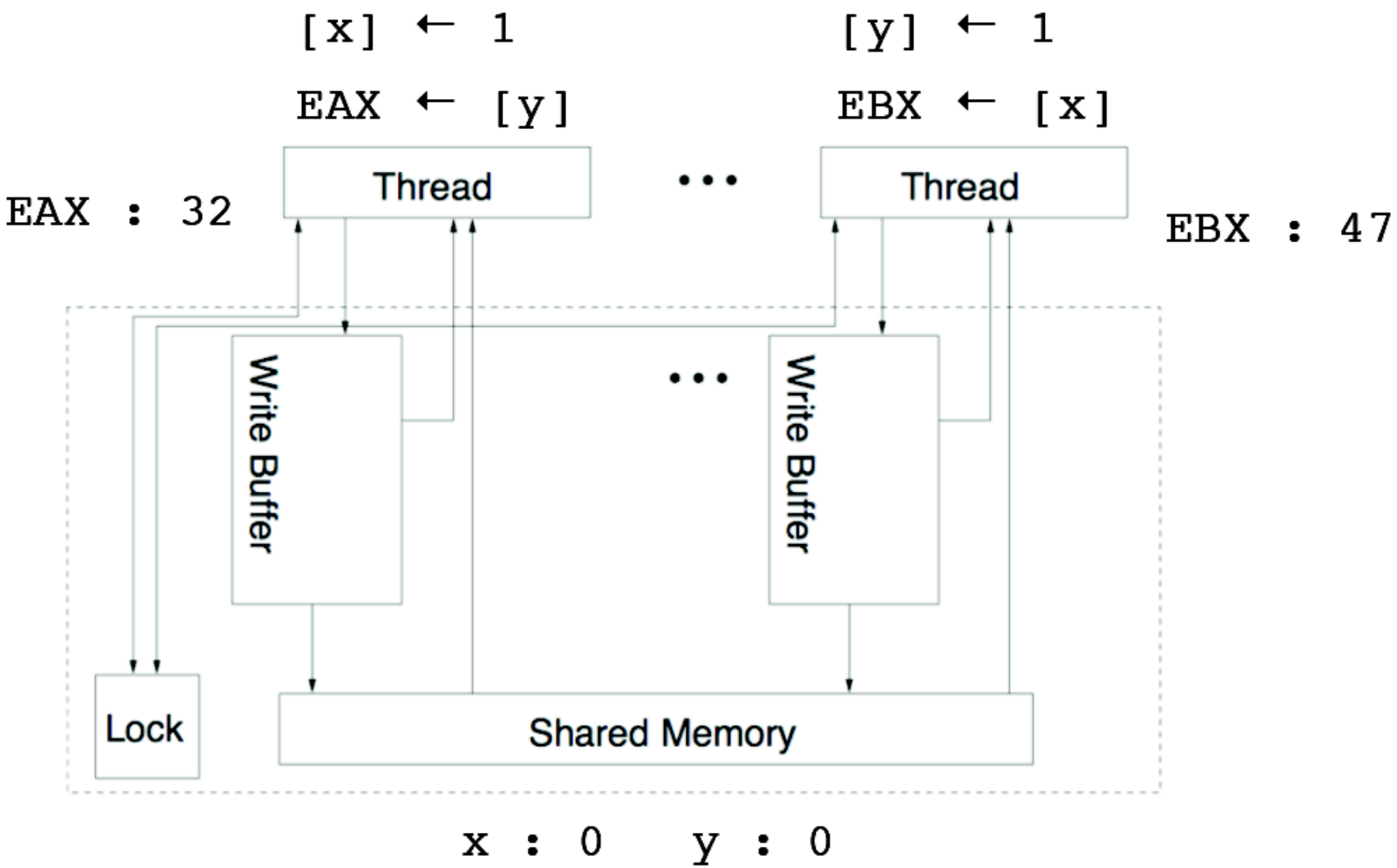
Est-ce que l'état où `EAX`=0 et `EBX`=0 est atteignable à la fin de l'exécution du programme ?

Analyses d'atteignabilité

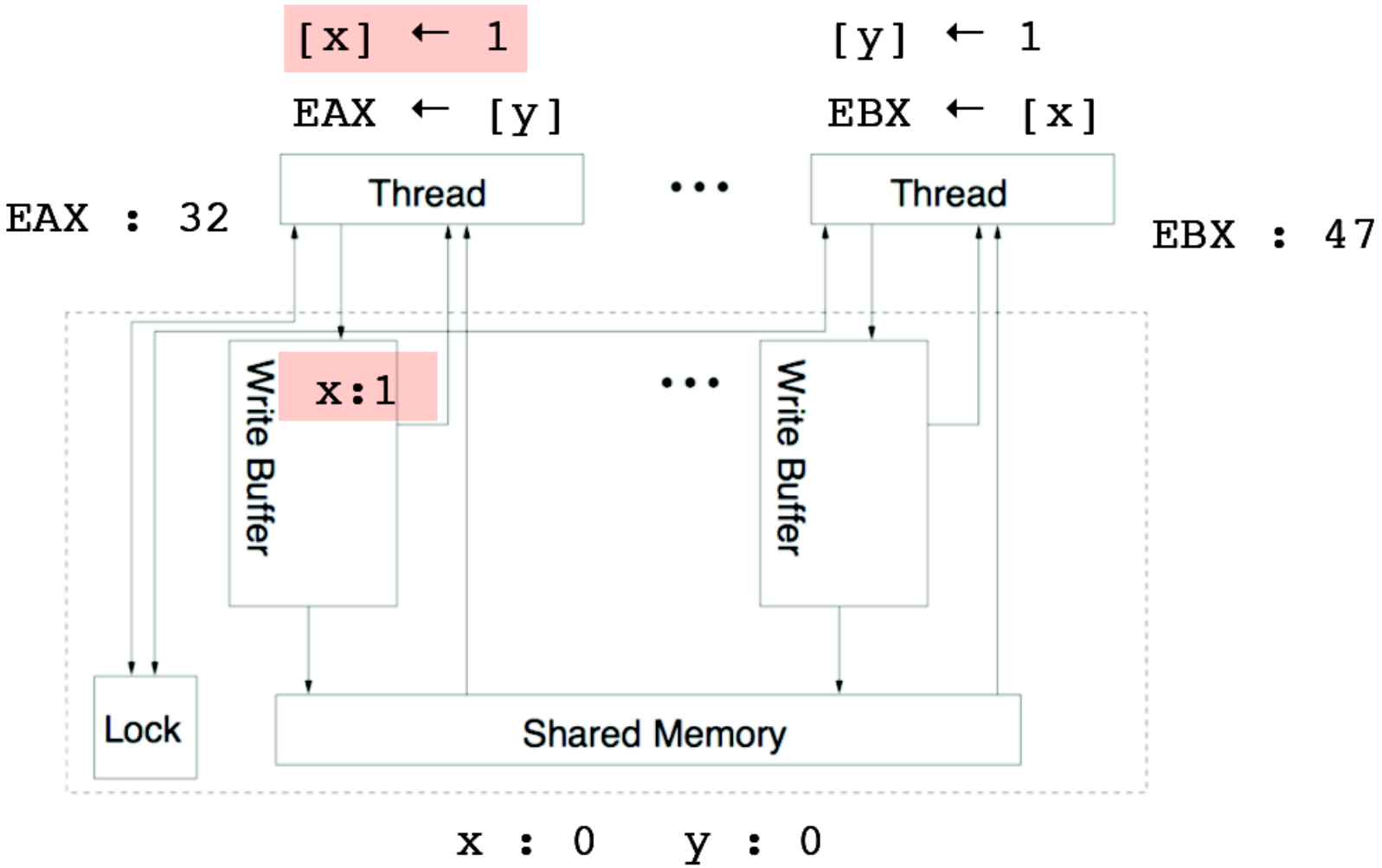
Deux techniques :

- En avant :
  Construire l'ensemble des états atteignables à partir de l'état initial et vérifier que l'état *unsafe* n'est pas dans cet ensemble.
- En arrière :
  Construire l'ensemble des états pouvant atteindre l'état *unsafe* et vérifier que l'état initial n'est pas inclus dans cet ensemble.

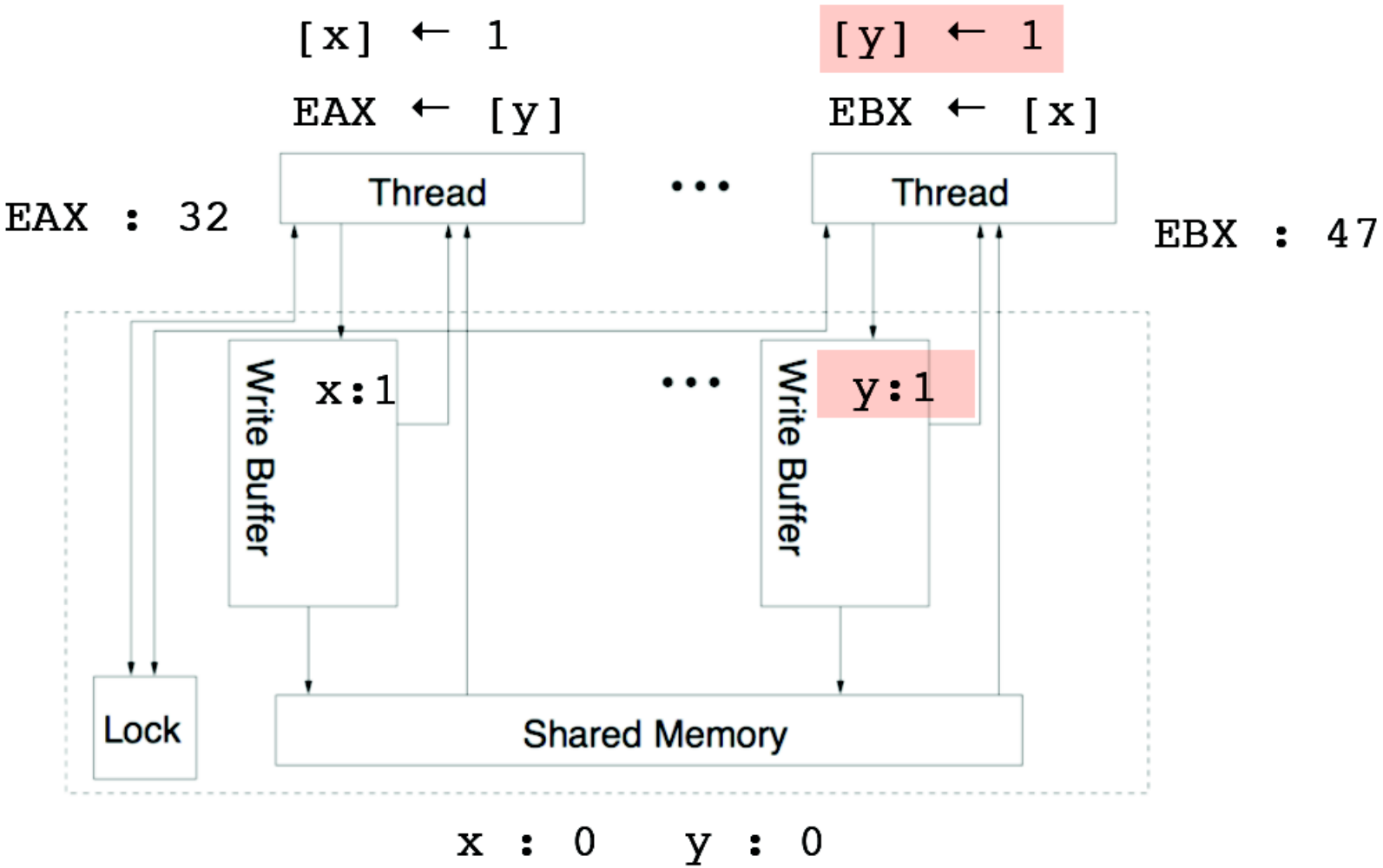
Exemple 3 : le même programme avec cache mémoire



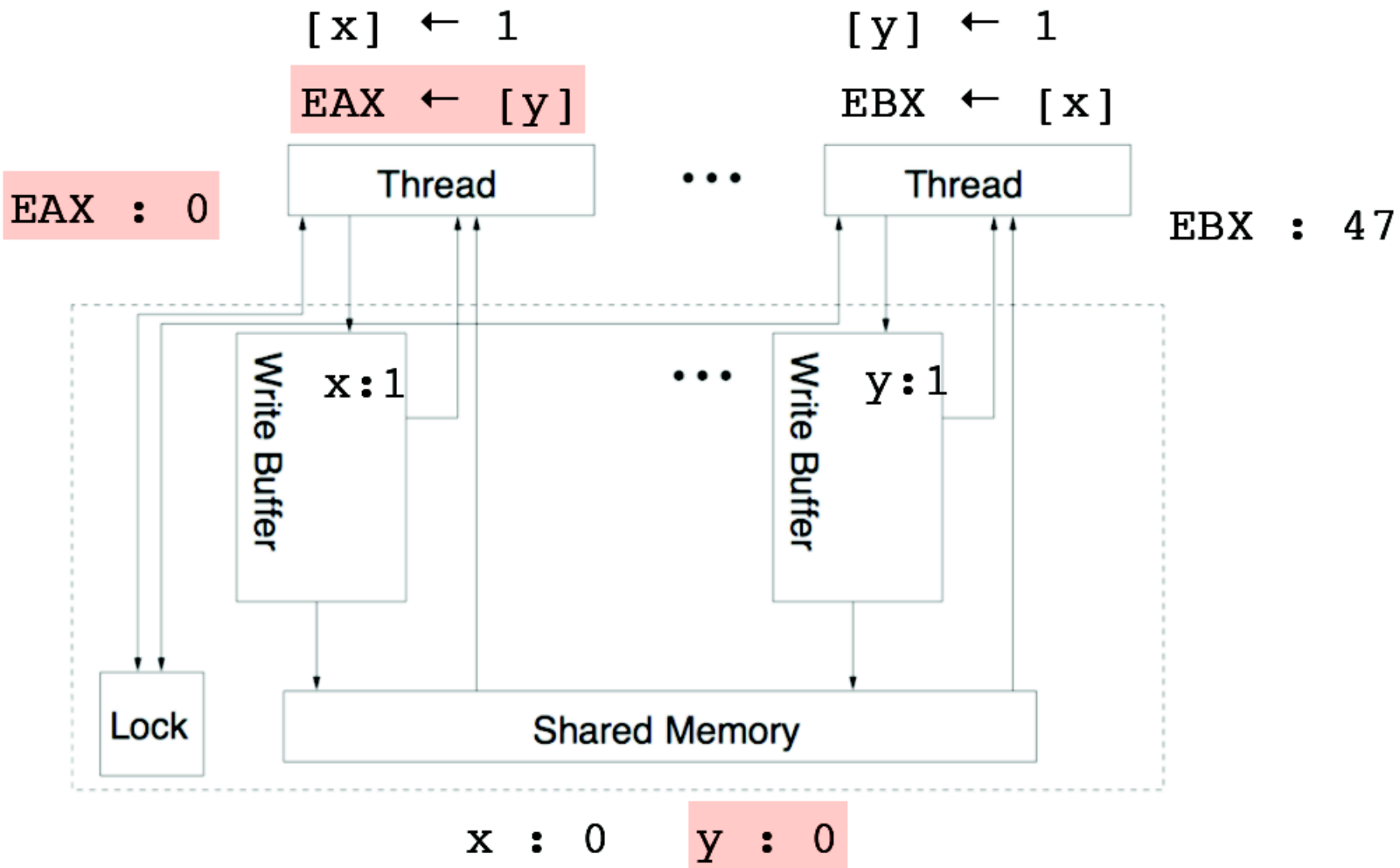
Exemple 3 : le même programme avec cache mémoire



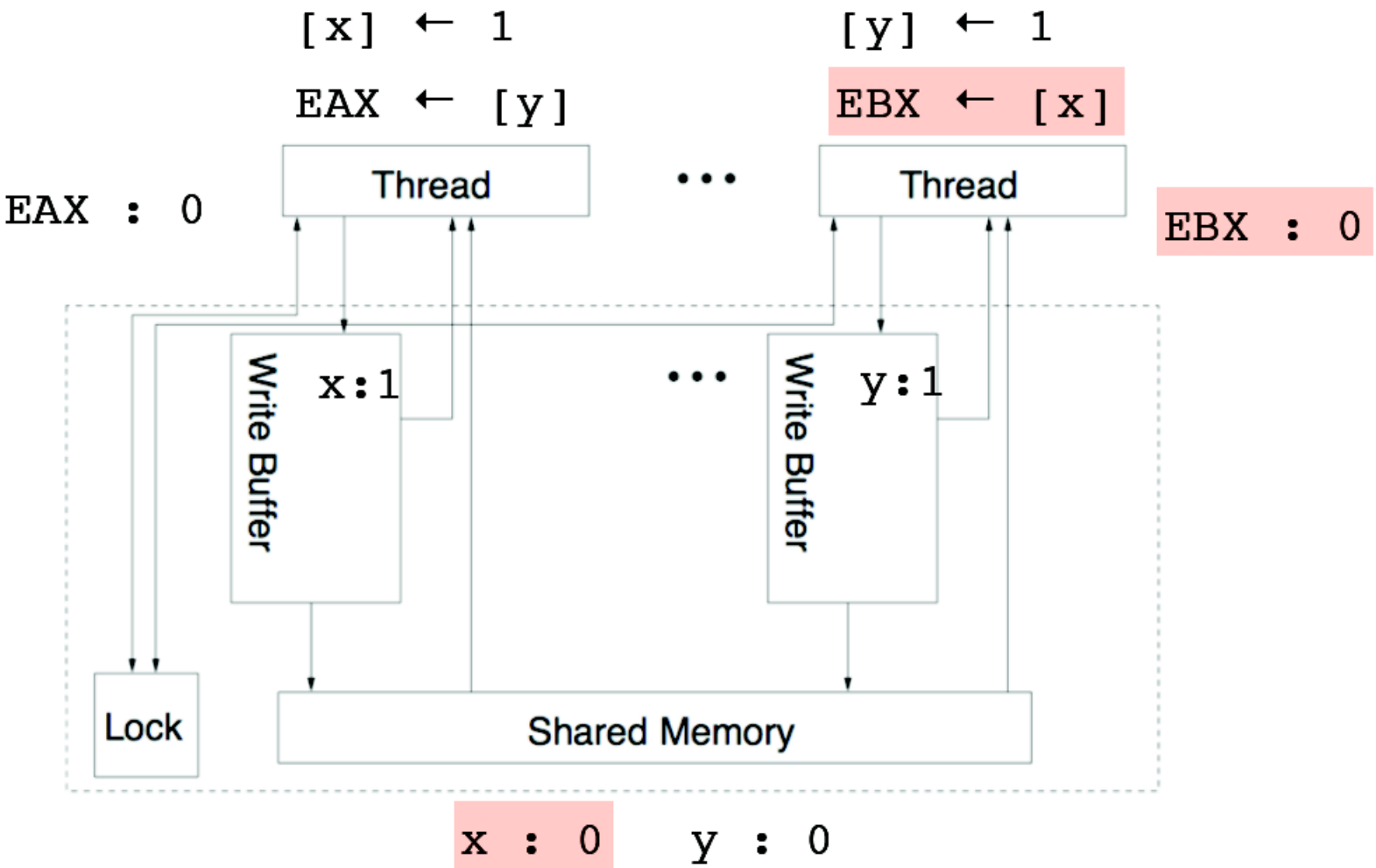
Exemple 3 : le même programme avec cache mémoire



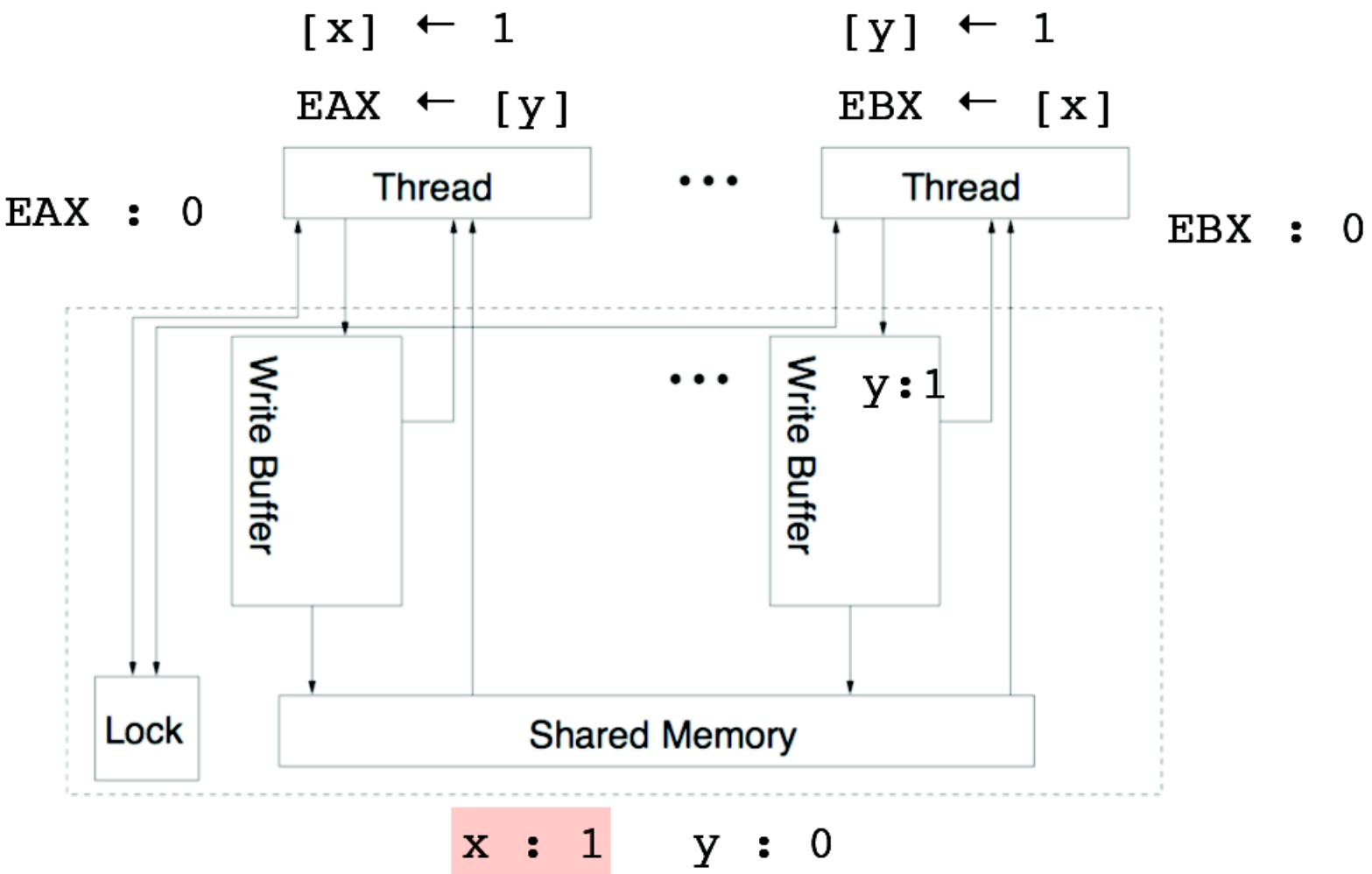
Exemple 3 : le même programme avec cache mémoire



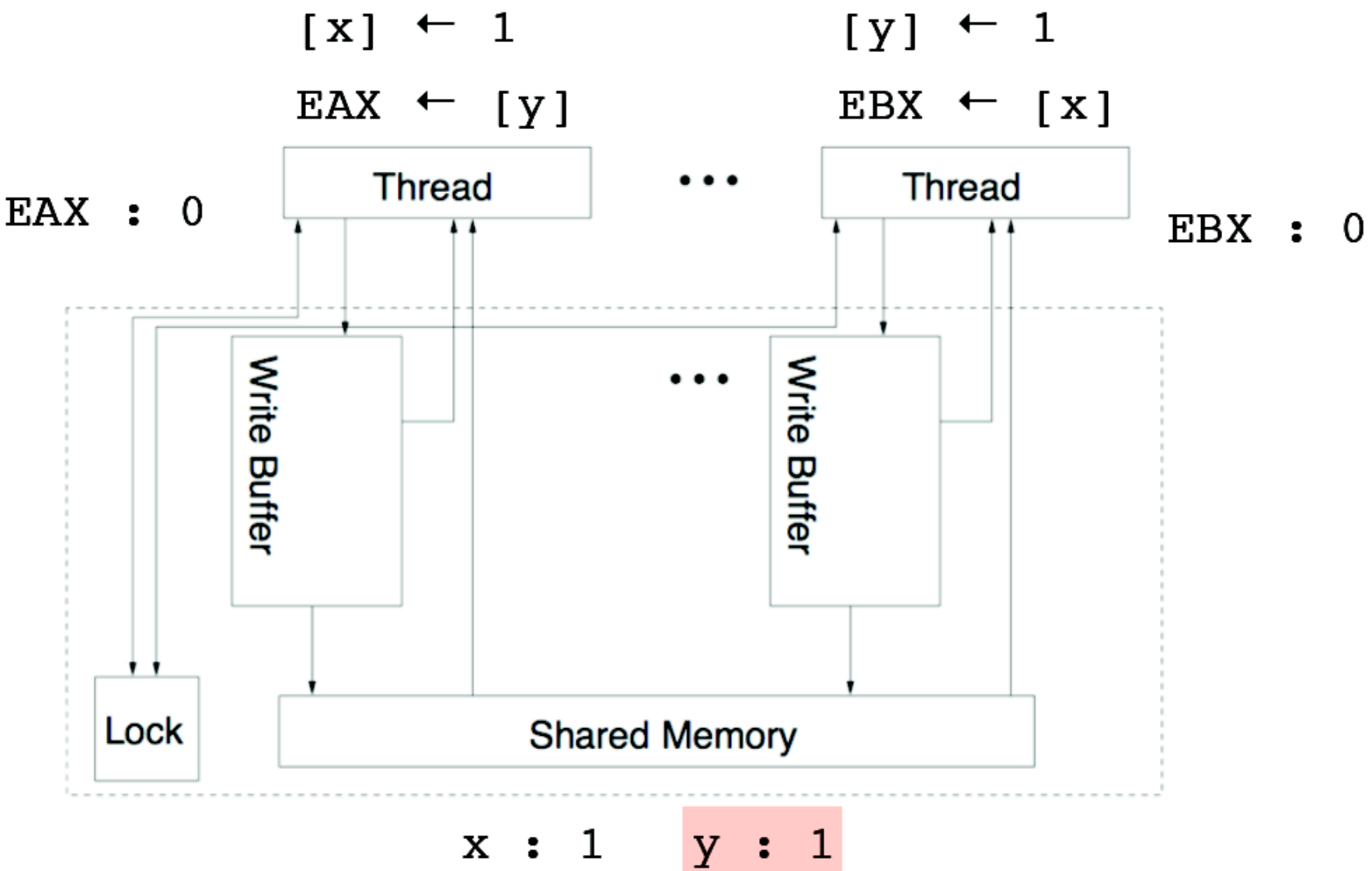
Exemple 3 : le même programme avec cache mémoire



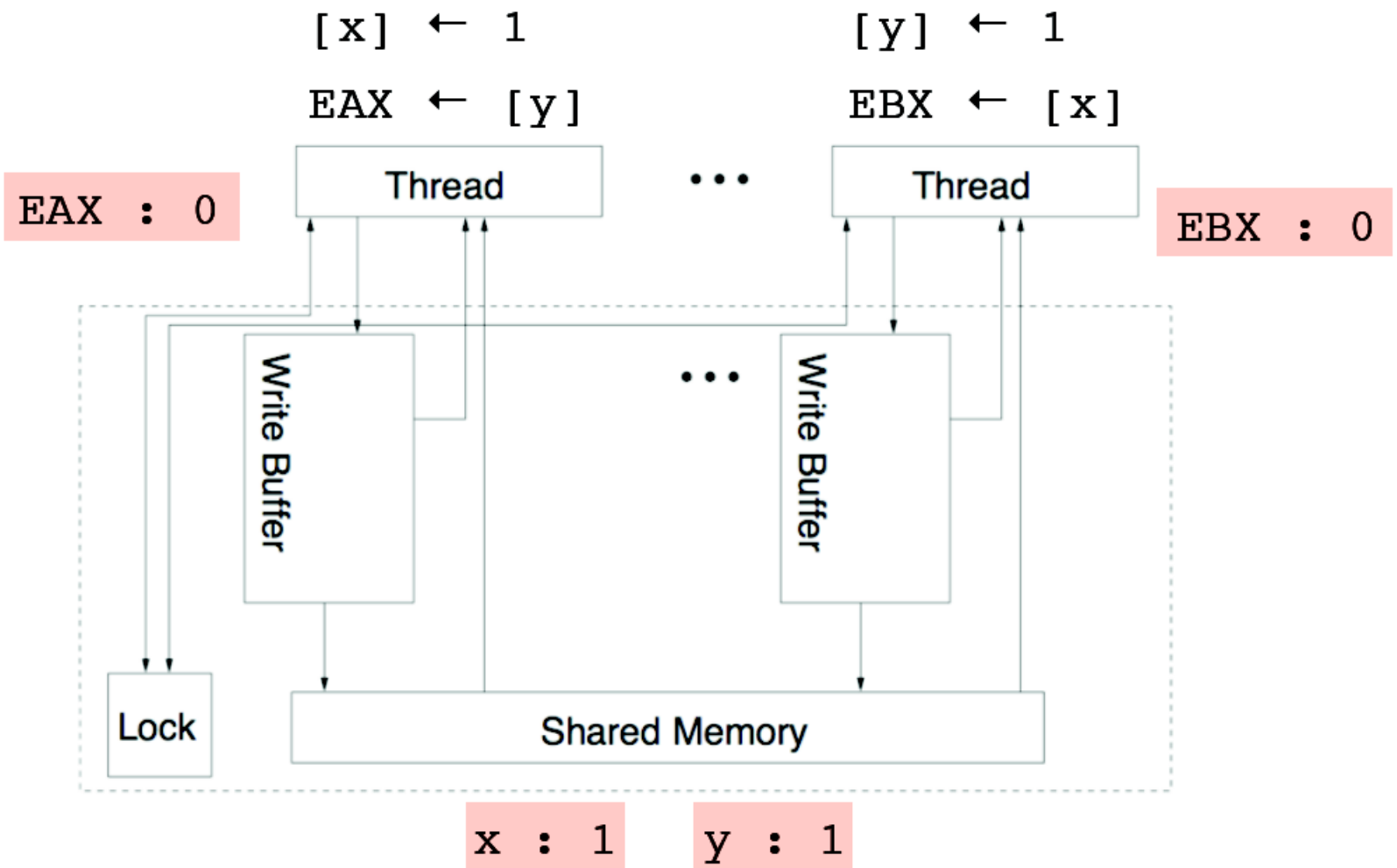
Exemple 3 : le même programme avec cache mémoire



Exemple 3 : le même programme avec cache mémoire



Exemple 3 : le même programme avec cache mémoire



Vérification de systèmes infinis

Si les types des données (e.g. entiers, réels) sont infinis alors les systèmes ont un nombre infini d'états

Il faut alors représenter des ensembles d'états infinis de manière symbolique à l'aide

- de structures de données *ad hoc*
- de formules logiques

Systèmes paramétrés

- Comment modéliser un système composé d'un nombre arbitraire de processus ?

  Exemples:
  - protocoles de cohérence de cache
  - algorithmes d'exclusion mutuelle
  - algorithmes de tolérance aux pannes

- Peut-on espérer vérifier automatiquement des propriétés de sûreté sur de tels systèmes ?

Vérification de systèmes paramétrés

Le problème est indécidable dans le cas général [Apt& Kozen, LICS 1986]

$\Rightarrow$ Trouver un cadre restreint mais suffisamment expressif pour que ce problème soit décidable

Vérification de systèmes paramétrés avec Cubicle

Cubicle permet de modéliser et vérifier des systèmes paramétrés et infinis

- Les systèmes de transitions sont paramétrés par un ensemble de processus "arbitraire"
- Les états peuvent contenir des entiers ou réels (mathématique)
- On utilise des formules logiques pour représenter des ensembles (infinis) d'états

Cubicle : systèmes de transition à tableaux

La modélisation d'un programme en Cubicle contient

- des déclaration de type
- des variables globales et des tableaux (infinis) indexés par des identificateurs de processus
- une formule quantifiée universellement pour les états initiaux
- des transitions sous la forme garde/action paramétrées par des identificateurs de processus (les gardes étant décrites par des formules logiques)

Cubicle : exemple de syntaxe d'entrée

```
type st = Idle | Want | Crit

var Timer : real
array State[proc] : st
array Flag[proc] : bool

init (z) { Flag[z] = False && State[z] = Idle  && Timer = 0.0 }

unsafe (x y) { State[x] = Crit && State[y] = Crit }

transition t1 (i j)
requires { i < j && State[i] = Idle && Flag[i] = False &&
           forall_other k. (Flag[k] = Flag[j] || State[k] <> Want) }
{
  Timer := Timer + 1.0;
  Flag[i] := True;
  State[k] := case
                | k = i : Want
                | State[k] = Crit && k < i : Idle
                | _ : State[k]
}

transition t2 (i)
...
```

Sémantique

L'éxécution d'un système infini est une boucle infinie :

1. choisi de façon non déterministe une instance d'une transition dont la garde est vraie
2. met à jour les variables d'état en fonction de l'action de la transistion choisie

Les gardes sont de la forme
$C \wedge \forall \bar{x}.(distinct(\bar{x}, \bar{y}) \Rightarrow C_1 \vee \ldots \vee C_k)$ où $\bar{y}$ sont les paramètres de la transition

Les actions sont des mises à jour des tableaux (encodée par une construction par cas) et des affectations (eventuellement non-déterministes) des variables globales

Propriétés de sûreté

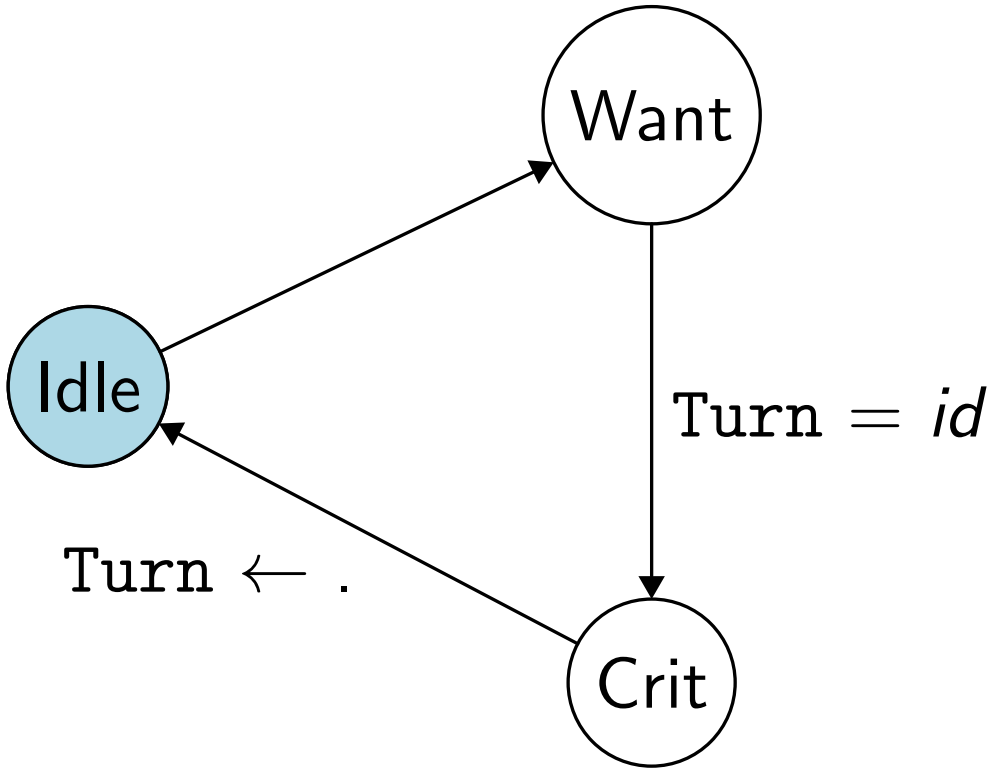
Les propriétés de sûreté sont exprimées sous leur forme négative comme des formules (closes) spéciales appelées des cubes, qui représentes les états *unsafe* :

$$\exists \bar{x}.(distinct(\bar{x}) \wedge C)$$

Un système est *sûr* si les états *unsafe* ne sont pas atteignable à partir d'un état initial

Exemple : un algorithme de mutex

Le système de transition pour un processus d'identificateur id



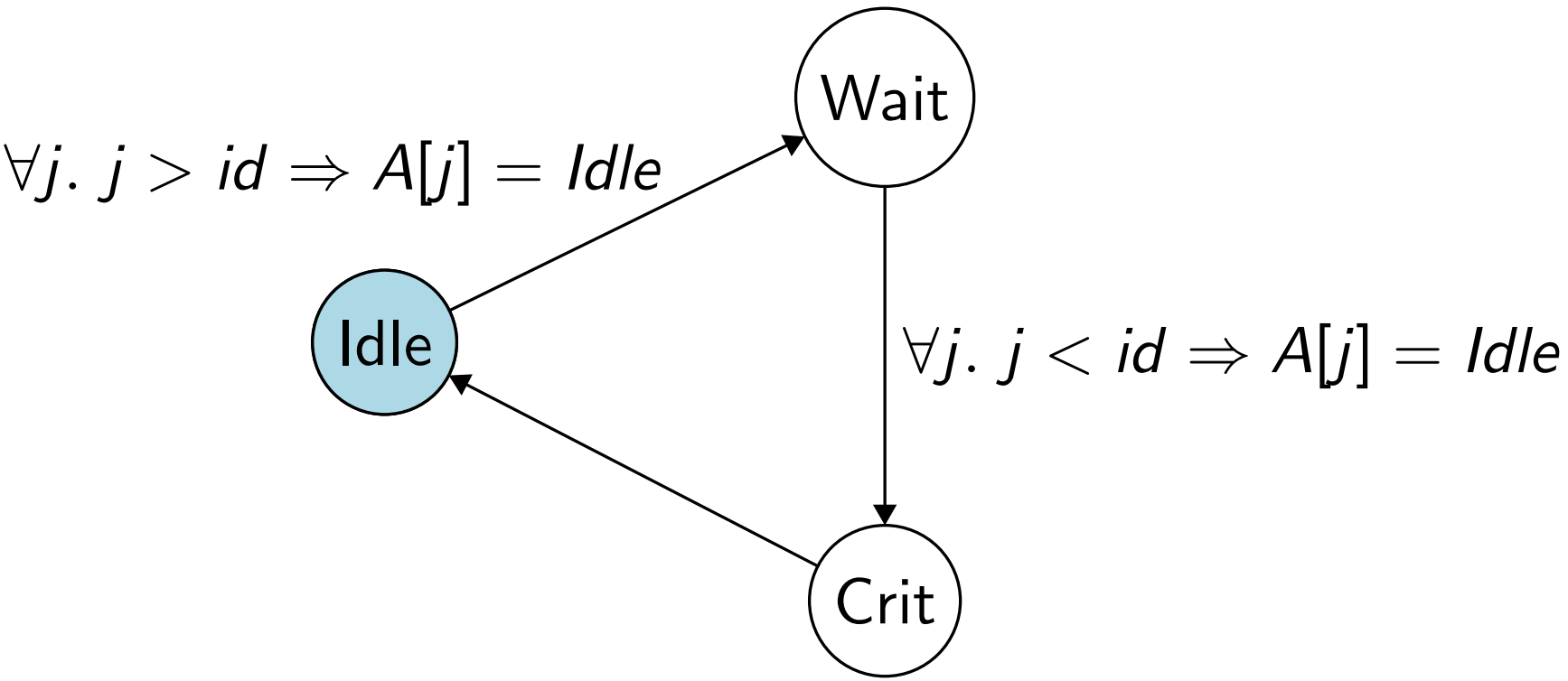
Exemple : algorithme de mutex en Cubicle

```
type st = Idle | Want | Crit
var Turn : proc
array S[proc] : st

init (z) { S[z] = Idle }

unsafe (x y) {
  S[x] = Crit && S[y] = Crit
}

transition req (i)
requires { S[i] = Idle }
{ S[i] := Want  }
```

```
transition enter (i)
requires { S[i] = Want &&
           Turn = i }
{ S[i] := Crit }

transition exit (i)
requires { S[i] = Crit }
{ Turn := . ;
  S[i] := Idle }
```

Autre exemple : l'algorithme de la boulangerie

Le système de transition pour un processus d'identificateur id



Autre exemple : l'algorithme de la boulangerie en Cubicle

```
type t = Idle | Wait | Crit
array A[proc] : t

init (z) { A[z] = Idle }

unsafe (x y) {
  A[x] = Crit && A[y] = Crit
}

transition tr1 (i)
requires { A[i] = Idle &&
 forall_other j.
  (j < i || A[j] = Idle) }
{ A[i] := Want  }

transition tr2 (i)
requires { A[i] = Wait &&
 forall_other j.
  (z < j || A[j] = Idle)  }
{ A[i] := Crit }

transition tr3 (i)
requires { A[i] = Crit }
{ A[i] := Idle }
```

Comment ça marche ?

Algorithme d'atteignabilité arrière

I : inital states U : unsafe states (cubes) $\mathcal{T}$: transitions

---

BWD ():

V := $\emptyset$;

push(Q, U) ;

while not empty(Q) **do**

  φ := pop(Q);

  if $\varphi \wedge$ I sat **then return** unsafe

  if $\neg(\varphi \models \bigvee_{\psi \in \mathtt{V}} \psi)$ **then**

    V := V $\cup \{\varphi\}$;

    push(Q, $\mathtt{pre}_{\mathcal{T}}(\varphi)$);

return safe

Démonstrateurs SMT

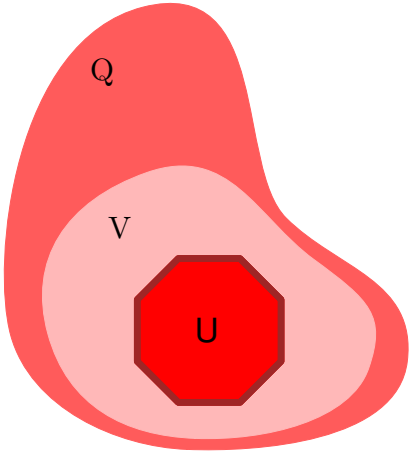
Les tests de sûreté et de point fixe

- $\varphi \wedge I$ sat ?
- $\varphi \models \bigvee_{\psi \in \mathbf{v}} \psi$

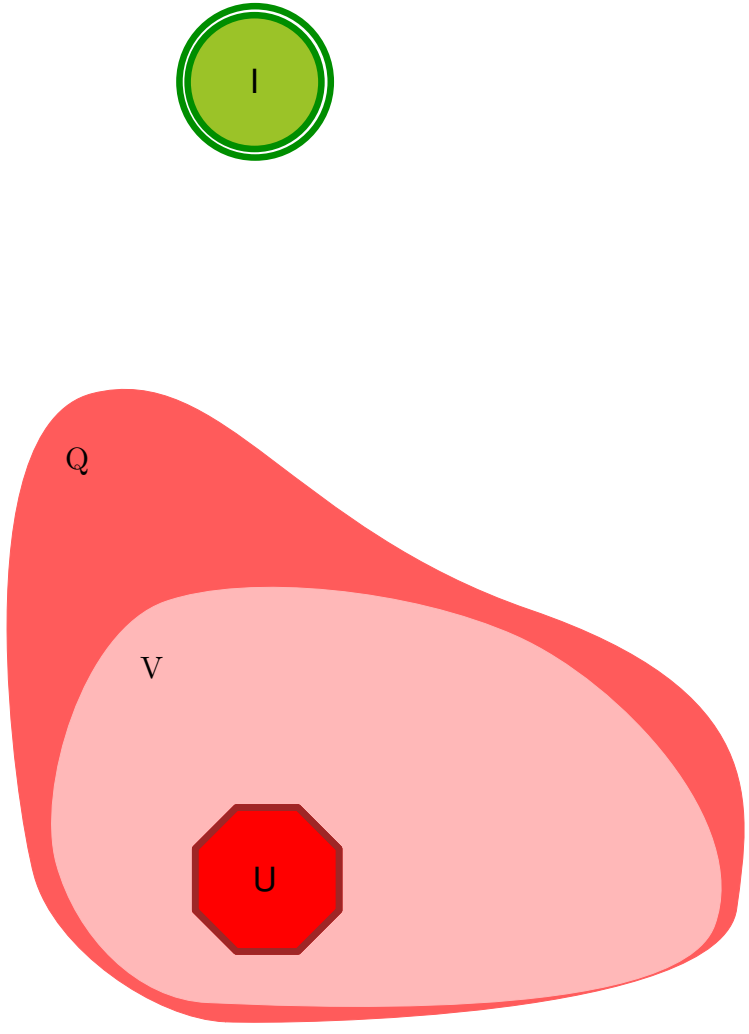
sont réalisés à l'aide d'un démonstrateur automatique de la famille SMT (Satisfiabilité Modulo Théories)

Algorithme d'atteignabilité arrière

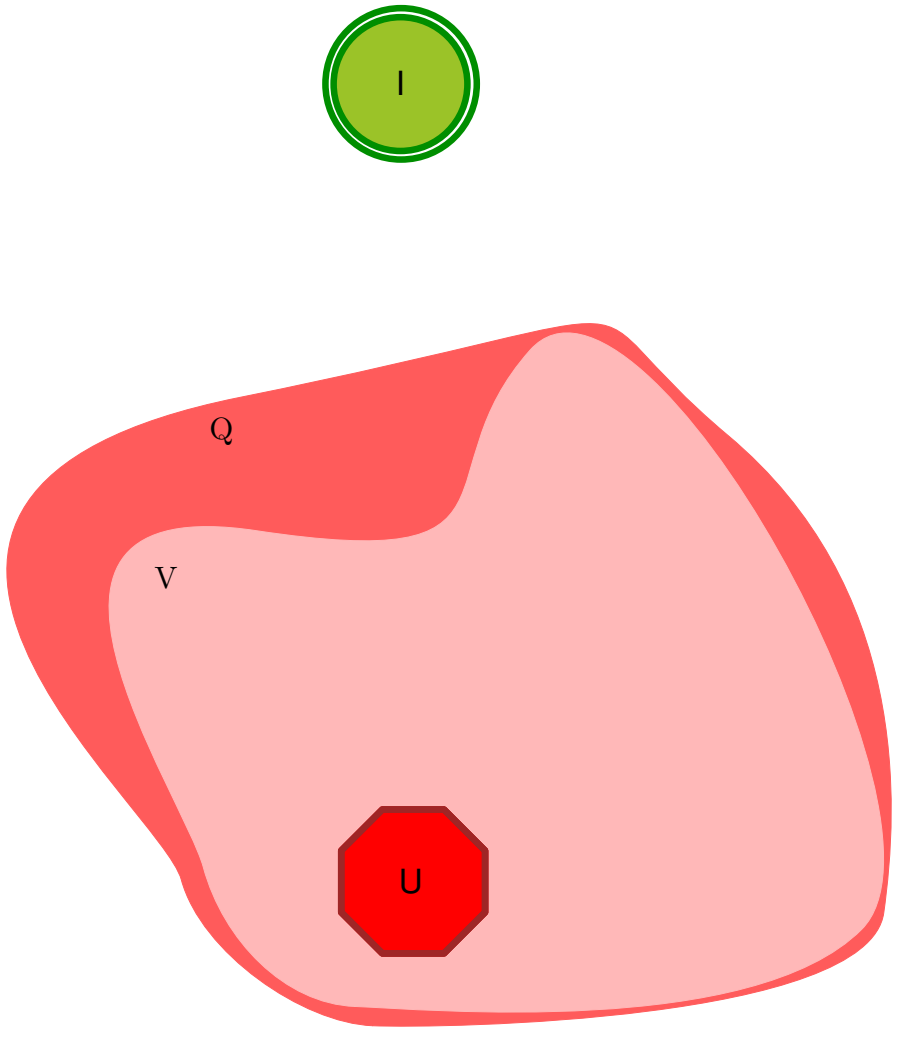
Algorithme d'atteignabilité arrière



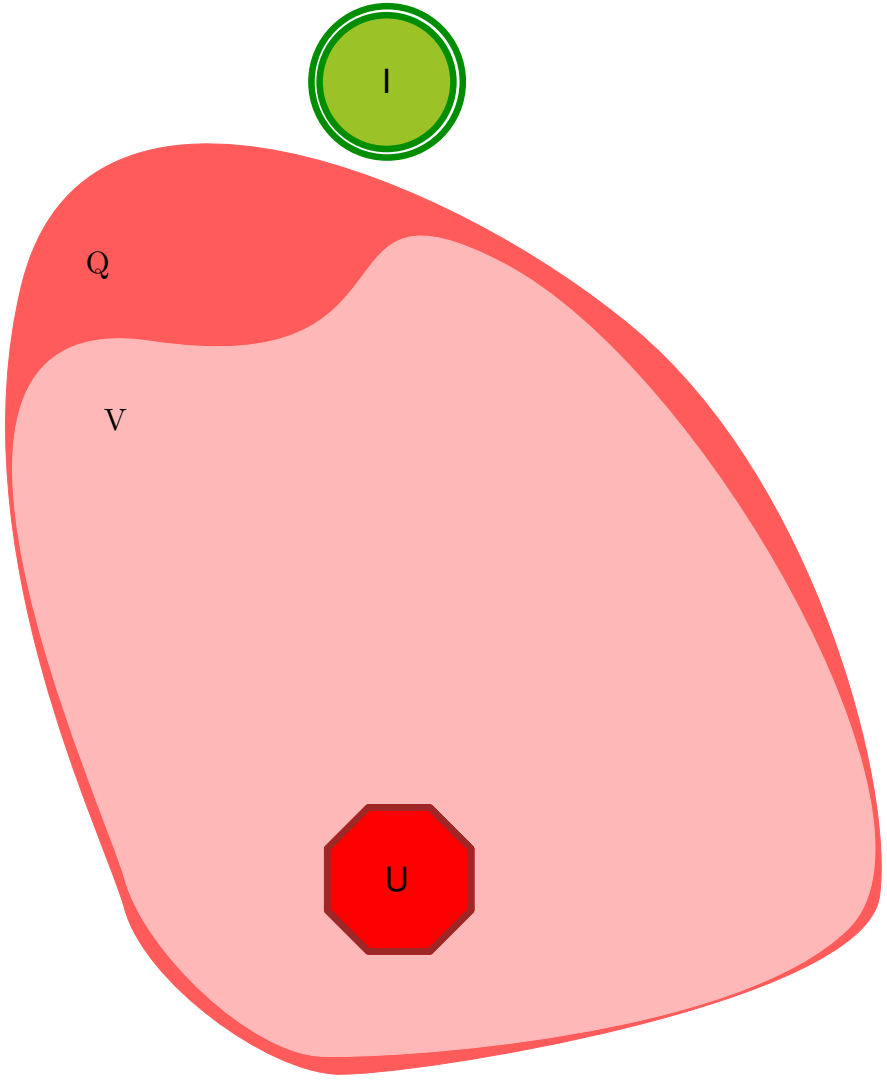
Algorithme d'atteignabilité arrière



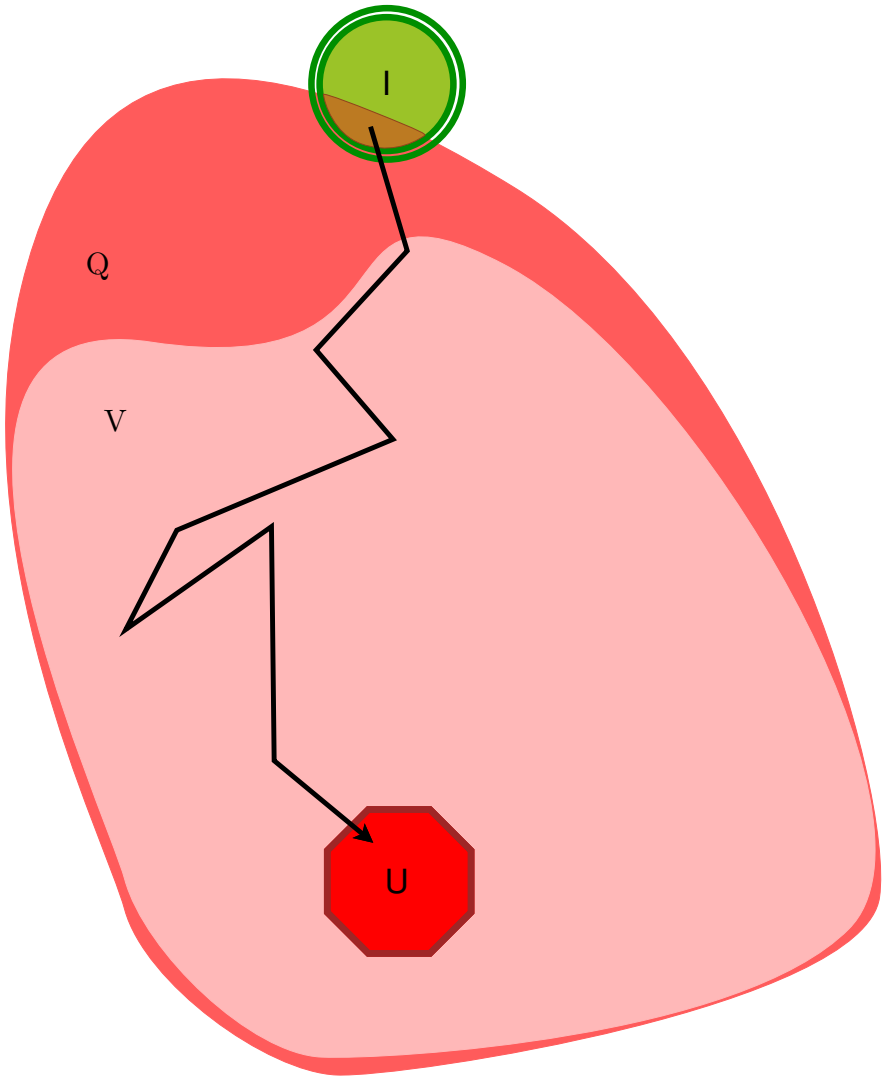
Algorithme d'atteignabilité arrière



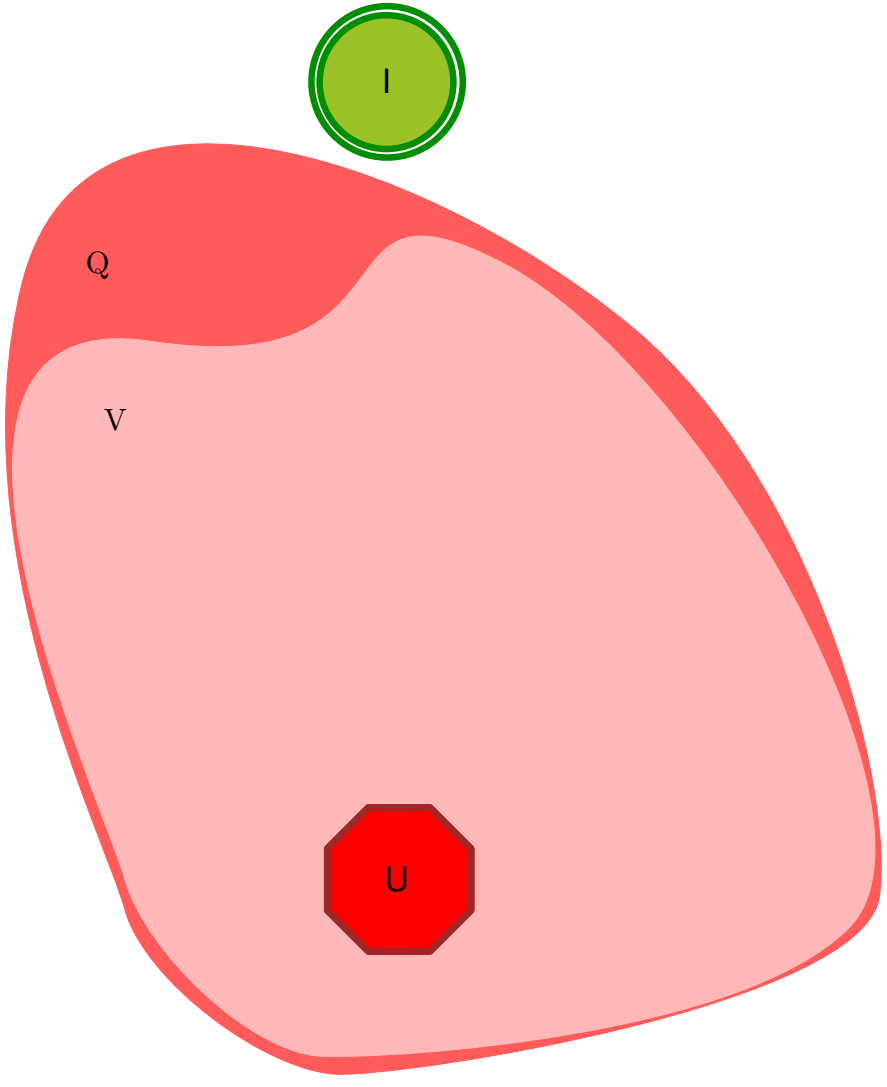
Algorithme d'atteignabilité arrière



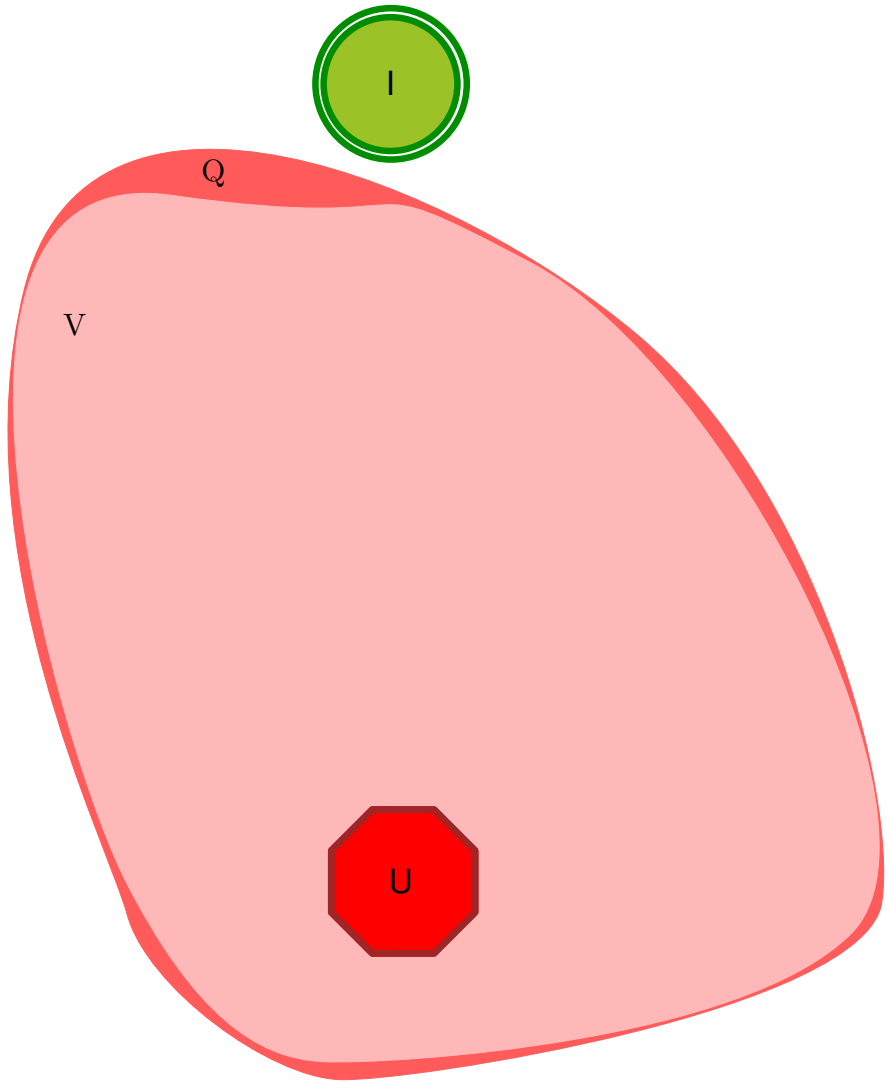
Algorithme d'atteignabilité arrière



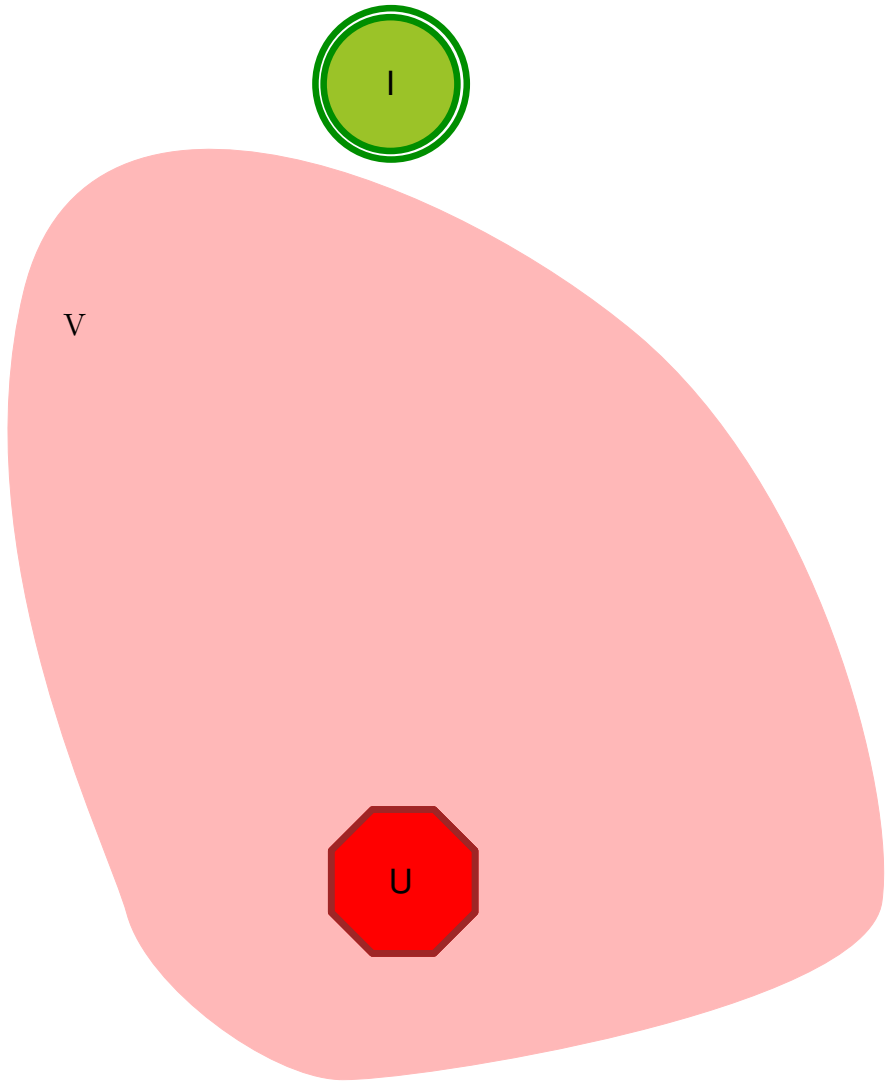
Algorithme d'atteignabilité arrière



Algorithme d'atteignabilité arrière



Algorithme d'atteignabilité arrière



Pre-images

Soit une transition $t \in \mathcal{T}$, $\texttt{pre}_t(\varphi)$ est une formule qui décrit l'ensemble des états à partir des quels un état φ est atteignable en un t-step.

$$\texttt{pre}_{\mathcal{T}}(\varphi) = \bigvee_{t \in \mathcal{T}} \texttt{pre}_t(\varphi)$$

Si φ est un cube, alors $\texttt{pre}_{\mathcal{T}}(\varphi)$ peut aussi être représentée comme une disjonction de cubes

Exemple d'analyse d'atteignabilité arrière

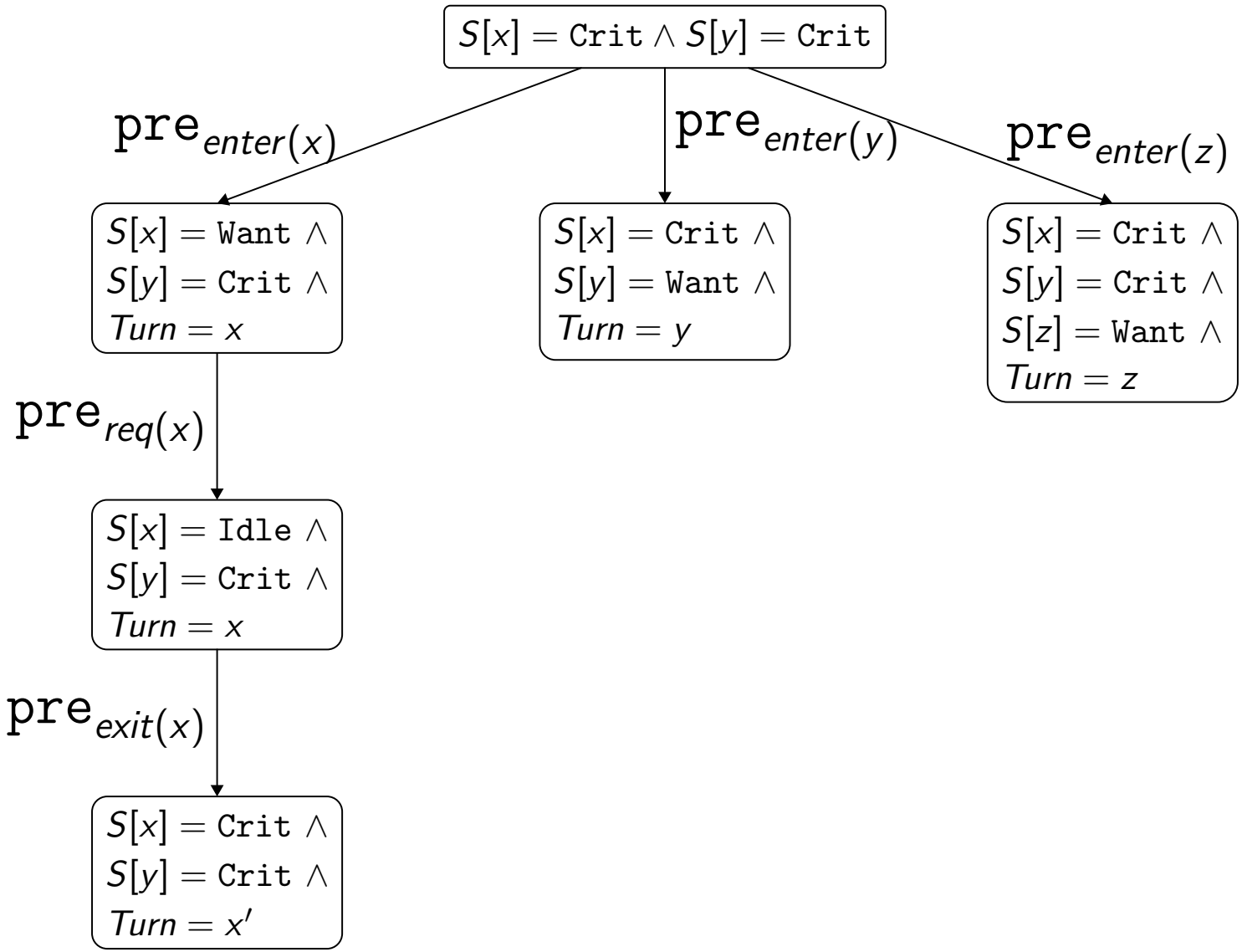
Rappel : algorithme du mutex

```
type st = Idle | Want | Crit
var Turn : proc
array S[proc] : st

init (z) { S[z] = Idle }

unsafe (x y) {
  S[x] = Crit && S[y] = Crit
}

transition req (i)
requires { S[i] = Idle }
{ S[i] := Want  }
```

```
transition enter (i)
requires { S[i] = Want &&
           Turn = i }
{ S[i] := Crit }
```

```
transition exit (i)
requires { S[i] = Crit }
{ Turn := . ;
  S[i] := Idle }
```

Exemple d'analyse d'atteignabilité arrière



Terminaison de l'analyse

La terminaison de l'analyse arrière n'est pas garantie en général, mais elle l'est sous certaines conditions

- étudions l'évolution de l'ensemble V et considérons la séquence

  $$V_0 \subseteq V_1 \subseteq V_1 \subseteq \ldots \subseteq V_n \subseteq \ldots$$

  où V_n représente l'ensemble V à la $n^{\text{ième}}$ itération de la boucle `while`
- pour que la boucle ne termine pas, il faut nécessairement que de nouveaux cubes soient ajoutés régulièrement dans V, *i.e.* qu'il existe une infinité d'ensembles de nœuds visités V_{k_i} tels que $V_{k_1} \subset V_{k_2} \subset \cdots$

On va définir les condiditions suffisantes pour que cette sous-séquence d'inclusions strictes soit finie

Terminaison: configurations

Soit $\mathcal{S}$ un système paramétré à tableaux

Une configuration de $\mathcal{S}$ est un état concret du système, *i.e.* un modèle pour

- les types
- les variables globales
- les tableaux du système

En particulier, une configuration doit fixer le nombre de processus du système, *i.e.* la cardinalité du type `proc`.

Un système à tableaux $\mathcal{S}$ a un nombre potentiellement infini de configurations.

Terminaison: pré-ordres et idéaux

Définition. un pré-ordre bien fondé $\preceq$ est une relation binaire réflexive et transitive sans suite infinie strictement décroissante

On note $[\![\varphi]\!]$ l'ensemble des configurations qui satisfont un cube φ

Si l'ensemble des configurations est muni d'un pré-ordre bien fondé $\preceq$, alors on peut montrer [Ghilardi&Ranise, LMCS 2010] que $[\![\varphi]\!]$ est un idéal, c'est-à-dire que

$$\text{si } s \in [\![\varphi]\!] \text{ et } s \preceq s' \text{ alors } s' \in [\![\varphi]\!]$$

Par extension, il est immédiat de montrer que chaque ensemble V_n de nœuds visités est également un idéal.

Terminaison: bel ordre

Définition. Un pré-ordre bien fondé $\preceq$ est un bel ordre si pour toute séquence infinie de configurations $s_1, s_2, \ldots$, il existe nécessairement $i < j$ tels que $s_i \preceq s_j$

Théorème. Si $\preceq$ est un bel ordre, alors toute séquence d'inclusions strictes d'idéaux $[\![V_{k_1}]\!] \subset [\![V_{k_2}]\!] \subset \cdots$ est finie.

De plus, $[\![.]\!]$ est monotone, *i.e.* si $V \subset V'$ alors $[\![V]\!] \subset [\![V']\!]$ donc la finitude de la séquence d'inclusion $[\![V_{k_1}]\!] \subset [\![V_{k_2}]\!] \subset \cdots$ implique bien la finitude de la séquence $V_0 \subseteq V_1 \subseteq \cdots \subseteq V_n \subseteq \cdots$

Terminaison: preuve du théorème

On raisonne par contradiction

- Si la séquence $[\![V_{k_1}]\!] \subset [\![V_{k_2}]\!] \subset \cdots$ est infinie, alors il existe également une séquence infinie de configurations $s_1, s_2, \ldots$ telle que $s_i \in [\![V_{k_i}]\!]$ et $s_i \notin [\![V_{k_j}]\!]$, pour tout $j < i$.
- De plus, $s_j \not\preceq s_i$, sinon $s_i \in [\![V_{k_j}]\!]$ puisque chaque $[\![V_{k_j}]\!]$ est un idéal.
- L'existence de cette suite infinie contredit donc l'hypothèse que $\preceq$ est un bel ordre

Terminaison: conditions suffisantes

Les conditions suffisantes pour que l'ensemble des configurations soit muni d'un bel ordre dépendent essentiellement du choix des opérations de comparaison sur le type `proc` et sur les types des éléments des tableaux

```
type t = A | B | C
array V[proc] : t
```

On peut exhiber un bel ordre grâce au lemme de Dickson :

$$(\mathbb{N}^k, \leq_k) \text{ est muni d'un bel ordre pour tout } k$$

Ainsi, la configuration `V[#1]=A, V[#2]=B, V[#3]=A, V[#4]=C` est encodée par le triplet d'entiers $(2, 1, 1)$

Comment développer un algorithme efficace ?

Optim 1 : Calcul des points-fixes

$\mathtt{V} \leftarrow \emptyset$
push_queue(Q, U)
while not_empty(Q)
  $\varphi \leftarrow$ pop_queue(Q)
  if $\neg(\varphi \wedge \mathtt{I} \vdash \bot)$ then return(*unsafe*)
  if $\neg(\varphi \models \bigvee_{\psi \in \mathtt{V}} \psi)$ then
    $\mathtt{V} \leftarrow \mathtt{V} \cup \{\varphi\}$
    push_queue(Q, $\mathtt{pre}_{\mathcal{T}}(\varphi)$)
return(*safe*)

Optim 1 : Calcul de point fixe

Le défi du calcul des points fixes

$$\varphi \models \bigvee_{\psi \in \mathtt{V}} \psi$$

Optim 1 : Calcul de point fixe

Le défi du calcul des points fixes

$$\exists \bar{x}.F \models \bigvee_{\psi \in \mathbf{V}} \exists \bar{y}.G_\psi$$

Optim 1 : Calcul de point fixe

Le défi du calcul des points fixes

$$\not\models F \wedge \bigwedge_{\psi \in \mathtt{V}} \forall \bar{y}. \neg G_{\psi}$$

Optim 1 : Calcul de point fixe

Le défi du calcul des points fixes

$$F \models \bigvee_{\psi \in \mathtt{V}} \bigvee_{\sigma \in \Sigma} (G_\psi)\sigma$$

Optim 1 : Calcul de point fixe

Le défi du calcul des points fixes

$$F \models \bigvee_{\psi \in \mathtt{V}} \bigvee_{\sigma \in \Sigma} (G_\psi)\sigma$$

Supposons $|\mathtt{V}| = 20000$ et $|\Sigma| = 120$ (e.g. $|\bar{x}| = |\bar{y}| = 5$) alors

$$|\bigvee_{\psi \in \mathtt{V}} \bigvee_{\sigma \in \Sigma} (G_\psi)\sigma| \sim 2.10^6 \text{clauses}$$

Optim 1 : Calcul des points-fixes

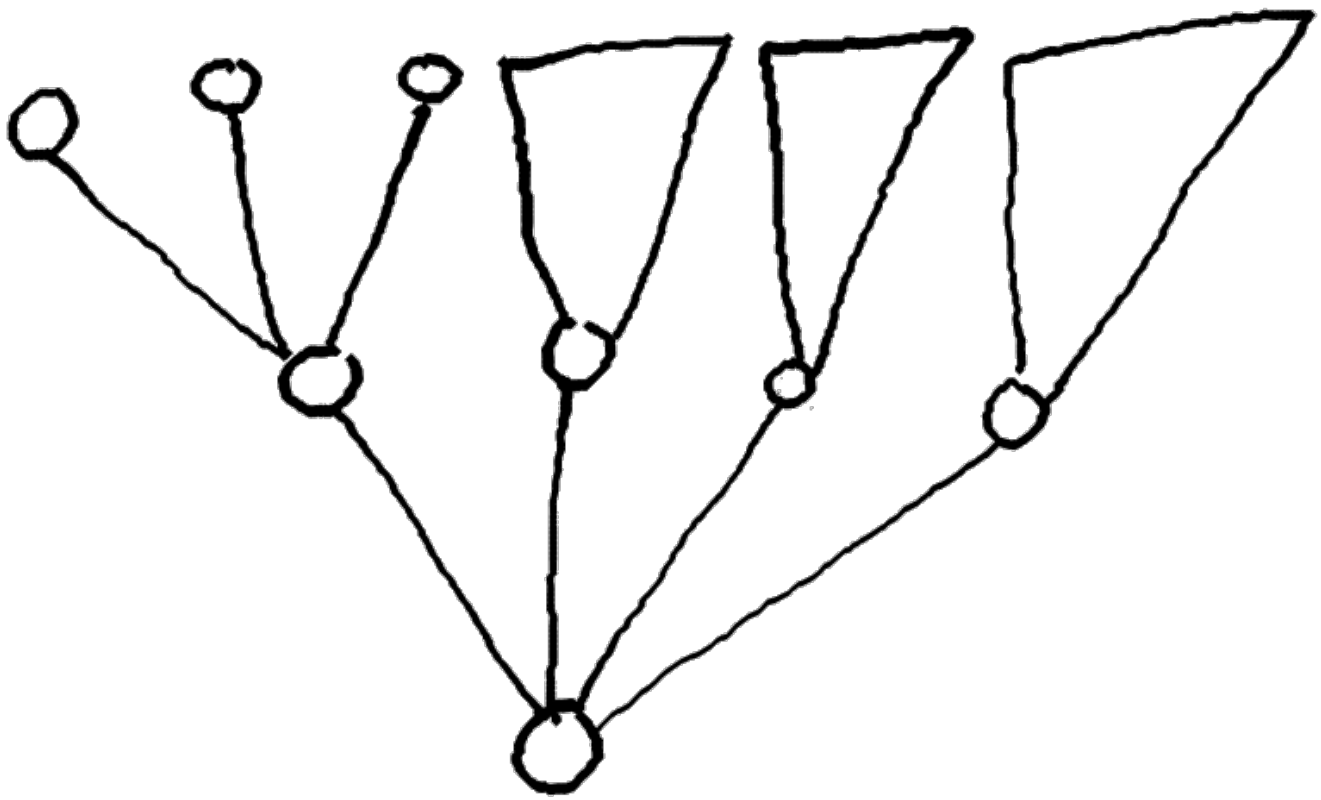
- Tests rapides : $G_\psi\sigma \subseteq F$
- Permutations non pertinentes :

  $L \in G_\psi\sigma$ et $L' \in F$ et $\neg(L \wedge L')$ est immédiat
- Un unique contexte pour le solveur SMT est utilisé pour chaque test de point-fixe; il est juste incrémenté et sa cohérence est vérifiée

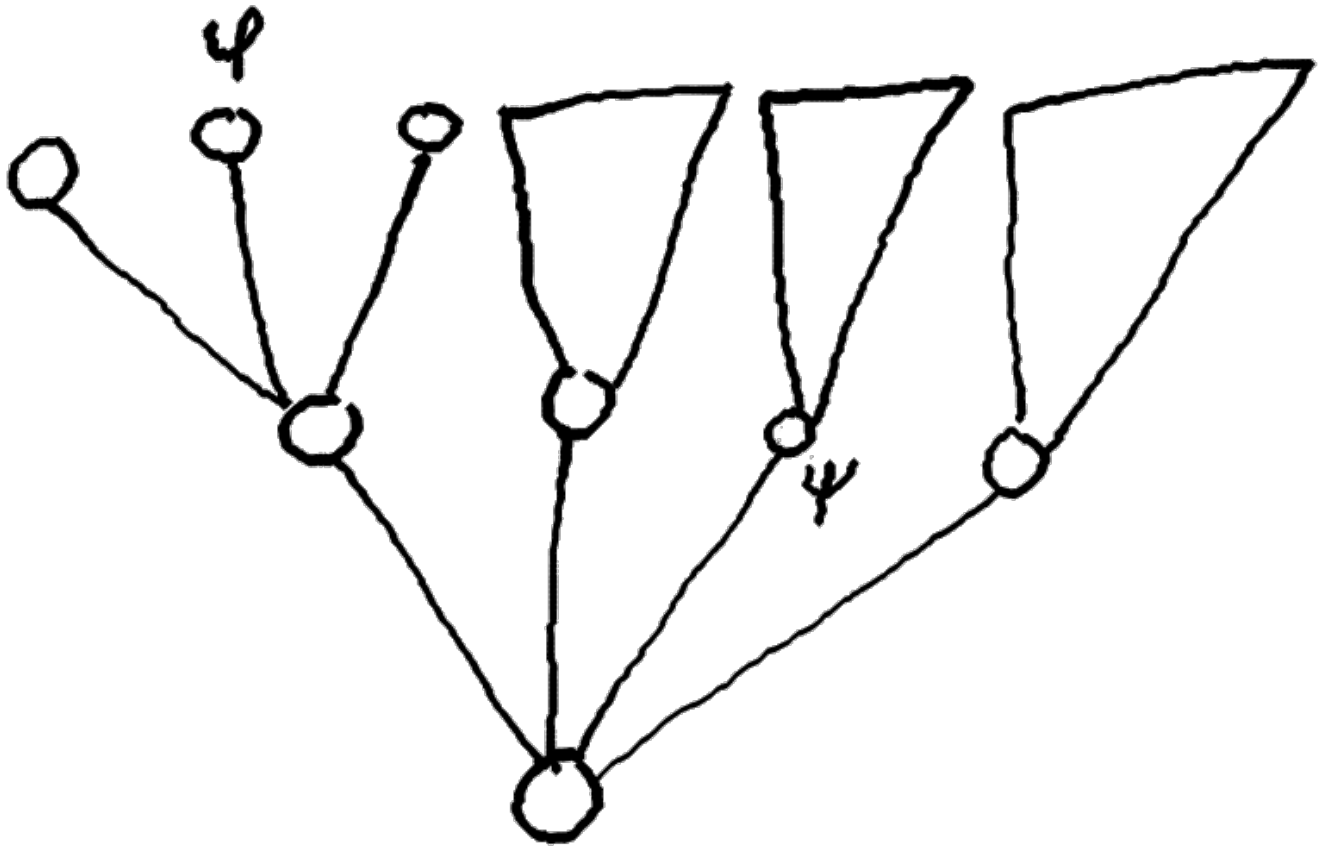
Optim 2 : Suppression de nœuds

```
push_queue(Q, U)
while not_empty(Q)
  φ ← pop_queue(Q)
  if ¬(φ ∧ I ⊢ ⊥) then return(unsafe)
  if ¬(φ ⊢ ⋁_{ψ∈V} ψ) then
    V ← V ∪ {φ}
    push_queue(Q, pre(φ))
return(safe)
```

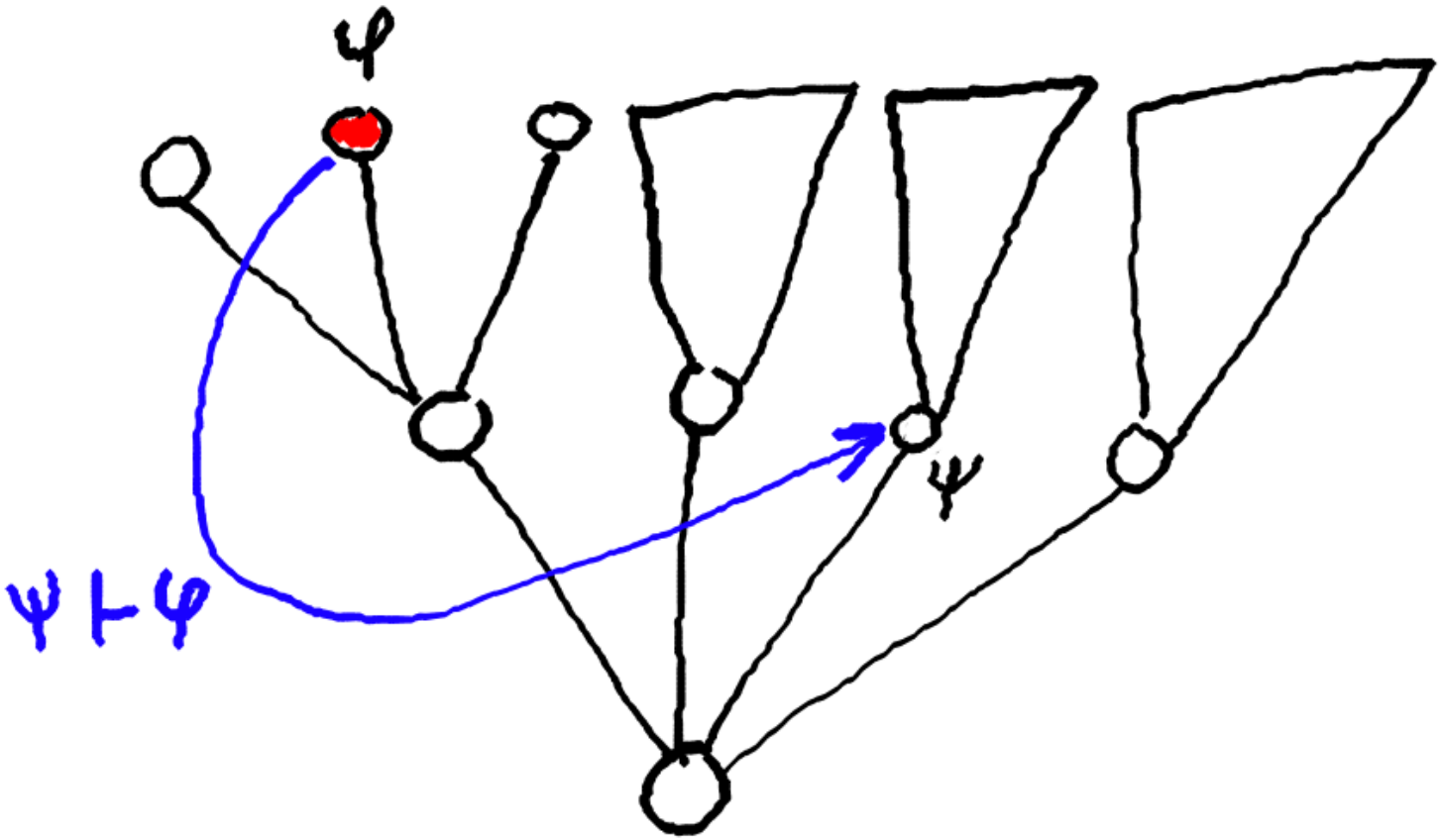
Suppression de nœuds : backward simplification



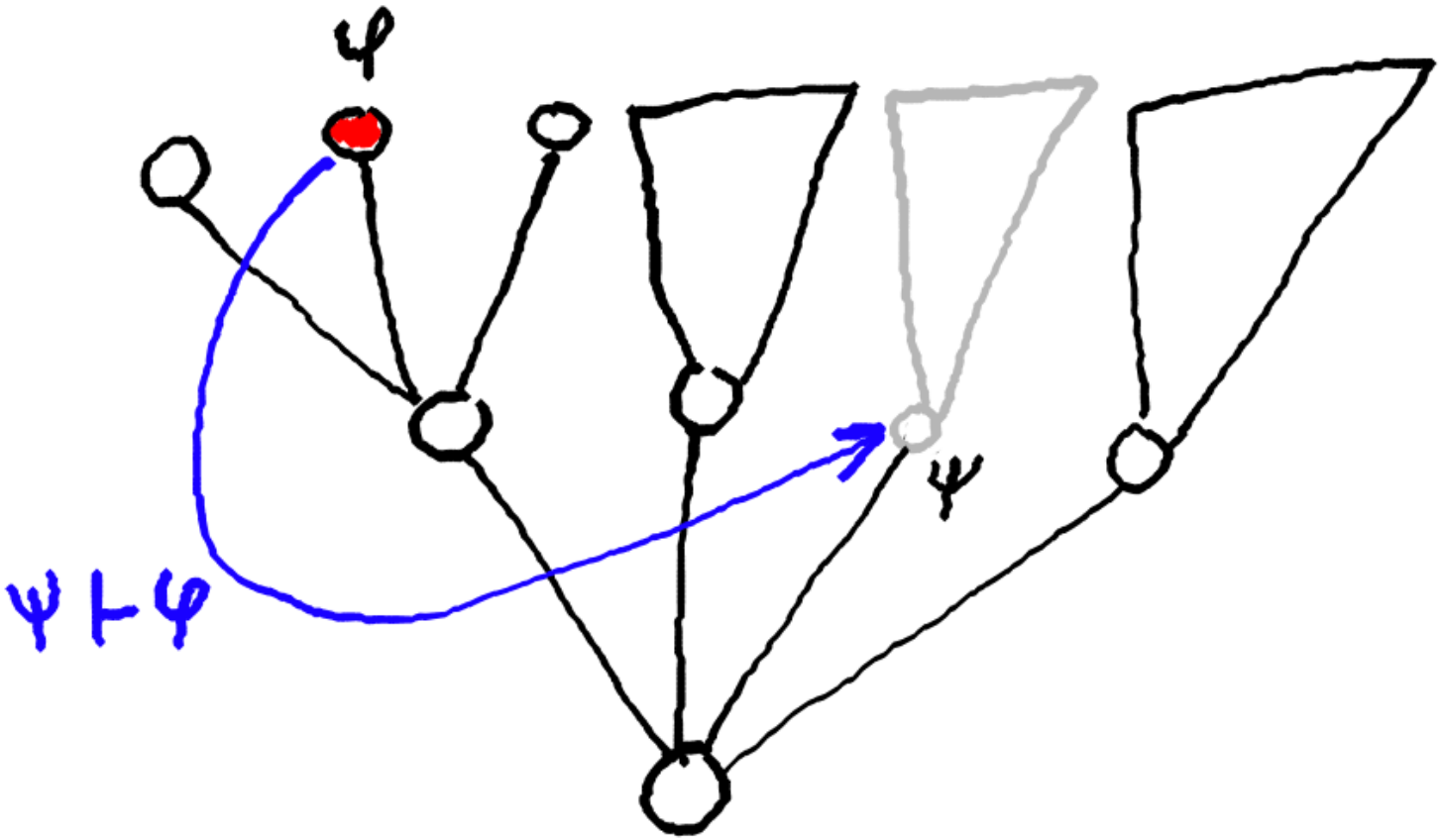
Suppression de nœuds : backward simplification



Suppression de nœuds : backward simplification



Suppression de nœuds : backward simplification



Optim 3 : Calcul de la pré-image

```
push_queue(Q, U)
while not_empty(Q)
  φ ← pop_queue(Q)
  if ¬(φ ∧ I ⊢ ⊥) then return(unsafe)
  if ¬(φ ⊢ ⋁_{ψ∈V} ψ) then
    V ← V ∪ {φ}
    push_queue(Q, pre(φ))
return(safe)
```

Optim 3 : Calcul de la pré-image

- Les cubes sont maintenus sous forme normale (renomage des variables, littéraux impliqués supprimés etc.) pour des tests de redondances et de sous-ensembles efficaces
- Des simplifications arithmétiques sont appliquées pour réduire le nomber d'inégalités
- Différentes stratégies DFS ou BFS sont combinées et certains traitements sont différés de manière heuristiques pour donner la priorité aux cubes les plus génériques

Optim 4 : Invariants de sous-Typage

(supportés nativement par le solveur SMT de Cubicle)

```
type t = A | B | C | D
var X : t
var Y : t

init () { X = A && Y = A }

transition t1 ()
requires {...}
{ X := B; Y := D }

transition t2 ()
requires {...}
{ X := A; Y := X }
```

Optim 4 : Invariants de sous-Typage

(supportés nativement par le solveur SMT de Cubicle)

```
type t = A | B | C | D
var X : t                                   τ_X <: {A, B, C, D}
var Y : t

init () { X = A && Y = A }

transition t1 ()
requires {...}
{ X := B; Y := D }

transition t2 ()
requires {...}
{ X := A; Y := X }
```

Optim 4 : Invariants de sous-Typage

(supportés nativement par le solveur SMT de Cubicle)

```
type t = A | B | C | D
var X : t                          τ_X <: {A, B, C, D}
var Y : t                          τ_Y <: {A, B, C, D}

init () { X = A && Y = A }

transition t1 ()
requires {...}
{ X := B; Y := D }

transition t2 ()
requires {...}
{ X := A; Y := X }
```

Optim 4 : Invariants de sous-Typage

(supportés nativement par le solveur SMT de Cubicle)

```
type t = A | B | C | D
var X : t
var Y : t

init () { X = A && Y = A }

transition t1 ()
requires {...}
{ X := B; Y := D }

transition t2 ()
requires {...}
{ X := A; Y := X }
```

$\tau_X <: \{A, B, C, D\}$

$\tau_Y <: \{A, B, C, D\}$

$\tau_X :> \{A\} \wedge \tau_Y :> \{A\}$

Optim 4 : Invariants de sous-Typage

(supportés nativement par le solveur SMT de Cubicle)

```
type t = A | B | C | D
var X : t                                  τX <: {A,B,C,D}
var Y : t                                  τY <: {A,B,C,D}

init () { X = A && Y = A }                 τX :> {A} ∧ τY :> {A}

transition t1 ()
requires {...}
{ X := B; Y := D }                         τX :> {B} ∧ τY :> {D}

transition t2 ()
requires {...}
{ X := A; Y := X }
```

Optim 4 : Invariants de sous-Typage

(supportés nativement par le solveur SMT de Cubicle)

```
type t = A | B | C | D
var X : t                                  τX <: {A,B,C,D}
var Y : t                                  τY <: {A,B,C,D}

init () { X = A && Y = A }                 τX :> {A} ∧ τY :> {A}

transition t1 ()
requires {...}
{ X := B; Y := D }                         τX :> {B} ∧ τY :> {D}

transition t2 ()
requires {...}
{ X := A; Y := X }                         τX :> {A} ∧ τY :> τX
```

Optim 4 : Invariants de sous-Typage

(supportés nativement par le solveur SMT de Cubicle)

```
type t = A | B | C | D
var X : t                                  τX <: {A,B,C,D}
var Y : t                                  τY <: {A,B,C,D}

init () { X = A && Y = A }                 τX :> {A} ∧ τY :> {A}

transition t1 ()
requires {...}
{ X := B; Y := D }                         τX :> {B} ∧ τY :> {D}

transition t2 ()
requires {...}
{ X := A; Y := X }                         τX :> {A} ∧ τY :> τX
```

$\tau_X = \{A, B\} \wedge \tau_Y = \{A, B, D\}$

En pratique

Voici l'effet de quelques optimisations sur une version du protocole de cohérence de cache de Steve German [Baukus et. al, VMCAI 2002]

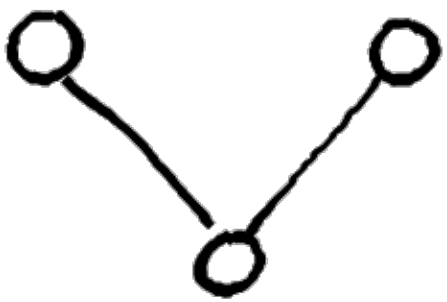
Optimizations			Real Time (# nodes)
permut.	delete	subtyping	
No	No	No	∞
Yes	No	No	50m8s (22580)
Yes	Yes	No	35m16s (20405)
Yes	Yes	Yes	25.0s (3322)

Vers une implémentation concurrente

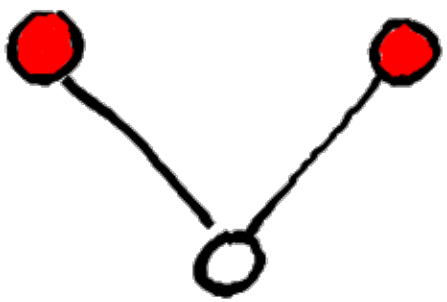
- Basé sur la bibliothèque Functory [Filliâtre, Krishnamani] : architecture maître / esclave

- La recherche peut être parallelisée :
  - Tâches coûteuses = tests de points-fixes
  - Synchronisation pour garder un guidage précis (BFS)
  - La suppression devient dangereuse

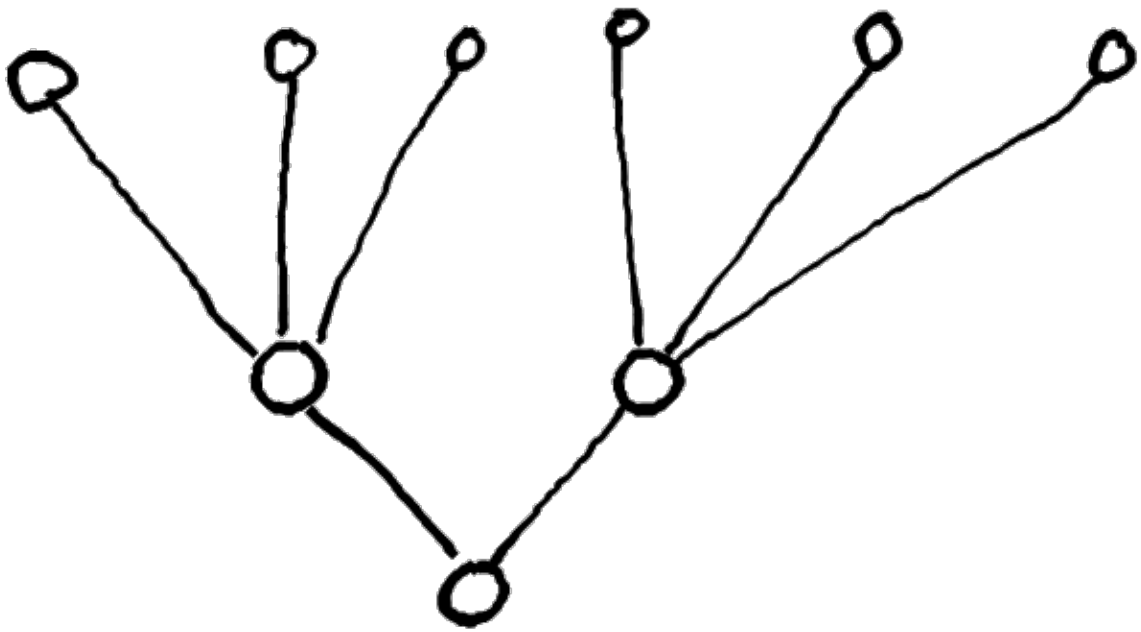
Suppression de nœuds dans un BFS parallèle



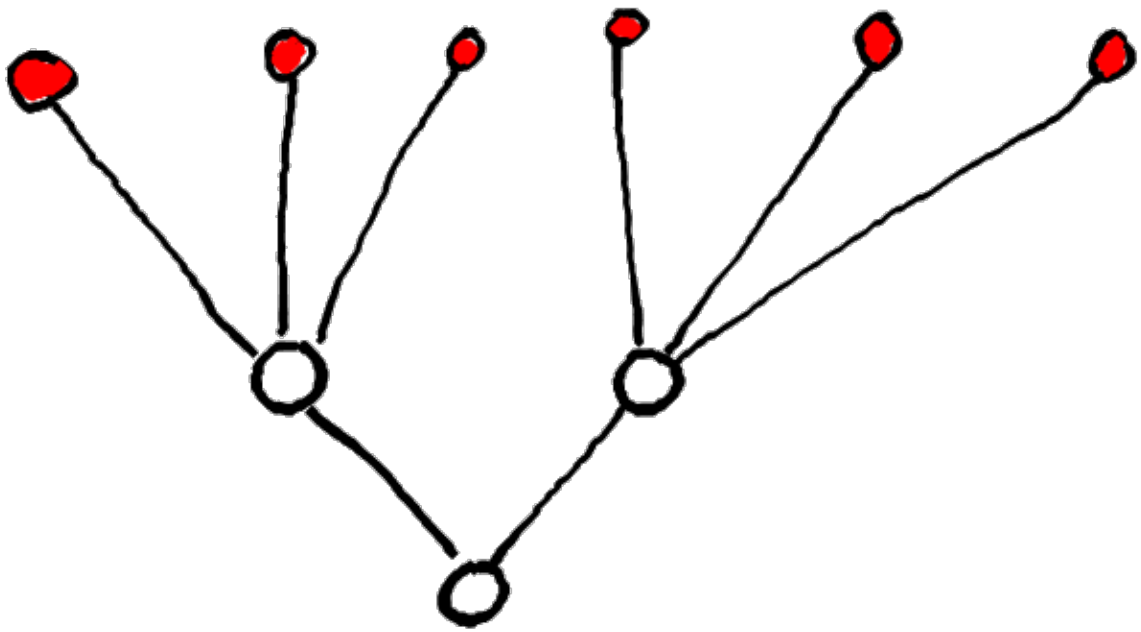
Suppression de nœuds dans un BFS parallèle



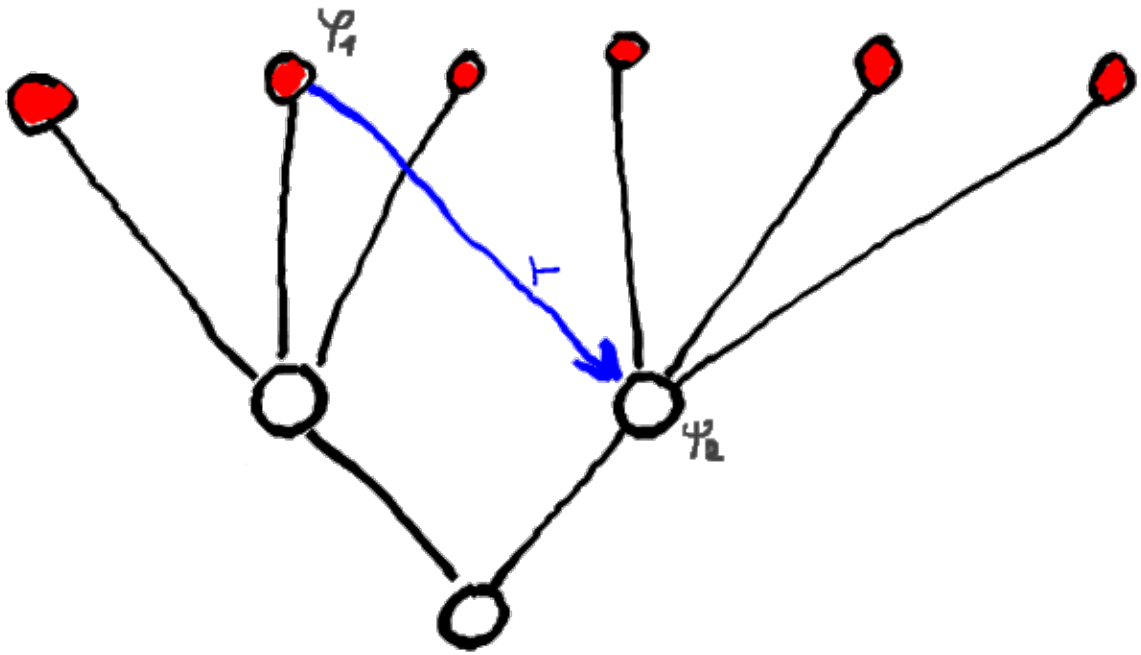
Suppression de nœuds dans un BFS parallèle



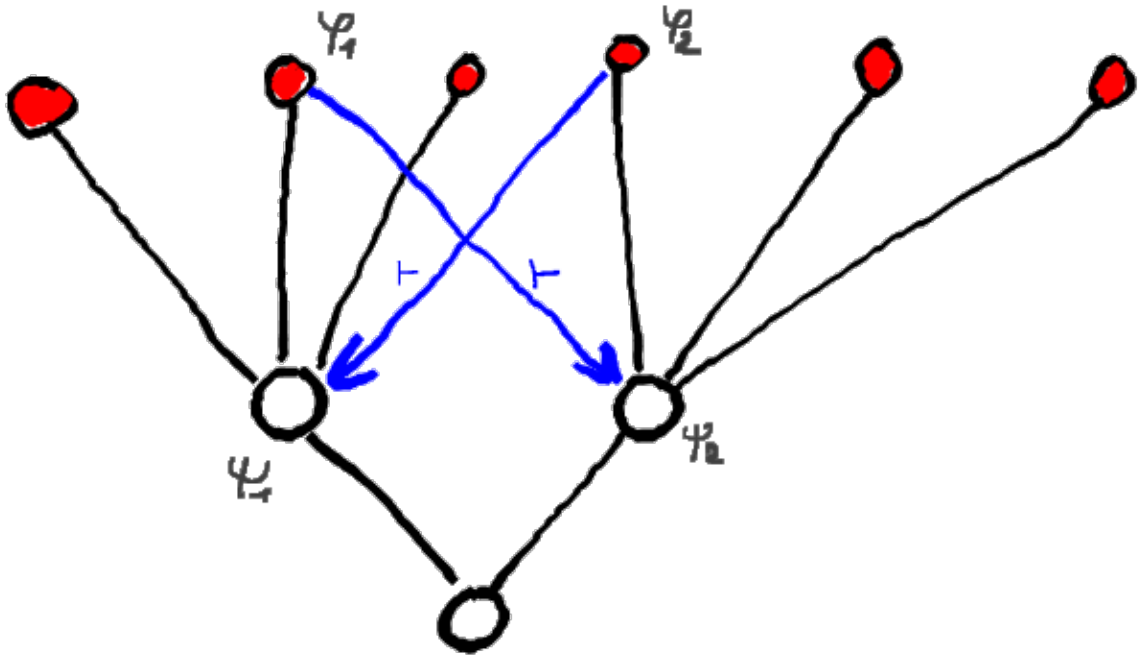
Suppression de nœuds dans un BFS parallèle



Suppression de nœuds dans un BFS parallèle



Suppression de nœuds dans un BFS parallèle



D'autres examples

Évaluation de Cubicle sur des algorithmes et protocoles classiques et plus difficiles

- Algorithmes d'exclusion mutuelle : Bakery, Dijkstra, Distributed Lamport, Ricart-Agrawala, Szymanski etc.
- Protocole de cohérence de cache : Berkley, Illinois, MESI, MOESI, German etc.

Ces exemples ont été tirés pour la plupart de distributions des outils MCMT (Ghilardi & Ranise), PFS (Ben Henda, Rezine), Undip (Rezine)

Benchmarks

	Cubicle		MCMT	Undip	PFS
	seq	4 cores			
bakery	0.01s	-	0.01s	0.04s	0.01s
Dijkstra	0.24s	-	0.99s	**0.04s**	0.26s
Distrib_Lamport	**2.3s**	-	12.7s	unsafe	X
Java_Mlock	0.04s	-	0.06s	0.25s	0.02s
Ricart_Agrawala	**1.8s**	-	1m12s	4.3s	X
Szymanski_at	**0.12s**	-	0.71s	13.5s	timeout
Berkeley	0.01s	-	0.01s	0.01s	0.01s
German_Baukus	**25.0s**	17.1s	3h39m	9m43s	X
German_pfs	**6m23s**	3m8s	11m31s	timeout	47m22s
German_undip	**0.17s**	-	0.57s	1m32	X
Illinois	0.02s	-	0.04s	0.06s	0.06s
Moesi	0.01s	-	0.01s	0.01s	0.01s

Lectures pour aller plus loin

Le Model Checking Modulo Theories a été initialement proposé par Silvio Ghilardi et Silvio Ranise

- `http://users.mat.unimi.it/users/ghilardi/mcmt/`

En particulier, lire

Backward Reachability of Array-based Systems by SMT Solving: Termination and Invariant Synthesis [LMCS, vol.6, n.4, 2010]

Plus d'informations sur Cubicle

- Invariants for Finite Instances and Beyond [FMCAD 2013]
- Cubicle: A Parallel SMT-based Model Checker for Parameterized Systems [CAV 2012]