

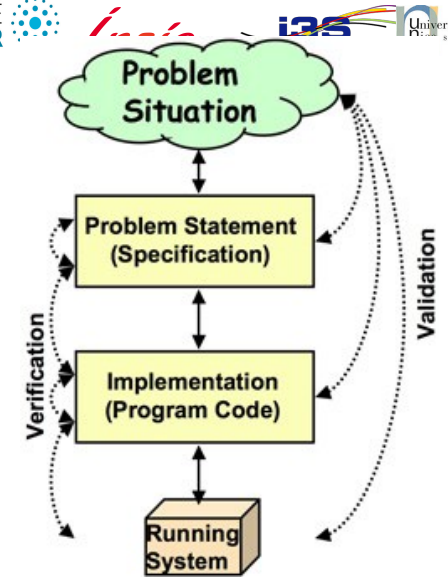
# Finite State Machine, state charts

V&V

# V&V ?

- Disclaimer:
  - on ne verra ici qu'une introduction aux notions et problèmes de V&V. Beaucoup de raccourcis sont fait mais cela devrait être suffisant pour vous donner l'intuition derrière ces notions et vous permettre de les approfondir par vous même si besoin.

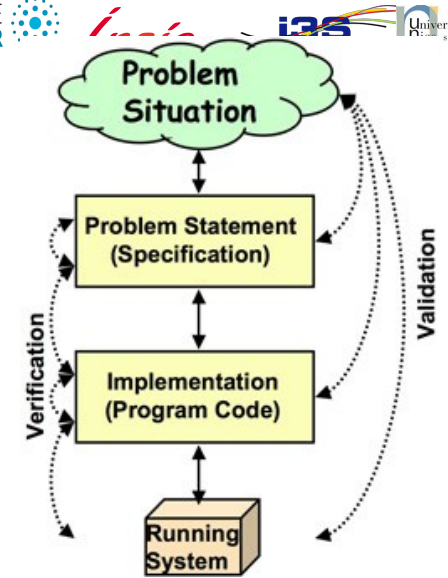
# V&V ?



- Verification and Validation

- Verification: Construisons-nous le système correctement ?
  - Est-ce que le système est implémenté de manière correcte ? (sans erreur, avec les bonnes performances, sans fuites mémoires, raffinement correct, etc)
- Validation:

# V&V ?



- Verification and Validation

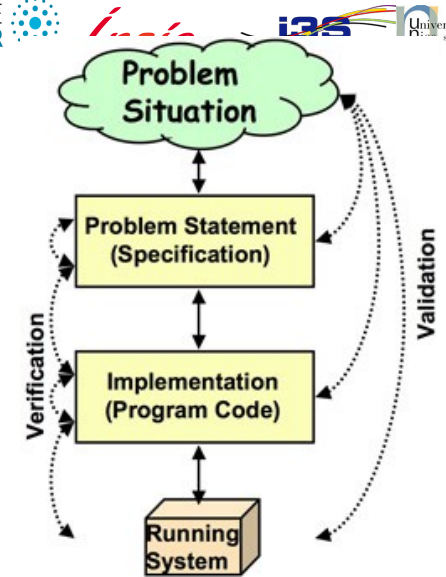
- Verification: Construisons-nous le système correctement ?
  - Est-ce que le système est implémenté de manière correcte ? (sans erreur, avec les bonnes performances, sans fuites mémoires, raffinement correct, etc)
- Validation: Construisons-nous le bon système ?
  - Est-ce que le système que nous développons est celui que le client voulait / qui répond au problème initial ?

# V&V ?

- **Verification:**
  - “Are we building the product right?”
- **Validation:**
  - “Are we building the right product?”

- Verification and Validation

- Verification: Construisons-nous le système correctement ?
  - Est-ce que le système est implémenté de manière correcte ? (sans erreur, avec les bonnes performances, sans fuites mémoires, raffinement correct, etc)
- Validation: Construisons-nous le bon système ?
  - Est-ce que le système que nous développons est celui que le client voulait / qui répond au problème initial ?



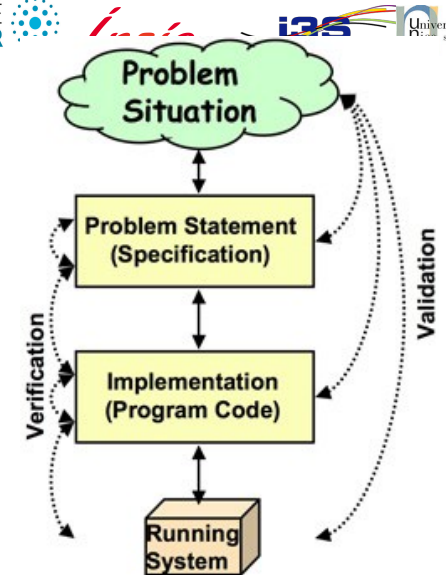
# V&V ?

- **Verification:**
  - “Are we building the product right?”
- **Validation:**
  - “Are we building the right product?”

- Verification and Validation

- Verification: Construisons-nous le système correctement ?
  - Est-ce que le système est implémenté de manière correcte ? (sans erreur, avec les bonnes performances, sans fuites mémoires, raffinement correct, etc)
- Validation: Construisons-nous le bon système ?
  - Est-ce que le système que nous développons est celui que le client voulait / qui répond au problème initial ?

→ dans les deux cas on se demande si le système est prêt à être publié...



# V&V ?

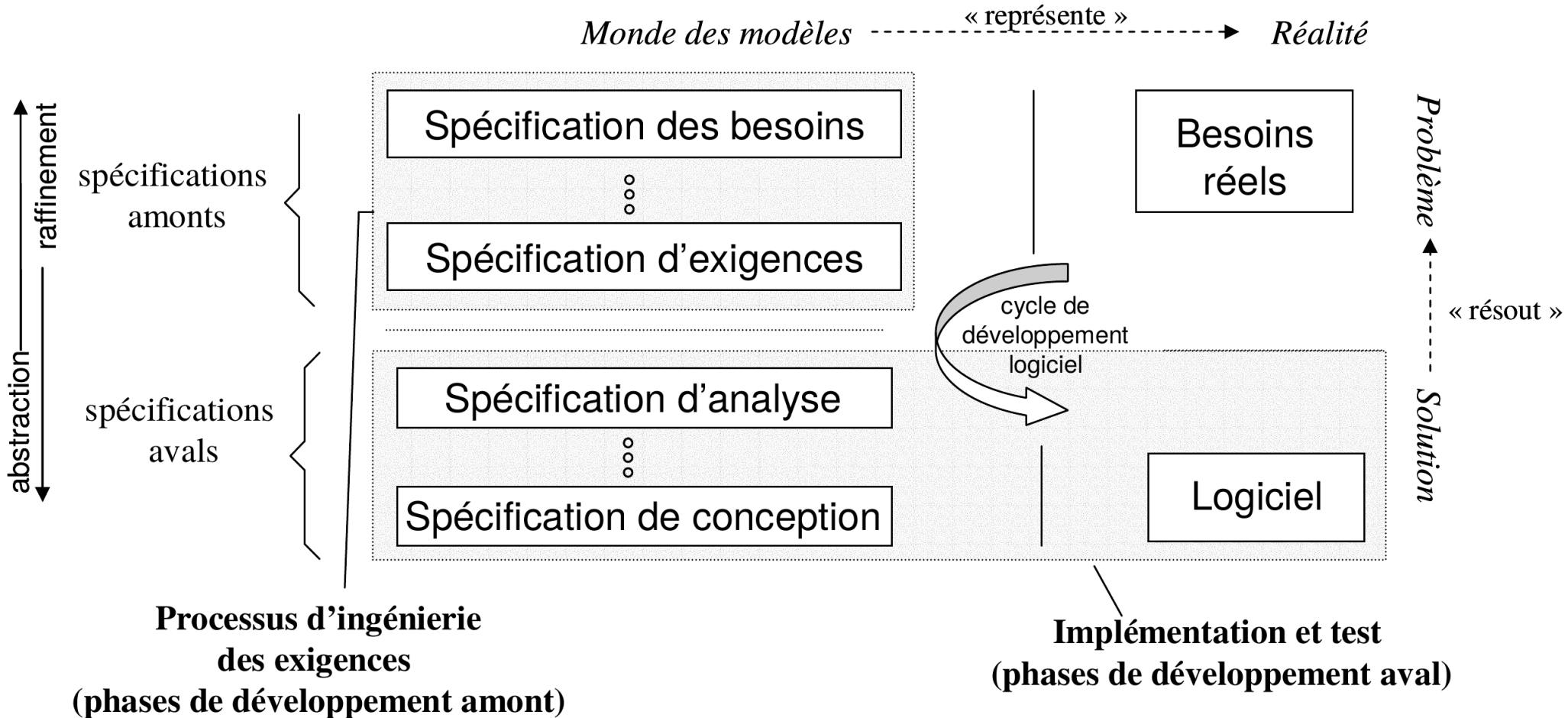
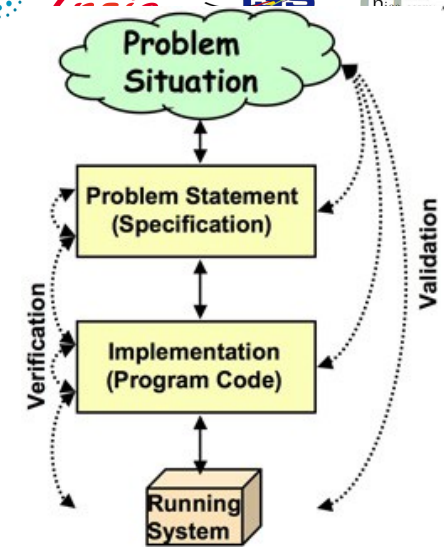


Figure 2 – Le processus d'ingénierie des exigences au sein du processus de développement logiciel.

Figure taken from Erwan Brottier's phd thesis: <https://tel.archives-ouvertes.fr/tel-00512174/file/Brottier09.pdf>

# V&V ?



- Verification and Validation

- Verification: Construisons-nous le système correctement ?

- Est-ce que le système est implémenté de manière correcte ? (sans erreur, avec les bonnes performances, sans fuites mémoires, raffinement correct, etc)

- Validation: Construisons-nous le bon système ?

- Est-ce que le système que nous développons est celui que le client voulait / qui répond au problème initial ?

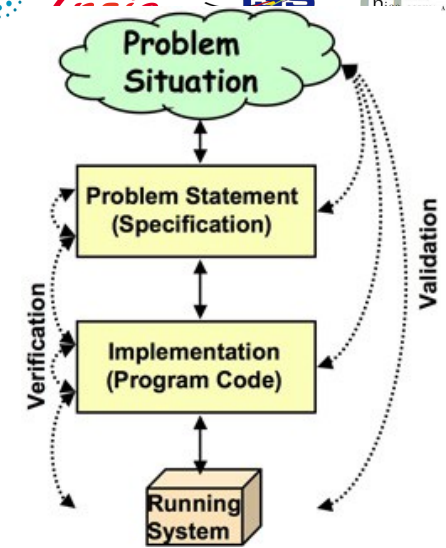
→ Dans les deux cas on **“pose des questions au système”** :

- Verification :

- Validation :



# V&V ?



- Verification and Validation

- Verification: Construisons-nous le système correctement ?

- Est-ce que le système est implémenté de manière correcte ? (sans erreur, avec les bonnes performances, sans fuites mémoires, raffinement correct, etc)

- Validation: Construisons-nous le bon système ?

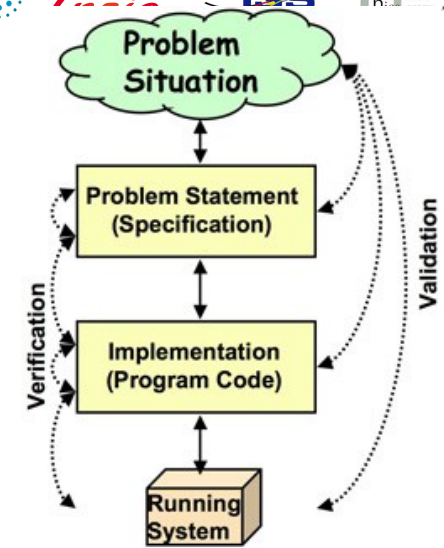
- Est-ce que le système que nous développons est celui que le client voulait / qui répond au problème initial ?

→ Dans les deux cas on **“pose des questions au système”** :

- Verification : en lien avec la manière dont le système est implémenté. Est-ce qu'à chaque réservation mémoire il existe une libération de la mémoire ?

- Validation :

# V&V ?



- Verification and Validation

- Verification: Construisons-nous le système correctement ?

- Est-ce que le système est implémenté de manière correcte ? (sans erreur, avec les bonnes performances, sans fuites mémoires, raffinement correct, etc)

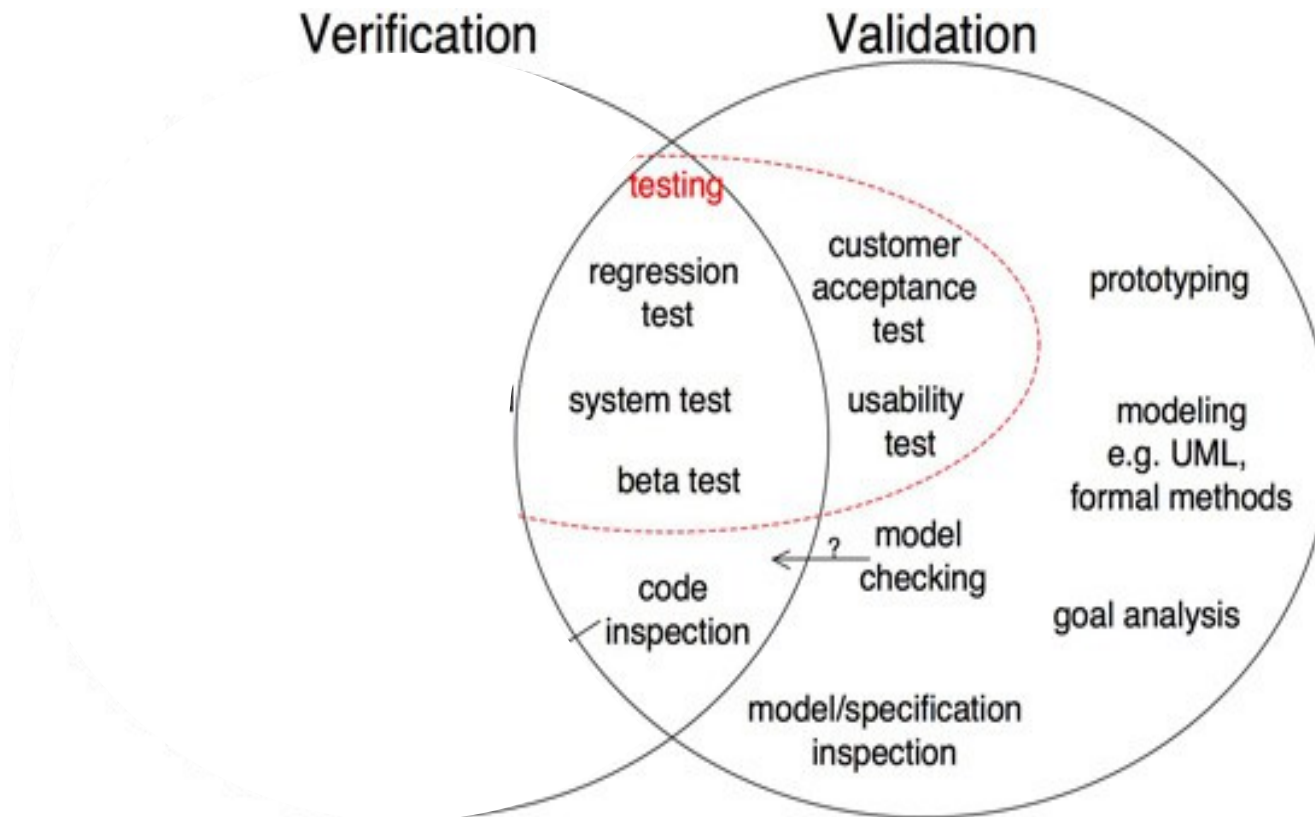
- Validation: Construisons-nous le bon système ?

- Est-ce que le système que nous développons est celui que le client voulait / qui répond au problème initial ?

→ Dans les deux cas on **“pose des questions au système”** :

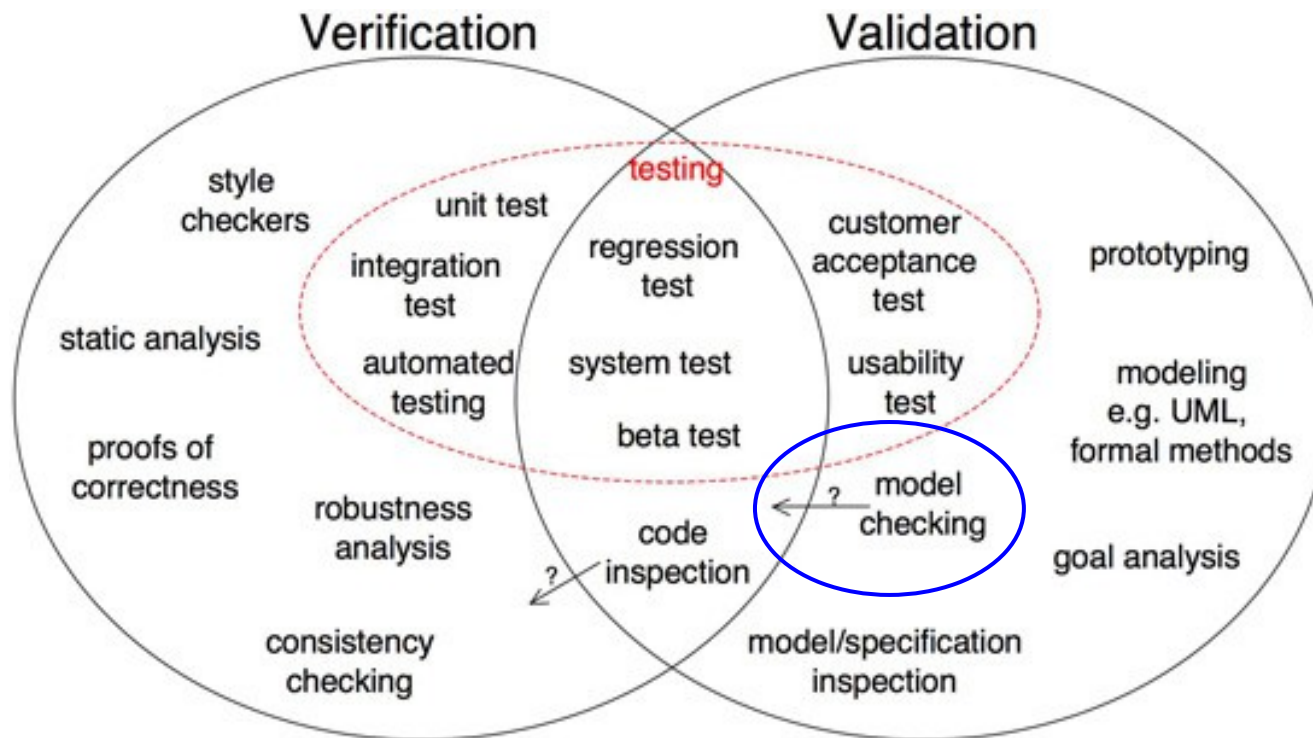
- Verification : en lien avec la manière dont le système est implémenté. Est-ce qu'à chaque réservation mémoire il existe une libération de la mémoire ?
    - Validation : en lien avec les exigences du client : est-il vrai que *à chaque instant il doit être possible d'arrêter le système en moins de 2 secondes.*

# Techniques de V&V



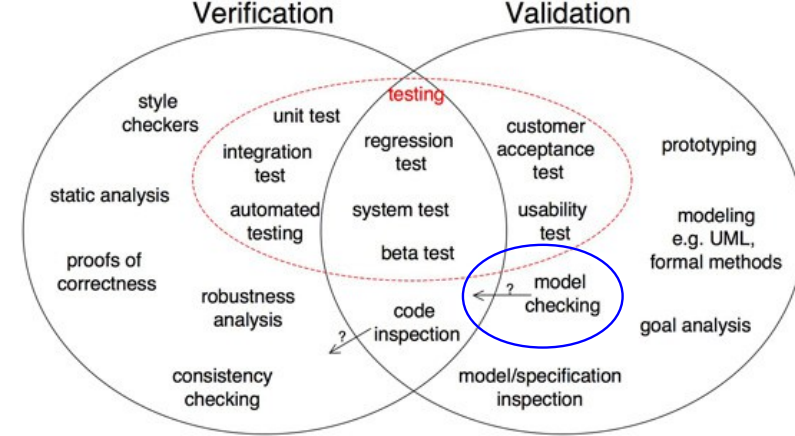
Picture taken from Steve EasterBrook  
 (<http://www.easterbrook.ca/steve/2010/11/the-difference-between-verification-and-validation/>)

# Techniques de V&V



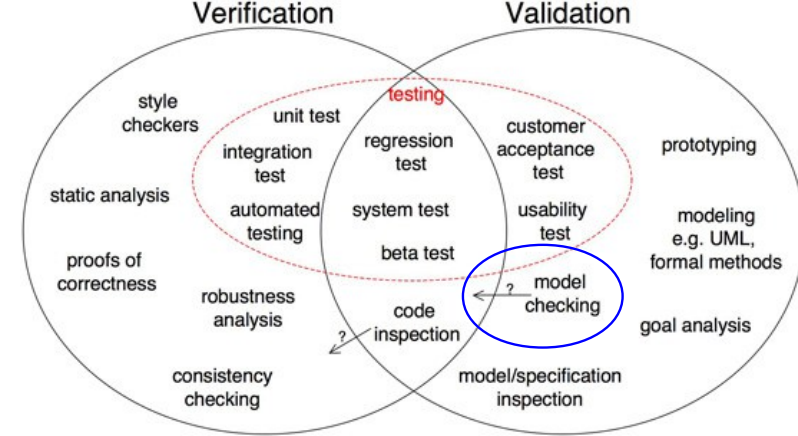
Picture taken from Steve EasterBrook  
 (<http://www.easterbrook.ca/steve/2010/11/the-difference-between-verification-and-validation/>)

# Techniques de V&V



- Testing or model checking ? (intuition)
  - testing: regarder si certains chemins d'exécutions donnent le résultat attendu. On pose autant de questions que nécessaires pour vérifier une propriété du système.
    - taux de couverture ? Nombre de tests ?

# Techniques de V&V

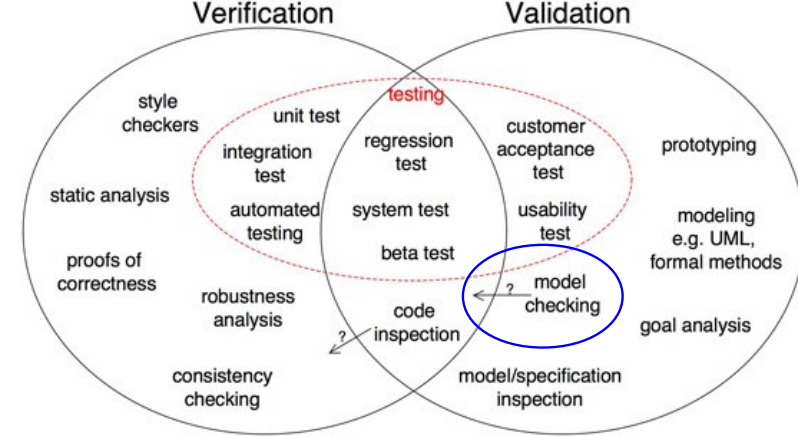


- Testing or model checking ? (intuition)
  - testing: regarder si certains chemins d'exécutions donnent le résultat attendu. On pose autant de questions que nécessaires pour vérifier une propriété du système.

→ taux de couverture ? Nombre de tests ?

```
bool foo(unsigned int i)
{
    bool res = false;
    if (i > 30){
        res = true;
    }
    if (i < 40){
        res = res ^ true;
    }
    return res;
}
```

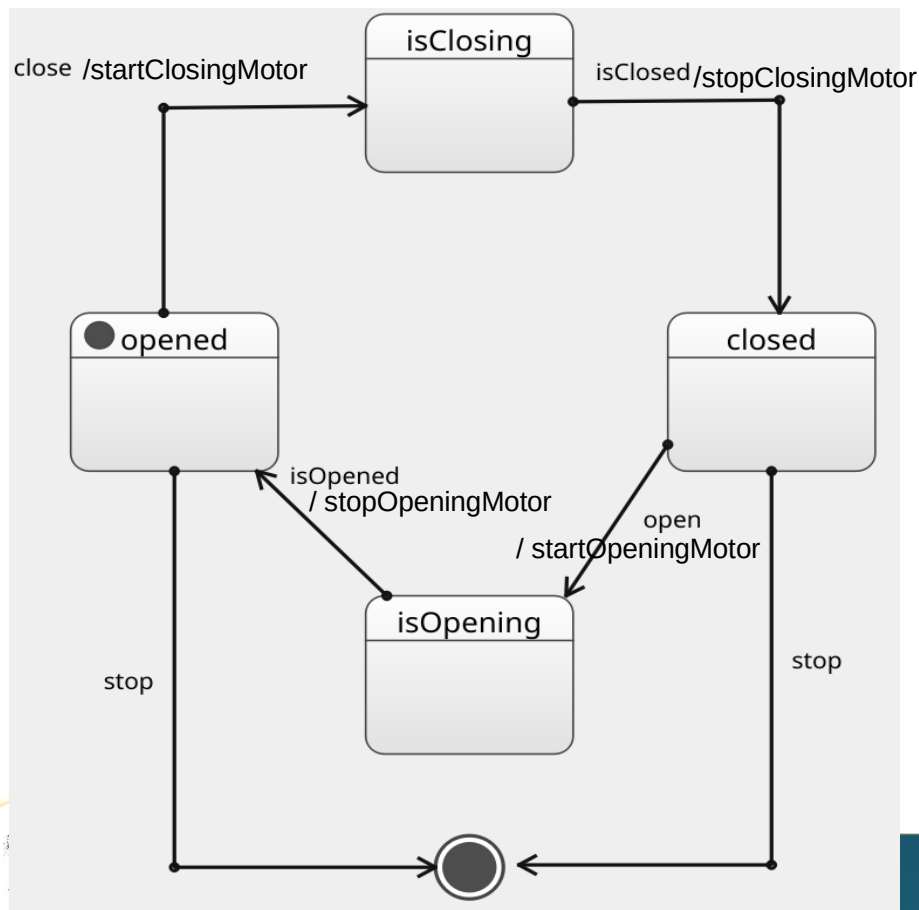
# Techniques de V&V



- Testing or model checking ? (intuition)
  - testing: regarder si certains chemins d'exécutions donnent le résultat attendu. On pose autant de questions que nécessaires pour vérifier une propriété du système.
    - taux de couverture ? Nombre de tests ?
  - model checking: regarder si **tous les chemins d'exécution** donnent le résultat attendu. On exprime une propriété sous la forme d'une expression
    - ensemble de chemins d'exécutions finis ? Quel type de propriétés ?

# trace (run), state space and real life

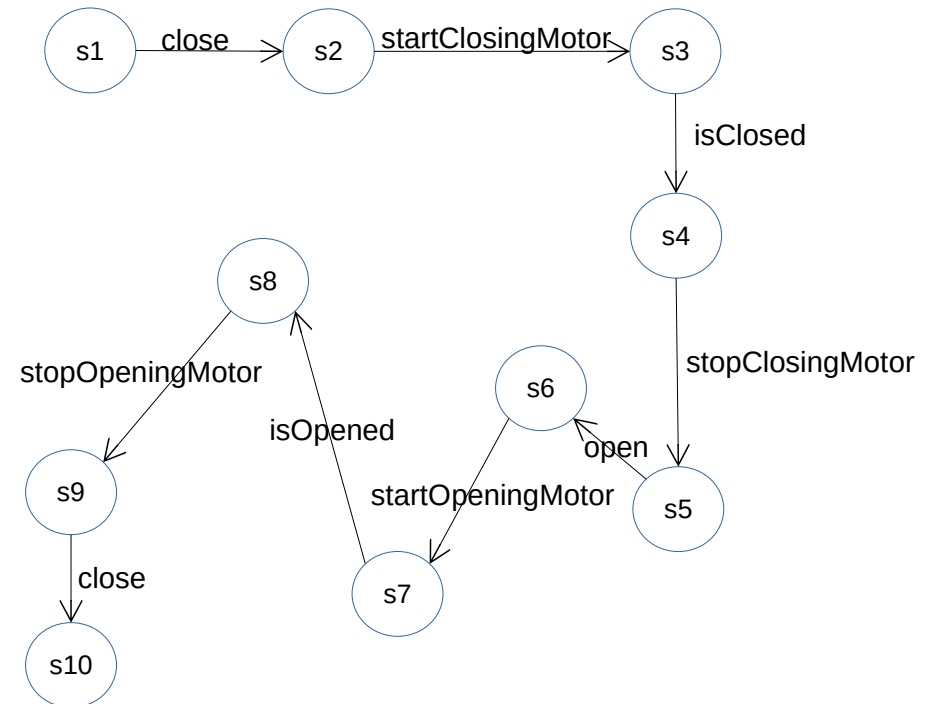
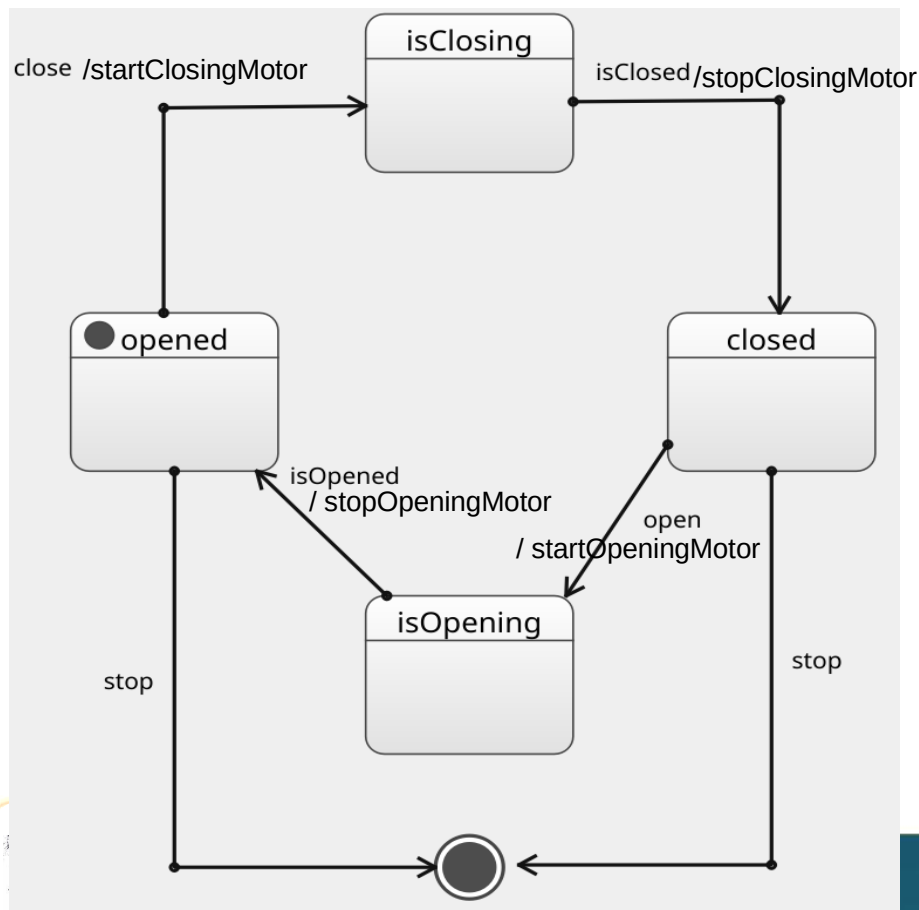
- Each test or run of a program creates a trace.
- Speaking automata, a trace is a, possibly infinite, sequence of states and transitions. **It is a word accepted by the automata.**





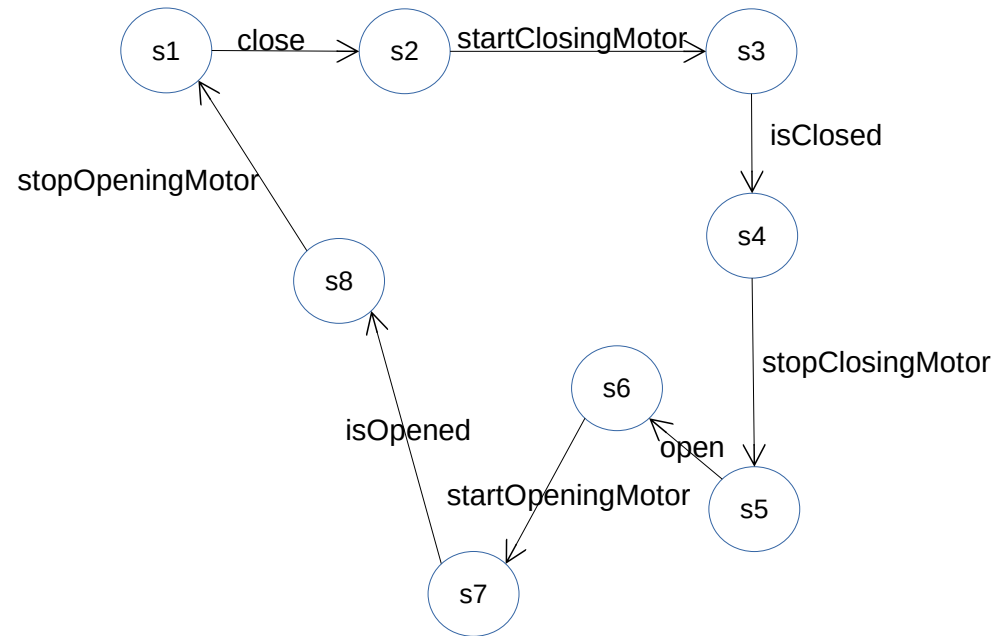
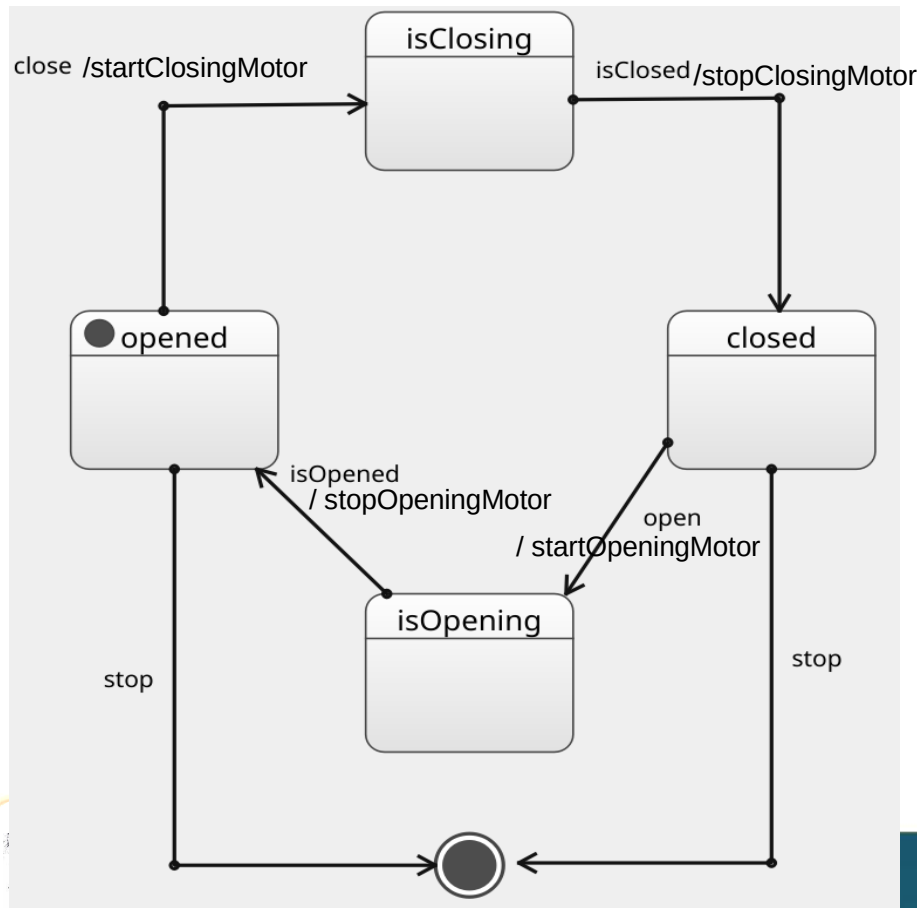
# trace (run), state space and real life

- Each test or run of a program creates a trace.
- Speaking automata, a trace is a, possibly infinite, sequence of states and transitions. It is a word accepted by the automata.



# trace (run), state space and real life

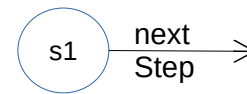
- Each test or run of a program creates a trace.
- Speaking automata, a trace is a, possibly infinite, sequence of states and transitions. It is a word accepted by the automata. **It is actually a bit more since there is possibly information in states.**



# trace (run), state space and real life

- Each test or run of a program creates a trace.
- Speaking automata, a trace is a, possibly infinite, sequence of states and transitions. It is a word accepted by the automata.
- Speaking code, a trace is a, possibly infinite, sequence of states and transitions :)

```
1 #include <stdio.h>
2 #include <assert.h>
3
4 void foo( int *array ) {
5     for ( int i = 0; i <= 4; ++i ) {
6         printf( "writing at index %d\n", i );
7         array[i] = 42;
8     }
9 }
10
11 int main() {
12     int x[4];
13     foo( x );
14     assert( x[3] == 42 );
15 }
```



# trace (run), state space and real life

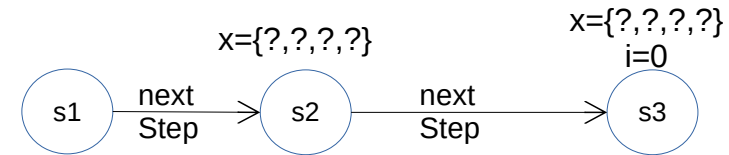
- Each test or run of a program creates a trace.
- Speaking automata, a trace is a, possibly infinite, sequence of states and transitions. It is a word accepted by the automata.
- Speaking code, a trace is a, possibly sequence of states and transitions :)

transitions :)

```

1 #include <stdio.h>
2 #include <assert.h>
3
4 void foo( int *array ) {
5     for ( int i = 0; i <= 4; ++i ) {
6         printf( "writing at index %d\n", i );
7         array[i] = 42;
8     }
9 }
10
11 int main() {
12     int x[4];
13     foo( x );
14     assert( x[3] == 42 );
15 }

```



# trace (run), state space and real life

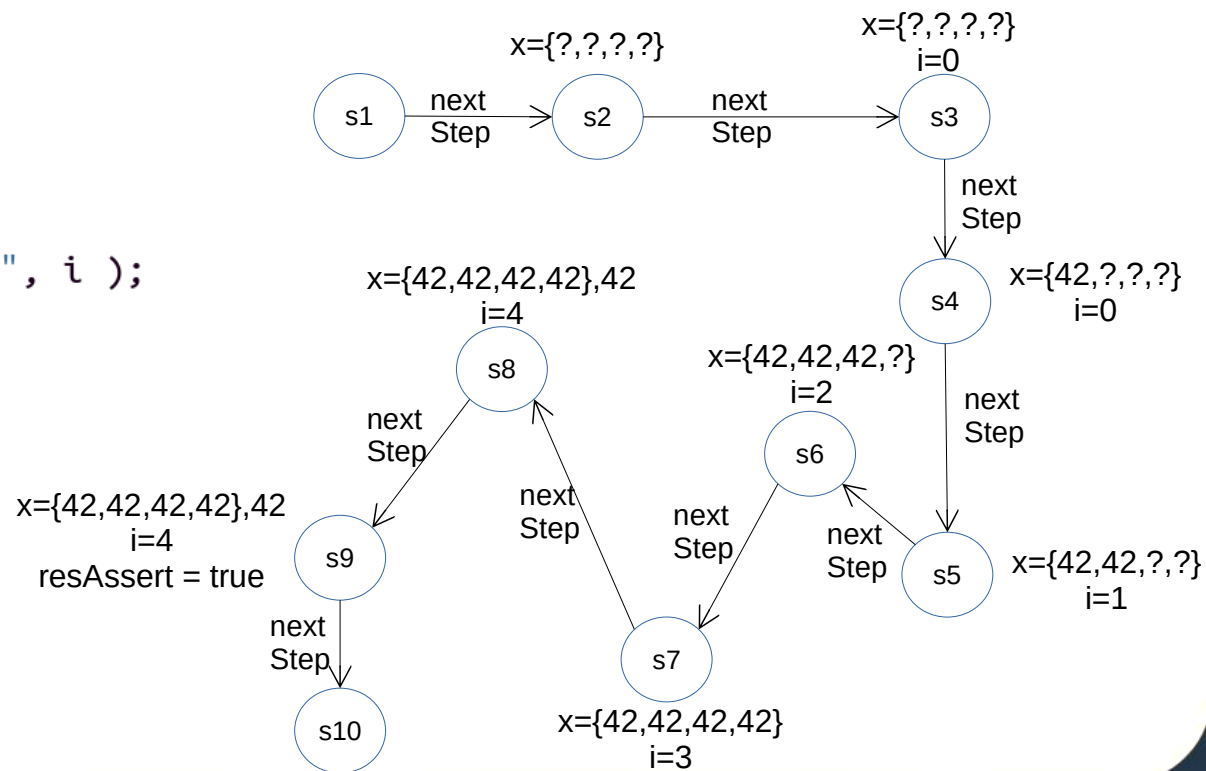
- Each test or run of a program creates a trace.
- Speaking automata, a trace is a, possibly infinite, sequence of states and transitions. It is a word accepted by the automata.
- Speaking code, a trace is a, possibly sequence of states and

transitions :)

```

1 #include <stdio.h>
2 #include <assert.h>
3
4 void foo( int *array ) {
5     for ( int i = 0; i <= 4; ++i ) {
6         printf( "writing at index %d\n", i );
7         array[i] = 42;
8     }
9 }
10
11 int main() {
12     int x[4];
13     foo( x );
14     assert( x[3] == 42 );
15 }

```



# trace (run), state space and real life

- Each test or run of a program creates a trace.
- Speaking automata, a trace is a, possibly infinite, sequence of states and transitions. It is a word accepted by the automata.
- Speaking code, a trace is a, possibly sequence of states and transitions :)

```
1 #include <stdio.h>
2 #include <assert.h>
3
4 void foo( int *array ) {
5     for ( int i = 0; i <= 4; ++i ) {
6         printf( "writing at index %d\n", i );
7         array[i] = 42;
8     }
9 }
10
11 int main() {
12     int x[4];
13     foo( x );
14     assert( x[3] == 42 );
15 }
```

```
$ divine verify program.c
```

DIVINE will now compile your program and run the verifier on the compiled code. After a short while, it will produce the following output:

```
compiling program.c
loading bitcode ... LART ... RR ... constants ... done
booting ... done
found 83 states in 0:00,
```

<http://divine.fi.muni.cz/manual.html>

# trace (run), state space and real life

- Each test or run of a program creates a trace.
- Speaking automata, a trace is a, possibly infinite, sequence of states and transitions. It is a word accepted by the automata.
- Speaking code, a trace is a, possibly sequence of states and transitions :)

```

1 #include <stdio.h>
2 #include <assert.h>
3
4 void foo( int *array ) {
5     for ( int i = 0; i <= 4; ++i ) {
6         printf( "writing at index %d\n", i );
7         array[i] = 42;
8     }
9 }
10
11 int main() {
12     int x[4];
13     foo( x );
14     assert( x[3] == 42 );
15 }

```

```
$ divine verify program.c
```

DIVINE will now compile your program and run the verifier on the compiled code. After a short while, it will produce the following output:

```

compiling program.c
loading bitcode ... LART ... RR ... constants ... done
booting ... done
found 83 states in 0:00,

```

```
error found: yes
```

```
error trace: |
```

```

[0] writing at index 0
[0] writing at index 1
[0] writing at index 2
[0] writing at index 3
[0] writing at index 4

```

```
FAULT: access of size 4 at [heap* 53e6ba2a 10 ddp] is 4 bytes out of bounds
```

```
[0] Fault in userspace: memory
```

```
[0] Backtrace:
```

```

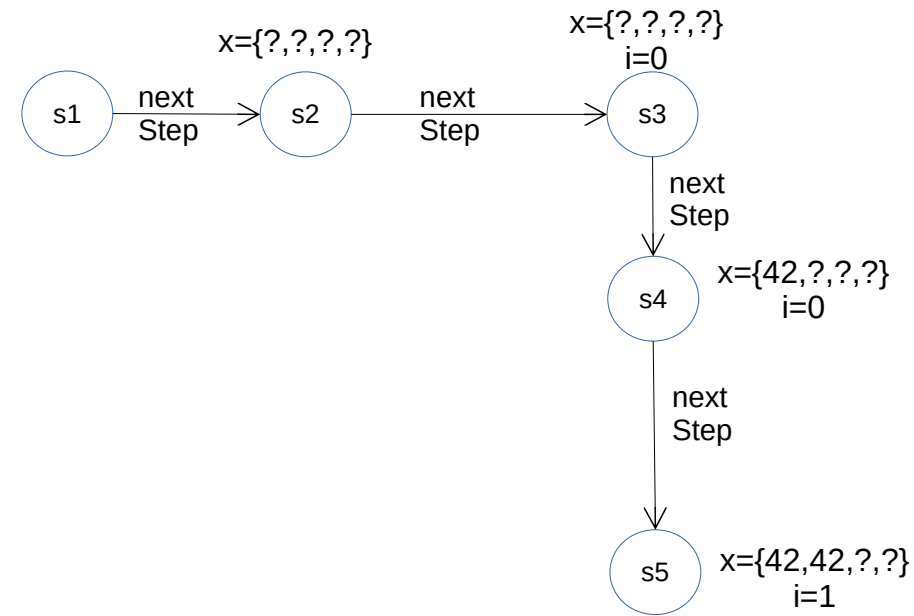
[0] 1: foo
[0] 2: main
[0] 3: _start

```

<http://divine.fi.muni.cz/manual.html>

# trace (run), state space and real life

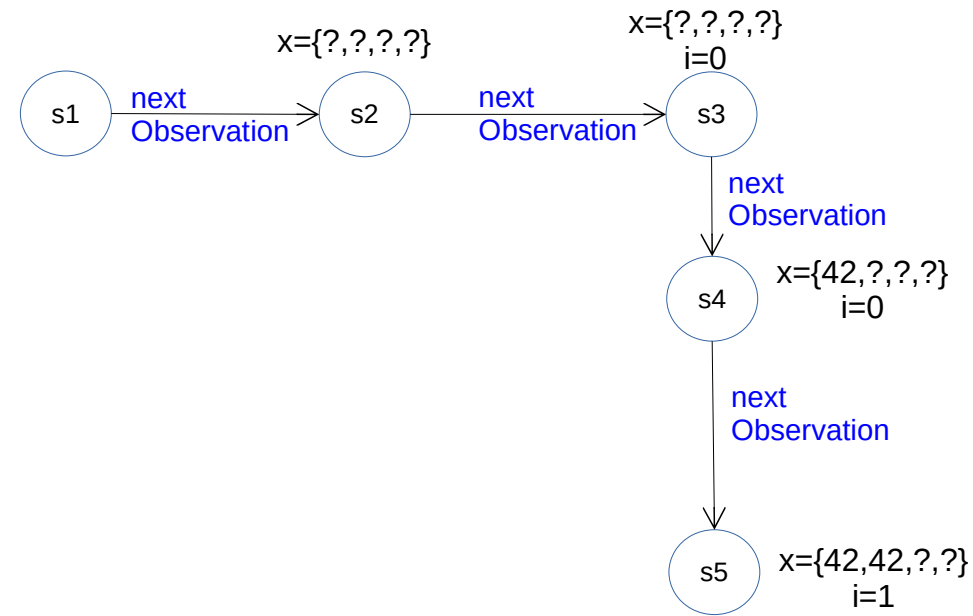
- Each test or run of a program creates a trace.
- Speaking automata, a trace is a, possibly infinite, sequence of states and transitions. It is a word accepted by the automata.
- Speaking code, a trace is a, possibly sequence of states and transitions :)





# trace (run), state space and real life

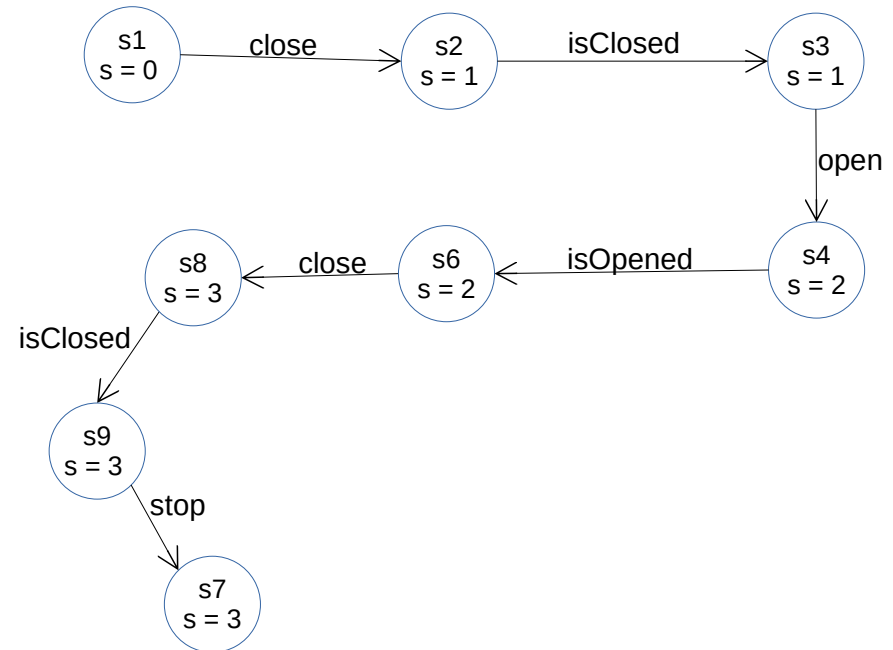
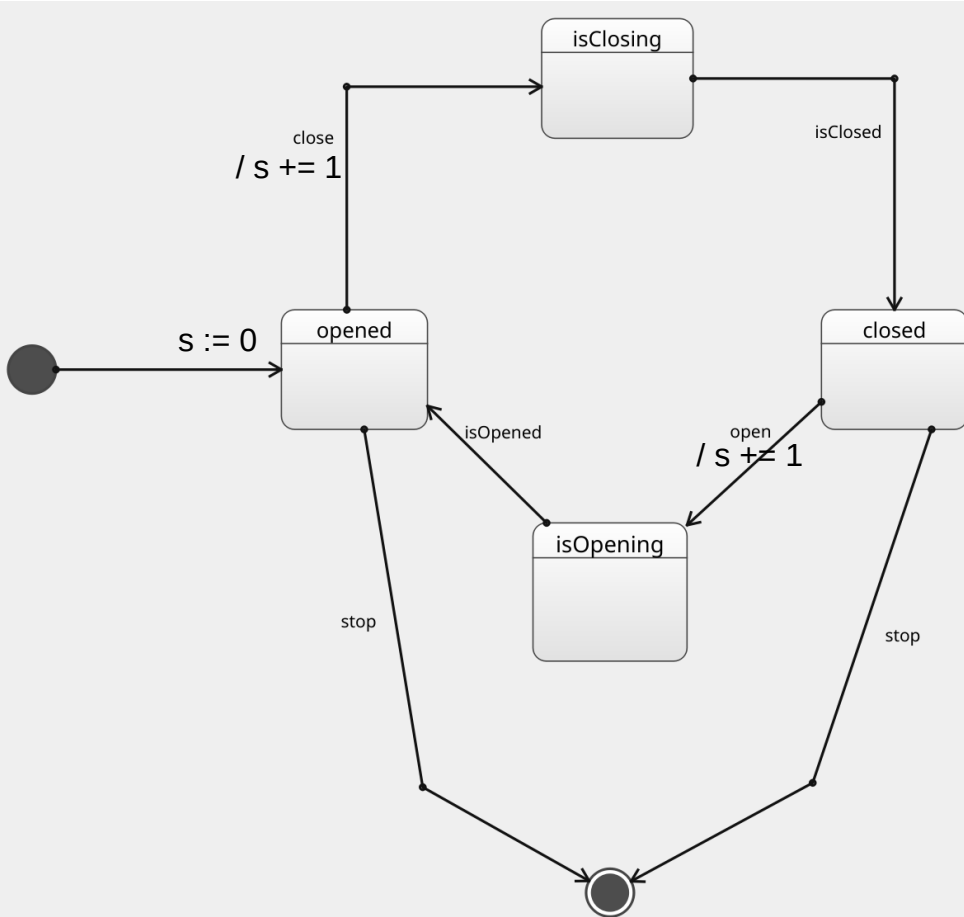
- Each test or run of a program creates a trace.
- Speaking automata, a trace is a, possibly infinite, sequence of states and transitions. It is a word accepted by the automata.
- Speaking code, a trace is a, possibly sequence of states and transitions :)



# trace (run), state space and real life

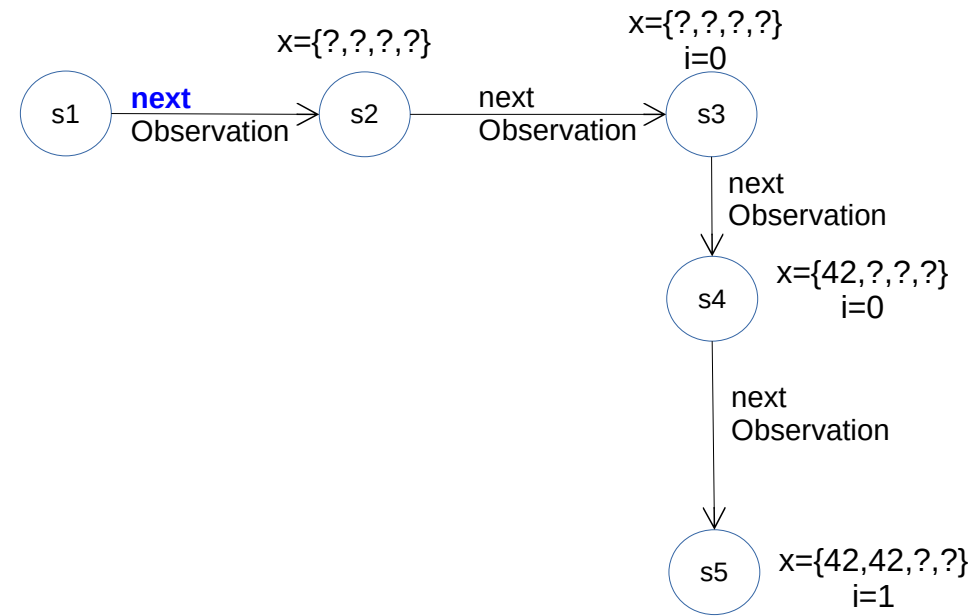
To obtain traces:

- Take your favorite programming/modeling language
- Equip it with discrete transition semantics (e.g., S.O.S)
- Determine what should be **observable events** / conditions / **execution states**



# Time

- All traces are dealing with **time**



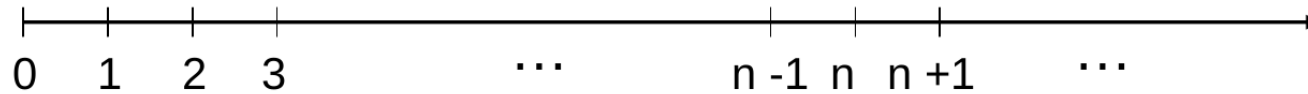
# Time

- All traces are dealing with **time**

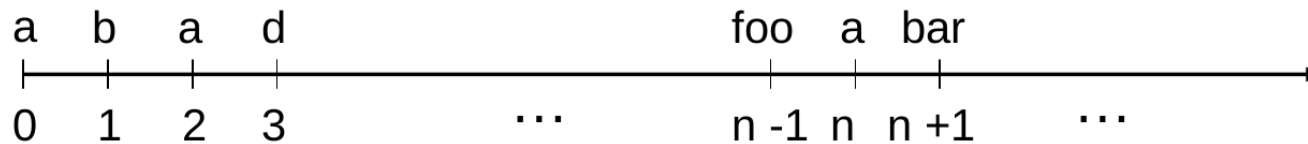
Time is discrete

Starts at 0

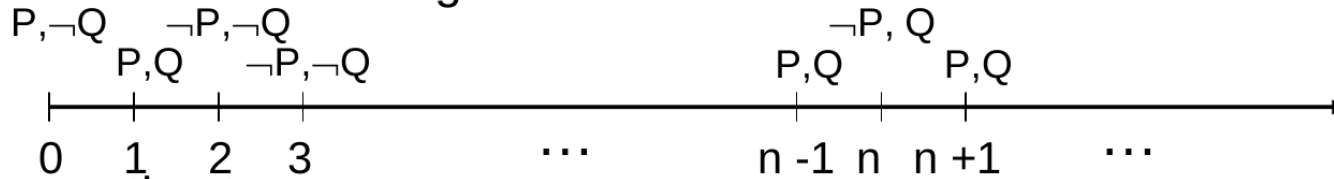
Goes on forever



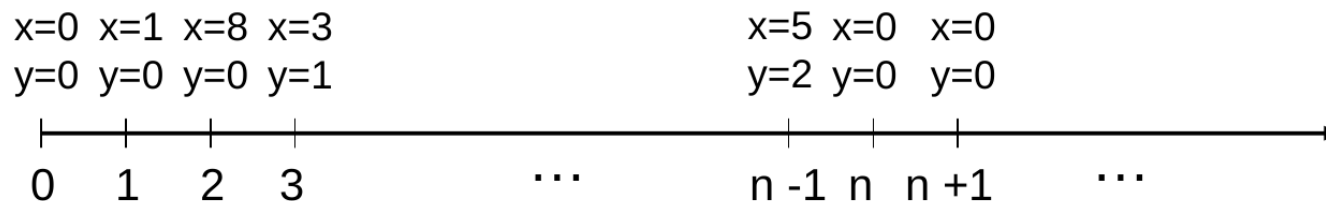
Time points decorated by events



Or conditions/truth assignments/valuations



Or execution traces



# Checking properties on traces

- Specifying properties over time.
- Two types of properties:
  - **Safety property:** asserts that nothing bad happens.
  - **Liveness property:** asserts that something good eventually happens.

# Checking properties on traces

## Définition [\[ modifier | modifier le code \]](#)

Un **Mutex** M.M.S est une [primitive de synchronisation](#) utilisée en [programmation informatique](#) pour éviter que des [ressources](#) partagées d'un système ne soient utilisées en même temps.

Certains algorithmes utilisent un *état* pour commander l'exécution : les mutex doivent savoir si les programmes concernés sont occupés (*busy*) ou s'ils ont terminé et sont en attente (*wait*). De tels algorithmes sont par exemple :

- l'[algorithme de Dekker](#) ;
- l'[algorithme de Peterson](#) ;

La plupart des mutex ont des effets secondaires, par exemple les [sémaphores](#) (mutex avec un compteur) qui peuvent bloquer l'exécution, créer des [goulots d'étranglement](#), voire ne pas remplir leur rôle en permettant tout de même l'accès aux données protégées. Un autre effet est le blocage total des ressources, si le programme qui les utilisait n'a pas informé le système qu'il n'en avait plus besoin.

[https://fr.wikipedia.org/wiki/Exclusion\\_mutuelle](https://fr.wikipedia.org/wiki/Exclusion_mutuelle)

## Interblocage [\[ modifier | modifier le code \]](#)

 Article détaillé : [Interblocage](#).

L'[interblocage](#) (de l'anglais *deadlock*) se produit, par exemple, lorsqu'un thread T1 ayant déjà acquis la ressource R1 demande l'accès à une ressource R2, pendant que le thread T2, ayant déjà acquis la ressource R2, demande l'accès à la ressource R1. Chacun des deux threads attend alors la libération de la ressource possédée par l'autre. La situation est donc bloquée.

Plusieurs méthodes existent pour les éviter :

- imposition de l'ordre d'acquisition des ressources ;
- élimination lors de la conception par une analyse détaillée des algorithmes ;
- système préventif qui détecte un risque d'interblocage avant que celui-ci ne se produise durant l'exécution ;
- système de récupération si un interblocage se produit, le système doit pouvoir repartir dans un état valide.

# Checking properties on traces

**Définition** [ [modifier](#) | [modifier le code](#) ]

Un **Mutex** M.M.S est une **primitive de synchronisation** utilisée en **programmation informatique** pour éviter que des **ressources** partagées d'un système ne soient utilisées en même temps.

Certains algorithmes utilisent un *état* pour commander l'exécution : les mutex doivent savoir si les programmes concernés sont occupés (*busy*) ou s'ils ont terminé et sont en attente (*wait*). De tels algorithmes sont par exemple :

task1

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (true);
```

Solution to the critical-section problem using mutex locks.

<https://cdynamicprogramming.blogspot.com/p/mutex-vs-semaphore.html>

task2

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (true);
```

Solution to the critical-section problem using mutex locks.

Le lock ne peut pas être acquis 2 fois. L'acquisition est habituellement bloquante.

# Checking properties on traces

- Specifying properties over time.
- Two types of properties:
  - **Safety property:** asserts that nothing bad happens.
    - *If user1 possesses a mutex then user2 cannot take it until user1 releases it.*
  - **Liveness property:** asserts that something good eventually happens.



# Checking properties on traces

- Specifying properties over time.
- Two types of properties:
  - **Safety property:** asserts that nothing bad happens.
    - *If user1 possesses a mutex then user2 cannot take it until user1 releases it.*
  - **Liveness property:** asserts that something good eventually happens.
    - If user1 asks to take a mutex, she/he will eventually possess it.

# Checking properties on traces

- Specifying properties over time.
- Two types of properties:
  - **Safety property:** asserts that nothing bad happens.
    - *If user1 possesses a mutex then user2 cannot take it **until** user1 releases it.*
  - **Liveness property:** asserts that something good eventually happens.
    - If user1 asks to take a mutex, she/he will **eventually** possess it.

# Checking properties on traces

- Specifying properties over time.
- Two types of properties:
  - **Safety property:** asserts that nothing bad happens.
    - *If user1 possesses a mutex then user2 cannot take it **until** user1 releases it.*
  - **Liveness property:** asserts that something good eventually happens.
    - If user1 asks to take a mutex, she/he will **eventually** possess it.



Safety properties violation can be determinate over finite execution while liveness properties cannot (something good can always happen latter)

# Logiques temporelles

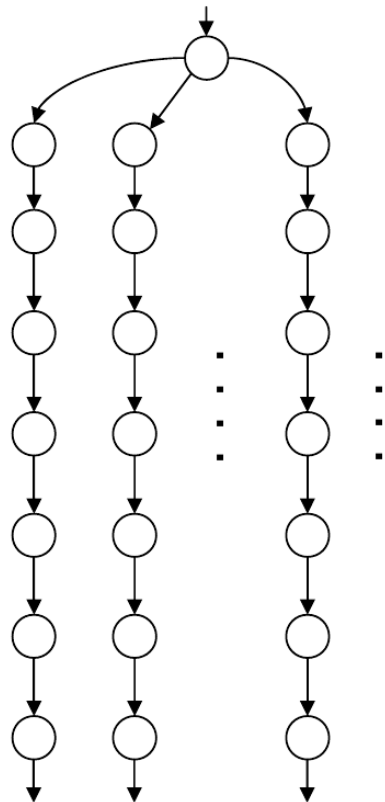
- Elles rajoutent une notion de temporalité au dessus de la logique Booleenne.
- Deux classes principales: *Linear Temporal Logic* (LTL) et *Computational Tree Logic* (CTL)

# trace (run), state space and real life

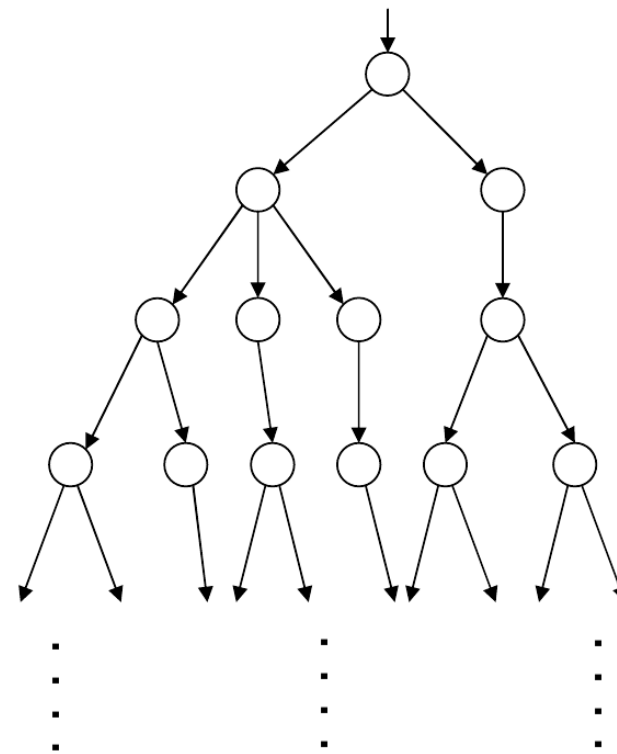
- All traces are speaking about **time**

## Branching Time Logic

Sets of paths?



Or computation tree?



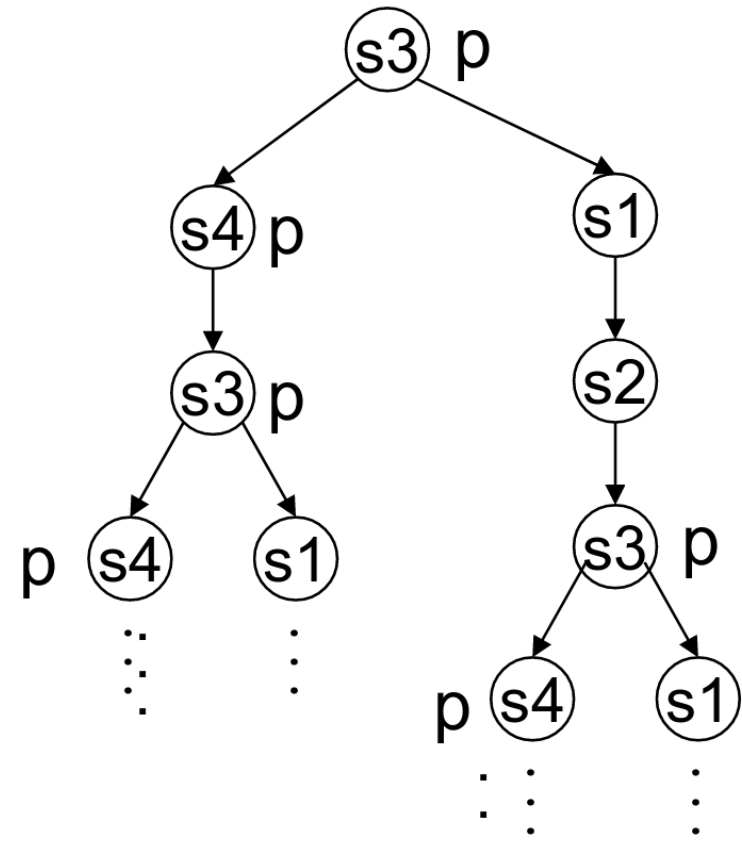
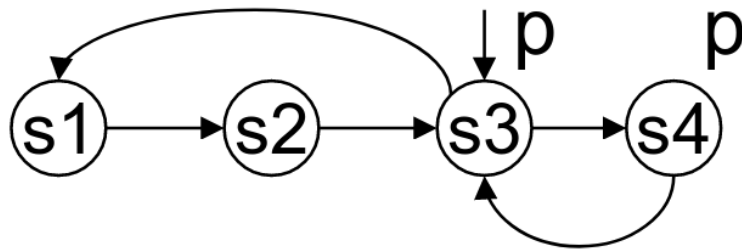
Taken from Mads Dam Theoretical Computer Science KTH, 2009

# trace (run), state space and real life

- All traces are speaking about **time**.
- When possible, a state space (also named transition system) represents in a finite way an infinite (set of) traces

## Computation Tree

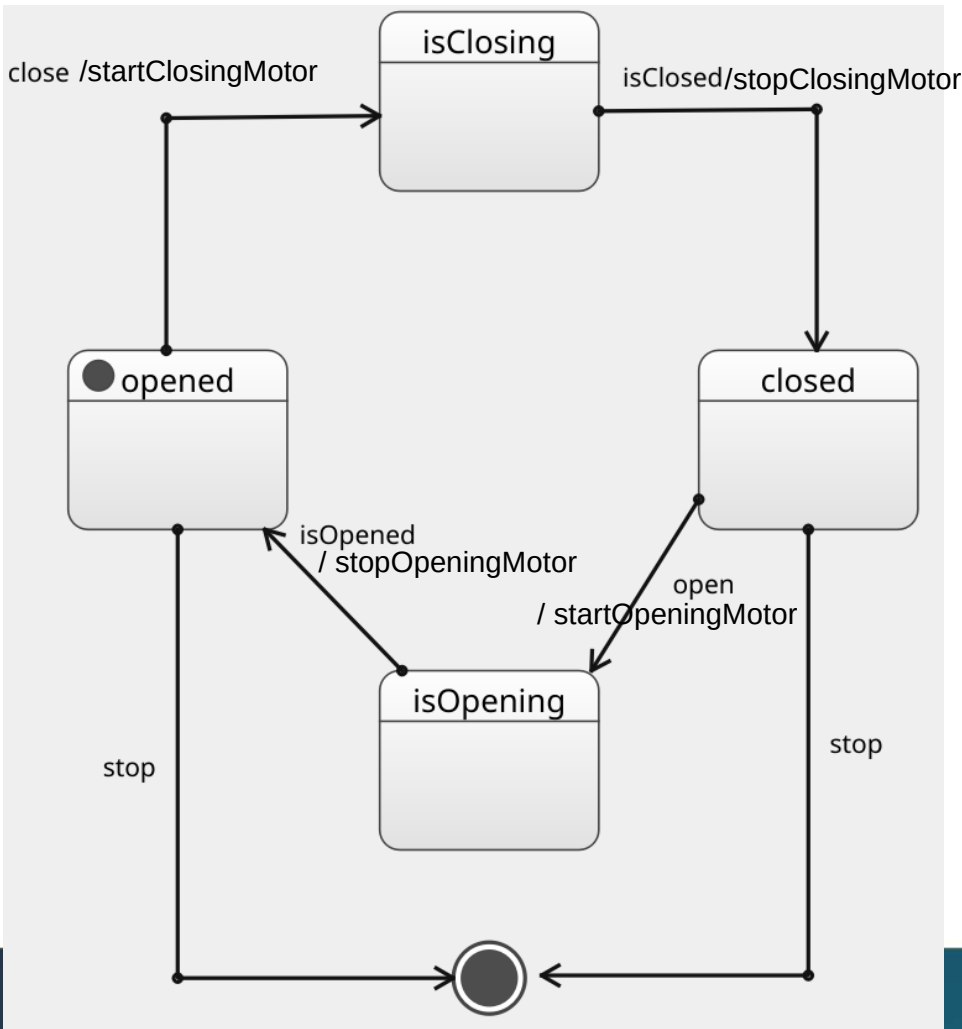
## Transition System



Taken from Tevfik Bultan, Model Checking Foundations and Applications

# V&V ?

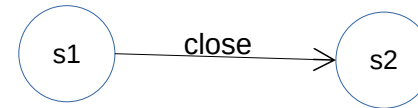
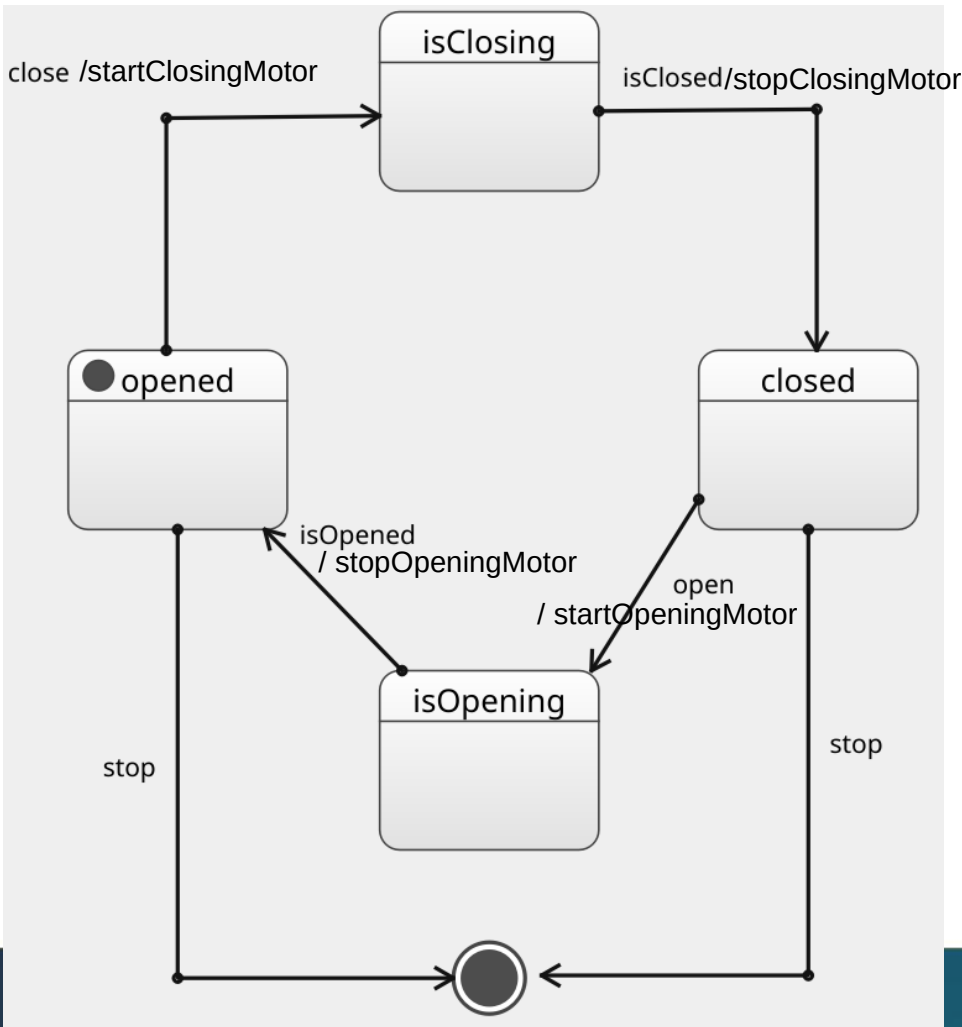
- ensemble de chemins d'exécutions finis ?
  - Énumération de l'espace d'état (habituellement un graphe orienté d'une forme particulière : *Labelled Transition system* or *Kripke structure*)



s1

# V&V ?

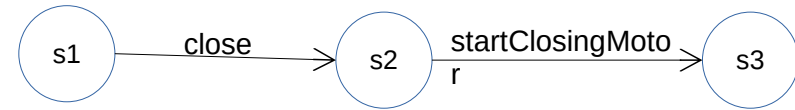
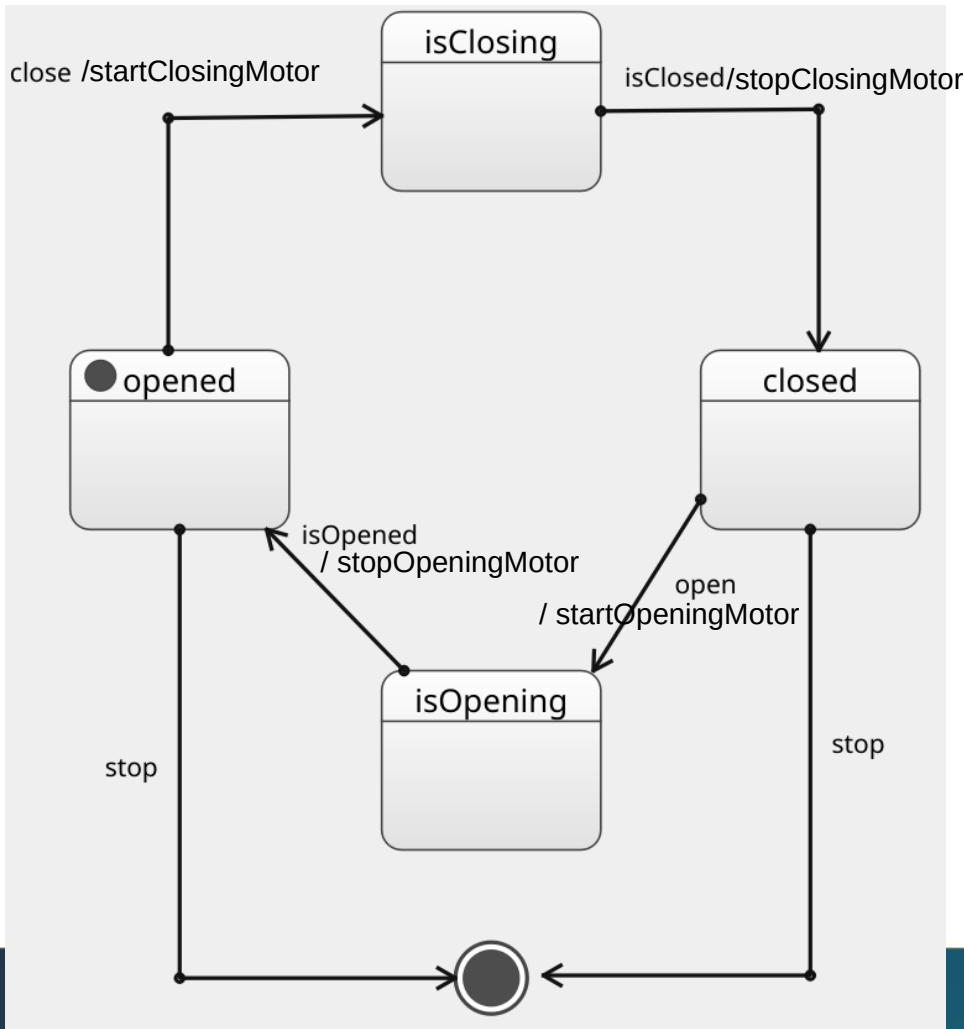
- ensemble de chemins d'exécutions finis ?
  - Énumération de l'espace d'état (habituellement un graphe orienté d'une forme particulière : *Labelled Transition system* or *Kripke structure*)





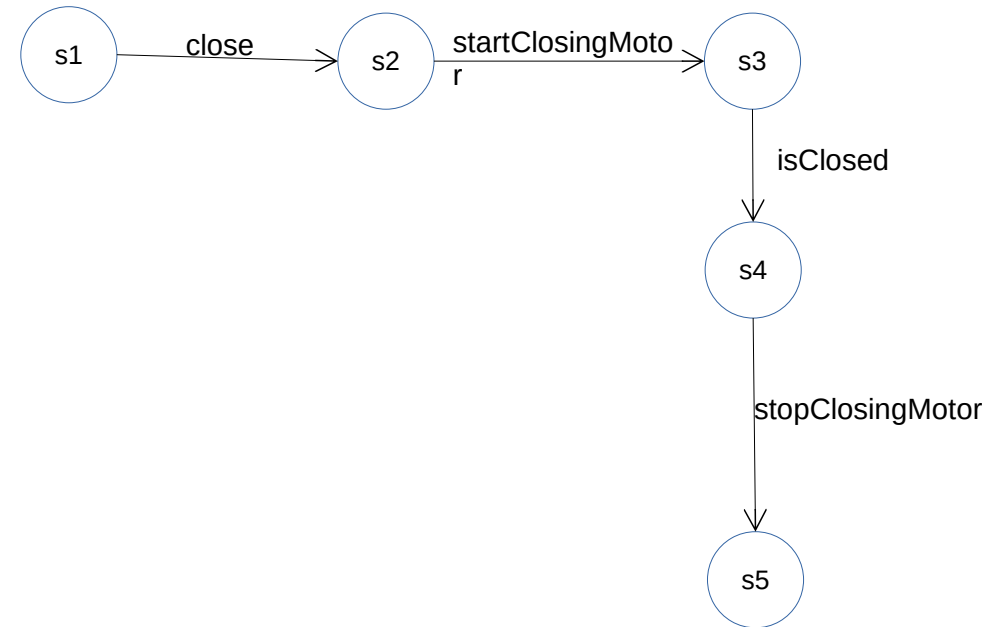
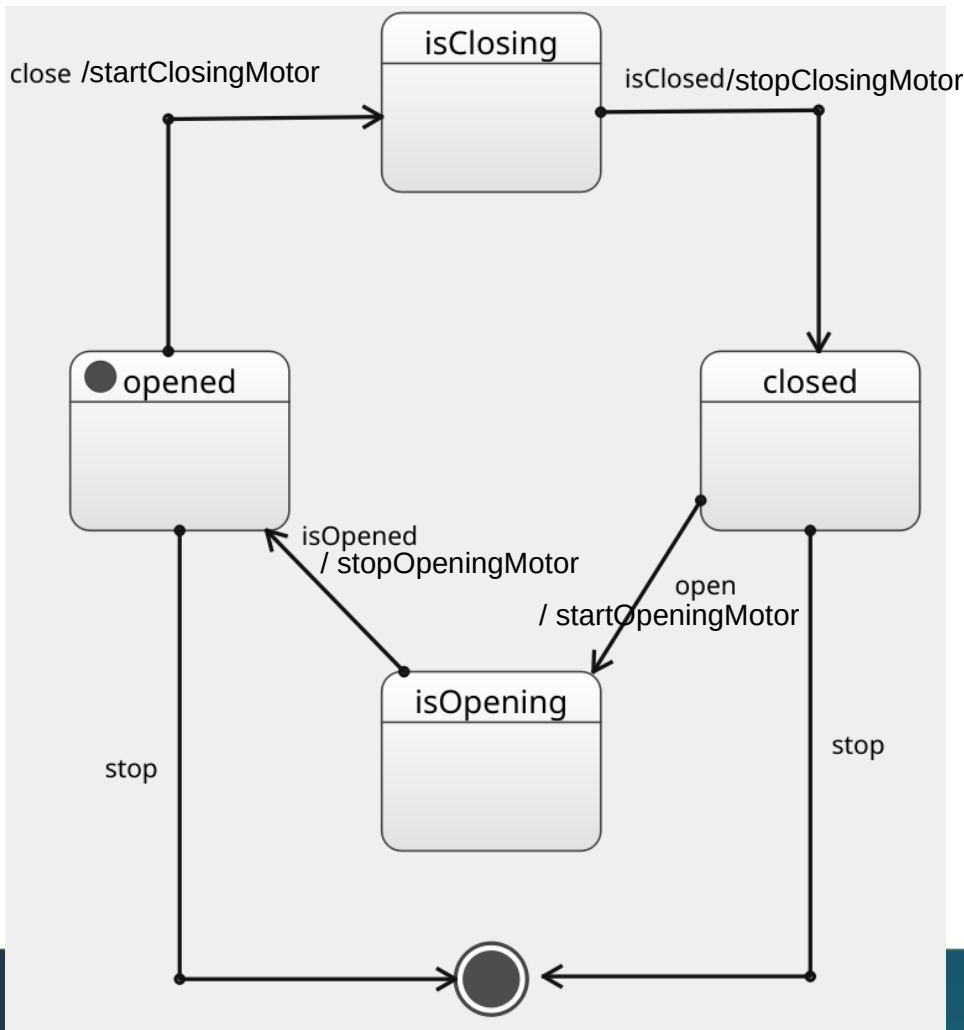
# V&V ?

- ensemble de chemins d'exécutions finis ?
  - Énumération de l'espace d'état (habituellement un graphe orienté d'une forme particulière : *Labelled Transition system* or *Kripke structure*)



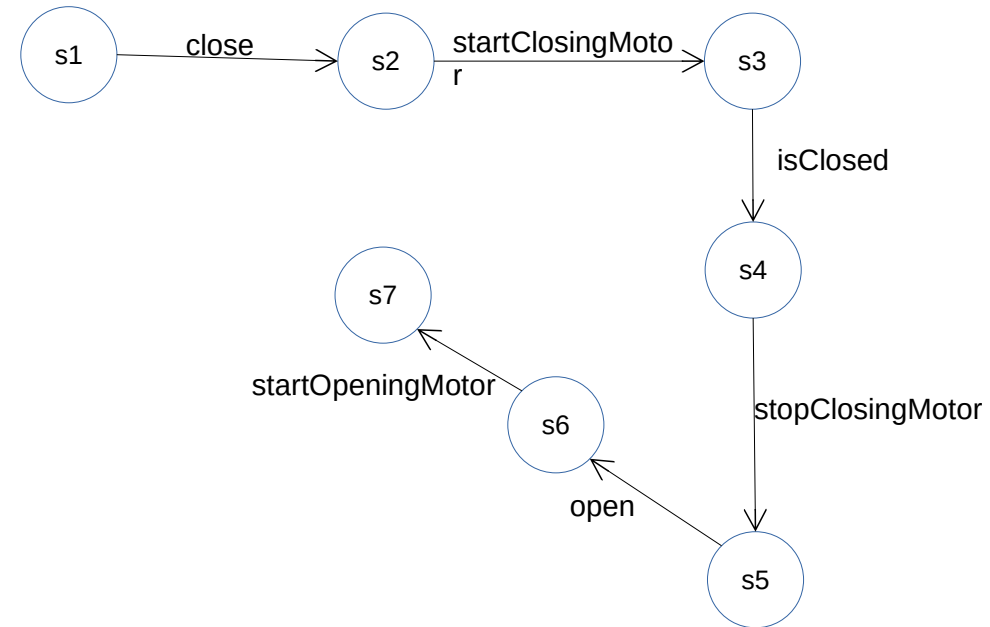
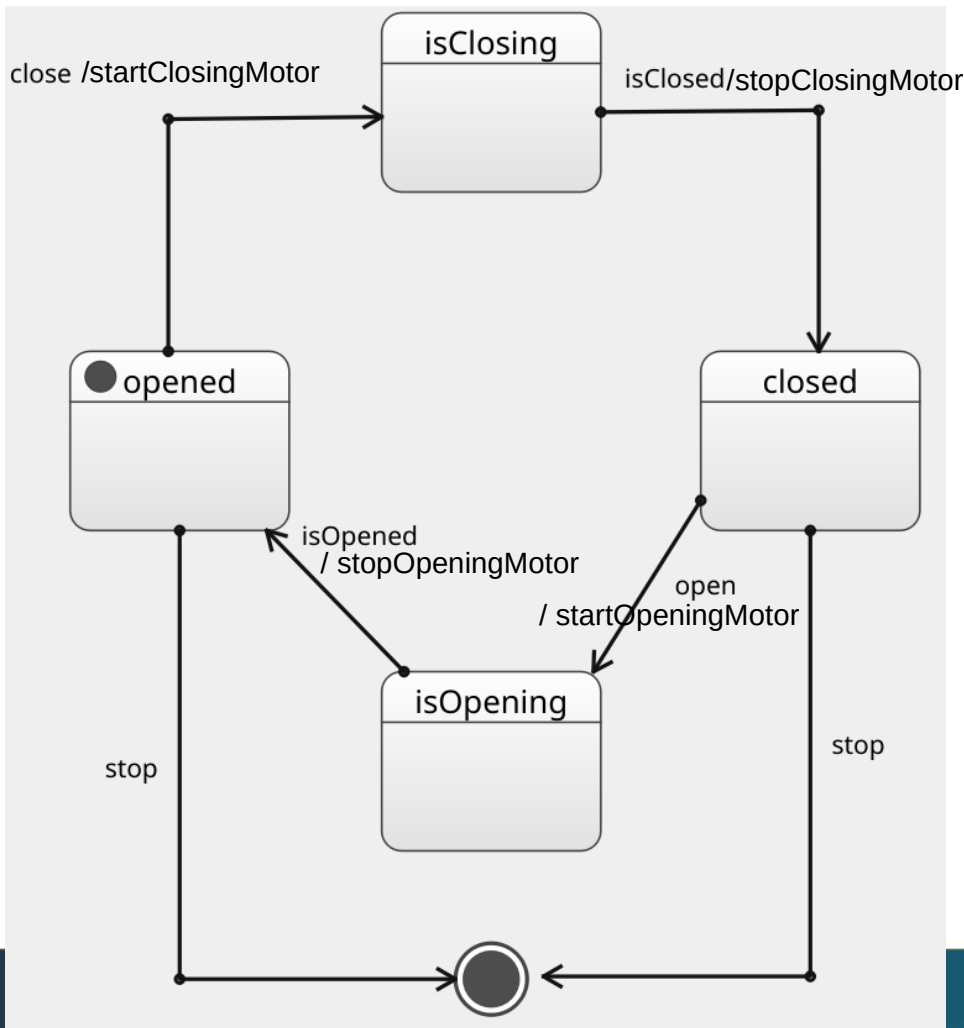
# V&V ?

- ensemble de chemins d'exécutions finis ?
  - Énumération de l'espace d'état (habituellement un graphe orienté d'une forme particulière : *Labelled Transition system* or *Kripke structure*)



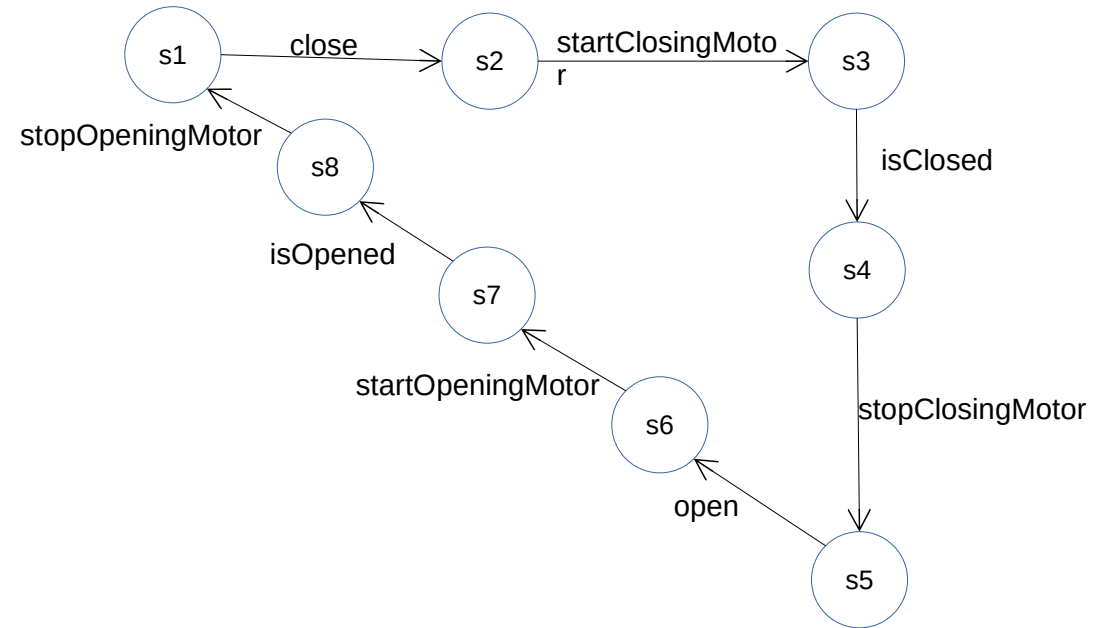
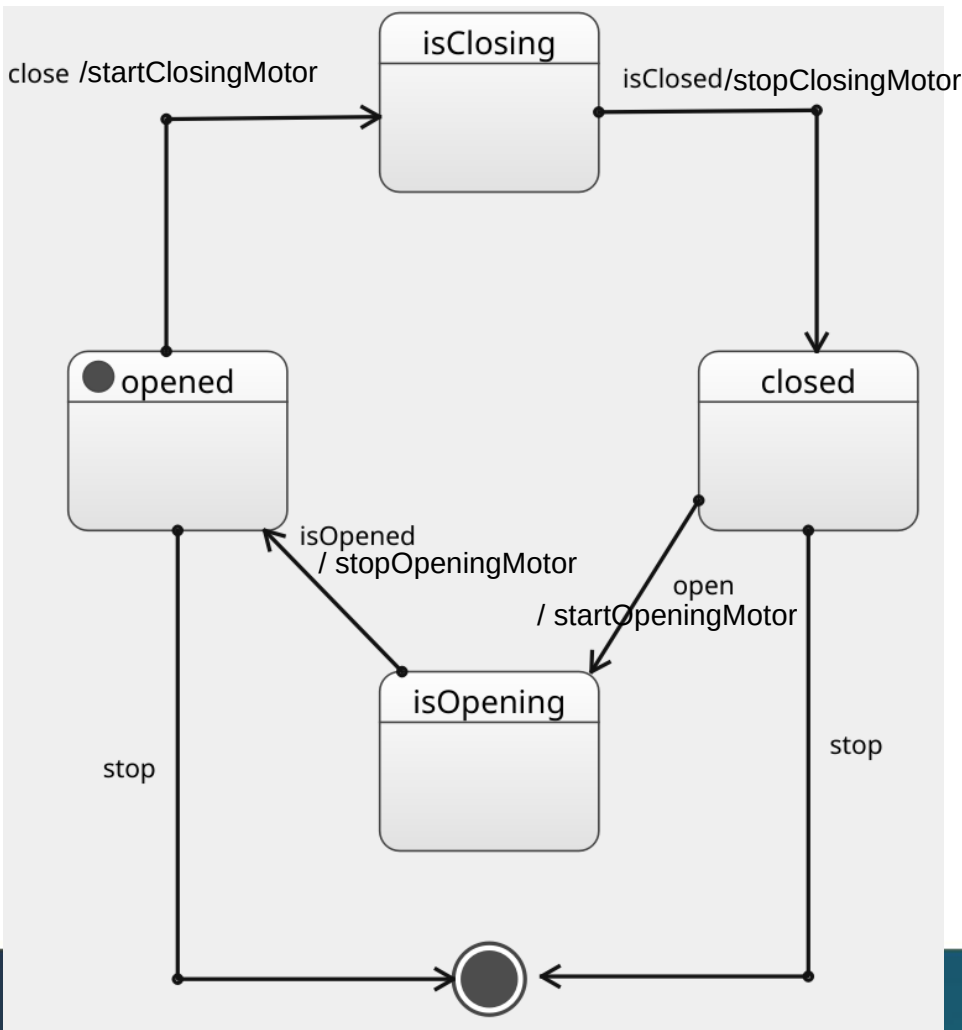
# V&V ?

- ensemble de chemins d'exécutions finis ?
  - Énumération de l'espace d'état (habituellement un graphe orienté d'une forme particulière : *Labelled Transition system* or *Kripke structure*)



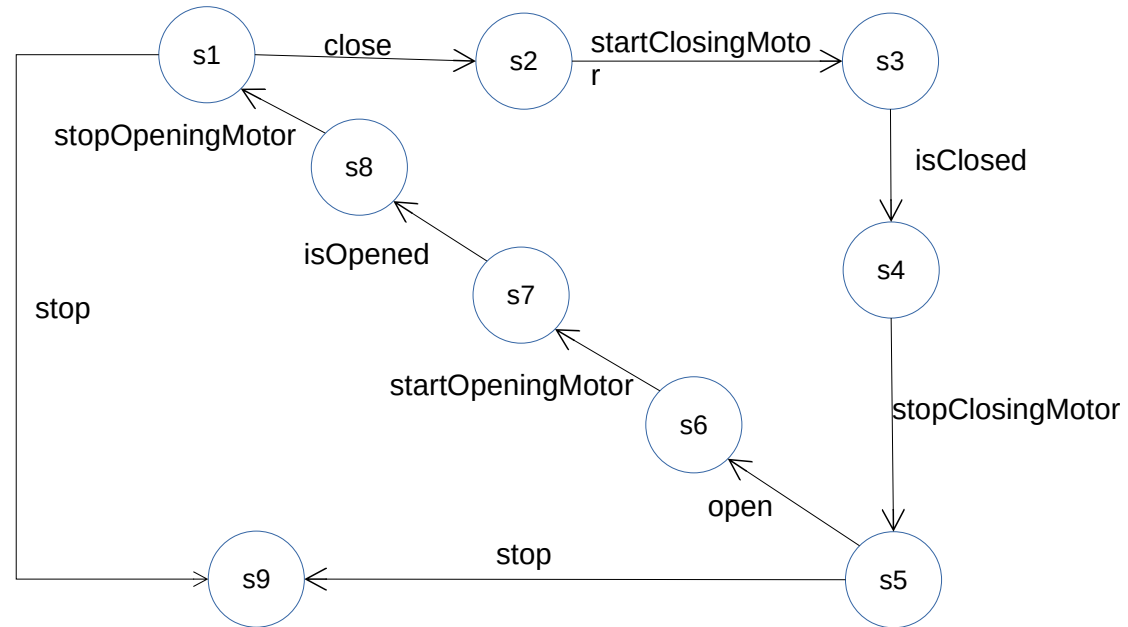
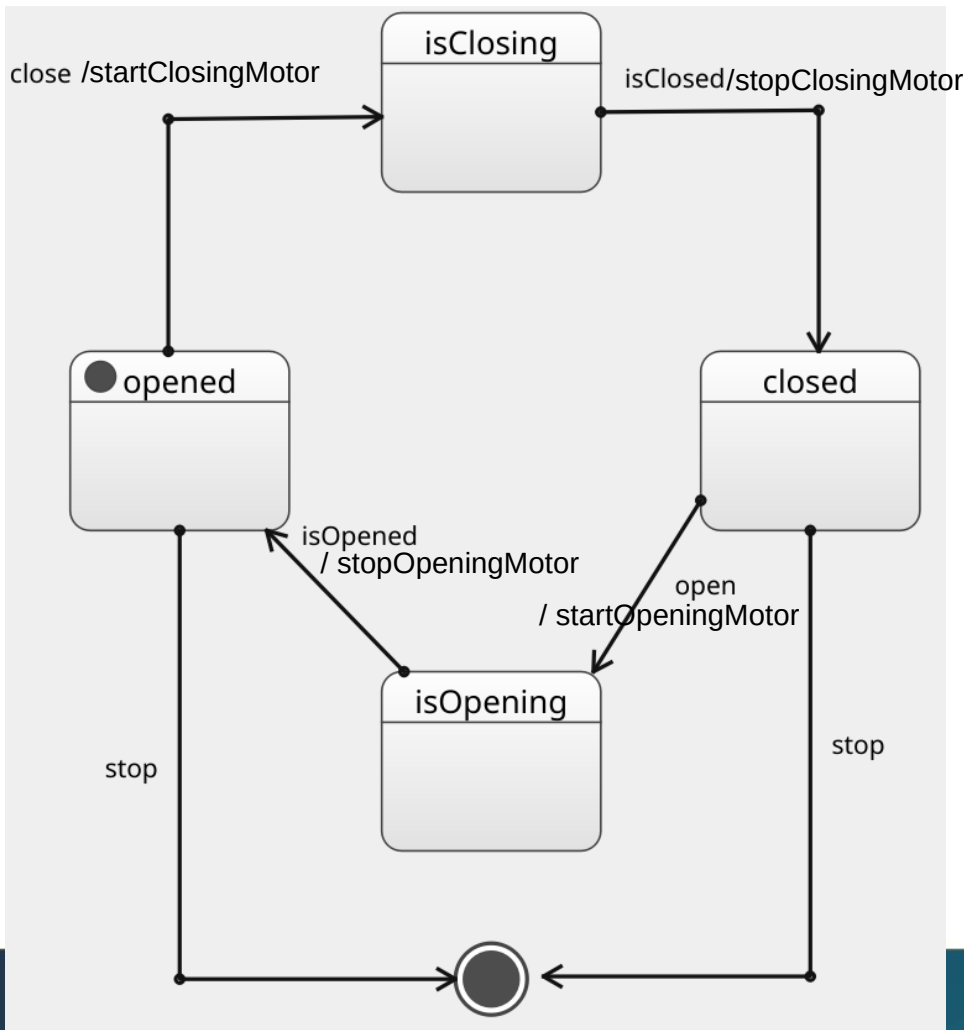
# V&V ?

- ensemble de chemins d'exécutions finis ?
- Énumération de l'espace d'état (habituellement un graphe orienté d'une forme particulière : *Labelled Transition system* or *Kripke structure*)



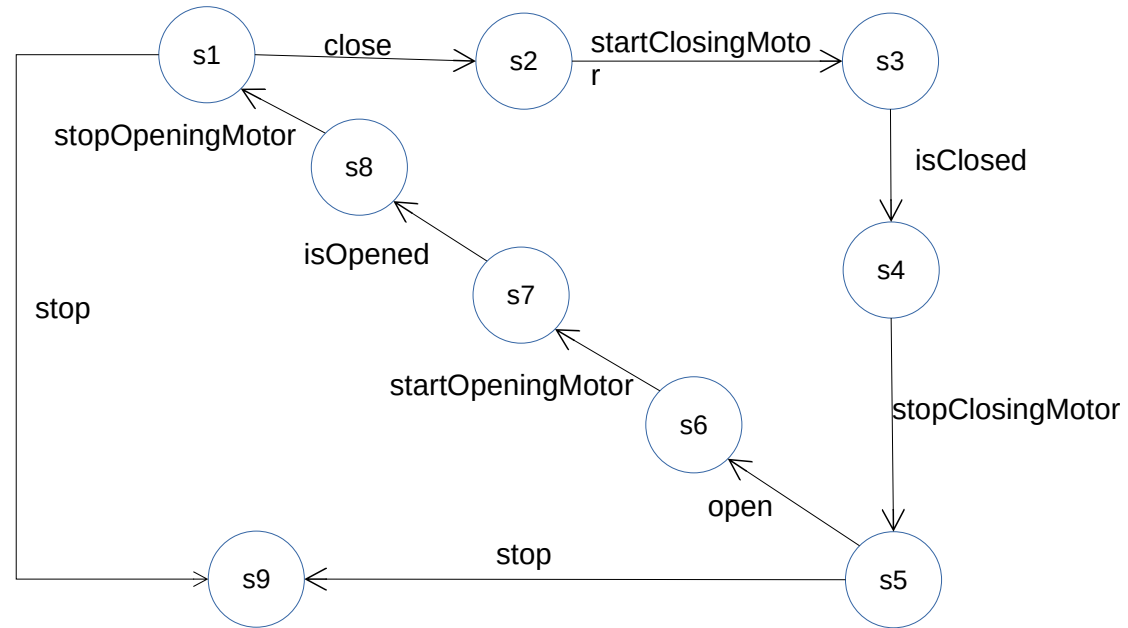
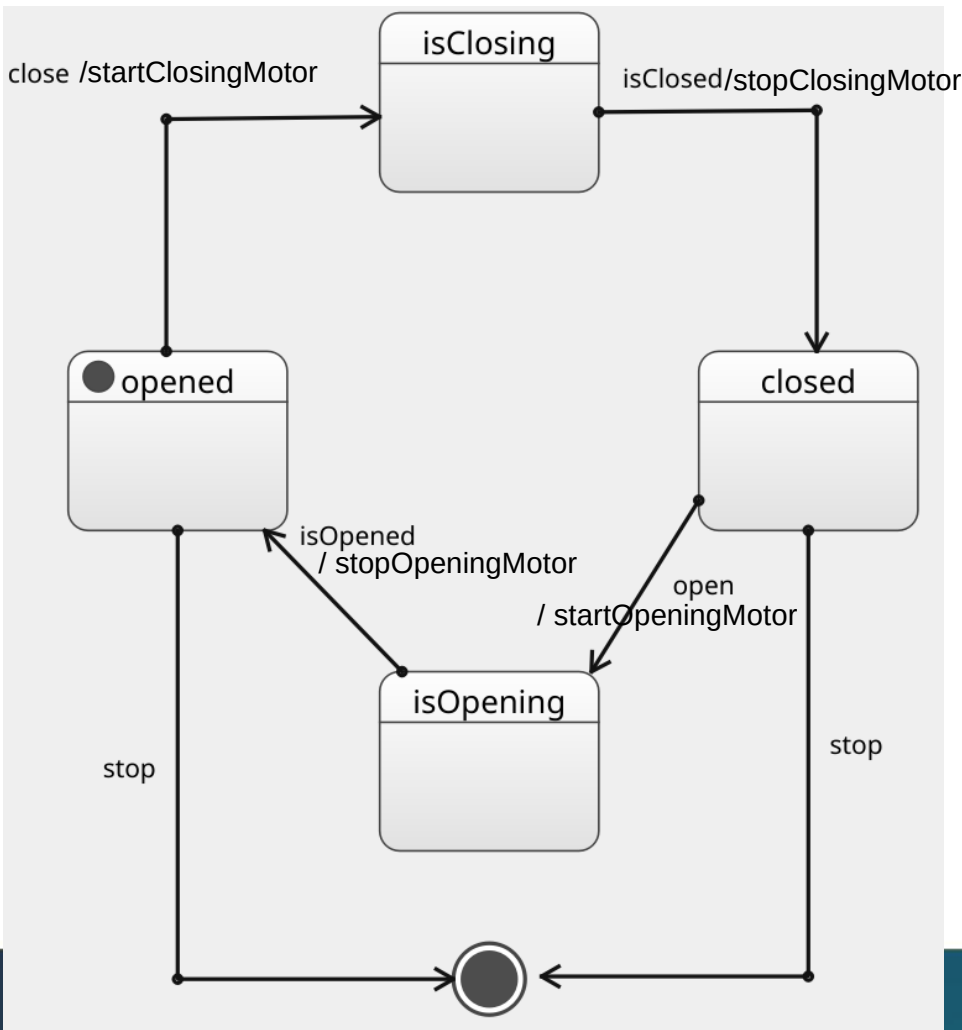
# V&V ?

- ensemble de chemins d'exécutions finis ?
- Énumération de l'espace d'état (habituellement un graphe orienté d'une forme particulière : *Labelled Transition system* or *Kripke structure*)



# V&V ?

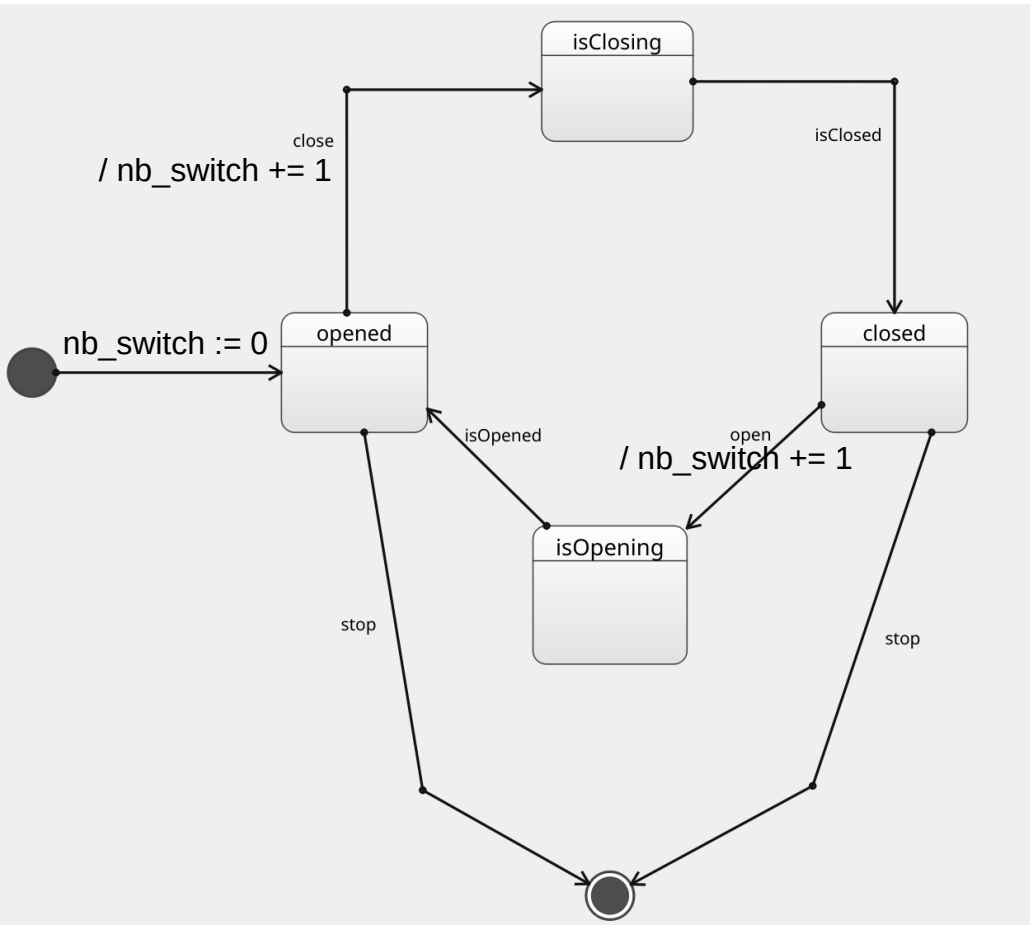
- ensemble de chemins d'exécutions finis ?
- Énumération de l'espace d'état (habituellement un graphe orienté d'une forme particulière : *Labelled Transition system* ou *Kripke structure*)



Plus les actions *onEnter* et *onExit* !

# V&V ?

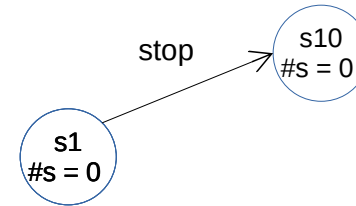
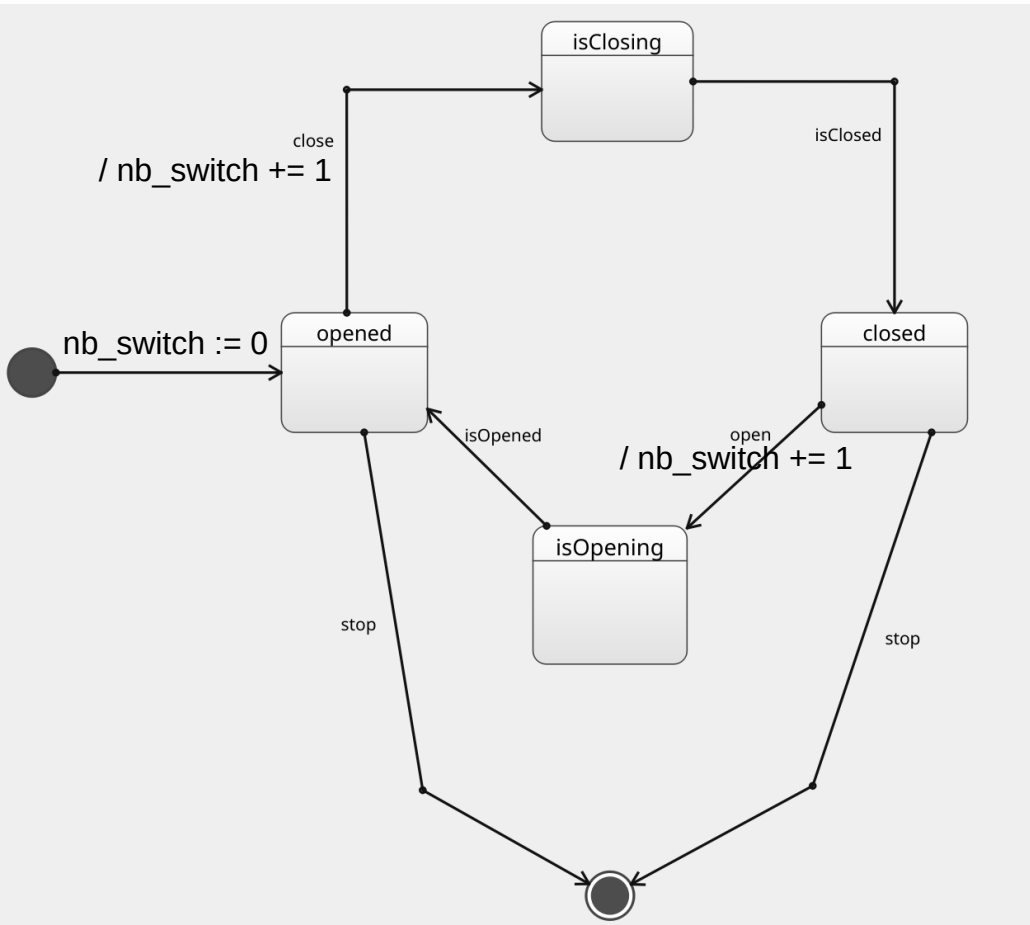
- ensemble de chemins d'exécutions finis ?



s1  
#s = 0

# V&V ?

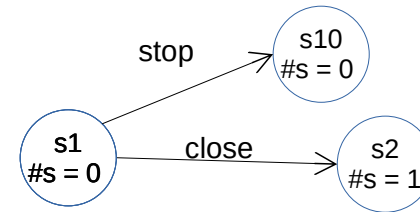
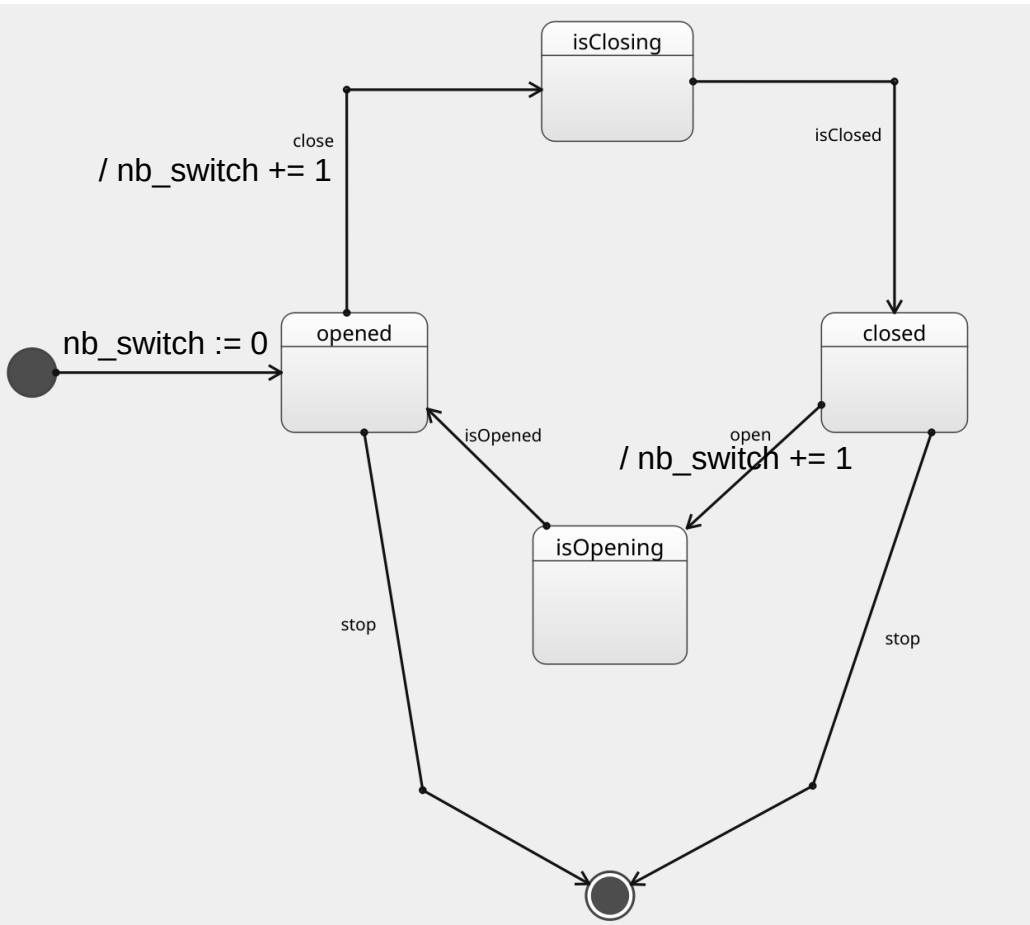
- ensemble de chemins d'exécutions finis ?





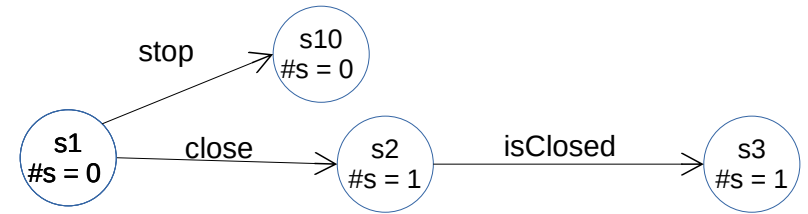
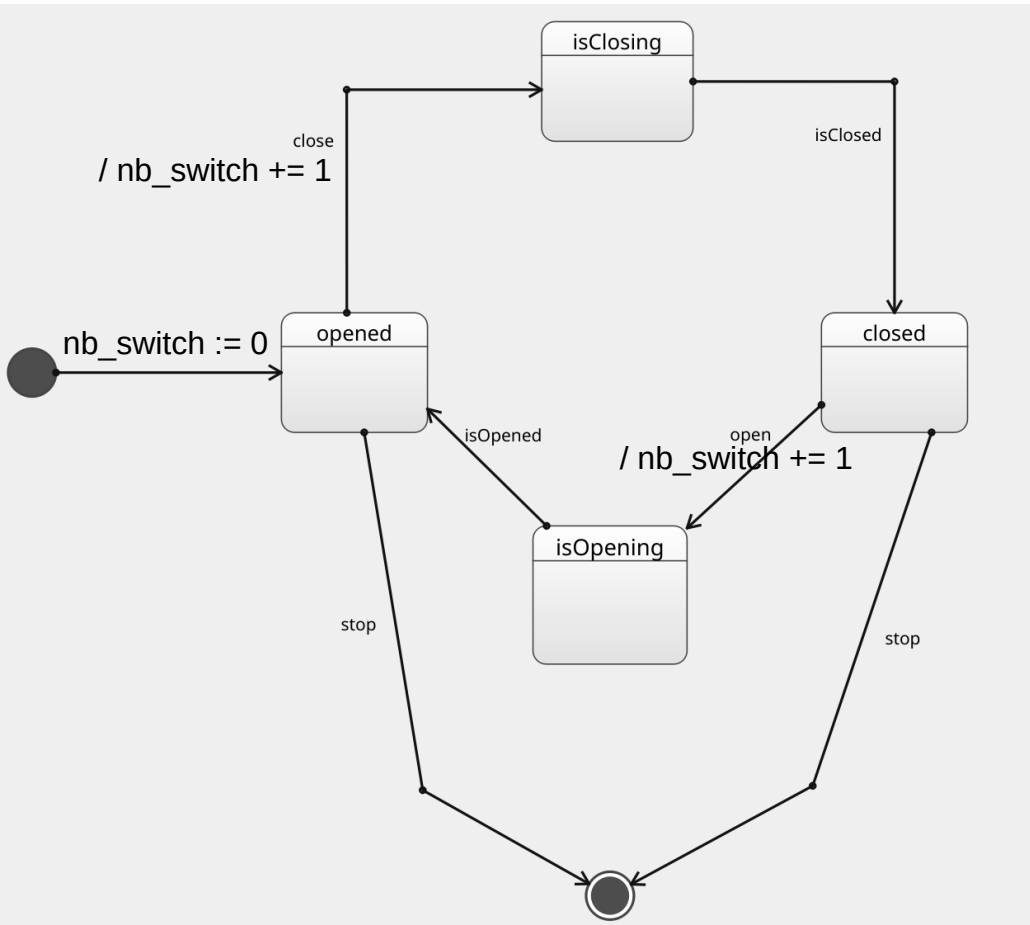
# V&V ?

- ensemble de chemins d'exécutions finis ?



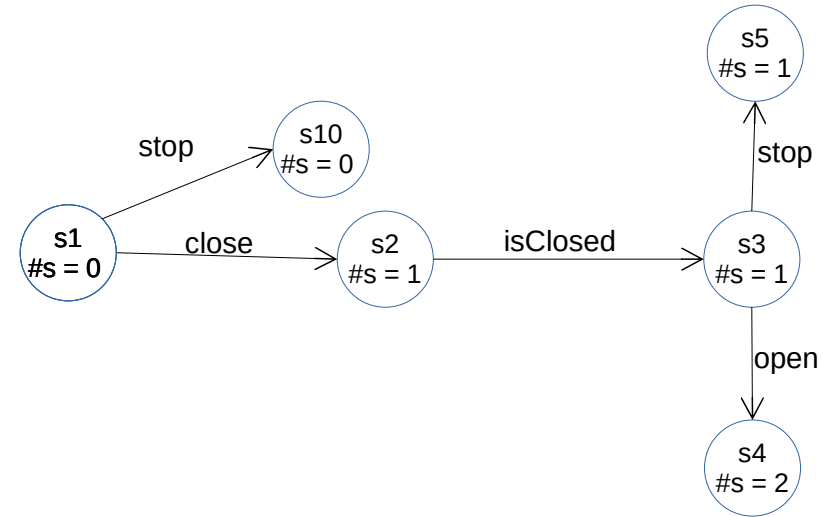
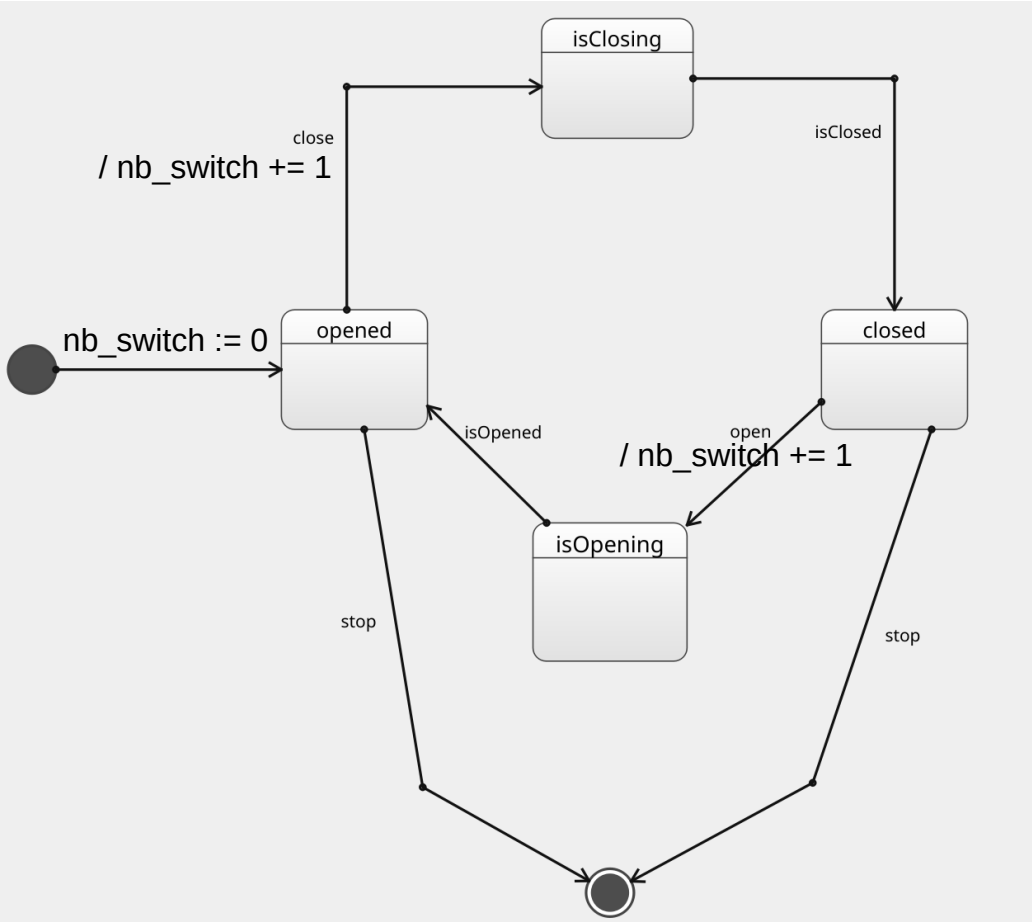
# V&V ?

- ensemble de chemins d'exécutions finis ?



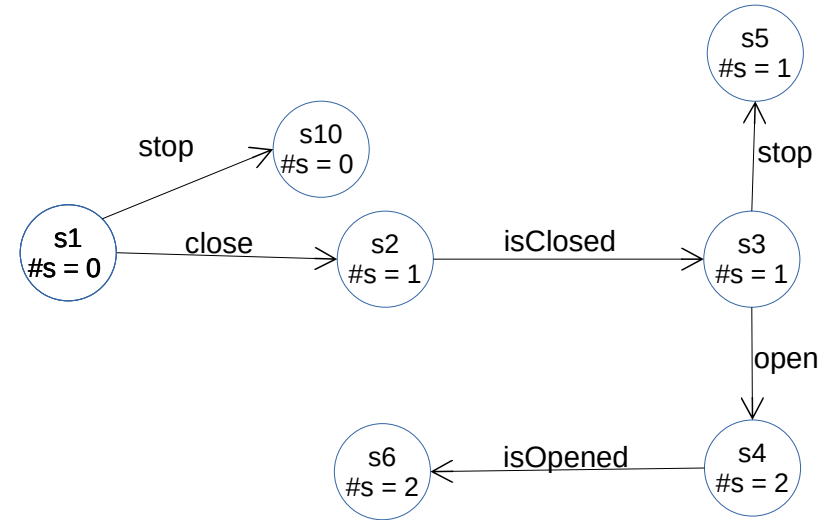
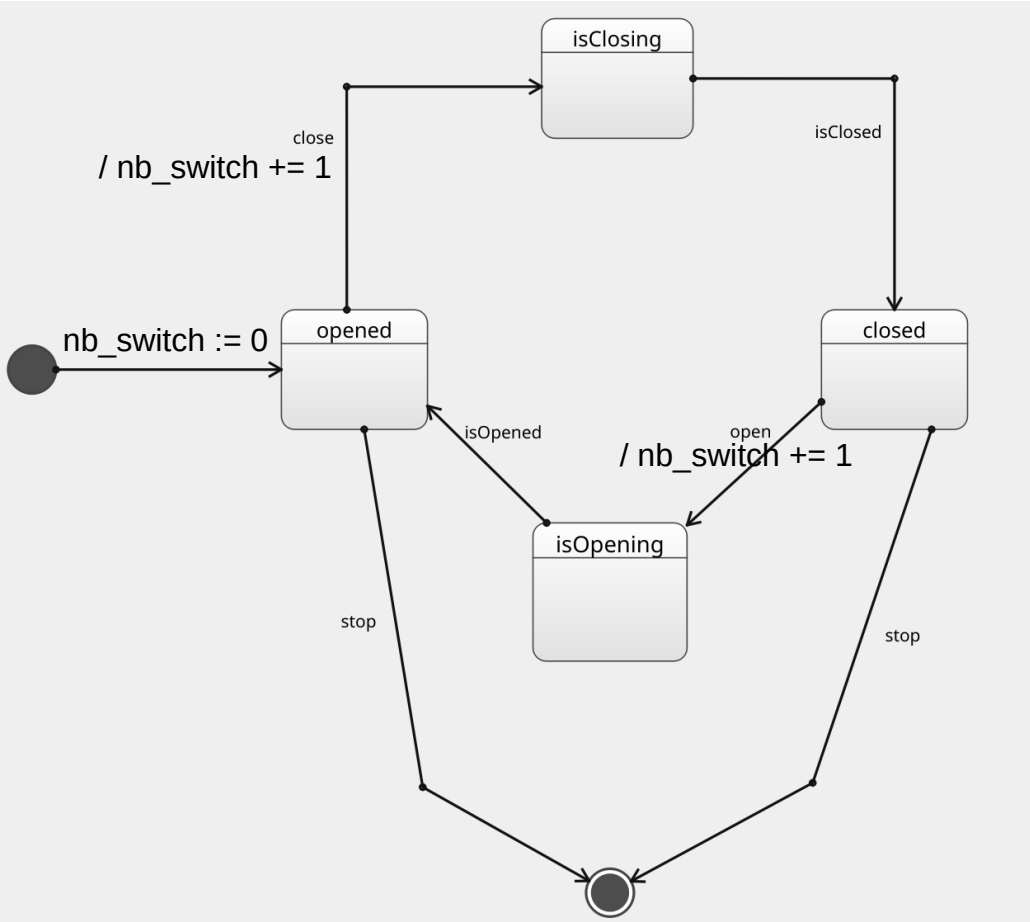
# V&V ?

- ensemble de chemins d'exécutions finis ?



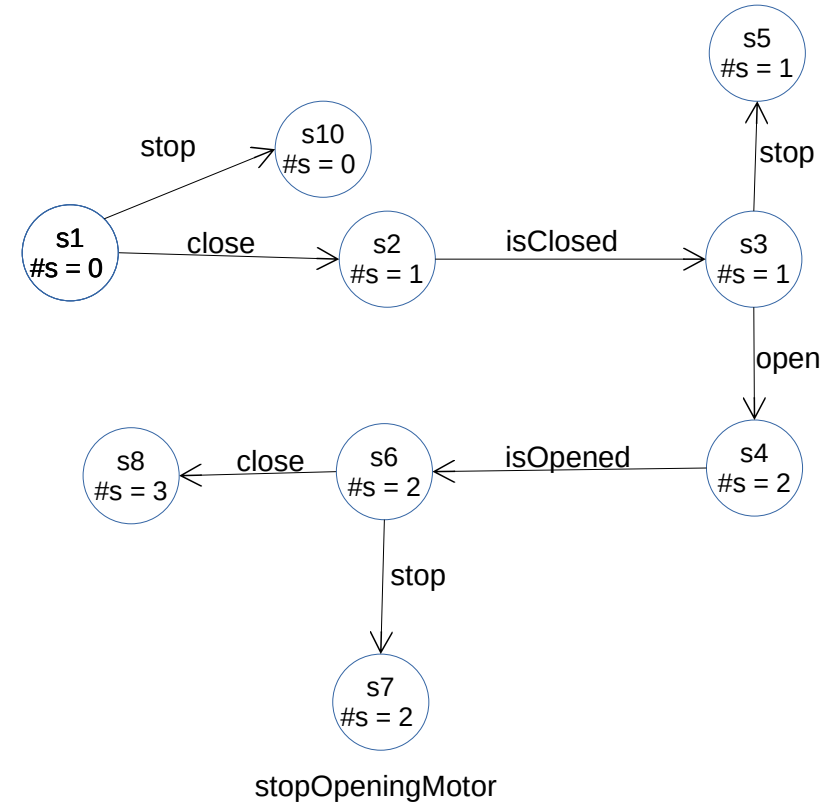
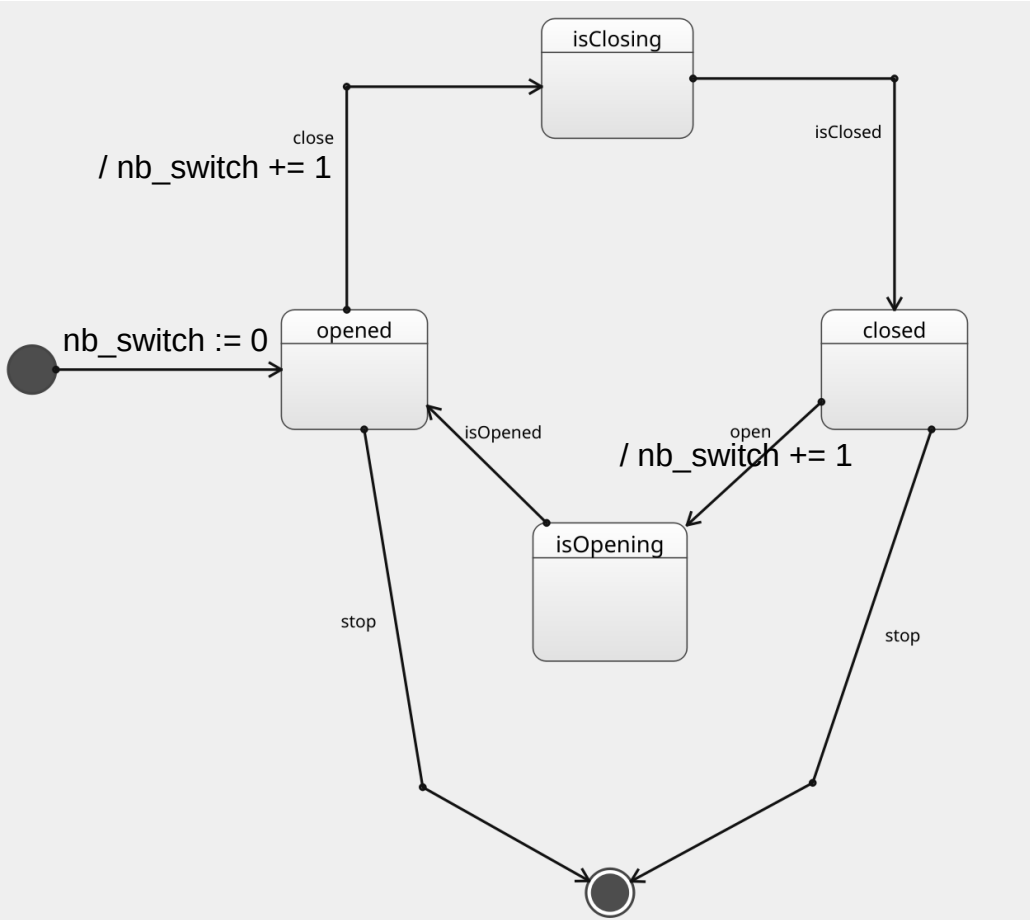
# V&V ?

- ensemble de chemins d'exécutions finis ?



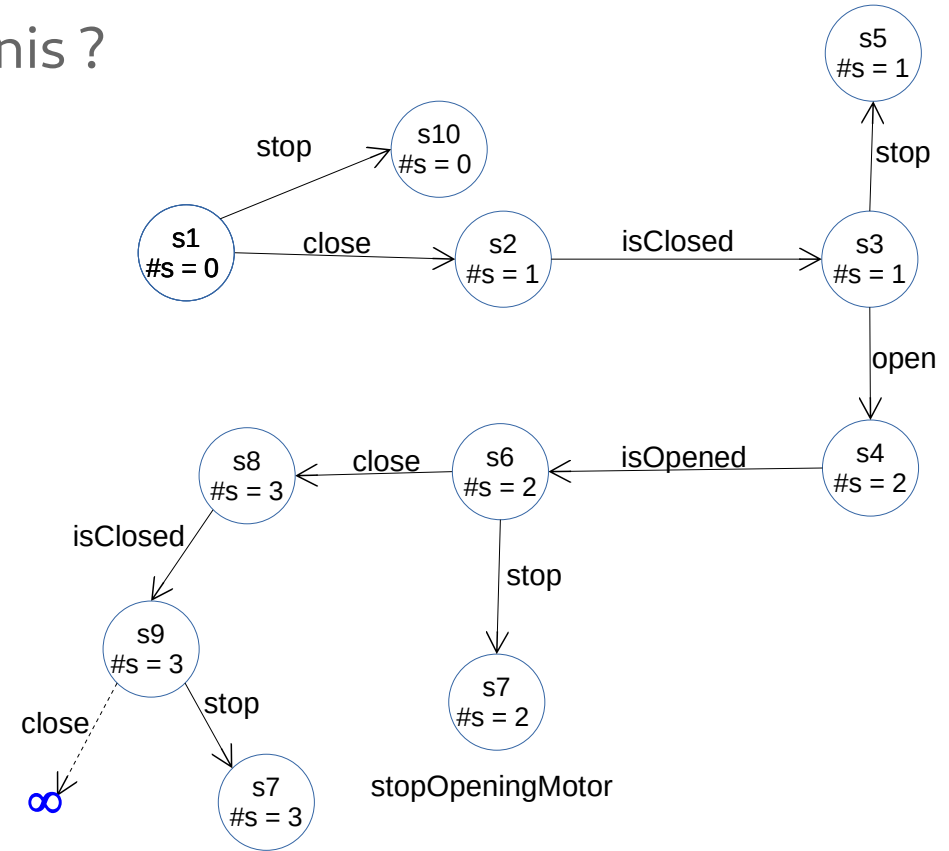
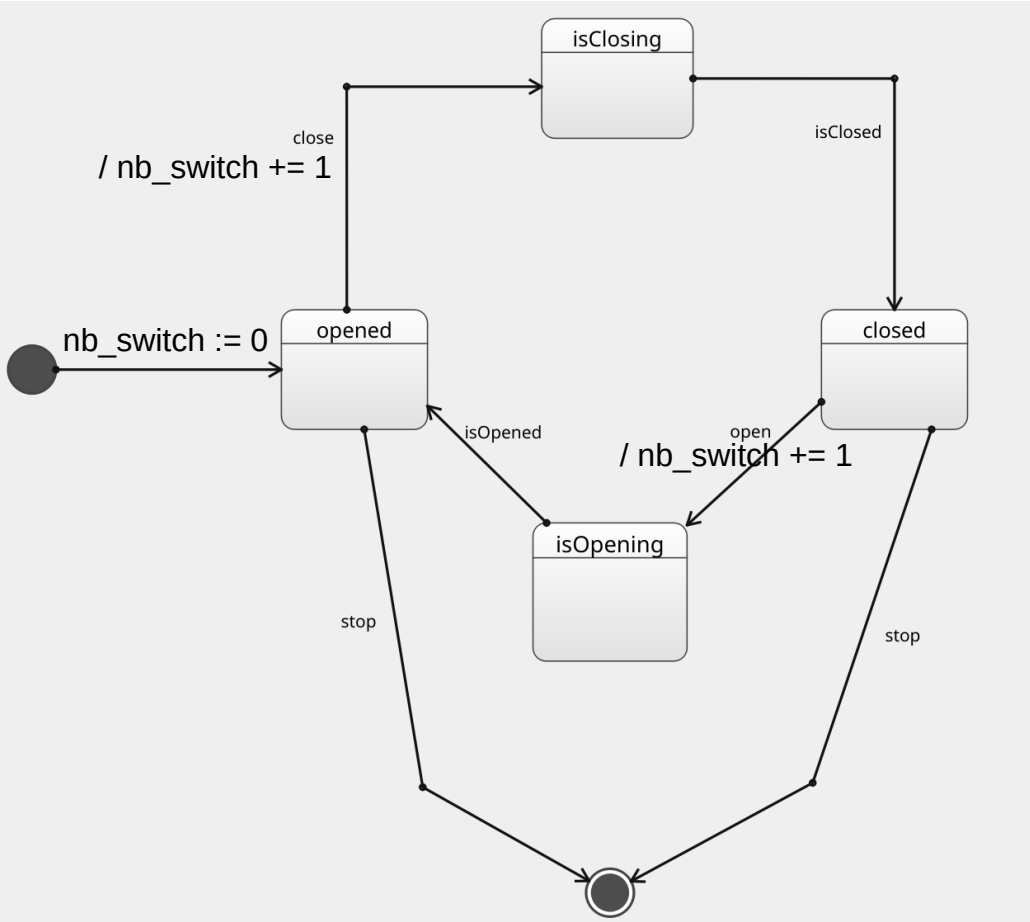
# V&V ?

- ensemble de chemins d'exécutions finis ?



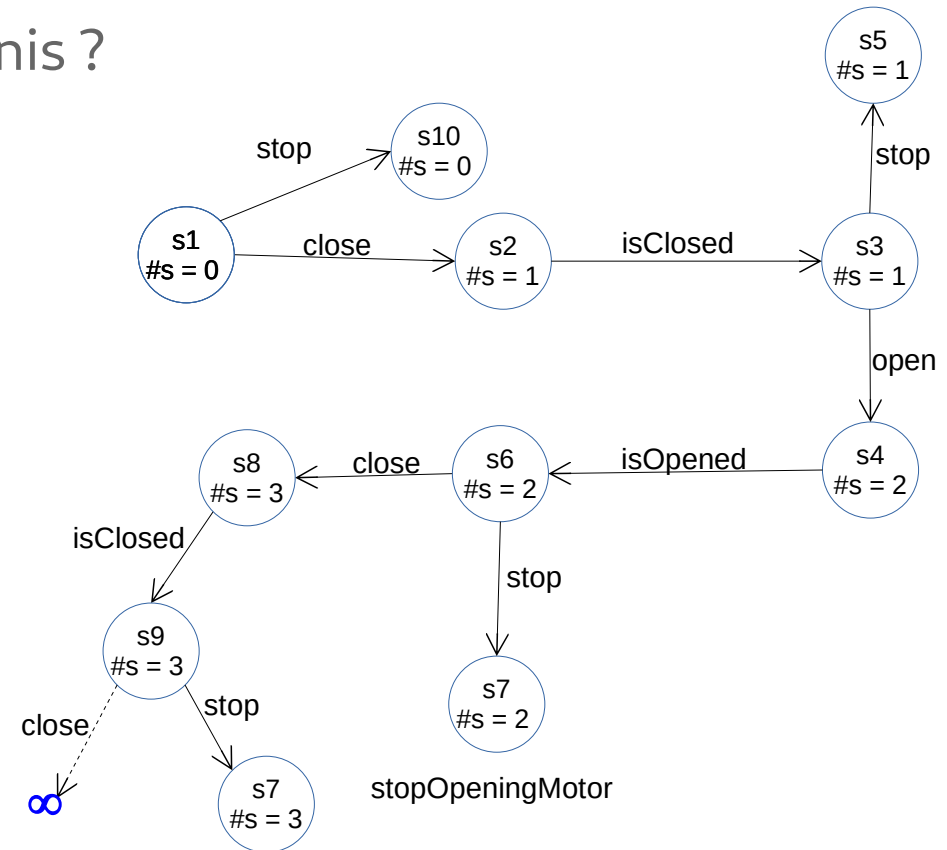
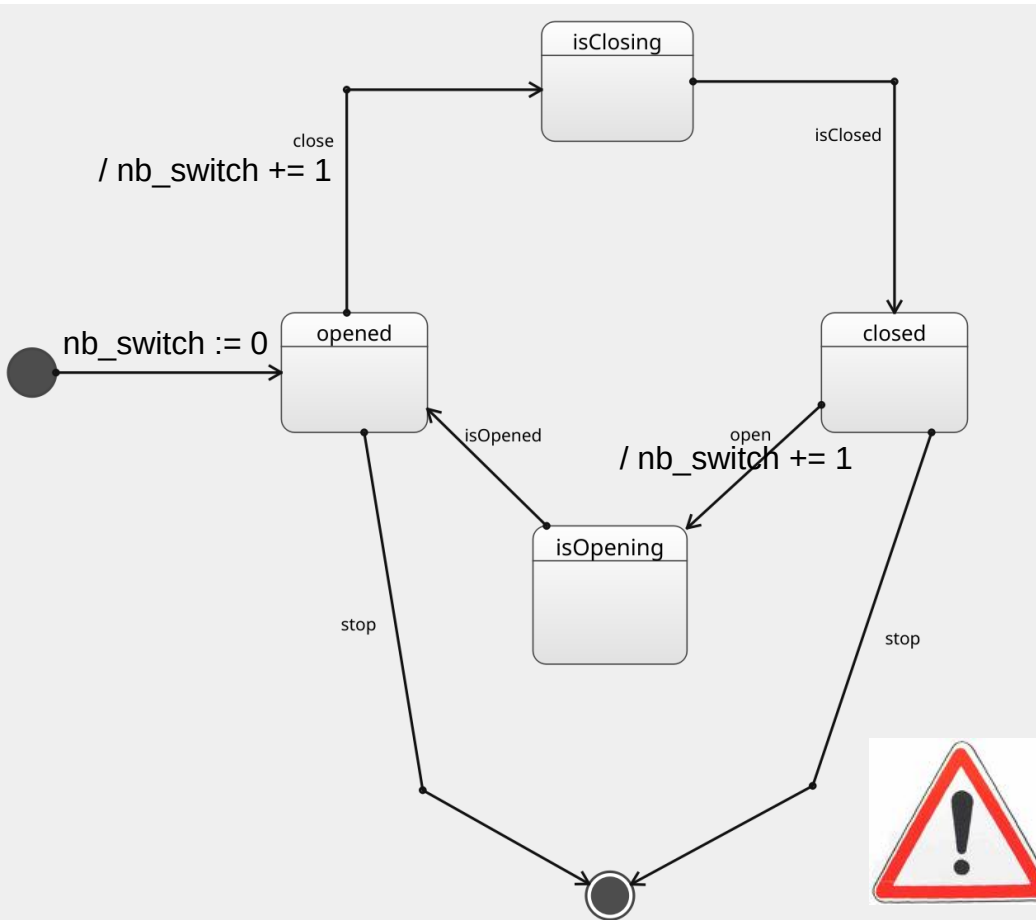
# V&V ?

- ensemble de chemins d'exécutions finis ?



# V&V ?

- ensemble de chemins d'exécutions finis ?

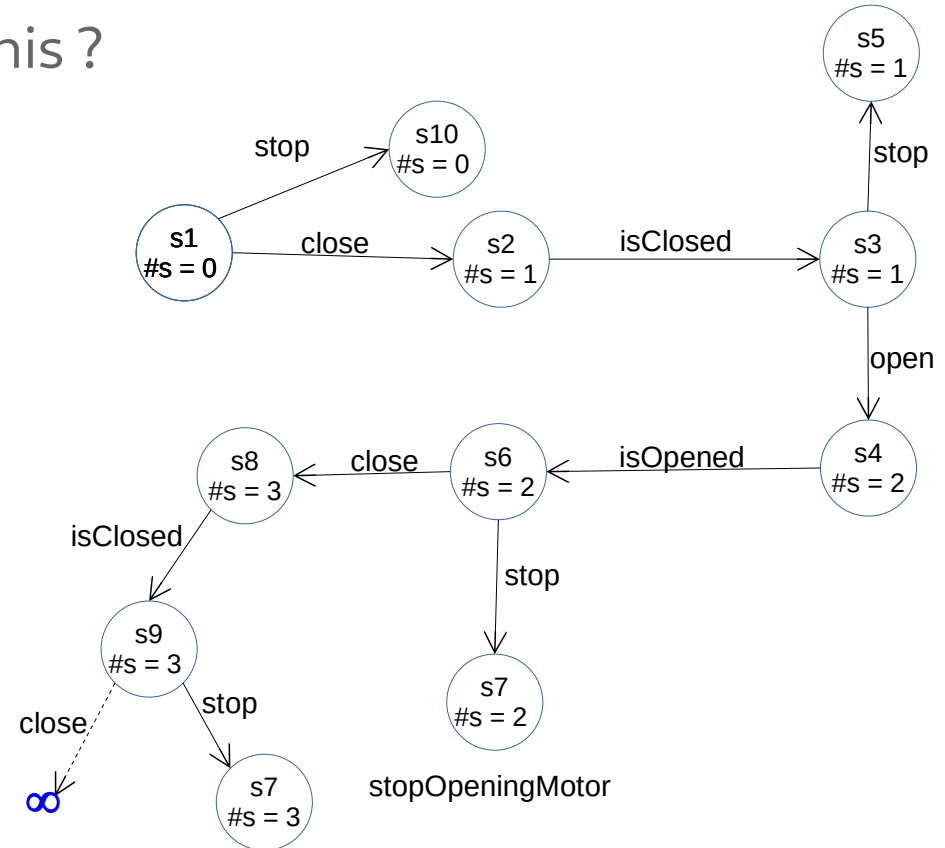
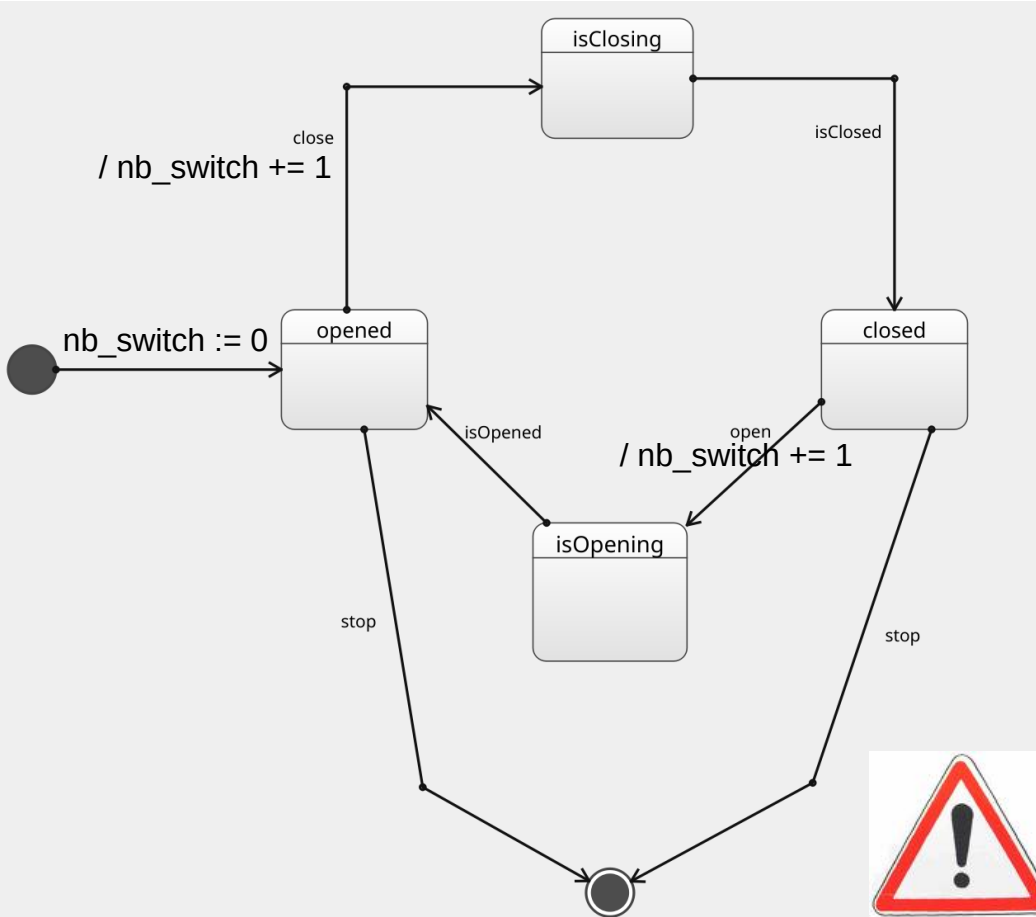


Tout ce qui est dans la state machine est considéré dans la construction de l'espace d'état.

Tout ce qui n'est pas dans la state machine ne peut pas être utilisé pour "poser des questions"

# V&V ?

- ensemble de chemins d'exécutions finis ?



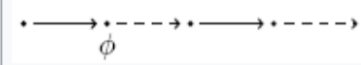
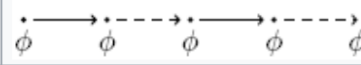
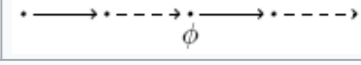
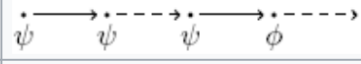
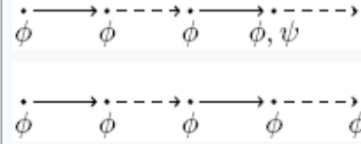
- Determine what should be **observable events** / **conditions** / **execution states**



# Logiques temporelles

- Elles rajoutent une notion de temporalité au dessus de la logique Booleenne.
- Deux classes principales: *Linear Temporal Logic* (LTL) et *Computational Tree Logic* (CTL)

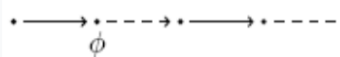
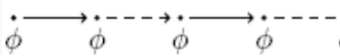
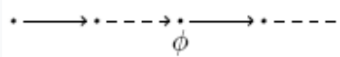
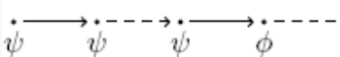
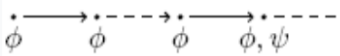
Textual	Symbolic	
Unary operators:		
$X \phi$	$\bigcirc \phi$	neXt:
$G \phi$	$\square \phi$	Globally:
$F \phi$	$\diamond \phi$	Finally:
Binary operators:		
$\psi U \phi$	$\psi \mathcal{U} \phi$	Until:
$\psi R \phi$	$\psi \mathcal{R} \phi$	Release:

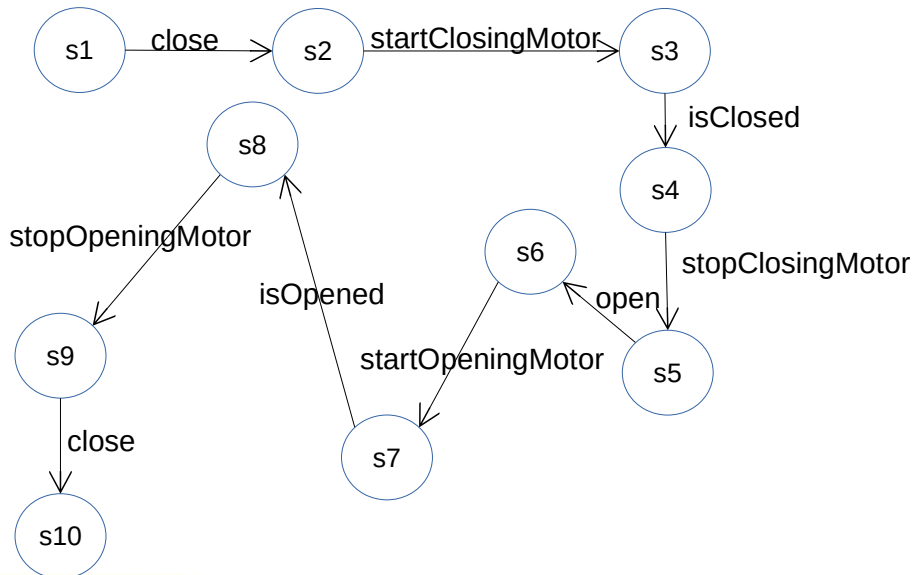
Textual	Symbolict	Explanation	Diagram
Unary operators:			
$X \phi$	$\bigcirc \phi$	neXt: $\phi$ has to hold at the next state.	
$G \phi$	$\square \phi$	Globally: $\phi$ has to hold on the entire subsequent path.	
$F \phi$	$\diamond \phi$	Finally: $\phi$ eventually has to hold (somewhere on the subsequent path).	
Binary operators:			
$\psi U \phi$	$\psi \mathcal{U} \phi$	Until: $\psi$ has to hold at least until $\phi$ , which holds at the current or a future position.	
$\psi R \phi$	$\psi \mathcal{R} \phi$	Release: $\phi$ has to be true until and including the point where $\psi$ first becomes true; if $\psi$ never becomes true, $\phi$ must remain true forever.	

## Model checking problem:

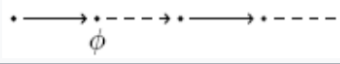
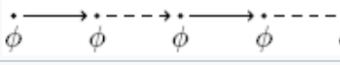
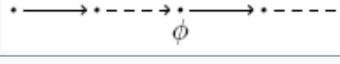
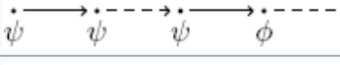
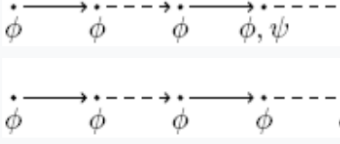
Given a model M and an LTL formula  $\varphi$ , all traces of M must satisfy  $\varphi$

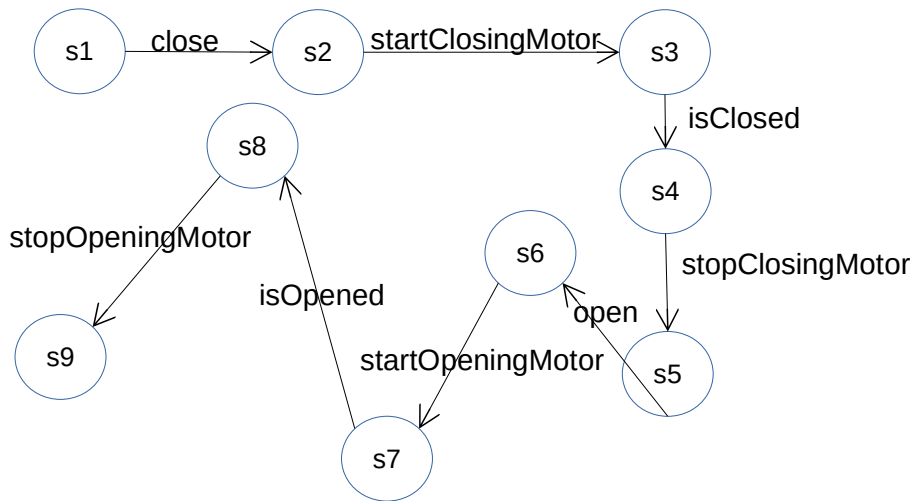
Given a transition system T and an LTL property, determine if T is a model for p

Textual	Symbolic	Explanation	Diagram
<b>Unary operators:</b>			
$X \phi$	$\bigcirc \phi$	neXt: $\phi$ has to hold at the next state.	
$G \phi$	$\square \phi$	Globally: $\phi$ has to hold on the entire subsequent path.	
$F \phi$	$\diamond \phi$	Finally: $\phi$ eventually has to hold (somewhere on the subsequent path).	
<b>Binary operators:</b>			
$\psi U \phi$	$\psi \mathcal{U} \phi$	Until: $\psi$ has to hold at least until $\phi$ , which holds at the current or a future position.	
$\psi R \phi$	$\psi \mathcal{R} \phi$	Release: $\phi$ has to be true until and including the point where $\psi$ first becomes true; if $\psi$ never becomes true, $\phi$ must remain true forever.	

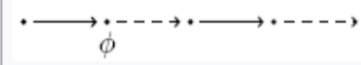
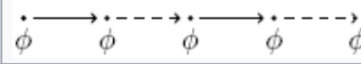
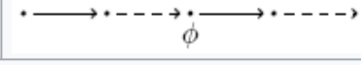
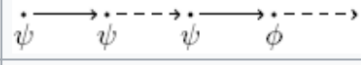
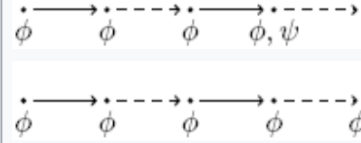


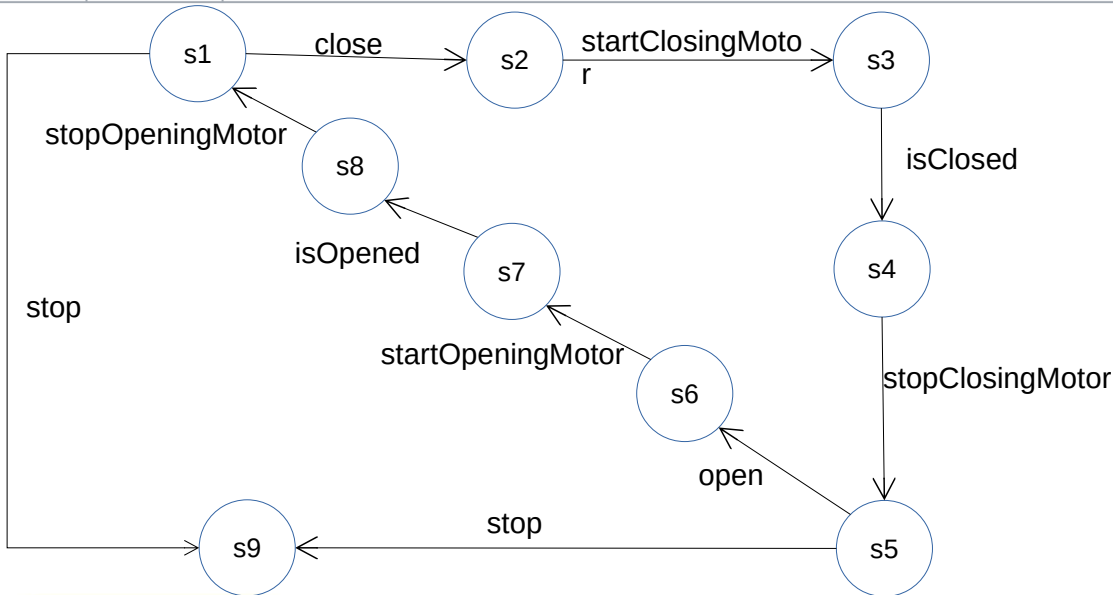
$\models \square (close \Rightarrow \diamond isClosed) ?$

Textual	Symbolic	Explanation	Diagram
Unary operators:			
$X \phi$	$\bigcirc \phi$	neXt: $\phi$ has to hold at the next state.	
$G \phi$	$\square \phi$	Globally: $\phi$ has to hold on the entire subsequent path.	
$F \phi$	$\diamond \phi$	Finally: $\phi$ eventually has to hold (somewhere on the subsequent path).	
Binary operators:			
$\psi U \phi$	$\psi \mathcal{U} \phi$	Until: $\psi$ has to hold at least until $\phi$ , which holds at the current or a future position.	
$\psi R \phi$	$\psi \mathcal{R} \phi$	Release: $\phi$ has to be true until and including the point where $\psi$ first becomes true; if $\psi$ never becomes true, $\phi$ must remain true forever.	



$\models \square (close \Rightarrow \diamond isClosed) ?$

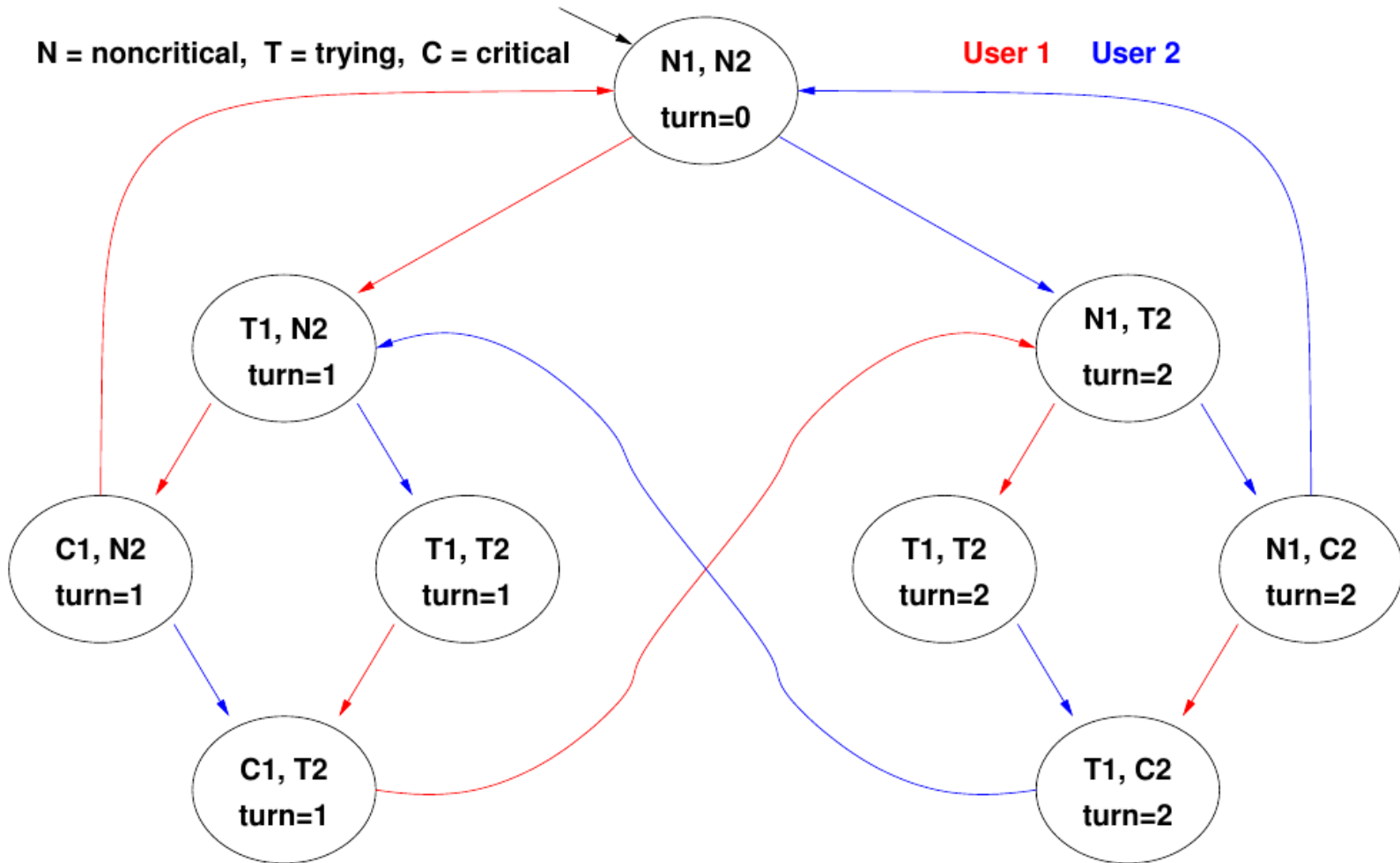
Textual	Symbolict	Explanation	Diagram
Unary operators:			
$X \phi$	$\bigcirc \phi$	neXt: $\phi$ has to hold at the next state.	
$G \phi$	$\square \phi$	Globally: $\phi$ has to hold on the entire subsequent path.	
$F \phi$	$\diamond \phi$	Finally: $\phi$ eventually has to hold (somewhere on the subsequent path).	
Binary operators:			
$\psi U \phi$	$\psi \mathcal{U} \phi$	Until: $\psi$ has to hold at least until $\phi$ , which holds at the current or a future position.	
$\psi R \phi$	$\psi \mathcal{R} \phi$	Release: $\phi$ has to be true until and including the point where $\psi$ first becomes true; if $\psi$ never becomes true, $\phi$ must remain true forever.	



$\models \square (close \Rightarrow \diamond isClosed) ?$

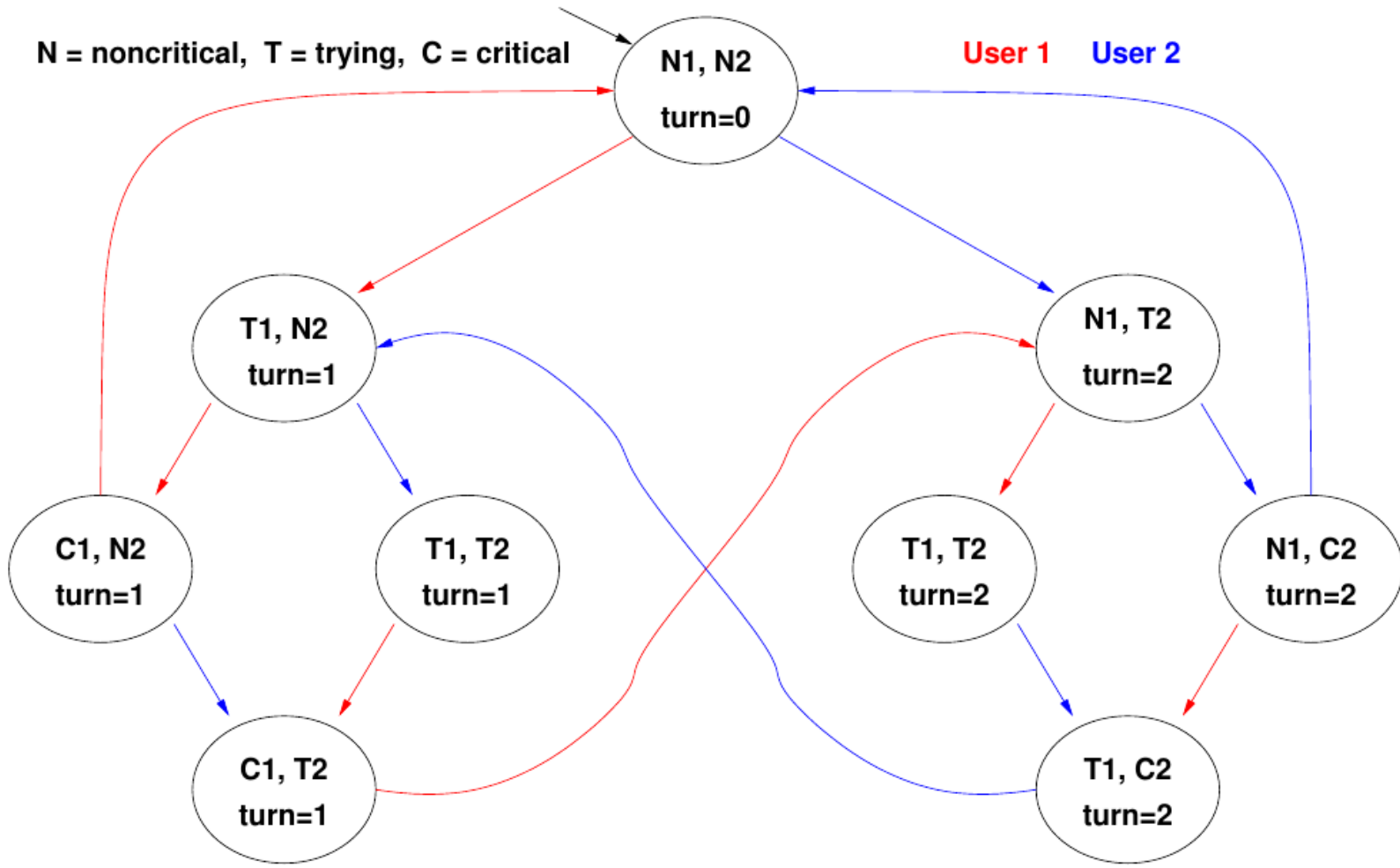
# Mutual exclusion system

taken from lectures of Alessandro Artale



# Mutual exclusion system

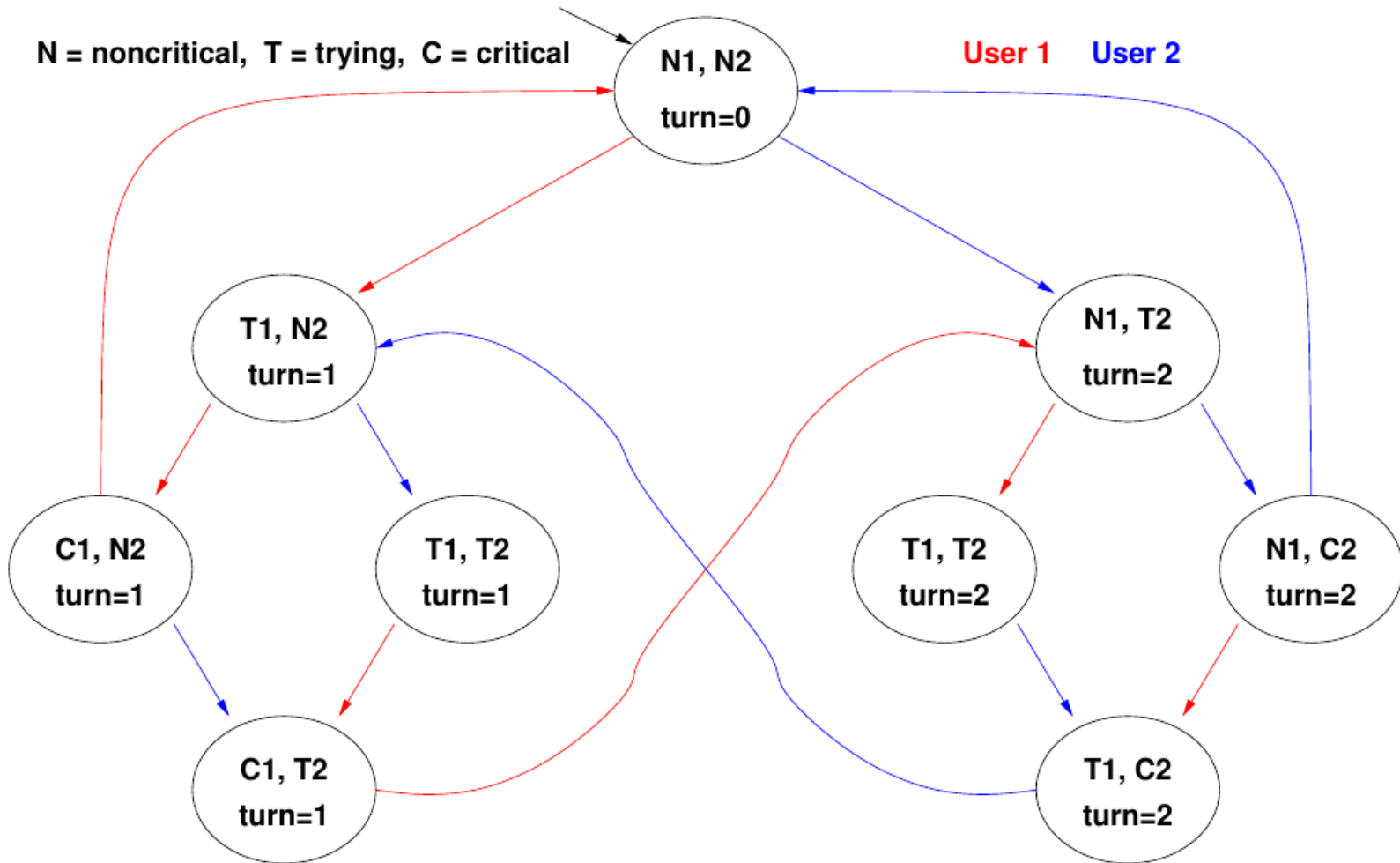
taken from lectures of Alessandro Artale



$$\models \square \neg (C_1 \wedge C_2) ?$$

# Mutual exclusion system

taken from lectures of Alessandro Artale

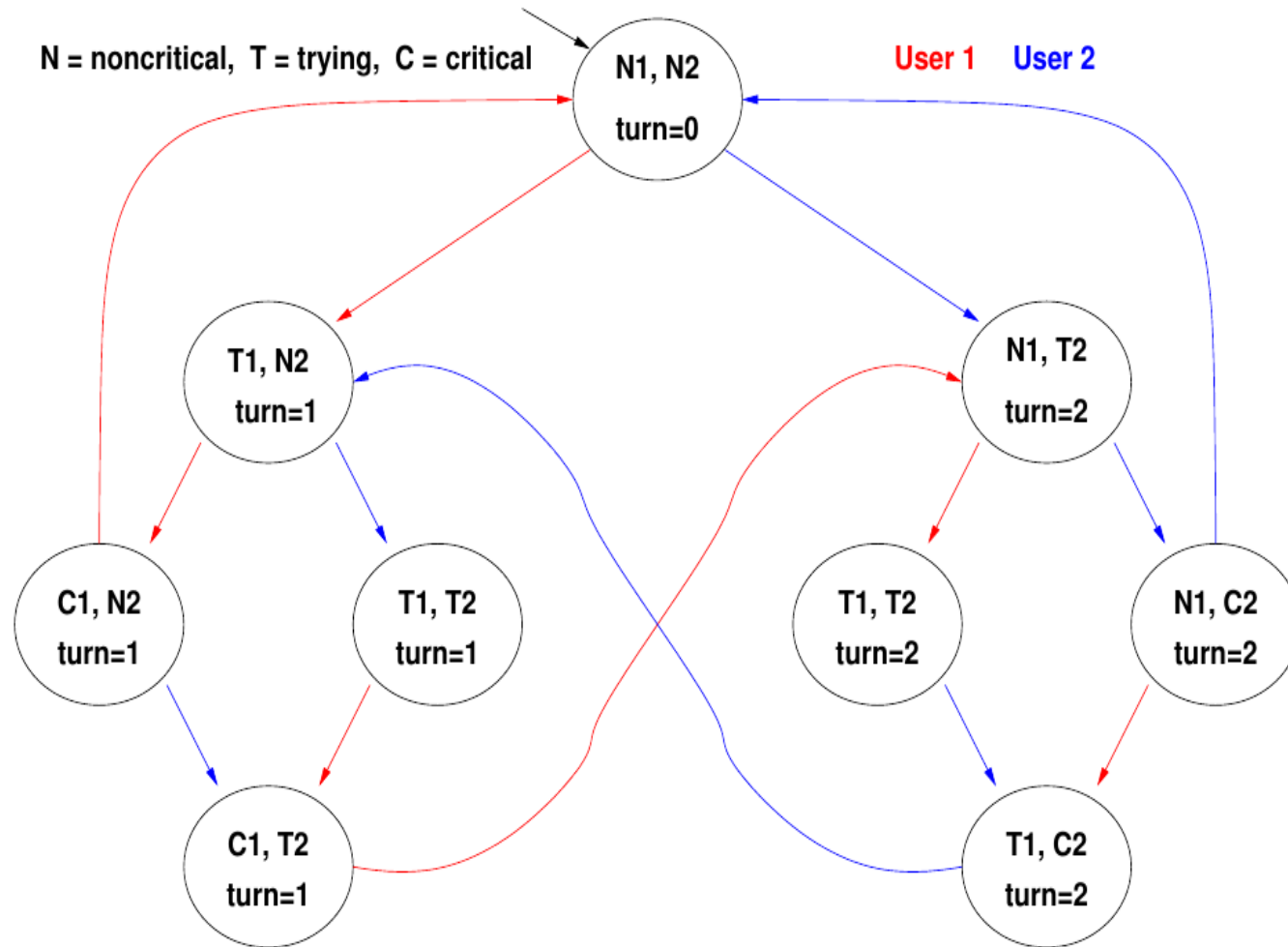


$$\models \diamond C_1 ?$$



# Mutual exclusion system

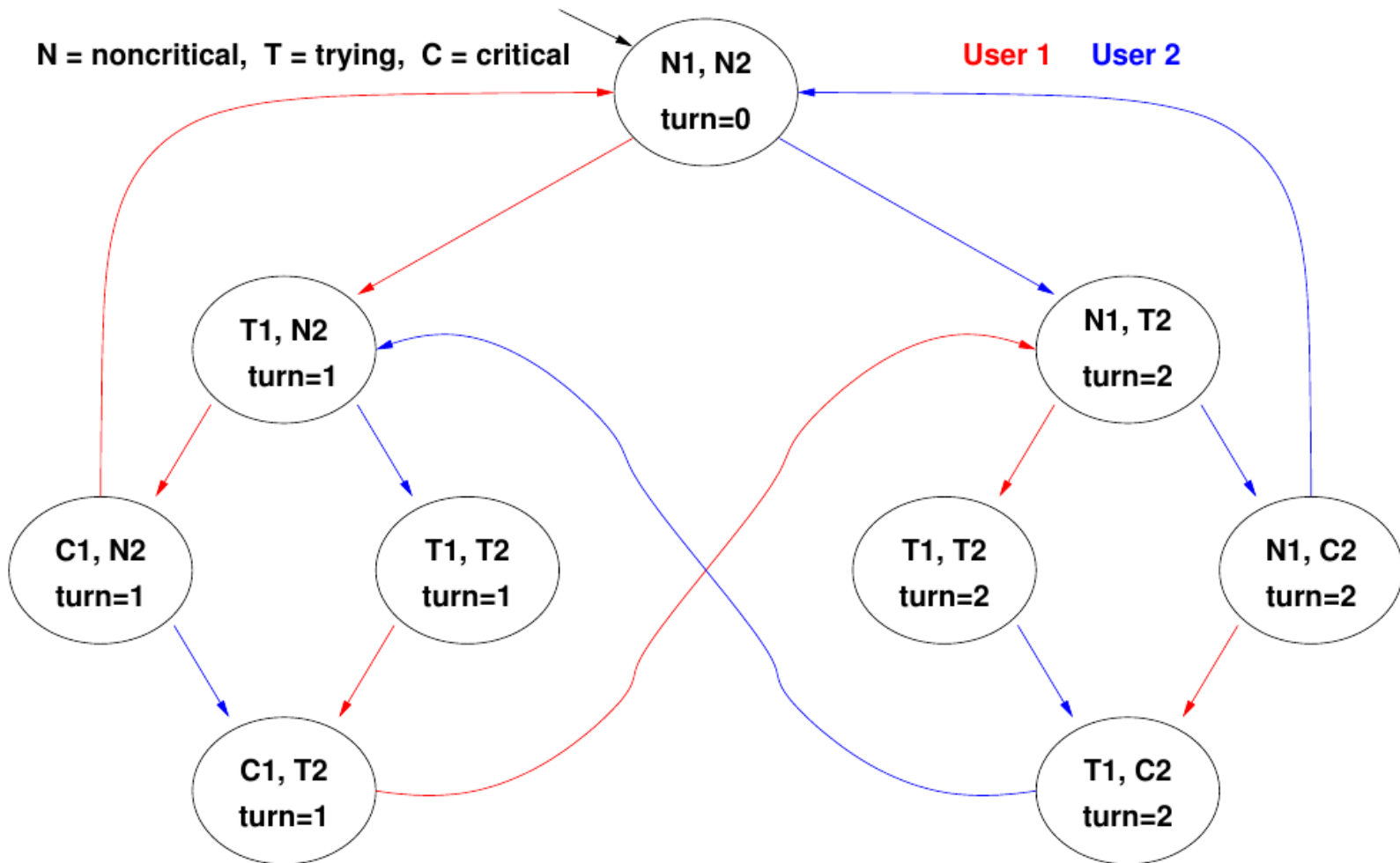
taken from lectures of Alessandro Artale



$$\models \square (T_1 \Rightarrow \diamond C_1) ?$$

# Mutual exclusion system

taken from lectures of Alessandro Artale



Il existe de nombreuses formes en langage naturel contrainc des logiques temporelles (pattern de Dwyer, de Cheng, de Dhaussy)