Génération efficace de grands espaces d'états

Alexandre Hamez

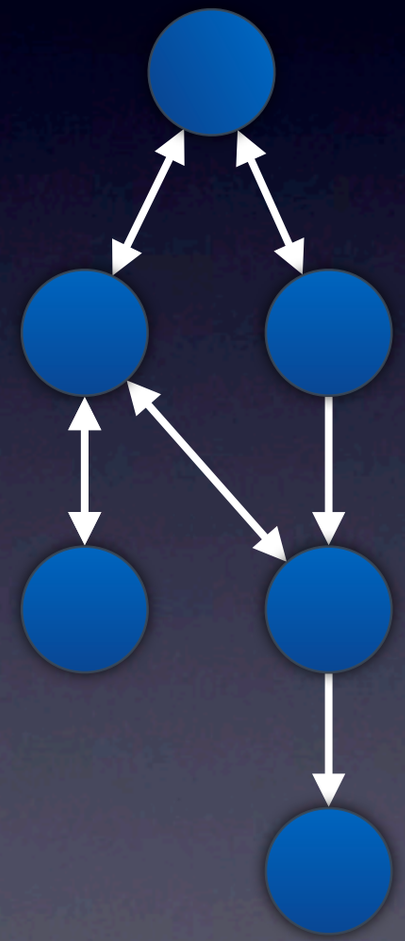
Fabrice Kordon	Directeur	UPMC
Yann Thierry-Mieg	Encadrant	UPMC
Didier Buchs	Rapporteur	Université de Genève
Patrice Moreaux	Rapporteur	Université de Savoie
Béatrice Bérard	Examinateur	UPMC
Alexandre Duret-Lutz	Examinateur	EPITA
François Vernadat	Examinateur	INSA Toulouse

Problématique

- Nécessité de vérification
  - Systèmes complexes
  - Systèmes répartis
- Vérification formelle
  - Nombreuses techniques
- Model checking
  - Méthode automatique
  - Génération exhaustive de l'espace d'états d'un système
  - Vérification de propriétés

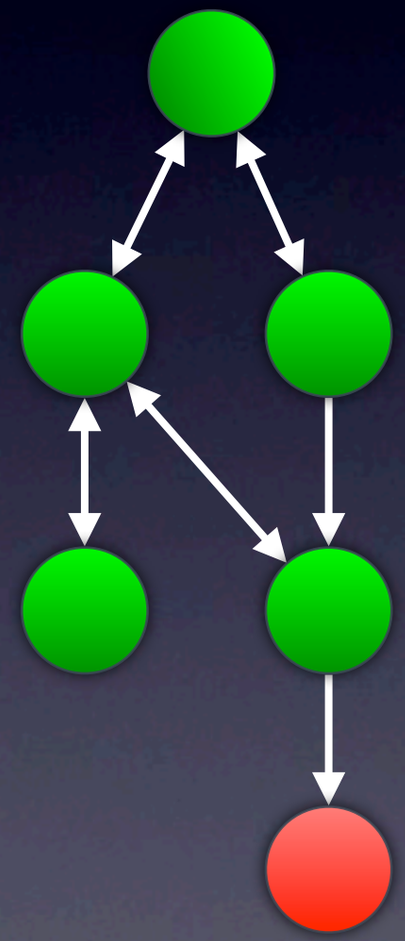
Problématique

- Nécessité de vérification
  - Systèmes complexes
  - Systèmes répartis
- Vérification formelle
  - Nombreuses techniques
- Model checking
  - Méthode automatique
  - Génération exhaustive de l'espace d'états d'un système
  - Vérification de propriétés



Problématique

- Nécessité de vérification
  - Systèmes complexes
  - Systèmes répartis
- Vérification formelle
  - Nombreuses techniques
- Model checking
  - Méthode automatique
  - Génération exhaustive de l'espace d'états d'un système
  - Vérification de propriétés



Problématique

- Nécessité de vérification
  - Systèmes complexes
  - Systèmes répartis
- **Explosion combinatoire**
- Model checking
  - Méthode automatique
  - Génération exhaustive de l'espace d'états d'un système
  - Vérification de propriétés

Contributions : réduire les effets de l'explosion combinatoire

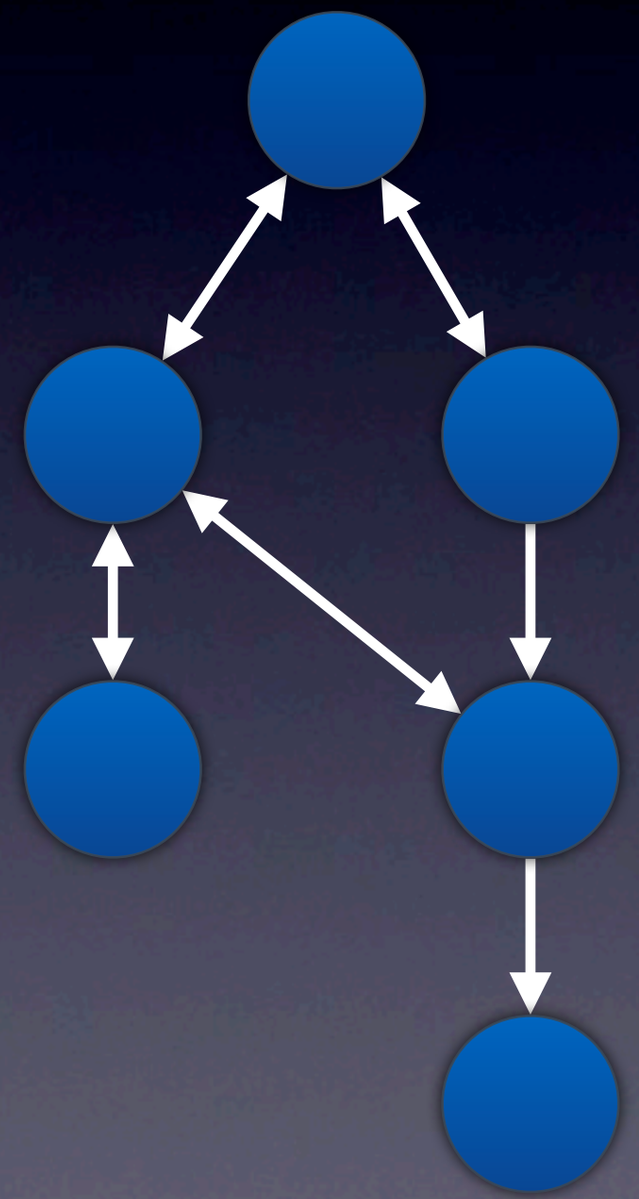
- Ordres partiels, abstractions, graphes quotients, approches symboliques, répartition et parallélisation, …

1. Model checking sur cluster et machines multi-processeurs
   - **Répartition et parallélisation**
   - [POHLL 07], [Petri Nets 07]
2. Model checking symbolique
   - **Saturation automatique**
   - [Petri Nets 08], [Fundamenta Informaticae 09], [TACAS 09]

Répartition et parallélisme

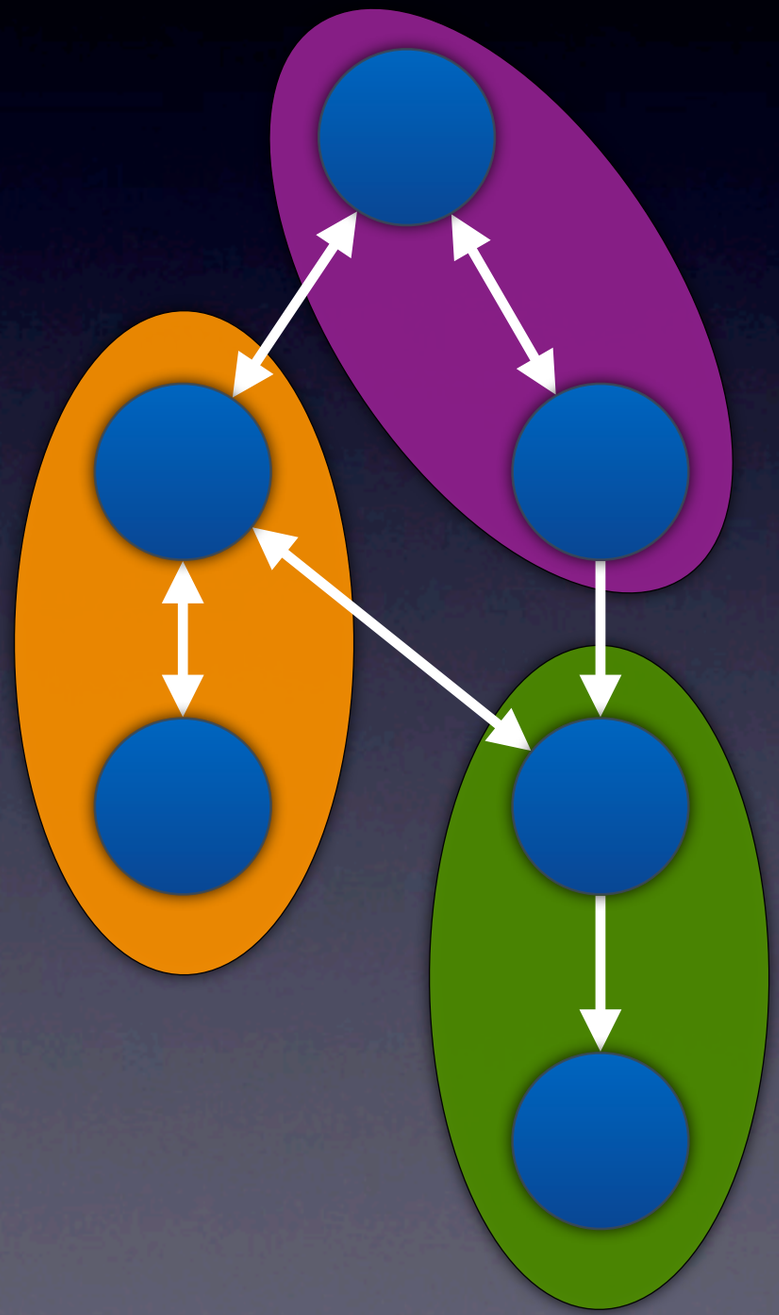
Répartition et parallélisation : enjeux et problématiques

- Model checkers : algorithmes évolués
  - Réutilisation
- Accélération linéaire
  - Équilibrage de charge

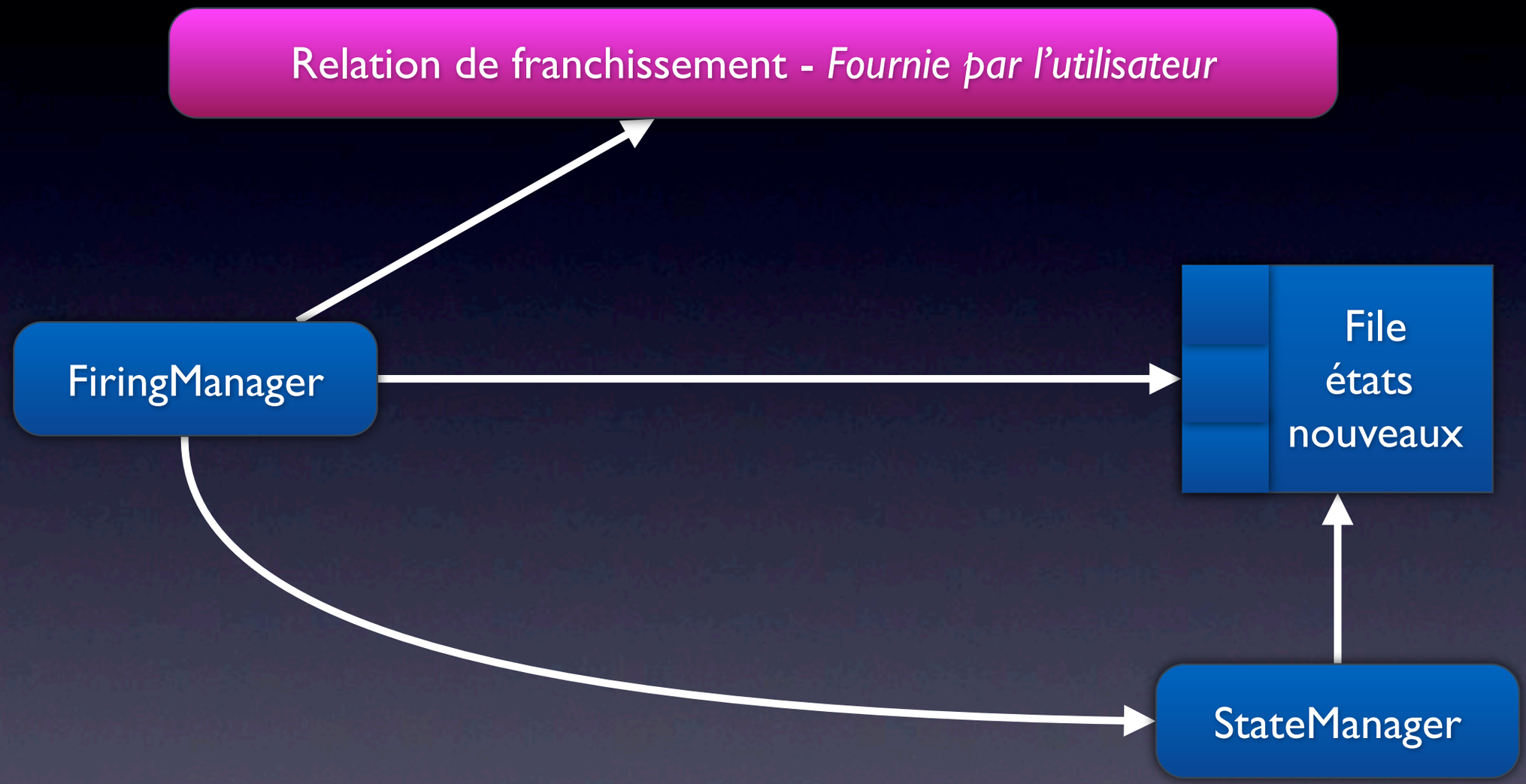


Répartition et parallélisation : enjeux et problématiques

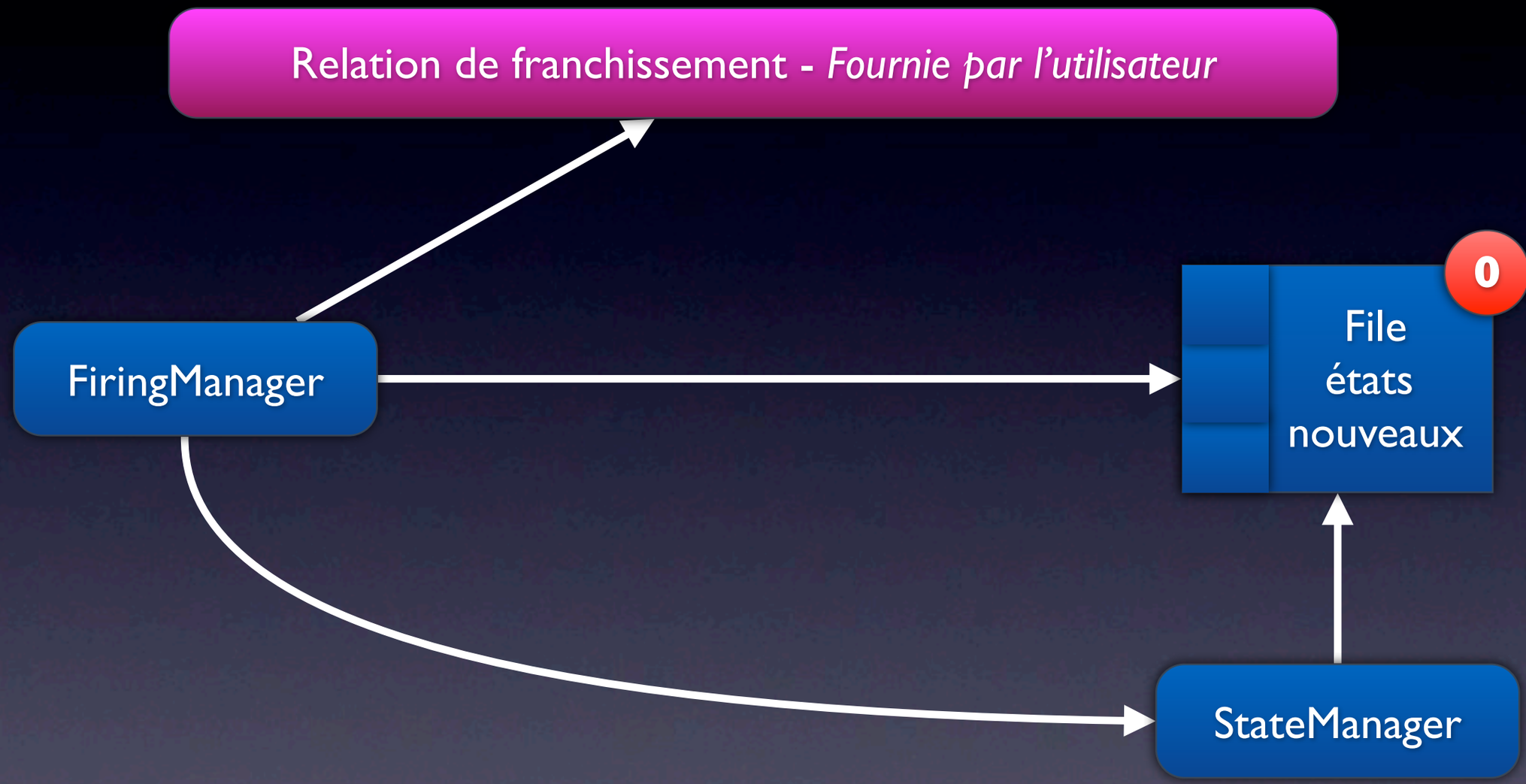
- Model checkers : algorithmes évolués
  - Réutilisation
- Accélération linéaire
  - Équilibrage de charge



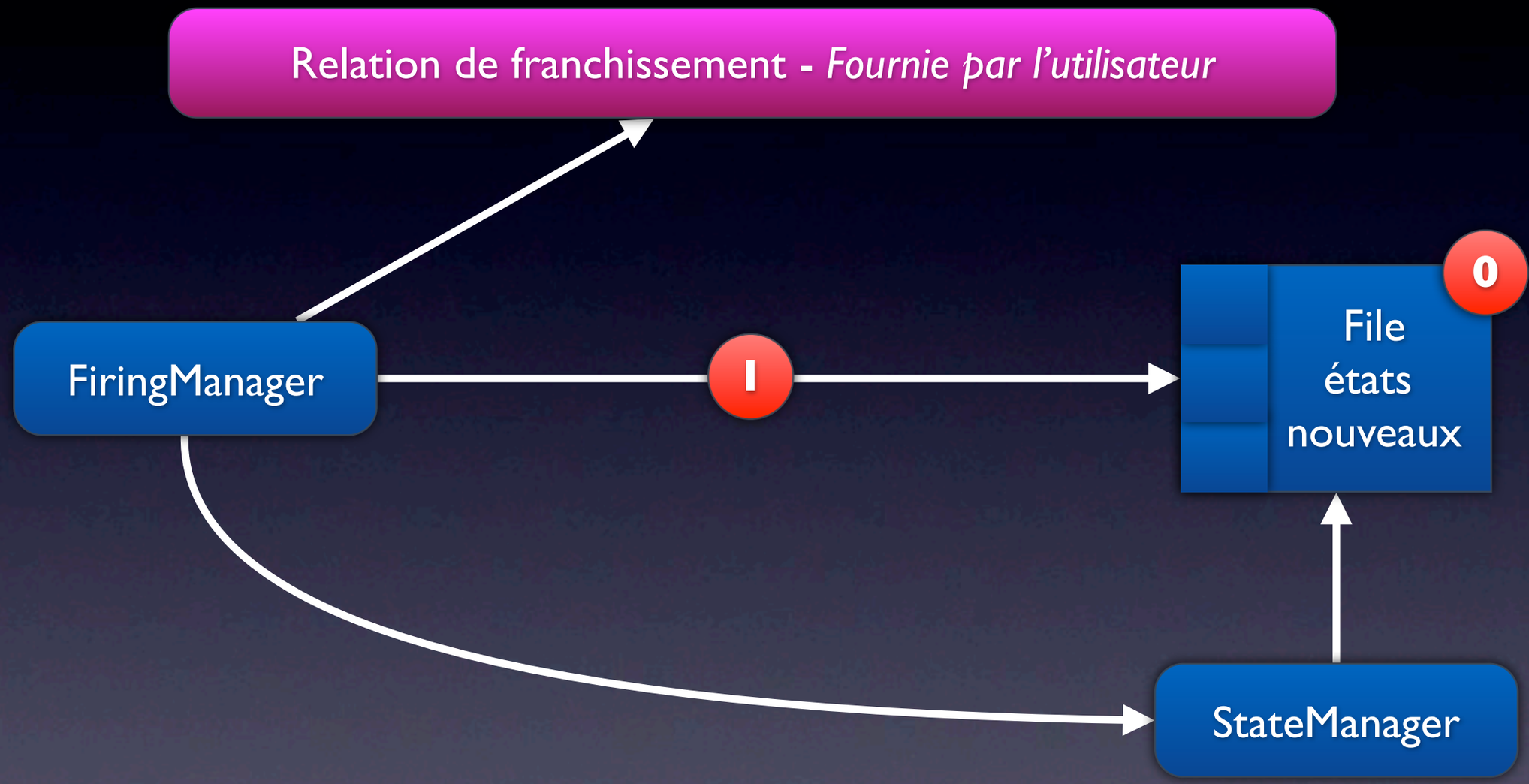
Architecture pour le model checking réparti et parallèle : libdmc



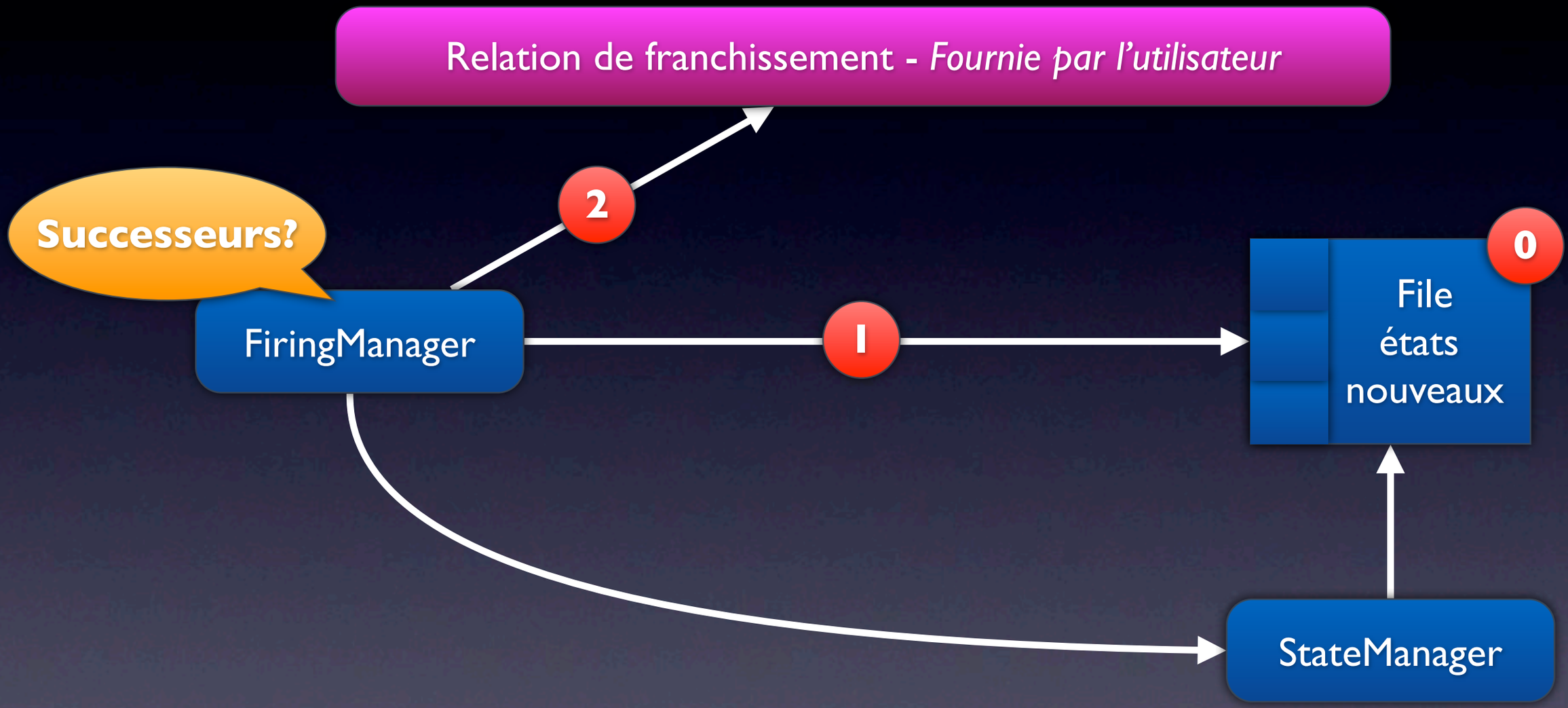
Architecture pour le model checking réparti et parallèle : libdmc



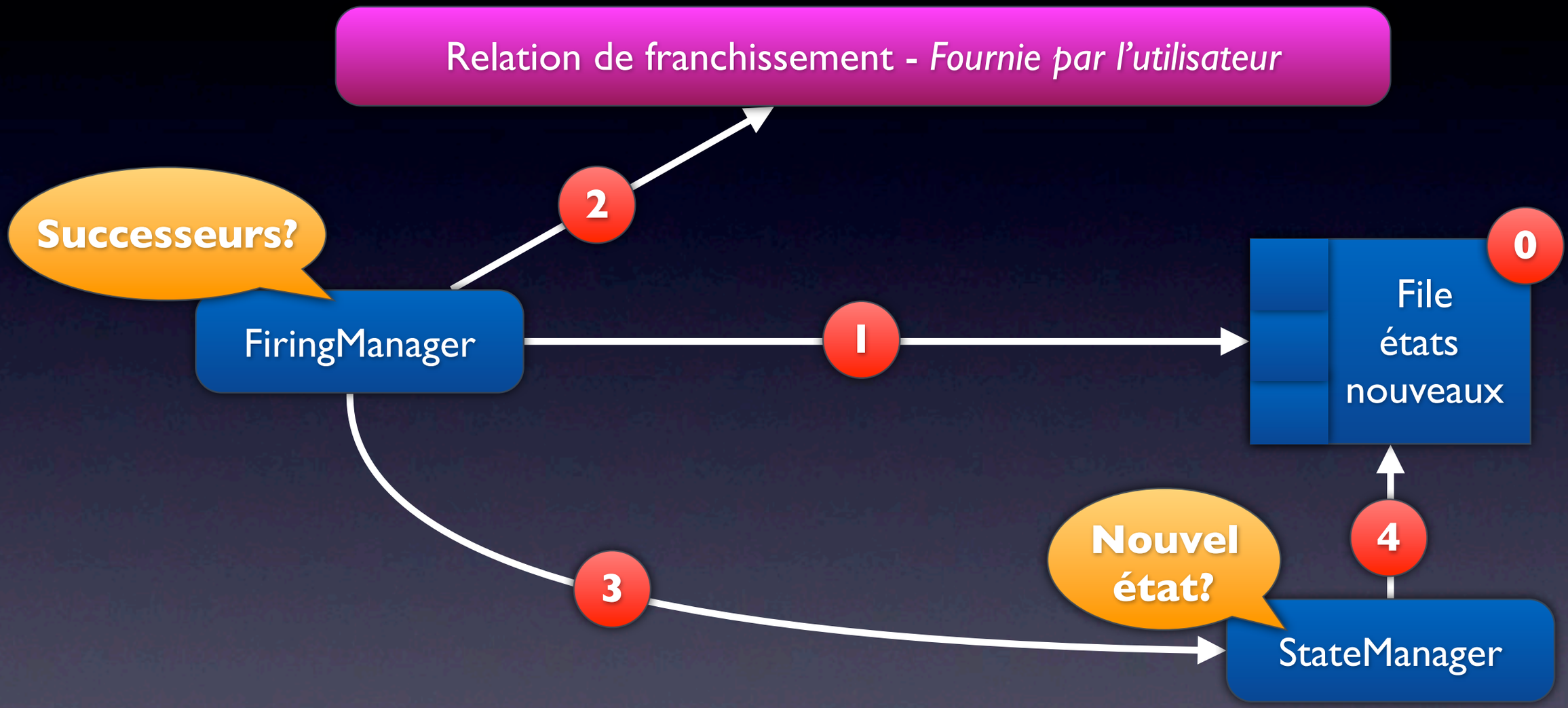
Architecture pour le model checking réparti et parallèle : libdmc



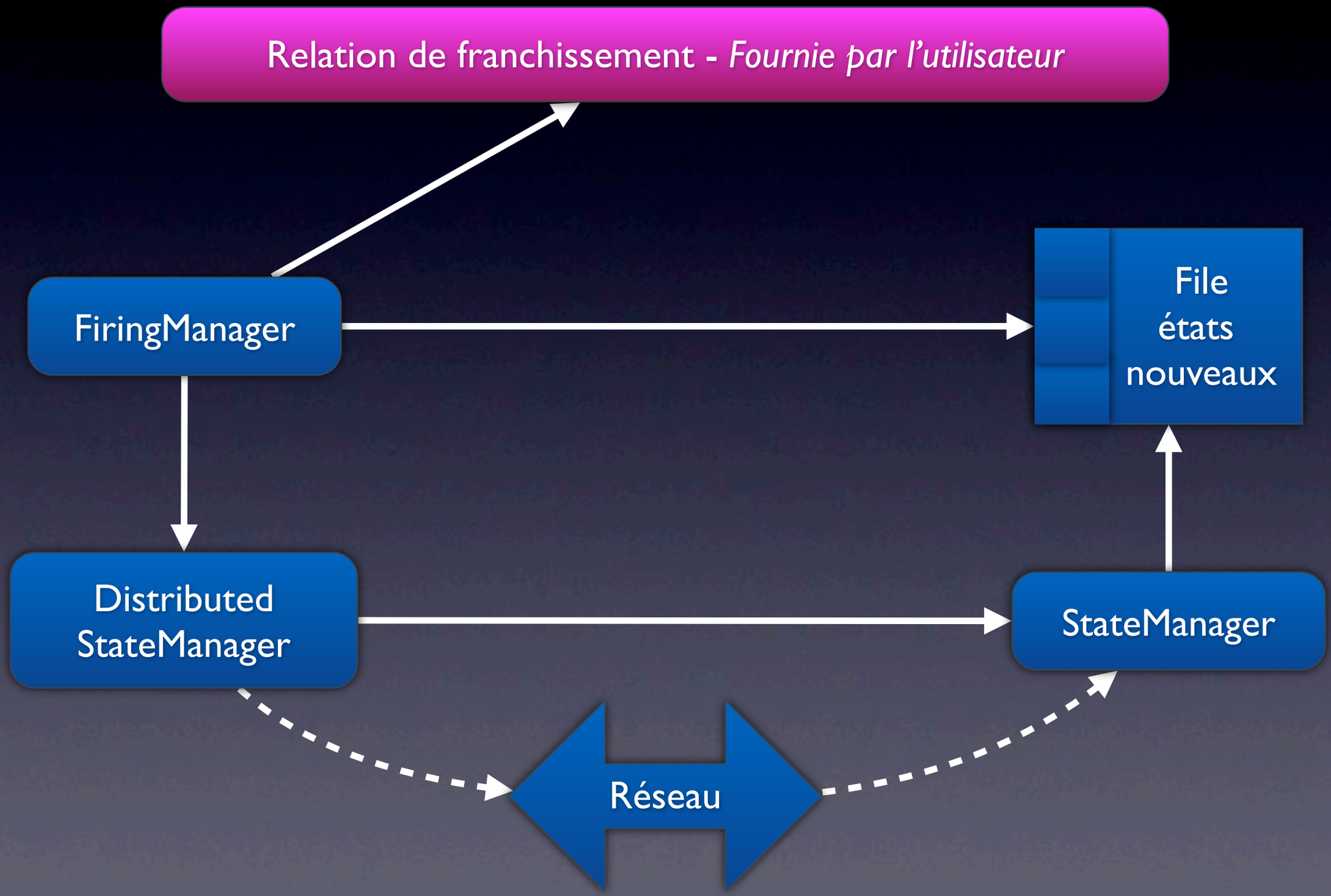
Architecture pour le model checking réparti et parallèle : libdmc



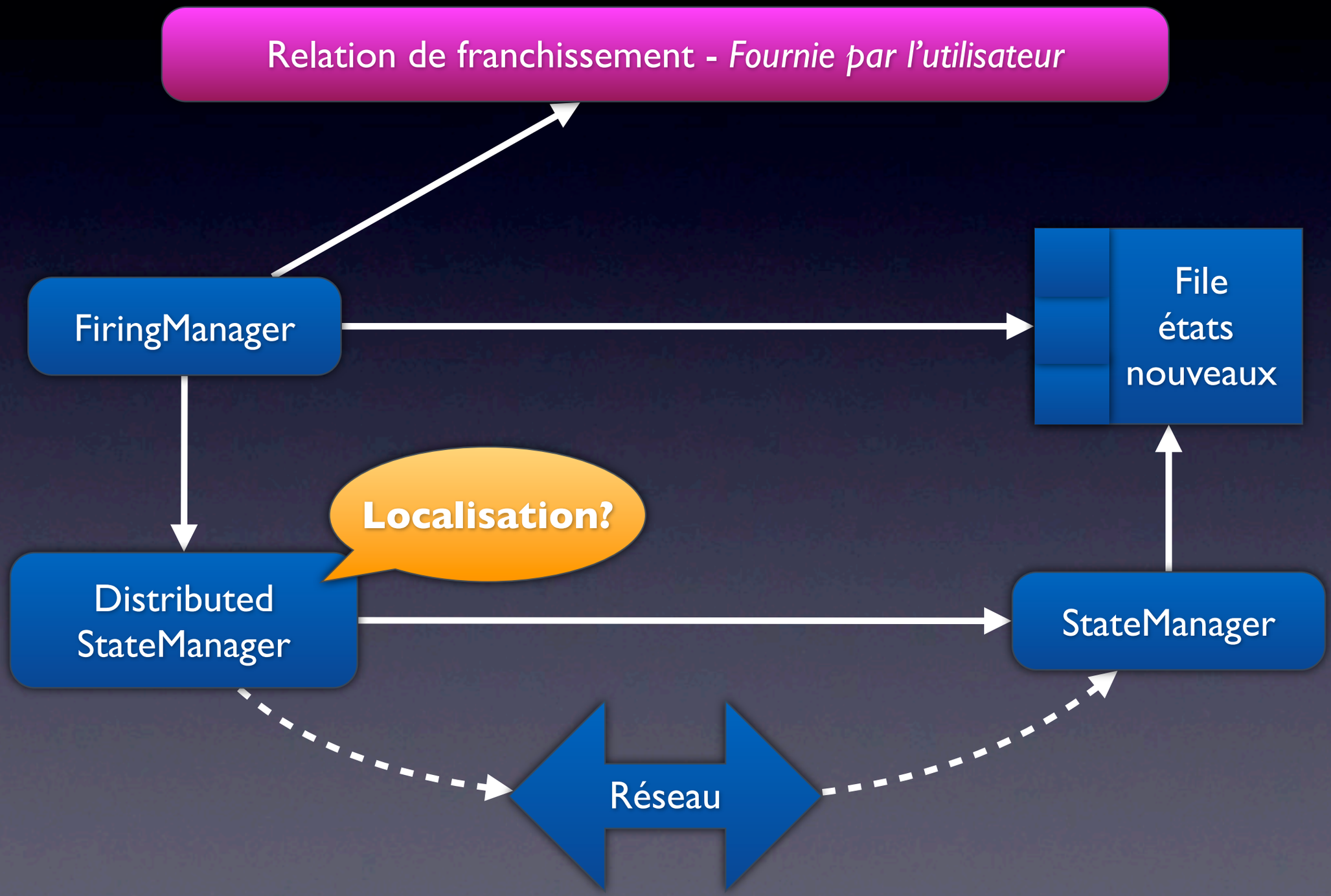
Architecture pour le model checking réparti et parallèle : libdmc



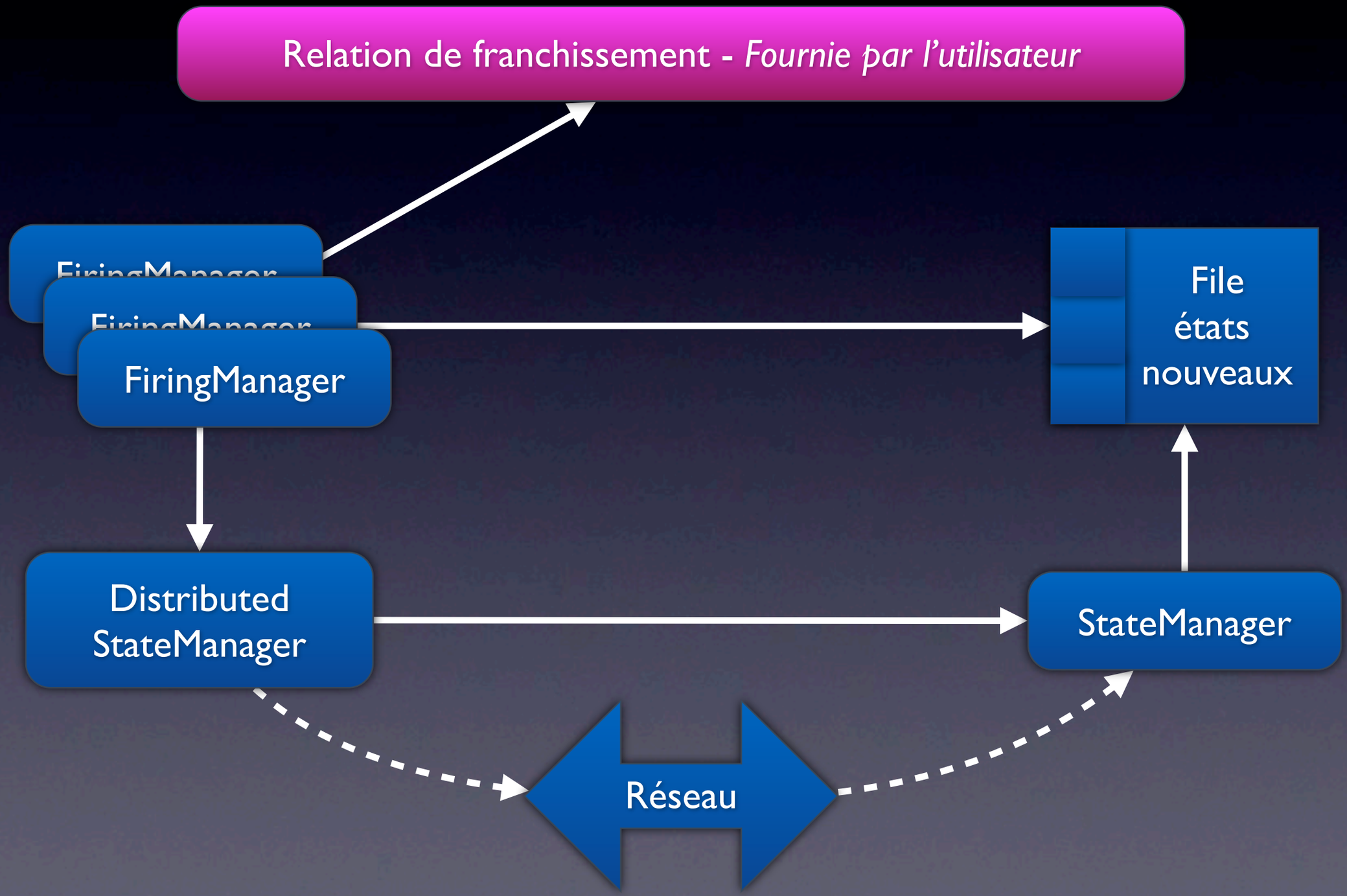
Architecture pour le model checking réparti et parallèle : libdmc



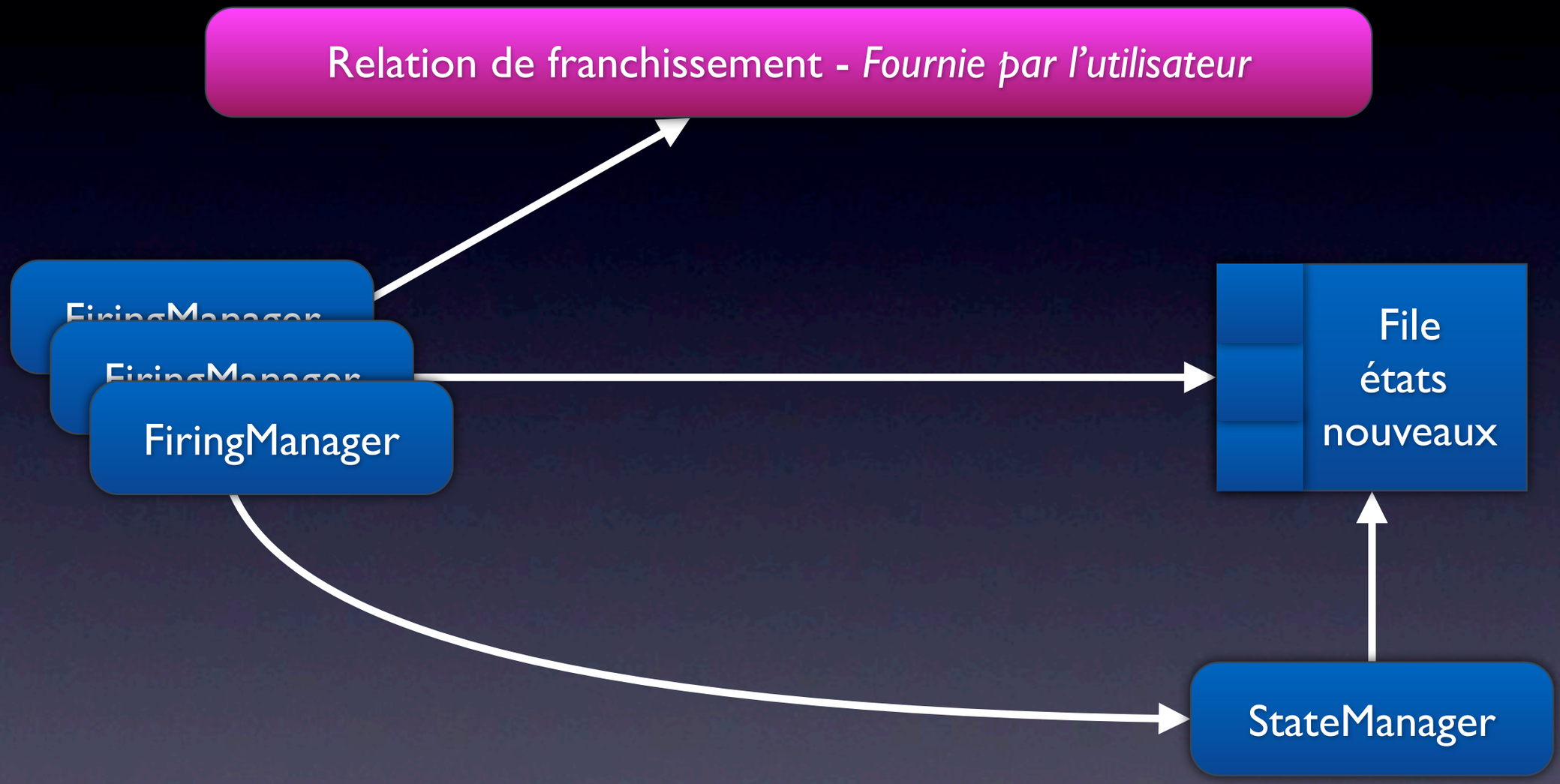
Architecture pour le model checking réparti et parallèle : libdmc



Architecture pour le model checking réparti et parallèle : libdmc



Architecture pour le model checking réparti et parallèle : libdmc



Positionnement

- Localisation
  - Statique [SD97,LS99,PP07,…], dynamique [LV01,Sch03,…]
  - Choix : statique (MD5)
- Recouvrement des communications
  - Plusieurs threads [Eddy], un seul thread […]
  - Choix : plusieurs threads

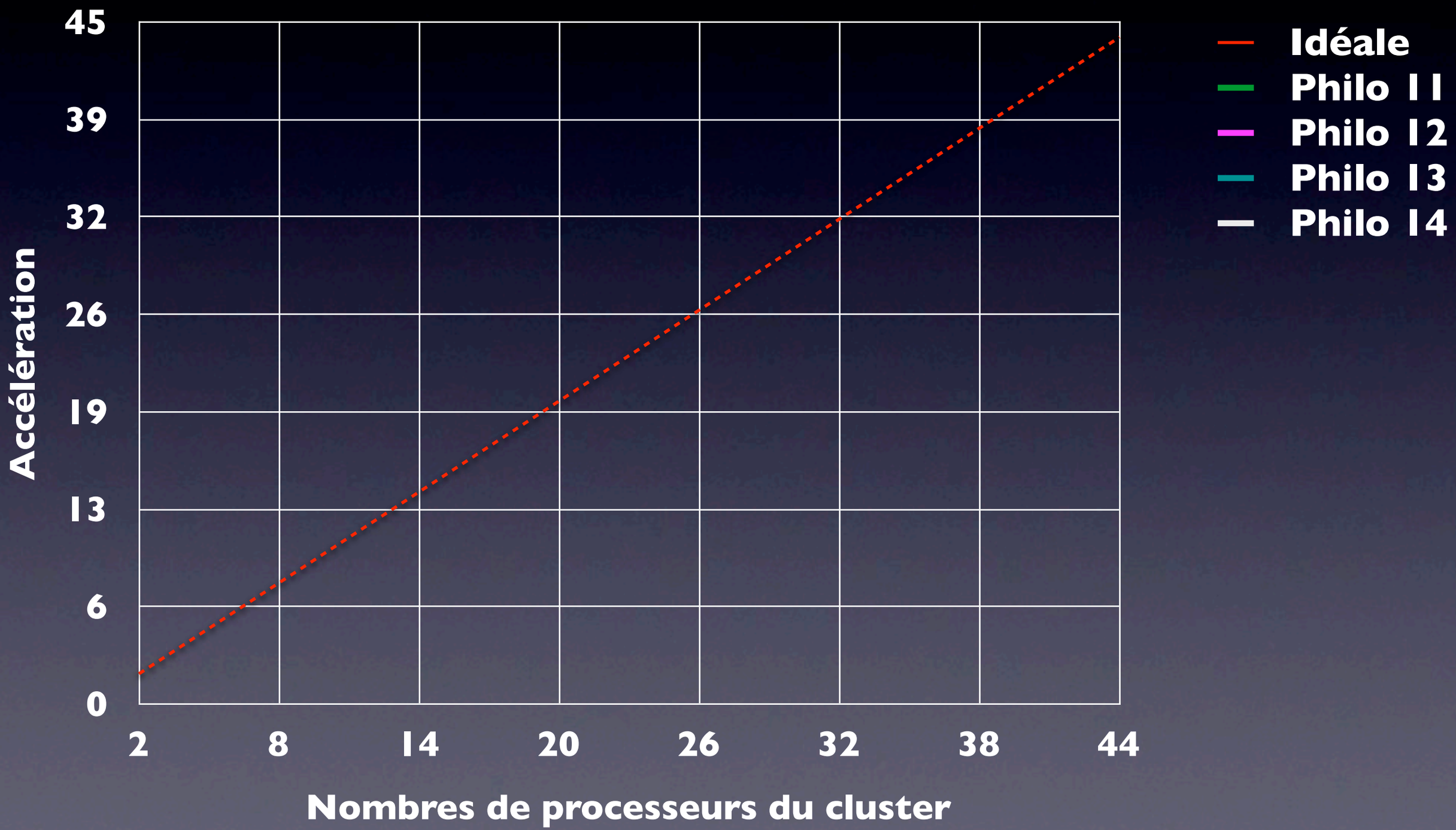
Génération répartie d'un espace d'états avec libdmc

- Structure de Kripke
  - État initial, règle de franchissement, étiquettes
- Contraintes
  - États "sérialisables"
- Validation
  - GreatSPN (*Université de Turin*)
  - Aucun algorithme à développer
  - Ré-entrance assurée par duplication de bibliothèques

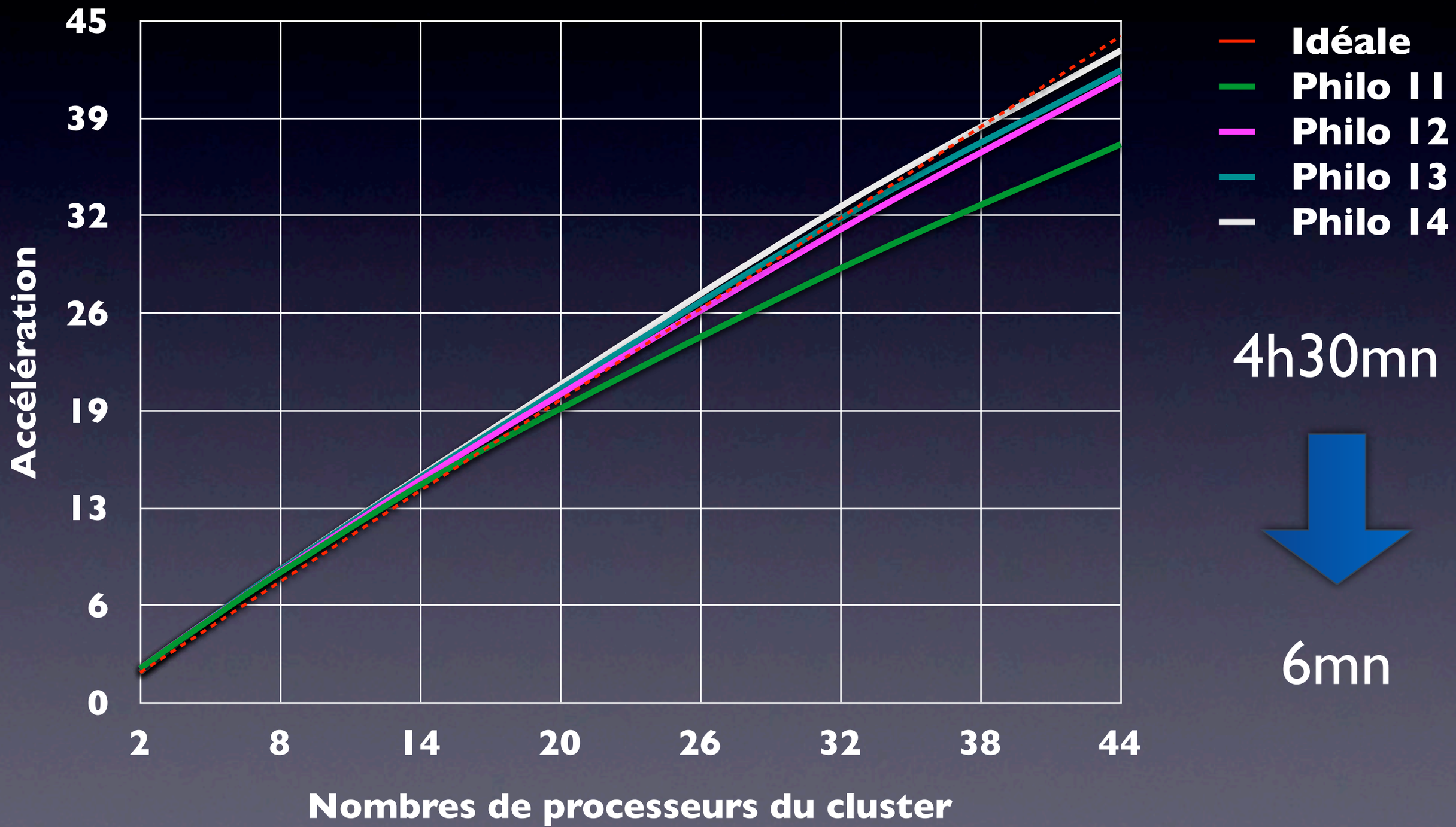
Expérimentations : Philosophes

Idéale
Philo 11
Philo 12
Philo 13
Philo 14

Expérimentations : Philosophes



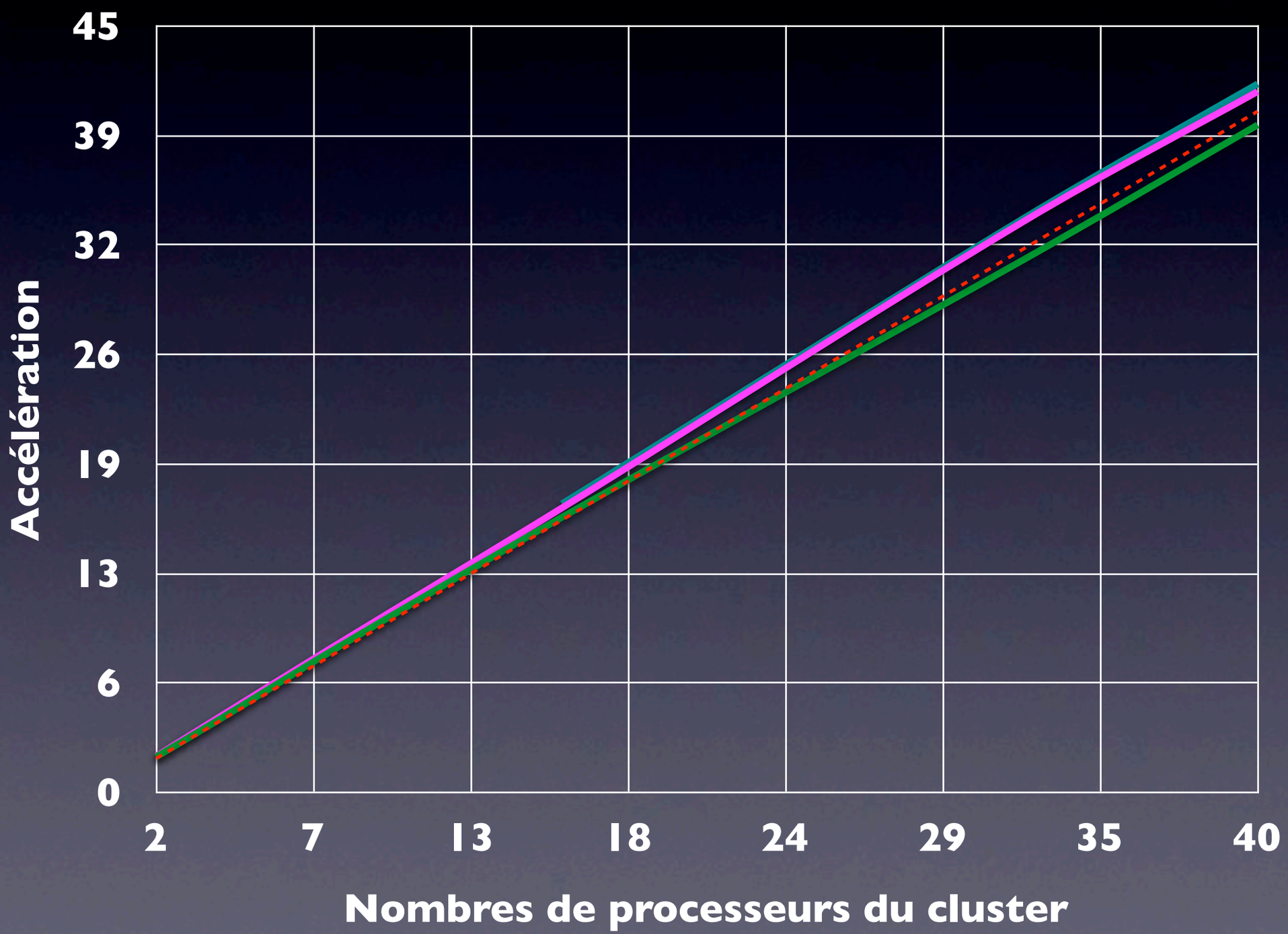
Expérimentations : Philosophes



4h30mn

6mn

Expérimentations : PolyORB [HVP05]



46h30mn

1h

Model checking réparti et parallèle : synthèse

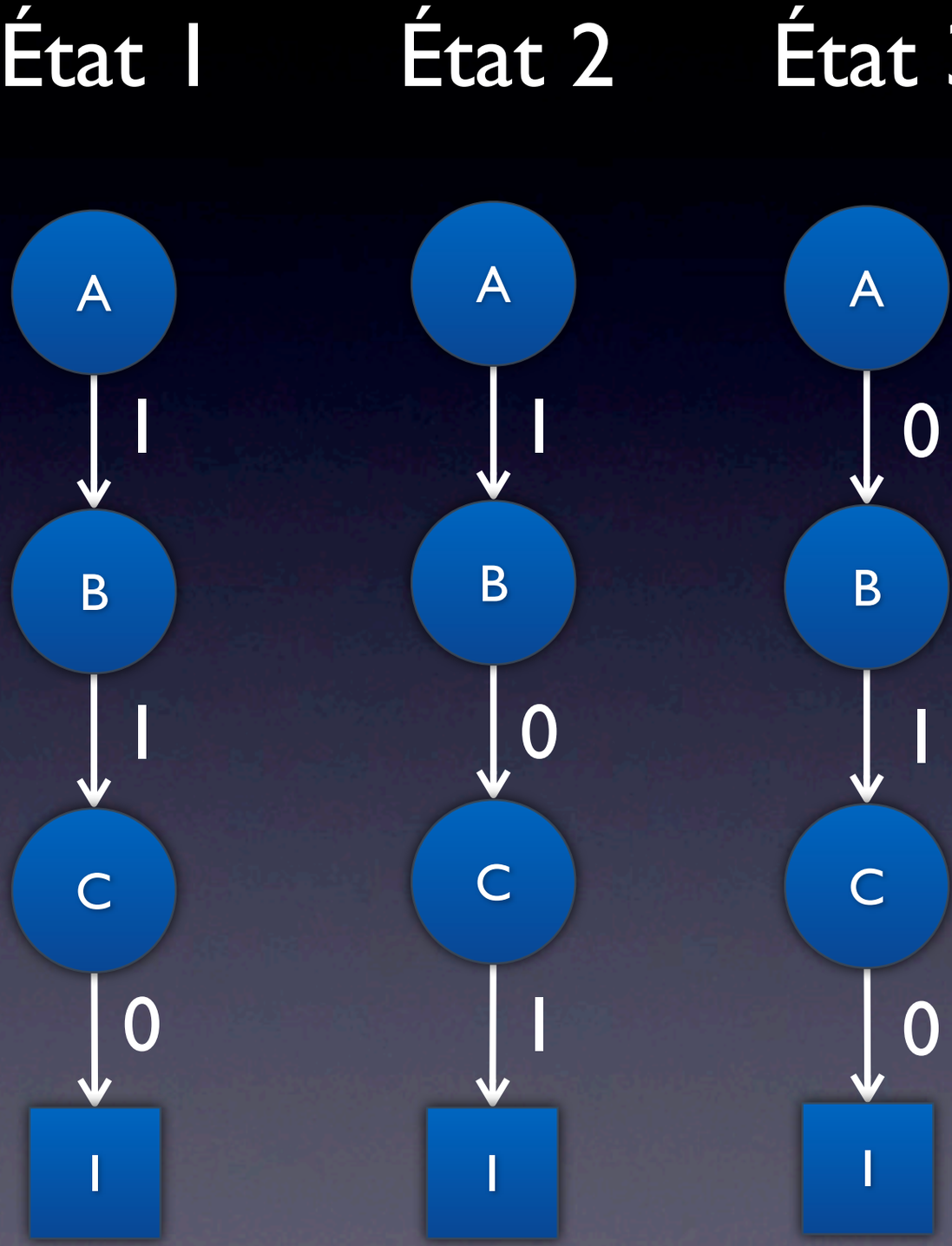
- Bibliothèque générique
  - Contraintes réalistes
- Validation avec résultats expérimentaux (GreatSPN)
- Conditions d'efficacité
  - Exploitation maximale des ressources calcul et mémoire
  - Relation de franchissement coûteuse en temps de calcul
  - Adapté au model checking explicite (symétries, …)

Saturation automatique

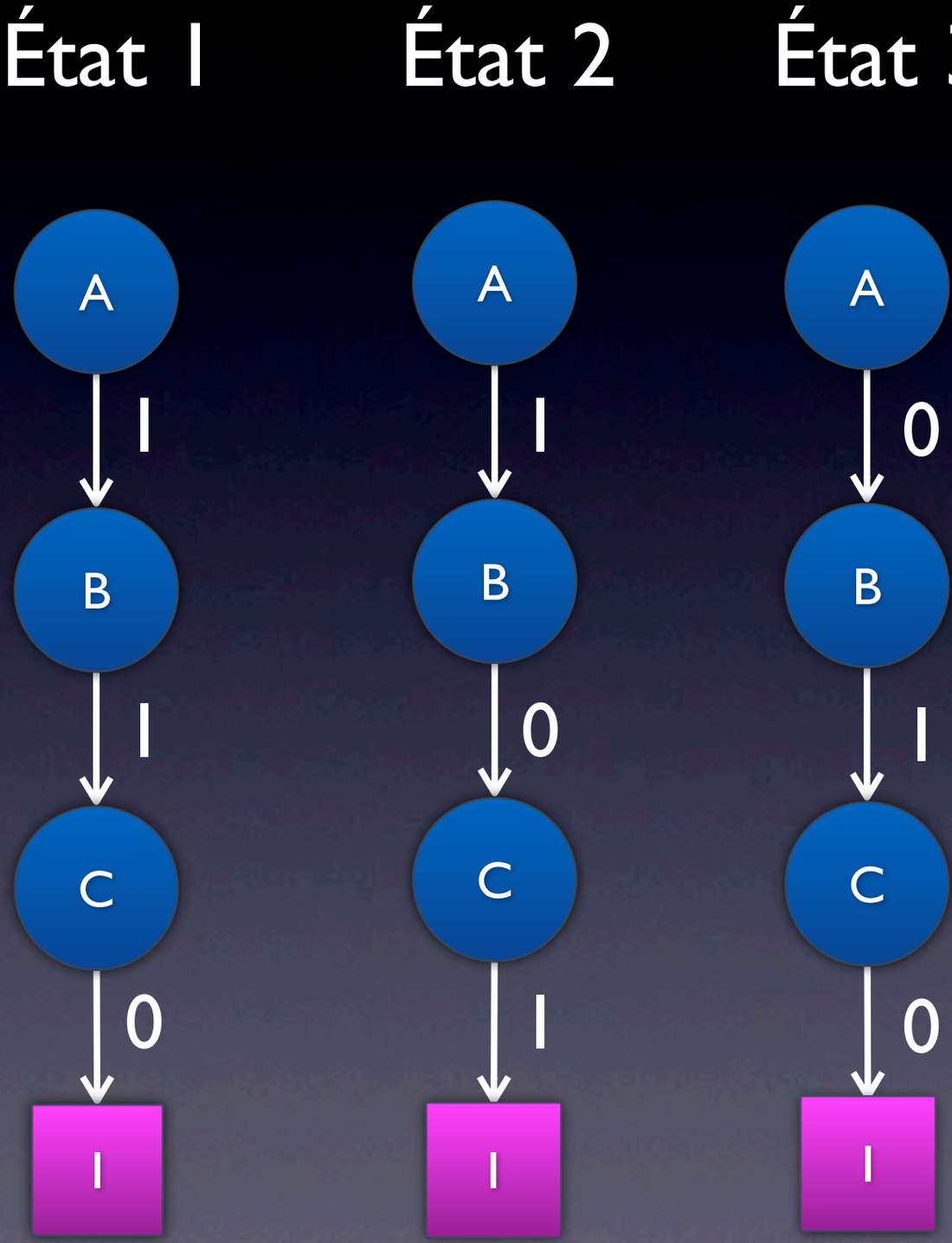
Model checking symbolique

- Diagrammes de décisions
  - BDD [86], MDD [95], DDD [02], SDD [04], …
- Caractéristiques des DD
  - Compression
  - Canonicité
  - Cache
- Nombreux outils : VIS, NuSMV, SMART, …

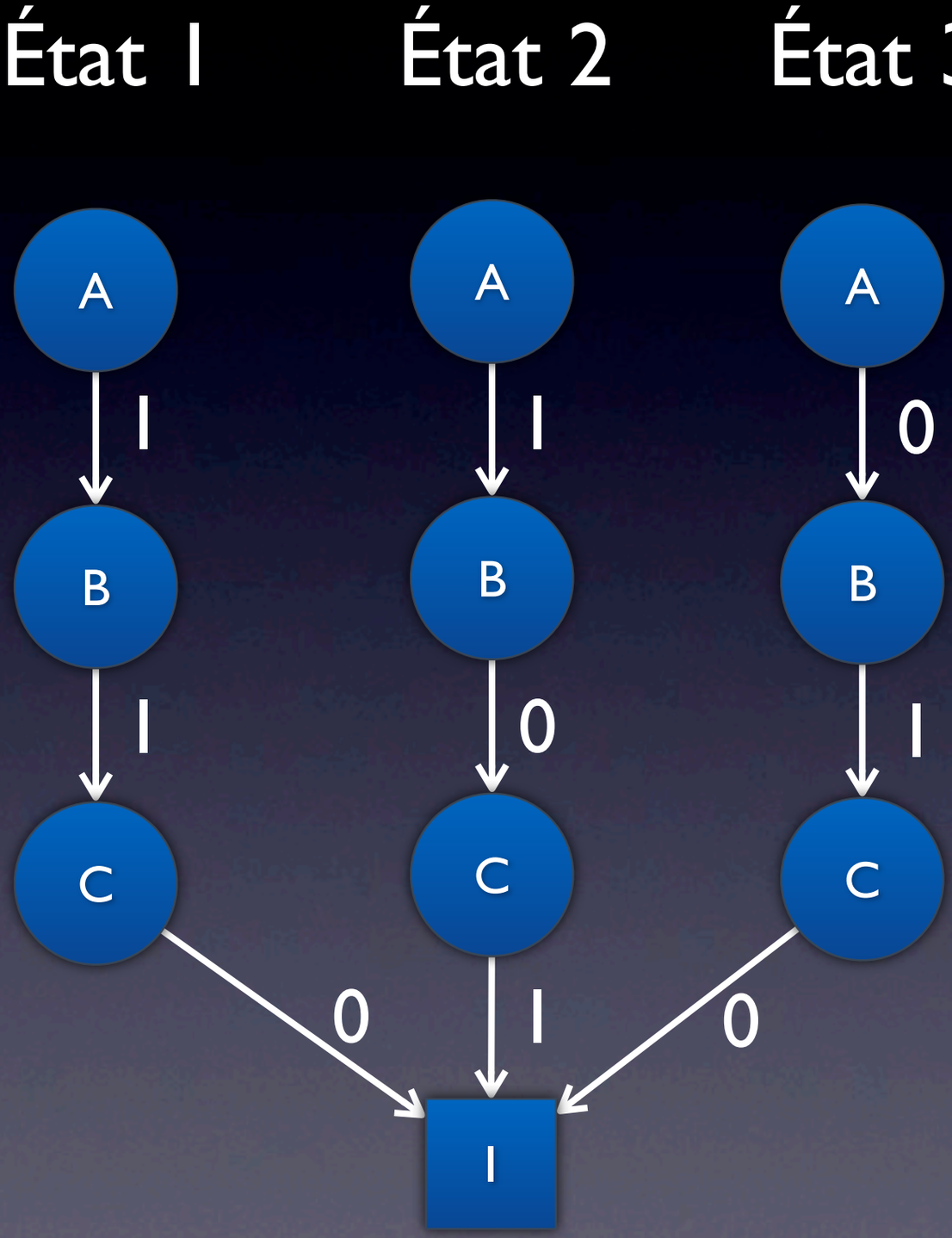
Diagrammes de décision pour l'encodage d'espaces d'états



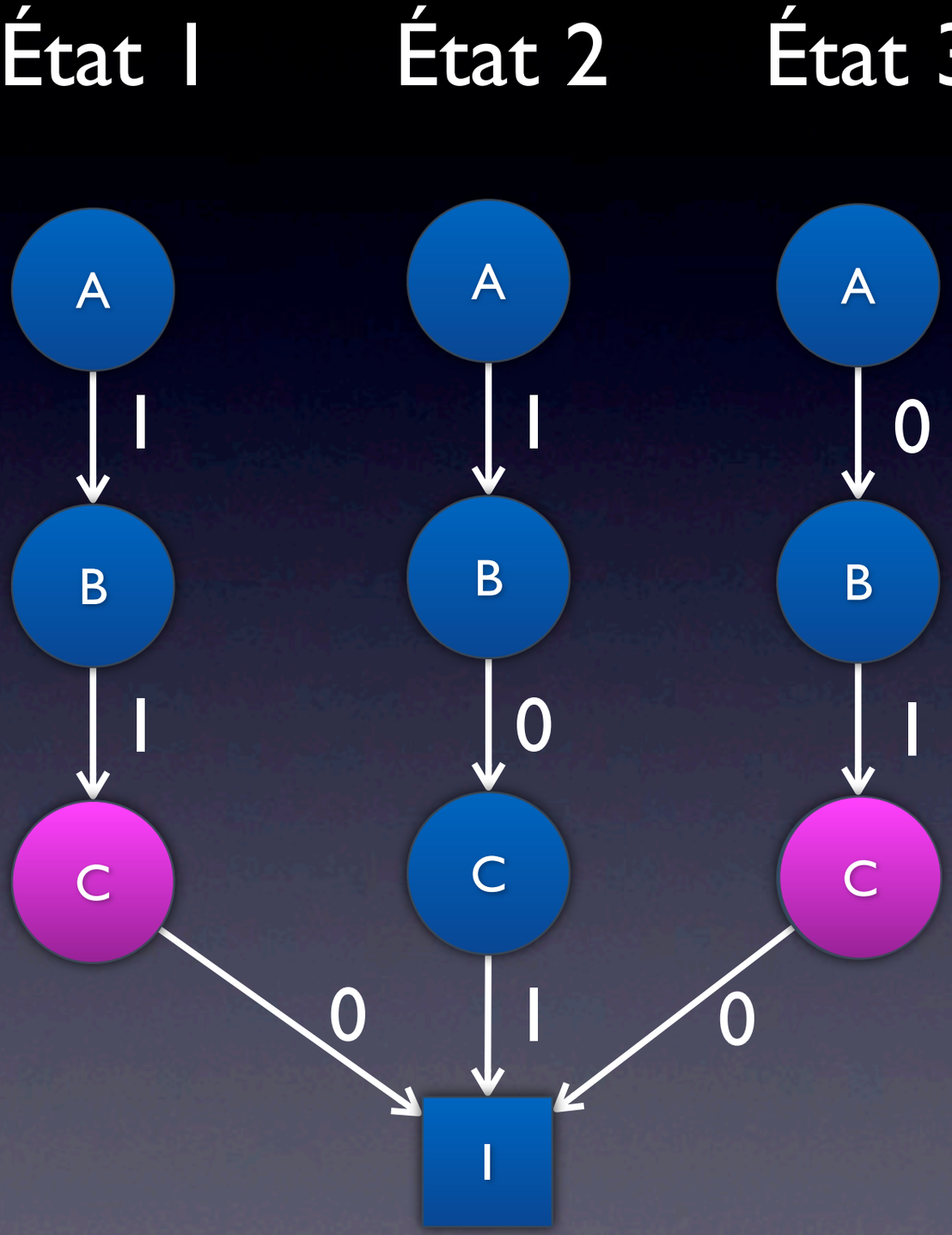
Diagrammes de décision pour l'encodage d'espaces d'états



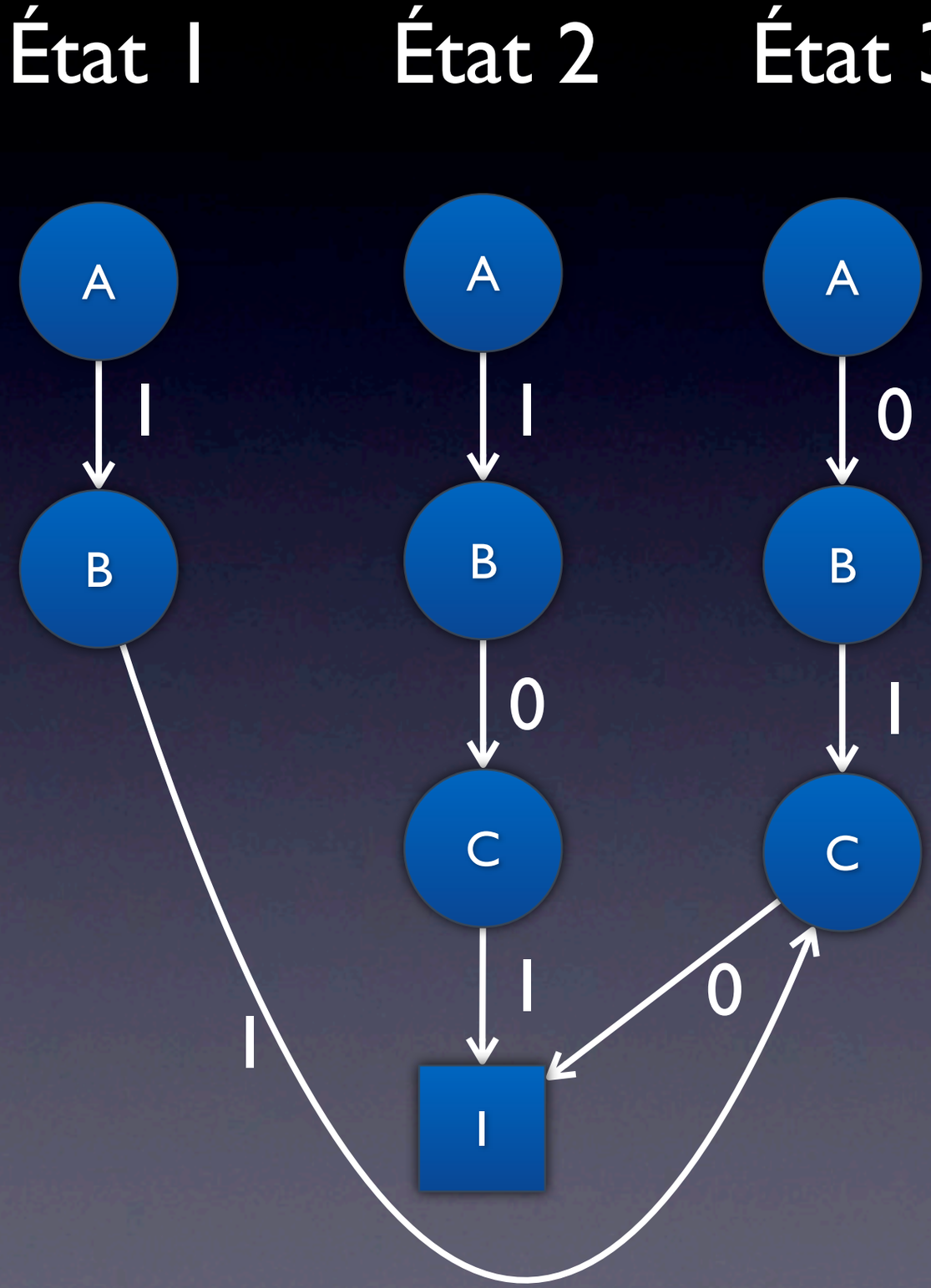
Diagrammes de décision pour l'encodage d'espaces d'états



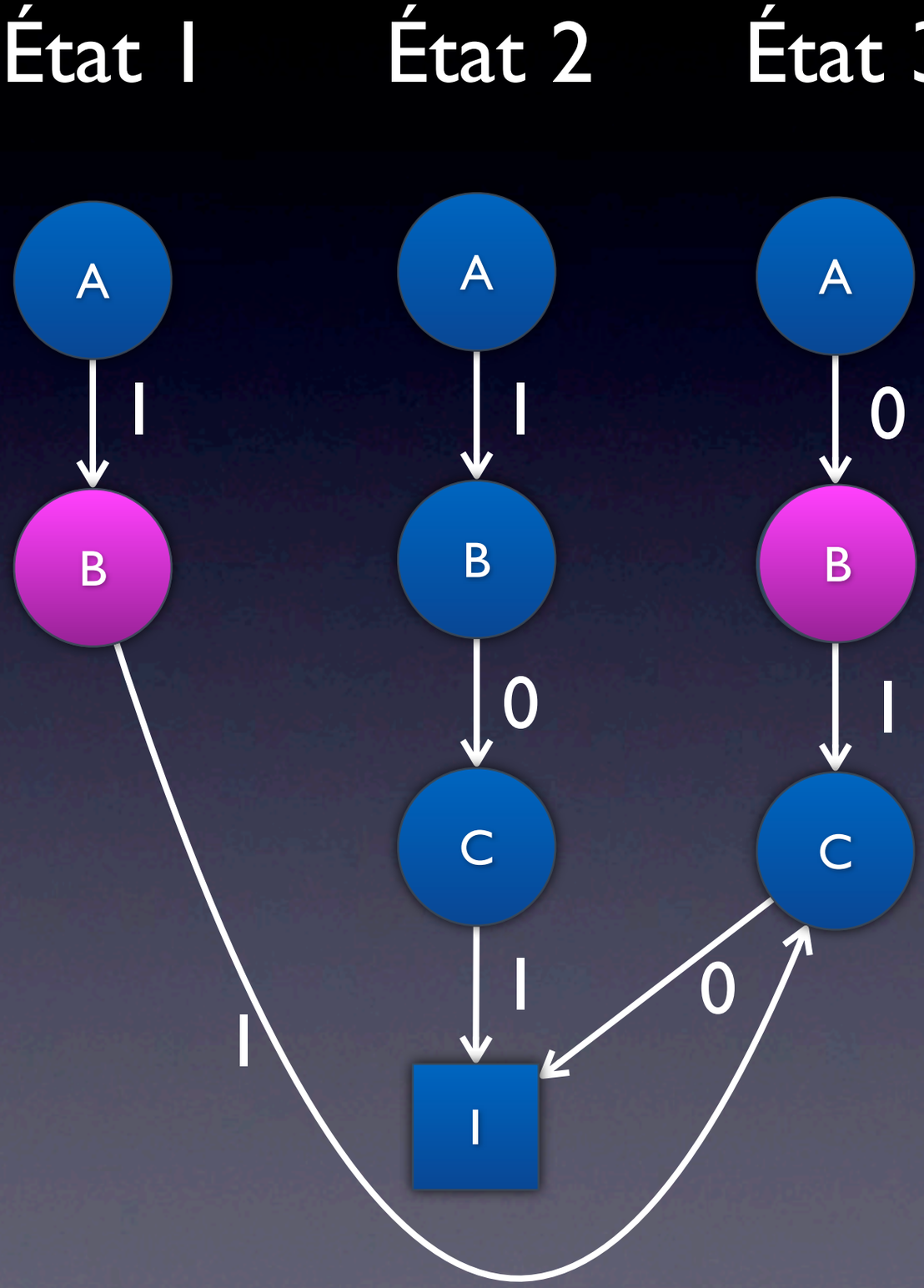
Diagrammes de décision pour l'encodage d'espaces d'états



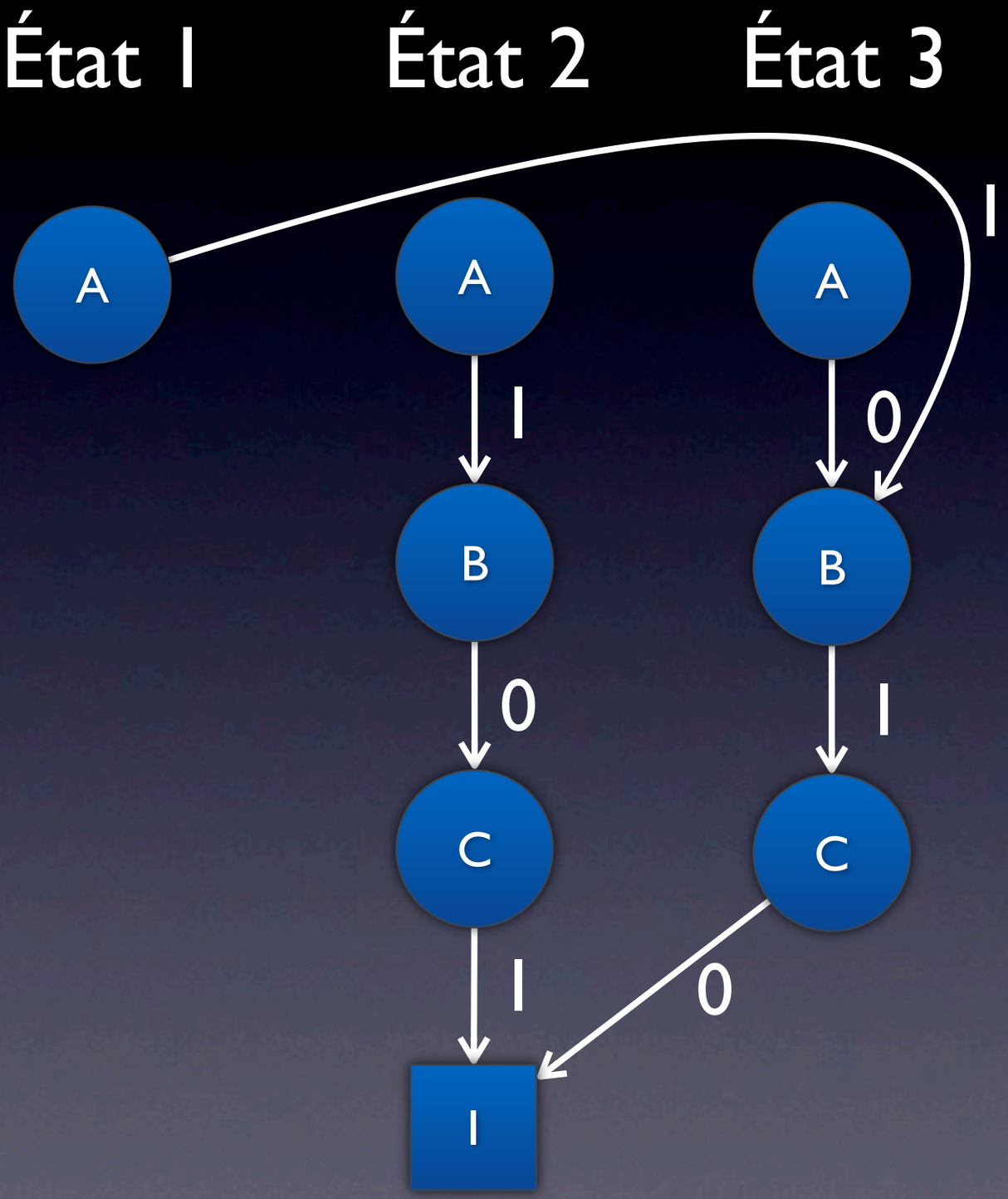
Diagrammes de décision pour l'encodage d'espaces d'états



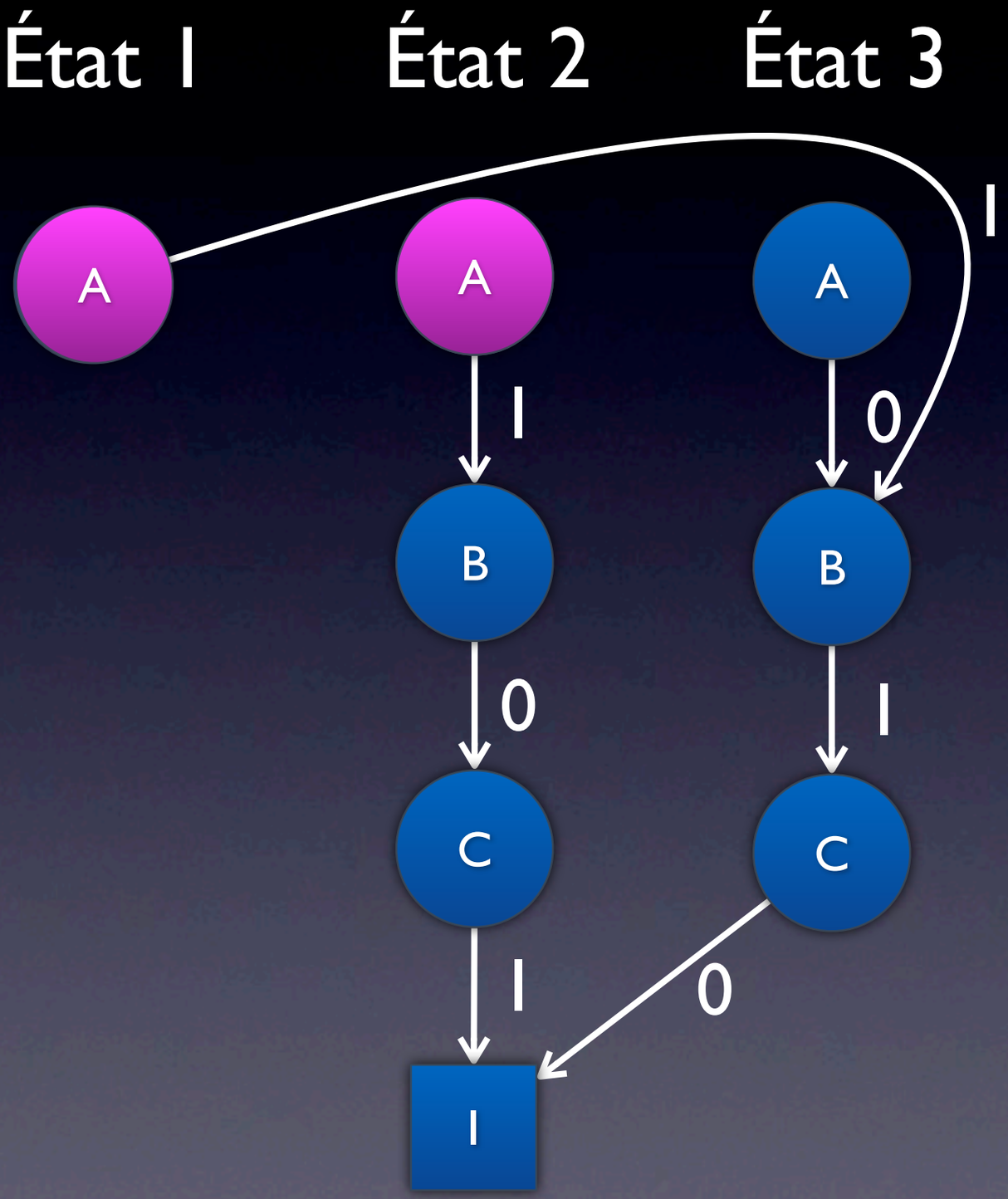
Diagrammes de décision pour l'encodage d'espaces d'états



Diagrammes de décision pour l'encodage d'espaces d'états

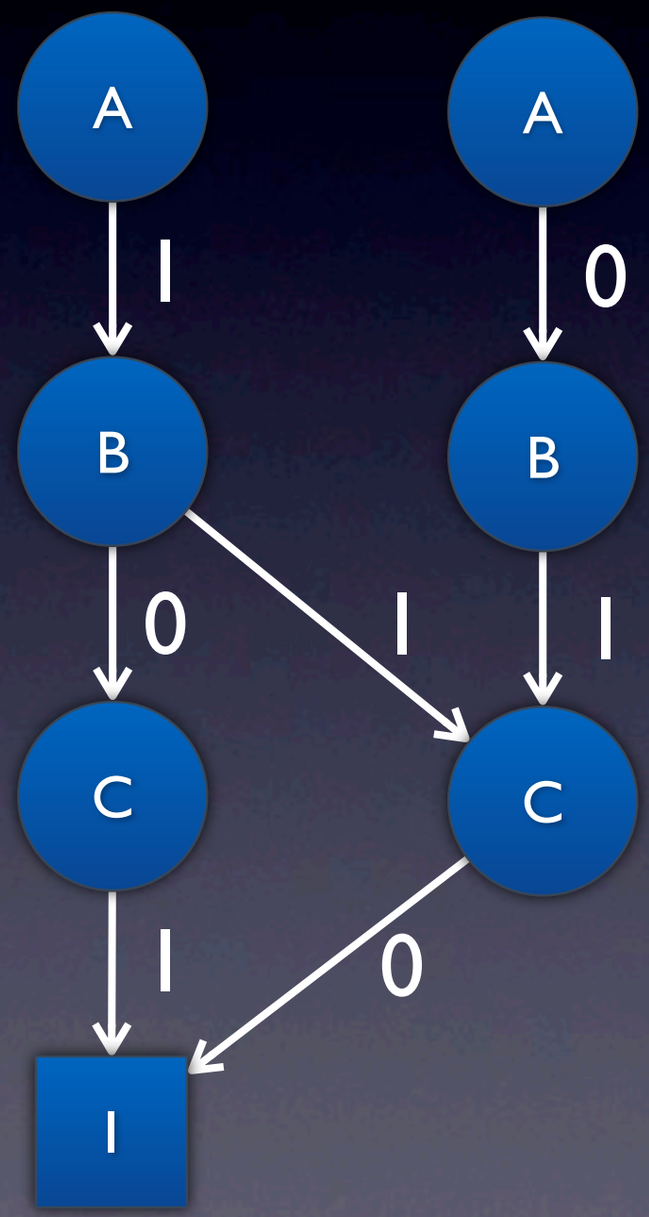


Diagrammes de décision pour l'encodage d'espaces d'états



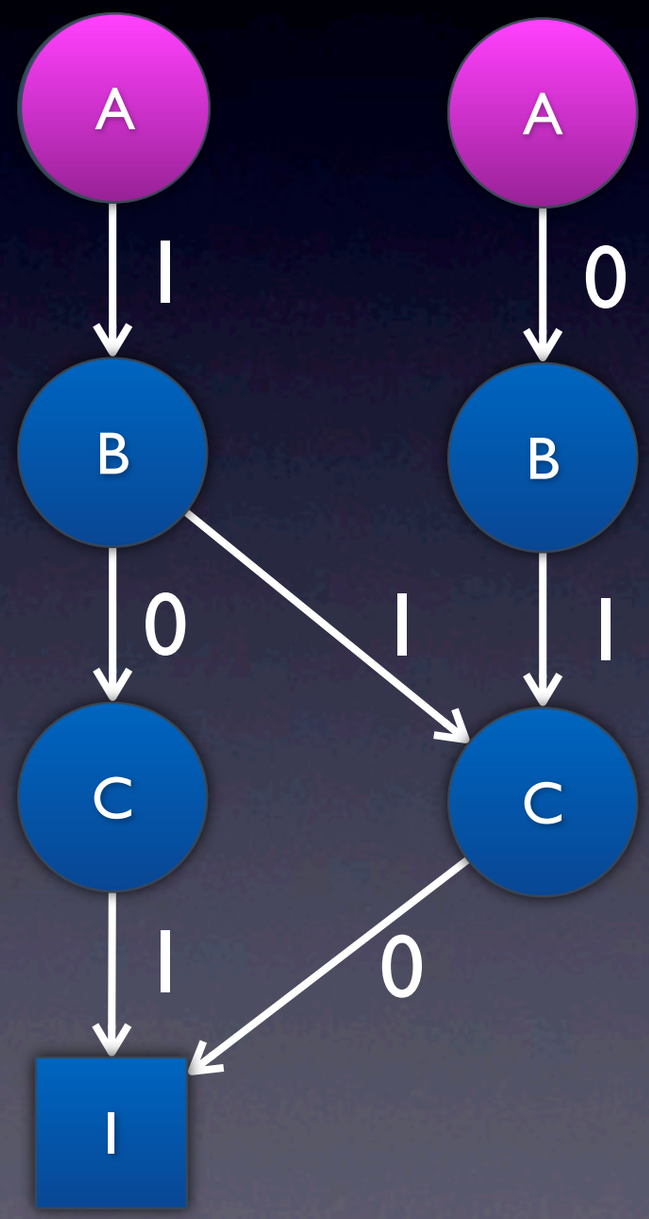
Diagrammes de décision pour l'encodage d'espaces d'états

État 1 État 2 État 3



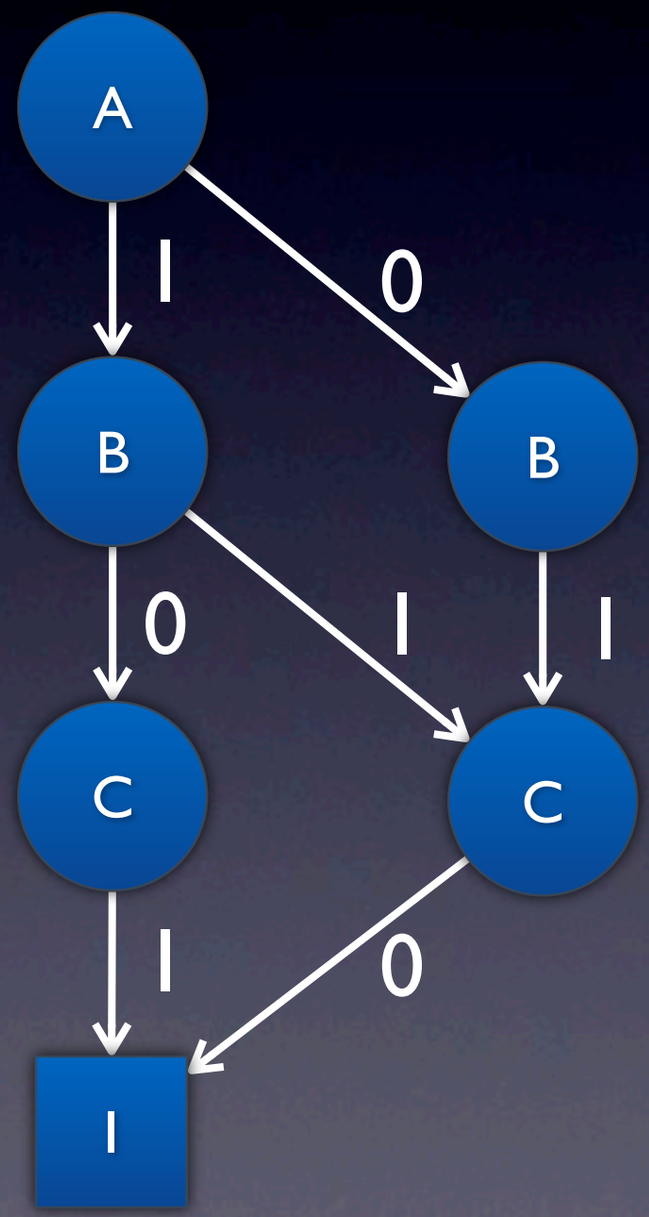
Diagrammes de décision pour l'encodage d'espaces d'états

État 1 État 2 État 3



Diagrammes de décision pour l'encodage d'espaces d'états

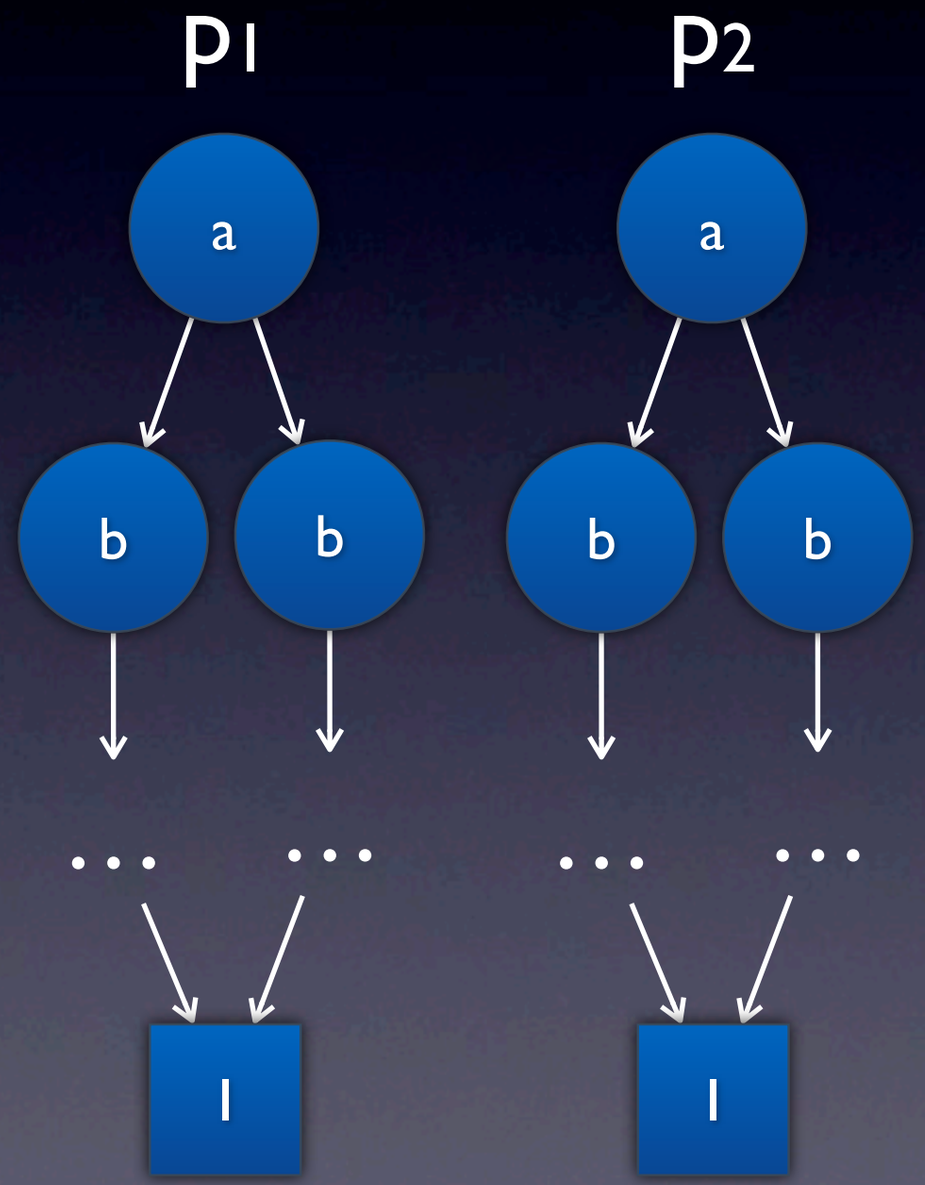
Espace d'états



Présentation des SDD

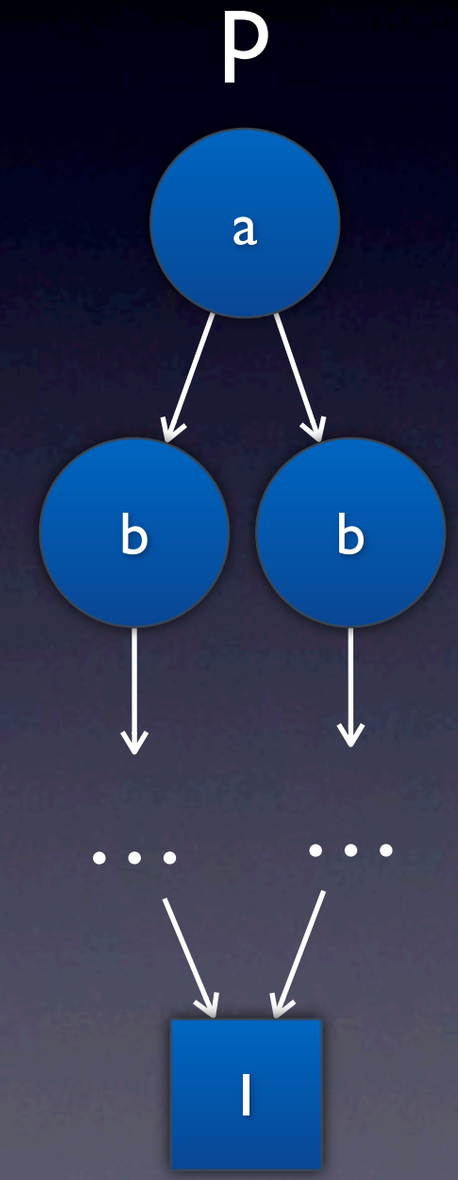
- BDD : binaire
- MDD : entiers
- SDD : ensembles

- Hiérarchie
- Factoriser les similarités
- DD “traditionnels” : les espaces d’états sont placés successivement
- Partage favorisé



Présentation des SDD

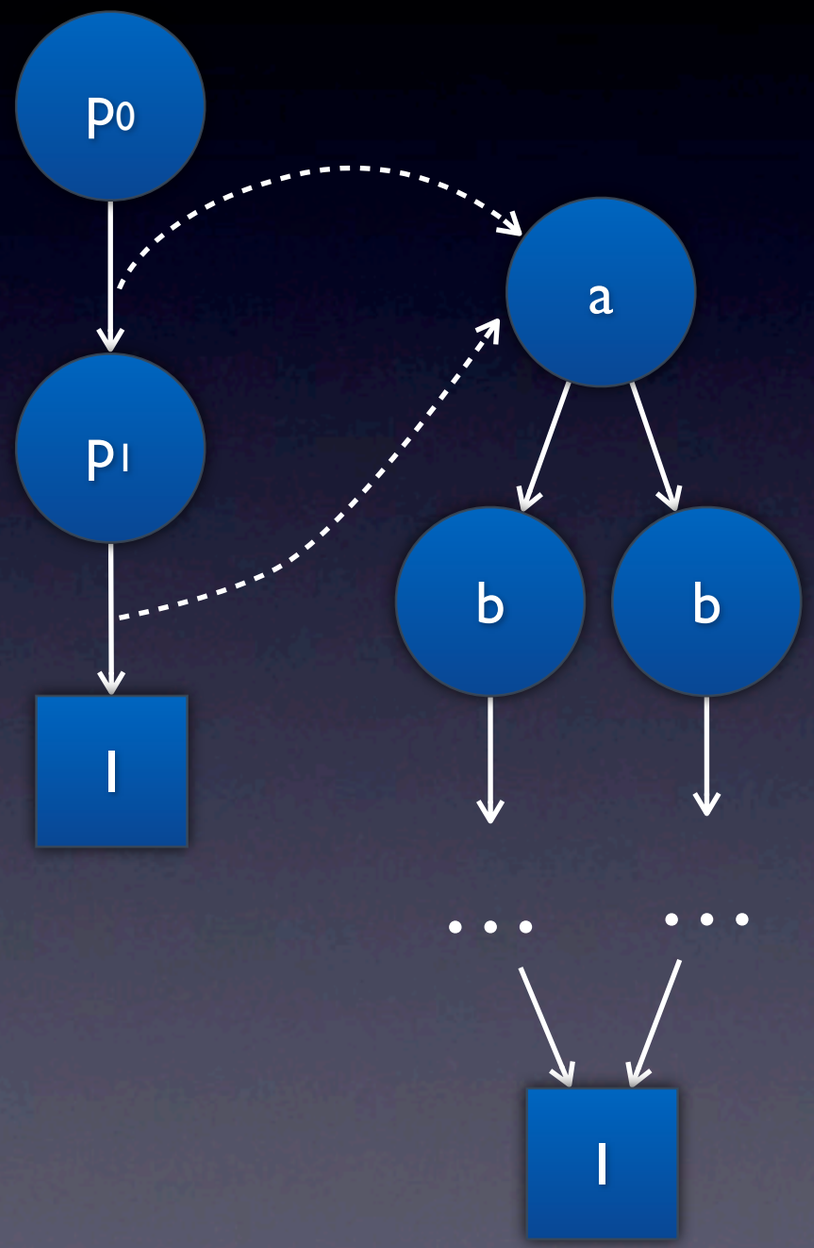
- BDD : binaire
- MDD : entiers
- SDD : ensembles



- Hiérarchie
- Factoriser les similarités
- DD “traditionnels” : les espaces d’états sont placés successivement
- Partage favorisé

Présentation des SDD

- BDD : binaire
- MDD : entiers
- SDD : ensembles
  - Hiérarchie
  - Factoriser les similarités
  - DD “traditionnels” : les espaces d’états sont placés successivement
  - Partage favorisé



Homomorphismes pour coder la relation de franchissement

- Homomorphismes **SDD → SDD**
- Homomorphismes de base
  - Constante, Identité, …
- Homomorphismes inductifs
  - Pour étendre les homomorphismes de base
- Combinaisons d'homomorphismes
  - somme, composition, …
- Point fixe
  - $h^{\bigstar} = h^{n}$ (cas finis)

Homomorphismes pour coder la relation de franchissement

- Homomorphismes **SDD → SDD**
- Homomorphismes de base
  - Constante, Identité, …

- Homomorphismes inductifs
  - Pour étendre les homomorphismes de base
- Combinaisons d'homomorphismes
  - somme, composition, …
- Point fixe
  - $h^{\bigstar} = h^{n}$ (cas finis)

Homomorphismes pour coder la relation de franchissement

- Homomorphismes **SDD → SDD**
- Homomorphismes de base
  - Constante, Identité, …

Id

- Homomorphismes inductifs
  - Pour étendre les homomorphismes de base

d++ d < 2

- Combinaisons d'homomorphismes
  - somme, composition, …
- Point fixe
  - $h^{\bigstar} = h^{n}$ (cas finis)

Homomorphismes pour coder la relation de franchissement

- Homomorphismes **SDD → SDD**
- Homomorphismes de base
  - Constante, Identité, …

Id

- Homomorphismes inductifs
  - Pour étendre les homomorphismes de base

d++ d < 2

- Combinaisons d'homomorphismes
  - somme, composition, …

(d++) o (d < 2)

- Point fixe
  - $h^{\bigstar} = h^{n}$ (cas finis)

Homomorphismes pour coder la relation de franchissement

- Homomorphismes **SDD → SDD**
- Homomorphismes de base
  - Constante, Identité, …

Id

- Homomorphismes inductifs
  - Pour étendre les homomorphismes de base

d++ d < 2

- Combinaisons d'homomorphismes
  - somme, composition, …

(d++) ∘ (d < 2)

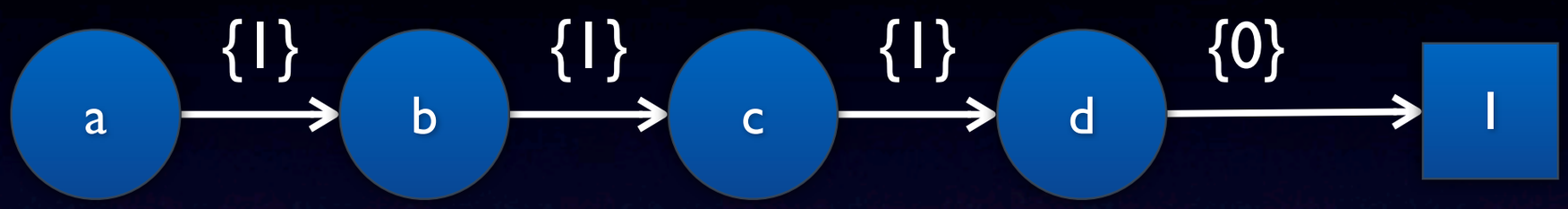
- Point fixe
  - $h^{\bigstar} = h^{n}$ (cas finis)

((d++) ∘ (d < 2)+Id)★

Calcul d'un espace d'états symbolique

h : (d++) o (d < 2)

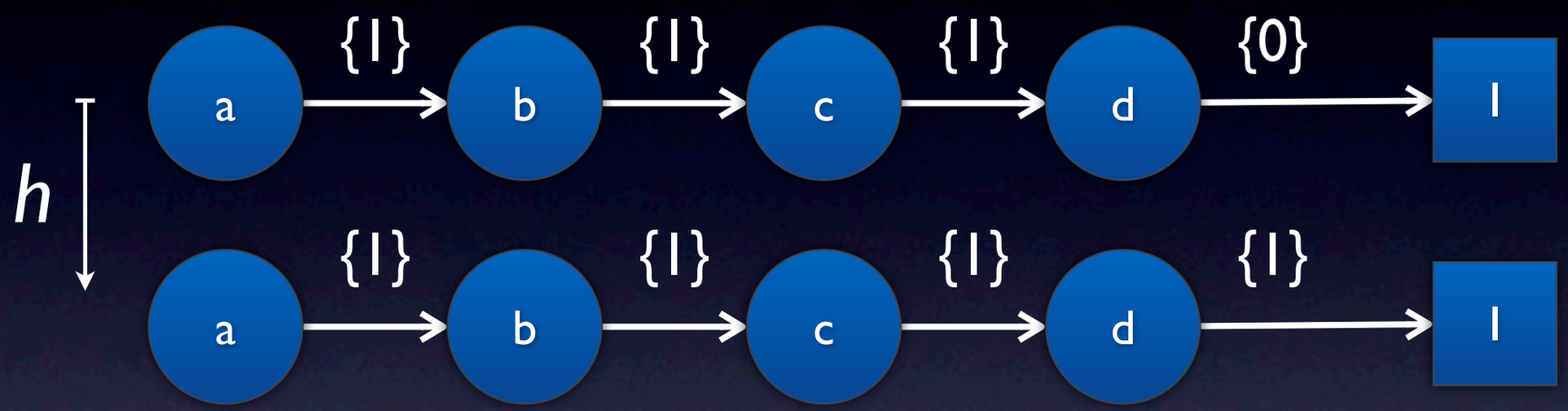
(h+Id)★



Calcul d'un espace d'états symbolique

h : (d++) o (d < 2)

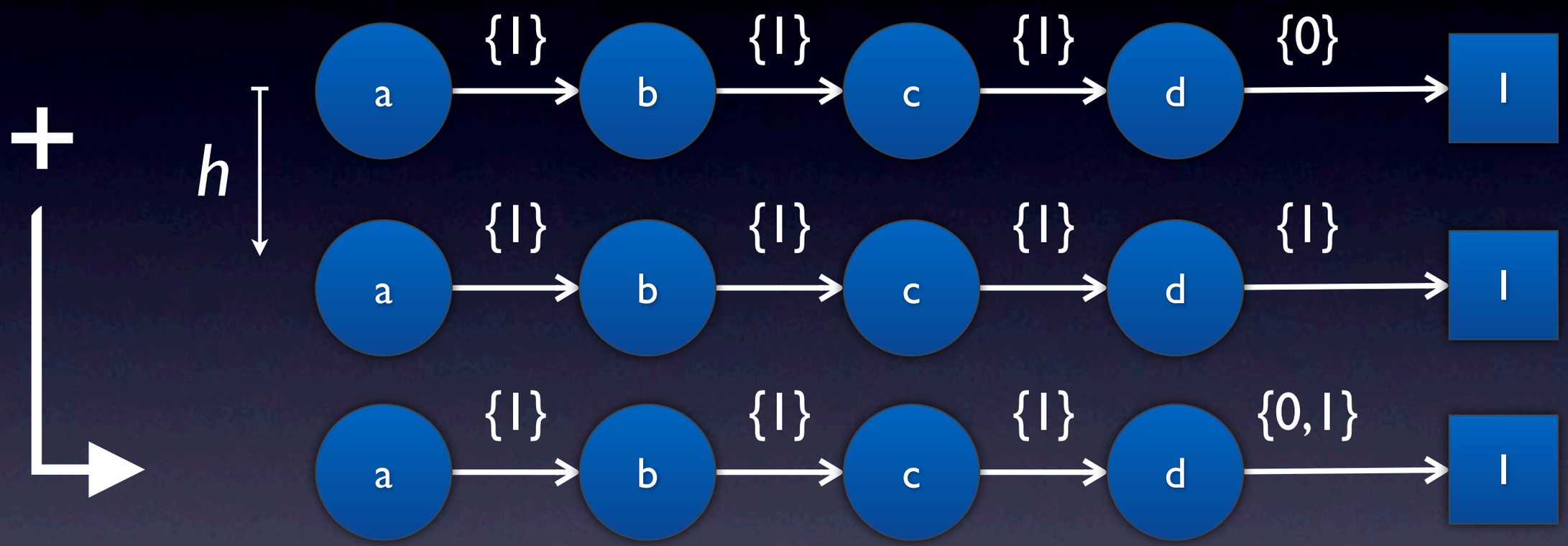
(h+Id) ★



Calcul d'un espace d'états symbolique

h : (d++) o (d < 2)

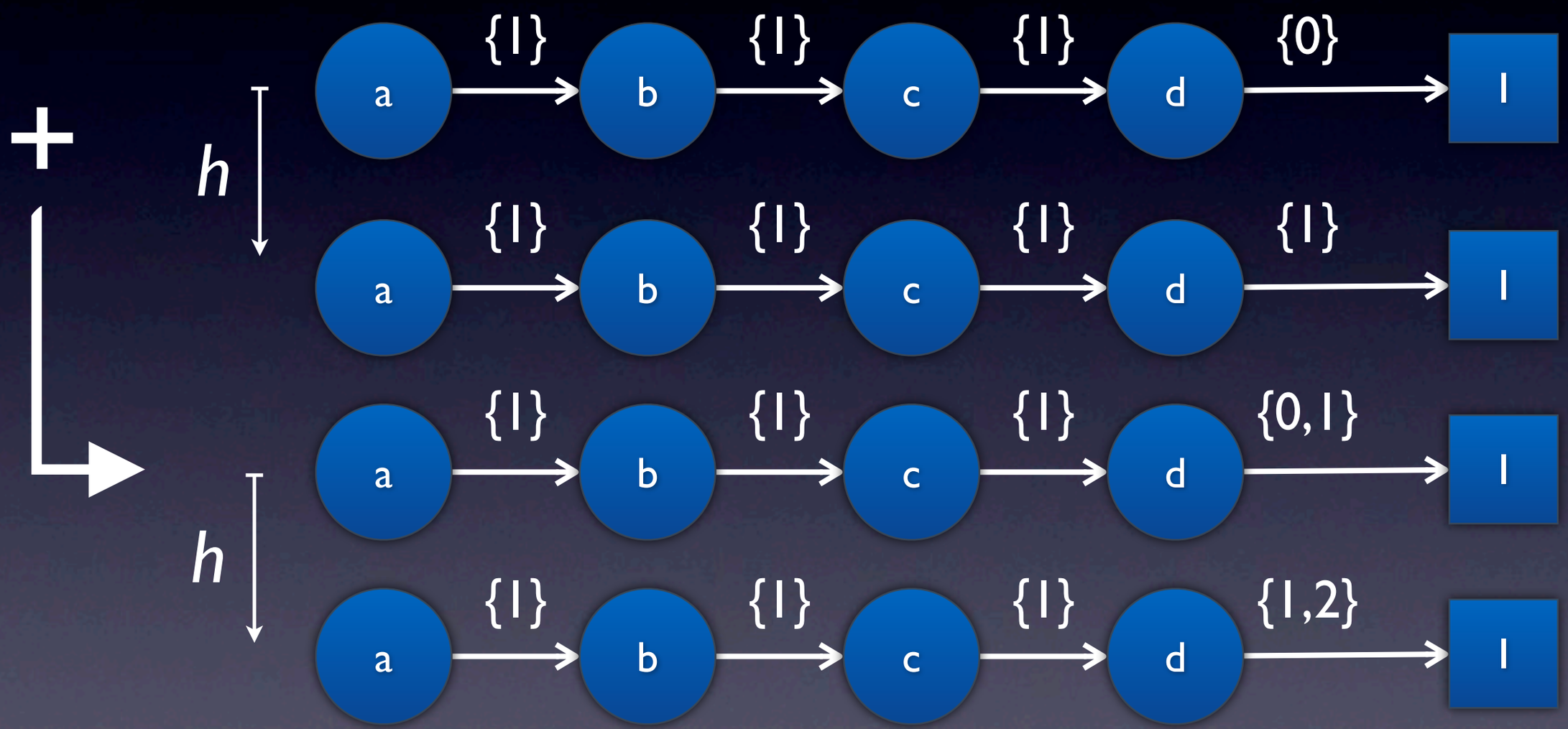
(h+Id)★



Calcul d'un espace d'états symbolique

h : (d++) o (d < 2)

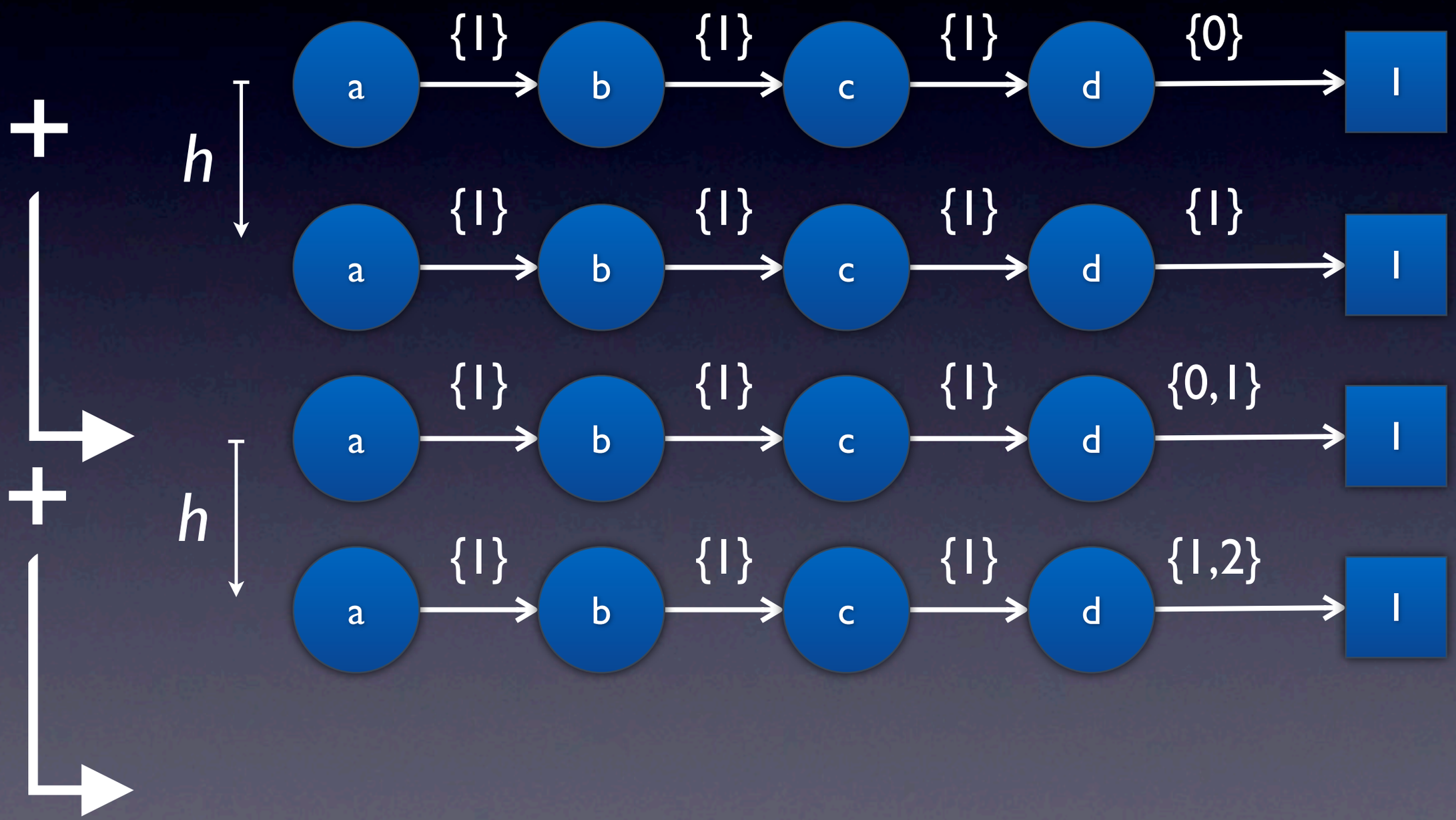
(h+Id)★



Calcul d'un espace d'états symbolique

h : (d++) ∘ (d < 2)

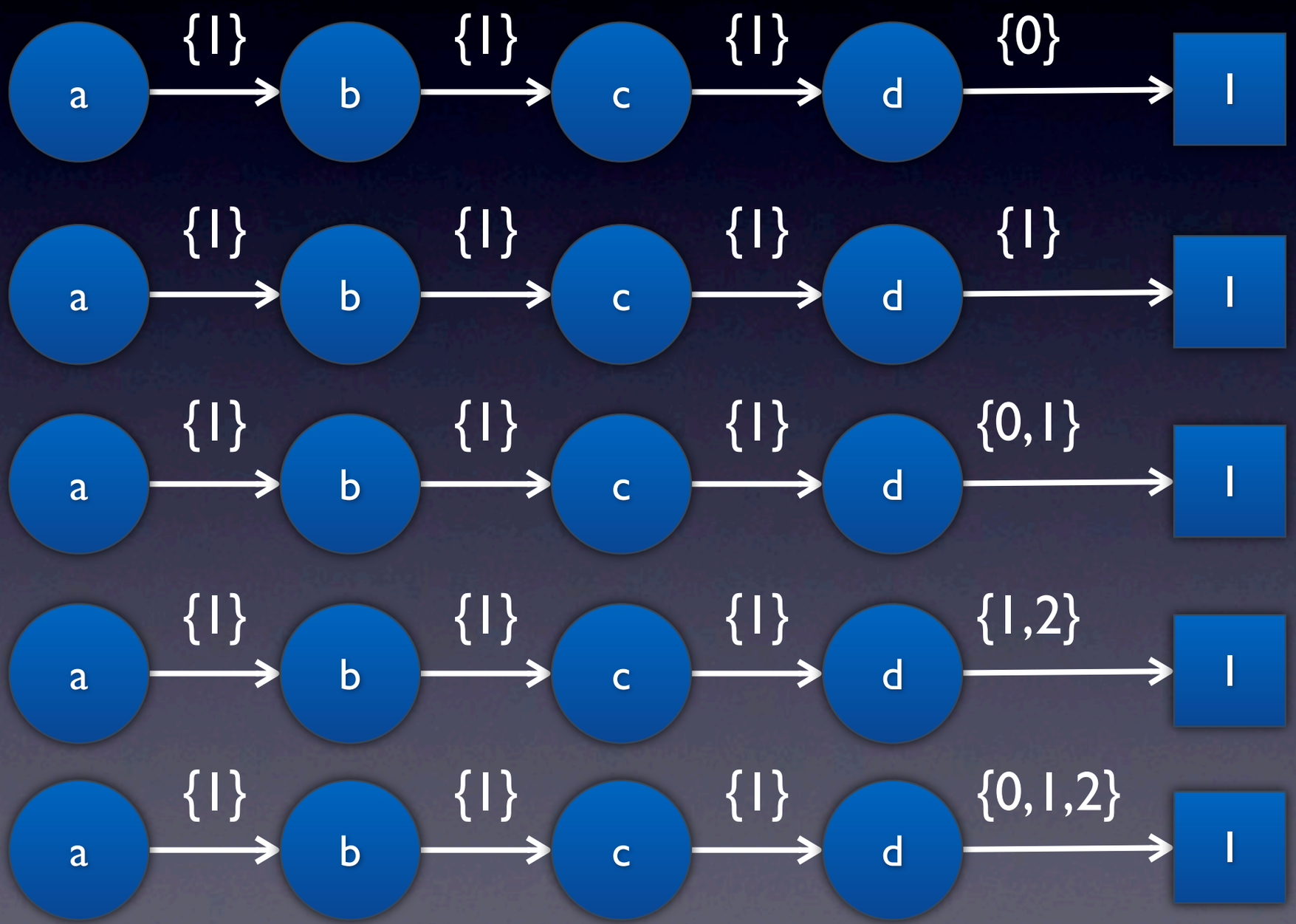
(h+Id)★



Calcul d'un espace d'états symbolique

h : (d++) o (d < 2)

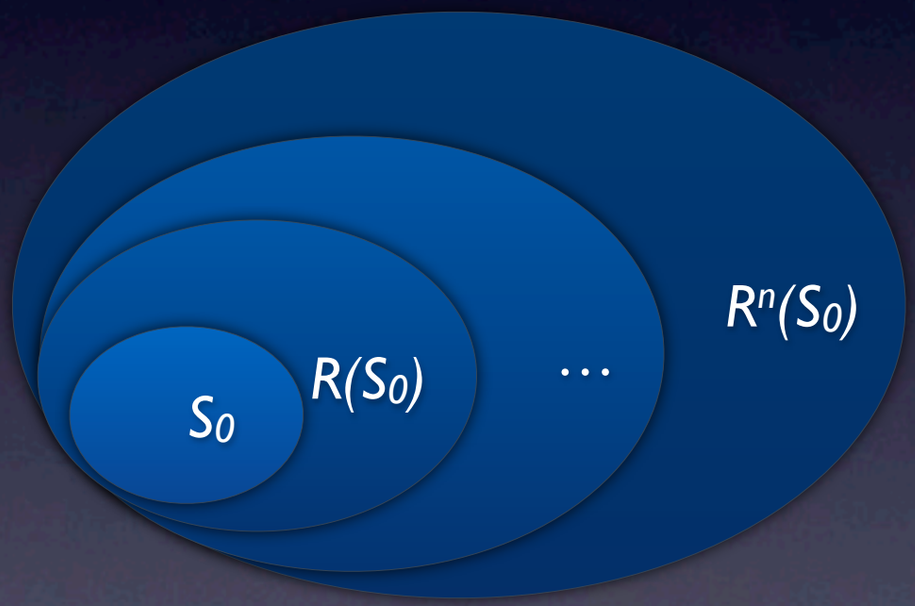
(h+Id)★



Calcul d'un point fixe : BFS et chaînage

$$R = t_1 + t_2 + \ldots + Id$$

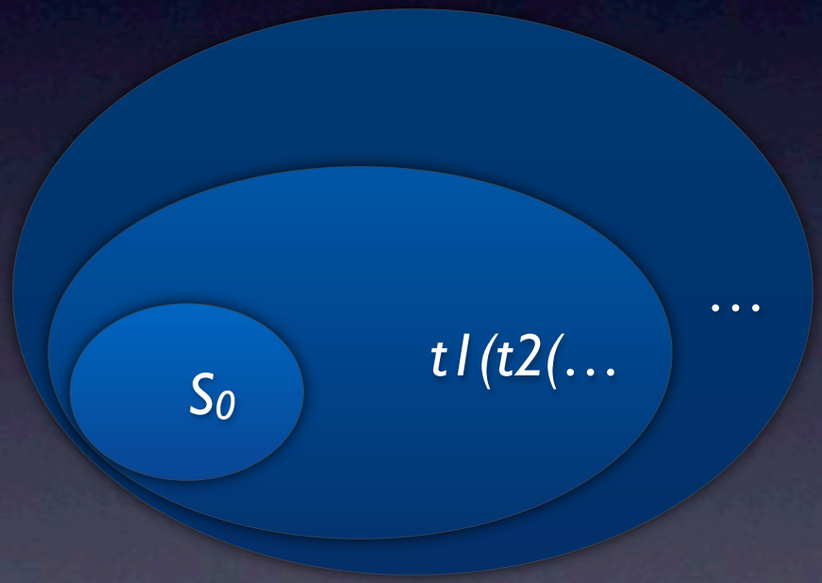
BFS [BCM90]



n itérations

relation de franchissement monolithique

Chaînage [RCP95]



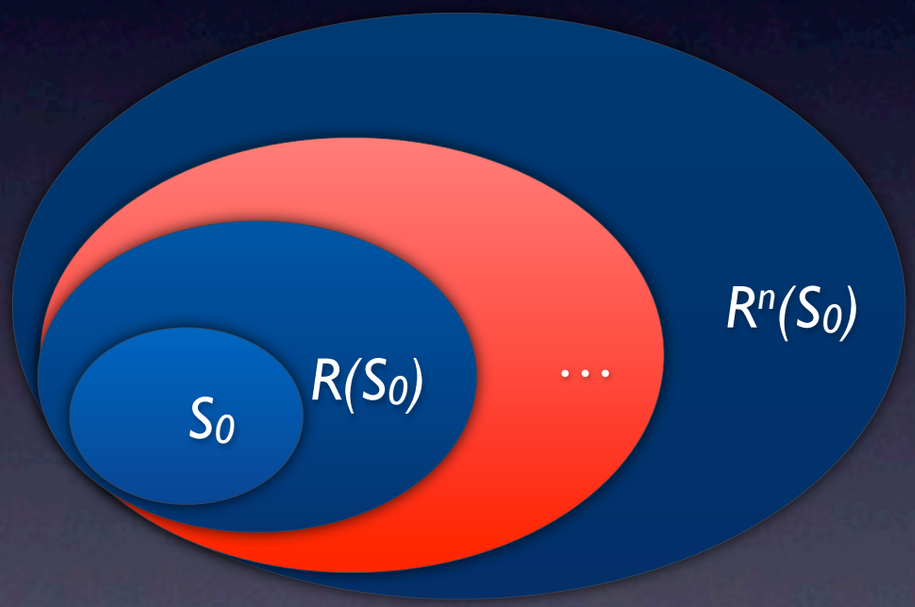
≤ n itérations

relation de franchissement éclatée

Calcul d'un point fixe : BFS et chaînage

$$R = t_1 + t_2 + \ldots + Id$$

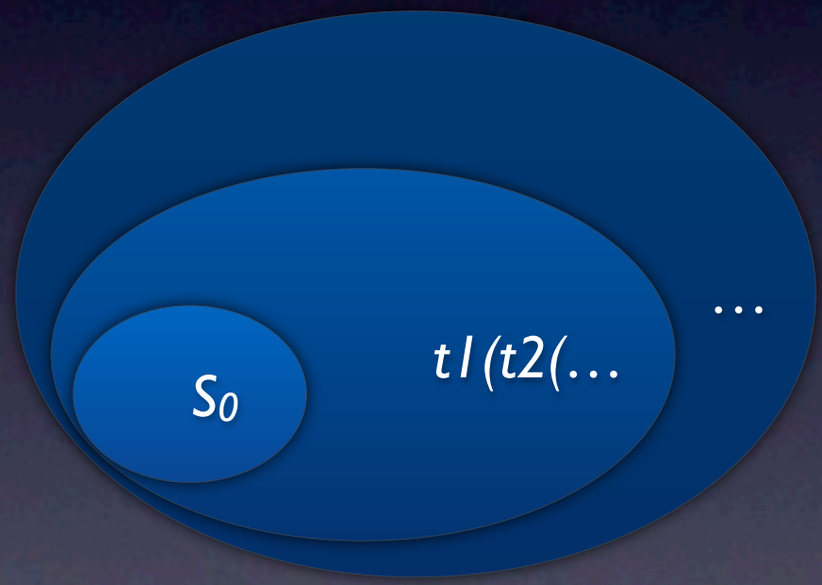
BFS [BCM90]



n itérations

relation de franchissement monolithique

Chaînage [RCP95]



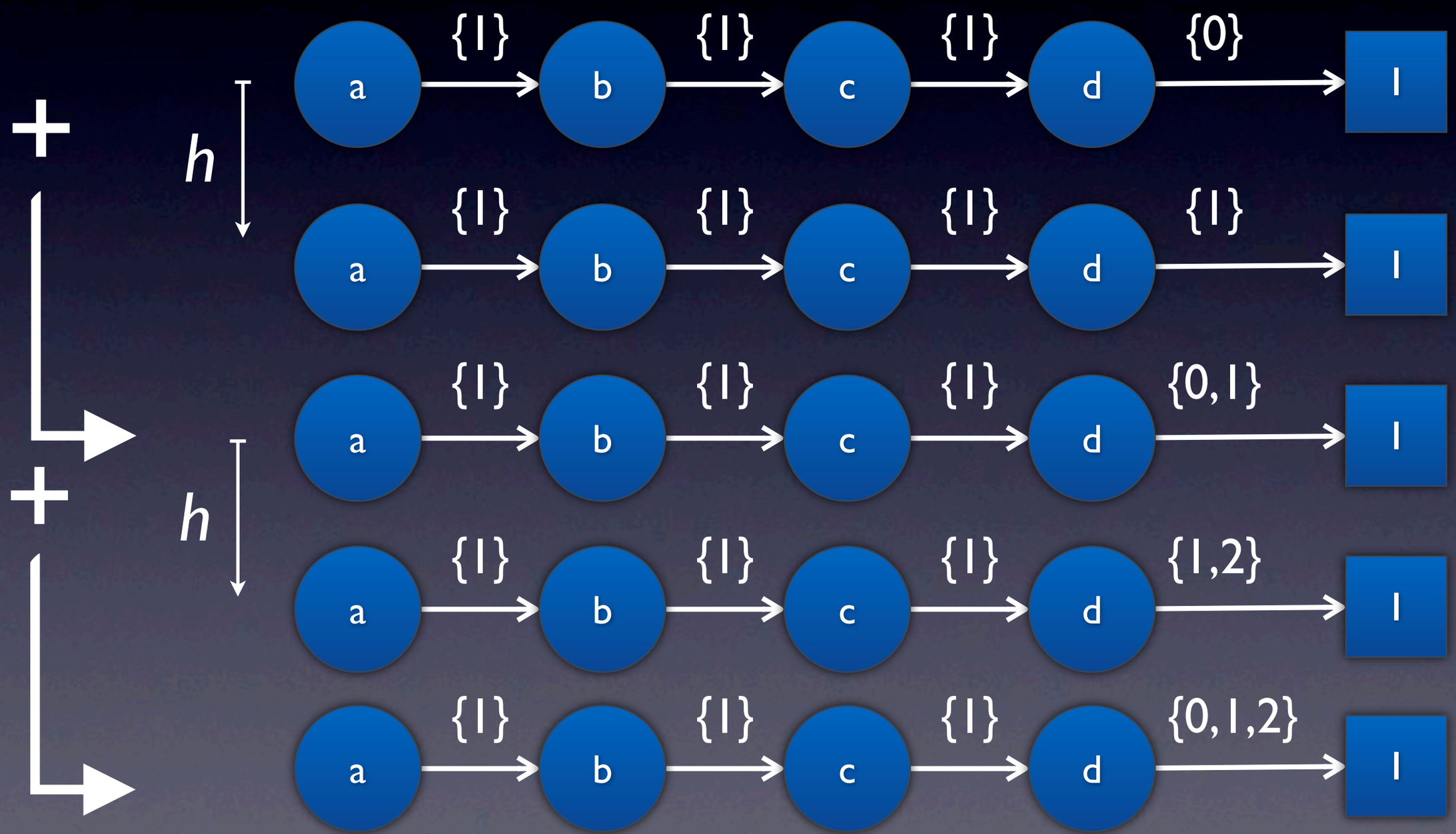
≤ n itérations

relation de franchissement éclatée

Nœuds inutiles

h : (d++) ∘ (d < 2)

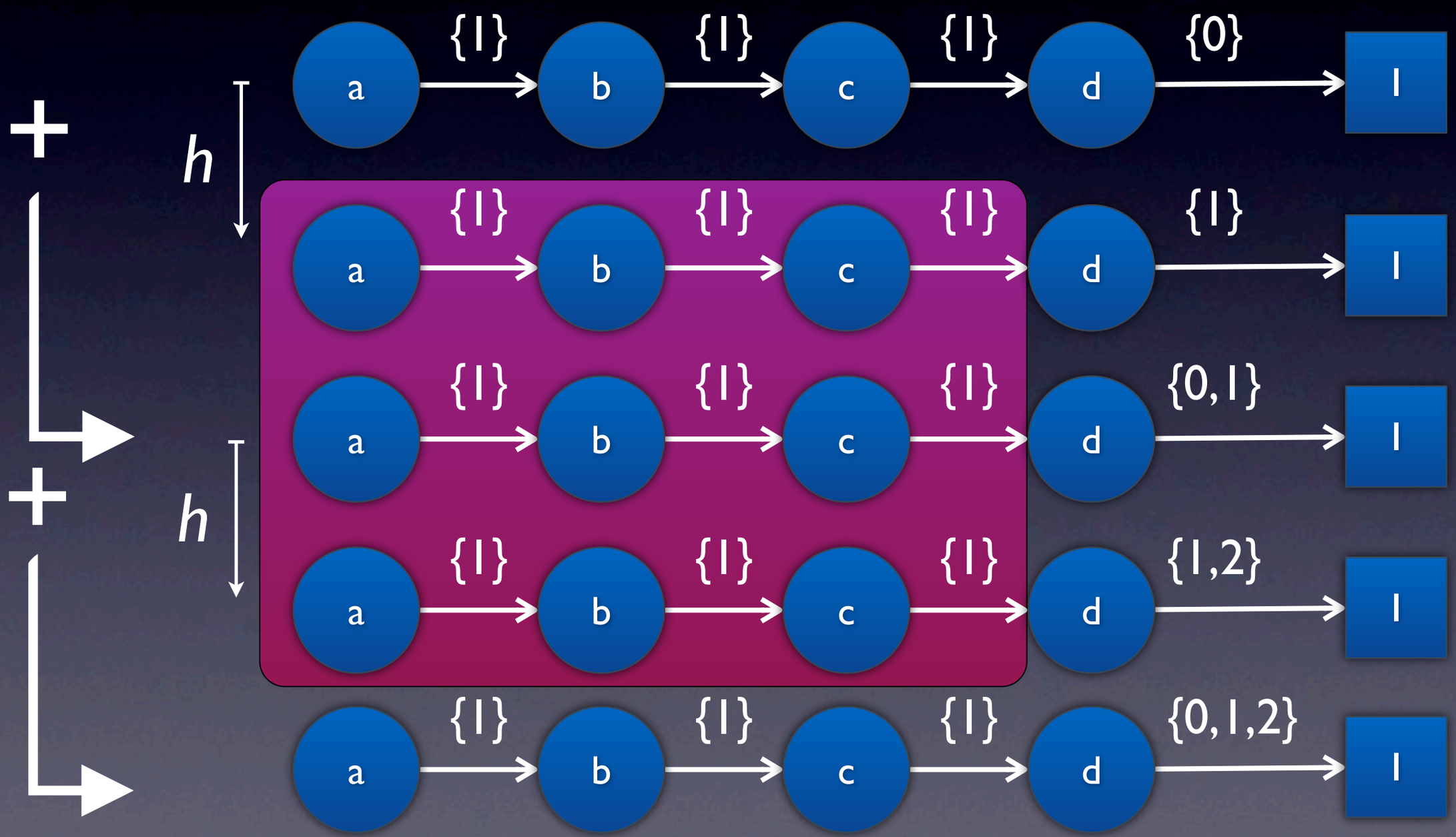
(h+Id)★



Nœuds inutiles

h : (d++) ∘ (d < 2)

(h+Id)★



Calcul d'un point fixe : saturation

- [Ciardo *et a*
- Gains expor
- Adapter le p
  - Groupes d'
  - Point fixe d
    ("saturer")
  - Si des varia
    les sature à

Generate(in s : array $[1..K]$ of *lcl*) : *node*

Build an MDD rooted at r, in $UT[K]$, encoding $\mathcal{N}^*_{\mathcal{E}}(\mathbf{s})$ and return r.

```
1. declare r, p : node;
2. declare l : lvl;
3. p ← 1;
4. for l = 1 to K do
5.    r ← NewNode(l);
6.    r[s[l]] ← p;
7.    Saturate(r);
8.    p ← r;
9. return r;
```

Saturate(in p : *node*)

Saturate node p and put it in $UT[p.lvl]$.

```
1. declare α : evnt;
2. declare f, u : node;
3. declare i, j : lcl;
4. declare F : set of evnt;
5. F ← E_{p.lvl};
6. while F ≠ ∅ do
7.    α ← Pick(F);
8.    foreach i ∈ Locals(α, p) do
9.       f ← Fire(α, p[i]);
10.      foreach j ∈ N_{p.lvl,α}(i) do
11.         u ← Union(f, p[j]);
12.         if u ≠ p[j] then
13.            p[j] ← u;
14.            F ← E_{p.lvl};
15. p ← Check(p);
```

Locals(in α : *evnt*, p : *node*) : set of *lcl*

Return $\{i \in \mathcal{S}_l : p[i] \neq z_{l-1} \wedge \mathcal{N}_{l,\alpha}(i) \neq \emptyset\}$, the local states in p, a node at level l, that are on a path to **1** and locally enable α. In particular, if p is a duplicate of z_l, return $\emptyset$.

Check(in p : *node*) : *node*

If $p[0] = \cdots = p[n_l - 1] = z_{l-1}$, delete p, a node at level l, and return z_l, since $\mathcal{B}(p)$ is $\emptyset$. If p duplicates q in $UT[l]$, delete p and return q. Otherwise, insert p in $UT[l]$ and return p.

Pick(inout $\mathcal{F}$: set of *evnt*) : *evnt*

Remove and return an element from $\mathcal{F}$.

Fire(in α : *evnt*, q : *node*) : *node*

Return s s.t. $\mathcal{B}(s) = \mathcal{N}^*_{\leq q.lvl}(\mathcal{N}_\alpha(\mathcal{B}(q)))$. It is called with $q.lvl < Top(\alpha)$.

```
1. declare f, s : node;
2. declare i, j : lcl;
3. if q.lvl < Bot(α) then return q;
4. if Find(FC[q.lvl],(q,α),s) then return s;
5. s ← NewNode(q.lvl);
6. foreach i ∈ Locals(α, q) do
7.    f ← Fire(α, q[i]);
8.    foreach j ∈ N_{l,α}(i) do
9.       s[j] ← Union(f, s[j]);
10. Saturate(s);
11. Insert(FC[q.lvl], (q, α), s);
12. return s;
```

Union(in p : *node*, q : *node*) : *node*

Assuming that $p.lvl = q.lvl = l$, build an MDD rooted at s encoding $\mathcal{B}(p) \cup \mathcal{B}(q)$, and return s, which is in $UT[l]$.

```
1. declare i : lcl;
2. declare s : node;
3. if p = z_{p.lvl} or p = q then return q;
4. if q = z_{q.lvl} then return p;
5. if Find(UC[p.lvl],{p,q},s) then return s;
6. s ← NewNode(p.lvl);
7. for i = 0 to n_{p.lvl} − 1 do
8.    s[i] ← Union(p[i], q[i]);
9. s ← Check(s);
10. Insert(UC[p.lvl], {p, q}, s);
11. return s;
```

NewNode(in l : *lvl*) : *node*

Create node p at level l with arcs set to z_{l-1}, return p.

Insert(inout *tab*, in *key*, *v*)

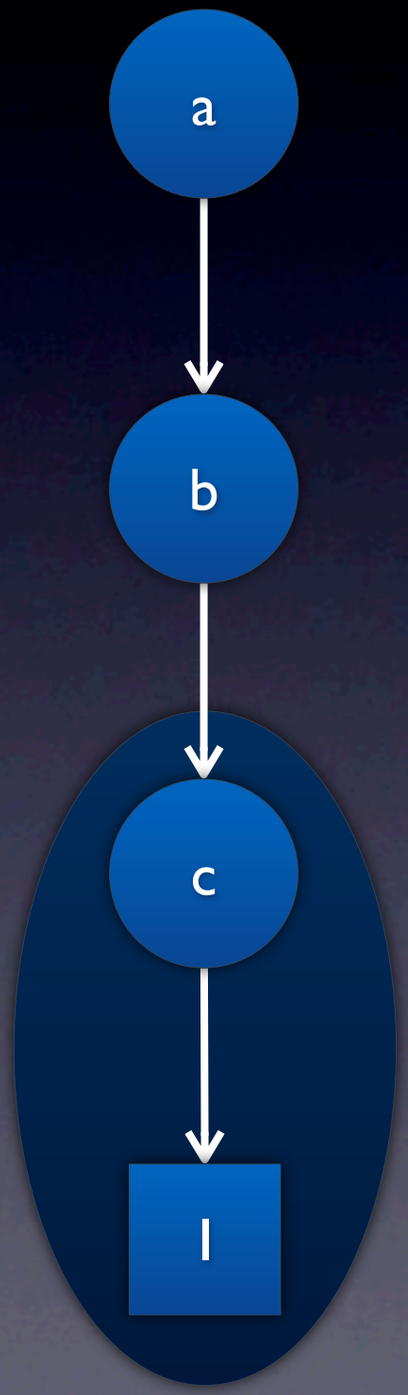
Insert (key, v) in hash table *tab*.

Find(in *tab*, *key*, out *v*) : *bool*

If (key, x) is in hash table *tab*, set v to x and return *true*. Otherwise, return *false*.

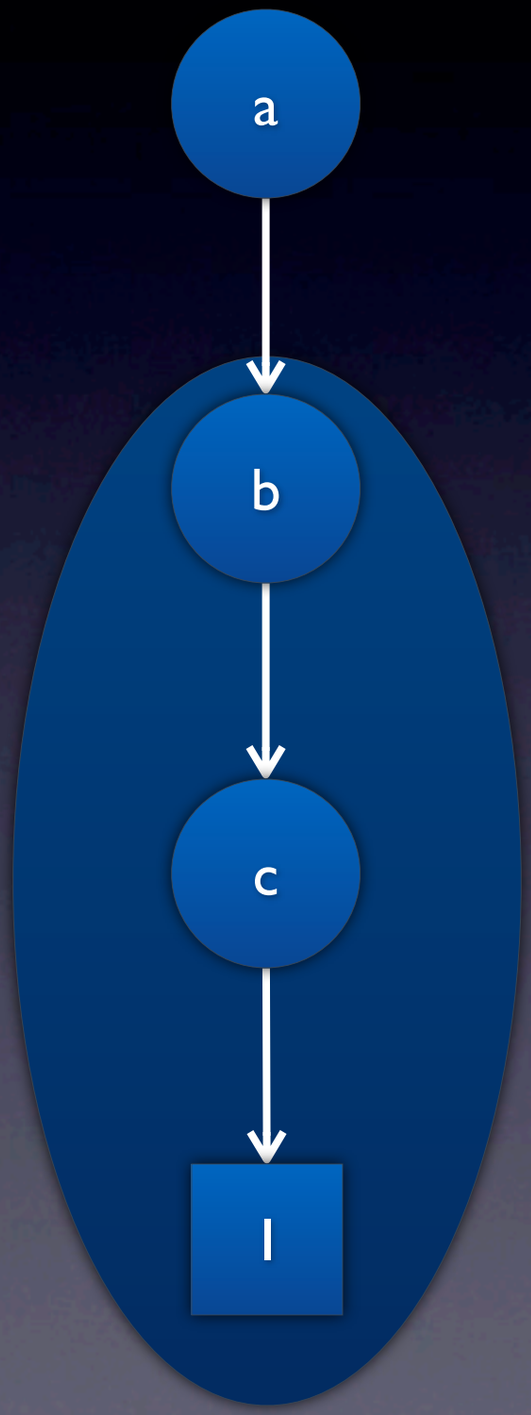
Calcul d'un point fixe : saturation

- [Ciardo *et al* TACAS'01]
- Gains exponentiels en pratique
- Adapter le point fixe à la structure du DD
  - Groupes d'actions touchant une variable
  - Point fixe des actions associées à cette variable ("saturer")
  - Si des variables plus basses sont touchées, on les sature à nouveau



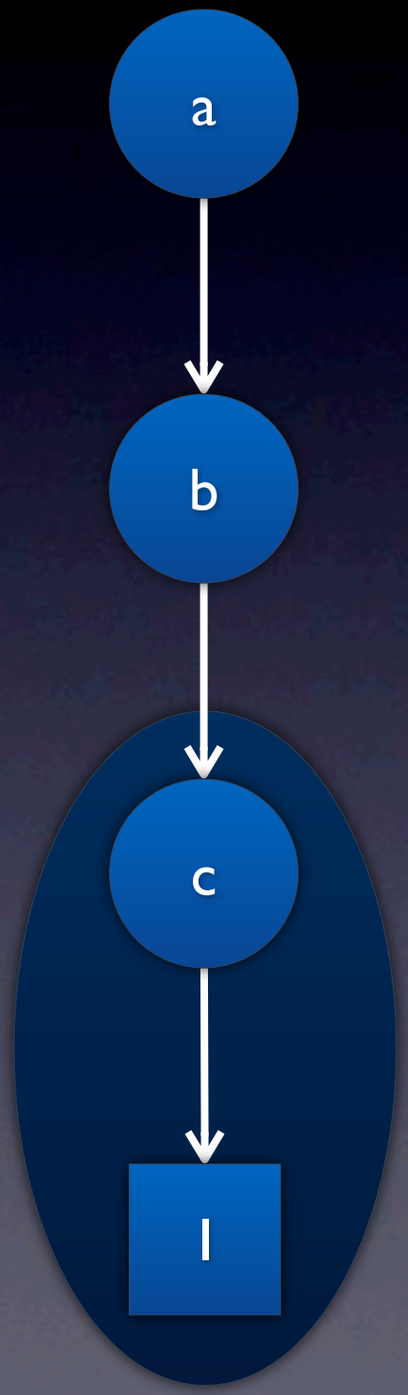
Calcul d'un point fixe : saturation

- [Ciardo *et al* TACAS'01]
- Gains exponentiels en pratique
- Adapter le point fixe à la structure du DD
  - Groupes d'actions touchant une variable
  - Point fixe des actions associées à cette variable ("saturer")
  - Si des variables plus basses sont touchées, on les sature à nouveau



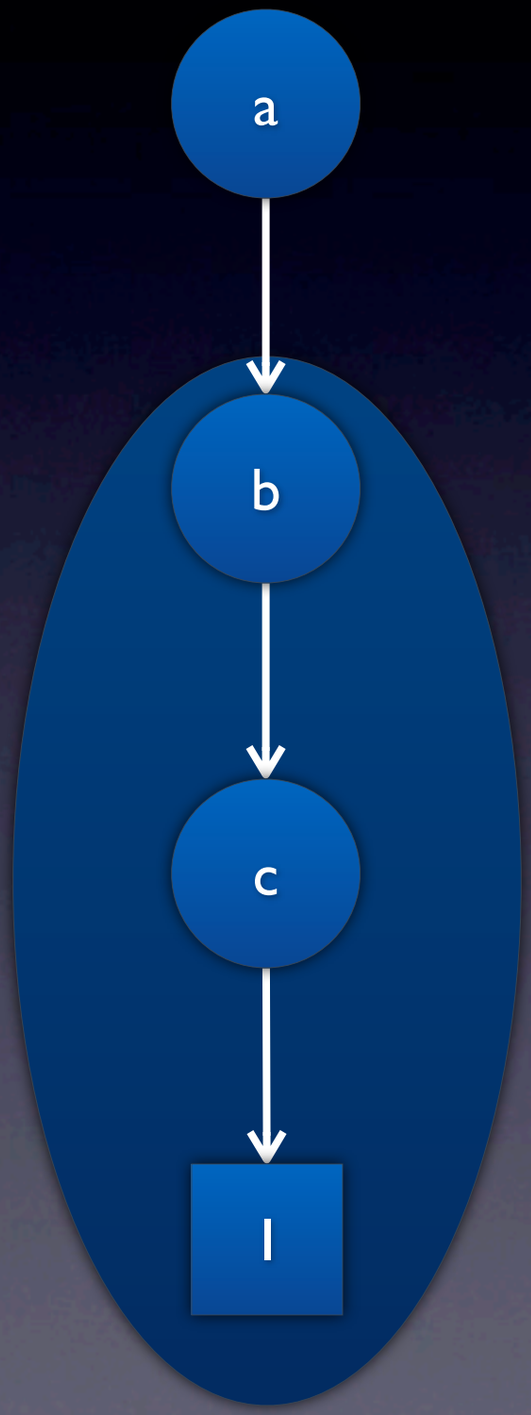
Calcul d'un point fixe : saturation

- [Ciardo *et al* TACAS'01]
- Gains exponentiels en pratique
- Adapter le point fixe à la structure du DD
  - Groupes d'actions touchant une variable
  - Point fixe des actions associées à cette variable ("saturer")
  - Si des variables plus basses sont touchées, on les sature à nouveau



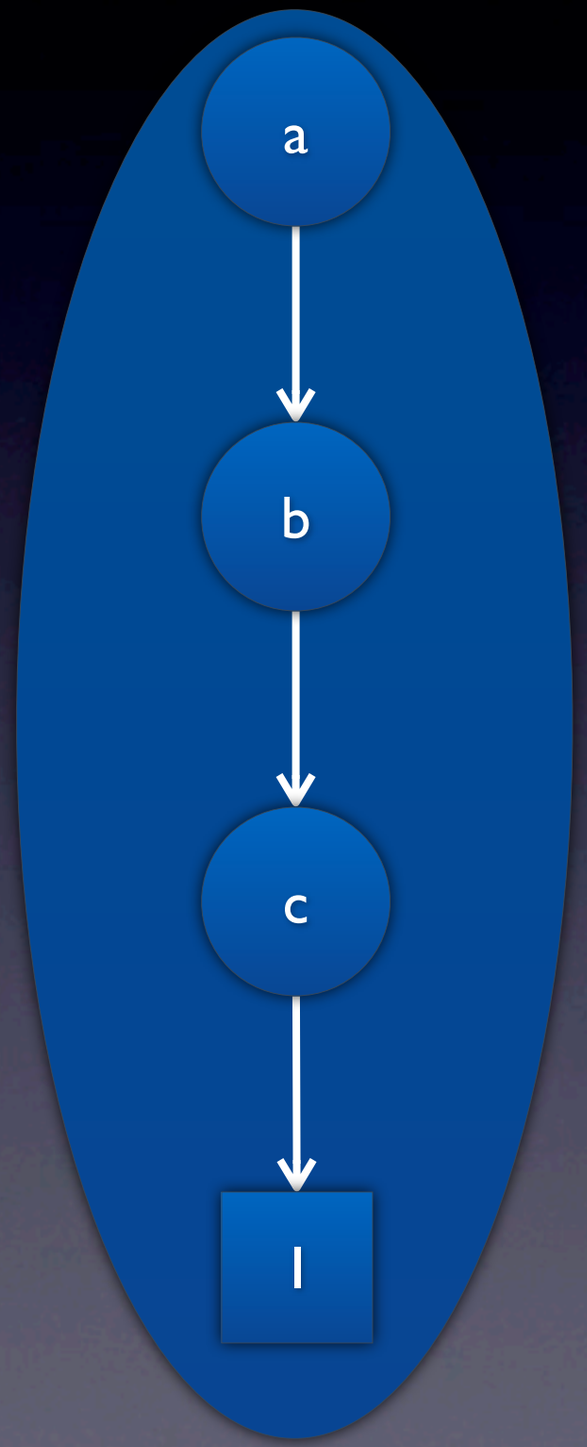
Calcul d'un point fixe : saturation

- [Ciardo *et al* TACAS'01]
- Gains exponentiels en pratique
- Adapter le point fixe à la structure du DD
  - Groupes d'actions touchant une variable
  - Point fixe des actions associées à cette variable ("saturer")
  - Si des variables plus basses sont touchées, on les sature à nouveau



Calcul d'un point fixe : saturation

- [Ciardo *et al* TACAS'01]
- Gains exponentiels en pratique
- Adapter le point fixe à la structure du DD
  - Groupes d'actions touchant une variable
  - Point fixe des actions associées à cette variable ("saturer")
  - Si des variables plus basses sont touchées, on les sature à nouveau



Calcul d'un point fixe : saturation

- [Ciardo *et a*
- Gains expor
- Adapter le p
  - Groupes d'
  - Point fixe d
    ("saturer")
  - Si des varia
    les sature à

Generate(in s : array $[1..K]$ of *lcl*) : *node*

Build an MDD rooted at r, in $UT[K]$, encoding $\mathcal{N}^*_{\mathcal{E}}(\mathbf{s})$ and return r.

```
1.  declare r, p : node;
2.  declare l : lvl;
3.  p ← 1;
4.  for l = 1 to K do
5.     r ← NewNode(l);
6.     r[s[l]] ← p;
7.     Saturate(r);
8.     p ← r;
9.  return r;
```

Saturate(in p : *node*)

Saturate node p and put it in $UT[p.lvl]$.

```
1.  declare α : evnt;
2.  declare f, u : node;
3.  declare i, j : lcl;
4.  declare F : set of evnt;
5.  F ← E_{p.lvl};
6.  while F ≠ ∅ do
7.     α ← Pick(F);
8.     foreach i ∈ Locals(α, p) do
9.        f ← Fire(α, p[i]);
10.       foreach j ∈ N_{p.lvl,α}(i) do
11.          u ← Union(f, p[j]);
12.          if u ≠ p[j] then
13.             p[j] ← u;
14.             F ← E_{p.lvl};
15. p ← Check(p);
```

Locals(in α : *evnt*, p : *node*) : set of *lcl*

Return $\{i \in \mathcal{S}_l : p[i] \neq z_{l-1} \wedge \mathcal{N}_{l,\alpha}(i) \neq \emptyset\}$, the local states in p, a node at level l, that are on a path to $\mathbf{1}$ and locally enable α. In particular, if p is a duplicate of z_l, return $\emptyset$.

Check(in p : *node*) : *node*

If $p[0] = \cdots = p[n_l - 1] = z_{l-1}$, delete p, a node at level l, and return z_l, since $\mathcal{B}(p)$ is $\emptyset$. If p duplicates q in $UT[l]$, delete p and return q. Otherwise, insert p in $UT[l]$ and return p.

Pick(inout $\mathcal{F}$: set of *evnt*) : *evnt*

Remove and return an element from $\mathcal{F}$.

Fire(in α : *evnt*, q : *node*) : *node*

Return s s.t. $\mathcal{B}(s) = \mathcal{N}^*_{\leq q.lvl}(\mathcal{N}_\alpha(\mathcal{B}(q)))$. It is called with $q.lvl < Top(\alpha)$.

```
1.  declare f, s : node;
2.  declare i, j : lcl;
3.  if q.lvl < Bot(α) then return q;
4.  if Find(FC[q.lvl],(q,α),s) then return s;
5.  s ← NewNode(q.lvl);
6.  foreach i ∈ Locals(α, q) do
7.     f ← Fire(α, q[i]);
8.     foreach j ∈ N_{l,α}(i) do
9.        s[j] ← Union(f, s[j]);
10. Saturate(s);
11. Insert(FC[q.lvl], (q, α), s);
12. return s;
```

Union(in p : *node*, q : *node*) : *node*

Assuming that $p.lvl = q.lvl = l$, build an MDD rooted at s encoding $\mathcal{B}(p) \cup \mathcal{B}(q)$, and return s, which is in $UT[l]$.

```
1.  declare i : lcl;
2.  declare s : node;
3.  if p = z_{p.lvl} or p = q then return q;
4.  if q = z_{q.lvl} then return p;
5.  if Find(UC[p.lvl],{p,q},s) then return s;
6.  s ← NewNode(p.lvl);
7.  for i = 0 to n_{p.lvl} − 1 do
8.     s[i] ← Union(p[i], q[i]);
9.  s ← Check(s);
10. Insert(UC[p.lvl], {p, q}, s);
11. return s;
```

NewNode(in l : *lvl*) : *node*

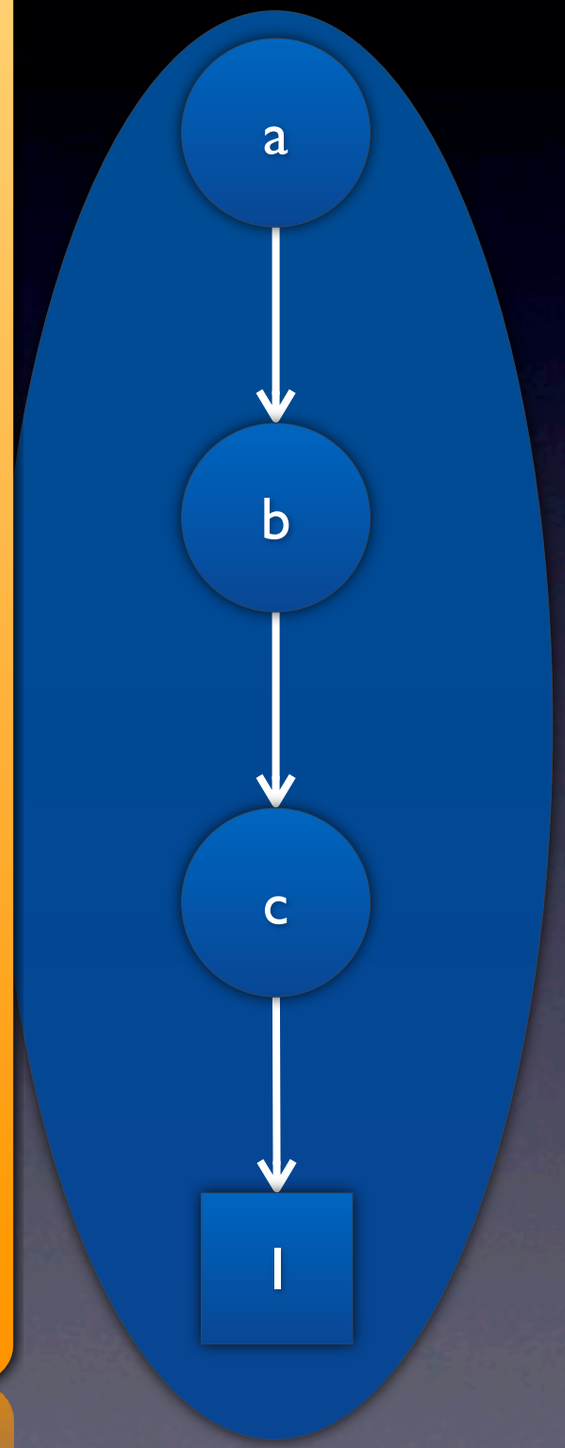
Create node p at level l with arcs set to z_{l-1}, return p.

Insert(inout *tab*, in *key*, *v*)

Insert (key, v) in hash table *tab*.

Find(in *tab*, *key*, out *v*) : *bool*

If (key, x) is in hash table *tab*, set v to x and return *true*. Otherwise, return *false*.



Calcul d'un point fixe : saturation

- [Ciardo *et al* TACAS'01]
- Gains exponentiels en pratique
- Adapter le point fixe à la structure du DD

$$(t_1 + \ldots + t_n + Id)^\star$$

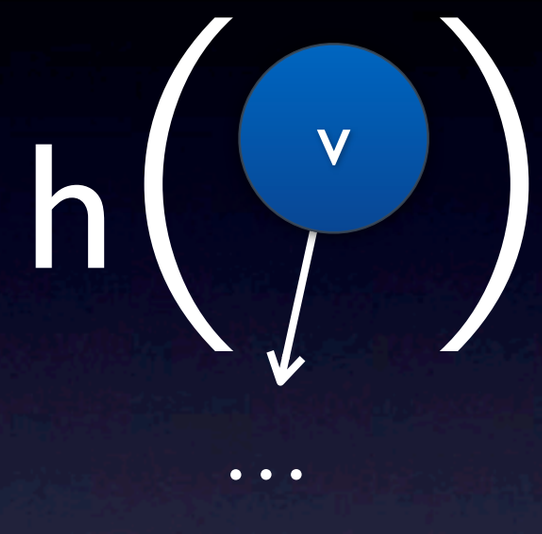
  - ("saturer")
  - Si des variables plus basses sont touchées, on les sature à nouveau

Contribution : saturation automatique

- Automatisation
  - Transparence
- Règles de réécritures
  - Motifs de point fixe - *i*.e. $(t_1 + t_2 + \ldots + \mathrm{Id})^\star$
  - Généralisation
  - Invariance locale : critère de regroupement des opérations
- Implémentation au sein de libddd

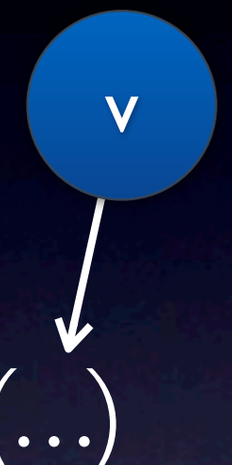
Invariance locale pour automatiser la saturation

- Prédicat *skip(h,v)* exprimant l'invariance locale d'un homomorphisme *h*
  - *v* n'est ni lue, ni modifiée par *h*
- Propagation sans modification de l'opération sur les successeurs
- Prédicat utilisé pour déclencher la saturation automatique
  - À définir pour les homomorphismes inductifs
  - Prédéfini pour les homomorphismes de base et les compositions



Invariance locale pour automatiser la saturation

- Prédicat *skip(h,v)* exprimant l'invariance locale d'un homomorphisme *h*
  - *v* n'est ni lue, ni modifiée par *h*
- Propagation sans modification de l'opération sur les successeurs
- Prédicat utilisé pour déclencher la saturation automatique
  - À définir pour les homomorphismes inductifs
  - Prédéfini pour les homomorphismes de base et les compositions



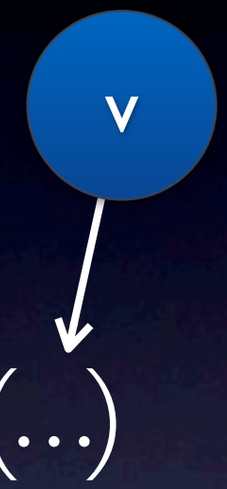
Invariance locale pour automatiser la saturation

- Prédicat *skip(h,v)* exprimant l'invariance locale d'un homomorphisme *h*
  - *v* n'est ni lue, ni modifiée par *h*
- Propagation sans modification de l'opération sur les successeurs
- Prédicat utilisé pour déclencher la saturation automatique
  - À définir pour les homomorphismes inductifs

d++

Invariance locale pour automatiser la saturation

- Prédicat *skip(h,v)* exprimant l'invariance locale d'un homomorphisme *h*
  - *v* n'est ni lue, ni modifiée par *h*



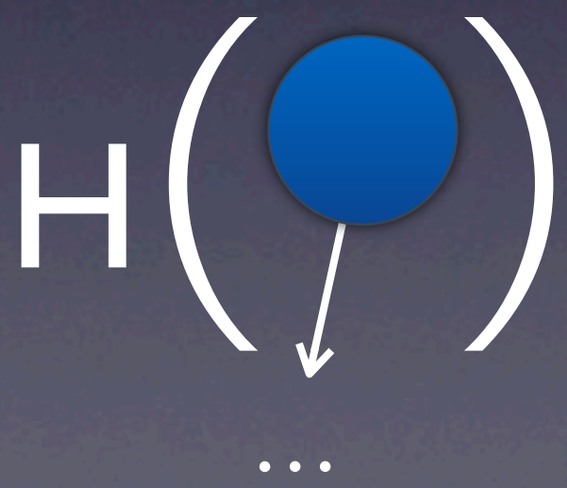
- Propagation sans modification de l'opération sur les successeurs
- Prédicat utilisé pour déclencher la saturation automatique
  - À définir pour les homomorphismes inductifs
  - Prédéfini pour les homomorphismes de base et les compositions

d++

(d++) o (d < 2)

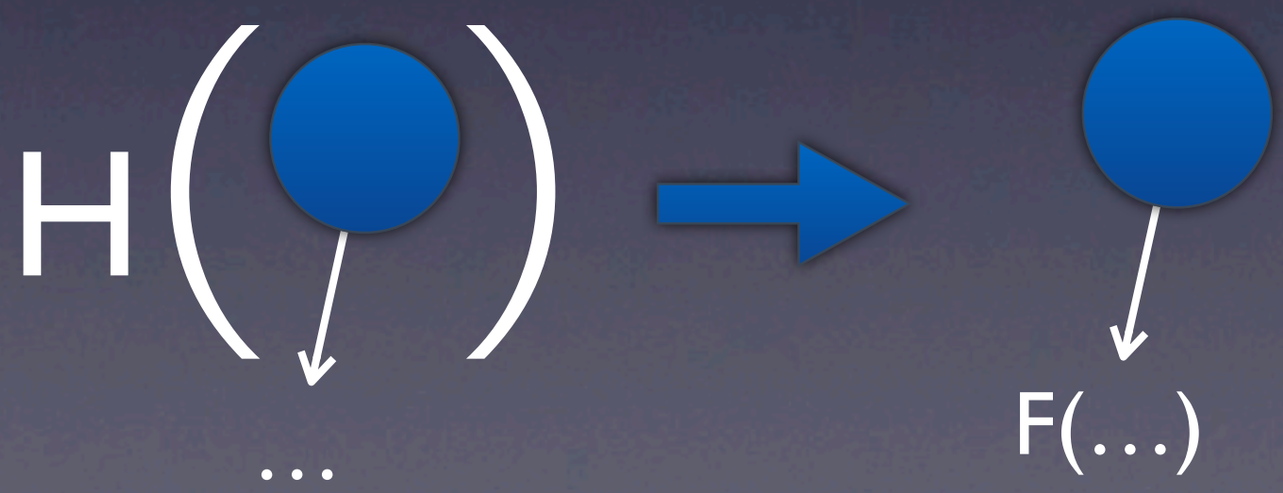
Réécriture d'union d'homomorphismes

- $H = h_1 + h_2 + \ldots + h_n$
- Partition des opérations $h_1, h_2, \ldots, h_n$
  - F : opérations localement invariantes
  - G = H \ F



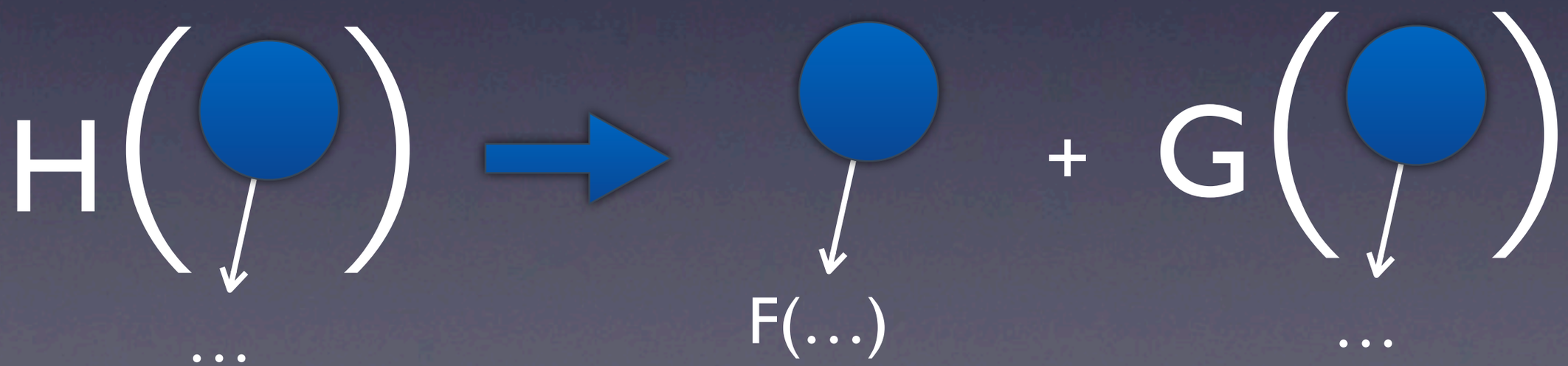
Réécriture d'union d'homomorphismes

- $H = h_1 + h_2 + \ldots + h_n$
- Partition des opérations $h_1, h_2, \ldots, h_n$
  - F : opérations localement invariantes
  - G = H \ F



Réécriture d'union d'homomorphismes

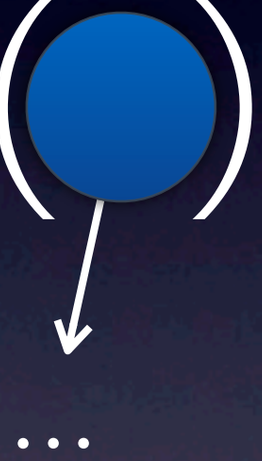
- $H = h_1 + h_2 + \ldots + h_n$
- Partition des opérations $h_1, h_2, \ldots, h_n$
  - F : opérations localement invariantes
  - G = H \ F



Réécriture de points fixes

Réécriture de points fixes

$$(G + F + Id)^{\star}(\bullet)$$



…

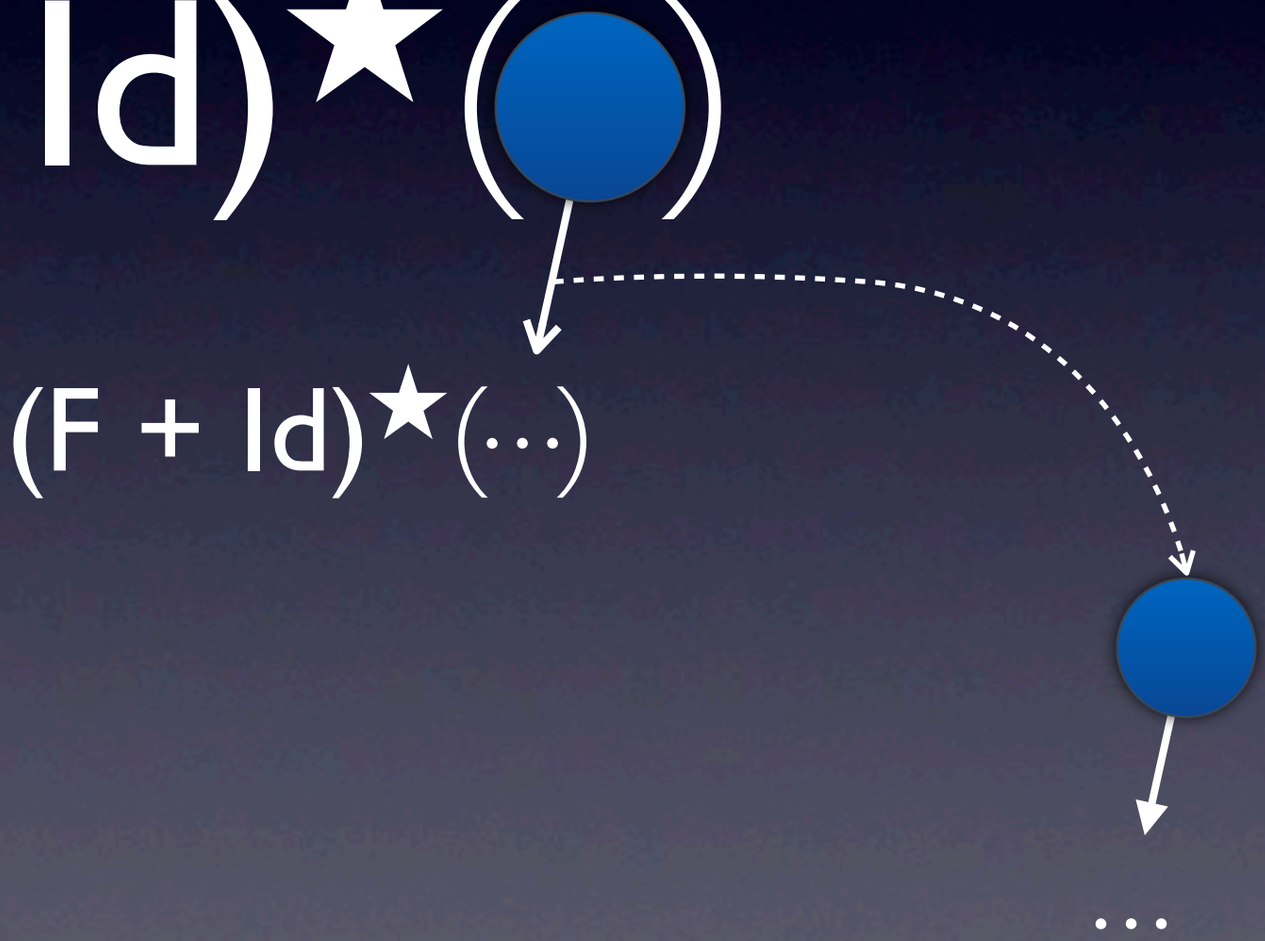
Réécriture de points fixes

$$(G + Id)^{\star}(\bullet)$$

$$(F + Id)^{\star}(\dots)$$

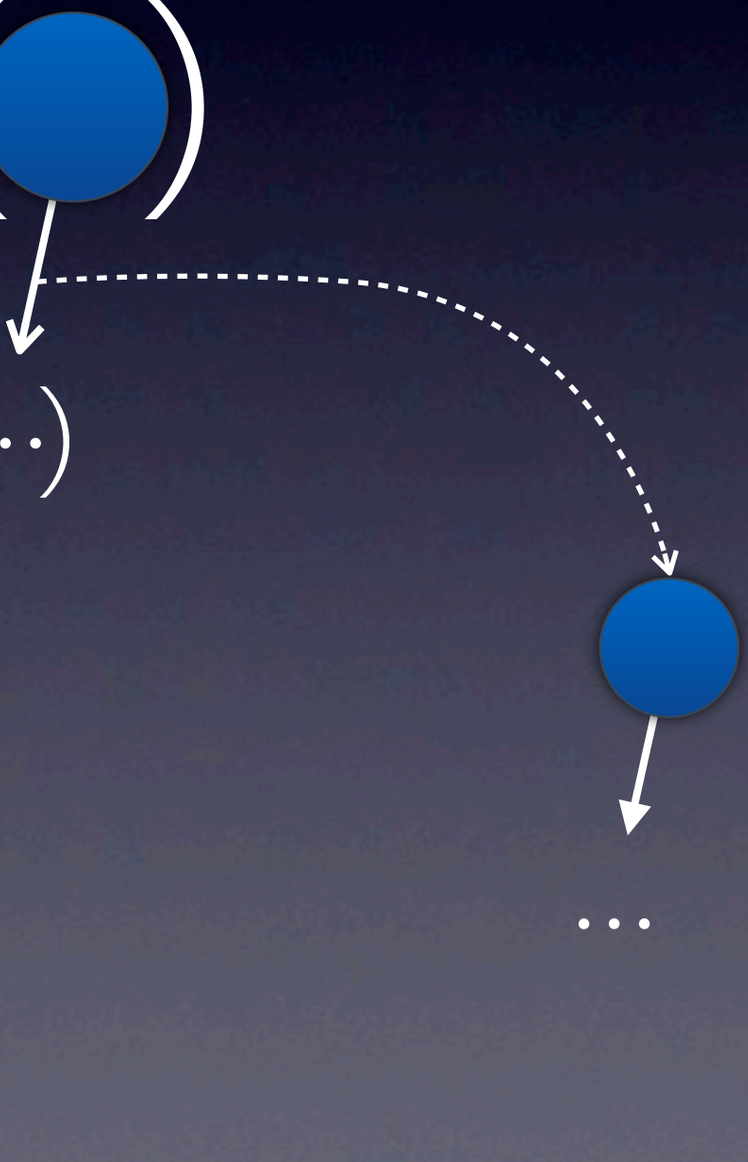
Réécriture de points fixes

$$(G + Id)^{\star}(\bullet)$$



Réécriture de points fixes

$(G' + L + Id)^{\bigstar}(\bullet)$



Réécriture de points fixes

$(G' + Id)^{\star}(\bullet)$

$(F + Id)^{\star}(\dots)$

$(L + Id)^{\star}(\bullet)$

…

Réécriture de points fixes

$(O_{g \in G'}(g + Id))^{\bigstar}(\bullet)$

$(F + Id)^{\bigstar}(\dots)$

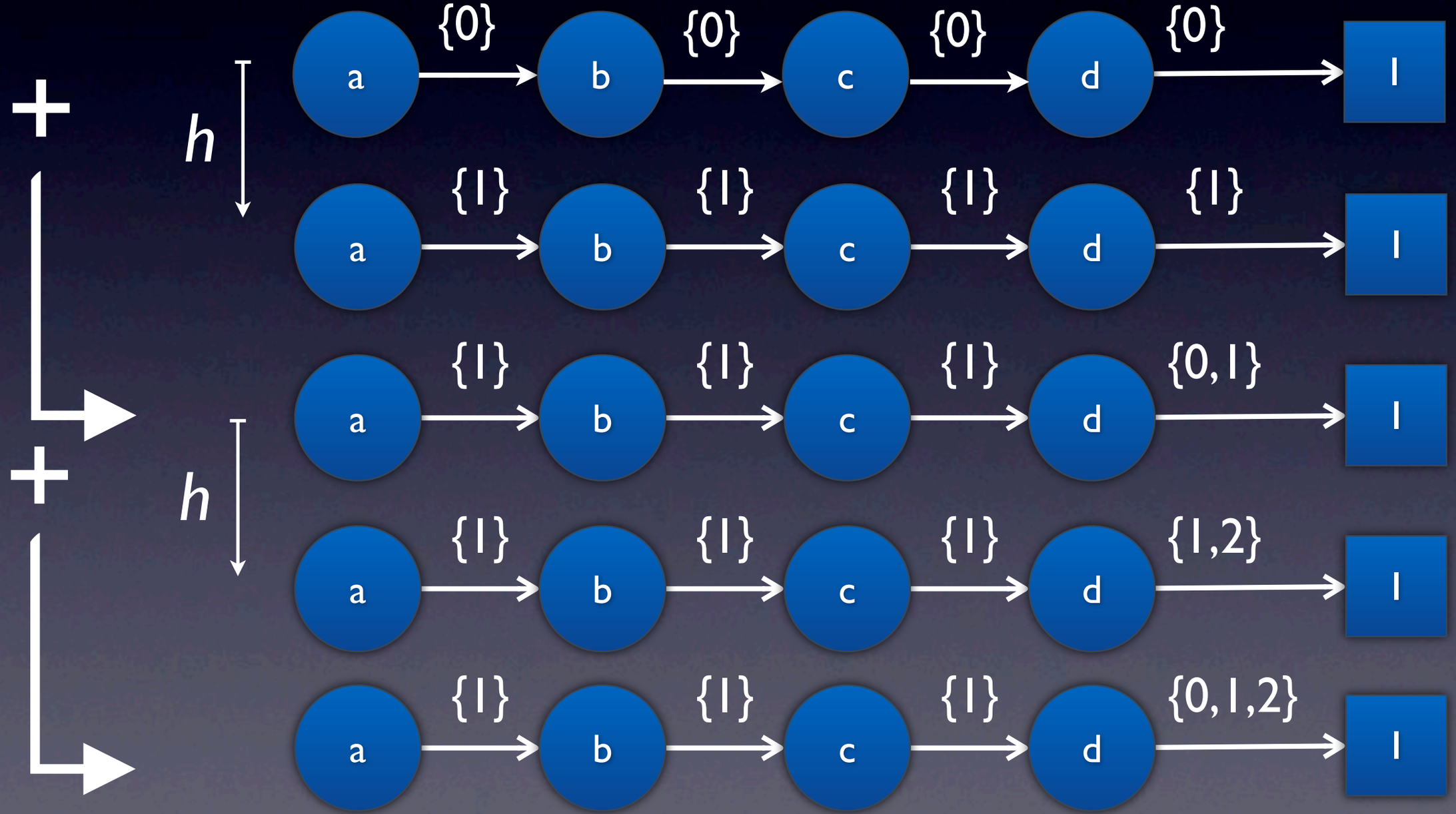
$(L + Id)^{\bigstar}(\bullet)$

…

Exemple d'application de la saturation automatique

h : (d++) o (d < 2)

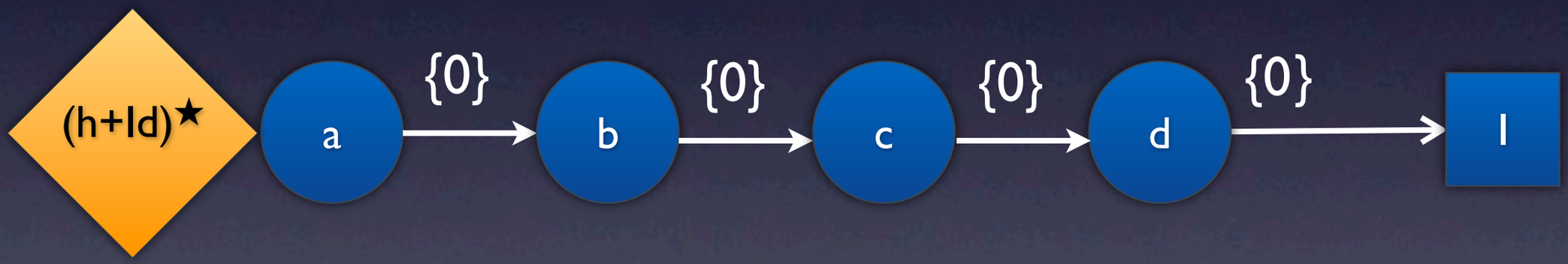
(h+Id)★



Exemple d'application de la saturation automatique

h : (d++) o (d < 2)

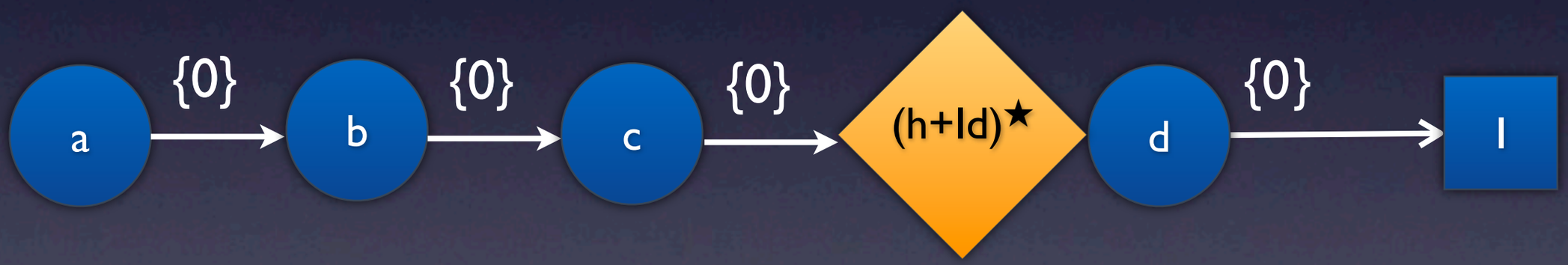
(h+Id)★



Exemple d'application de la saturation automatique

h : (d++) o (d < 2)

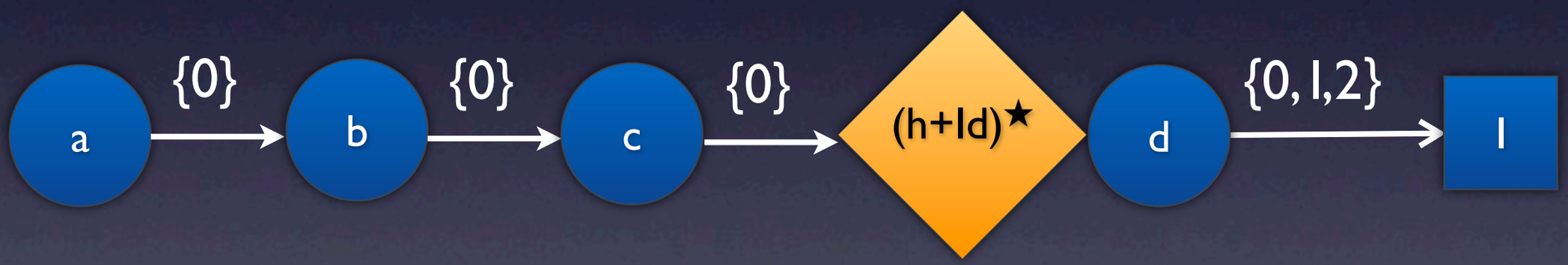
(h+Id)★



Exemple d'application de la saturation automatique

h : (d++) o (d < 2)

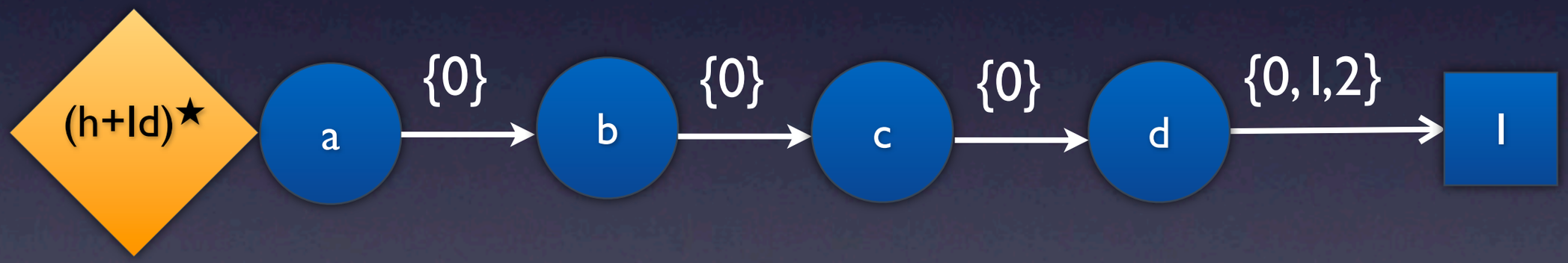
(h+Id)★



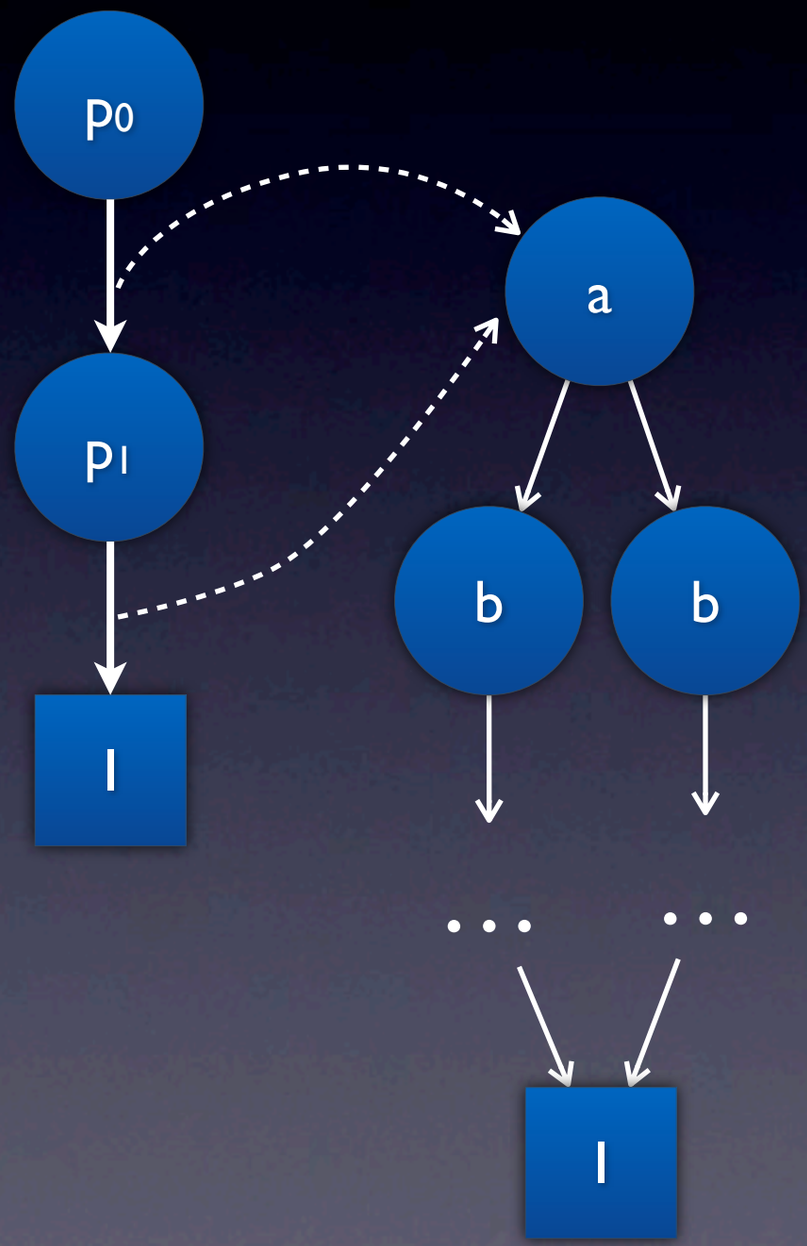
Exemple d'application de la saturation automatique

h : (d++) o (d < 2)

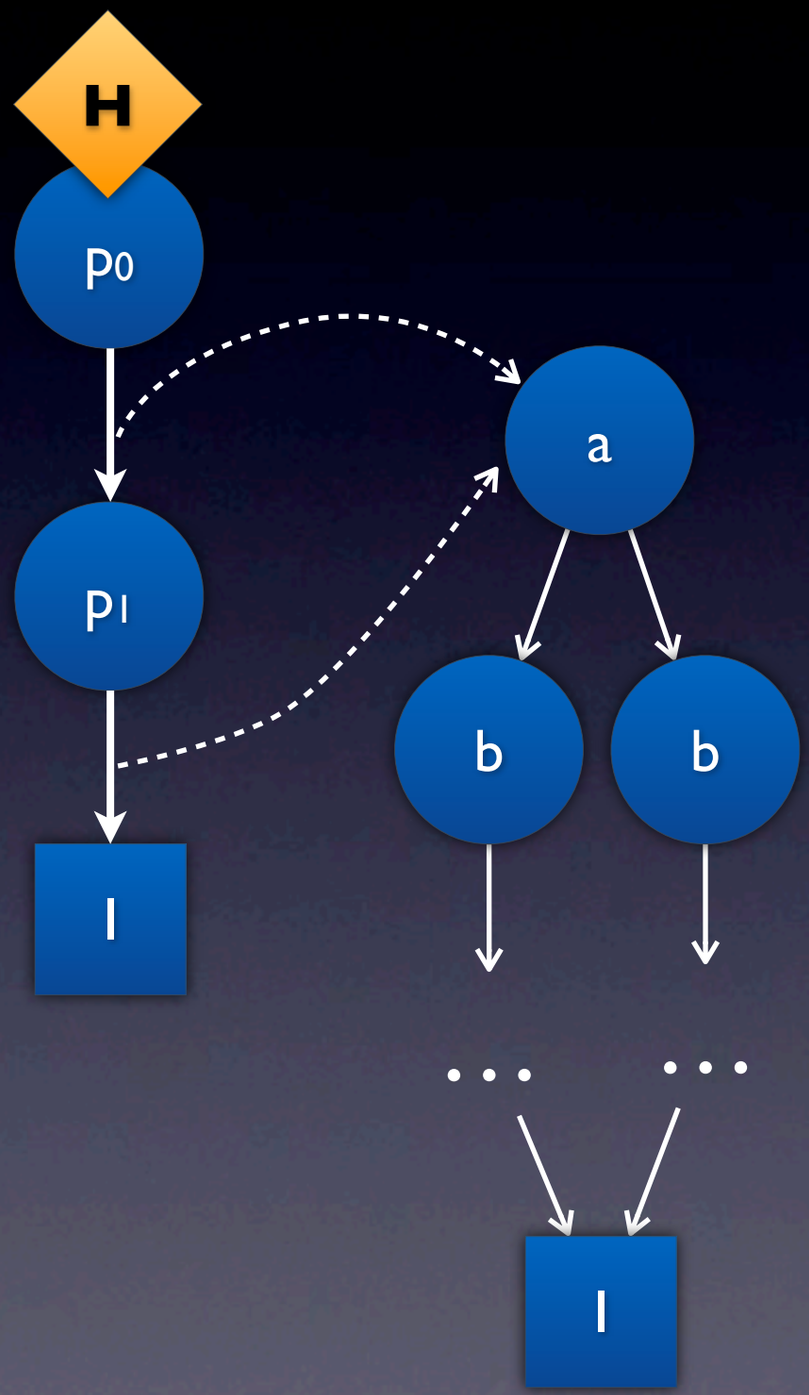
(h+Id)★



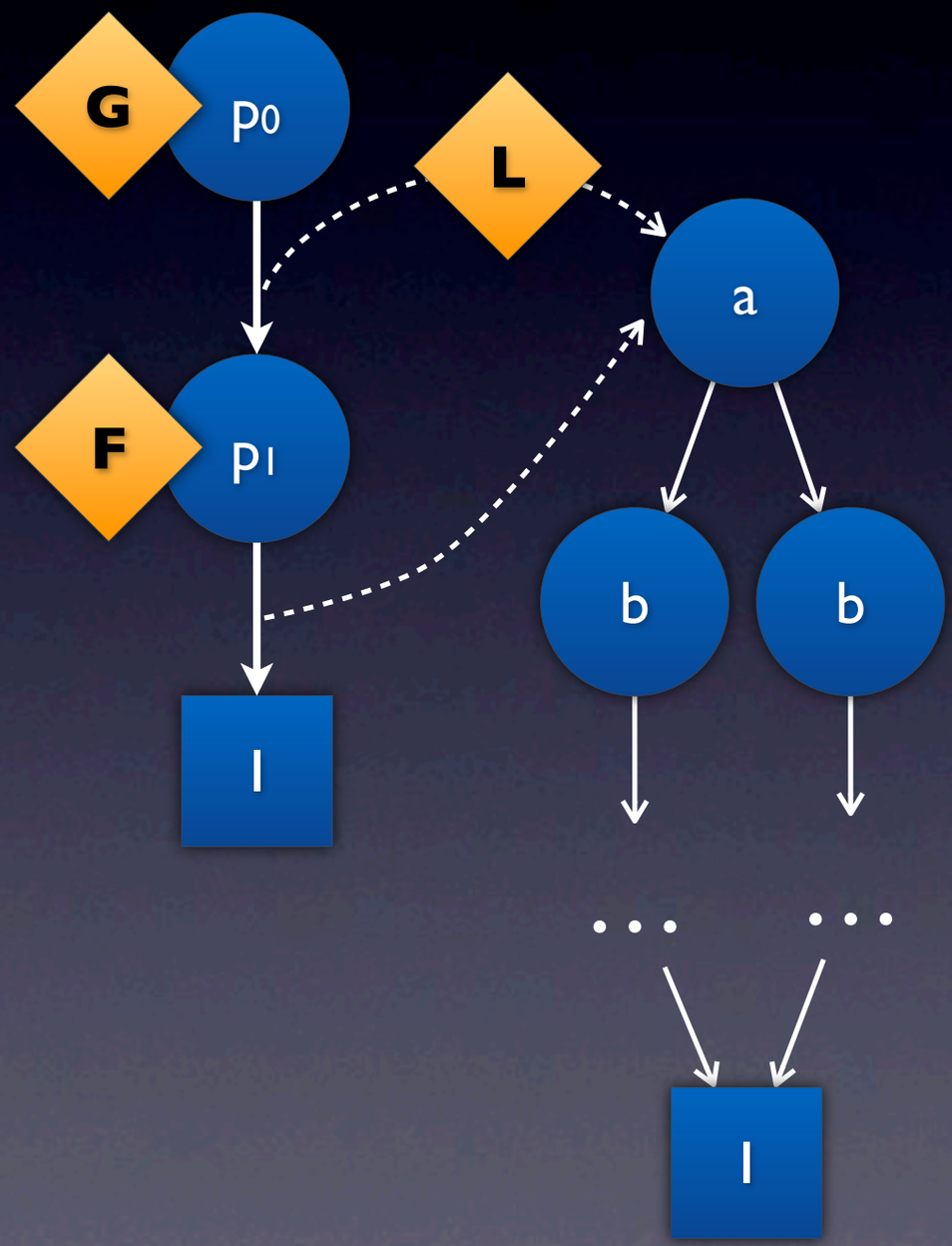
Modèles hiérarchiques et réguliers



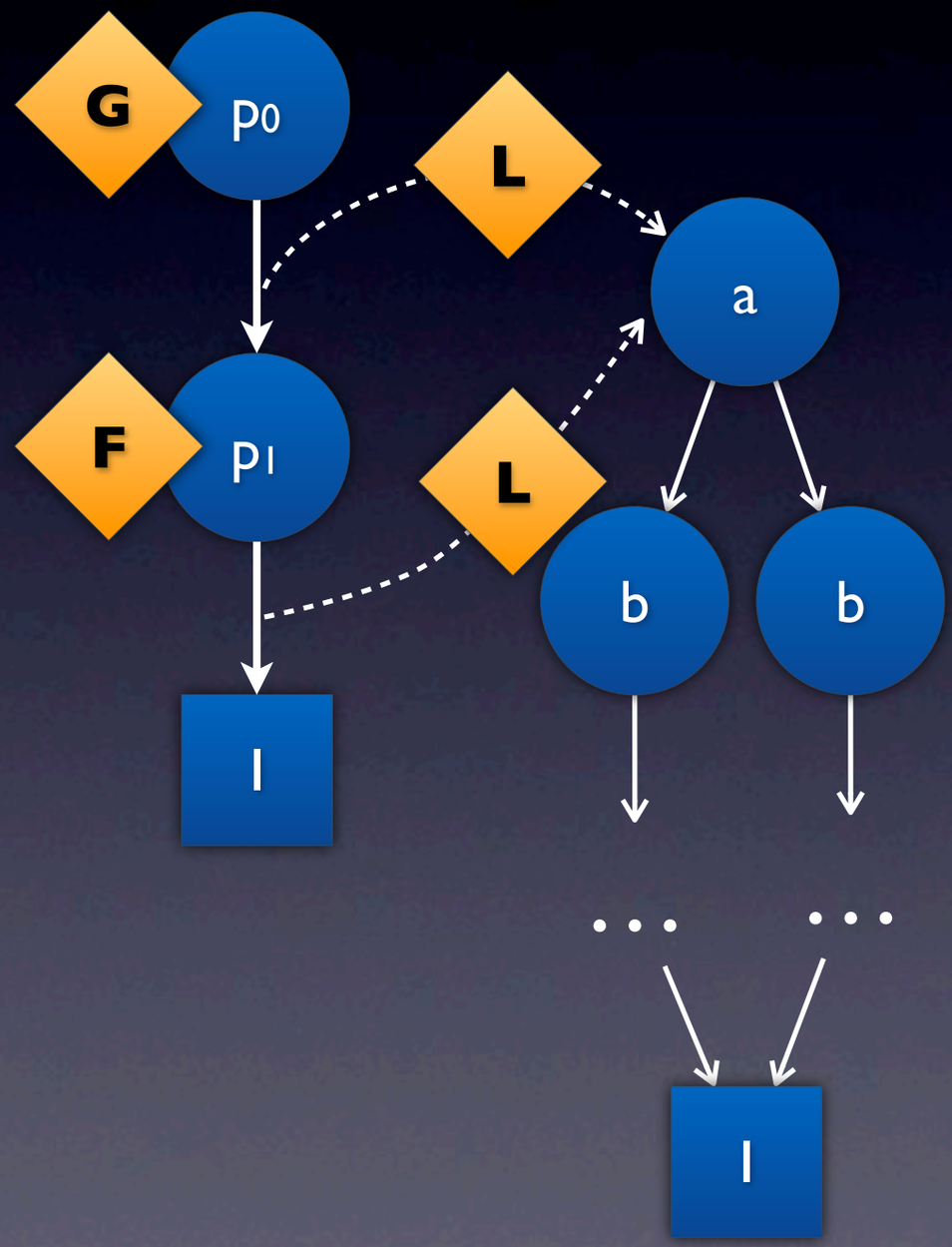
Modèles hiérarchiques et réguliers



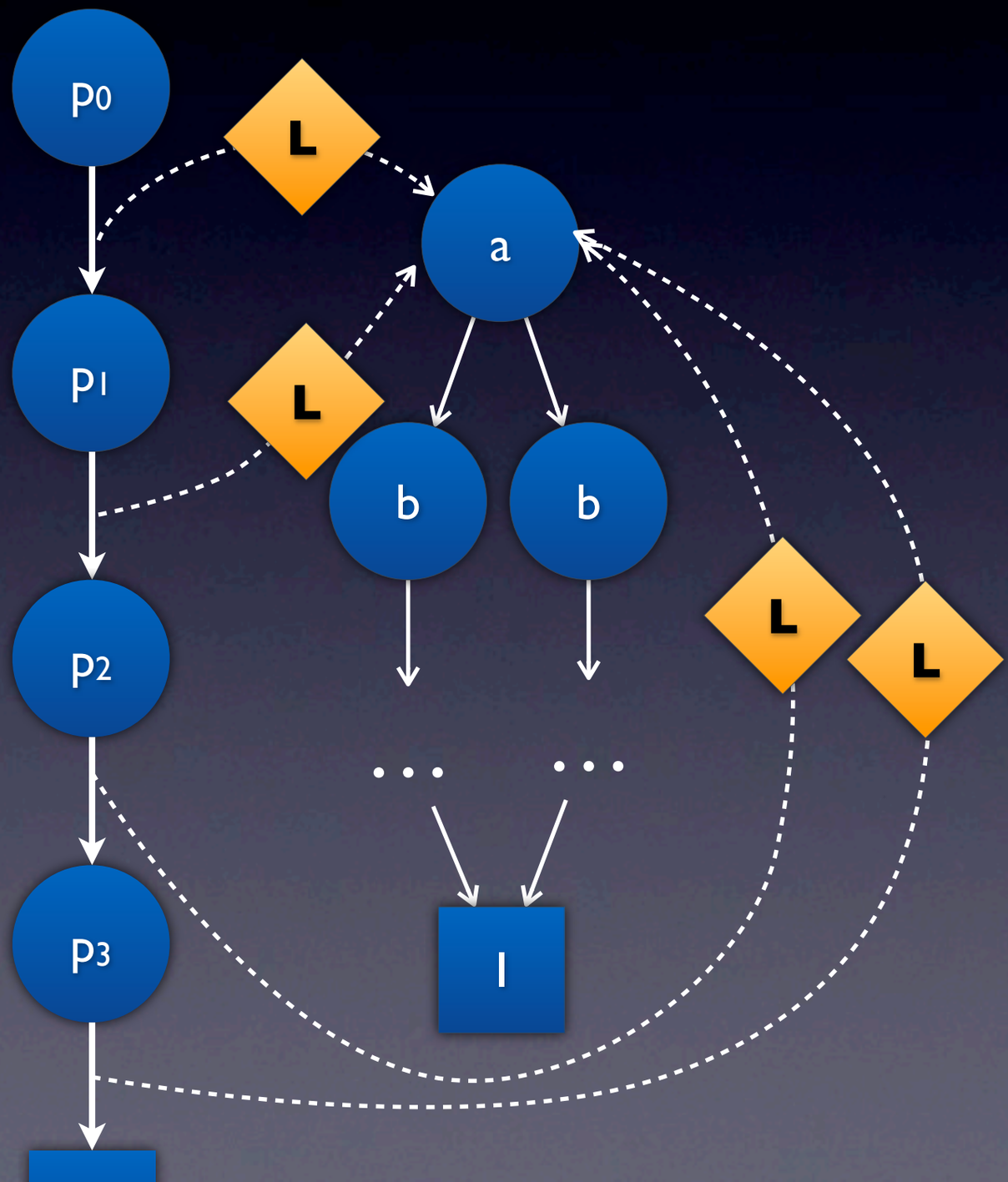
Modèles hiérarchiques et réguliers



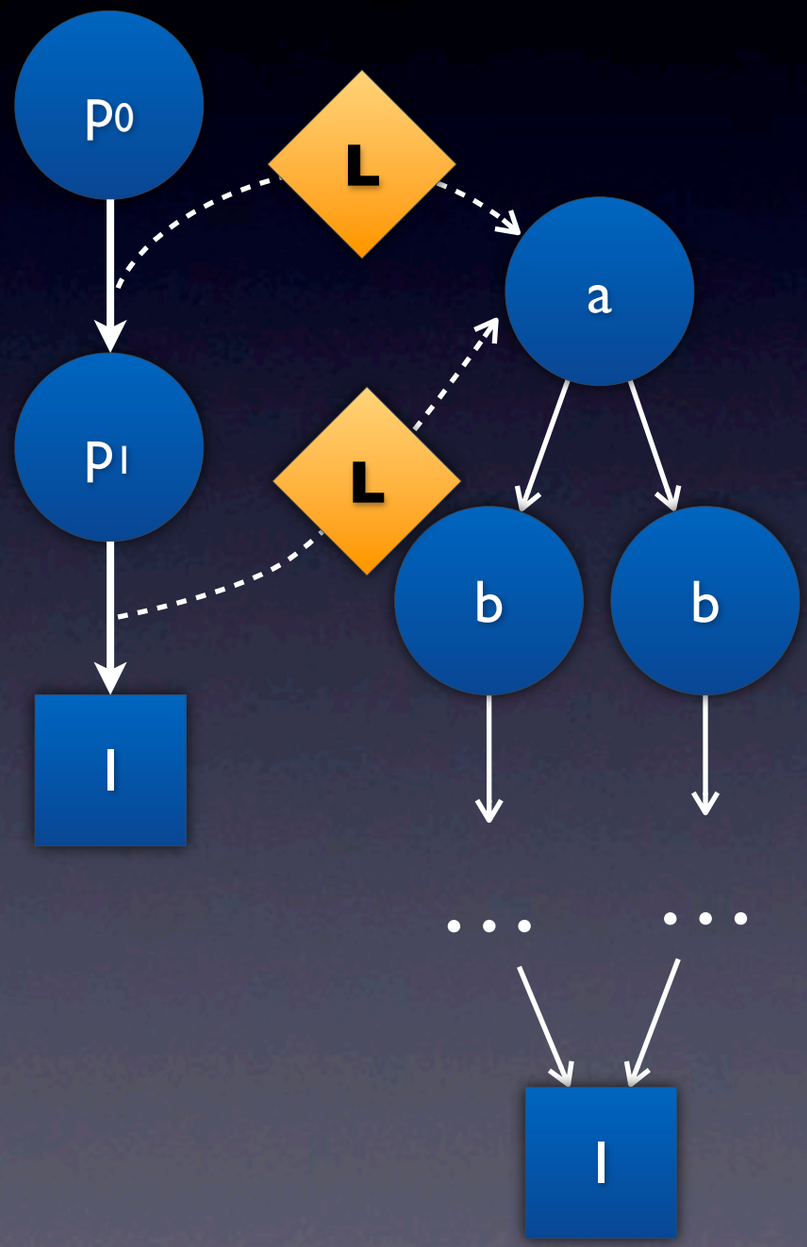
Modèles hiérarchiques et réguliers



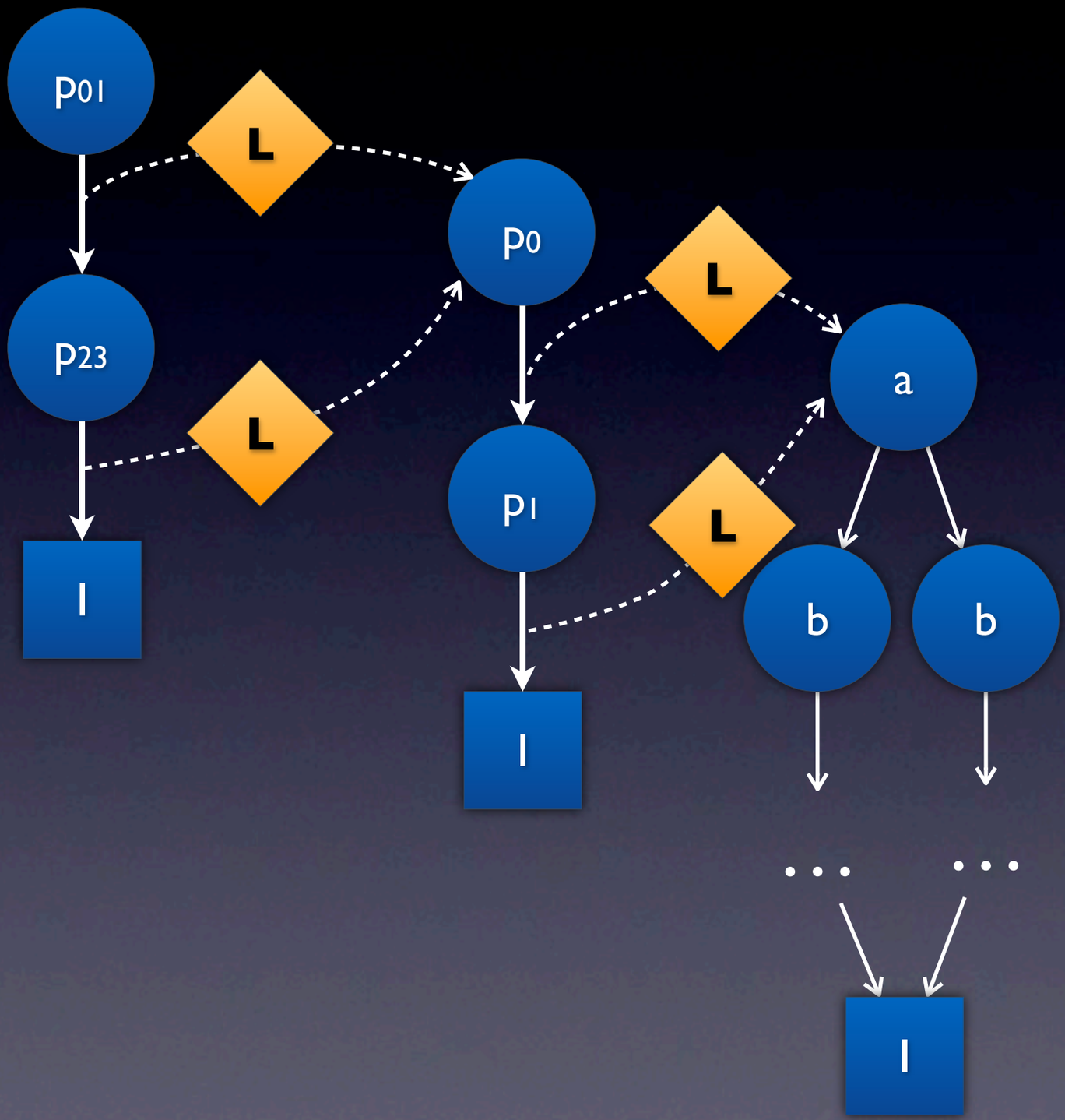
Modèles hiérarchiques et réguliers



Modèles hiérarchiques et réguliers



Modèles hiérarchiques et réguliers



Expérimentations : nombre de variables fixe

		BFS		Sat. auto		SMART	
Taille	États	*s*	*Mo*	*s*	*Mo*	*s*	*Mo*
FMS							
50	$4{,}2\ 10^{17}$	13,0	430,0	0,4	15,5	543,6	881,2
250	$3{,}4\ 10^{26}$	-	-	14,9	275,5	-	-
700	$2{,}1\ 10^{32}$	-	-	220,5	2062,4	-	-
Kanban							
100	$1{,}7\ 10^{19}$	12,0	145,0	0,38	10,4	-	-
500	$7{,}1\ 10^{26}$	-	-	44,7	194,7	-	-
1000	$1{,}4\ 10^{30}$	-	-	574,4	758,6	-	-

Expérimentations : nombre de variables non-fixe

		Enc. plat		Enc. rég.		SMART	
Taille	États	*s*	*Mo*	*s*	*Mo*	*s*	*Mo*
Ring							
10	$8{,}3\ 10^{09}$	0,03	3,5	0,01	1,7	0,20	7,0
100	$2{,}6\ 10^{105}$	22,0	816,0	16,35	324,8	33,0	92,4
250	$1{,}6\ 10^{265}$	-	-	-	-	528,8	651,8
Philosophes							
100	$4{,}9\ 10^{62}$	0,07	5,2	0,01	1,6	0,60	53,0
1000	$9{,}2\ 10^{626}$	4,3	115,0	0,01	1,7	6,50	631,8
10^{3000}	-	-	-	36,0	386,0	-	-

Saturation automatique : synthèse

- **Automatisation** de la saturation
- Ensemble de réécritures extensibles
- Accès **transparent** à la saturation pour les concepteurs de model checkers symboliques
  - Orthogonalité à l'algorithme
  - Logiques temporelles CTL/LTL
- Performances excellentes
- Applicable à d'autres diagrammes de décision

Conclusion

- Deux contributions au model checking
- Parallélisme et répartition
  - Accélérations linéaires
- Saturation
  - Automatisation et Généralisation
- Transparence à l'utilisateur
- Bibliothèques opérationnelles
  - C++, GPL, http://ddd.lip6.fr

Perspectives

- Extension continue des règles de réécritures
- Parallélisation et répartition des SDD
- Ordonnancement des variables
  - Utiliser la structuration des opérations
  - Découpage hiérarchique

Merci!

Parallélisme et répartition

[1] A. Hamez, F. Kordon, and Y. Thierry-Mieg. **libdmc: a Library to Operate Efficient Distributed Model Checking.** *In IPDPS, pages 1–8. IEEE, 2007.*

[2] A. Hamez, F. Kordon, Y. Thierry-Mieg, and F. Legond-Aubry. **dmcG : A Distributed Symbolic Model Checker Based on GreatSPN.** *Petri Nets and Other Models of Concurrency –ICATPN 2007, pages 495–504, 2007.*

Saturation automatique

[3] A. Hamez, Y. Thierry-Mieg, and F. Kordon. **Hierarchical Set Decision Diagrams and Automatic Saturation.** *Applications and Theory of Petri Nets, pages 211–230, 2008.*

[4] Y. Thierry-Mieg, D. Poitrenaud, A. Hamez, and F. Kordon. **Hierarchical Set Decision Diagrams and Regular Models.** *In TACAS, pages 1–15, 2009.*

[5] A. Hamez, Y. Thierry-Mieg, and F. Kordon. **Building Efficient Model Checkers Using Hierarchical Set Decision Diagrams and Automatic Saturation.** *(Journal, selected from best papers of PN'08) Fundamenta Informaticae, 94(3-4):413–437, 2009.*

Autres

[6] A. Hamez, L. Hillah, F. Kordon, A. Linard, E. Paviot-Adet, X. Renault, and Y. Thierry-Mieg. **New Features in CPN-AMI 3: Focusing on the Analysis of Complex Distributed Systems**. *ACSD, 0:273–275, 2006.*