

# CONCEPTION ET VÉRIFICATION DE SYSTÈMES CRITIQUES

## INTRODUCTION AUX MÉTHODES FORMELLES

🎓 2A Coursus Ingénieurs - ST5 : Modélisation fonctionnelle et régulation

🏛️ CentraleSupélec - Université Paris-Saclay - 2025/2026



**Idir AIT SADOUNE**

[idir.aitsadoune@centralesupelec.fr](mailto:idir.aitsadoune@centralesupelec.fr)

# PLAN

- La nécessité des méthodes formelles/vérification
- La vérification d'un programme
- Le principe du Model-Checking
- La modélisation d'un logiciel/système

[Retour au plan](#) - [Retour à l'accueil](#)

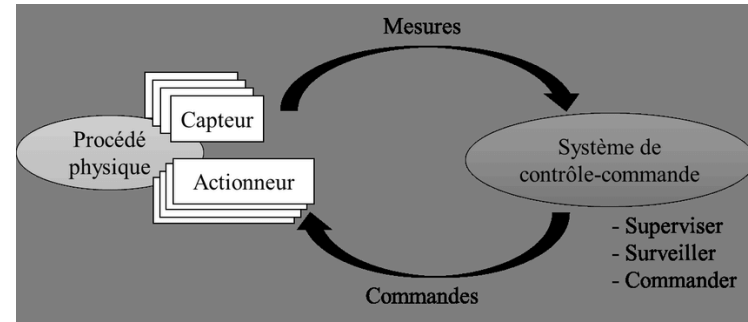
# PLAN

- La nécessité des méthodes formelles/vérification
- La vérification d'un programme
- Le principe du Model-Checking
- La modélisation d'un logiciel/système

[Retour au plan](#) - [Retour à l'accueil](#)

# FIABILITÉ DES SYSTÈMES DE CONTRÔLE

- Un système de contrôle est composé de 3 parties :
  1. Capteurs
  2. Actionneurs
  3. **Logiciel** de contrôle qui est **critique** dans le contexte d'un **système critique**!



**Logiciel critique** → pour lequel une défaillance peut être catastrophique

- **mortelle ou/et extrêmement coûteuse**

Quelques défaillances spectaculaires de logiciels critiques :

- **Crash of Ariane 5**
- **LASCAD** : Crash of London Ambulance CAD service
- **Therac-25** : 7 deaths of cancer patients due to overdoses of radiation

# SOLUTIONS

- Les **règles** et les **techniques** de programmation.
- Le **support** des langages de programmation.
- Les **méthodologies** de conception et de développement.
- Le test.
- Les **méthodes formelles**
  - ➡ **méthodes d'ingénierie** basées sur des **approches mathématiques** utilisées pour développer et analyser des systèmes (logiciels).
  - ➡ **démarche globale (langages et outils de vérification)**.



## Formal Methods for Software Engineering

Languages, Methods, Application Domains  
Springer 2022

# SPÉCIFICATION, CONCEPTION ET VÉRIFICATION

- **La spécification formelle** → **description rigoureuse** et **non ambiguë** du comportement attendu d'un système (logiciel).
  - ▢ modèle mathématique décrivant **ce que doit faire** le système (logiciel).
  - ▢ modélisation par un **langage mathématique** (syntaxe, logique, sémantique...).
- **La conception formelle** → **description rigoureuse** et **non ambiguë** de la réalisation du système (logiciel).
  - ▢ modèle mathématique décrivant **la construction** du système (logiciel).
  - ▢ modélisation par un **langage mathématique** (syntaxe, logique, sémantique...).
- La **Vérification formelle** → **démontrer mathématiquement** qu'un système (logiciel) **respecte les exigences** identifiées dans la spécification.
  - ▢ **démonstration** que la conception correspond bien à la spécification.
  - ▢ simulation, preuve de théorèmes, model checking...

# VÉRIFICATION DU LOGICIEL DE CONTRÔLE

## Le processus de vérification

1. Déterminer ce que le logiciel est censé faire (**Spécification**)
2. Prendre le logiciel (**Conception**)
3. Démontrer que le logiciel fait ce qu'il est censé faire (**Vérification**)

Imposée par les organismes de certification ( $\implies$  quelques exemples)

# TEST vs VERIFICATION

- Les tests sont une technique dynamique courante où le système est exécuté
- **Procédure de test :**
  - prendre une implémentation du système
  - la stimuler avec certaines données en entrée (les cas de tests)
  - observer la réaction et vérifier si elle est "souhaitable"
- **Inconvénients des tests :**
  - ✗ le **nombre de cas possibles** est très important (voire infini)
  - ✗ les **comportements inexplorés** peuvent contenir un bug fatal
  - ✗ les tests privilégient **les scénarios les plus probables**

- Les tests peuvent prouver la présence d'erreurs, et **non leur absence !**
- **La vérification prouve l'absence d'erreurs (ou les trouve)**

# LA NÉCESSITÉ DES MÉTHODES FORMELLES

⇒ les slides de la présentation de la ST58

# PLAN

- La nécessité des méthodes formelles/vérification
- La vérification d'un programme
- Le principe du Model-Checking
- La modélisation d'un logiciel/système

[Retour au plan](#) - [Retour à l'accueil](#)

# DEFINITION

## D'UN PROGRAMME SÉQUENTIEL

- Une séquence d'instructions qui **se termine** et dont le **résultat** est calculé à partir des **données initiales** (les entrées)
- **Exemple** → programme de tri
  - Initial data : Array  $T$  of size  $N$
  - Result : Sorted Array  $T$  of size  $N$

```
1 def bubble_sort(T):
2     N = len(T)
3     for i in range(N):
4         # Last i elements are already in place
5         for j in range(0, N - i - 1):
6             if T[j] > T[j + 1]:
7                 # Swap if element is greater than the next
8                 T[j], T[j + 1] = T[j + 1], T[j]
```

# VÉRIFICATION FORMELLE DES PROGRAMMES SÉQUENTIELS

## Définitions

- **PreCondition** → propriété satisfaite par les données initiales du programme avant l'exécution des instructions
  - **PostCondition** → propriété satisfaite par le résultat et les variables du programme après l'exécution des instructions
- 
- Prouver que si la **PreCondition** est satisfaite, alors la **PostCondition** est satisfaite  $\Rightarrow$  trouver la **PreCondition** qui permet d'aboutir à la **PostCondition**
  - **Exemple** → programme de tri
    - Initial data : Array  $T$  of size  $N$
    - Result : Sorted Array  $T$  of size  $N$
    - Post-Condition :  $\forall n, m \in [1..N], n < m \implies T[n] < T[m]$
    - Pre-Condition :  $\forall n, m \in [1..N], n \neq m \implies T[n] \neq T[m]$

# VÉRIFICATION FORMELLE DES PROGRAMMES SÉQUENTIELS

**Spécification**  $\rightarrow$  programme de tri

- Initial data : Array  $T$  of size  $N$
- Result : Sorted Array  $T$  of size  $N$
- Post-Condition :  $\forall n, m \in [1..N], n < m \implies T[n] < T[m]$
- Pre-Condition :  $\forall n, m \in [1..N], n \neq m \implies T[n] \neq T[m]$

**Conception**  $\rightarrow$  programme de tri

```
input T : array [N] of integer
for i=0 to N-1 do
  for j=0 to N-i-1 do
    if T[j] > T[j + 1] do
      T[j], T[j + 1] = T[j + 1], T[j]
    end
  end
end
```

# VÉRIFICATION FORMELLE DES PROGRAMMES SÉQUENTIELS

**Spécification** → programme de tri

- Initial data : Array  $T$  of size  $N$
- Result : Sorted Array  $T$  of size  $N$
- Post-Condition :  $\forall n, m \in [1..N], n < m \implies T[n] < T[m]$
- Pre-Condition :  $\forall n, m \in [1..N], n \neq m \implies T[n] \neq T[m]$

**Implémentation** → programme de tri

```
1 def bubble_sort(T):
2     N = len(T)
3     for i in range(N):
4         for j in range(0, N - i - 1):
5             if T[j] > T[j + 1]:
6                 T[j], T[j + 1] = T[j + 1], T[j]
```

# VÉRIFICATION FORMELLE

## QUESTION ?

**Spécification** → programme de tri

- Post-Condition :  $\forall n, m \in [1..N], n < m \implies T[n] < T[m]$
- Pre-Condition :  $\forall n, m \in [1..N], n \neq m \implies T[n] \neq T[m]$

**Conception** → programme de tri

```
input T : array [N] of integer
for i=0 to N-1 do
  for j=0 to N-i-1 do
    if T[j] > T[j + 1] do
      T[j], T[j + 1] = T[j + 1], T[j]
    end
  end
end
```

⇒ Prouver que si la **PreCondition** est satisfaite, alors la **PostCondition** est satisfaite

# LE SYSTÈME DE PREUVE DE HOARE

- Le système de preuve de **Hoare** fournit pour chaque type d'instruction **une règle** pour trouver la **PreCondition** la plus générale  $P$  (la forme générale :  $\{P\} I \{Q\}$  )
- Le triplet  $\{P\} I \{Q\}$  exprime la propriété suivante :
  - Si  $P$  (**PreCondition**) est vérifiée avant l'exécution de  $I$ ,
  - alors  $Q$  (**PostCondition**) est vérifiée après son exécution.
- Une règle de **Hoare** est constituée :
  - d'hypothèses  $H_1, \dots, H_n$
  - d'une conclusion  $C$

$$\frac{H_1, \dots, H_n}{C}$$

# LE SYSTÈME DE PREUVE DE HOARE

## LA RÈGLE DE LA SÉQUENCE

$$\frac{\{P\} I \{R\} \quad \{R\} J \{Q\}}{\{P\} I ; J \{Q\}}$$

$$\frac{\{x > 0\} x := x + 1 \{x > 1\} \quad \{x > 1\} y := 2 * x \{y > 2\}}{\{x > 0\} x := x + 1 ; y := 2 * x \{y > 2\}}$$

# LE SYSTÈME DE PREUVE DE HOARE

## LA RÈGLE DE LA CONDITIONNELLE

$$\frac{\{P \wedge C\} I \{Q\} \quad \{P \wedge \neg C\} J \{Q\}}{\{P\} \text{ if } C \text{ then } I \text{ else } J \{Q\}}$$

$$\frac{\{true \wedge y \geq 0\} x := y \{x \geq 0\} \quad \{true \wedge \neg(y \geq 0)\} x := 0 \{x \geq 0\}}{\{true\} \text{ if } y \geq 0 \text{ then } x := y \text{ else } x := 0 \{x \geq 0\}}$$

# LE SYSTÈME DE PREUVE DE HOARE

## LA RÈGLE DE LA BOUCLE

$$\frac{\{P \wedge C\} I \{P\}}{\{P\} \text{ while } C \text{ do } I \{P \wedge \neg C\}}$$

$$\frac{\frac{\{x \leq b \wedge x < b\} \ x := x + 1 \ \{x \leq b\}}{\{x \leq b\} \text{ while } x < b \text{ do } x := x + 1 \ \{x \leq b \wedge x \geq b\}}}{\{x \leq b\} \text{ while } x < b \text{ do } x := x + 1 \ \{x = b\}}$$

# LE SYSTÈME DE PREUVE DE HOARE

## LA RÈGLE DE L'AFFECTATION

$$\frac{}{\{Q[x \leftarrow E]\} x := E \{Q\}}$$

Raisonnement arrière  $\rightarrow$  pour que  $Q$  soit vraie après cette affectation, il faut qu'elle soit déjà vraie pour la valeur que va prendre  $x$ .

$$\frac{}{\{x + 2 > 2\} x := x + 2 \{x > 2\}}$$

# LE SYSTÈME DE PREUVE DE HOARE

## LES RÈGLES LOGIQUES

$$\frac{P \Rightarrow P' \quad \{P'\} I \{Q'\} \quad Q' \Rightarrow Q}{\{P\} I \{Q\}}$$

$$\frac{\{true \wedge y \geq 0\} x := y \{x \geq 0\} \quad \{true \wedge \neg(y \geq 0)\} x := 0 \{x \geq 0\}}{\{true\} \text{ if } y \geq 0 \text{ then } x := y \text{ else } x := 0 \{x \geq 0\}}$$

$$\frac{\{true \wedge y \geq 0\} \Rightarrow \{y \geq 0\} \quad \{y \geq 0\} x := y \{x \geq 0\}}{\{true \wedge y \geq 0\} x := y \{x \geq 0\}} \quad \frac{\{true \wedge \neg(y \geq 0)\} \Rightarrow \{0 \geq 0\} \quad \{0 \geq 0\} x := 0 \{x \geq 0\}}{\{true \wedge \neg(y \geq 0)\} x := 0 \{x \geq 0\}}$$

# LA LOGIQUE DU PREMIER ORDRE (FOL)

- La **preuve du programme** est basée sur le **système formel** de la **logique du premier ordre (FOL)**
  - PreConditions, PostConditions, Invariants, Assertions . . .
- **FOL** est **la logique** que vous avez l'habitude d'utiliser en **mathématiques**
- La syntaxe :
$$t ::= c \mid x \mid f(t, \dots, t)$$
$$\phi ::= true \mid a \mid t = t \mid P(t, \dots, t) \mid \phi \wedge \phi \mid \neg \phi \mid \exists x. \phi$$
- La **sémantique** est l'interprétation habituelle utilisée en **mathématiques**

# LE SYSTÈME DE PREUVE DE HOARE

## EN PRATIQUE ...

Le système de **Hoare** est complexe à utiliser :

- ✗ règles lourdes à appliquer
- ✗ taille des arbres de preuve
- ✓ mieux adapté à une preuve automatisée (ex. **Méthode B**)

# PLAN

- La nécessité des méthodes formelles/vérification
- La vérification d'un programme
- Le principe du Model-Checking
- La modélisation d'un logiciel/système

[Retour au plan](#) - [Retour à l'accueil](#)

# HISTOIRE DE LA VÉRIFICATION FORMELLE

## Avant . . .

- Le code du logiciel était séquentiel
- Les propriétés sont exprimées en utilisant la **logique du premier ordre**
- **La preuve de théorème** : ex. la méthode B
- Difficilement automatisé : **semi-décidable**

## Après les années 80

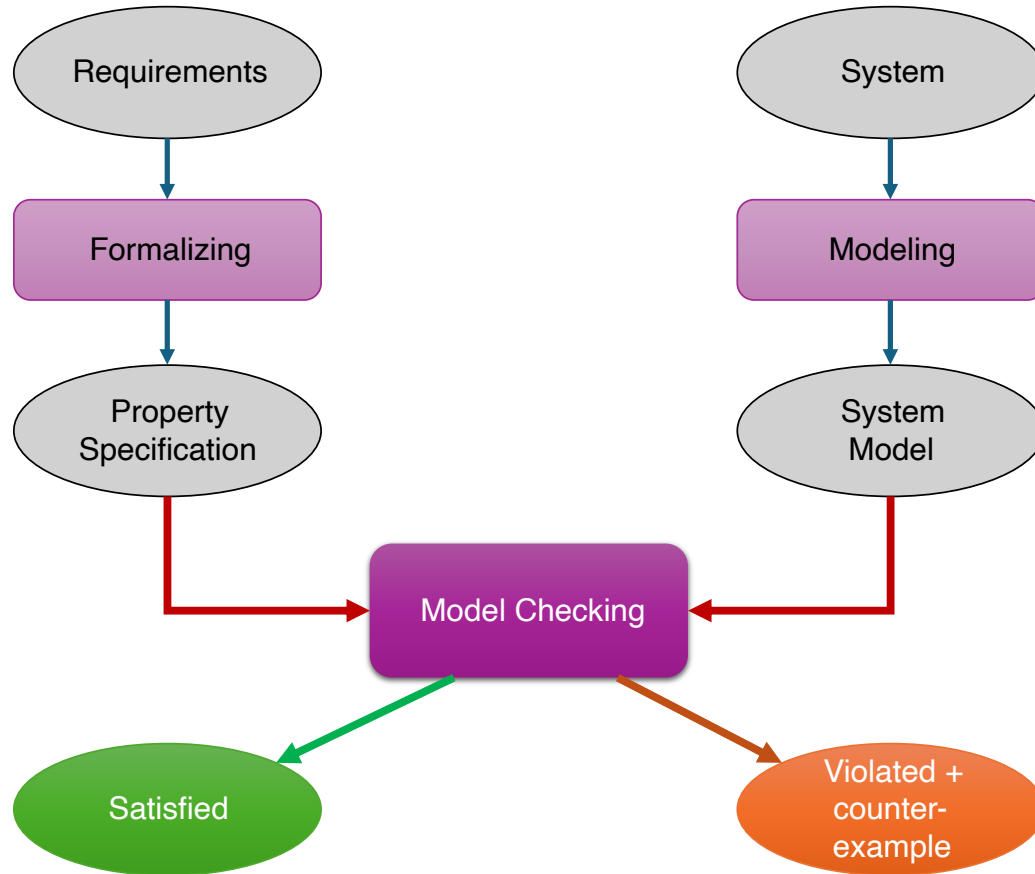
- Le logiciel est **concurrent** et réactif
- Les propriétés sont exprimées en **logique temporelle**
- Démontrer des propriétés telles que la sécurité, la vivacité et l'équité . . .
- ex. Model Checking
- Méthodes automatisées : **décidable**

# LA FORMALISATION DE LA VÉRIFICATION D'UN MODÈLE

La forme formelle du problème de vérification est  $M \models^? \varphi$  avec:

- $M$  est la représentation formelle du système observé
- $\varphi$  est la représentation formelle de la propriété à vérifier

# PRINCIPE DU MODEL-CHECKING

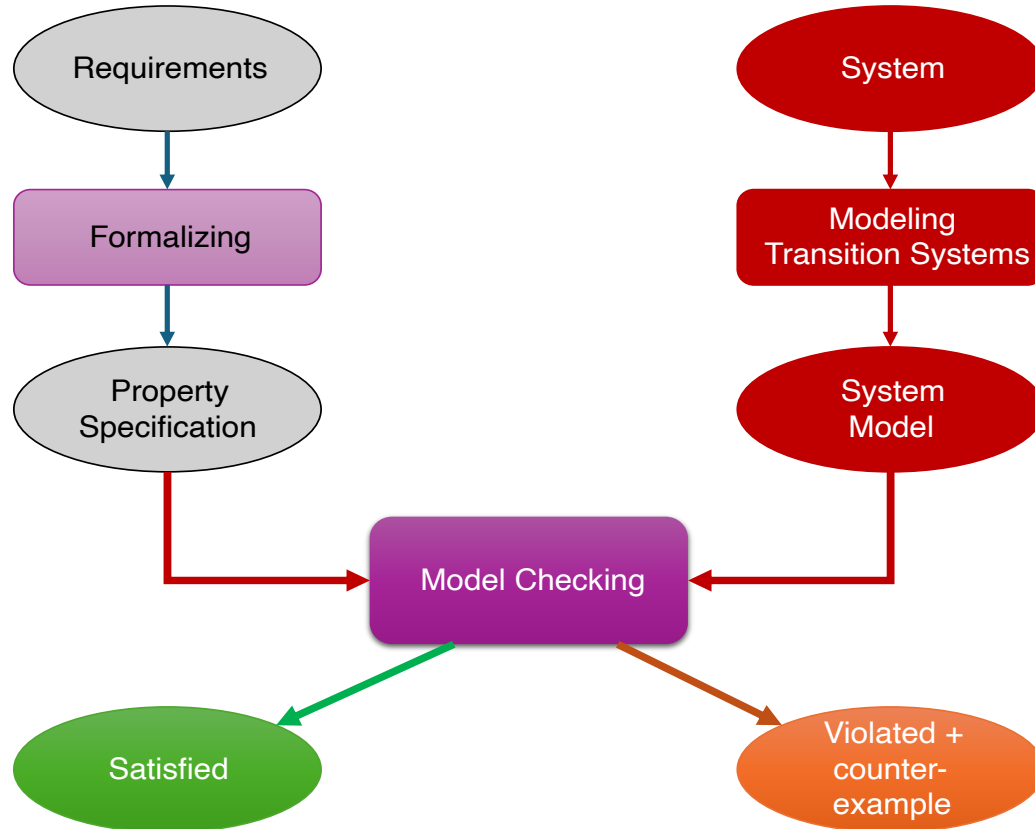


# PLAN

- La nécessité des méthodes formelles/vérification
- La vérification d'un programme
- Le principe du Model-Checking
- La modélisation d'un logiciel/système

[Retour au plan](#) - [Retour à l'accueil](#)

# PRINCIPE DU MODEL-CHECKING



# SYSTÈMES DE TRANSITION

- **modèle** pour décrire le **comportement des systèmes**
- **digraphes** où les **nœuds** représentent les **états** et les **arêtes** représentent les **transitions**
- **états** :
  - la couleur actuelle d'un feu de circulation : rouge, vert, orange.
  - **software** : les valeurs actuelles de toutes les variables du programme...
  - **hardware** : la valeur actuelle des registres ainsi que les valeurs des bits d'entrée
- **transitions** : ("changement d'états")
  - un passage d'une couleur à une autre
  - **software** : l'exécution d'une instruction de programme
  - **hardware** : le changement des registres et des bits de sortie pour une nouvelle entrée

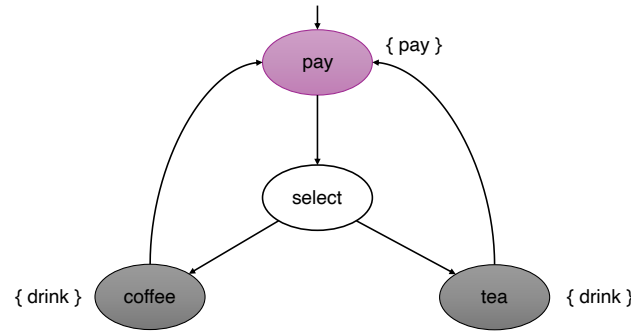
# SYSTÈMES DE TRANSITION

## LA DÉFINITION FORMELLE

- Un **systèmes de transition**  $TS$  est un tuple  $(S, \delta, I, AP, \mathcal{L})$  avec :
  - $S$  est un ensemble d'**états**
  - $\delta \subseteq S \times S$  est une **relation de transition**  
Notation:  $s \rightarrow s'$  au lieu de  $(s, s') \in \delta$
  - $I \subseteq S$  est un ensemble d'**états initiaux**
  - $AP$  est un ensemble de **propositions atomiques**
  - $\mathcal{L} : S \longrightarrow 2^{AP}$  est une **fonction d'étiquetage**

# SYSTÈMES DE TRANSITION

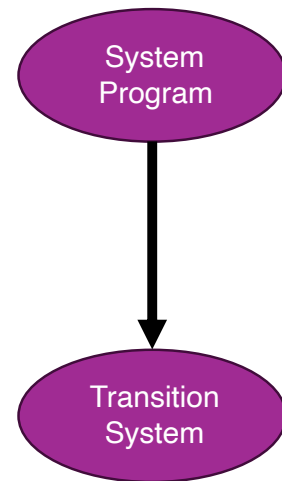
## EXAMPLE



- **Ensemble d'états :**  $S = \{\text{pay}, \text{select}, \text{tea}, \text{coffee}\}$
- **Les états initiaux :**  $I = \{\text{pay}\}$
- **Les propositions atomiques, La fonction d'étiquetage :**
  - cas 1:  $AP = S$ ,  $\mathcal{L}(s) = \{s\}$
  - cas 2:  $AP = \{\text{pay}, \text{drink}\}$ ,  $\mathcal{L}(\text{tea}) = \mathcal{L}(\text{coffee}) = \{\text{drink}\}$   
 $\mathcal{L}(\text{pay}) = \{\text{pay}\}$ ,  $\mathcal{L}(\text{select}) = \emptyset$

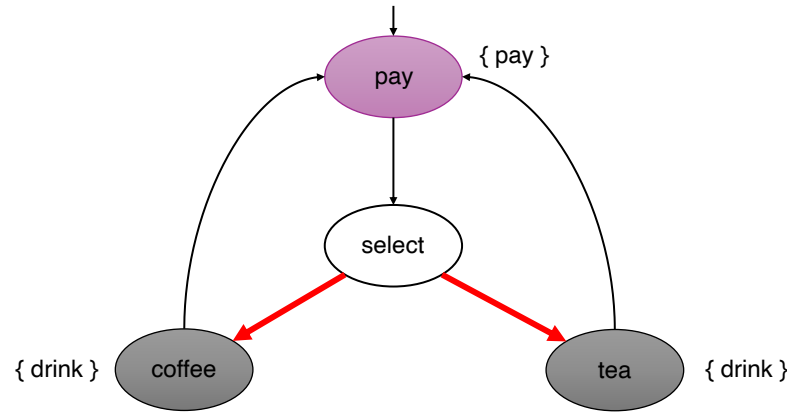
# DES LANGAGES DE PROGRAMMATION AUX SYSTÈMES DE TRANSITION

- Les systèmes de transition sont un langage de modélisation **élémentaire**
  - décrit **tous** les états que le système peut atteindre
  - décrit le comportement du système (transitions)
- Même un système de base peut avoir des milliers d'états !
  - `int i=0; while(i<1000) i++;`
  - la modélisation peut être fastidieuse !
- Et si le système de transition était **généré automatiquement** à partir du programme du système ?
  - la modélisation serait automatique !
  - de nombreux outils existent, de **C**, **Java** ... à **TS**

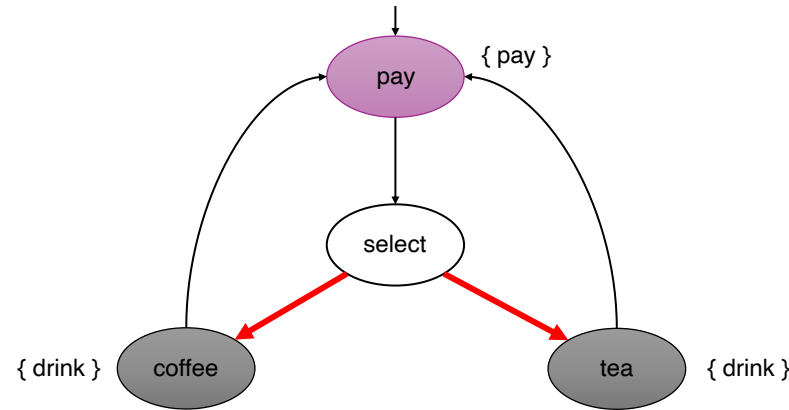


# DÉTERMINISME ET NON-DÉTERMINISME

- Soit  $TS = (S, \delta, I, AP, \mathcal{L})$  un système de transition,  $TS$  est **déterministe**
  - iff  $\forall s, s'_1, s'_2 \in S, \quad s \rightarrow s'_1, s \rightarrow s'_2 \in \delta \Rightarrow s'_1 = s'_2$
  - iff  $\forall s \in S, \quad \#(\delta(s)) \leq 1$



# SOURCES DU NON-DÉTERMINISME



- Informations incomplètes sur l'environnement du système
  - Sélection de l'utilisateur
  - Événements déclenchés

# ENTRELACEMENT DE SYSTÈMES CONCURRENTS

- le système est composé de nombreux composants concurrents
- **un système de transition** pour modéliser le comportement d'**un composant**
- ex. le threading, les algorithmes distribués et les protocoles de communication

# PRINCIPE D'ENTRELACEMENT

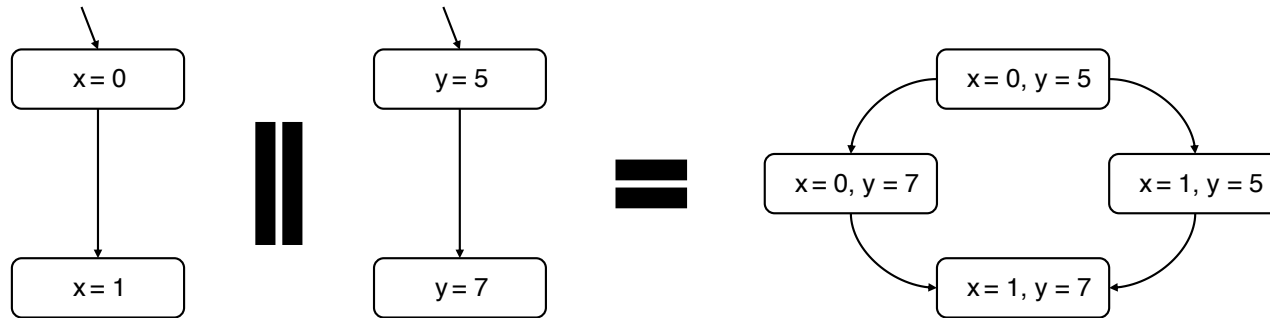
- Les actions des composants indépendants sont **entrelacées**
  - un seul processeur est disponible
  - sur lequel chaque composant s'exécute pendant un quantum de temps
- Aucune hypothèse n'est faite sur l'ordre des exécutions
  - ordres possibles pour des composants indépendants infinis  
 $Loop(P) \parallel Loop(Q)$ :

$P \ Q \ P \ Q \ P \ Q \ P \ Q \ P \ Q \dots$   
 $P \ P \ Q \ P \ P \ Q \ P \ P \ Q \ P \dots$   
 $P \ Q \ P \ P \ Q \ P \ P \ P \ Q \ P \dots$

source principale de **non-déterminisme** qui peut être évitée en ajoutant un **ordonnanceur** avec une stratégie particulière

# PRINCIPE D'ENTRELACEMENT

## EXEMPLE D'ENTRELACEMENT



# L'ENTRELACEMENT $TS_1 \parallel TS_2$

## DÉFINITION FORMELLE

Soient  $TS_i = (S_i, \delta_i, I_i, AP_i, \mathcal{L}_i), i = 1, 2$  deux systèmes de transition.

Le **produit entrelacé** (**produit asynchrone**) est le système de transition :

$$TS_1 \parallel TS_2 = (S_1 \times S_2, \delta, I_1 \times I_2, AP_1 \cup AP_2, \mathcal{L})$$

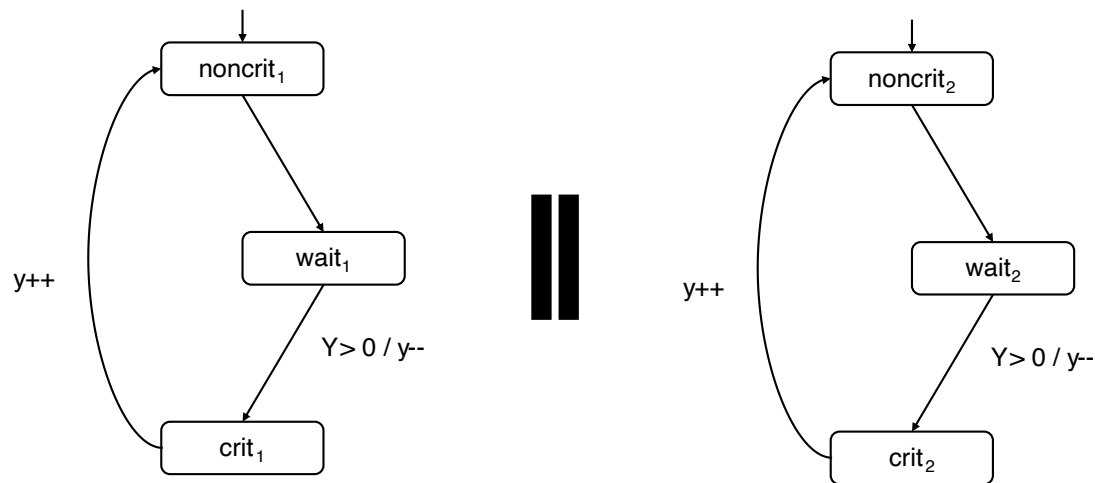
où  $\delta$  vérifie :

$$\frac{s_1 \longrightarrow s'_1}{\langle s_1, s_2 \rangle \longrightarrow \langle s'_1, s_2 \rangle} \quad and \quad \frac{s_2 \longrightarrow s'_2}{\langle s_1, s_2 \rangle \longrightarrow \langle s_1, s'_2 \rangle}$$

et  $\mathcal{L}$  vérifie :

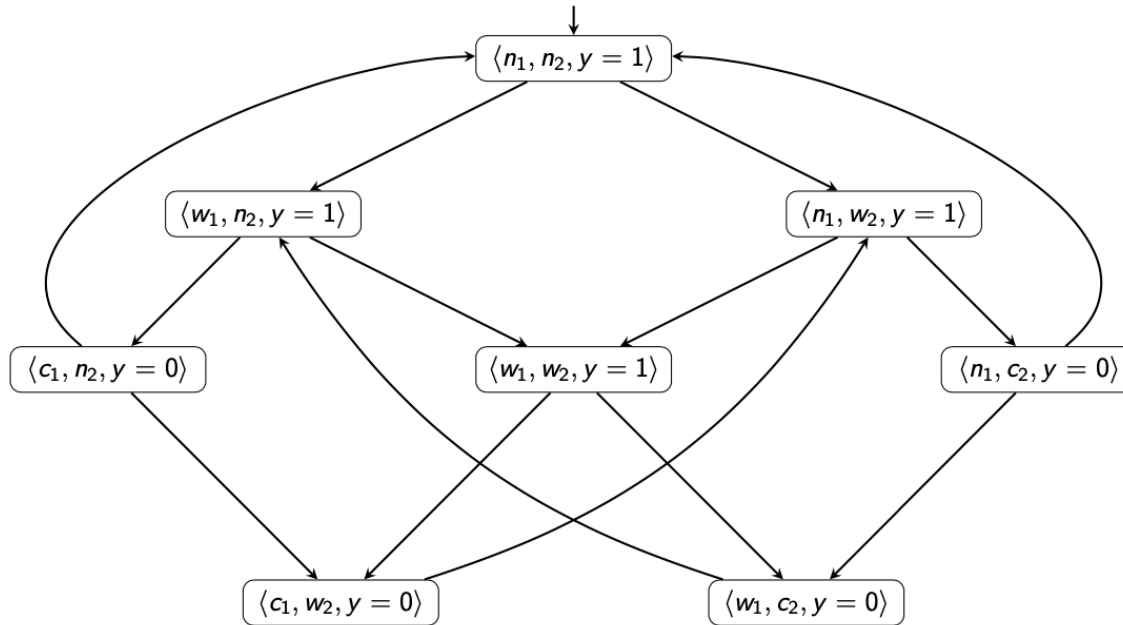
$$\mathcal{L}(\langle s_1, s_2 \rangle) = \mathcal{L}_1(s_1) \cup \mathcal{L}_2(s_2)$$

# EXCLUSION MUTUELLE BASÉE SUR LES SÉMAPHORES



$y = 0$  signifie "le verrou est actuellement possédé";  
 $y = 1$  signifie "le verrou est libre"

# PRODUIT D'ENTRELACEMENT



$n_i : noncrit_i, w_i : wait_i, c_i : crit_i$

**Source typique d'explosion d'état** → supposons qu'il y ait 3 composants concurrents

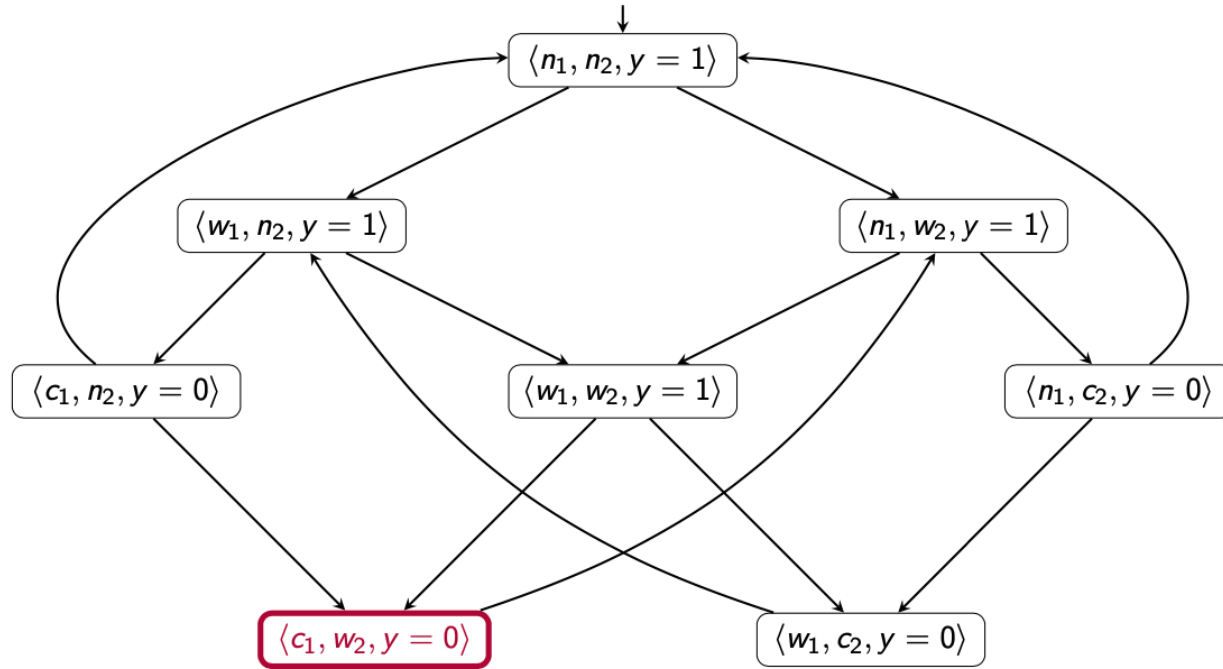
# CHEMINS ET ÉTATS ATTEIGNABLES

- Un **fragment de chemin infini**  $\pi$  est une séquence d'états infinie :  
 $\pi = s_0 s_1 s_2 \dots$  tel que  $\forall i \geq 0, s_i \longrightarrow s_{i+1} \in \delta$
- $Paths(s)$  est l'ensemble des fragments de chemin infinis  $\pi$  avec  $first(\pi) = s$
- $Paths(TS) = \bigcup_{s \in I} Paths(s)$  est l'ensemble des fragments de chemin initiaux
- Un état  $s \in S$  est dit **accessible** dans  $TS$  s'il existe un fragment de chemin initial  $\pi$  tel que

$$\pi = s_0 s_1 \dots s_{n-1} (s_n = s) s_{n+1} \dots \in Paths(TS)$$

- $Reach(TS)$  désigne l'ensemble de tous les états atteignables dans  $TS$

# REVENONS À NOTRE EXEMPLE



$Paths(\langle c_1, w_2, y = 0 \rangle) = ?$

$Paths(TS) = ?$

$Reach(TS) = ?$

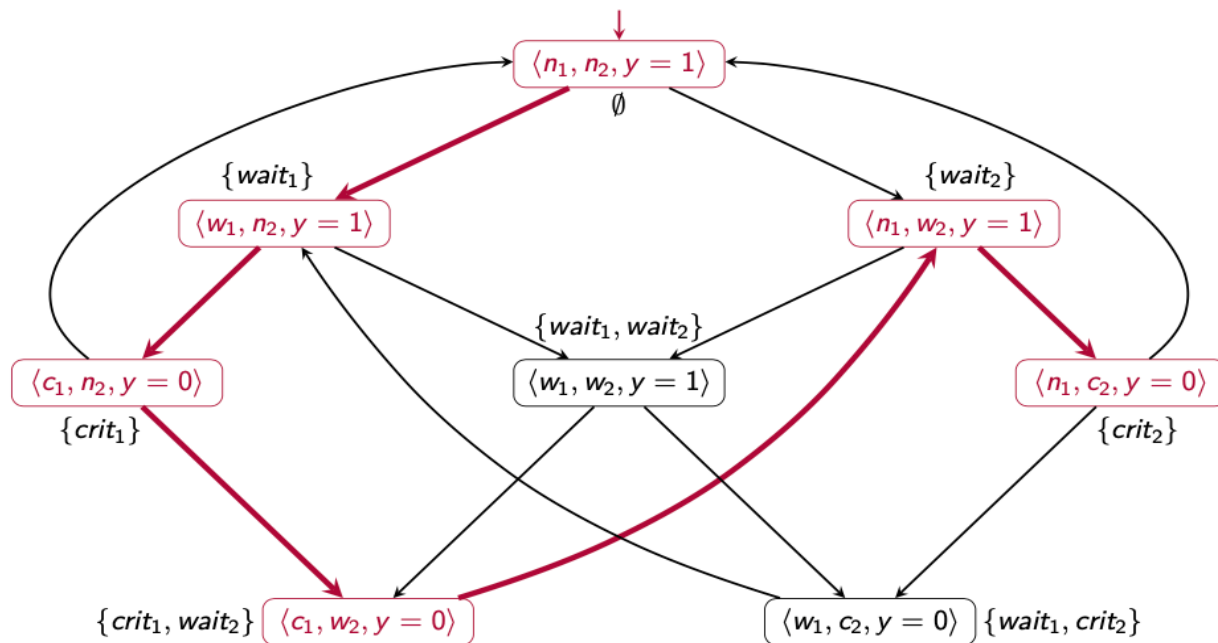
# TRACES

- Les états sont observables à travers leurs **propositions atomiques**
- Les **traces** se concentrent uniquement sur l'ensemble des **propositions atomiques qui sont valides** le long du chemin d'exécution
- La trace du chemin  $\pi = s_0 s_1 s_2 \dots \in S^\omega$  avec  $\mathcal{L} : S \longrightarrow 2^{AP}$ 
  - $trace(\pi) = \mathcal{L}(s_0)\mathcal{L}(s_1)\mathcal{L}(s_2)\dots \in (2^{AP})^\omega$
- Les traces sont des mots **infinis** sur l'alphabet  $2^{AP}$
- $trace(\Pi) = \{trace(\pi) | \pi \in \Pi\}, \quad Traces(s) = trace(Paths(s))$

$$\text{et } Traces(TS) = \bigcup_{s \in I} Traces(s)$$

# REVENONS À NOTRE EXEMPLE

Soit  $AP = \{wait_1, crit_1, wait_2, crit_2\}$



$Trace(\pi \dots) = \emptyset \{wait_1\} \{crit_1\} \{crit_1, wait_2\} \{wait_2\} \{crit_2\} \dots$

# MERCI

[Version PDF des slides](#)

[Retour à l'accueil](#) - [Retour au plan](#)