

Vérification sur modèle

J. Leneutre

jean.leneutre@telecom-paristech.fr

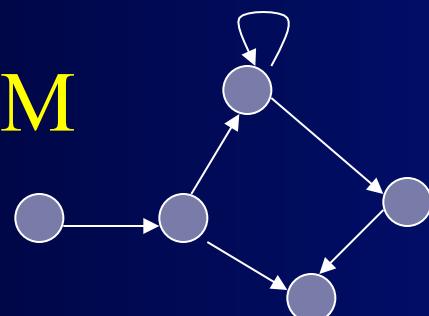
Objectif

- Modèle : description formelle (d'une partie) du comportement du système
 - Spécification : description formelle des propriétés attendues du système
 - Vérification : le modèle satisfait-il sa spécification ?

Système



M



Propriétés



Formalisation

?

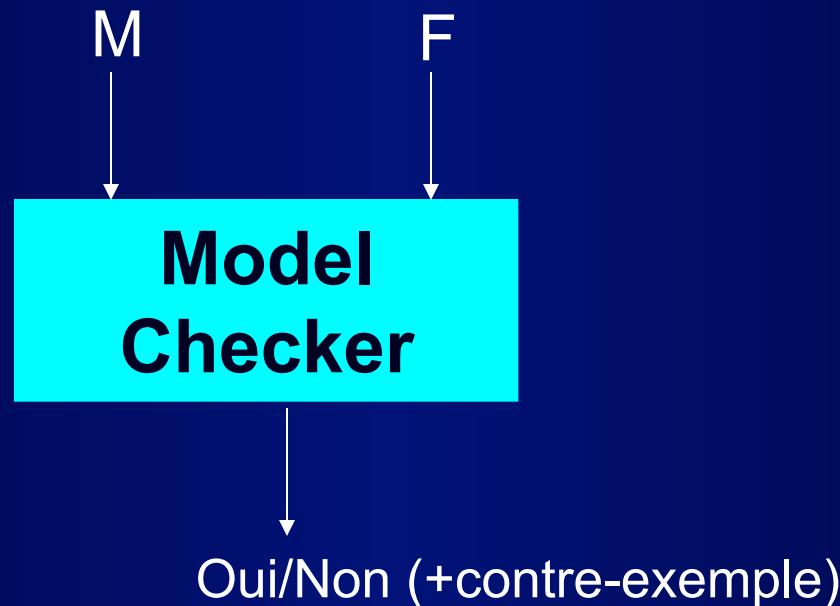
1

F



Objectif

- Model checking : vérifier $M \models F$ par un simple calcul



- Approche totalement automatisée
- Obtenir un contre-exemple si la propriété n'est pas vérifiée

Modèle comportemental

- On prend comme modèle M du système, un automate (machine à états finis).
- Définition : un automate fini (FSA) est un quadruplet $\langle e0, Q, A, T \rangle$
 - Q est un ensemble d' états, contenant $e0$, l' état initial, et un sous-ensemble F d' états finaux
 - A est un ensemble d' actions (ou étiquettes)
 - $T \subseteq Q \times A \times Q$ est l' ensemble des transitions
- **Système de transitions** : sémantique d' une machine à états
- **Structure de Kripke** : on abandonne l' étiquetage des transitions dans le système de transitions et on considère un ensemble fini de prédictats atomiques, PA, et L une fonction associant à chaque état de Q une partie de PA
- Exemple : au tableau



Rappels

- Définition : le langage reconnu par un automate est l' ensemble des mots qui terminent dans un état final
- Définition : Le produit cartésien de deux automates $\langle e0', Q', A', T', F' \rangle$ et $\langle e0'', Q'', A'', T'', F'' \rangle$ est l' automate $\langle e0, Q, A, T, F \rangle$ tel que
 - $Q = Q' \times Q''$, $e0 = (e0', e0'')$,
 - $F = (F' \times Q'') \cup (Q' \times F'')$
 - $A = A' \times A''$
 - $T = \{((q1', q1''), (a', a''), (q2', q2'')), \text{ tels que } (q1', a', q2') \in T' \text{ et } (q1'', a'', q2'') \in T''\}$

Spécification des propriétés

- Propriétés du distributeur de billets :
 - « après trois erreurs successives dans le pin code, la carte est capturée »
 - « si les billets sont délivrés, alors la carte est éjectée »
 - « quand la carte est éjectée, une sonnerie est produite jusqu’ à ce que l’ utilisateur récupère sa carte »
 - « si les billets sont délivrés, le pin code correct a été entré auparavant »
- La logique temporelle permet de formaliser naturellement ces propriétés

Plan et référence

Réf : « Vérification de logiciels », P. Schnoebelen & allii , Vuibert, 1999.

- Logiques temporelles (LTL, CTL, CTL*)
- Les différentes classes de propriétés
- Procédures de Model Checking (LTL, CTL)
- Un exemple de langage/outil de model checking :
Promela/Spin

Logiques temporelles

Logique temporelle : motivations

- Exemple : distributeur de billet
 - « Si les billets sont délivrés, alors la carte est éjectée » =
 - « Après toute introduction d' une nouvelle carte, si les billets sont délivrés, cette carte sera éjectée »
- Formalisation en logique du premier ordre :
 - Intro_carte(t) : la carte est introduite à l' instant t
 - Billets(t) : les billets sont distribués à l' instant t
 - Eject_carte(t) : la carte est éjectée à l' instant t
$$\forall t . [\text{Intro_carte}(t) \wedge (\exists t' > t. (\text{Billets}(t') \wedge (\forall t'' . t < t'' \leq t' \Rightarrow \neg \text{Intro_carte}(t''))))]$$
$$\Rightarrow \exists t' > t. [\text{Eject_carte}(t') \wedge (\forall t'' . t < t'' \leq t' \Rightarrow \neg \text{Intro_carte}(t''))]$$
- Utilisation d' un paramètre modélisant le temps
- Très lourd !

Logique temporelle

- Logique temporelle :
 - forme de logique spécialisée intégrant la notion d' ordonnancement dans le temps (plus besoin de paramètre t)
 - opérateurs primitifs permettant d' exprimer des notions telles que « toujours », « tant que », ...
 - permet d' énoncer des propriétés sur les exécutions d' un système
- Il y a plusieurs logiques temporelles :
 - Logiques du temps linéaire (exple PLTL, Propositional Linear Temporal Logic utilisée dans SPIN) :
 - à chaque instant, il ne peut succéder qu' un seul futur
 - Logique temporelles du temps arborescent (« branching-time », exple CTL, Computation Tree Logic)
 - à chaque instant, il peut succéder plusieurs futurs

PLTL : formules

- Définition : les formules de PLTL sont définies par la grammaire suivante :

$\Phi, \Psi ::= p \mid q \mid \dots \mid \text{true} \mid \text{false}$ (propositions atomiques)

$\Phi \wedge \Psi \mid \Phi \vee \Psi \mid \Phi \Rightarrow \Psi \mid \neg \Phi \mid$ (connecteurs booléens)

$\mathbf{F}\Phi \mid \mathbf{G}\Phi \mid \Phi \mathbf{U} \Psi \mid \mathbf{X}\Phi$ (connecteurs temporels)

- Notations :

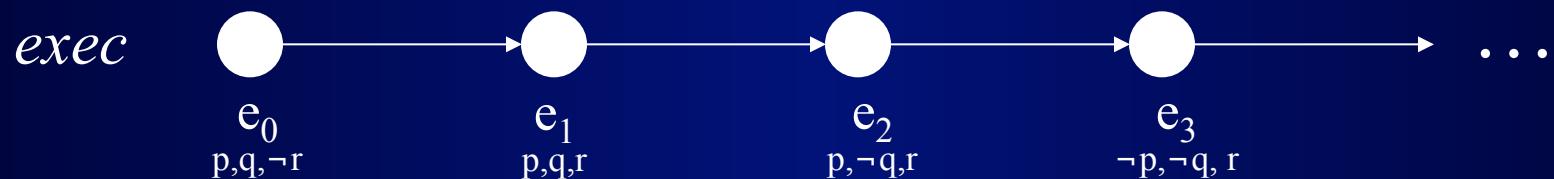
Le connecteur F (*Future*) est également noté \diamond

Le connecteur G (*Generally*) est également noté \square

- Les formules de LTL sont obtenues en considérant des prédictats atomiques à la place des propositions atomiques, et en ajoutant les quantificateurs \forall et \exists .

PLTL : interprétation des formules

- Les formules de PLTL sont interprétées sur des exécutions de notre modèle
- Une exécution $exec$ de M est une séquence éventuellement infinie (dans la suite on ne note pas les actions sur les transitions)
- On aura $M \models F$ ssi pour toute exécution $exec$ de M , $exec \models F$
- Référentiel de temps : e_0 correspond au présent, $e_1, e_2 \dots$ au futur



PLTL : connecteur F

- **F p** : « p est vérifiée dans le futur » (Future)



- exec1, exec2 $\models F p$ mais exec3 $\not\models F p$



- $\neg F p$: « p n'est jamais vérifiée dans le futur »

- $F(p \vee q)$ est équivalent à $Fp \vee Fq$



PLTL : connecteur G

- **G p** : « p est toujours vérifiée dans le futur » (Generally)



- **G p** est équivalent à $\neg F \neg p$
- **G(p \wedge q)** est équivalent à $Gp \wedge Gq$

PLTL : connecteurs F et G

- On pose : $F^\infty p \equiv G F p$

« p est vérifiée une infinité de fois dans le futur »



- On pose : $G^\infty p \equiv F G p$:

« p est toujours vérifiée à partir d'un certain état »



PLTL : connecteur X

- **X p** : « p est vérifiée à l' état suivant » (neXt)



- On a :

$$\mathbf{G} p \Rightarrow \mathbf{X} p$$

$$\mathbf{X} p \Rightarrow \mathbf{F} p$$

PLTL : connecteur U

- $p \mathbf{U} q$: « p est vérifiée jusqu' à ce que q le soit » (Until)



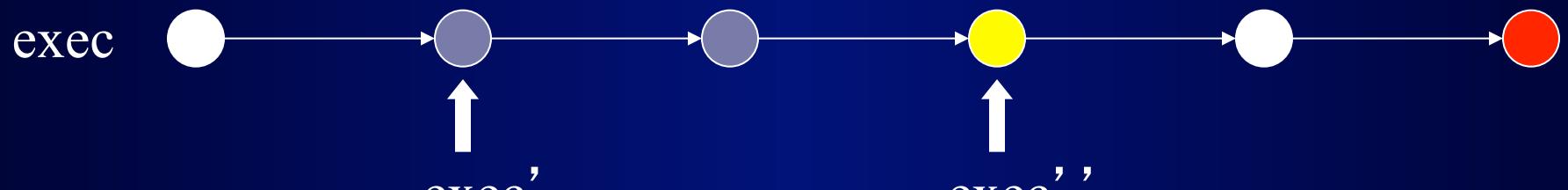
- exec1, exec2 $\models p \mathbf{U} q$ mais exec3 $\not\models p \mathbf{U} q$



 signifie $\models q$
 signifie $\models p$

PLTL : exemple

- L' exécution suivante satisfait-elle $\mathbf{X} (p \mathbf{U} (q \wedge F r))$?



- $exec \models \mathbf{X} (p \mathbf{U} (q \wedge F r))$?
- $exec' \models (p \mathbf{U} (q \wedge F r))$?
- $exec'' \models q \wedge F r$?

\Rightarrow OUI

	signifie $\models p$
	signifie $\models q$
	signifie $\models r$

PLTL : spécification

- « Après toute introduction d'une nouvelle carte, si les billets sont délivrés, cette carte sera éjectée »

$$\mathbf{G} [(\text{Intro_carte} \wedge \mathbf{X}((\neg \text{Intro_carte}) \mathbf{U} \text{Billets}))]$$

$$\Rightarrow (\mathbf{X}((\neg \text{Intro_carte}) \mathbf{U} \text{Eject_carte}))]$$

$$\forall t . [\text{Intro_carte}(t) \wedge \exists t' > t. (\text{Billets}(t') \wedge (\forall t'' . t < t'' \leq t' \Rightarrow \neg \text{Intro_carte}(t'')))]$$

$$\Rightarrow \exists t' > t. (\text{Eject_carte}(t') \wedge (\forall t'' . t < t'' \leq t' \Rightarrow \neg \text{Intro_carte}(t'')))$$

LTL : spécification

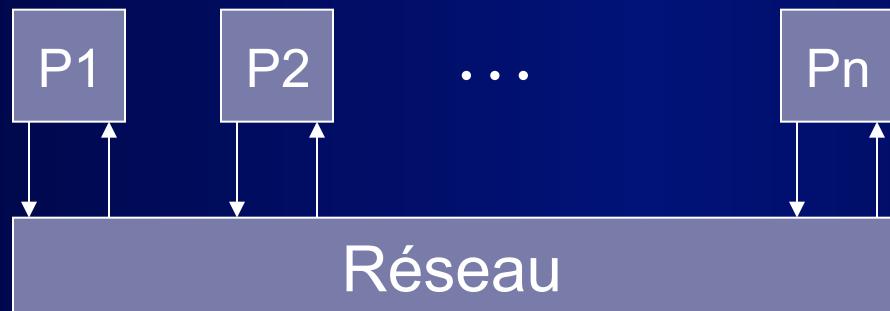
- Modèle de communication simplifié :
 - un émetteur (S) et un récepteur (R) communiquent
 - S (resp. R) possède un buffer en émission (resp. réception) S.out (resp. R.in)
 - S.out et R.in sont reliés par un canal unidirectionnel
 - S (resp. R) envoie (resp. reçoit) un message en l' insérant (resp. supprimant) dans S.out (resp. R.in)
 - S.out et R.in fonctionnent correctement (pas de modifs ni pertes, et un msg ne reste pas infiniment dans le buffer)
 - les messages sont identifiés de manière unique

LTL : spécification

- Un message ne peut être présent dans les deux buffers à un moment donné :
$$G \neg(m \in S.out \wedge m \in R.in) \quad (1)$$
- Le canal ne perd pas de messages : $G [m \in S.out \Rightarrow F(m \in R.in)]$
ou d'après (1) $G [m \in S.out \Rightarrow X F(m \in R.in)] \quad (2)$
- Le canal préserve l'ordre des messages :
$$\begin{aligned} G [m \in S.out \wedge \neg m' \in S.out \wedge F(m' \in S.out) \\ \Rightarrow F(m \in R.in \wedge \neg m' \in R.in \wedge F(m' \in R.in))] \end{aligned}$$
- Le canal ne génère pas « spontanément » de messages :
$$G [(\neg m \in R.in) U (m \in S.out)]$$

LTL : exercice

- Protocole d' élection ou de sélection dynamique de « leader »
 - Les capacités d' un processus à fournir un service sont abstraites par son identité



- On a un nombre fini de processus, N
- Le mode de communication entre les processus est asynchrone

LTL : exercice

- Hypothèses
 - Chaque processus a un identifiant unique, et un ordre total existe entre les différents processus
 - Un processus est initialement inactif (i.e. ne participe pas à l'élection) et devient arbitrairement actif
 - Un processus ne peut être inactif indéfiniment
 - Un processus actif ne peut redevenir inactif
 - Une élection du leader a lieu pour un ensemble donné de processus actifs; si un processus inactif devient actif une nouvelle élection a lieu si ce processus a une identité supérieure à celle du leader courant.
 - Ensemble des propositions atomiques : $AP = \{leader(i), active(i), i < j, i \neq j \mid 0 < i, j < N\}$

LTL : exercice

- Questions :
 1. modéliser dans PLTL la propriétés suivantes « Il y a toujours un unique leader » (1)
 2. modéliser « Il y a au maximum un leader » (2)
 3. Modéliser « Il y a aura toujours un leader élu dans le futur » (3)
 4. on préférera considérer (2)+(3) à la place de (1), pourquoi ?
 5. modéliser « en présence d' un processus actif ayant une identité supérieure, le processus leader sera déchargé de son rôle à un moment donné ».
 6. modéliser « un nouveau leader constitue une amélioration par rapport au précédent »

LTL : exercice

- Corrections :

1. « Il y a toujours un unique leader » : $G[\exists i.\text{leader}(i) \wedge (\forall j \neq i. \neg \text{leader}(j))]$
2. « Il y a au maximum un leader » : $G[\text{leader}(i) \Rightarrow (\forall j \neq i. \neg \text{leader}(j))]$
3. Modéliser « Il y a aura toujours un leader élu dans le futur » : $GF[\exists i.\text{leader}(i)]$
4. Tous les processus peuvent être inactifs initialement, de plus la communication étant asynchrone, le changement de leader peut difficilement se faire de manière atomique. On pourrait modifier (1) en $GF[\exists i.\text{leader}(i) \wedge (\forall j \neq i. \neg \text{leader}(j))]$ mais cela n' empêcherait pas qu'il y ait plusieurs leader à un moment donné
5. « en présence d'un processus actif ayant une identité supérieure, le processus leader sera déchargé de son rôle à un moment donné » :

$G[\forall i,j.((\text{leader}(i) \wedge \neg \text{leader}(j) \wedge \text{active}(j) \wedge i < j) \Rightarrow F \neg \text{leader}(i))]$

1. « un nouveau leader constitue une amélioration par rapport au précédent »

$G[\forall i,j.((\text{leader}(i) \wedge \neg X \text{leader}(i) \wedge X \text{leader}(j)) \Rightarrow i < j)]$

CTL : formules

- Définition : les formules de CTL sont définies par la grammaire suivante :

$\Phi, \Psi ::= p \mid q \mid \dots \mid \text{true} \mid \text{false}$ (propositions atomiques)

$\Phi \wedge \Psi \mid \Phi \vee \Psi \mid \Phi \Rightarrow \Psi \mid \neg \Phi \mid$ (connecteurs booléens)

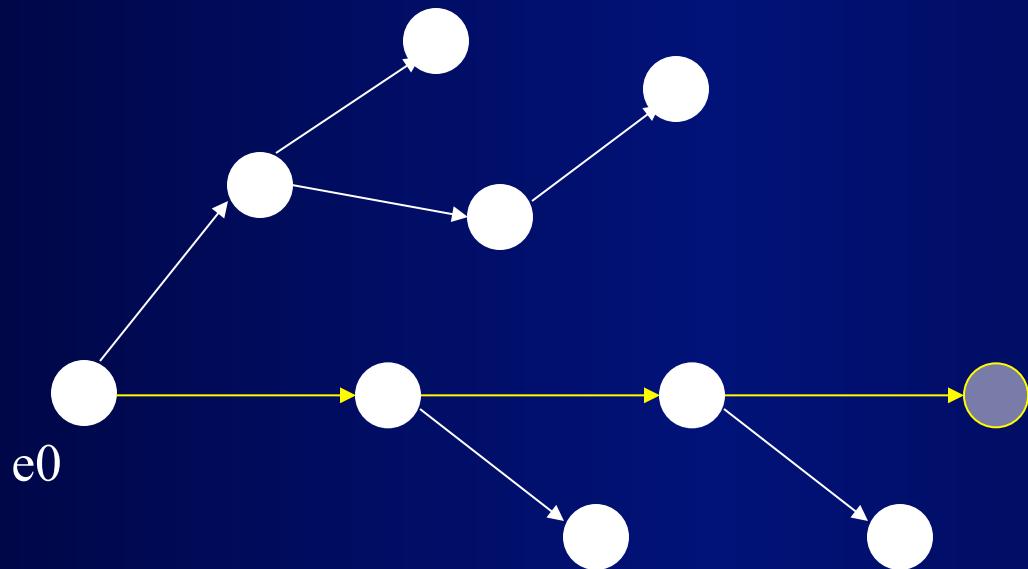
$\mathbf{EF}\Phi \mid \mathbf{EG}\Phi \mid \mathbf{E}\Phi \mathbf{U} \Psi \mid \mathbf{EX}\Phi \mid$

$\mathbf{AF}\Phi \mid \mathbf{AG}\Phi \mid \mathbf{A}\Phi \mathbf{U} \Psi \mid \mathbf{AX}\Phi$ (connecteurs temporels)

- E, A: quantifications sur toutes les exécutions possibles à partir de l' état courant
- Les formules sont interprétées sur des états de notre modèle, et non sur une exécution (« formule d' état » vs. « formule de chemin »)

CTL : connecteur EF

- **EF p** : « il est possible d' atteindre un état où p est vérifiée », ou « il existe une exécution (E) conduisant à un état où p est vérifiée (F) »

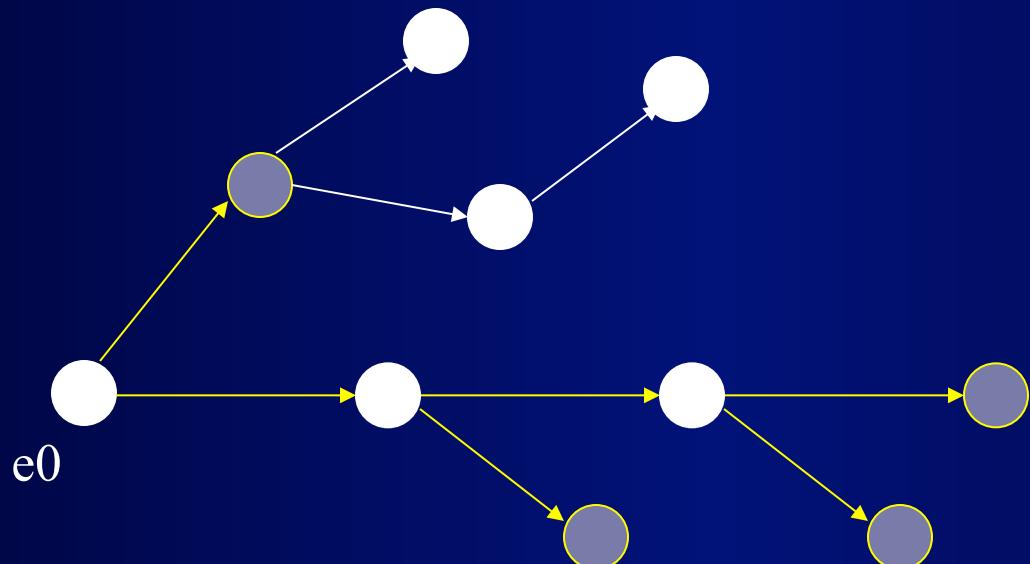


$e_0 \models \text{EF } p$

signifie $\models p$

CTL : connecteur AF

- AF p : « p est vérifiée dans le futur », ou « pour toute exécution (A), il existe un état où p est vérifiée (F) »

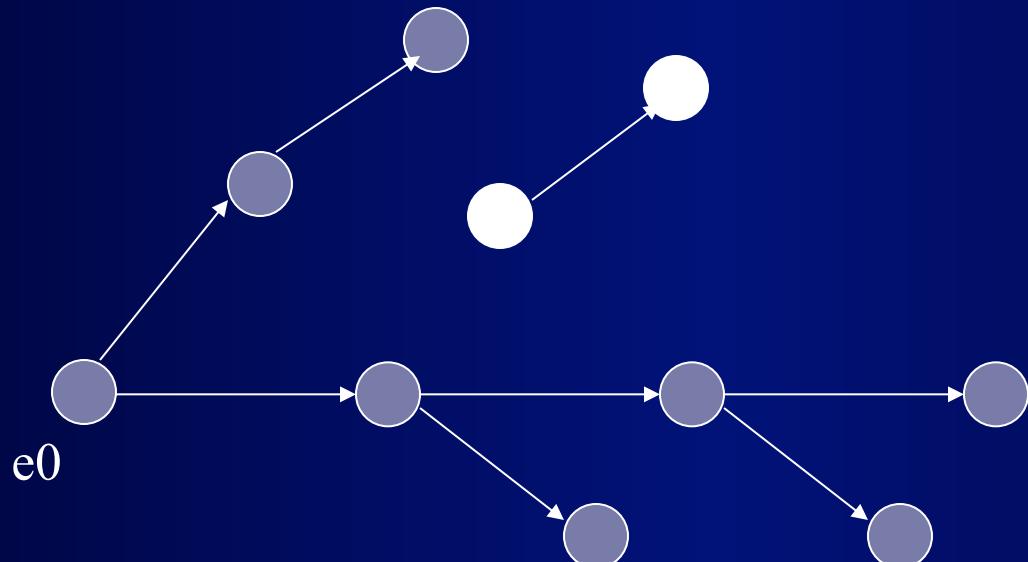


$$e0 \models AF p$$

 signifie $\equiv p$

CTL : connecteur AG

- **AG p** : «p est vérifié pour tout état atteignable »
«pour toute exécution (A) p est toujours vérifiée (G) »



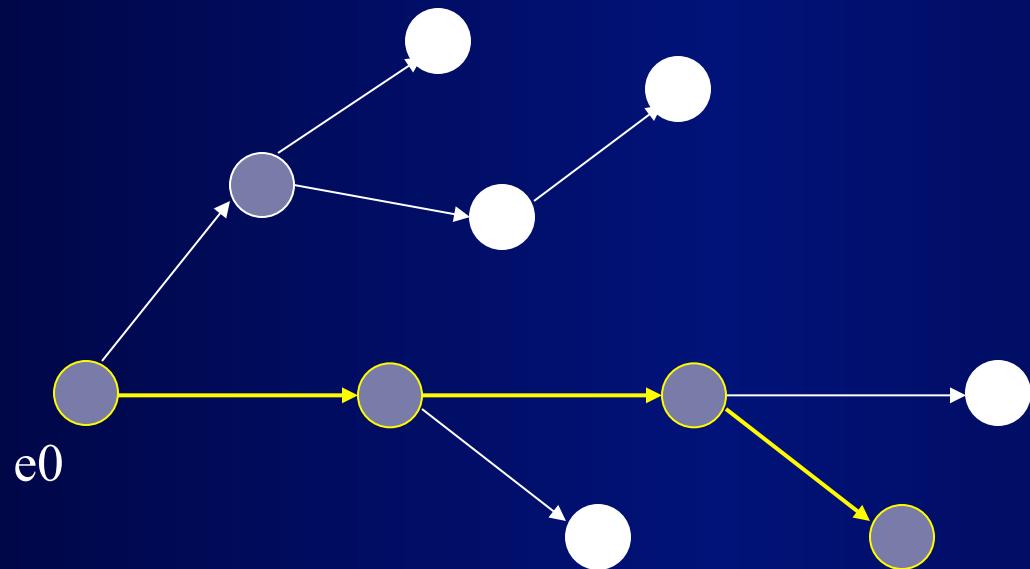
$e0 \models AG p$

signifie $\models p$

CTL : connecteur EG

- **EG p :**

« il existe une exécution (E) où p est toujours vérifiée (G) »



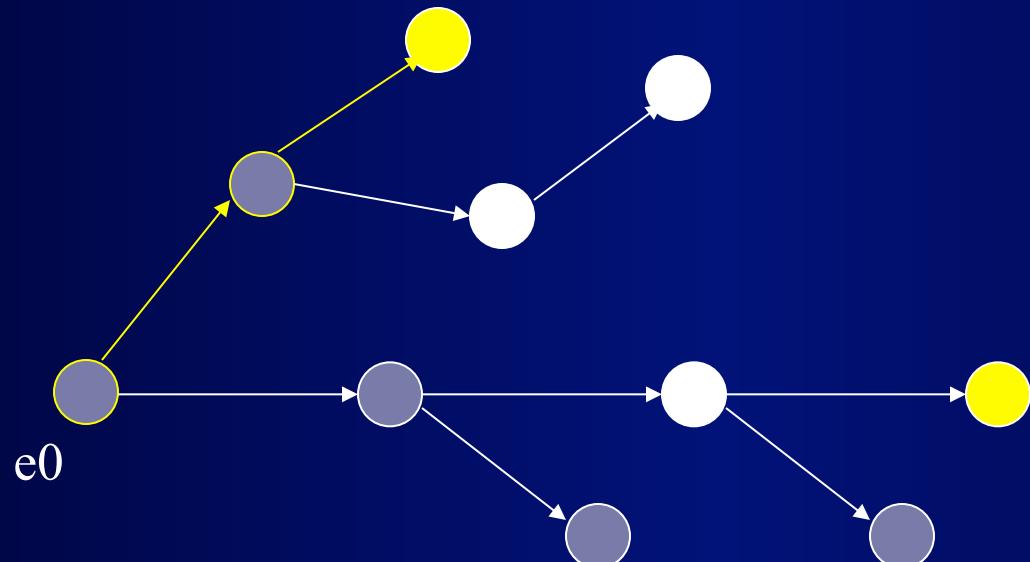
$e0 \models EG\ p$

signifie $\models p$

CTL : connecteur E_U_

- $E p U q$:

« il existe une exécution (E) durant laquelle p est vérifiée jusqu'à ce que q le soit ($p U q$) »



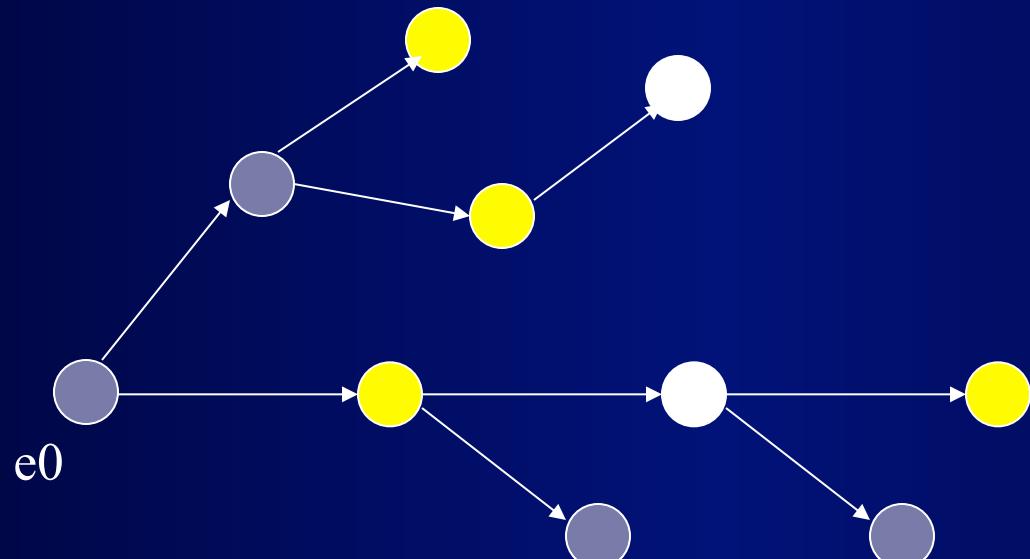
$$e_0 \models E p U q$$

	signifie $\models p$
	signifie $\models q$

CTL : connecteur A_U_

- A pUq :

« pour toute exécution (A) p est vérifiée jusqu' à ce que q le soit (pUq) »



$$e0 \models A p \ U q$$

●	signifie $\models p$
●	signifie $\models q$

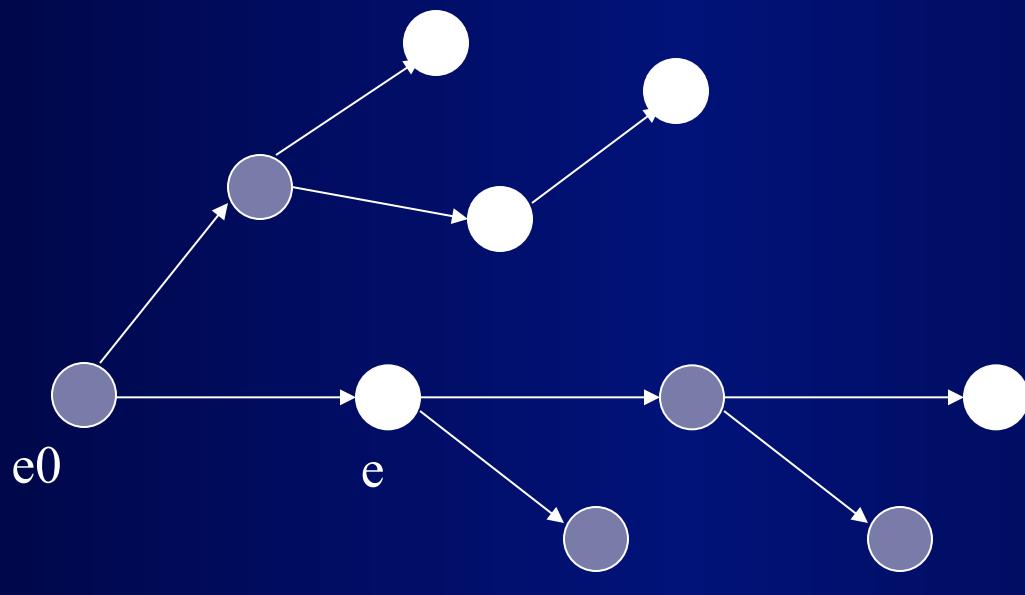
CTL : connecteurs EX et AX

- **EX p :**

« il existe une exécution (E) dont le prochain état satisfait p »

- **AX p :**

« tous les états immédiatement successeurs satisfont p »



$e0 \models \text{EX } p$

$e \models \text{AX } p$

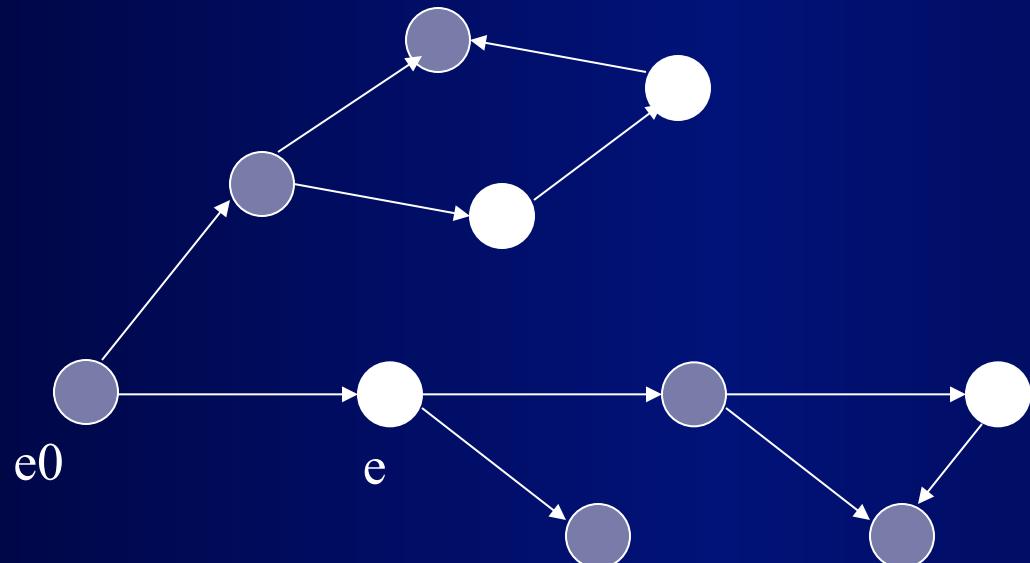
 signifie $\models p$



CTL : équivalences et composition

- $\mathbf{EF} p = \mathbf{E} \text{ true } \mathbf{U} p$
- $\mathbf{AF} p = \mathbf{A} \text{ true } \mathbf{U} p$
- $\mathbf{AX} p = \neg \mathbf{EX} \neg p$

- **AG EF p** : « A partir de n'importe quel état atteignable il est possible d'atteindre un état satisfaisant a »



signifie $\models p$

PLTL versus CTL

- Quelle logique doit-on utiliser ?
- Aucune n' est plus expressive que l' autre :
 - $G^\infty p \equiv F G p$ n' a pas d' équivalent dans CTL (p est toujours vérifiée à partir d' un certain état, qui n' est pas toujours le même)
 - $AG EF p$ n' a pas d' équivalent dans PLTL (à partir de n' importe quel état atteignable il est possible d' atteindre un état vérifiant p) ->rattrapage d' erreur
- PLTL permet d' énoncer les propriétés plus simplement
- CTL est préférable pour éviter les problèmes d' explosion combinatoire lors de la vérification exhaustive
- En fait PLTL et CTL peuvent être vues comme des fragments d' une même logique : CTL* (introduite postérieurement)

CTL* : formules

- Définition : les formules de CTL* sont définies par la grammaire suivante :

$\Phi, \Psi ::= p \mid q \mid \dots \mid$ (propositions atomiques)

$\Phi \wedge \Psi \mid \Phi \vee \Psi \mid \Phi \Rightarrow \Psi \mid \neg \Phi \mid$ (connecteurs booléens)

$F\Phi \mid G\Phi \mid \Phi U \Psi \mid X\Phi \mid$ (connecteurs temporels)

$A\Phi \mid E\Phi$ (quantificateurs de chemin)

- PLTL est obtenue en retirant les connecteurs A et E.
- CTL est obtenue quand on exige que tout connecteur temporel soit sous la portée d'un quantificateur de chemin

CTL* : modèles

- Les structures de Kripke sont les modèles de la logique temporelle.
- Contrairement aux automates :
 - on oublie les actions associées aux transitions,
 - on étiquette les états avec l' ensemble des propositions atomiques qu' ils vérifient
- Définition : une structure de Kripke est un quintuplet $\langle e_0, Q, PA, T, L \rangle$
 - Q est un ensemble fini d' états contenant e_0 , l' état initial. . .
 - PA est un ensemble fini de propositions atomiques
 - $T \subseteq Q \times Q$ est l' ensemble des transitions
 - $L : Q \rightarrow P(PA)$ est l' étiquetage des états par une partie de PA

CTL* : sémantique formelle (1)

- Une formule de CTL* se réfère à un moment donné d' une exécution
- On considère un temps discret : les instants sont des points le long des exécutions
- On écrira $e, t \models \Phi$ pour dire qu' au temps t , l' exécution e n' est pas finie et Φ est vraie
- $e(t)$ désigne l' état de e à l' instant t , et $L(e(t))$ l' ensemble des propriétés atomiques vérifiées par $e(t)$.
- On définit alors $e, t \models \Phi$ par induction sur Φ (transparent suivant)
- On dit qu' une structure de Kripke S vérifie Φ ($S \models \Phi$) ssi pour toute exécution e de S : $e, 0 \models \Phi$

CTL* : sémantique formelle (2)

- $e, t \models p$ ssi $p \in L(e(t))$
- $e, t \models \neg \Phi$ ssi il n'est pas vrai que $e, t \models \Phi$
- $e, t \models \Phi \wedge \Psi$ ssi $e, t \models \Phi$ et $e, t \models \Psi$
- ...
- $e, t \models X \Phi$ ssi $e, t+1 \models \Phi$
- $e, t \models F \Phi$ ssi il existe $t' \geq t$ tel que $e, t' \models \Phi$
- $e, t \models G \Phi$ ssi pour tout t' tel que $t \leq t'$ on a $e, t' \models \Phi$
- $e, t \models \Phi \vee \Psi$ ssi il existe $t' \geq t$ tel que $e, t' \models \Psi$ et pour tout t'' tel que $t \leq t' < t''$ on a $e, t'' \models \Phi$
- $e, t \models E \Phi$ ssi il existe e' t.q. $e(0) \dots e(t) = e'(0) \dots e'(t)$ et $e', t \models \Phi$
- $e, t \models A \Phi$ ssi pour tout e' t.q. $e(0) \dots e(t) = e'(0) \dots e'(t)$ on a $e', t \models \Phi$

Classes de propriétés

Propriété d'atteignabilité

- « Est-ce possible d'atteindre un état satisfaisant P ? »
 - « le compteur x peut prendre la valeur 0 »
 - « le point final du programme peut être atteint »
- Dans CTL :
 - EF P
 - Exemple, EF bloquer_carte
- Dans LTL :
 - une propriété est interprétée sur toutes les exécutions du système : on ne peut pas utiliser directement F
 - on vérifie à la place «Est-ce que $\neg P$ est toujours satisfaite?»
 - G \neg bloquer_carte

Propriété d' invariance

- « Tous les états atteignables satisfont P ? »
 - Pas de division par 0, pas de débordement de tableaux
 - Exclusion mutuelle : deux processus ne sont jamais simultanément en section critique
- Dans CTL :
 - AG P
- Dans LTL :
 - GP

Propriétés de sûreté

- «Sous certaines conditions, le problème P n' a jamais lieu»
 - Correction partielle : qd la pré-condition du pgm est respectée et que le pgm termine, alors la post-condition est respectée
- Dans CTL :
 - $\text{AG } \neg P$ (propriété duale de l'atteignabilité)
 - Exemple, $\text{AG } \neg(\text{bloquer_carte} \wedge \text{code_correct})$
- Dans LTL :
 - $\mathbf{G } \neg P$
 - Exemple, $\mathbf{G } \neg(\text{bloquer_carte} \wedge \text{code_correct})$

Propriétés de vivacité

- «**Sous certaines conditions, P finira par avoir lieu**»
 - Quand un message est envoyé il finira par être reçu
 - Correction totale : qd la pré-condition du pgm est respectée alors le pgm termine, et la postcondition est respectée
- Dans CTL :
 - Exemple, **AG** (code? \Rightarrow AF (bloquer_card \vee montant?))
- Dans LTL :
 - Exemple, **G** (code? \Rightarrow F (bloquer_card \vee montant?))

Absence de blocage

- «Il n'y a pas de blocage» =
«le système peut toujours exécuter une transition»
- Dans CTL :
 - **AG (EX true)** (différent d'une propriété de sûreté)
- Dans LTL :
 - **G (X true)**

Propriétés d'équité

- « Sous certaines conditions, P sera satisfaite un nombre infini de fois »
 - Équité faible : si un utilisateur demande continuellement un service, il finira par l'avoir
 - Équité forte : si un utilisateur demande infiniment souvent un service, il finira par l'avoir
- Dans CTL :
 - Pas exprimable
- Dans LTL :
 - Exemple équité forte, $(\text{GF } \text{code?}) \Rightarrow (\text{GF } \text{montant?})$

Model Checking

Model checking de CTL

- Algorithme dû à Queille, Sifakis, 82, Clarke, Emerson et Sistla, 86
- Principe :
 - repose sur un algorithme de *marquage* qui prend un automate M, une formule Φ de CTL, et consiste à associer à chaque état e de M, l' ensemble des sous-formules de Φ vérifiées (mémorisé sous la forme de paires $e.\Psi$).
 - ensuite on peut décider si $e, t \models \Phi$ en consultant les différents marquages
- Cas où $\Phi = \text{EX } \Psi$

```
Faire marquage( $\Psi$ ) ;  
Pour tout e dans Q faire  $e.\Phi := \text{false}$       /* initialisation */  
Pour tout  $(e, e')$  dans T,  
      si  $e'.\Psi = \text{true}$  alors faire  $e.\Phi := \text{true}$ 
```

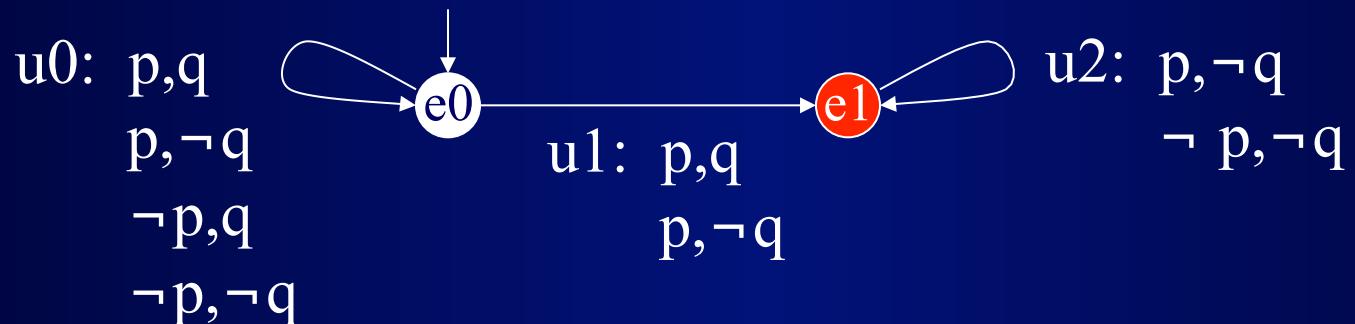
- Cas où $\Phi = \text{E } \Psi_1 \cup \Psi_2$: chaque transition est visitée une fois au plus
- Complexité de l' algorithme en $O((|Q| + |T|)^* |\Phi|)$

Model checking de PLTL

- Algorithme dû à [Lichtenstein, Pnueli, 85], [Vardi, Wolper 86]
- On ne peut plus marquer les états de l' automate : les formules portent sur des chemins (de longueur infinie, exple $F^\infty P$).
- Principe : étant donné un modèle M et une formule Φ ,
 - On construit un automate M' reconnaissant les exécutions qui ne satisfont pas Φ , appelé *observateur*
 - Ensuite on synchronise fortement M et M' , i.e. on fait en sorte que les deux automates avancent simultanément
 - Les seuls comportements de l' automate résultant M'' sont les exécutions de M qui ne vérifient pas Φ
 - Il suffit alors de vérifier si le langage reconnu par M'' est vide

Model checking de PLTL

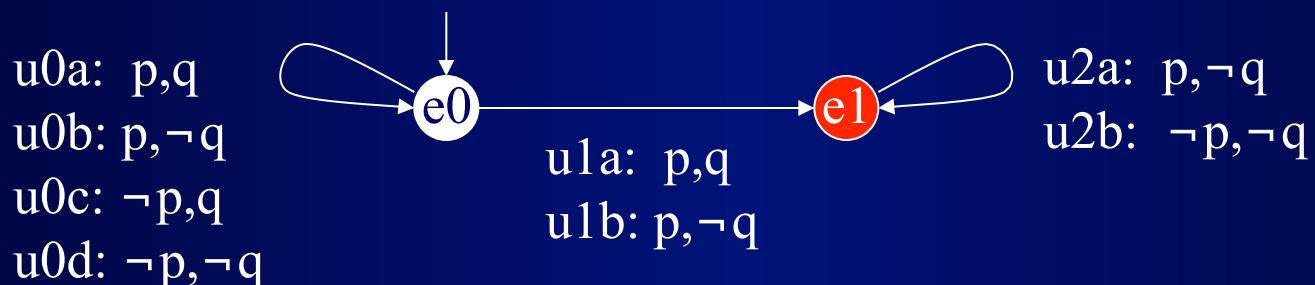
- Exemple : prenons $\Phi \equiv G(p \Rightarrow X F q)$
(toute occurrence de p doit être suivie plus tard d'une occurrence de q)
- $\neg\Phi$ signifie qu'il existe une occurrence de p après laquelle on ne rencontre plus jamais q
- L'automate ci-dessous reconnaît $\neg\Phi$ (e1 état final)



- L'automate doit reconnaître des mots infinis \Rightarrow automate de Büchi
- Exercice : quel est l'automate de Büchi reconnaissant $F^\infty p$?

Model checking de PLTL

- Automate de Büchi (BA) : mêmes composants qu'un automate fini (nombre fini d'états et de transitions), mais la condition d'acceptation est différente
- Définition : une exécution d'un BA est réussie si elle comporte au moins un état final apparaissant infiniment souvent. Le langage associé est celui des expressions ω -régulières (la répétition un nombre infini de fois du symbole a est noté a^ω)



- Expression ω -régulière associée :
 $((p,q)| (p,\neg q)| (\neg p,q)| (\neg p, \neg q))^*$ $((p,q)|(p,\neg q))$ $((p,\neg q)|(\neg p,\neg q))^\omega$

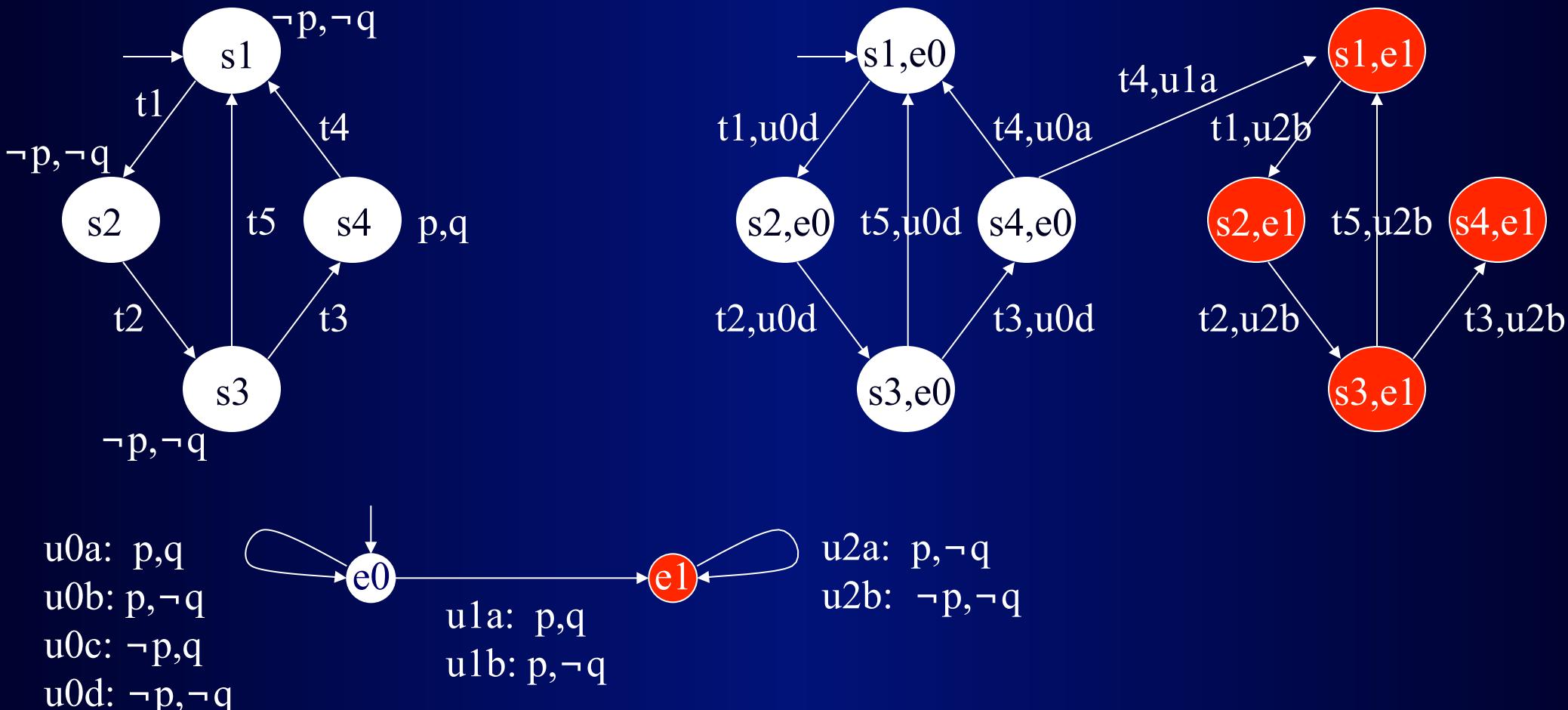
Model checking de PLTL

- Théorème [Wolper, Vardi, Sistla, 83] : pour toute formule ϕ de PLTL construite sur l' alphabet de propositions atomiques PA, il est possible de construire un automate de Büchi A sur l' alphabet 2^{PA} tel que $L(A)$ corresponde aux séquences d' ensembles de propositions atomiques satisfaisant ϕ .
- La définition du produit synchronisé doit être adaptée au cas du produit entre une structure de Kripke et un BA :

le produit d' une structure de Kripke $\langle e0', Q', PA, T', L \rangle$ et d' un BA $\langle e0'', Q'', 2^{PA}, T'', F \rangle$ est un BA $\langle (e0', e0''), Q' \times Q'', 2^{PA}, T, Q' \times F \rangle$ avec $T = \{((q1', q1''), A, (q2', q2'')), \text{ tels que } (q1', q2') \in T' \text{ et } (q1'', A, q2'') \in T'', \text{ et } A \subseteq L(q1')\}$
- De manière générale le produit synchronisé de deux automates de Büchi doit être redéfini

Model checking de PLTL

Produit synchronisé des automates



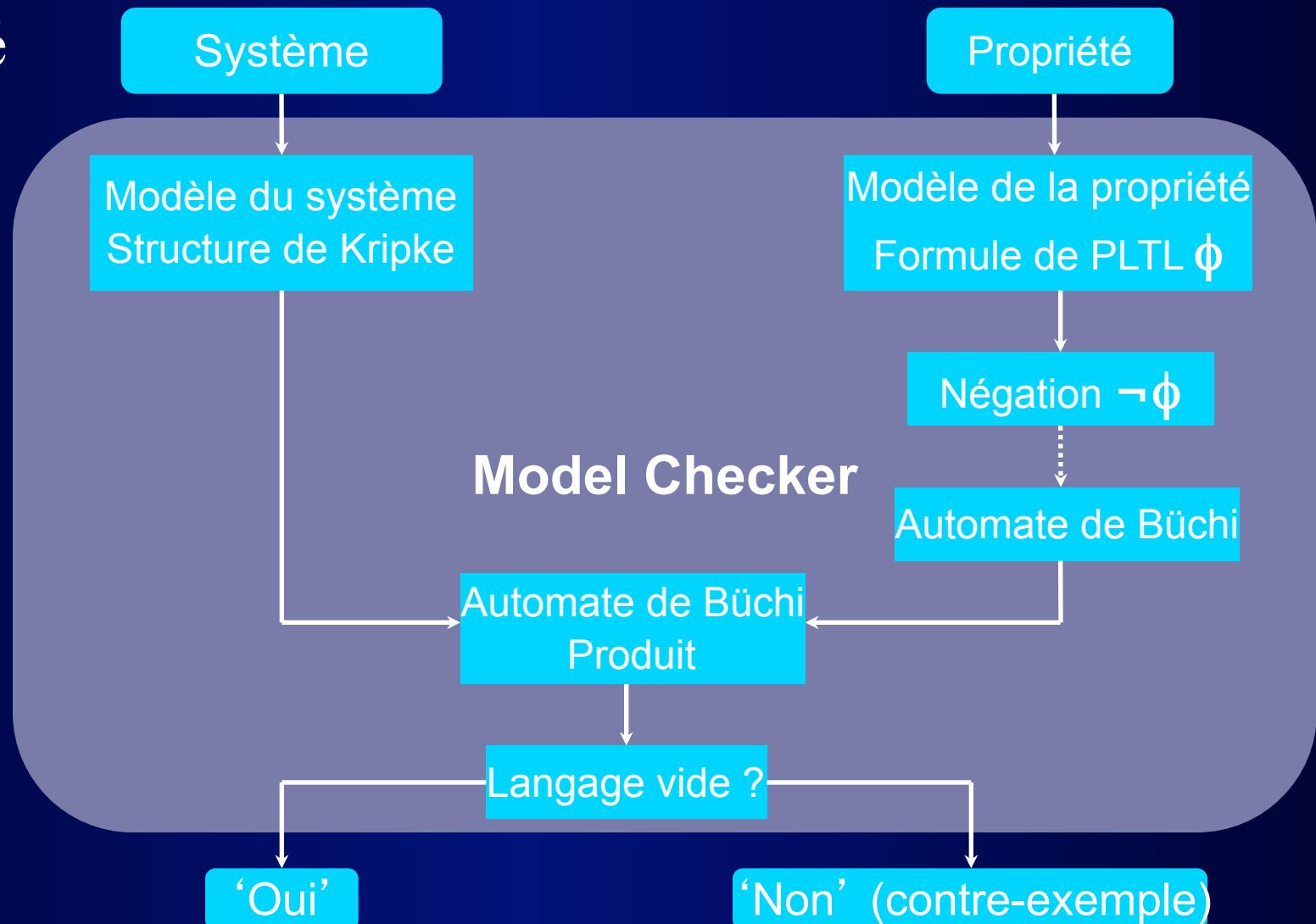
- La propriété n'est pas vérifiée !

Model checking de PLTL

- Vérifier que la propriété est satisfaite revient à vérifier que le langage reconnu par l' automate produit est vide
- Le langage reconnu par un BA est non vide ssi il existe un état final atteignable à partir de l' état initial et à partir de lui-même
- En terme de théorie des graphes : le graphe associé contient un cycle (non-trivial) atteignable à partir de l' état initial et incluant un état final
- L' algorithme calcule les états finaux atteignables à partir de l' état initial, puis vérifie pour chacun de ces états s' il appartient à un cycle

Model checking de PLTL

- En résumé



Coût et explosion combinatoire

- Pour CTL, le temps de vérification dépend linéairement de la taille de l' automate et de la taille de la formule.
- Pour PLTL, il dépend toujours linéairement de la taille de l' automate, mais exponentiellement de la taille de la formule (c-à-d $O((|Q|+|T|)*2^{|Φ|})$).
- Explosion combinatoire : lorsque l' on effectue le produit d' un réseau de k automates avec n états chacun on obtient une structure résultante S de n^k états !

Solutions à l'explosion combinatoire

- Solution : ne pas construire S explicitement
- Model checking symbolique :
 - on représente des ensembles d'états en utilisant des notations symboliques (BDD « boolean decision diagram », polynômes, ...)
 - il faut des algorithmes capables de manipuler ces notations efficacement
- Model checking à la volée :
 - on construit uniquement la partie de S qui est en train d'être explorée
 - évite la limitation de mémoire, pas de temps

Quelques outils

- Outils libres
 - SMV
 - Carnegie Mellon, <http://www.cs.cmu.edu/~modelcheck/smv.html>
 - réseaux d' automates avec variables partagées
 - vérification dans CTL, model checking symbolique avec des BDDs
 - + : vérification de systèmes de tailles très élevées
 - - : langage de description, pas de fonctionnalité de simulation
 - Promela/Spin
 - Bell Labs, <http://spinroot.com/spin/whatispin.html>
 - automates communiquant via des canaux bornés
 - vérification dans PLTL
 - + : techniques de réductions de l' espace des états (vérifications à la volée, techniques de hachage,...)
 - - : ne permet pas d' étudier des systèmes à une infinité d' états (systèmes temporisés, réseaux de Petri)
 - Design/CPN
 - CPN group, Université d' Arhus, <http://www.daimi.au.dk/designCPN/>
 - Réseaux de Petri colorés
 - vérification de propriétés d' atteignabilité, absence de blocage,, ..
 - + : permet d' étudier des systèmes à une infinité d' états, outil à l' échelle industrielle
 - - : possibilités de vérification limitées

Quelques outils

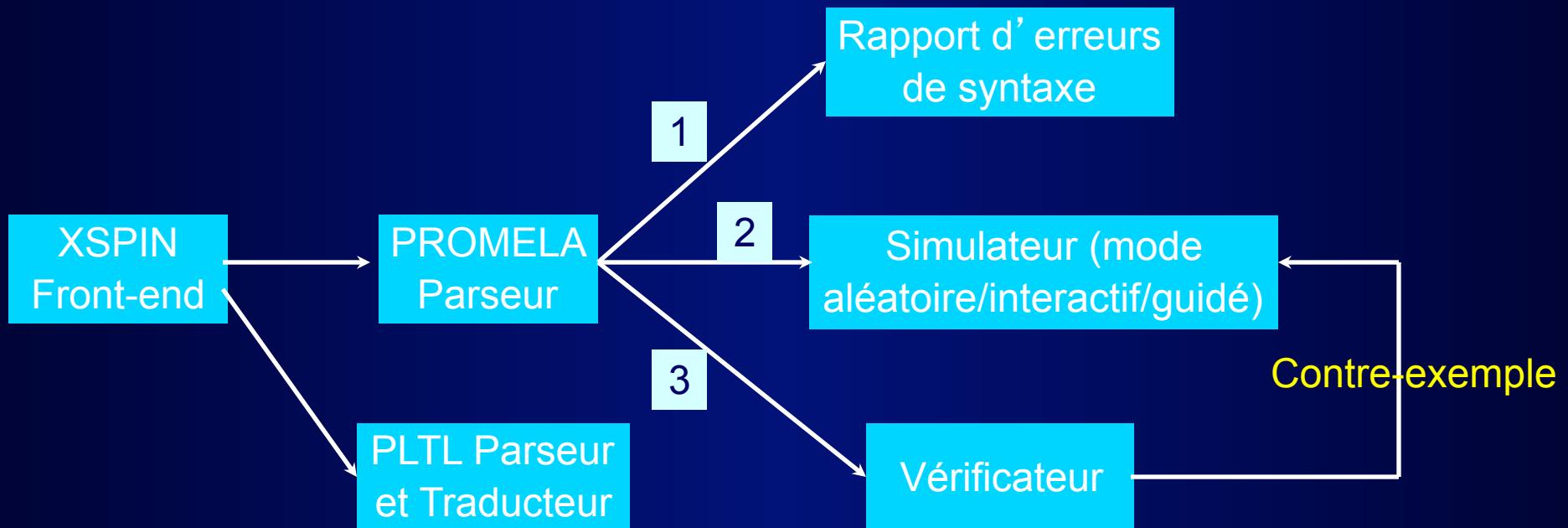
- Outils libres (suite)
 - UPPAAL
 - Université d' Uppsala et d' Aalborg, <http://www.uppaal.com/>
 - réseaux d' automates temporisés communiquant par synchronisation binaire
 - vérification de propriétés d' atteignabilité
 - + : vérification de systèmes temporisés
 - - : possibilités de vérification limitées
- Outils industriels
 - Rulebase
 - IBM, http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/
 - automates communiquant via des canaux bornés
 - vérification dans PSL (CTL + fairness), model checking symbolique
 - + : techniques vérifications efficaces (BDD, SAT)

Un exemple avec Promela/Spin

Promela/Spin

- Outil développé par G.J Holzmann aux Bell Labs.
<http://spinroot.com/spin/whatispin.html>
- Promela : langage de modélisation :
 - Langage impératif + primitives de communication
 - Permet de décrire chacun des processus du système et les interactions
 - Communication via des canaux fifo, par rendez-vous ou via des variables partagées
- Spin :
 - Mode simulation interactive, aléatoire, guidée
 - Mode vérification exhaustive de propriétés PLTL (sans l' opérateur X), code non atteignable
 - Vérification à la volée et techniques de hachage (dizaine de millions d' états)

Promela/Spin



Promela : principes

- Un programme Promela est constitué de déclarations de :
 - processus séquentiels définissant le comportement
 - constantes et de variables (locales ou globales) décrivant l' environnement des processus
 - canaux de communications reliant les processus
- Déclaration de constantes, variables globales, et canaux

```
#define Maxseq 10 /* size of buffer */

byte i;

chan c[N]=[Maxseq] of {byte}; /* N canaux chacun de capacité Max
Maxseq ≥ 0 */
```

- Déclaration d'un processus :
- Instanciation d'un processus et exécution :

```
init{run P(paramètres_instanciés);run Q (paramètres_instanciés);... }
```

Promela : principes

- Promela s' inspire du langage de commandes gardées de Dijkstra
- Une instruction Promela peut être soit exécutable soit bloquée
 - dans le second cas l' exécution du processus est stoppée jusqu' à ce que cette instruction devienne exécutable
 - par exemple l' instruction **(a==b)** est équivalente à **while (a!=b) do skip**
- Principales instructions
 - **skip** : instruction vide, ne fait rien
 - Affectation : **x=7**
 - Instruction conditionnelle

```
if :: guard1 -> instructions_1
    :: ...
    :: guardn -> instructions_n
fi
```

- parmi les alternatives dont la garde est exécutable, on en sélectionne une de manière non-déterministe.
- si toutes les gardes sont bloquées l' instruction **if** est bloquée jusqu' à ce qu' une garde devienne exécutable (l' instruction spéciale **else** est toujours exécutable)

Promela : principes

– Instruction de boucle

```
do :: guard1 -> instructions_1
      :: ...
      :: guardn -> instructions_n
od
```

- boucle infinie, interruptible par l' exécution d'une instruction **break** ou **goto**

– Instructions d' envoi et de réception de message

- **!** dénote l' envoi d' un message et **?** la réception
- Si le buffer de **c** a une capacité strictement positive, c se comporte comme une FIFO :
 - l' instruction **c!2** est exécutable si le buffer de **c** n' est pas rempli, et son exécution inclut **2** à la fin du buffer;
 - l' instruction **c?x** est exécutable si le buffer n' est pas vide, et son exécution supprime le premier élément du buffer et affecte sa valeur à la variable **x**
- Si la taille du buffer de **c** est nulle : communication synchrone
 - **c!** (resp. **c?**) est exécutable s' il existe une instruction **c?** (resp. **c!**) correspondante pouvant être exécutée simultanément

Promela : exemple (1)

- Exemple d' un ascenseur à 3 étages
- Déclaration de constantes et de variables globales

```
bit porteouverte[3]; /* tableau de longueur 3 */
/* ,1=porte ouverte, 0=fermée */,
chan ouvporte=[0] of {byte,bit};
/* canal ouvporte permet la comm. entre l'asc et les portes*/
/* buffer de longueur 0, communication par RDV */
/* byte du msg=num de l'étage, bit=1 ouvrir, 0 fermer */
```

- On déclare ensuite le comportement d' une porte (qui sera instancié dans un processus pour chaque étage) et celui de l' ascenseur



Promela : exemple (1)

- Processus porte :

- Prends pour paramètre le numéro de l' étage i
- Attend un ordre d' ouverture instruction (**instr 1**)
- Indique que la porte est ouverte, puis qu' elle est fermée (**instr 2 & 3**)
- Signale à l' ascenseur la fermeture (**instr 4**)

```
proctype porte(byte i) { /* déclaration d'un type de processus */
do
  :: ouvporte?eval(i),1;          /* instr 1 */
  porteouverte[i-1]=1;           /* instr 2 */
  porteouverte[i-1]=0;           /* instr 3 */
  ouvporte!i,0                   /* instr 4 */
od}
```

- « eval » force l'égalité du numéro d' étage reçu avec i
- Opérations sur les canaux :
 - réception de <msg> sur <chan> : <chan>?<msg>
 - envoi de <msg> sur <chan> : resp. <chan>!<msg>
- Remarque : les tableaux et les types de processus ne peuvent pas être passés en paramètre

Promela : exemple (1)

- Processus ascenseur :
 - Vérifie que l' on est pas au dernier étage avant de monter (**instr 1**)
 - Vérifie que l' on n' est pas au premier étage avant de descendre (**instr 2**)
 - Demande éventuellement à la porte de s' ouvrir (**instr 3**)
 - Attend que la porte se referme avant de repartir (**instr 4**)

```
proctype ascenseur() {
    byte etage=1;      /* etage : variable locale */
    do
        :: (etage != 3) -> etage++; /* instr 1 */
        :: (etage != 1) -> etage--; /* instr 2 */
        :: ouvporte!etage,1;        /* instr 3 */
            ouvporte?eval(etage),0 /* instr 4 */
    od}
```

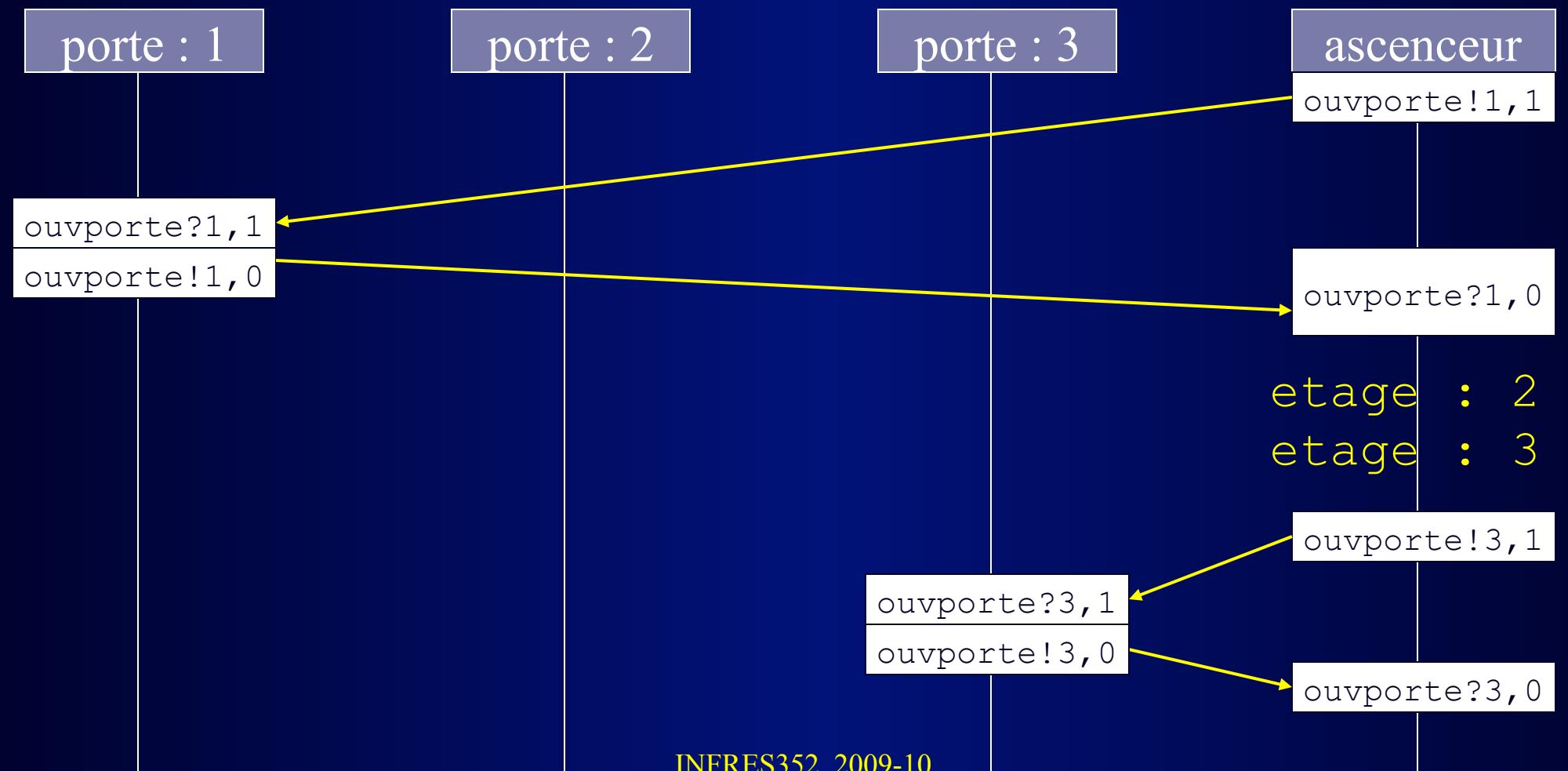
Promela : exemple (1)

- La boucle do ... od de Promela est une boucle infinie (interruptible par l' exécution d'une instruction ``break") dans laquelle un choix non déterministe est effectué entre toutes les séquences d' instructions commençant par ::
- On regroupe ces éléments dans un processus d' initialisation
 - les processus correspondant aux 3 portes et à l' ascenseur sont instanciés
 - les processus s' exécutent ensuite en parallèle

```
init{  
  
    atomic{ /* pas d'entrelacement entre les  
    instructions de chaque processus */  
        run ascenseur();run porte(1); run porte(2);  
        run porte (3)  
    }  
}
```

Promela : simulation

- Mode : interactif/aléatoire/guidé



Spin : vérification

- « Si la porte du premier étage est ouverte, elle est obligatoirement fermée dans l' état suivant »

```
#define ouv1 porteouverte[0]
#define ferm1 !porteouverte[0]
[] (ouv1 -> X ferm1) /* G(ouv1=>X ferm1) */
```

- « Inévitablement une des portes sera ouverte »

```
#define ouv1 porteouverte[0]
#define ouv2 porteouverte[1]
#define ouv3 porteouverte[2]
<>(ouv1||ouv2||ouv3) /*F(ouv1|ouv2|ouv3) */
```

Faux : l' ascenseur peut monter et descendre sans s' arrêter



TP

- Pour préparer le TP :
 - Basic Spin manual
<http://spinroot.com/spin/Man/Manual.html>
 - et regarder le protocole suivant

Exemple (2) : protocole d' élection

- Election de leader : protocole de Dolev, Klawe & Rodeh (82)
 - N processus en anneau ($N \geq 1$) connectés par des canaux non bornés
 - Un processus ne peut envoyer un message que dans le sens des aiguilles d'une montre
 - Chaque processus à un identifiant unique (entier naturel)
- Algorithme :

```
active:
d:=ident;
do forever
begin
    send(d);
    receive(e);
    if e=d then stop; (* d est
                           le leader *)
    send(e);
    receive(f);
    if e≥max(d,f) then d:= e
    else goto relay;
end
```

```
relay:
do forever
begin
    receive(d);
    send(d);
end
```

Exemple (2) : protocole d' élection

```

active:
d:=ident;
do forever
begin
  send(d);
  receive(e);
  if e=d then stop; (* d est
                           le leader *)
  send(e);
  receive(f);
  if e>=max(d,f) then d:= e
  else goto relay;
end

```

```

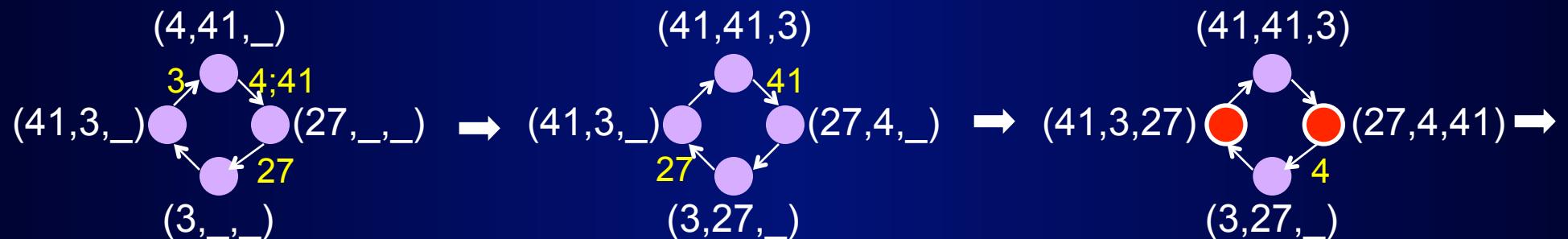
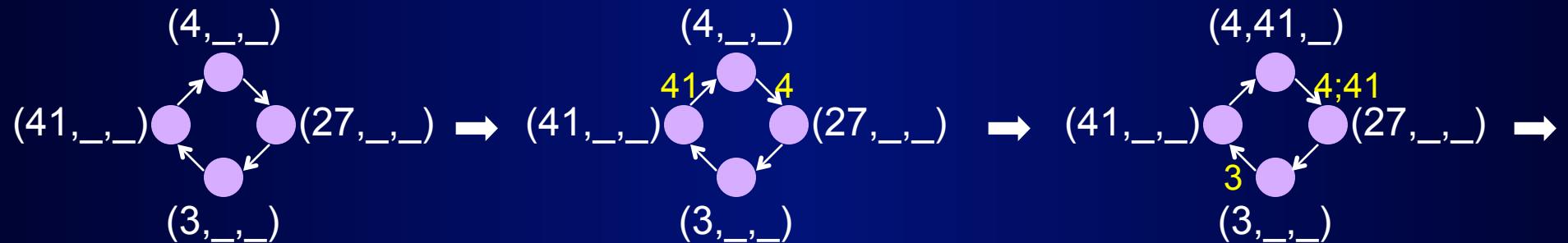
relay:
do forever
begin
  receive(d);
  send(d);
end

```

- Initialement chaque processus est actif
- Tant qu' il est actif un processus est responsable d' un numéro de processus stocké dans d : cette valeur évoluant au cours du déroulement du protocole
- Quand un processus détermine qu' il ne détient pas l' identité d' un leader, il transfère les numéros reçus (relay) de manière transparente
- Chaque processus actif envoie sa variable d à son voisin le plus proche, et attend de recevoir la valeur e du processus le précédent (actif) le plus proche
- Si le processus reçoit son propre d, il conclut qu' il est le seul processus actif, et que d est l' identité du nouveau leader et s' arrête
- S' il reçoit une valeur différente, il attend de recevoir la valeur f du second processus le précédent (actif) le plus proche
- Si e est la plus grande valeur, alors le processus met à jour d, sinon il rentre dans la phase de transfert : à chaque round du protocole, au moins un processus actif rentre dans la phase de transfert.

Exemple (2) : protocole d' élection

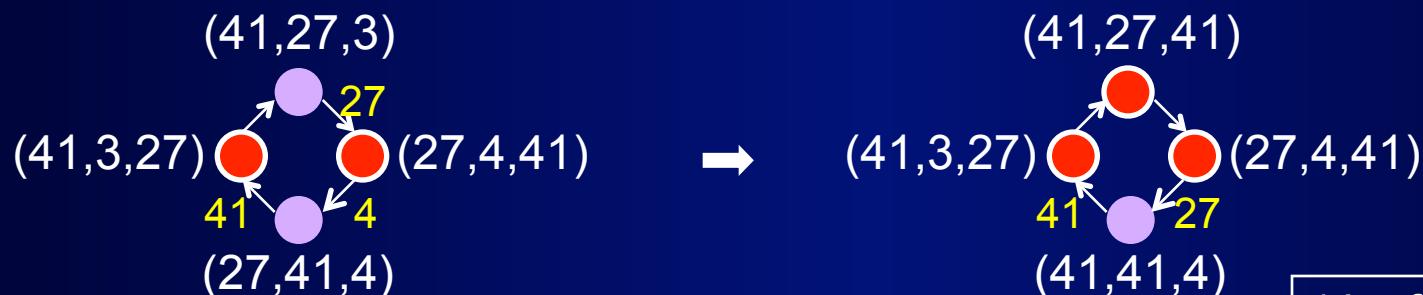
- Exemple d' exécution : 4 processus d' identité 3, 4, 27 et 41



	Processus actif
	Processus en phase de transfert

Exemple (2) : protocole d' élection

- Exemple d' exécution : suite



(d,e,f)	Processus actif
\bullet	Processus en phase de transfert

Conclusion

Conclusion

- Avantages de la vérification sur modèle
 - Approche bien fondée mathématiquement
 - Approche générique : applications possibles à la vérification Hardware, aux protocoles de communications, aux systèmes embarqués, au génie logiciel...
 - Approche calibrable : permet une vérification *partielle* d' un système
 - Ne nécessite pas d' interaction avec l' utilisateur
 - N' ajoute pas de réel coût supplémentaire en temps par rapport à la simulation ou au test
 - Permettrait de diminuer le temps de développement
 - Peut être utilisé pour la génération de cas de test

Conclusion

- Limitations
 - Non adapté aux applications orientées données
 - La vérification se limite au modèle du système : ne garantit pas que le système réel possède les propriétés -> peut être complétée par la génération automatique de cas de tests
 - Seules les propriétés formulées sont vérifiées
 - Trouver le bon niveau d' abstraction pour exprimer le modèle et les propriétés en logique temporelle requiert de l' expertise
 - Comme tout logiciel un model checker peut mal fonctionner
 - On ne peut généraliser la vérification à un nombre arbitraire de processus (-> utilisation d' un assistant de preuve)
- Permet d' améliorer le niveau de confiance accordé à un système

Conclusion

- Applications industrielles
 - Projet NewCoRe (AT&T) :
 - Bell labs 1990-92
 - 5ESS Switching Centre, partie du protocole de signalisation n°7 (ISSN user part)
 - 40-50 programmeurs pour plusieurs milliers de lignes de code sur 2 ans
 - 4-5 « ingénieurs en vérification » en parallèle et indépendamment
 - Spécification : 7500 lignes de SDL
 - Model Checker : SPIN
 - 112 erreurs importantes de design
 - 145 exigences formelles formulées en logique temporelle
 - 10000 vérifications réalisées (100 par semaine)
 - 55% des exigences initiales (informelles) ont été prouvées inconsistantes
 - Au bout de 6 mois du fait d' inconsistances graves les 2 process n' étaient plus indépendants : combien de ces erreurs auraient été trouvées par l' équipe de développement? Et quand ?