Cours de Model Checking

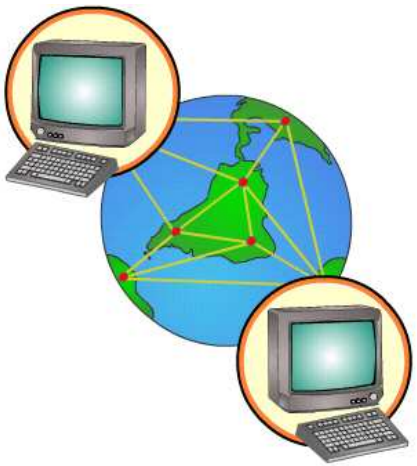
Leçon 1 : Introduction & Modélisation

Sébastien Bardin

CEA-LIST, Laboratoire de Sûreté Logicielle

sebastien.bardin@cea.fr
http://sebastien.bardin.free.fr/

Systèmes réactifs



Systèmes réactifs II

Programme classique

- termine
- retourne un résultat
- données complexes, contrôle séquentiel ($\approx$ simple)

exemple : compilateur, algo de tri

Systèmes réactifs II

Programme classique

- termine
- retourne un résultat
- données complexes, contrôle séquentiel ($\approx$ simple)

exemple : compilateur, algo de tri

Système réactif

- ne doit pas terminer
- ne retourne pas de résultat
- données simples, contrôle distribué ($\approx$ complexe)

exemple : protocole, OS

Vérifier un programme classique : Aspect temporel toujours identique, mais les prédicats sur les données peuvent être complexes.

- *"Le programme termine et le tableau est trié"*.

Systèmes réactifs III

Vérifier un programme classique : Aspect temporel toujours identique, mais les prédicats sur les données peuvent être complexes.

- *"Le programme termine et le tableau est trié".*

Vérifier un système réactif : Aspect temporel très varié mais les prédicats sur les données sont souvent simples.

- *"Si un processus demande infiniment souvent à être exécuté, alors l'OS finira par l'exécuter".*
- *"Il est toujours possible de revenir à l'état initial".*
- *"Chaque fois qu'une panne est détectée, une alarme est émise".*
- *"Chaque fois qu'une alarme est émise, une panne a été détectée".*

Le model checking

Model Checking

Technique de vérification automatique de systèmes réactifs

Ingrédients

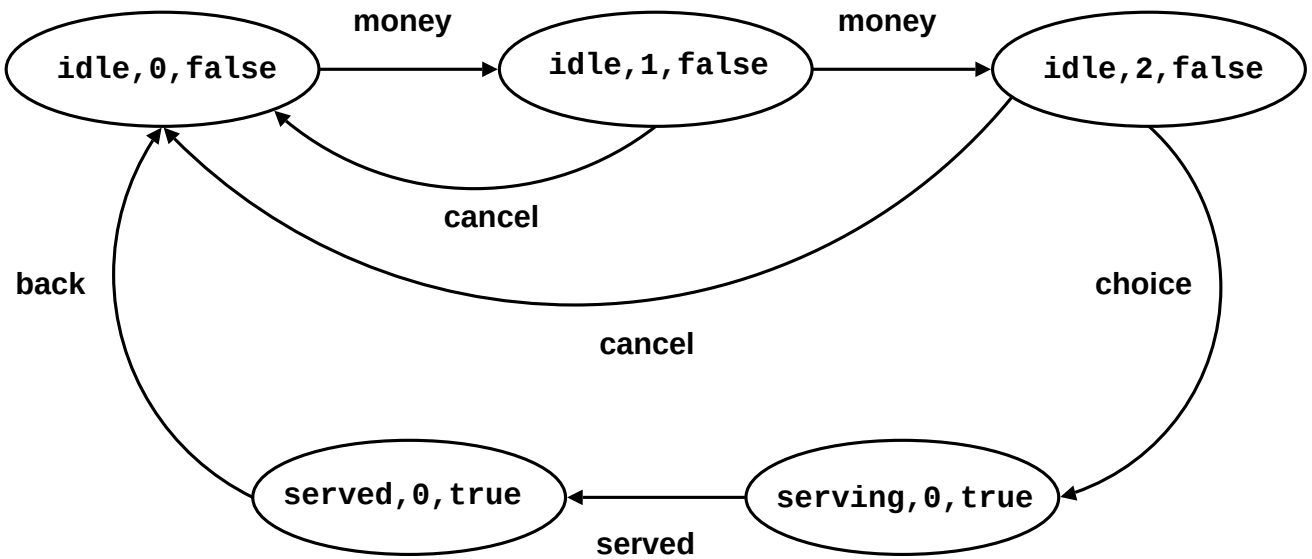
- $\mathcal{M}$ = système de transitions
- φ = formule temporelle
- MC = est-ce que $\mathcal{M} \models \varphi$?

Avantages

- Phases amonts : spécifs et design
- Automatisé
- Trouve mieux les bugs que le test
- Cost efficient (pour certains domaines)

Système de transitions

Système de transitions = comportement du système réactif



(voir plus loin)

Propriétés sur les systèmes réactifs

- Accessibilité *Une certaine situation peut être atteinte*
  `x` peut valoir 0, toute instruction peut être exécutée
- Invariance *Chaque état local respecte une bonne propriété*
  `x` ne vaut jamais 0, le tableau ne déborde jamais
- Sûreté *Quelquechose de mauvais n'arrive jamais*
  j'accède au fichier uniquement si j'ai entré le bon PIN
- Vivacité *Quelquechose de bon finit par arriver*
  le programme termine, le message finit toujours par être transmis
  le programme revient toujours à l'état initial
- Équité *Quelquechose de bon se répète infiniment souvent*
  si un processus demande toujours la main, il l'aura infiniment
- Équivalence comportementale *Est-ce que 2 systèmes sont équivalents ?* système simple de référence VS système optimisé

Logiques temporelles

On exprimera les propriétés grâce à des logiques temporelles

Avantages

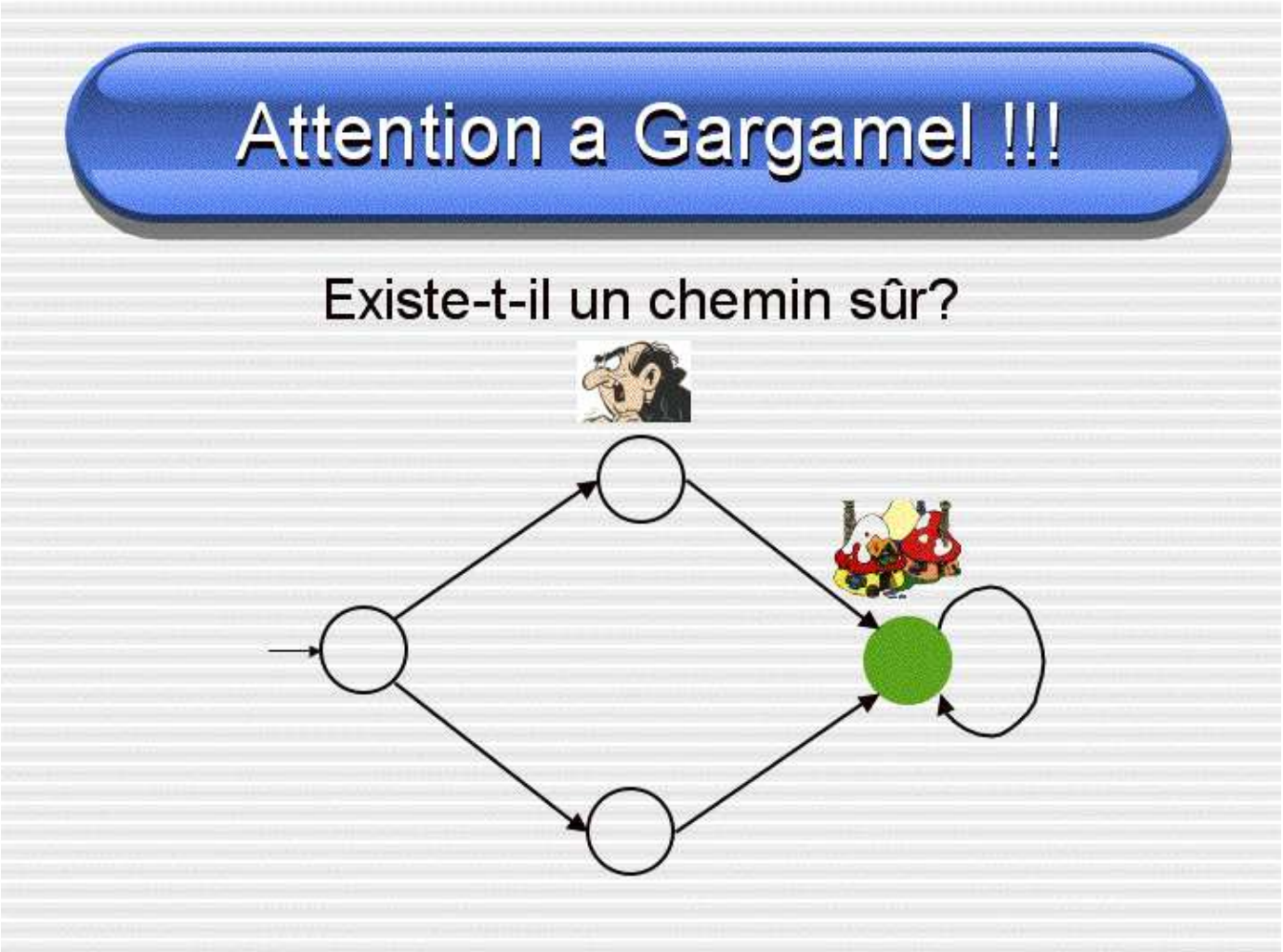
- non ambigue
- générique
- ouvre la voie à la vérification automatique

Plusieurs logiques temporelles possibles

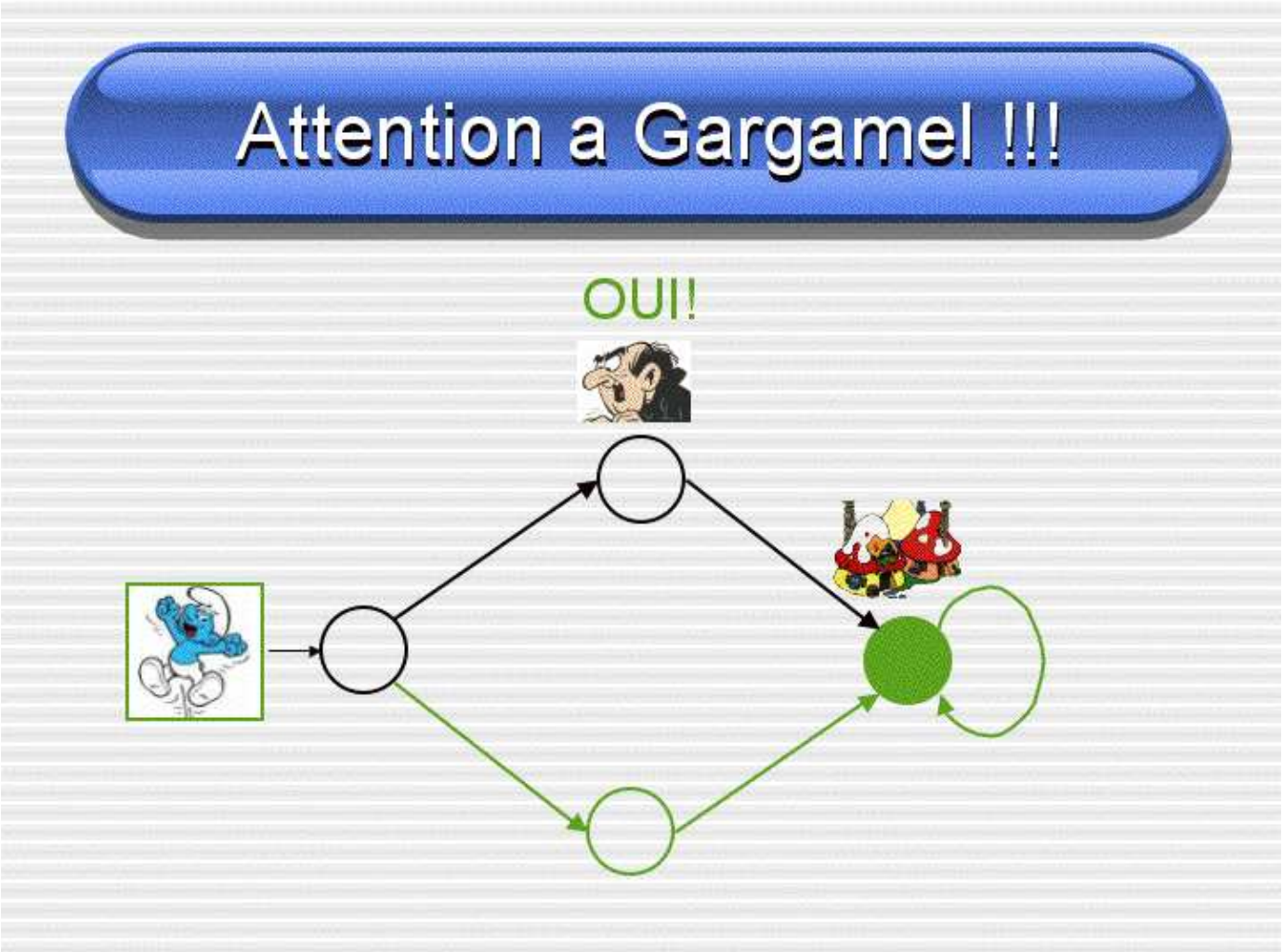
- LTL, CTL, CTL*, ...

(voir leçon 2)

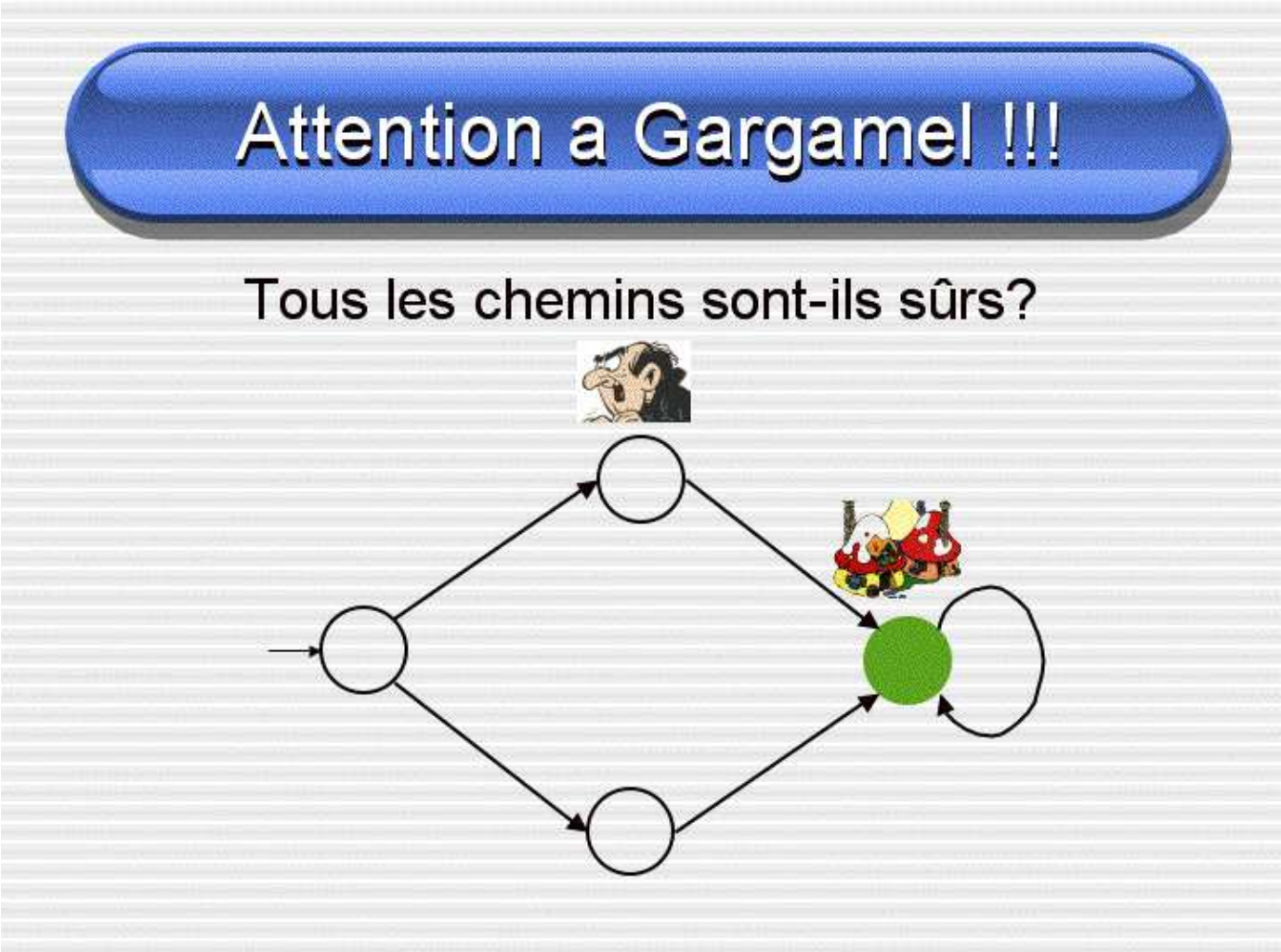
Exemple



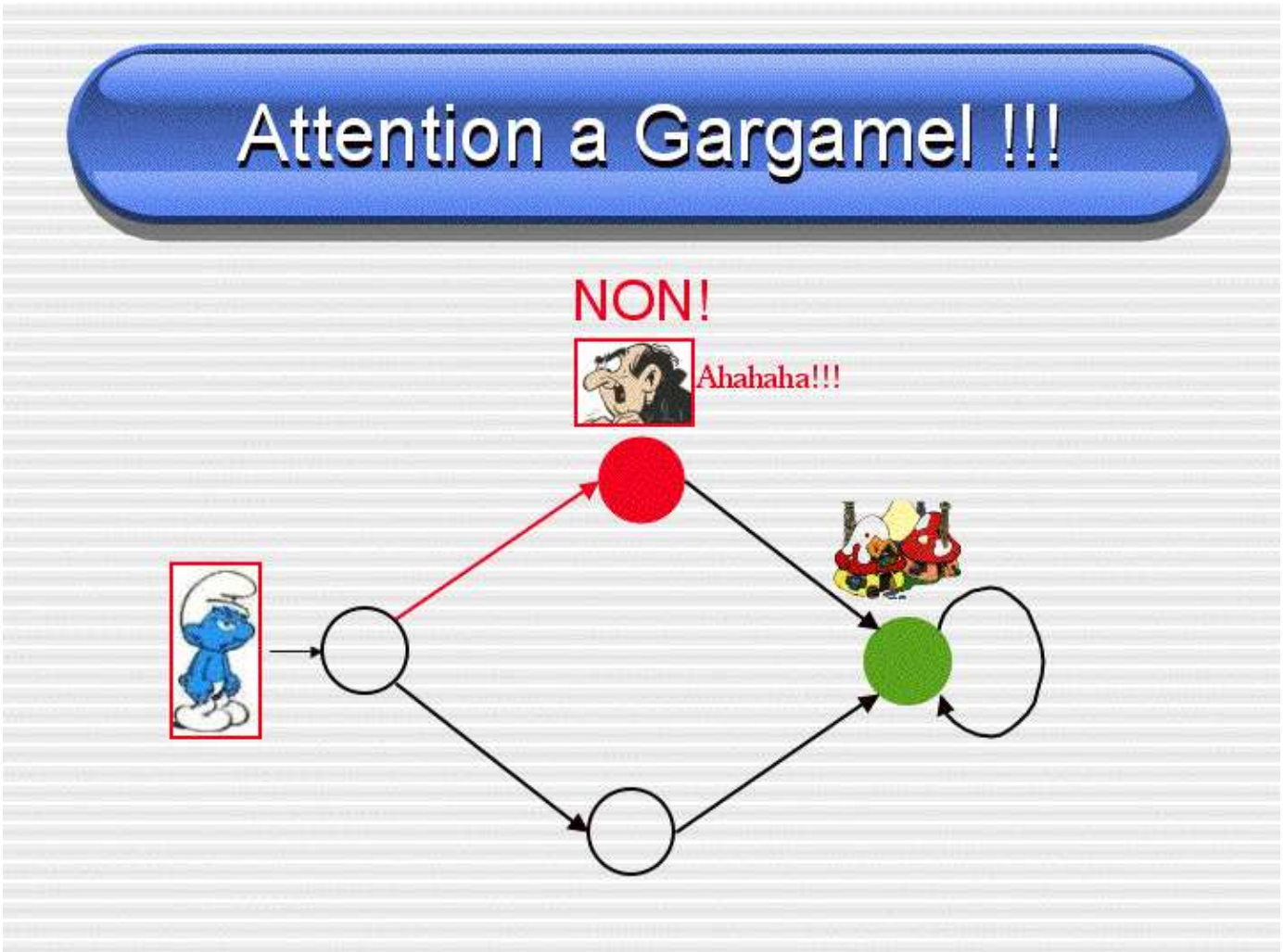
Exemple



Exemple



Exemple



Exemple

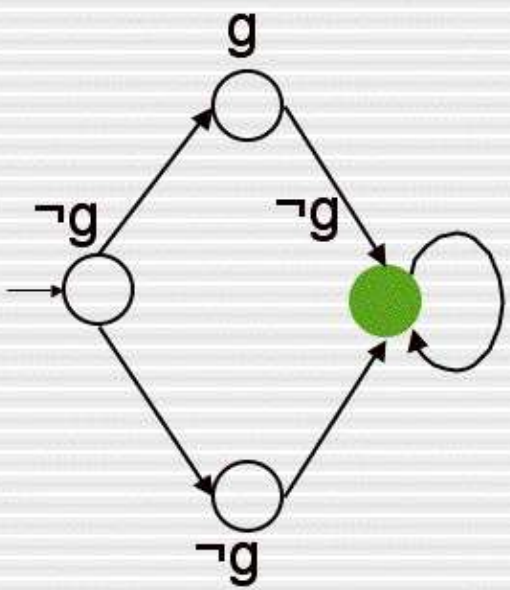
Soyons sérieux 5min...

- A = Pour tous les chemins
- E = Il existe un chemin
- G = Toujours
- g = 
- ¬ = négation

La formule

EG ¬g

est vraie pour le modèle.



La difficulté du model checking

2 difficultés principales

1. système de transitions fini
2. système de transitions suffisament petit

La difficulté du model checking

2 difficultés principales

1. système de transitions fini
2. système de transitions suffisament petit

Système fini : problème (parfois non trivial) de modélisation

- variables à domaines finis, nombre borné de tâches, etc.

La difficulté du model checking

2 difficultés principales

1. système de transitions fini
2. système de transitions suffisament petit

Système fini : problème (parfois non trivial) de modélisation

- variables à domaines finis, nombre borné de tâches, etc.

Système petit : problème algorithmique + modélisation

- 10 variables sur 8 bits : 10^{256} possibilités
- Stocké une possibilité = 10 octets
- Tout stocker $\approx 10^{245}$ To

La difficulté du model checking

2 difficultés principales

1. système de transitions fini
2. système de transitions suffisament petit

Système fini : problème (parfois non trivial) de modélisation

- variables à domaines finis, nombre borné de tâches, etc.

Système petit : problème algorithmique + modélisation

- 10 variables sur 8 bits : 10^{256} possibilités
- Stocké une possibilité = 10 octets
- Tout stocker $\approx 10^{245}$ To

Model Checking = gérer l'explosion d'états

Historique

1975	Constat : vérif. inadaptée à systèmes réactifs
1977	Pnueli propose d'utiliser les logiques temporelles
1981	Model checking de CTL par Clarke et al., Sifakis et al.
1980-1990	Nombreux résultats théoriques
1990-2000	Énorme amélioration des performances Extensions : proba, temps, infini
2000-...	MC adopté par les principaux fondeurs (Intel, etc.) Standardisation du langage temporel PSL Débuts du software model checking (Microsoft) Prix ACM Paris Kanellakis Award 1998 et 2005
2008	Prix Turing décerné à Clarke, Sifakis et Emerson

Bibliographie

Livres

- Principles of Model Checking [Baier-Kaoten 08]
- Model Checking [Clarke-Grumberg-Peled 99]
- Vérification de logiciels [LSV 99]

Cours avancés

- Automates et logiques temporelles (Cachan) [Comon 00]
- Temporal logics (MPRI) [Demri 05]
- Expressiveness of Temporal Logics [Markey 06]

Bibliographie II

Surveys sur divers aspects

- Branching vs linear time: Final showdown [Vardi 01]
- Automata-theoretic model checking revisited [Vardi 07]
- From Church and Prior to PSL [Vardi 07]

Impact industriel

- Formal Verification In a Commercial Setting [Kurshan 97]
- Formal Methods: State of the Art and Future Directions [Clarke-Wing 96]

Plan général du cours

- Leçon 1 : Introduction et Modélisation
- Leçon 2 : Logiques temporelles
- Leçon 3 : Algorithmes de Model checking

Plan

Plan

Champs d'application

Les systèmes finis = variables bornées

Deux cas typiques :

- protocoles de communication
  - difficulté = entrelacement des agents
- processeurs
  - difficulté = nombre de variables

Futur (proche ?)

- Web services
- Drivers logiciels
- Intégration au MDD

Champs d'application

Les systèmes finis = variables bornées

Deux cas typiques :

- protocoles de communication
  - difficulté = entrelacement des agents
- processeurs
  - difficulté = nombre de variables

Futur (proche ?)

- Web services
- Drivers logiciels
- Intégration au MDD

Champs d'application

Les systèmes finis = variables bornées

Deux cas typiques :

- protocoles de communication
  - difficulté = entrelacement des agents
- processeurs
  - difficulté = nombre de variables

Futur (proche ?)

- Web services
- Drivers logiciels
- Intégration au MDD

Process du model checking

Process en trois phases

1. Modéliser le système par $\mathcal{M}$ et la propriété par φ
2. $\mathcal{M} \models \varphi$? Si non, contre-exemple σ.
3. Analyser les résultats
   - Si oui, **ok**. Attention au lien avec système réel
   - Si non, rejouer σ sur système réel.
     - Si vrai bug, alors **erreur**.
     - Sinon goto (1) en raffinant $\mathcal{M}$ grâce à σ.

Utilisation industrielle I

Quelques rappels économiques

Coût des bugs : 64 milliards $/an rien qu'aux US

Coût de la vérification

- 10 milliards $/an en tests rien qu'aux US
- débug de 50 kLOC : 60 jours + 30 kEUR + coûts indirects
- plus de 50% du coût d'un logiciel critique
- en moyenne 30% du coût d'un logiciel quelconque

Coût de correction d'un bug (base = bug découvert au design)

- découvert en test unitaire : surcoût de 5x à 15x
- découvert en test intégration/système : surcoût de 15x à 90x
- découvert en production : surcoût de 50x à 200x

Répartition des bugs (exemple automobile, Bosch)

- 60 % requirement, 20 % design, 10 % code, 10 % système

Avantages du Model checking

- Phases amonts : spécifs et design
- Automatisé
- Trouve mieux les bugs que le test
- Cost efficient

	prise en main	assisté par ordi.	surcoût	debug	validation
test standard	++	-	jeux de tests	+	-
model checking (no modèle fini)	+	+	modélisation	++	+
model checking (modèle fini)	++	++		++	+
preuve	-	-	preuves	+	++

Avantages du Model checking

- Phases amonts : spécifs et design
- Automatisé
- Trouve mieux les bugs que le test
- Cost efficient

	prise en main	assisté par ordi.	surcoût	debug	validation
test standard	++	-	jeux de tests	+	-
model checking (no modèle fini)	+	+	modélisation	++	+
model checking (modèle fini)	++	++		++	+
preuve	-	-	preuves	+	++

Leçon

Model Checking facile à intégrer si le process intègre déjà des modèles finis

Utilisation industrielle III

Utilisation majeure actuelle : composants électroniques

- Process intègrent déjà des modèles formels
- Coût d'une erreur est prohibitif (bug Pentium : 500 millions $)
- Acteurs : Intel, IBM, Motorola, Siemens, Lucent, Bull, Cadence
- Récemment : langage de spécification temporel standardisé PSL

Futur plus ou moins proche

- Transport/énergie. Exemple de Bosch. Intérêt MDD.
- Services Web ?
- Logiciels bas niveau type drivers (Microsoft) ?

Autres applications : NASA, protocoles

Utilisation industrielle IV

En pratique Model Checking utilisé pour débugger

It has been an exciting twenty years, which has seen the research focus evolve [. . .] from a dream of automatic program verification to a reality of computer-aided design debugging.

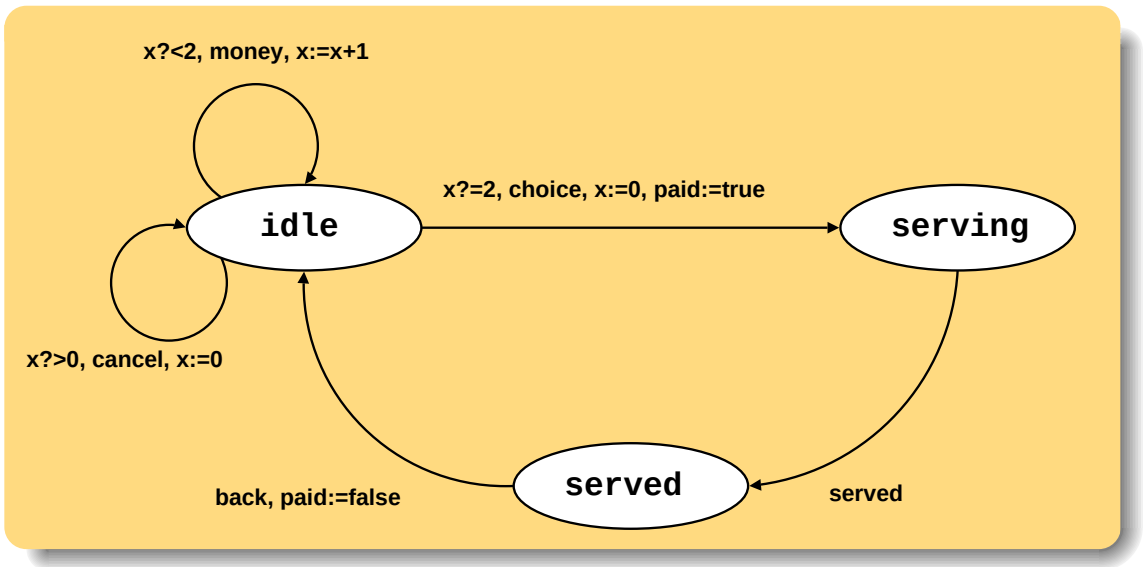
- Thomas A. Henzinger (2001)

Plan

Plan

Machine à états

Systèmes réactifs abstraits sous forme de machines à états

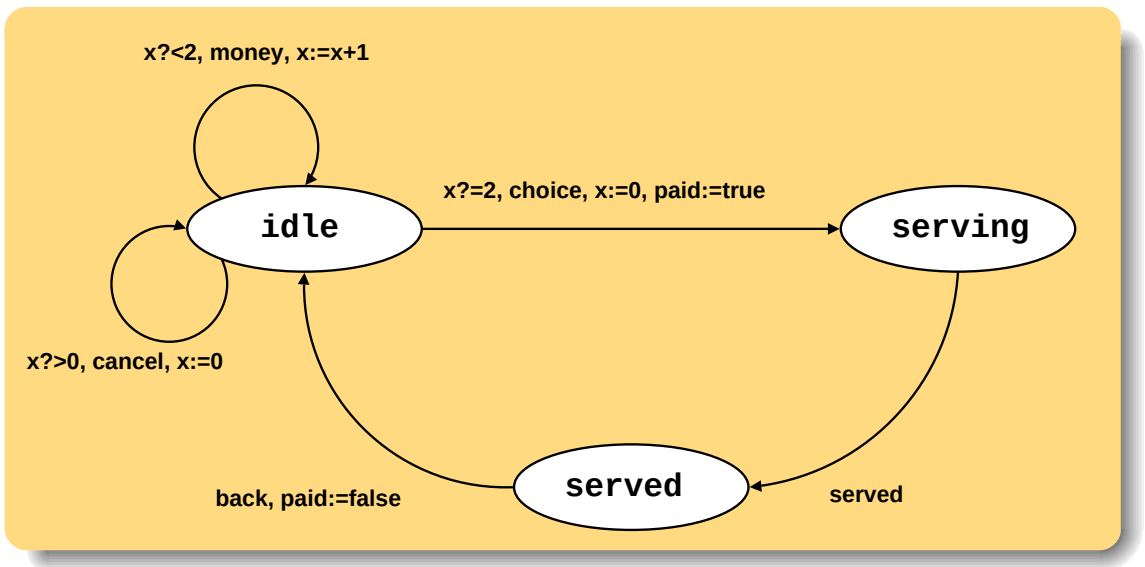


- États de contrôle `idle`, `serving`, `served`
- Variables `x:int`, `paid:bool`
- Transitions `money`, `choice`, `served`, `back`, `cancel`

Machine à états (II)

Machine à états $P = \langle C, V, A, T \rangle$

- C ensemble fini des états de contrôles
- V ensemble fini des variables
- A ensemble fini d'actions sur V
- $T \subseteq C \times A \times C$ ensemble fini de transitions



Systèmes de transitions

Système de transitions S = comportement de P

$S = \langle Q, T, \rightarrow \rangle$

- Q ensemble des états ou configurations
- T ensemble des transitions
- $\rightarrow \subseteq Q \times T \times Q$ relation de transition
- on peut ajouter un état initial $I \in Q$

Q représente les états possibles du système.
Une transition t peut faire passer de l'état q à l'état q' si $(q, t, q') \in \rightarrow$

Notation

on écrit $q \xrightarrow{t} q'$ pour $(q, t, q') \in \rightarrow$

Systèmes de transitions II

Machine à états $P = \langle C, V, A, T \rangle$
Système de transitions $S = \langle Q, T, \rightarrow \rangle$

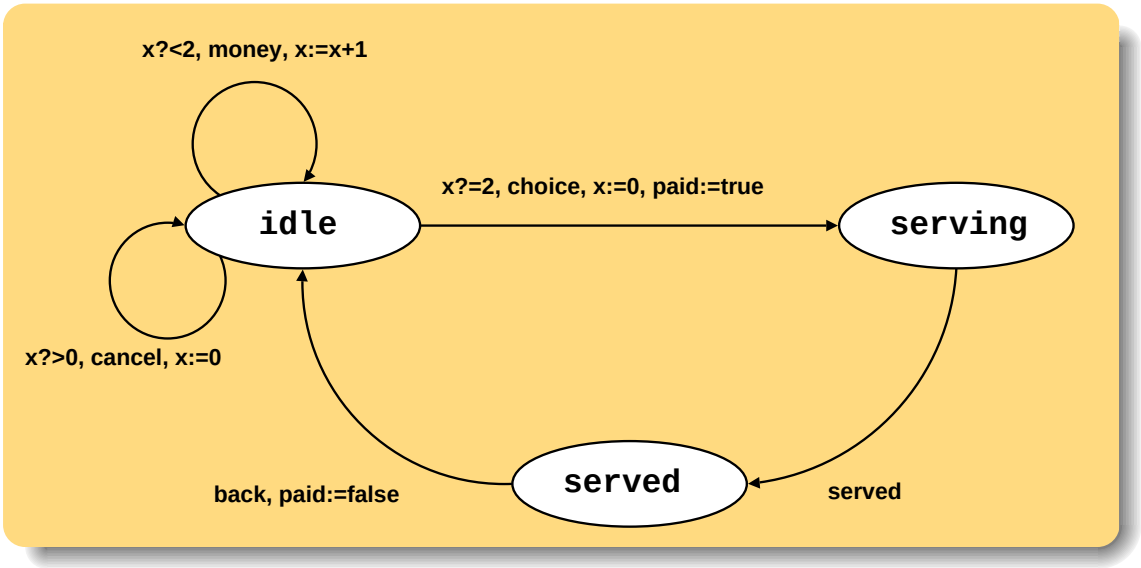
Lien machine à états P - système de transition S

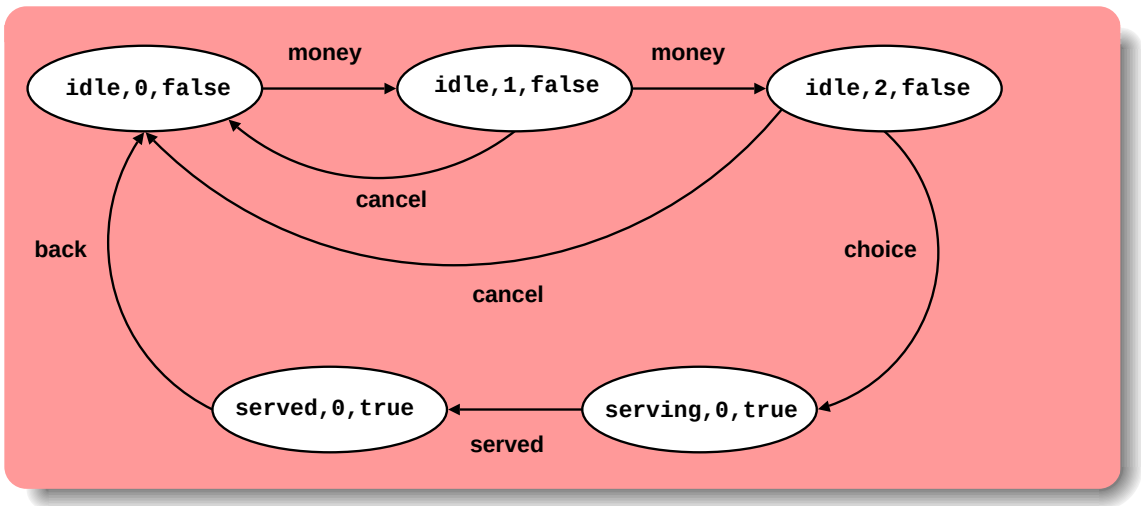
- Donner aux variables $v \in V$ un domaine D
- Associer aux actions $a \in A$ une fonction partielle $[\![a]\!] : D \rightarrow D$
- Alors $Q = C \times D$, et une configuration s'écrit (c, d)
- Et $(c, d) \xrightarrow{t} (c', d')$ si $t = (c, a, c')$ et $[\![a]\!](d) = d'$

Exemple :

- `x` $\mapsto \{0..2\}$, `paid` $\mapsto \{true, false\}$
- `x?=2,x:=2,paid:=true`
  $\mapsto \lambda(a : int, b : bool).ite(a = 2, (a, true), undef)$

Systèmes de transitions III



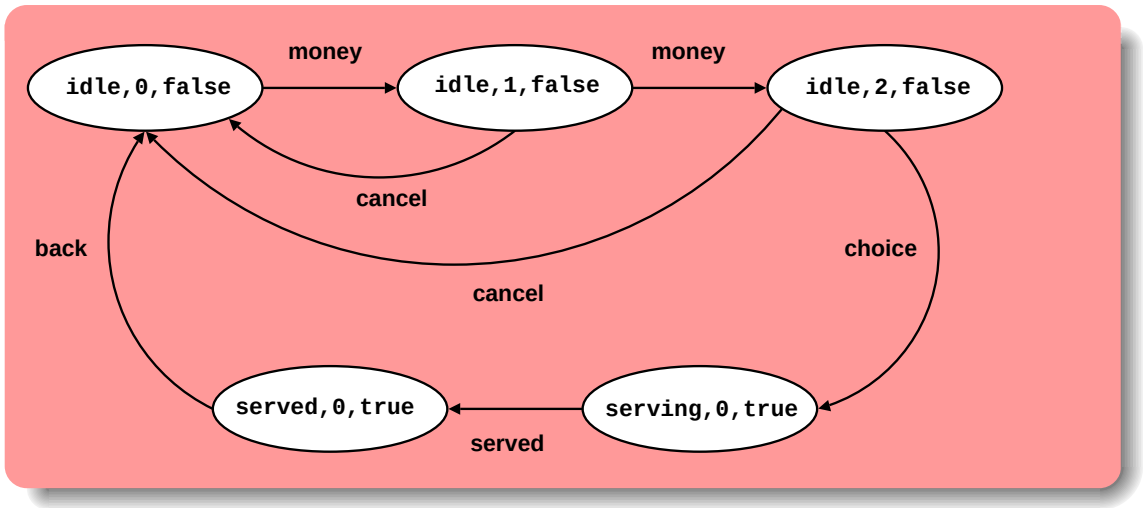


$q \xrightarrow{t} q'$: notation pour $(q, t, q') \in \rightarrow$

Quelques définitions

- $q \rightarrow q'$: il existe une transition $t \in T$ tq $q \xrightarrow{t} q'$
- $q \xrightarrow{t_1 \cdot \ldots \cdot t_k} q'$: il existe $q_1, \ldots, q_{k-1}$ tq $q \xrightarrow{t_1} q_1 \xrightarrow{t_2} \ldots q_{k-1} \xrightarrow{t_k} q$
- $q \xrightarrow{*} q'$: il existe $t_1, \ldots, t_k \in T$ tq $q \xrightarrow{t_1 \cdot \ldots \cdot t_k} q'$

Exécutions du système



- $(\mathrm{i},0,\mathrm{f}) \xrightarrow{\texttt{money}} (\mathrm{i},1,\mathrm{f}) \xrightarrow{\texttt{money}} (\mathrm{i},2,\mathrm{f}) \xrightarrow{\texttt{choice}} (\mathrm{sg},0,\mathrm{t}) \ldots$
- `money, money, choice,served,back,money ...`

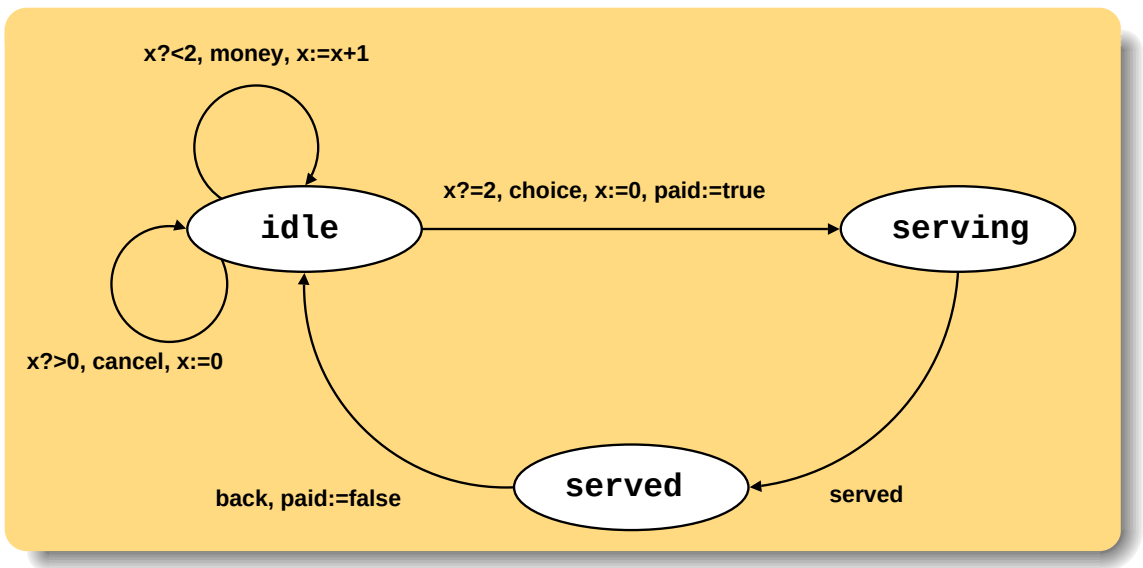
$\mathcal{L}(S) =$ Langage de $S =$ ensemble des exécutions de S

Récapitulatif

- Système réactif = système de départ, monde réel
- Machine à état P = syntaxe du modèle
- Système de transition S = sémantique de P

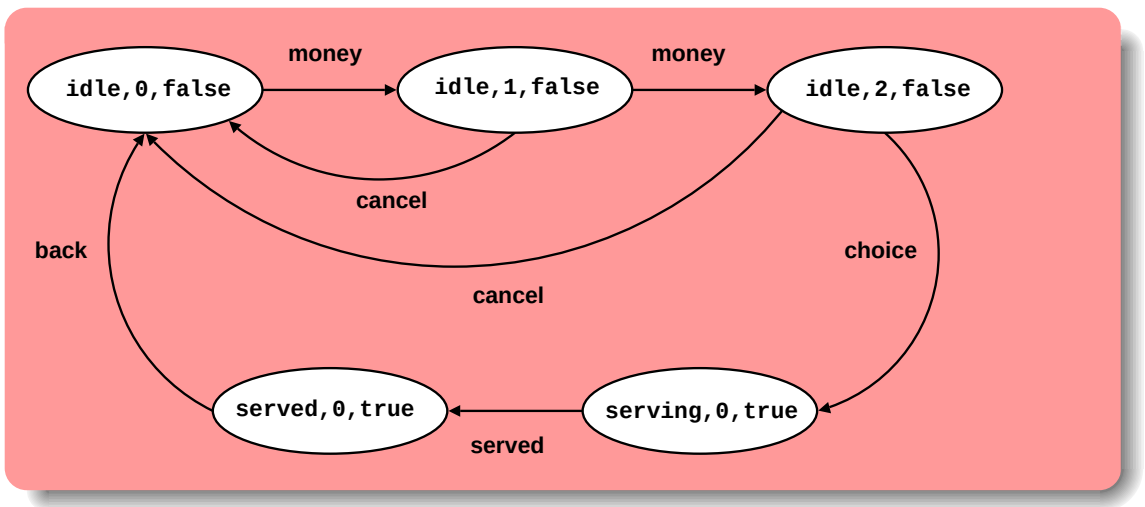
Récapitulatif

- Système réactif = système de départ, monde réel
- Machine à état P = syntaxe du modèle
- Système de transition S = sémantique de P



Récapitulatif

- Système réactif = système de départ, monde réel
- Machine à état P = syntaxe du modèle
- Système de transition S = sémantique de P



Au fait ...

Question : d'où vient la machine à états ?

Deux approches

- partir des spécifications
- partir du programme (besoin automatisation)

Plan

Plan

Propriétés sur les systèmes réactifs

- Accessibilité *Une certaine situation peut être atteinte*
- Invariance *Chaque état local respecte une bonne propriété*
- Sûreté *Quelquechose de mauvais n'arrive jamais*
- Vivacité *Quelquechose de bon finit par arriver*
- Équité *Quelquechose de bon se répète infiniment souvent*
- Équivalence comportementale *Est-ce que 2 systèmes sont équivalents ?*

Le même algorithme permet de traiter toutes ces propriétés

Propriétés sur les systèmes réactifs

- Accessibilité *Une certaine situation peut être atteinte*
- Invariance *Chaque état local respecte une bonne propriété*
- Sûreté *Quelquechose de mauvais n'arrive jamais*
- Vivacité *Quelquechose de bon finit par arriver*
- Équité *Quelquechose de bon se répète infiniment souvent*
- Équivalence comportementale *Est-ce que 2 systèmes sont équivalents ?*

Le même algorithme permet de traiter toutes ces propriétés

États accessibles

Un état $q \in Q$ est accessibles à partir de q_0 si il existe un chemin de $S = \langle Q, T, \rightarrow \rangle$ menant de q_0 à q.

Accessibilité en un coup : $\text{post} = \{(q, q') | \exists t \in T, q \xrightarrow{t} q'\}$

Ensemble d'accessibilité à partir de q_0 : $\text{post}^*(q_0)$

- facile à calculer dans le cas fini
- itération de $X \mapsto \text{post}(X) \cup X$ à partir de $\{q_0\}$ jusqu'à stabilisation
- permet de vérifier accessibilité et invariance

Invariance et accessibilité

Invariance : tous les états ont une bonne propriété

ex : on a toujours $x \neq 0$

Accessibilité : un certain ensemble d'états est accessible

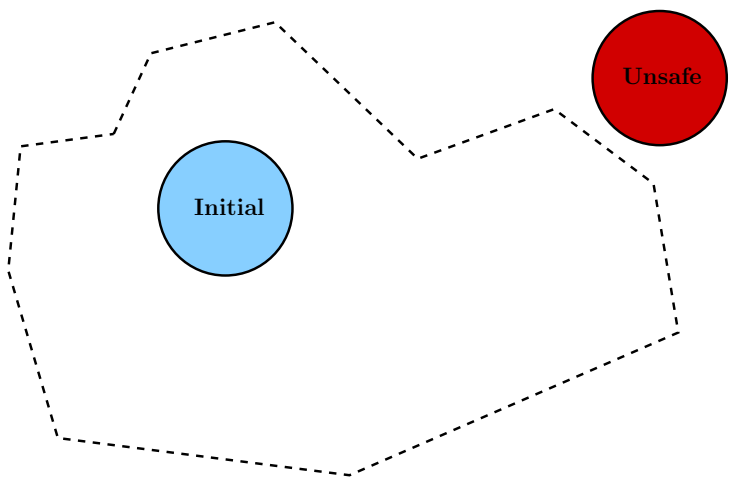
ex : tout point du programme peut être atteint

Propriétés simples mais fondamentales.

Facile à vérifier grâce à $\text{post}^*(q_0)$

- Acc : est-ce que $\text{post}^*(q_0) \cap A \neq \emptyset$?
- Inv : est-ce que $\text{post}^*(q_0) \subseteq I$?

Idée : itération de post + tests ensemblistes



Invariance et accessibilité

Invariance : tous les états ont une bonne propriété

ex : on a toujours $x \neq 0$

Accessibilité : un certain ensemble d'états est accessible

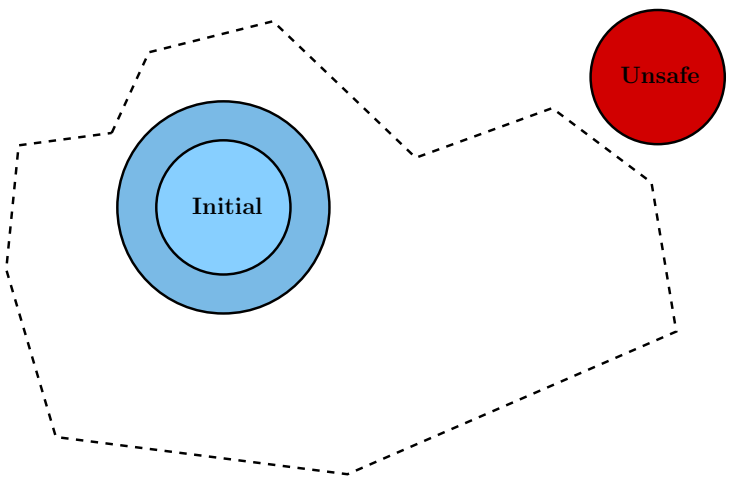
ex : tout point du programme peut être atteint

Propriétés simples mais fondamentales.

Facile à vérifier grâce à $\text{post}^*(q_0)$

- Acc : est-ce que $\text{post}^*(q_0) \cap A \neq \emptyset$?
- Inv : est-ce que $\text{post}^*(q_0) \subseteq I$?

Idée : itération de post + tests ensemblistes



Invariance et accessibilité

Invariance : tous les états ont une bonne propriété

ex : on a toujours $x \neq 0$

Accessibilité : un certain ensemble d'états est accessible

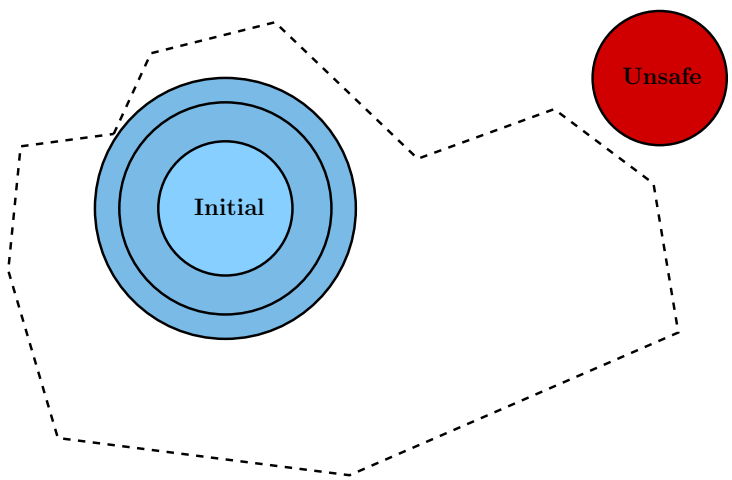
ex : tout point du programme peut être atteint

Propriétés simples mais fondamentales.

Facile à vérifier grâce à $\text{post}^*(q_0)$

- Acc : est-ce que $\text{post}^*(q_0) \cap A \neq \emptyset$?
- Inv : est-ce que $\text{post}^*(q_0) \subseteq I$?

Idée : itération de post + tests ensemblistes



Invariance et accessibilité

Invariance : tous les états ont une bonne propriété

ex : on a toujours $x \neq 0$

Accessibilité : un certain ensemble d'états est accessible

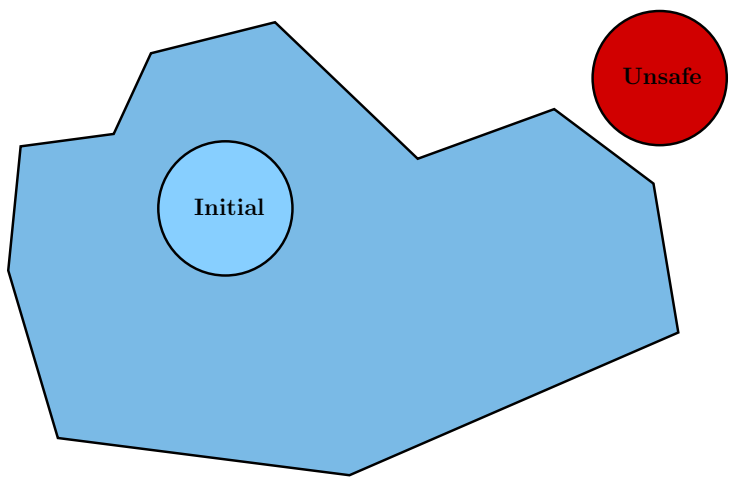
ex : tout point du programme peut être atteint

Propriétés simples mais fondamentales.

Facile à vérifier grâce à $\text{post}^*(q_0)$

- Acc : est-ce que $\text{post}^*(q_0) \cap A \neq \emptyset$?
- Inv : est-ce que $\text{post}^*(q_0) \subseteq I$?

Idée : itération de post + tests ensemblistes



Calcul de $\text{post}^*(q_0)$

Entrées : machine P avec sémantique sous-jacente S, état q_0

Sorties : ensemble d'accessibilité $\text{post}^*(q_0)$

$R := \emptyset$ /* *états atteints et traîtés* */
$Q := \{q_0\}$ /* *états atteints non traîtés* */
Tant que Q non vide **faire**
  choisir $q \in Q$, $Q := Q - q$
  Si $q \notin R$ **alors**
    $R := R + q$, $T' := T$, $post_q := \emptyset$
    Tant que T' non vide **faire** /* *calcul de post*(q) */
      choisir $t \in T'$
      Si $q \xrightarrow{t} q'$ **alors** $post_q := post_q + q'$ **Fin Si**
    Fin Tant que
    $Q := Q + \text{post}_q$
  Fin Si
Fin Tant que
retourner R

Complexité : linéaire en $|S|$, mais exponentielle en $|P|$ (variables)

Plan

Plan

Concurrence

Lire le cours

Concurrence = plusieurs composants en parallèle

- très courant
- mode d'exécution : synchrone ou asynchrone
- mode de communication : mémoire partagée, envoie de messages

Exemples

- threads java : asynchrone + mémoire partagée
- processus unix : asynchrone + envoie de messages
- circuit : synchrone + mémoire partagée

Modélisation pour le Model Checking

- produit de machines à états
- différents produits (sync/async), var partagées, synchro
- modélisation plus naturelle et plus concise
- attention à l'explosion combinatoire

Concurrence II

Soit les deux machines

- $P_1 = \langle C_1, V_1, A_1, T_1 \rangle$, sémantique $S_1 = \langle C_1 \times D_1, T_1, \rightarrow_1 \rangle$
- $P_2 = \langle C_2, V_2, A_2, T_2 \rangle$, sémantique $S_2 = \langle C_2 \times D_2, T_2, \rightarrow_2 \rangle$

Produit de transitions

$((q_1, q_2), (d_1, d_2)) \xrightarrow{t_1 \otimes t_2}_{\otimes} ((q_1', q_2'), (d_1', d_2'))$
si $(q_1, d_1) \xrightarrow{t_1}_1 (q_1', d_1')$ et $(q_2, d_2) \xrightarrow{t_2}_2 (q_2', d_2')$

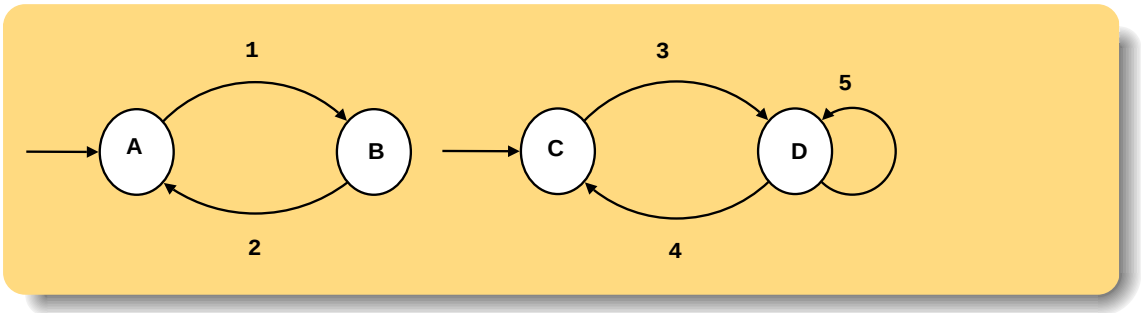
Produit de machines concurrentes

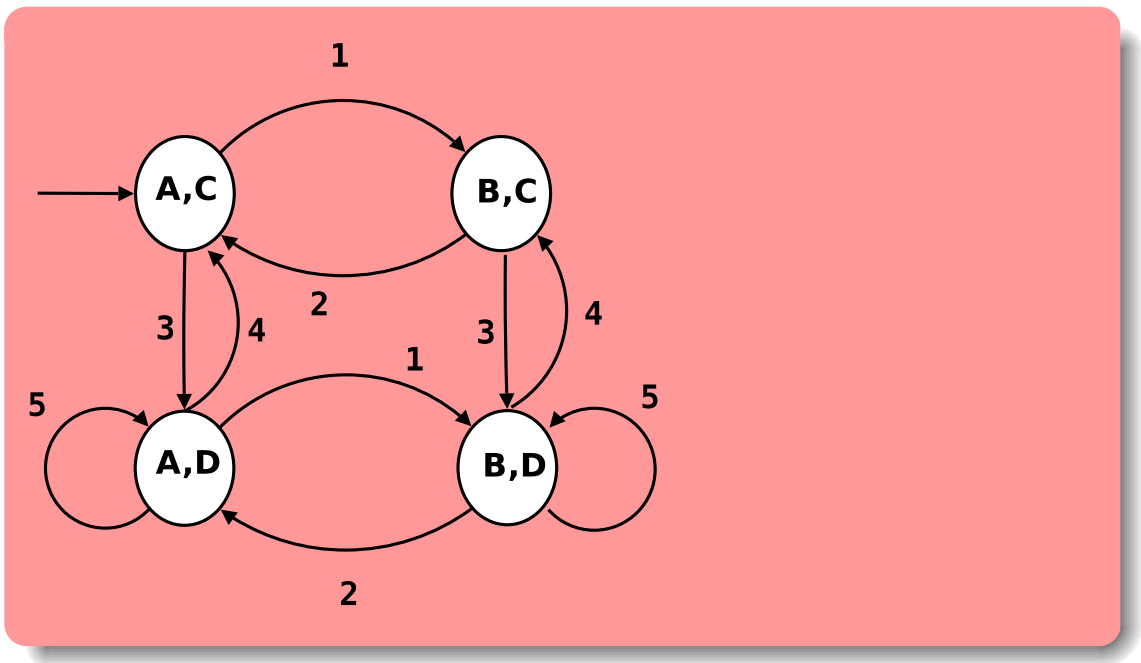
- $P_1 \otimes P_2 = \langle C_1 \times C_2, V_1 + V_2, (A_1 + \varepsilon) \times (A_2 + \varepsilon), T_\otimes \rangle$
- sémantique $S_{1\otimes 2} = \langle (C_1 \times C_2) \times (D_1 \times D_2), T_\otimes, \rightarrow_\otimes \rangle$

Où $T_\otimes$ contient exactement ($t_1 \in T_1$, $t_2 \in T_2$, ε transition vide)

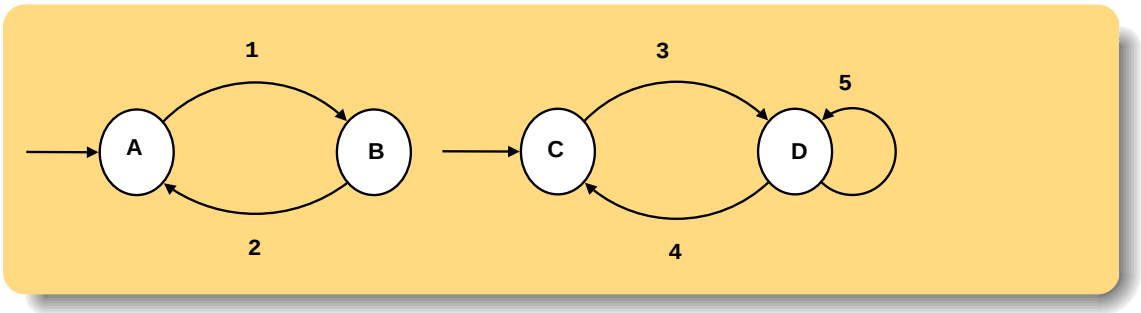
- produit synchrone : tous les (t_1, t_2)
- produit asynchrone : tous les (t_1, ε) et les (ε, t_2)

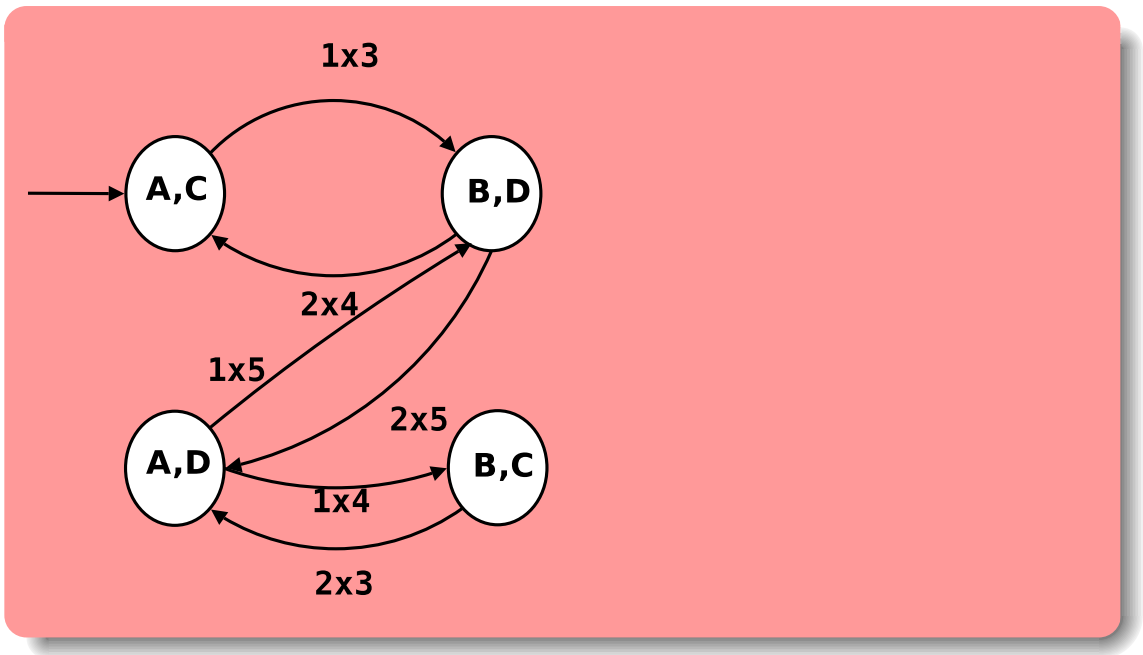
Produit asynchrone





Produit synchrone





Concurrence V

Contraintes de synchronisation

- ensembles de transitions ne pouvant se déclencher que simultanément
- si deux transitions, on parle de *Rendez-vous (RDV)*

Utilité pratique

- abstraction de certains mécanismes logiciels (ex : RDV par double envoie de message)
- modélisation de contraintes matérielles

Peut s'utiliser avec chaque type de produit

Équité

Lire le cours

Problème des chemins aberrants dus à la modélisation

- chemins impossibles ou très peu probables
- peuvent fausser l'analyse

Exemples

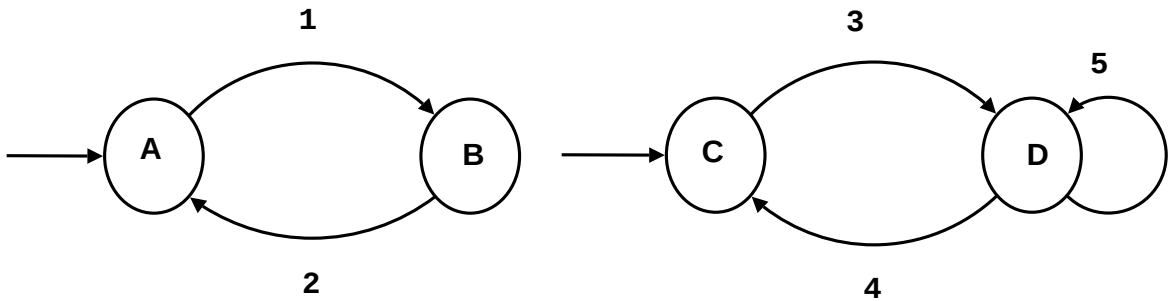
- un des composants n'a jamais la main
- un canal de communication perd systématiquement les messages

On rajoute une hypothèse d'équité H au système S

- H du genre : S "progresse" régulièrement
- permet d'écarter certains comportements aberrants
- H dans formule *(leçon 2)* ou dans modèle *(leçon 3)*

Très courant en pratique : ex des machines concurrentes

Équité II



En sémantique asynchrone

- le chemin 1.2.1.2.1.2.1.2. . . . (infini) est un chemin légal du système
- chemin certainement irréaliste
- rajoute une sorte de deadlock sur la machine 2

Contraintes d'équité

- sur le modèle : passer ∞ souvent par A, B, C, D
- dans la formule (but $= \varphi$) : $\mathbf{F}^{\infty}(A) \wedge \mathbf{F}^{\infty}(D) \implies \varphi$

Modélisation, suite

Lire le cours

- non déterminisme *(cf. lien concurrence)*
- actions observables
- types de variables
- modularité et hiérarchie
- systèmes ouverts

Sûreté

Lire le cours

Sûreté : quelquechose de mauvais n'arrive jamais.

Quand j'accède à mon compte, j'ai entré le bon password avant.

Autre définition : contre-exemples finis.

Plus général que invariance.

Comment vérifier : se ramener à accessibilité dans un système modifié

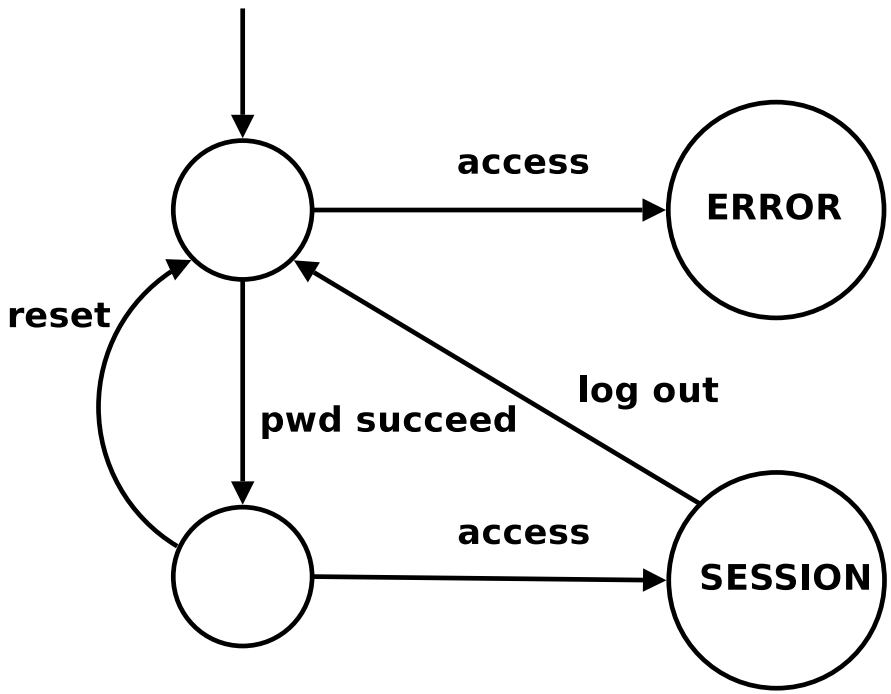
- Variables d'historique
- Amélioration : automates observeurs

Les techniques de calcul de $\text{post}^*(q_0)$ permettent de vérifier l'accessibilité, l'invariance et la sûreté.

Sûreté II

Propriété simple : Quand j'accède à mon compte, j'ai entré le bon password avant.

Automate observateur $\mathcal{O}$ pour la proriété



Sûreté III

Puis

- annoter dans le système original $\mathcal{S}$ les transitions correspondants à “entrer le bon password”, “accéder au compte”
- faire un produit de $\mathcal{S}$ et $\mathcal{O}$ (transitions : transitions non annotées de $\mathcal{S}$ + synchronisation transitions annotées de $\mathcal{S}$ et de $\mathcal{O}$)
- vérifier que l'état de contrôle $(\_, \mathit{Bad})$ n'est pas accessible

Sûreté IV

Remarques sur l'automate observeur

- peut lire les variables du système à vérifier, mais ne peut les écrire
- peut avoir ses propres variables (lire/écrire)
- pas forcément un automate fini (mais attention à décidabilité)

Remarques sur propriété de sûreté

- φ propriété de sûreté ssi φ a un automate observeur (arbitrairement compliqué, mais sur mots finis) qui modélise ses exécutions incorrectes

Approche par automate observeur n'est pas dédiée au model checking

- test, runtime monitoring
- preuve, analyse statique