Finite State Machine, state charts

model checking ?

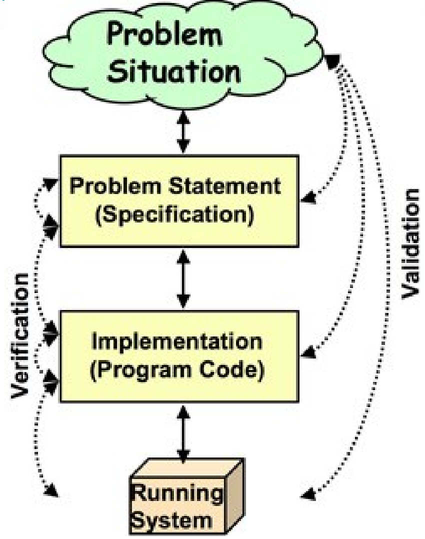
Finite State Machine, state charts and implementations

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 ?

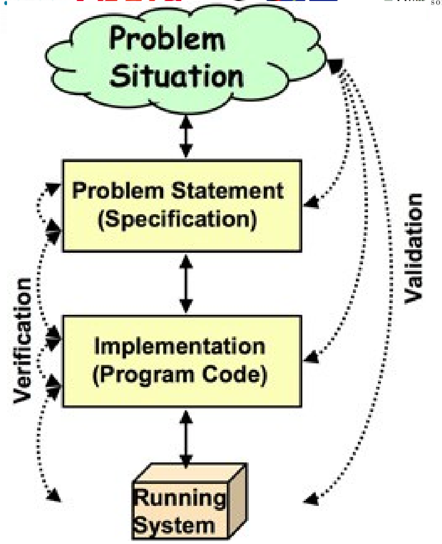


Problem Situation
Problem Statement (Specification)
Implementation (Program Code)
Running System
Verification
Validation

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

- 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, rafinement correct, etc)
  - Validation:

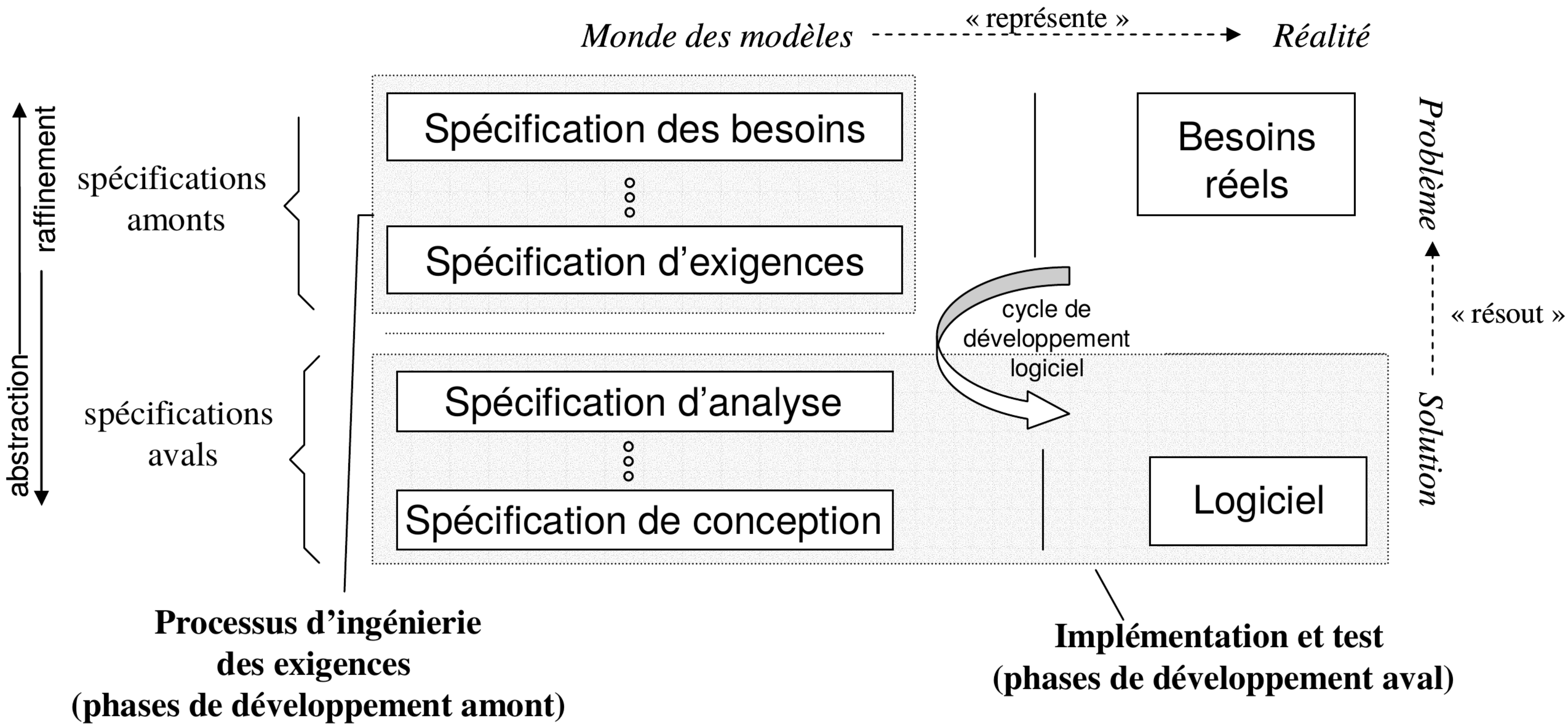
V&V ?


Problem Situation
Problem Statement (Specification)
Implementation (Program Code)
Running System
Verification
Validation

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

- 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, rafinement 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 ?

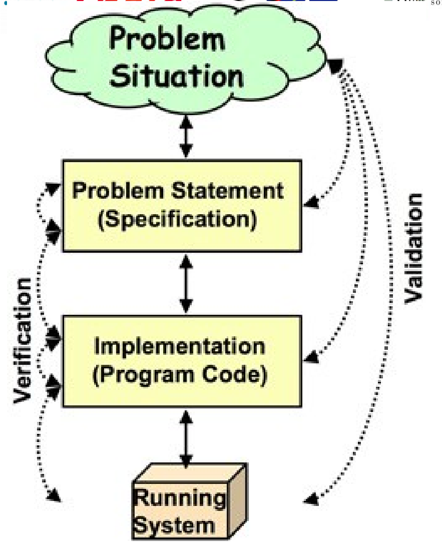


Monde des modèles
« représente »
Réalité
raffinement
abstraction
spécifications amonts
spécifications avals
Spécification des besoins
Spécification d'exigences
Spécification d'analyse
Spécification de conception
cycle de développement logiciel
Besoins réels
Logiciel
Problème
« résout »
Solution
Processus d'ingénierie des exigences (phases de développement amont)
Implémentation et test (phases de développement aval)

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 ?


Problem Situation
Problem Statement (Specification)
Implementation (Program Code)
Running System
Verification
Validation

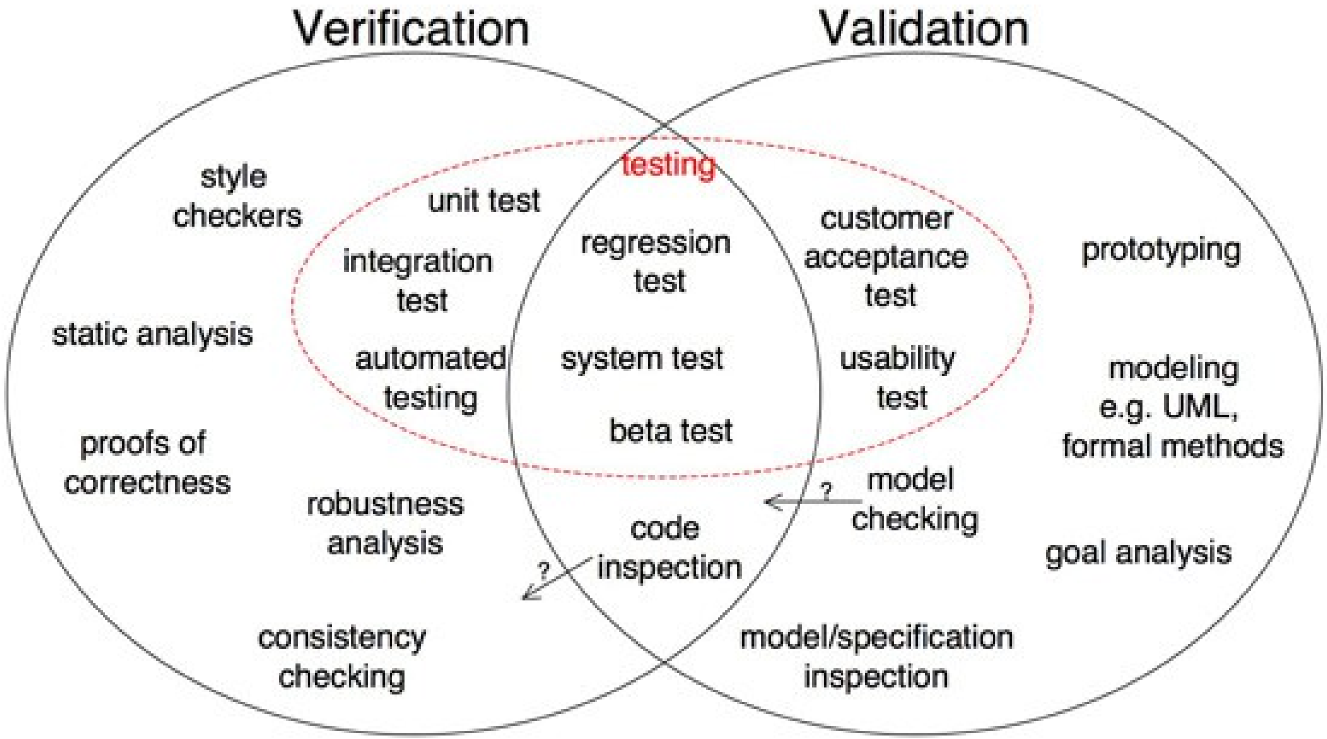
Picture taken from Steve EasterBrook (http://www.easterbrook.ca/steve/2010/11/the-difference-between-verification-and-validation/)

- 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, rafinement 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**".

V&V ?

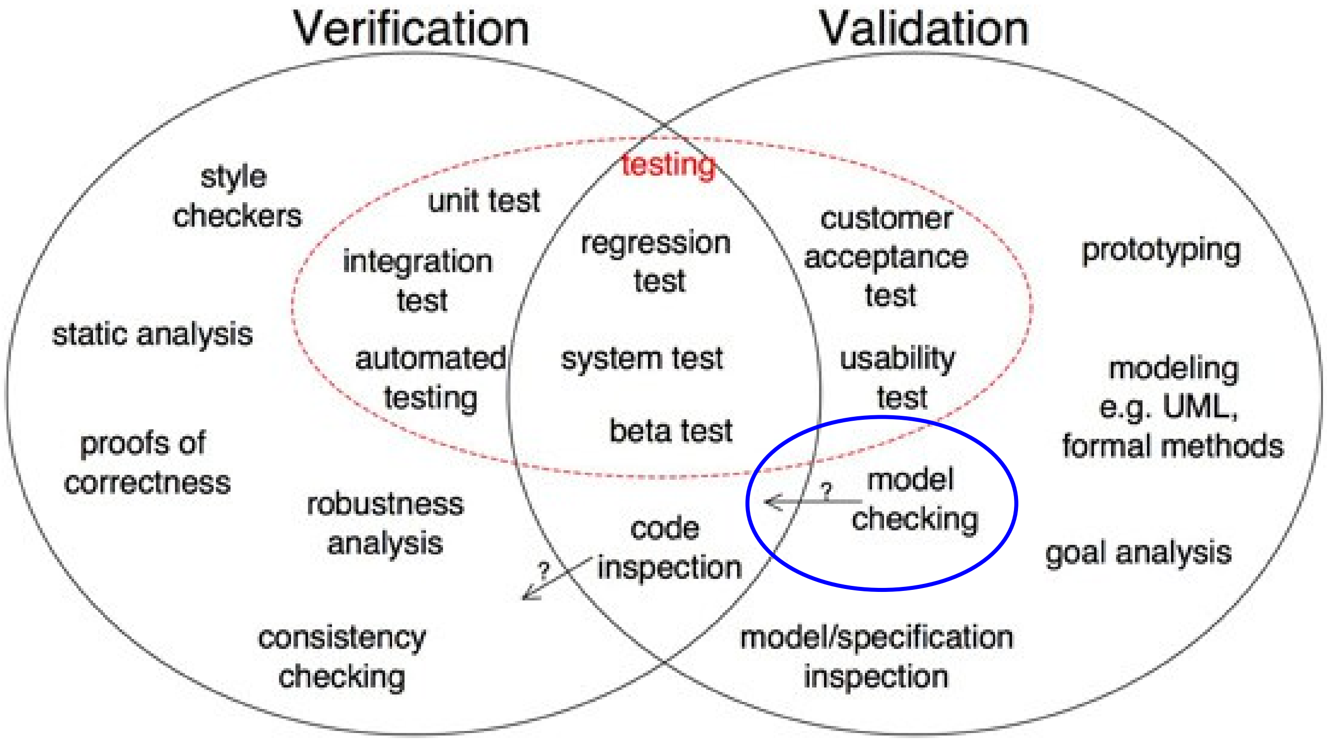
- Verification and Validation



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

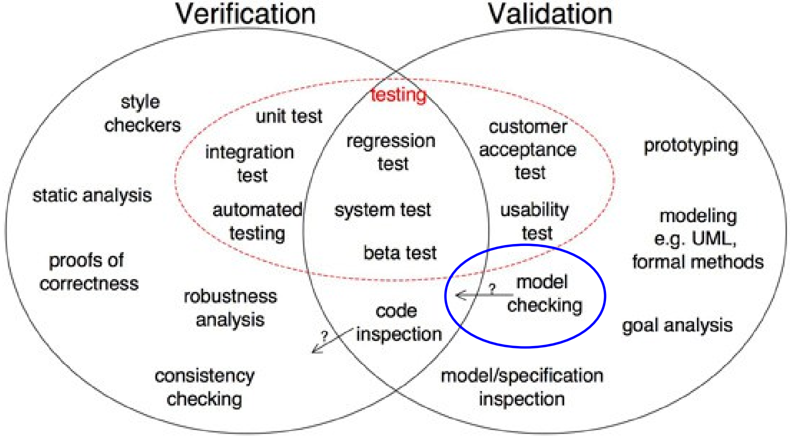
V&V ?

- Verification and Validation



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

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.

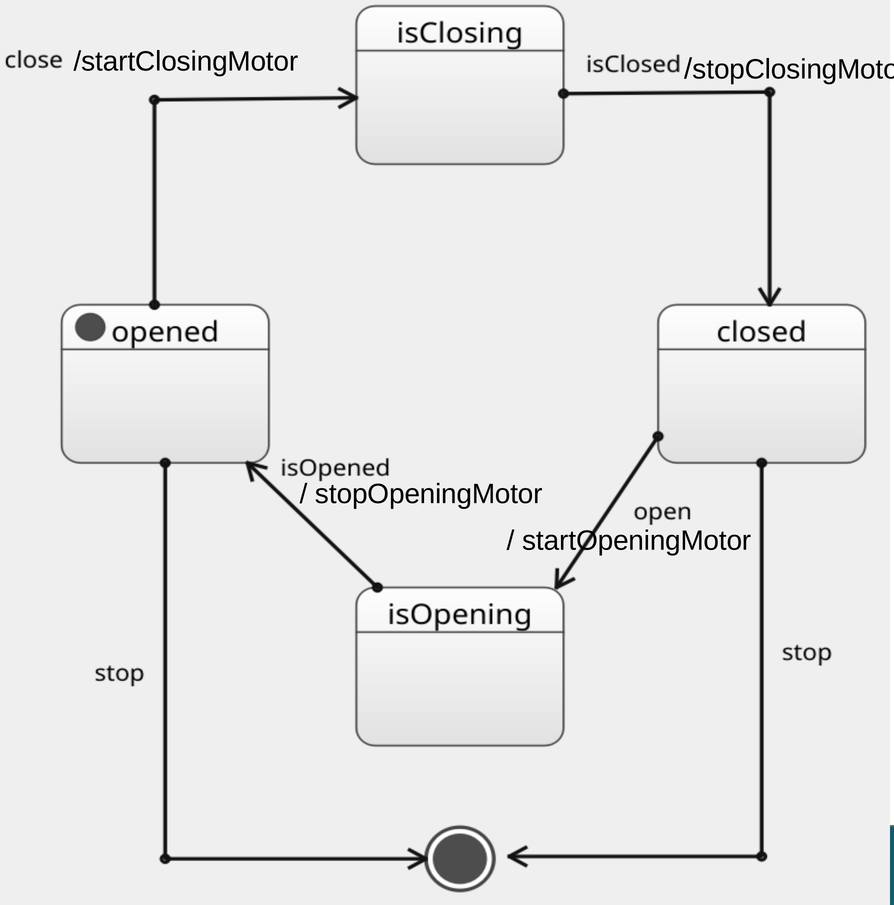
    → taux de couverture ? Nombre de tests ?

  - 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 ?

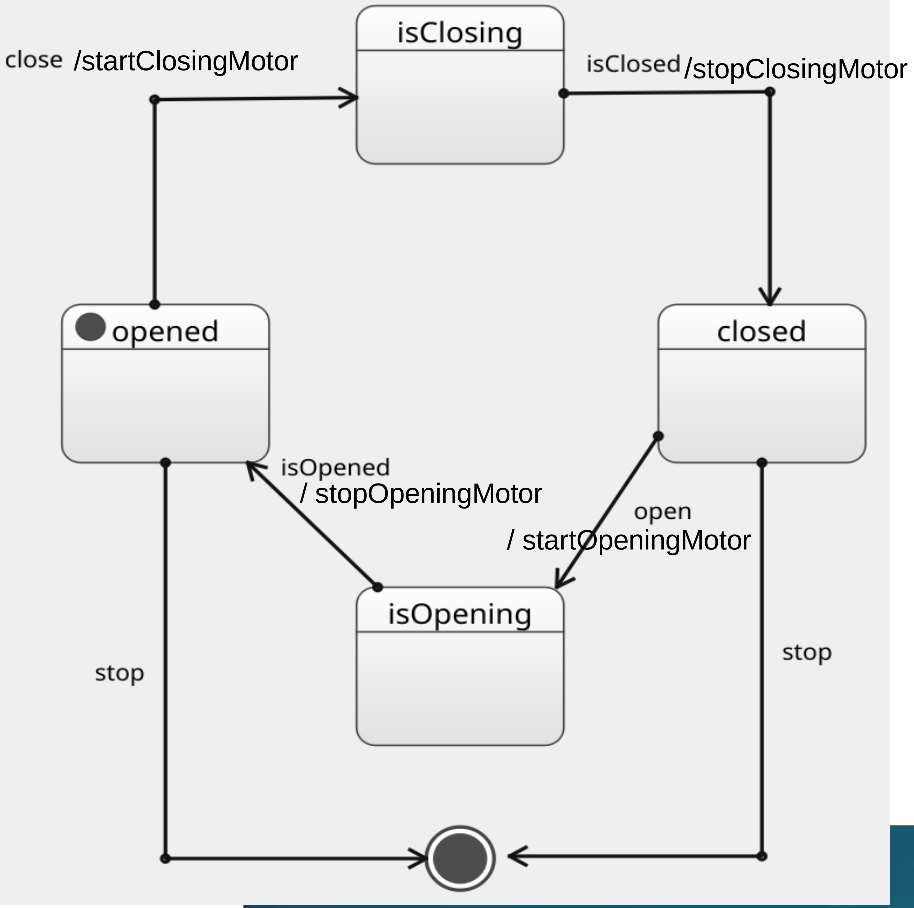
trace (run), state space and real life

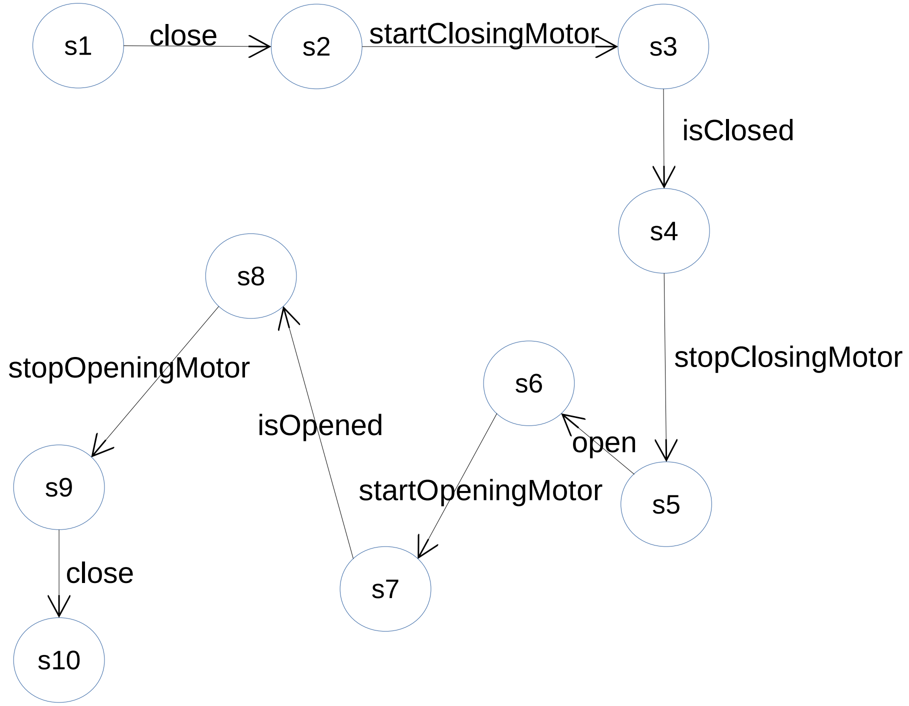
- Each test or run of a program creates a trace.
- Speaking automata, a trace is a, possibly infinite, sequence of states and transitions. It is a word accepted by the automata.



trace (run), state space and real life

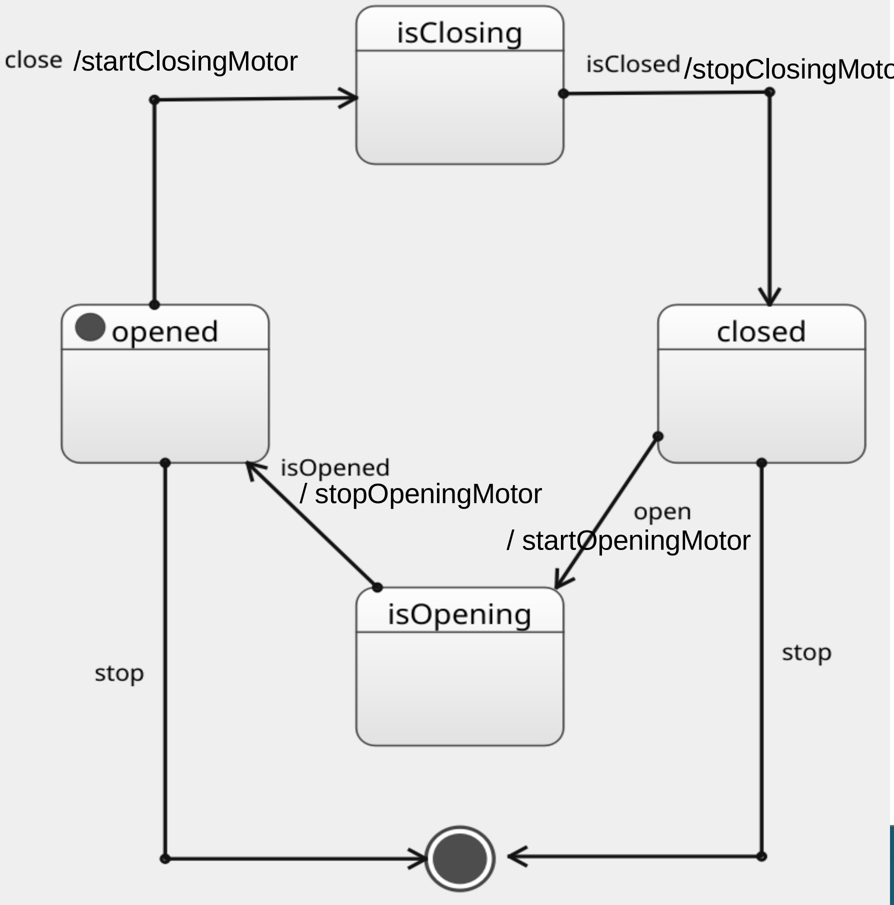
- Each test or run of a program creates a trace.
- Speaking automata, a trace is a, possibly infinite, sequence of states and transitions. It is a word accepted by the automata.

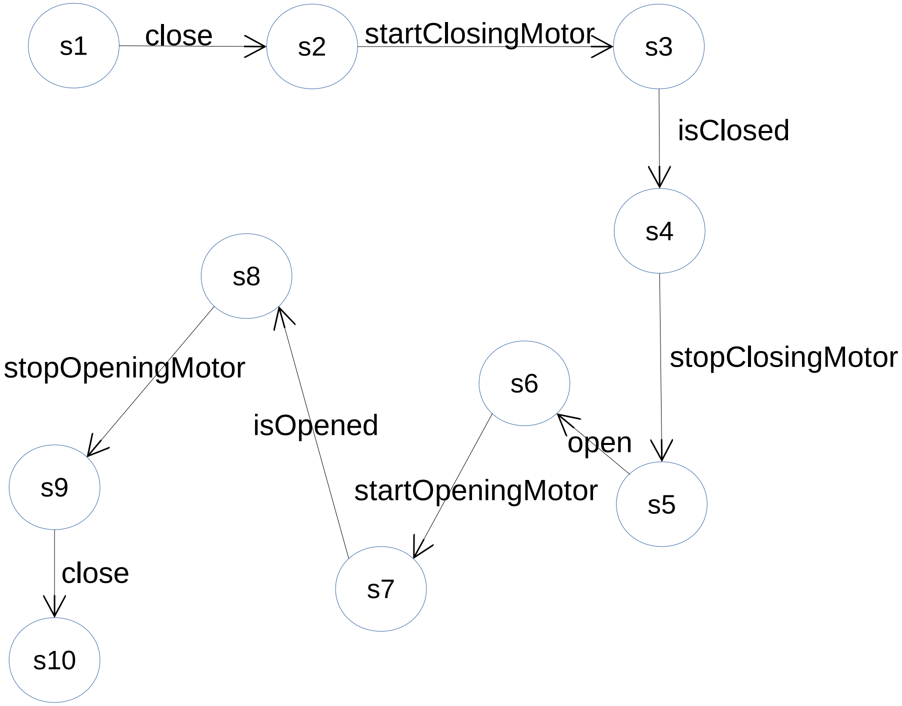




trace (run), state space and real life

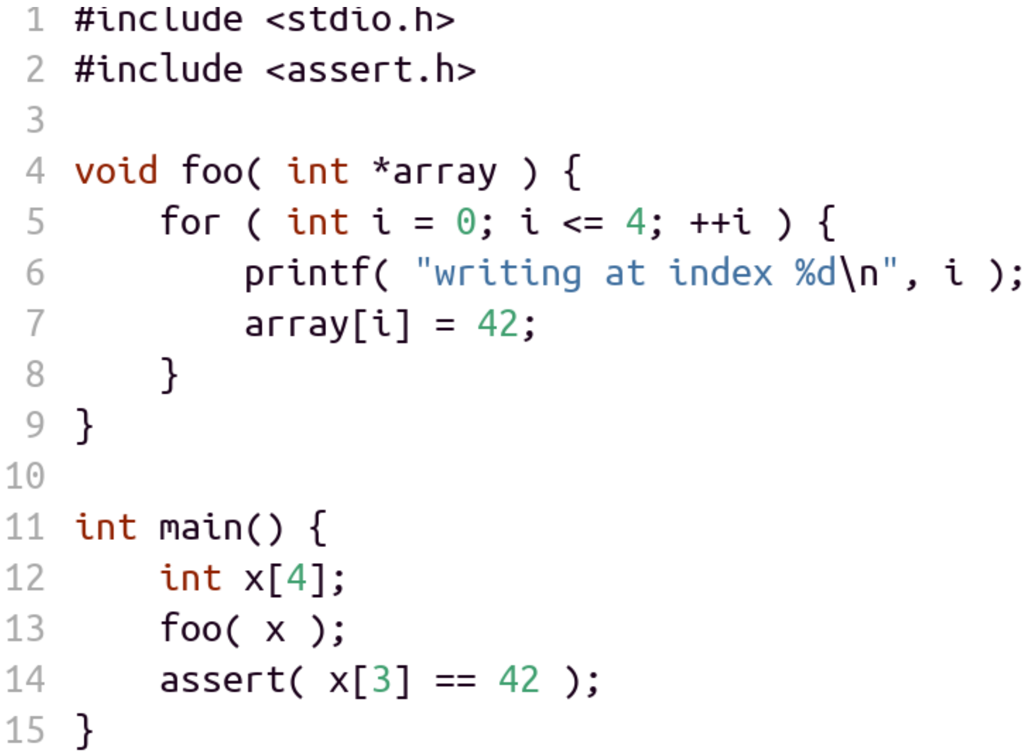
- Each test or run of a program creates a trace.
- Speaking automata, a trace is a, possibly infinite, sequence of states and transitions. It is a word accepted by the automata. It is actually a bit more since there is possibly information in states.



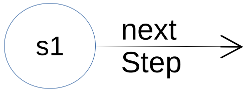


trace (run), state space and real life

- 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 sequence of states and transitions :)

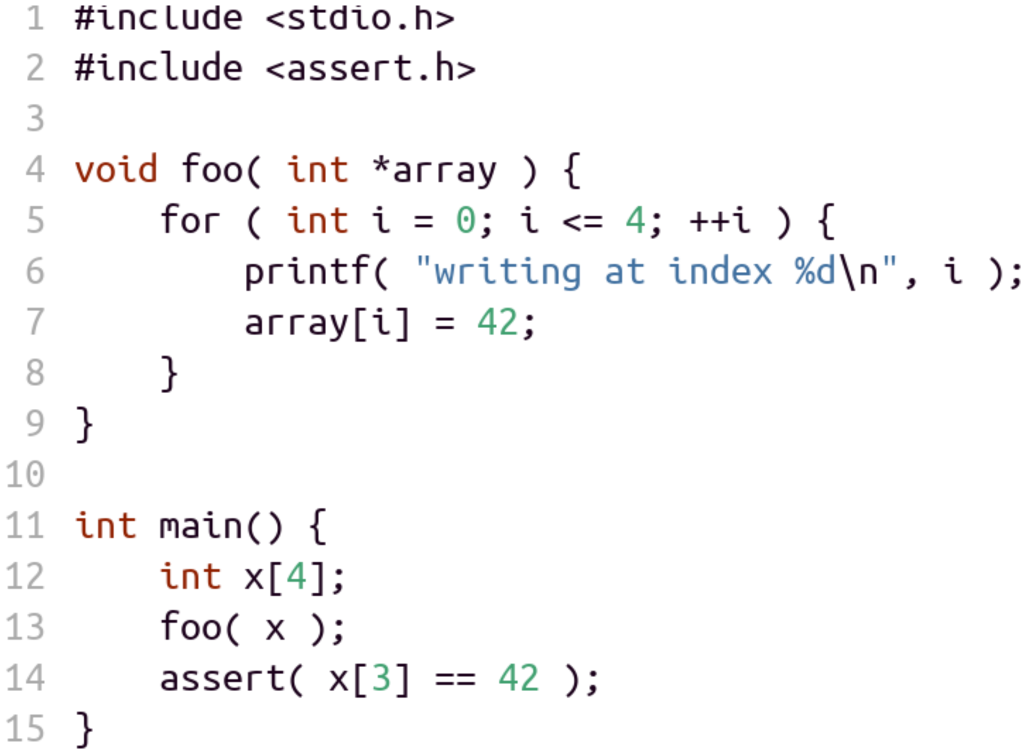


```c
#include <stdio.h>
#include <assert.h>

void foo( int *array ) {
    for ( int i = 0; i <= 4; ++i ) {
        printf( "writing at index %d\n", i );
        array[i] = 42;
    }
}

int main() {
    int x[4];
    foo( x );
    assert( x[3] == 42 );
}
```

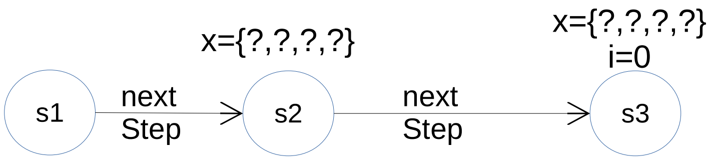


trace (run), state space and real life

- 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 :)

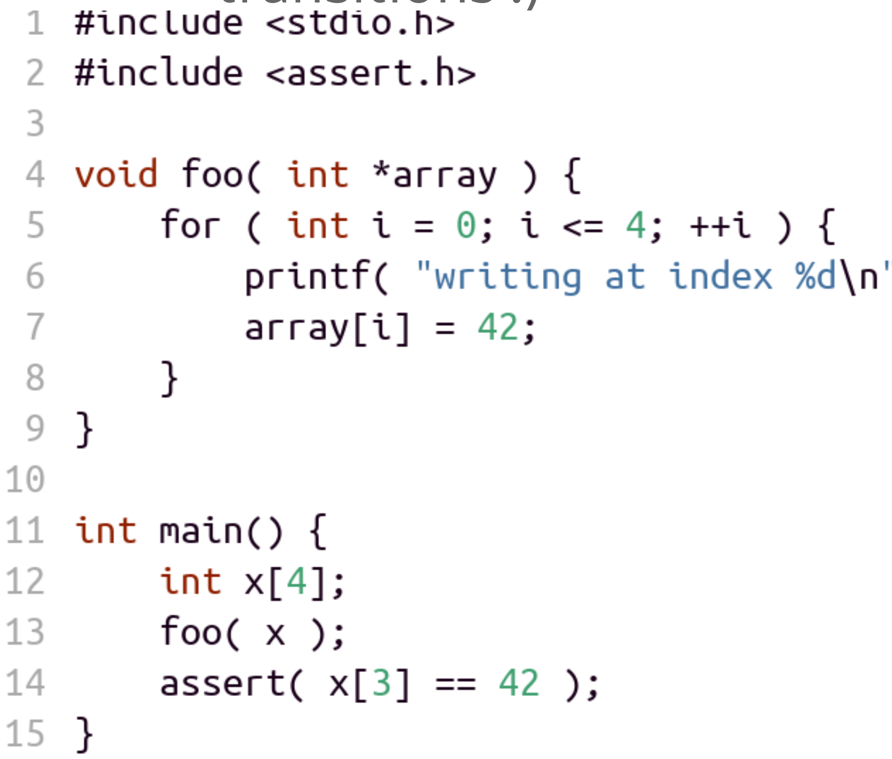


```c
#include <stdio.h>
#include <assert.h>

void foo( int *array ) {
    for ( int i = 0; i <= 4; ++i ) {
        printf( "writing at index %d\n", i );
        array[i] = 42;
    }
}

int main() {
    int x[4];
    foo( x );
    assert( x[3] == 42 );
}
```

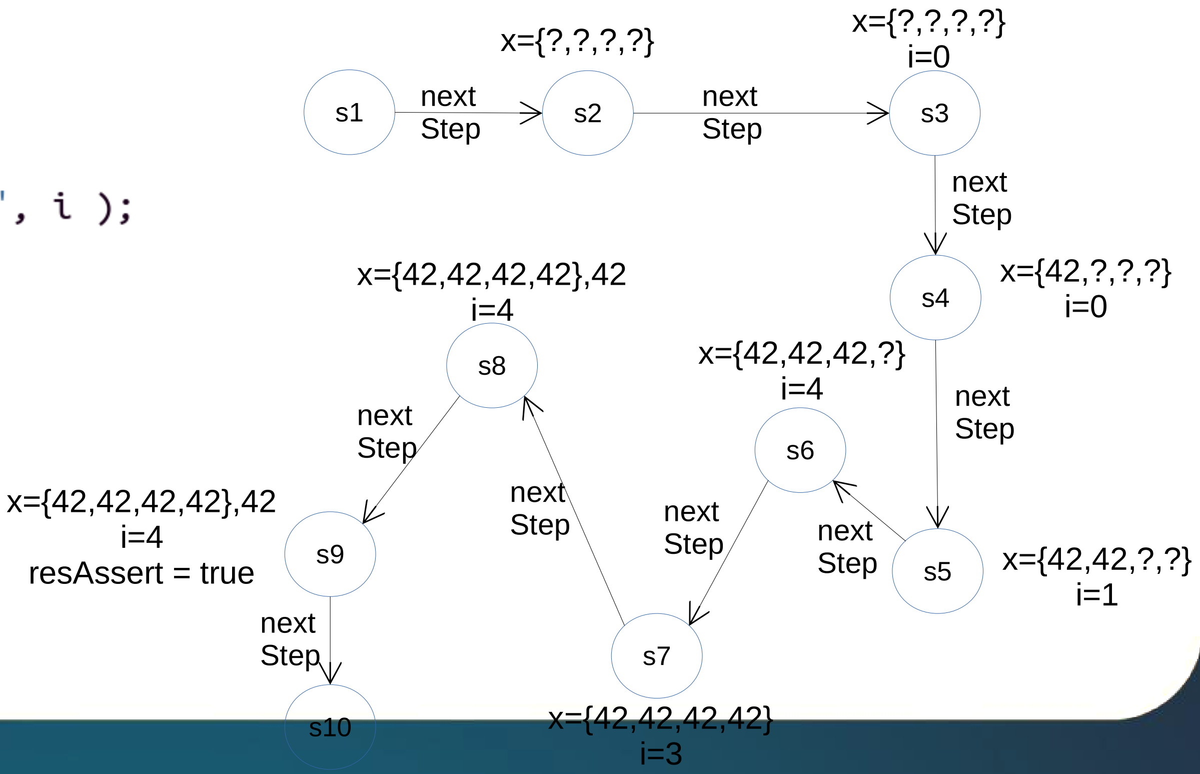


trace (run), state space and real life

- 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 :)

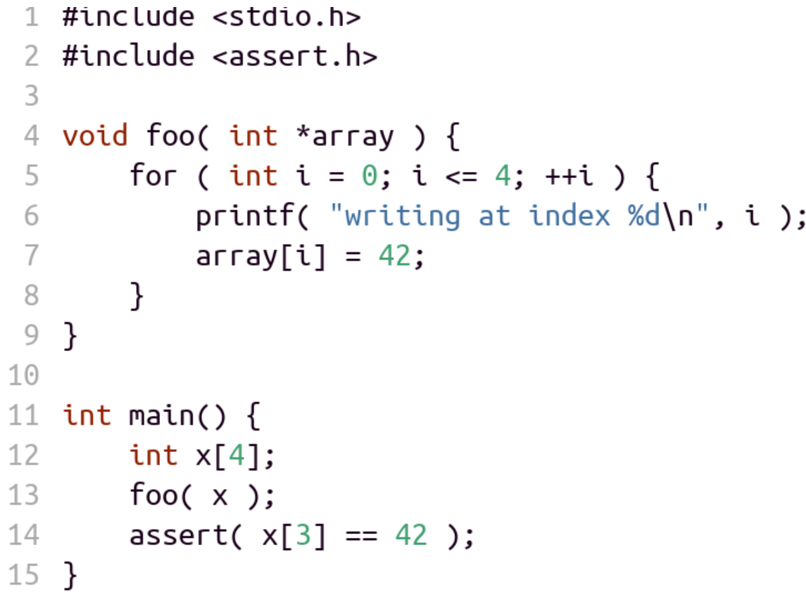


```c
#include <stdio.h>
#include <assert.h>

void foo( int *array ) {
    for ( int i = 0; i <= 4; ++i ) {
        printf( "writing at index %d\n", i );
        array[i] = 42;
    }
}

int main() {
    int x[4];
    foo( x );
    assert( x[3] == 42 );
}
```

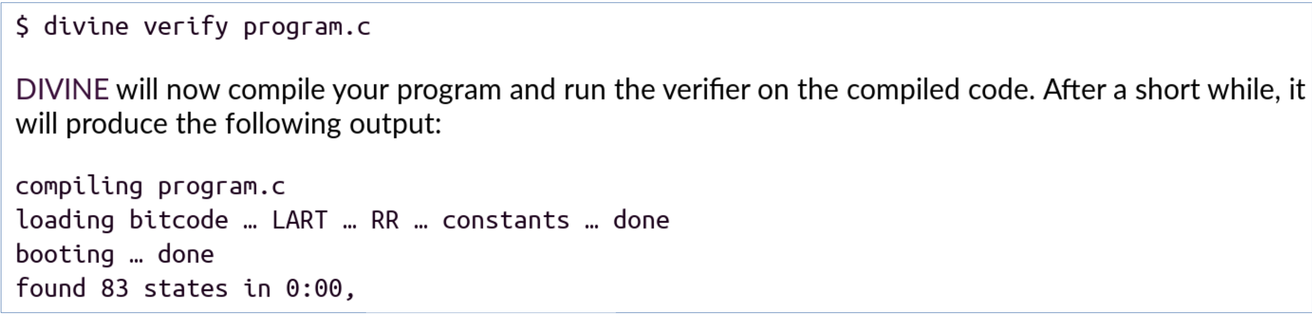


trace (run), state space and real life

- 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 :)



```c
#include <stdio.h>
#include <assert.h>

void foo( int *array ) {
    for ( int i = 0; i <= 4; ++i ) {
        printf( "writing at index %d\n", i );
        array[i] = 42;
    }
}

int main() {
    int x[4];
    foo( x );
    assert( x[3] == 42 );
}
```



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

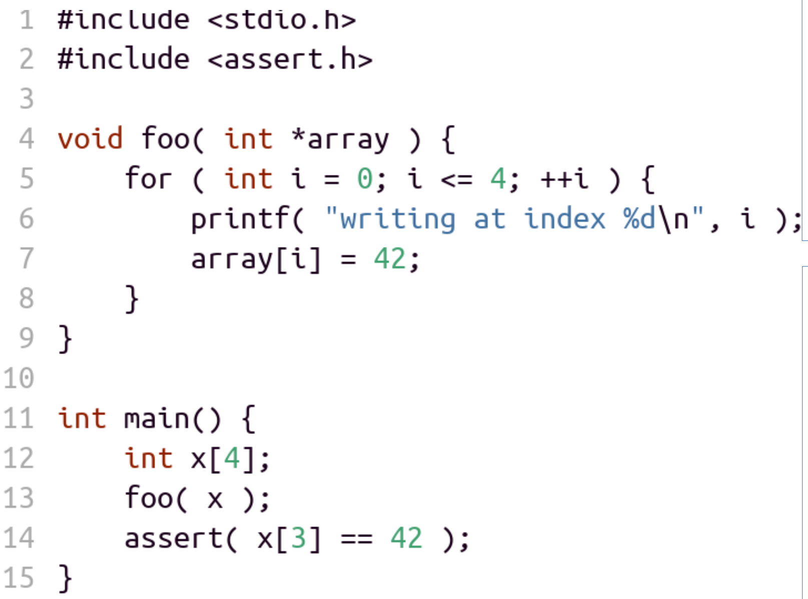
DIVINE will now compile your program and run the verifier on the compiled code. After a short while, it will produce the following output:

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

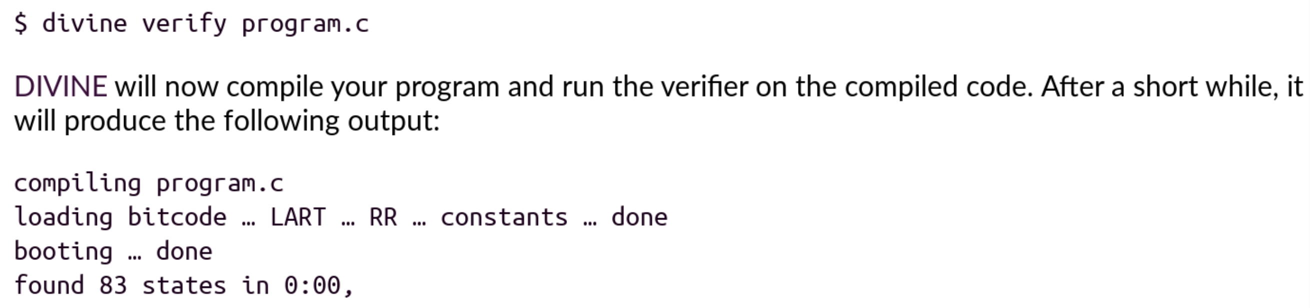
http://divine.fi.muni.cz/manual.html

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 :)



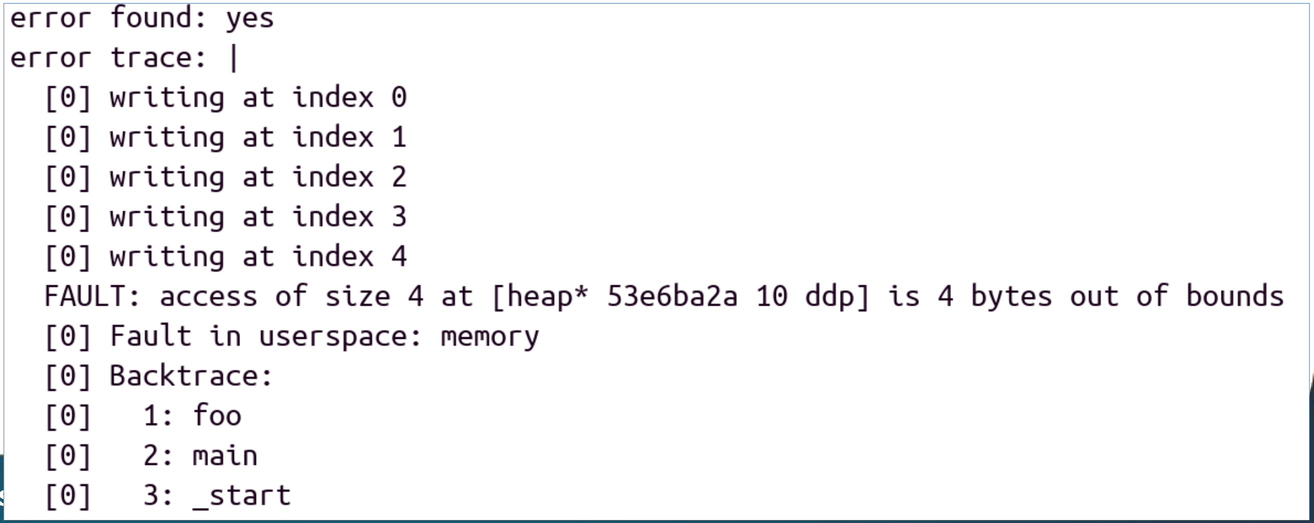
```c
#include <stdio.h>
#include <assert.h>

void foo( int *array ) {
    for ( int i = 0; i <= 4; ++i ) {
        printf( "writing at index %d\n", i );
        array[i] = 42;
    }
}

int main() {
    int x[4];
    foo( x );
    assert( x[3] == 42 );
}
```



```
$ 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,
```

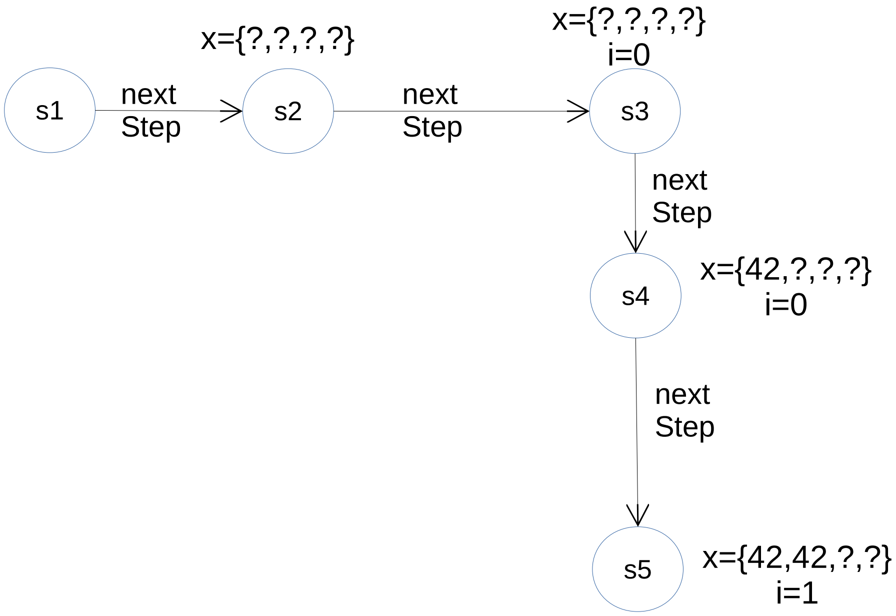


```
error found: yes
error trace: |
  [0] writing at index 0
  [0] writing at index 1
  [0] writing at index 2
  [0] writing at index 3
  [0] writing at index 4
  FAULT: access of size 4 at [heap* 53e6ba2a 10 ddp] is 4 bytes out of bounds
  [0] Fault in userspace: memory
  [0] Backtrace:
  [0]   1: foo
  [0]   2: main
  [0]   3: _start
```

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

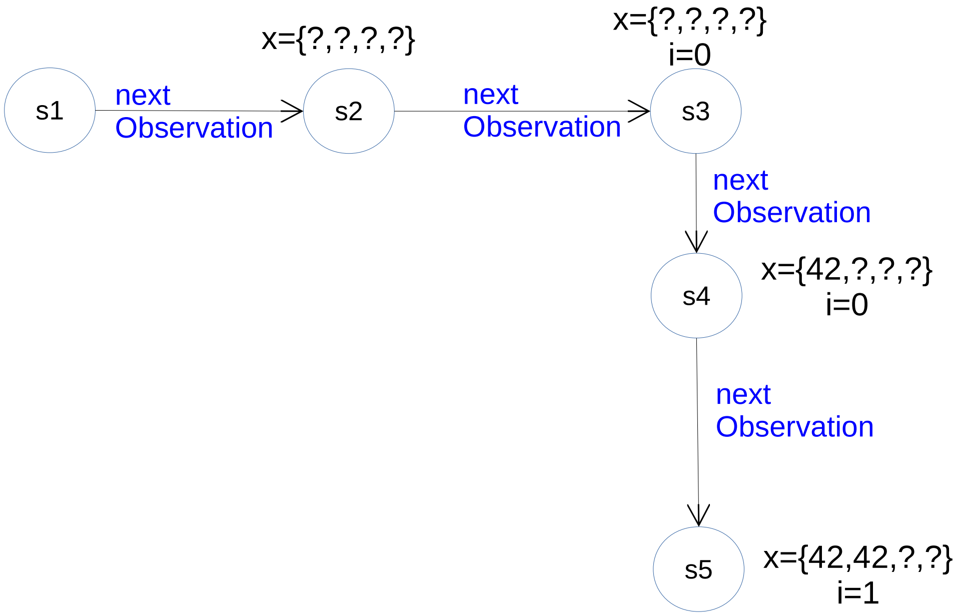
trace (run), state space and real life

- 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

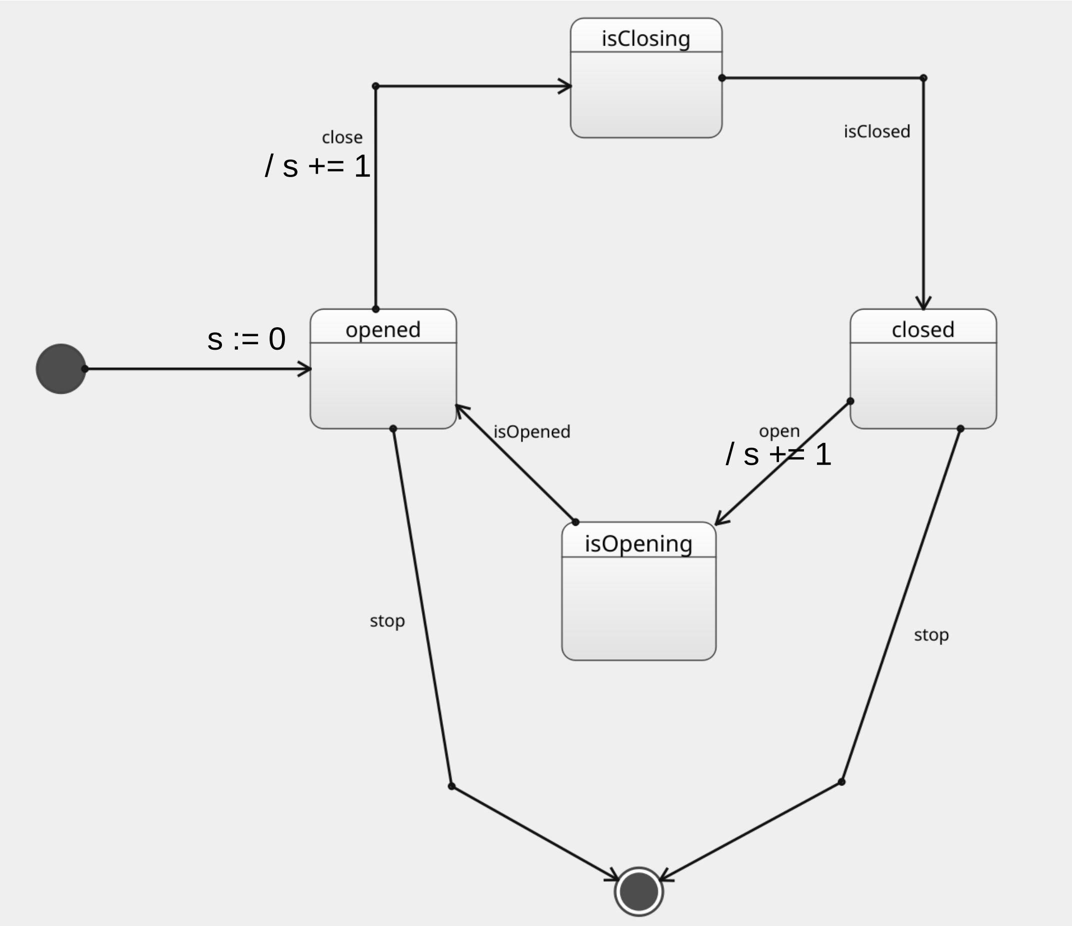
- Each test or run of a program creates a trace.
- Speaking automata, a trace is a, possibly infinite, sequence of states and transitions. It is a word accepted by the automata.
- Speaking code, a trace is a, possibly sequence of states and transitions :)

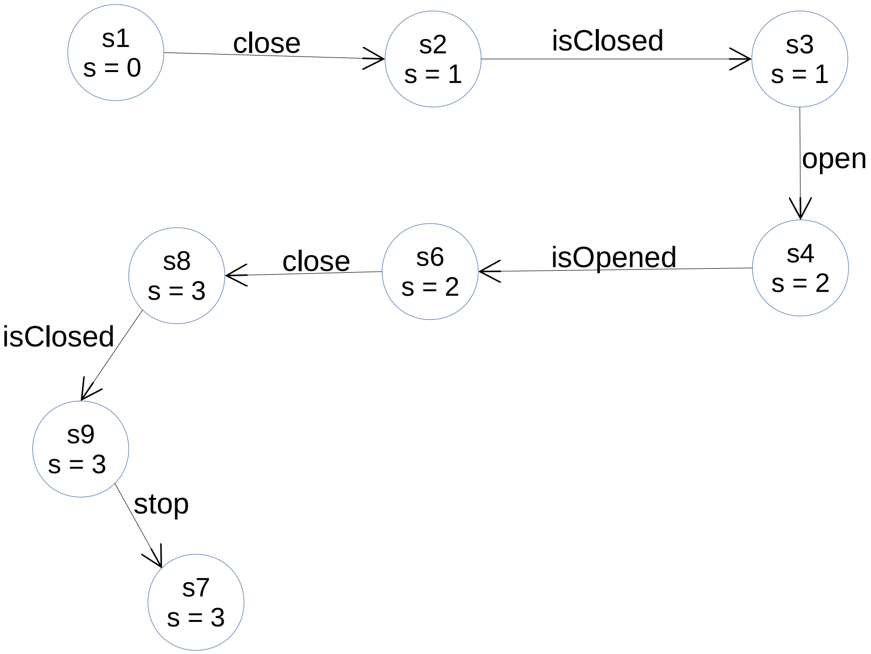


trace (run), state space and real life

To obtain traces:

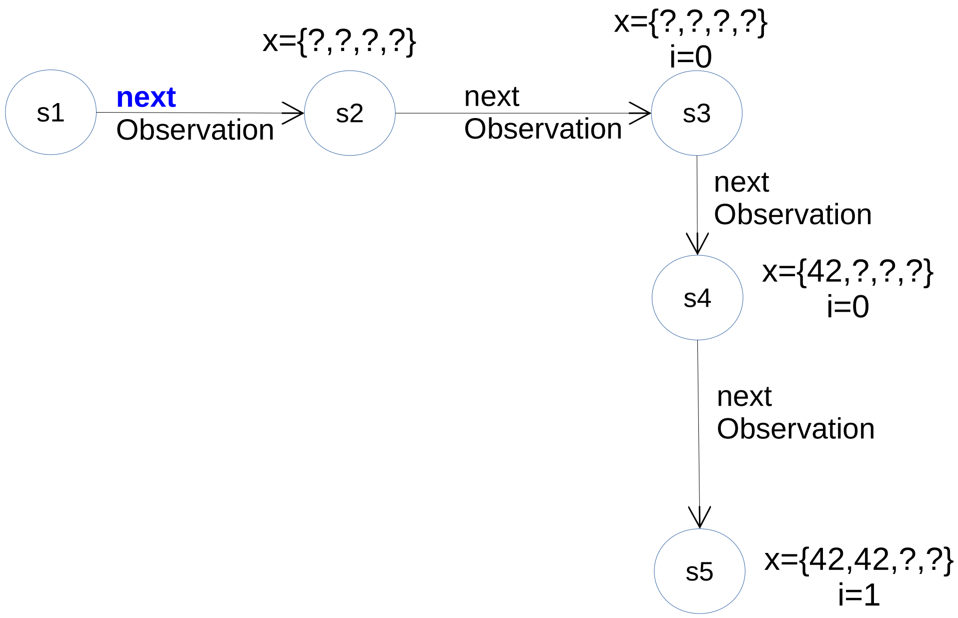
- Take your favorite programming/modeling language
- Equip it with discrete transition semantics (*e.g.*, S.O.S)
- Determine what should be observable events / conditions / execution states





Time

- All traces are dealing with time



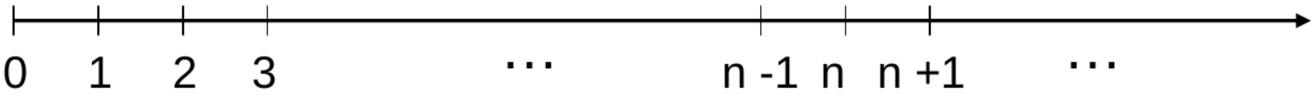
Time

- All traces are dealing with time

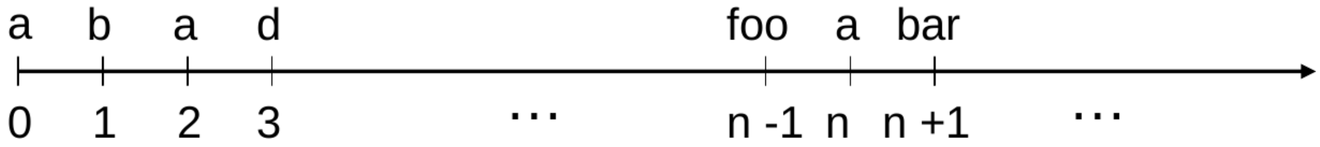
Time is discrete

Starts at 0

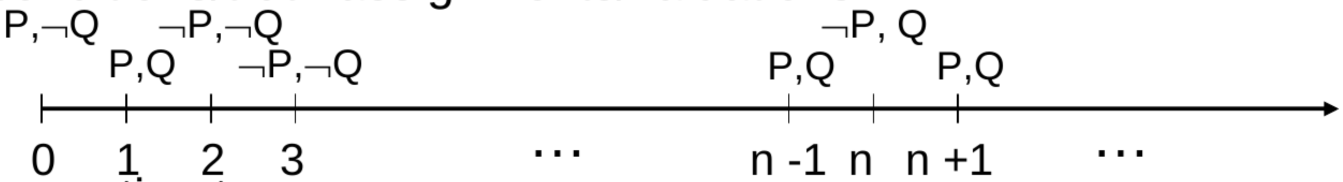
Goes on forever



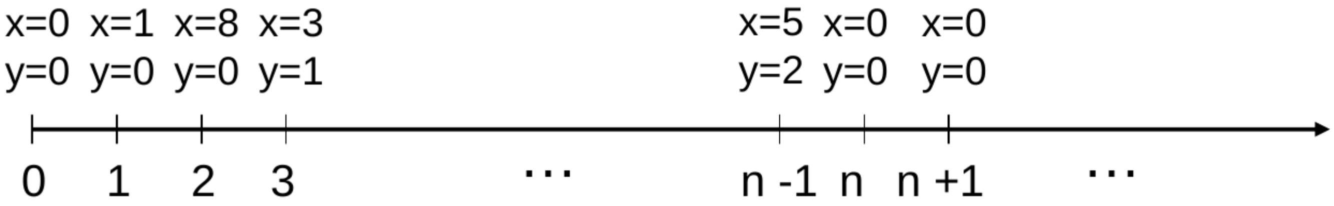
Time points decorated by events



Or conditions/truth assignments/valuations



Or execution traces



Taken from Mads Dam Theoretical Computer Science KTH, 2009

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

- 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 ask to take a mutex, 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 ask to take a mutex, 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 ask to take a mutex, 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 Booleénne.
- Deux classes principales: *Linear Temporal Logic* (LTL) et *Computational Tree Logic* (CTL)

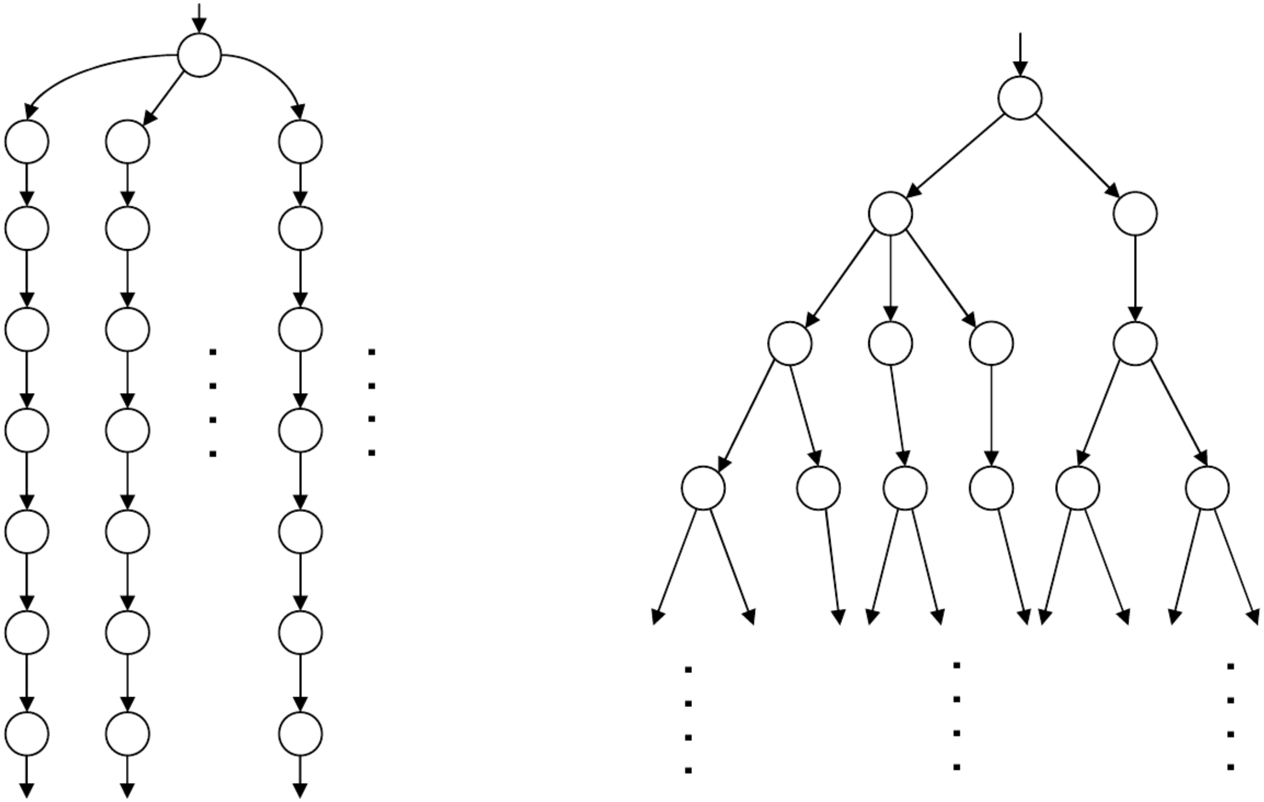
trace (run), state space and real life

- All traces are speaking about time

Branching Time Logic

Sets of paths?

Or computation tree?



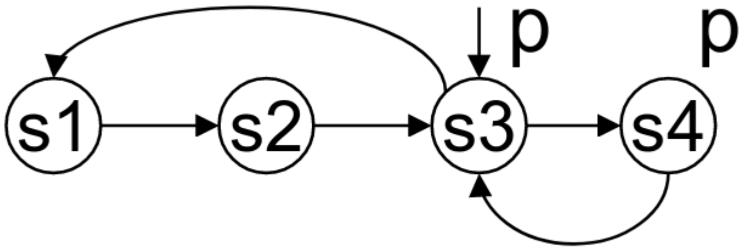
Taken from Mads Dam Theoretical Computer Science KTH, 2009

trace (run), state space and real life

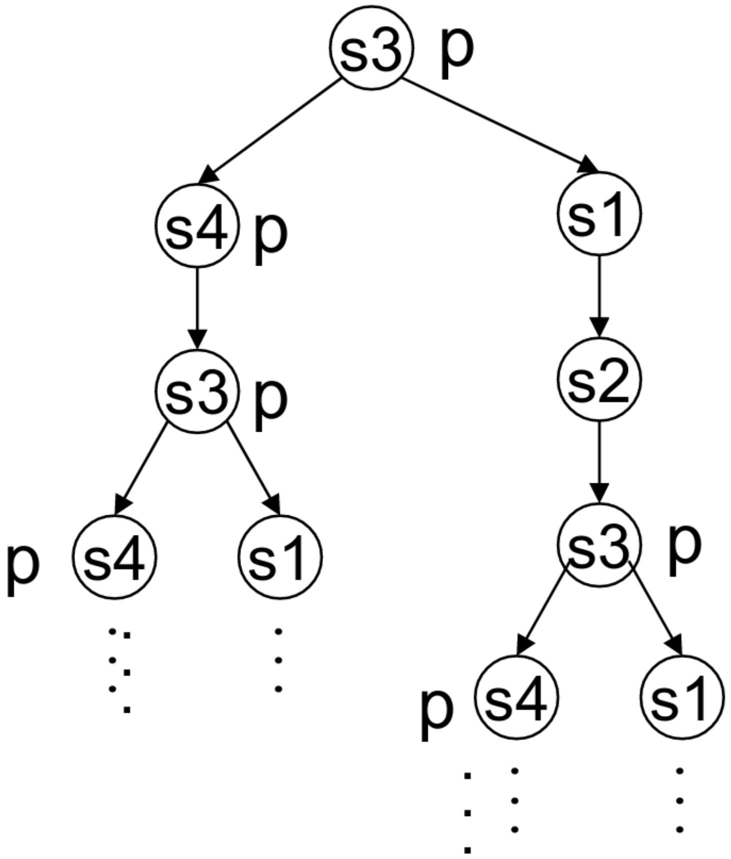
- All traces are speaking about time.
- When possible, a state space (also named transition system)

  represents in a finite way an infinite (set of) traces

Transition System



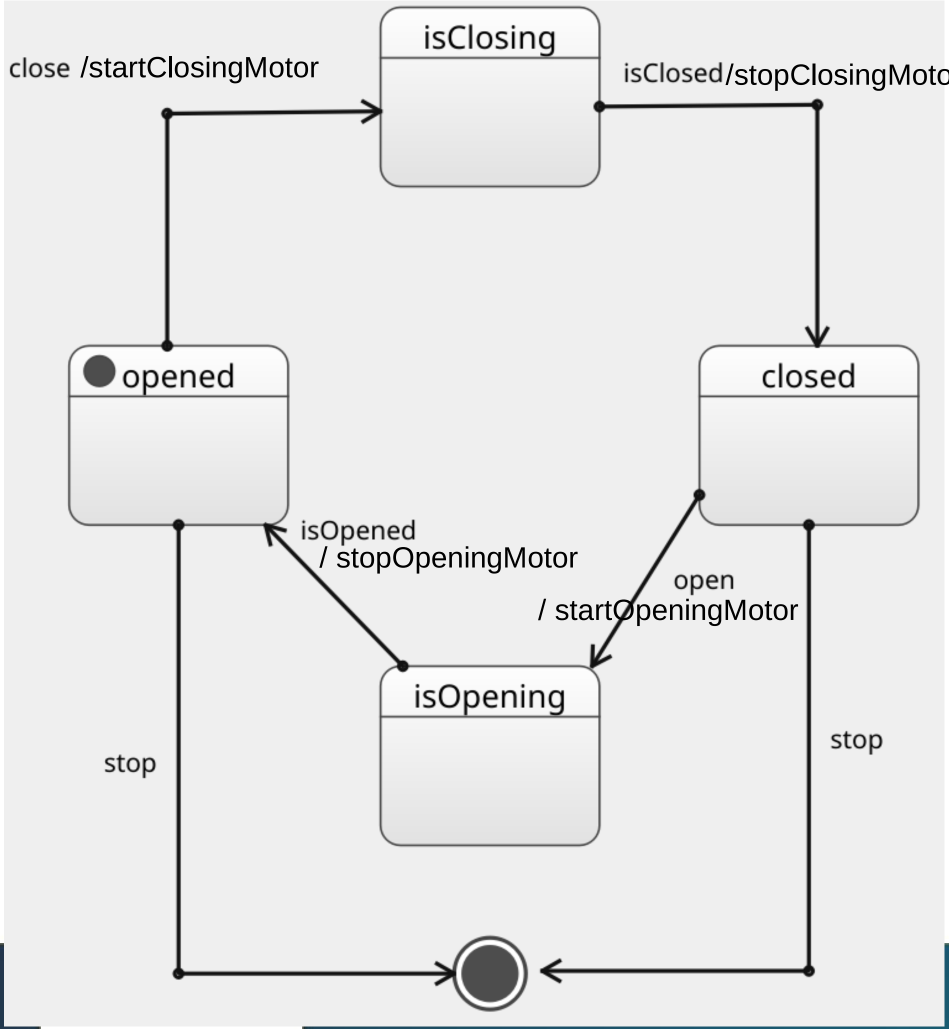
Computation Tree

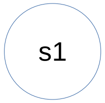


Taken from Tevfik Bultan, Model Checking Foundations and Applications

V&V ?

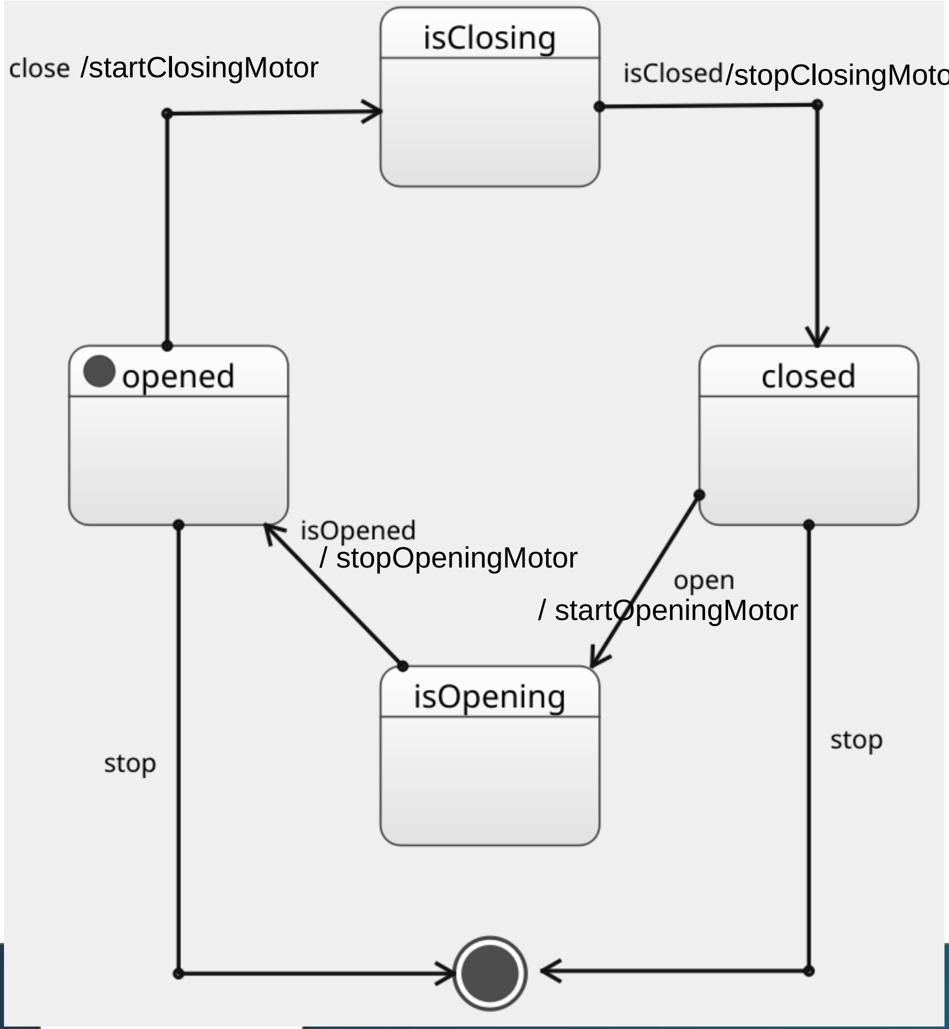
- ensemble de chemins d’exécutions finis ?
  - Énumération de l’espace d’état (habituellement un graphe orienté d’une forme particulière : *Labelled Transition system* or *Kripke structure*)

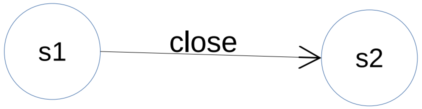




V&V ?

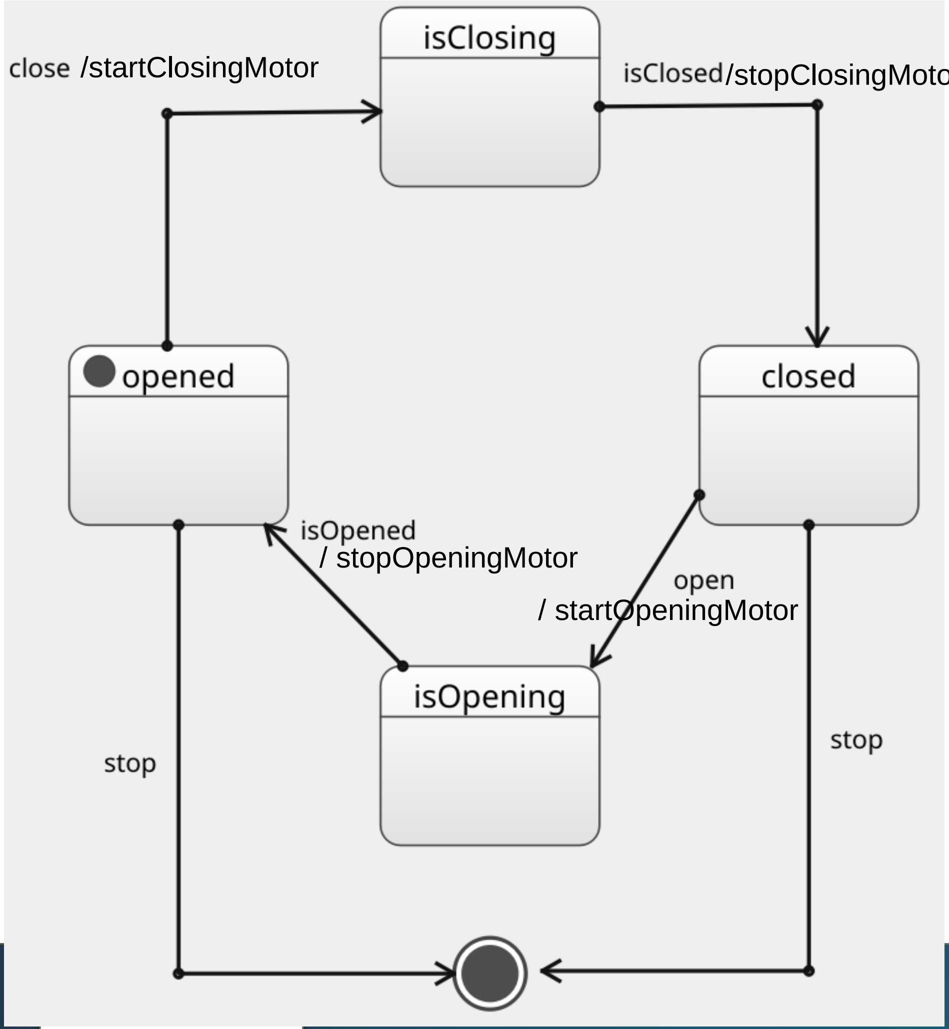
- ensemble de chemins d'exécutions finis ?
  - Énumération de l'espace d'état (habituellement un graphe orienté d'une forme particulière : *Labelled Transition system* or *Kripke structure*)

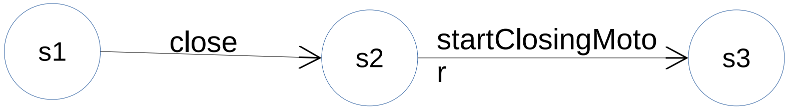




V&V ?

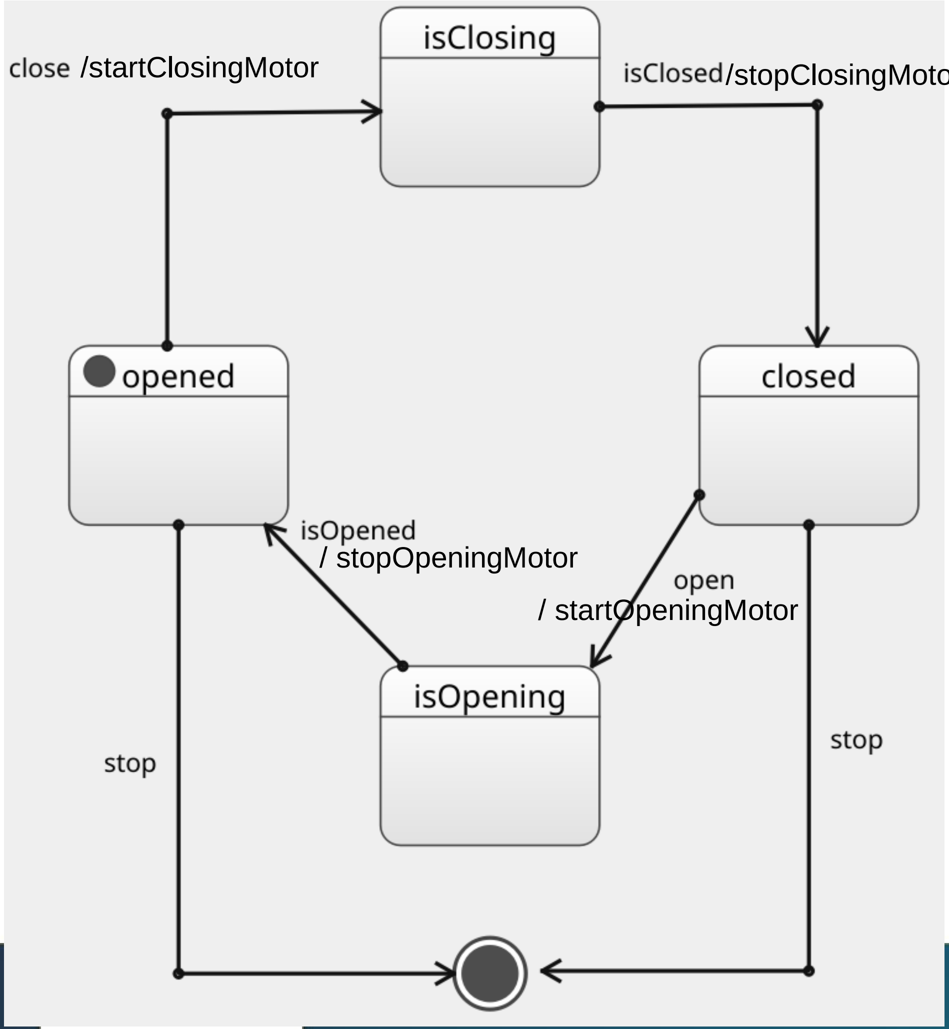
- ensemble de chemins d'exécutions finis ?
  - Énumération de l'espace d'état (habituellement un graphe orienté d'une forme particulière : *Labelled Transition system* or *Kripke structure*)

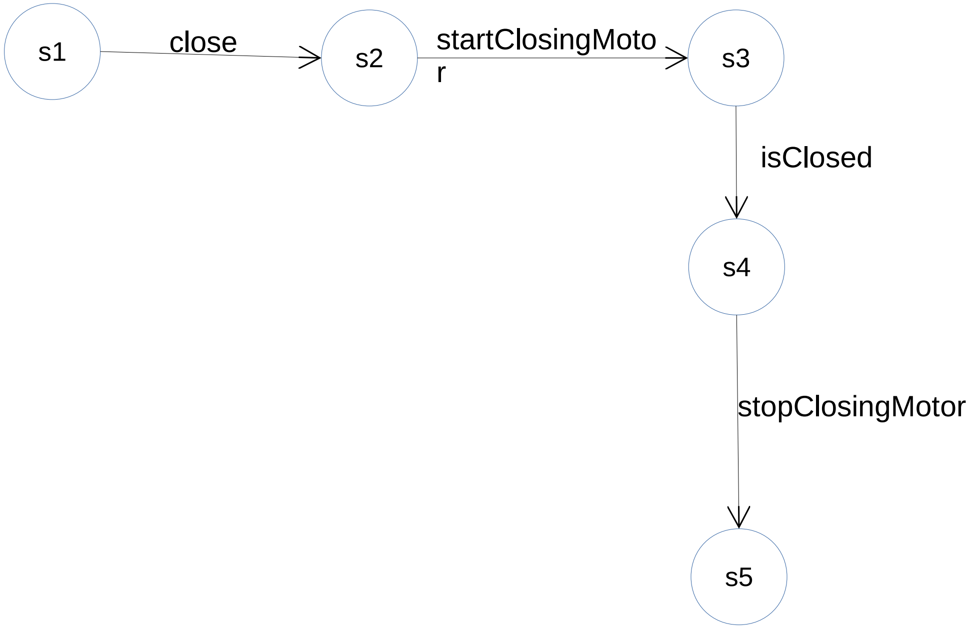




V&V ?

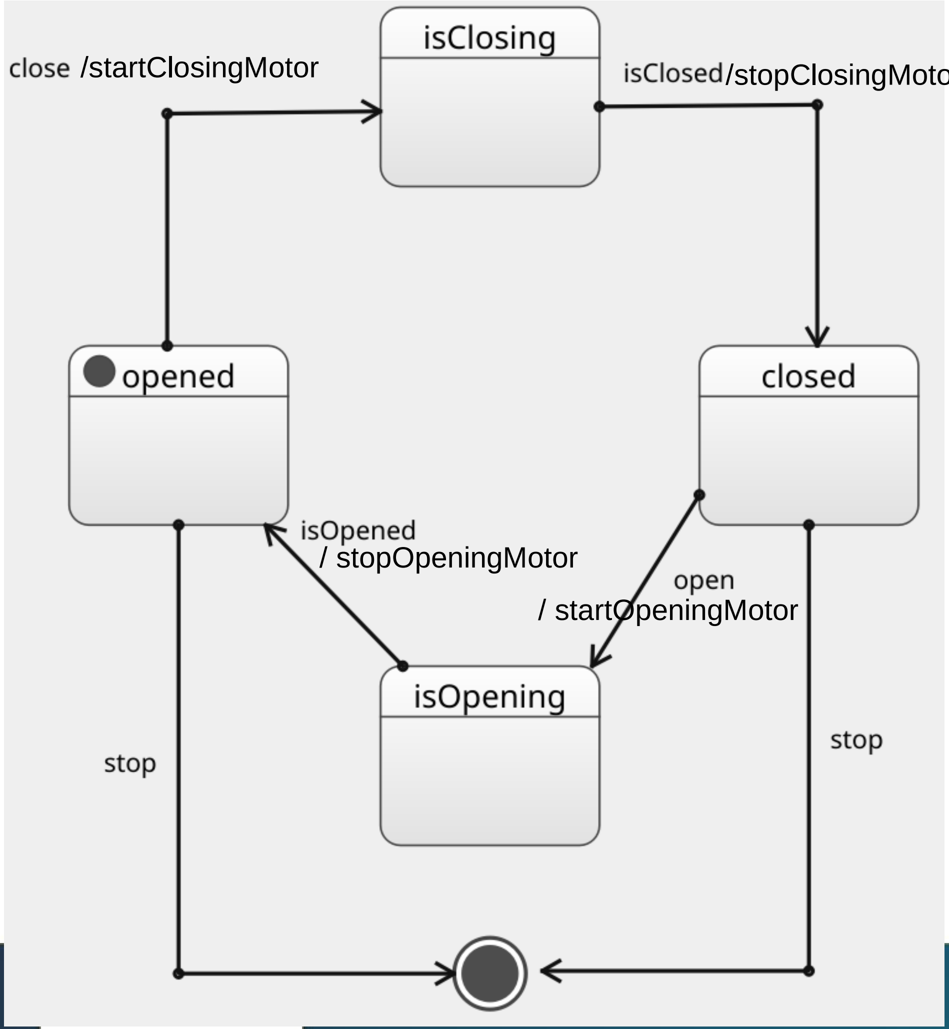
- ensemble de chemins d’exécutions finis ?
  - Énumération de l’espace d’état (habituellement un graphe orienté d’une forme particulière : *Labelled Transition system* or *Kripke structure*)

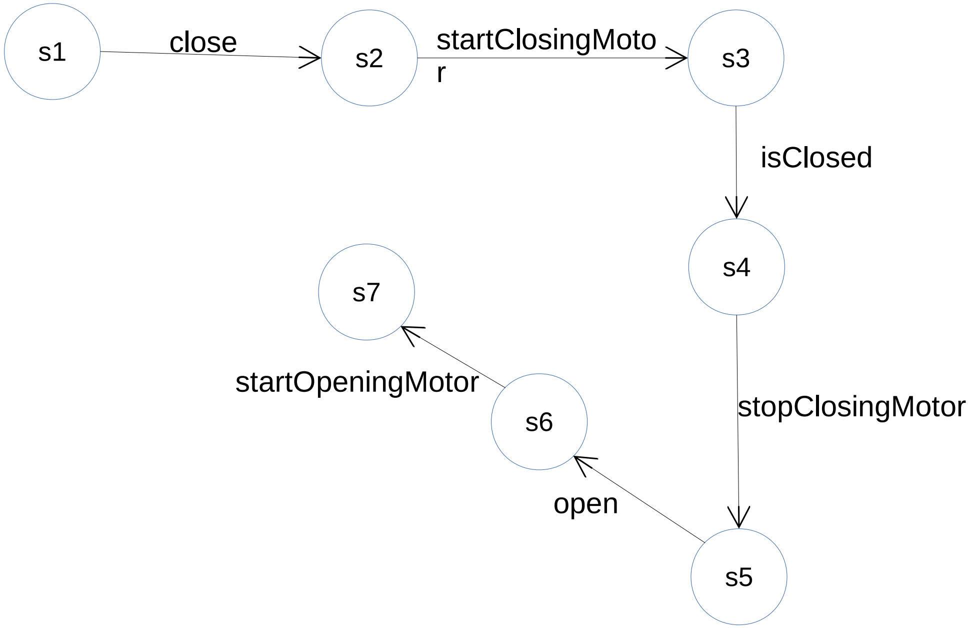




V&V ?

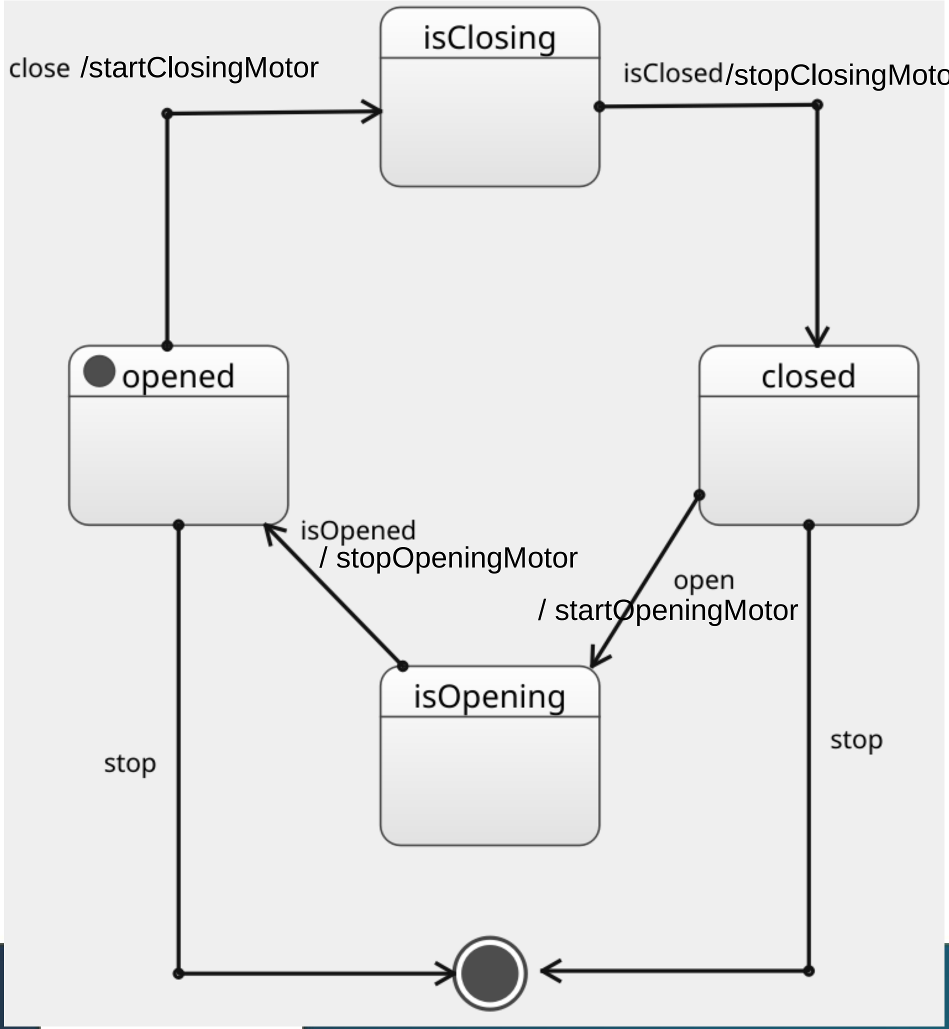
- ensemble de chemins d'exécutions finis ?
  - Énumération de l'espace d'état (habituellement un graphe orienté d'une forme particulière : *Labelled Transition system* or *Kripke structure*)

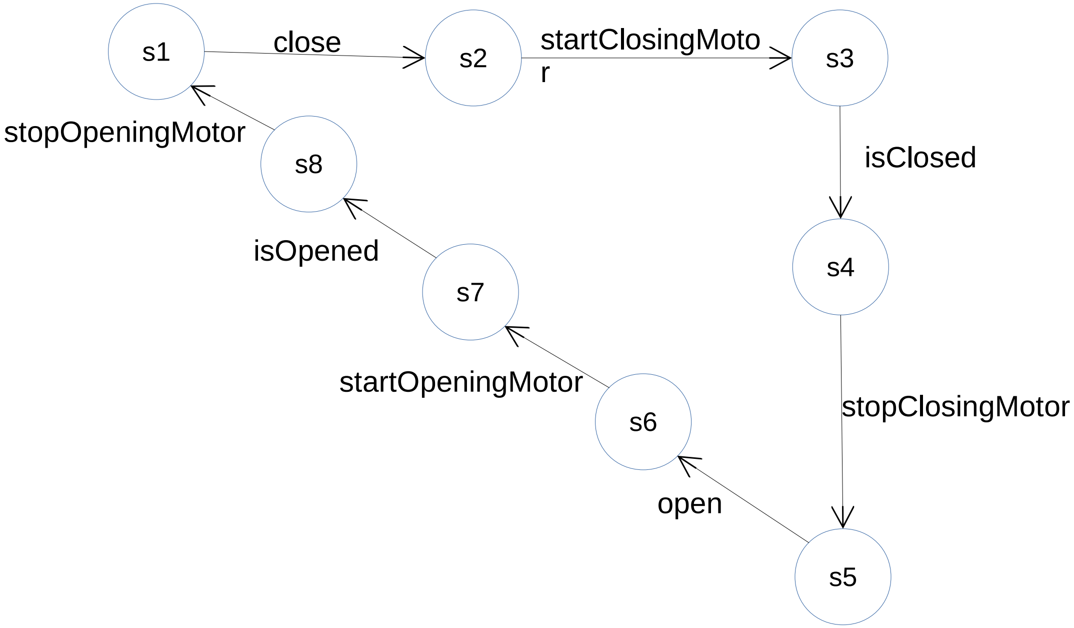




V&V ?

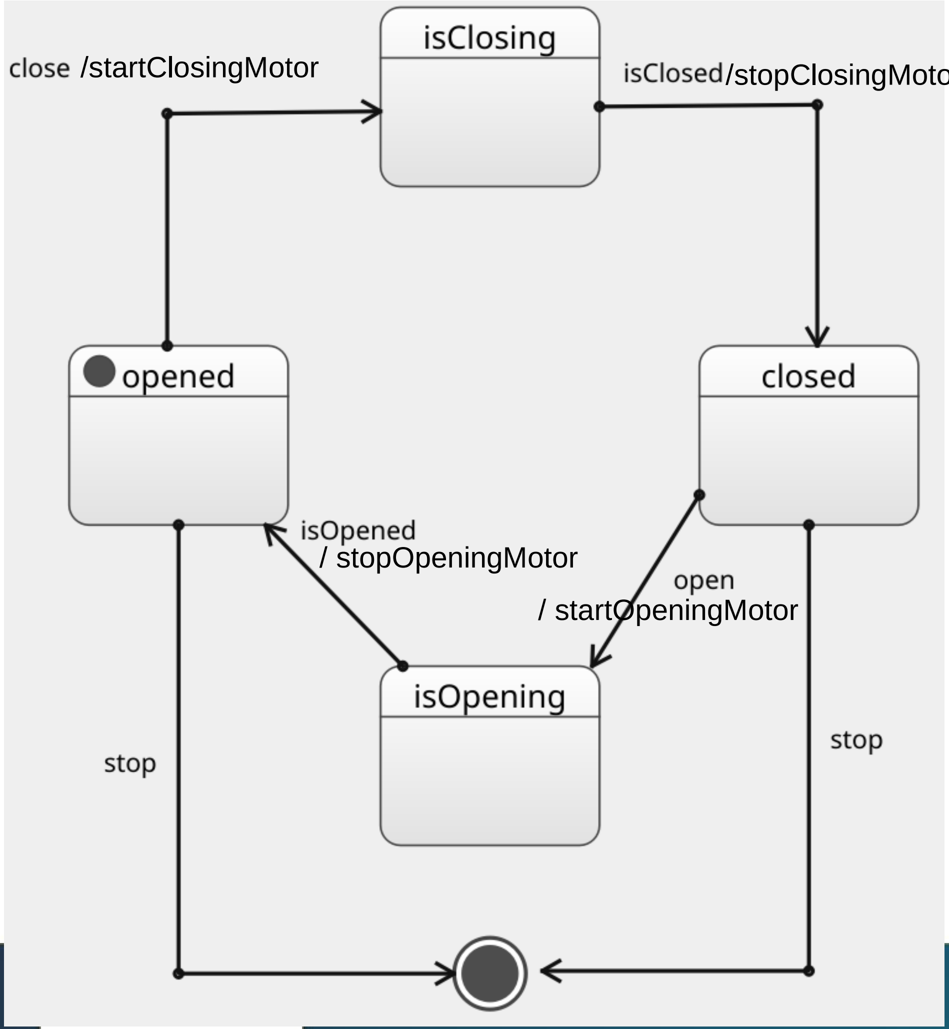
- ensemble de chemins d’exécutions finis ?
  - Énumération de l’espace d’état (habituellement un graphe orienté d’une forme particulière : *Labelled Transition system* or *Kripke structure*)

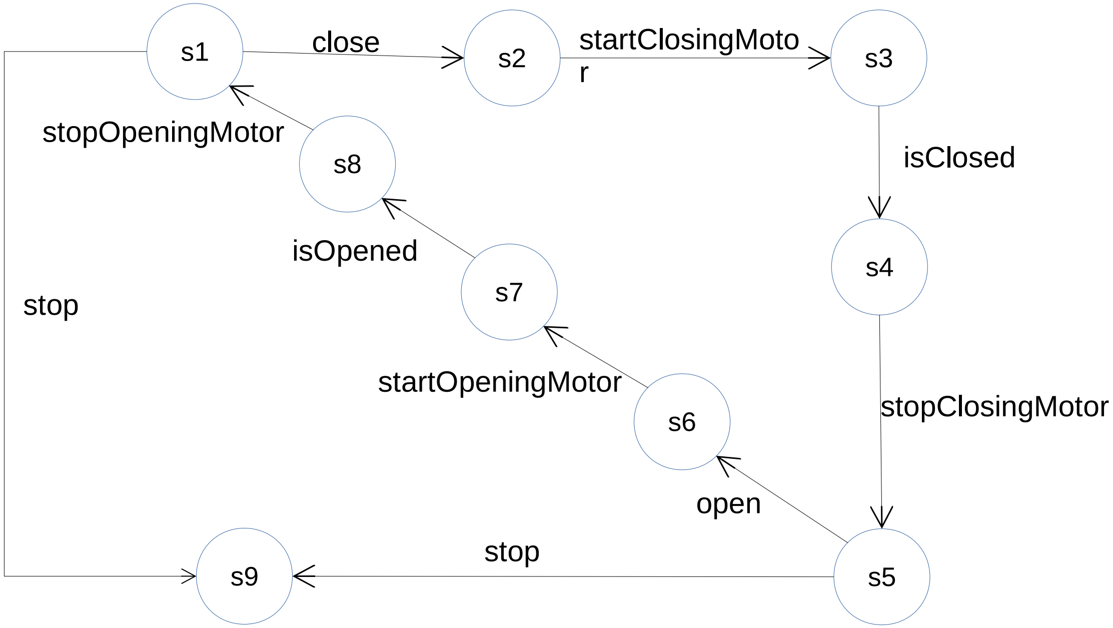




V&V ?

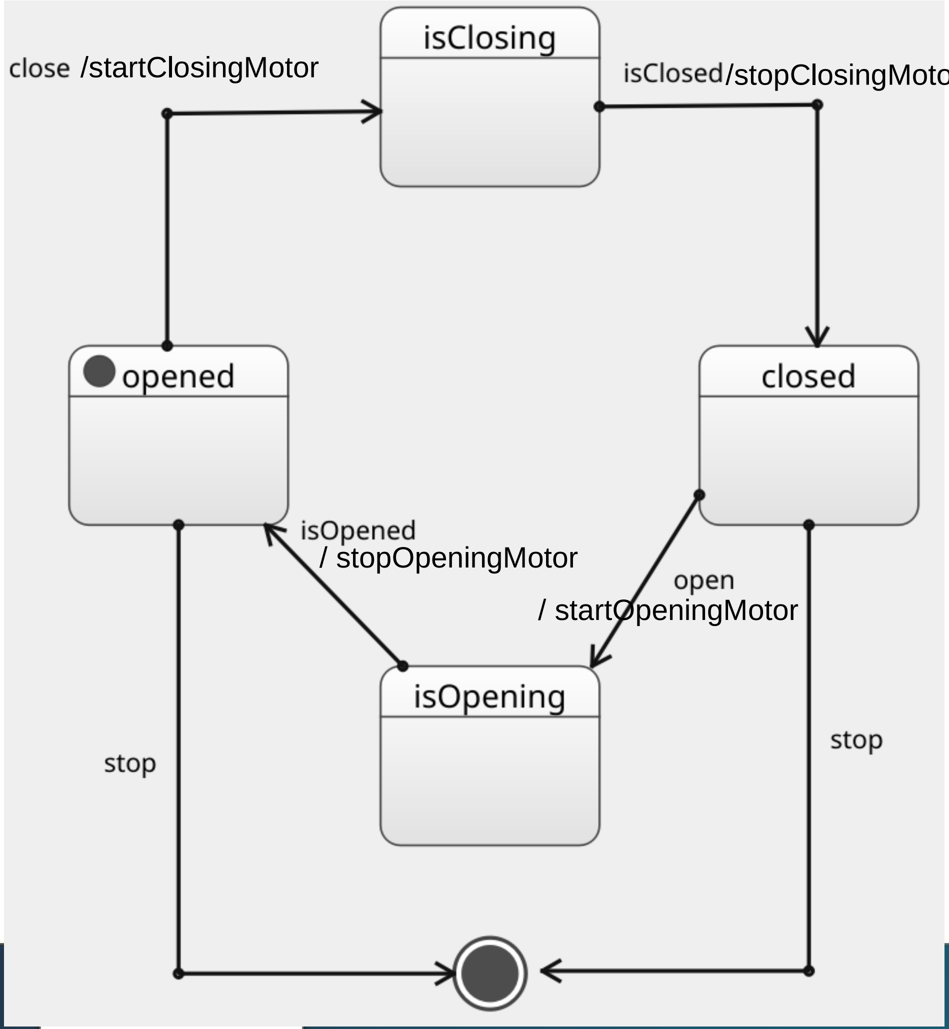
- ensemble de chemins d'exécutions finis ?
  - Énumération de l'espace d'état (habituellement un graphe orienté d'une forme particulière : *Labelled Transition system* or *Kripke structure*)

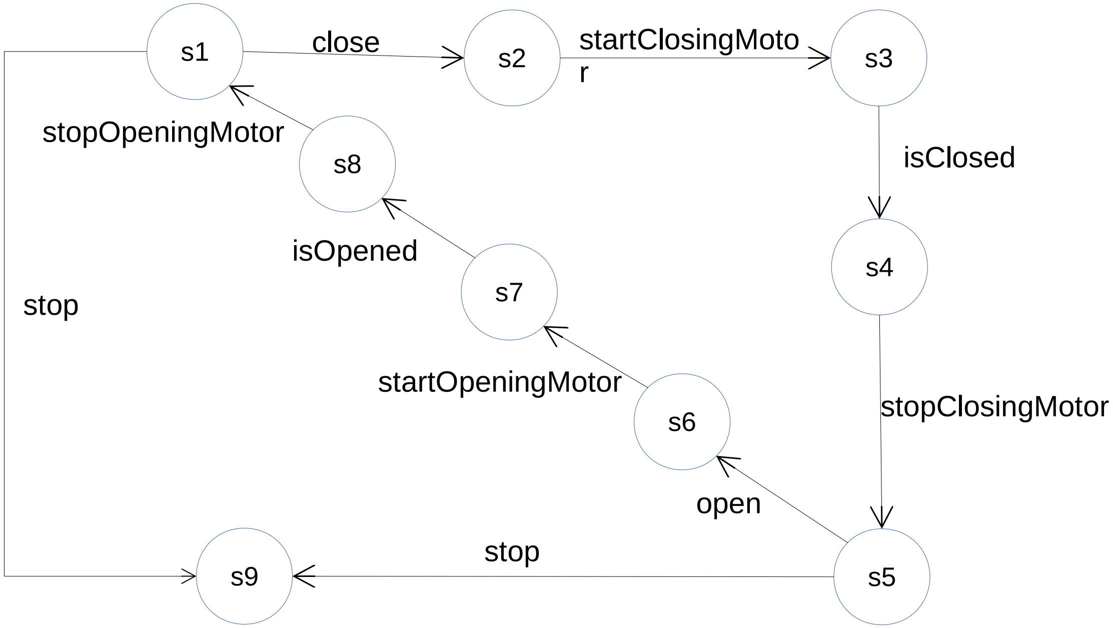




V&V ?

- 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*)

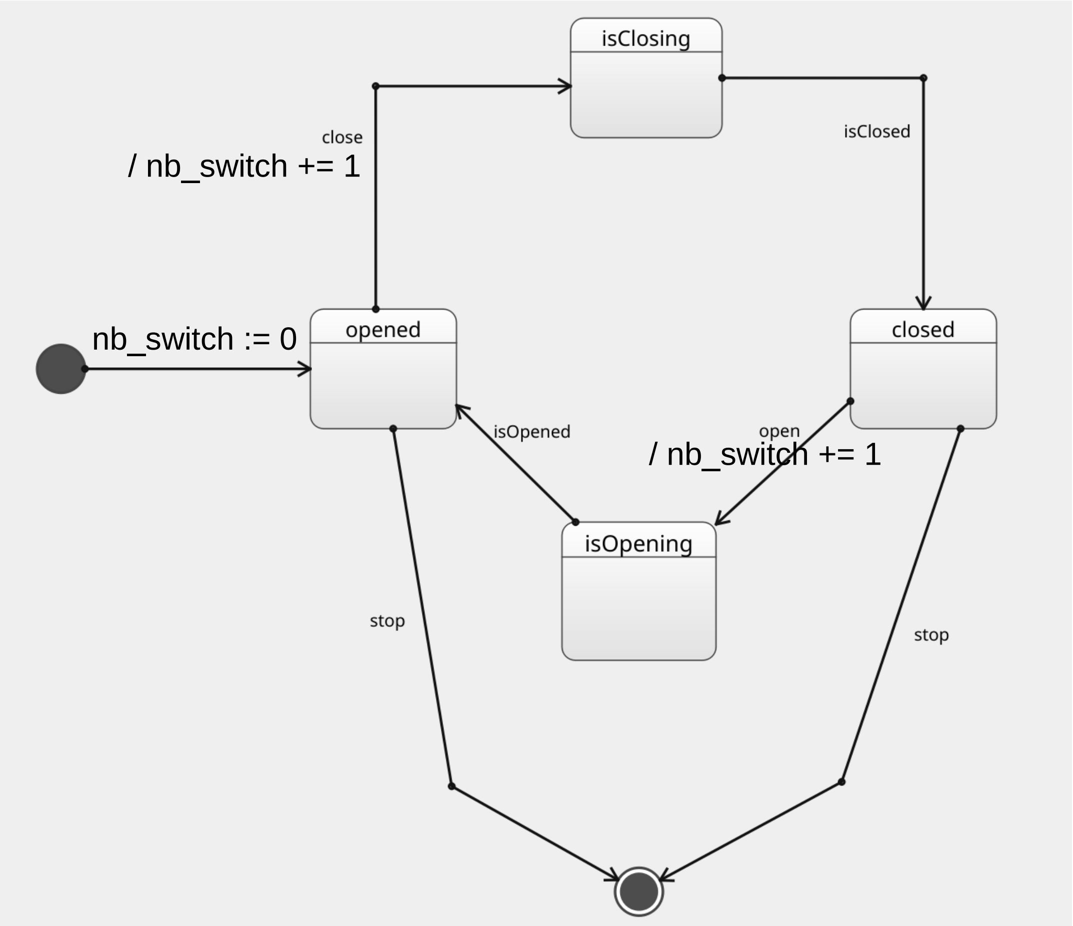


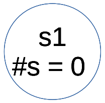


Plus les actions *onEnter* et *onExit !*

V&V ?

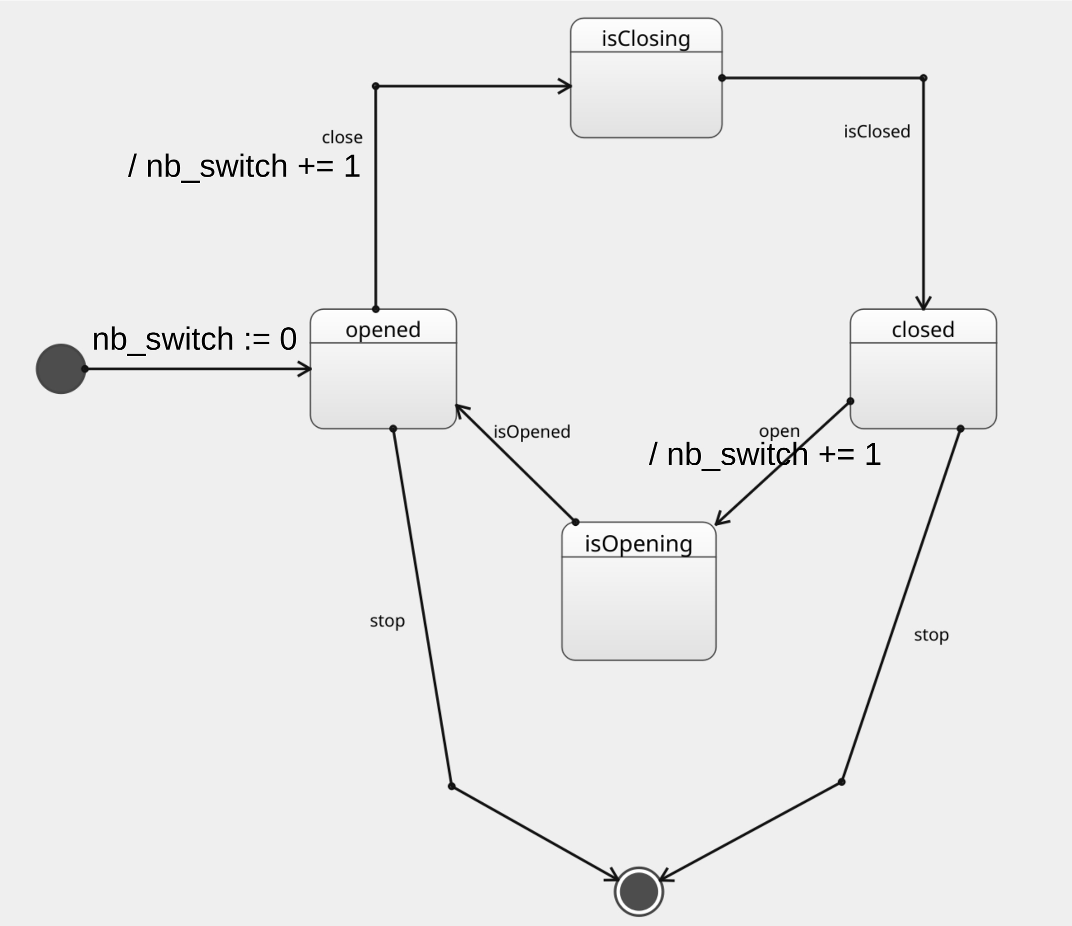
- ensemble de chemins d’exécutions finis ?

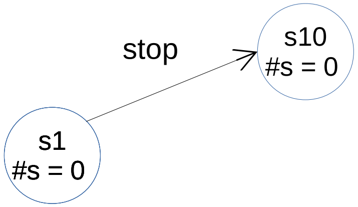




V&V ?

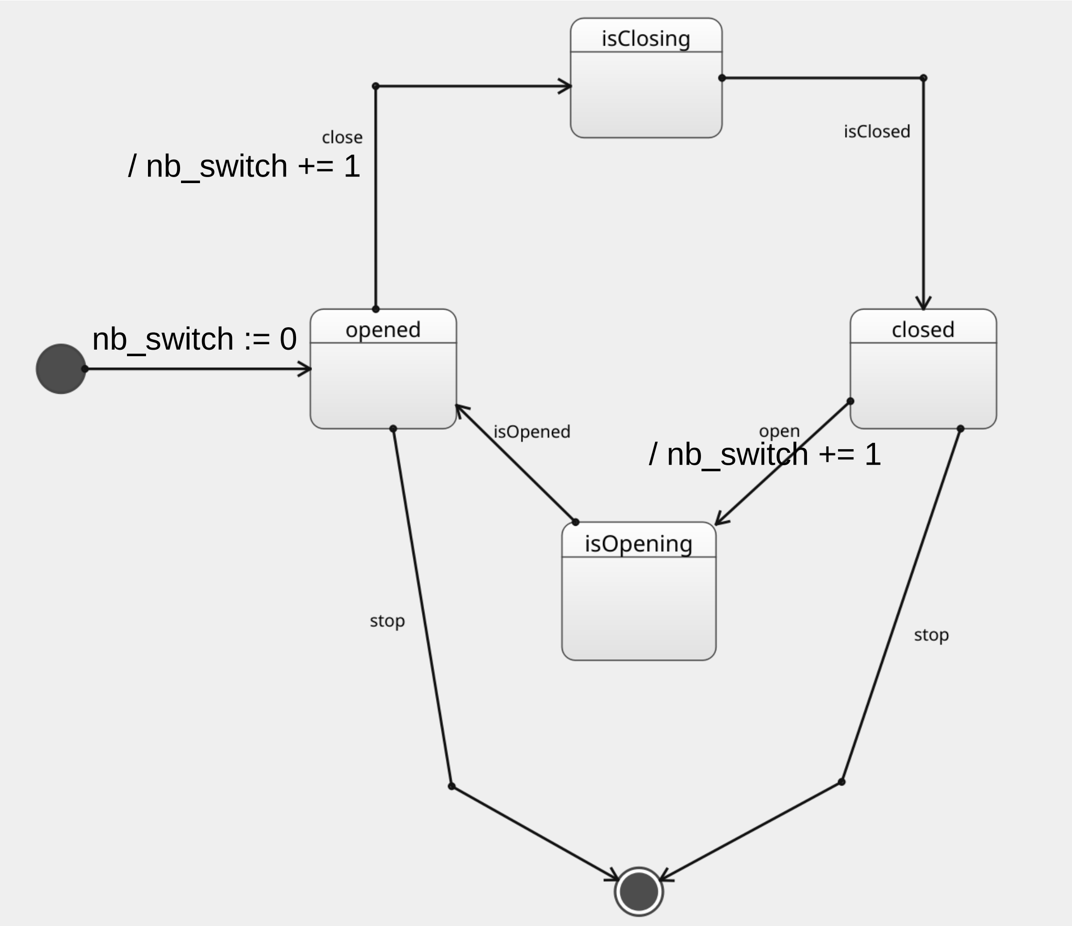
- ensemble de chemins d'exécutions finis ?

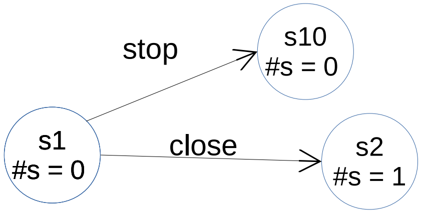




V&V ?

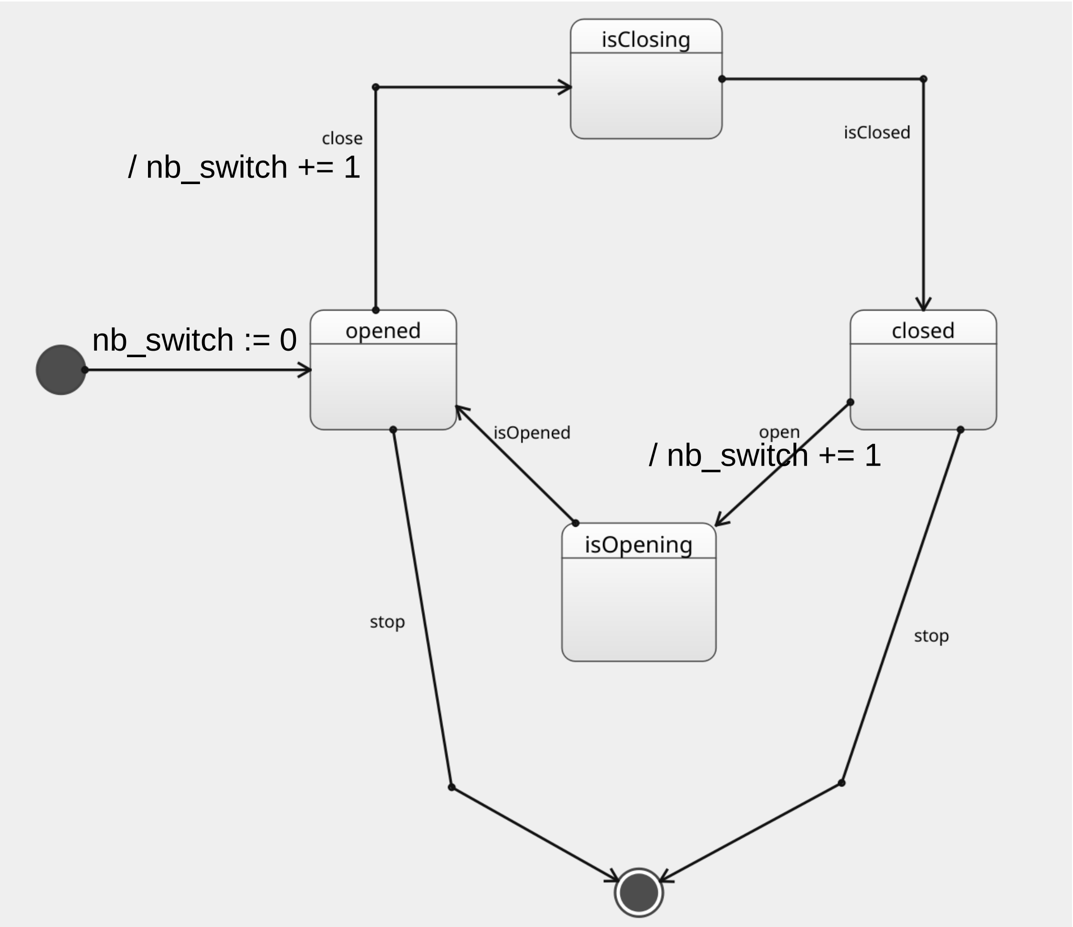
- ensemble de chemins d'exécutions finis ?

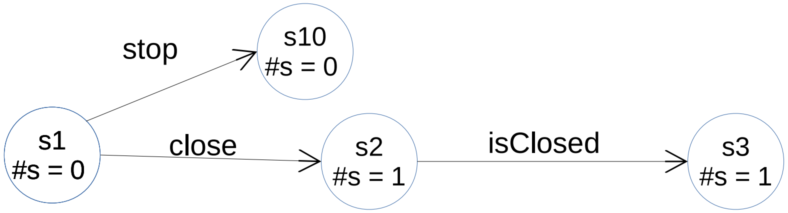




V&V ?

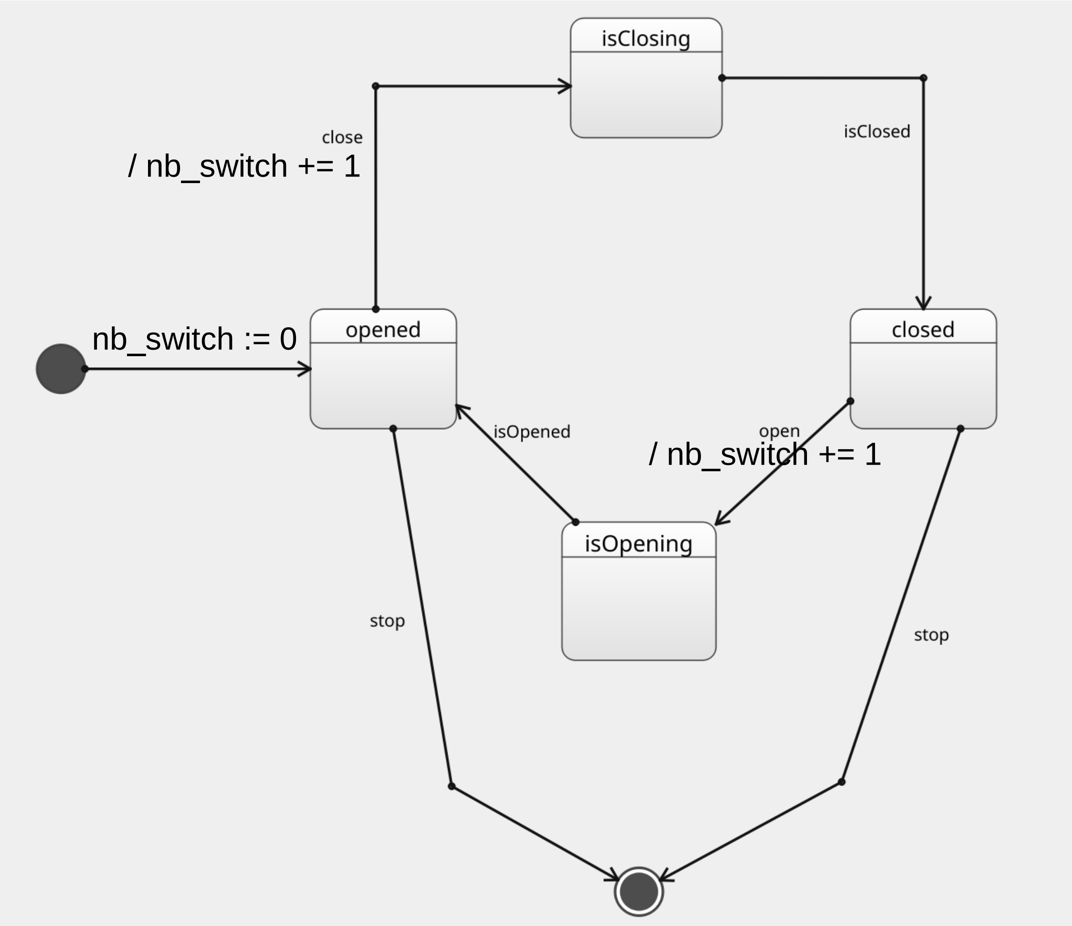
- ensemble de chemins d'exécutions finis ?

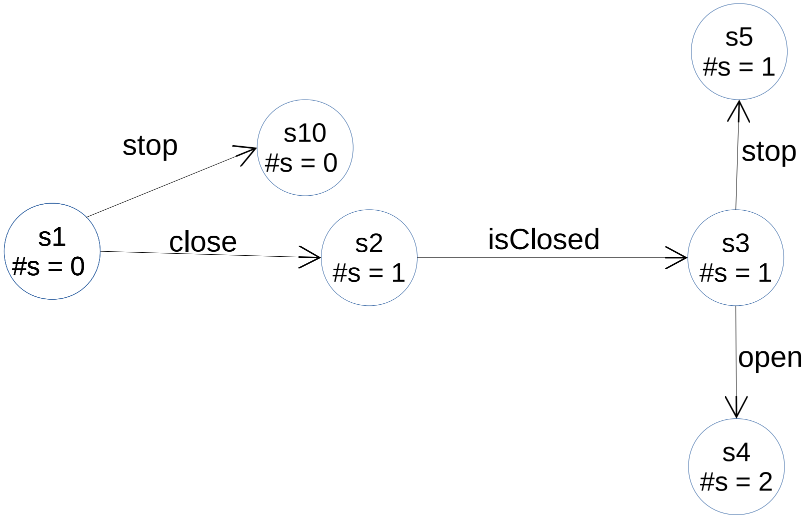




V&V ?

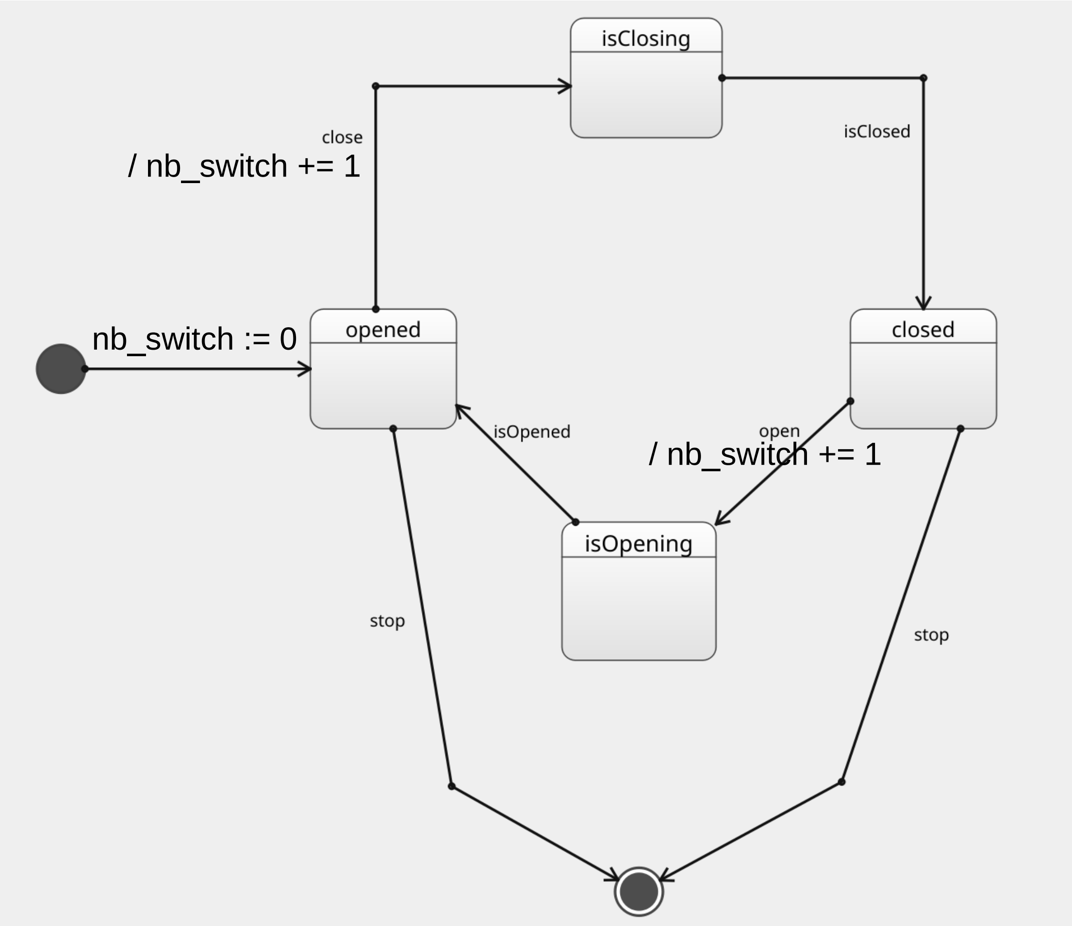
- ensemble de chemins d'exécutions finis ?

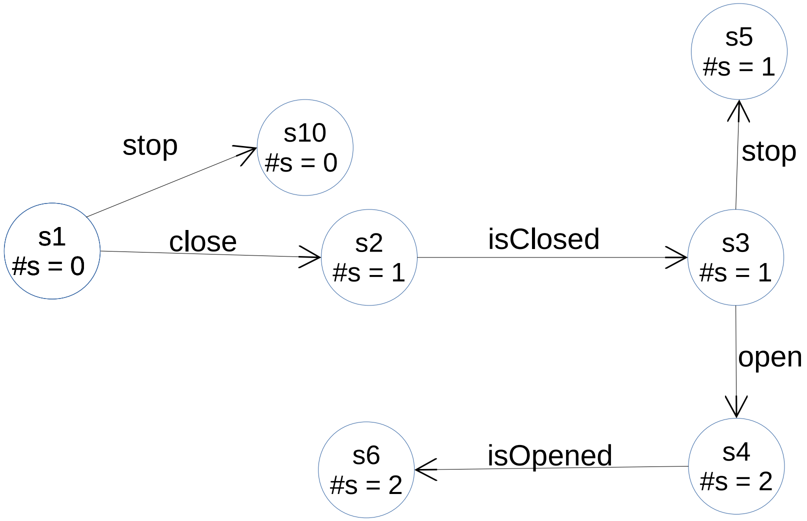




V&V ?

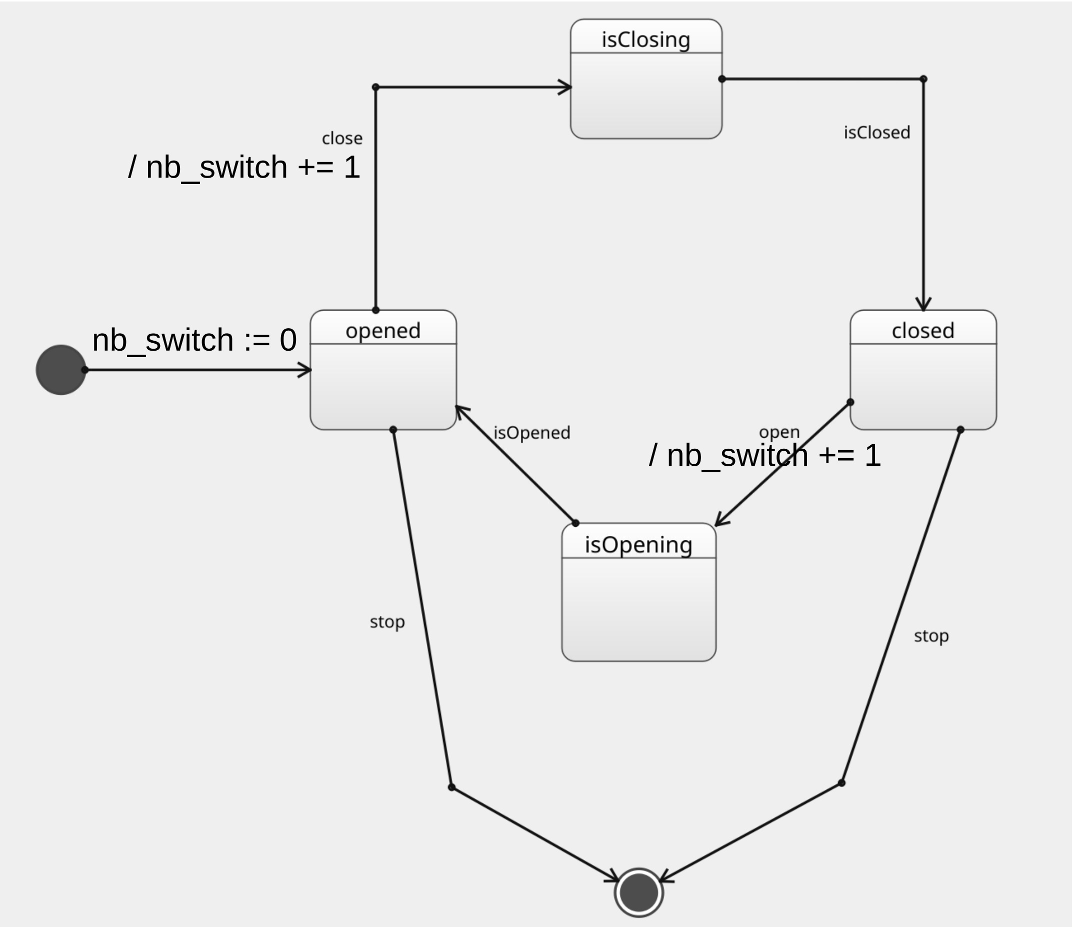
- ensemble de chemins d'exécutions finis ?

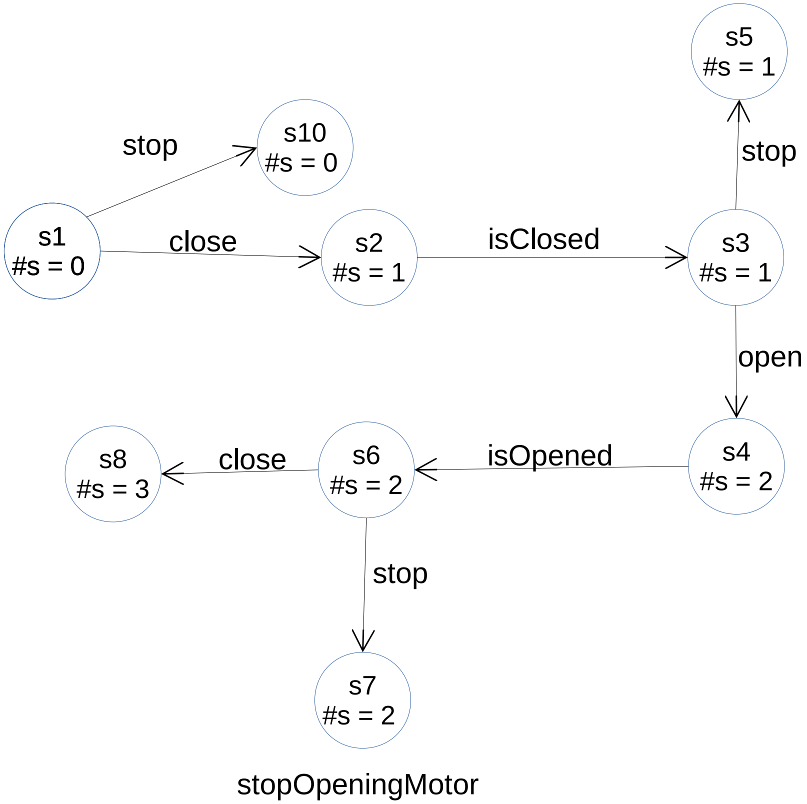




V&V ?

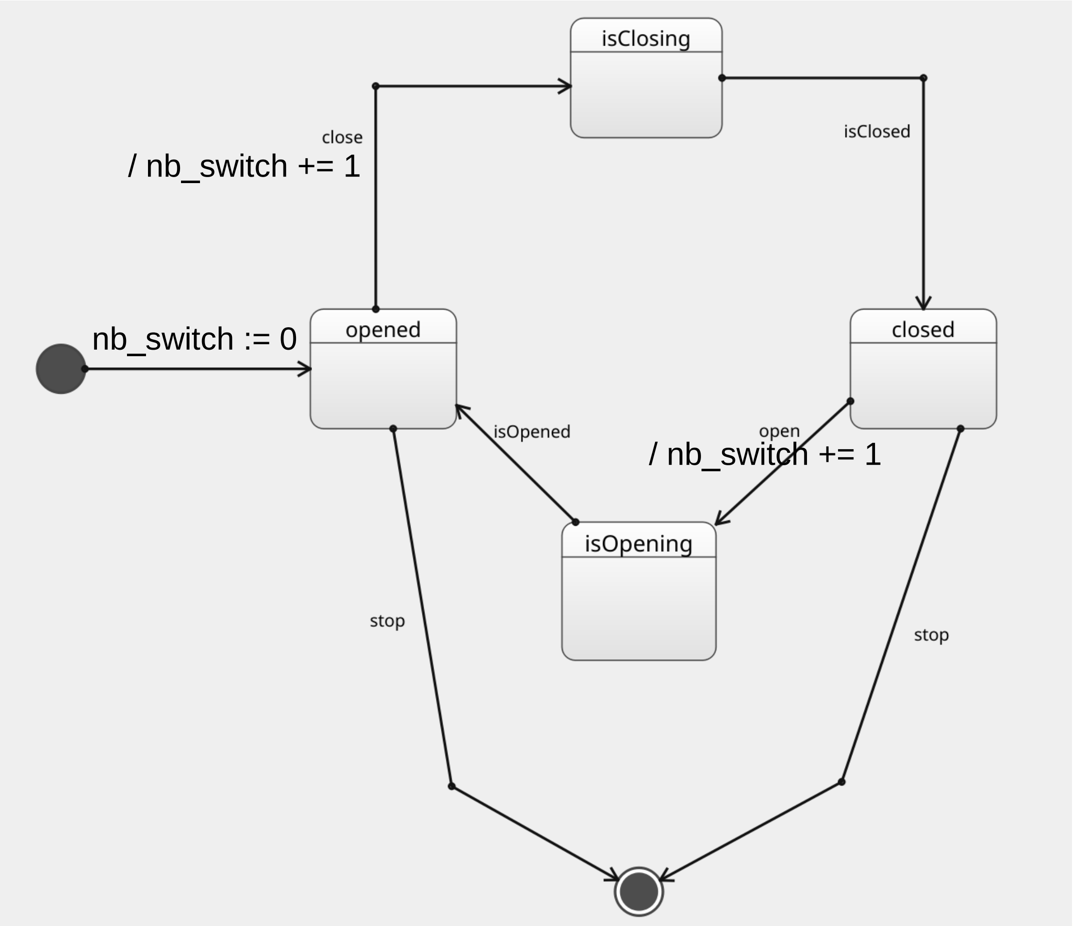
- ensemble de chemins d'exécutions finis ?

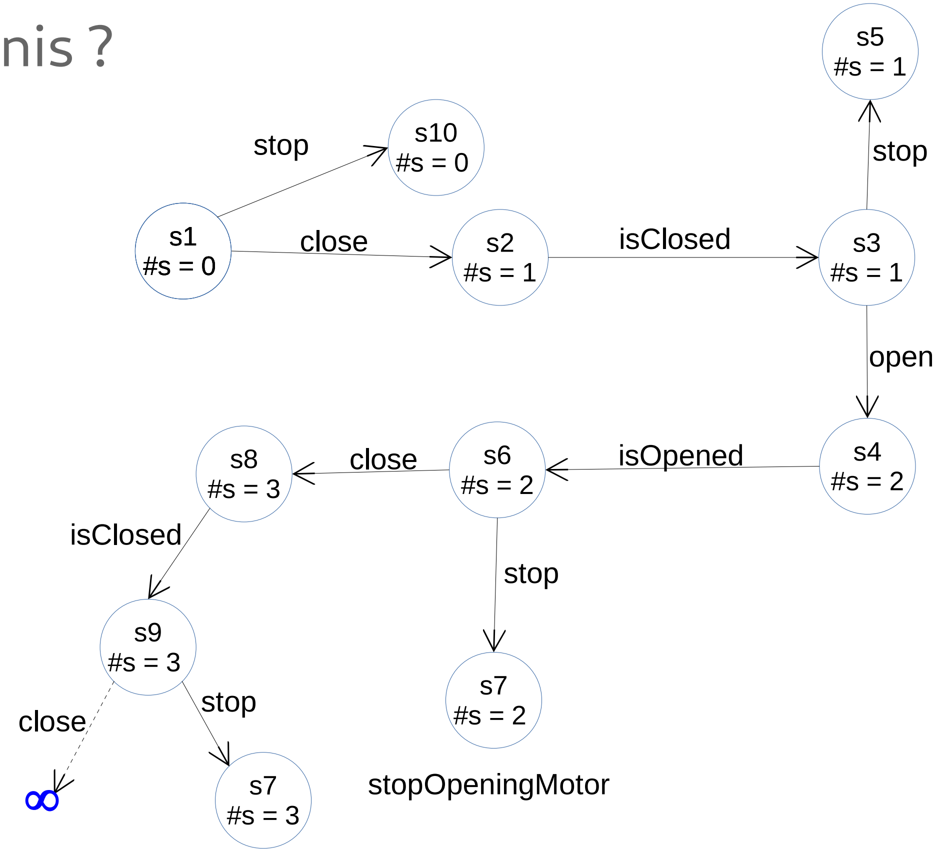




V&V ?

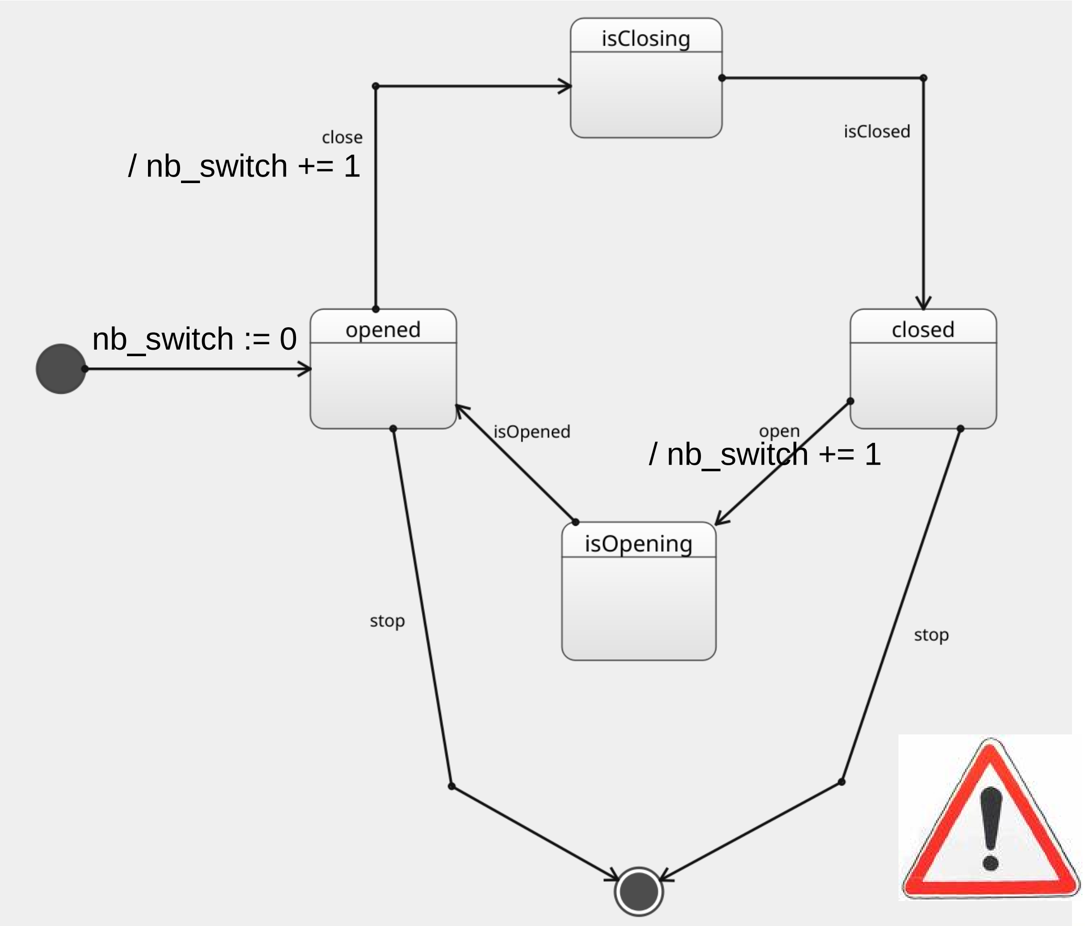
- ensemble de chemins d'exécutions finis ?

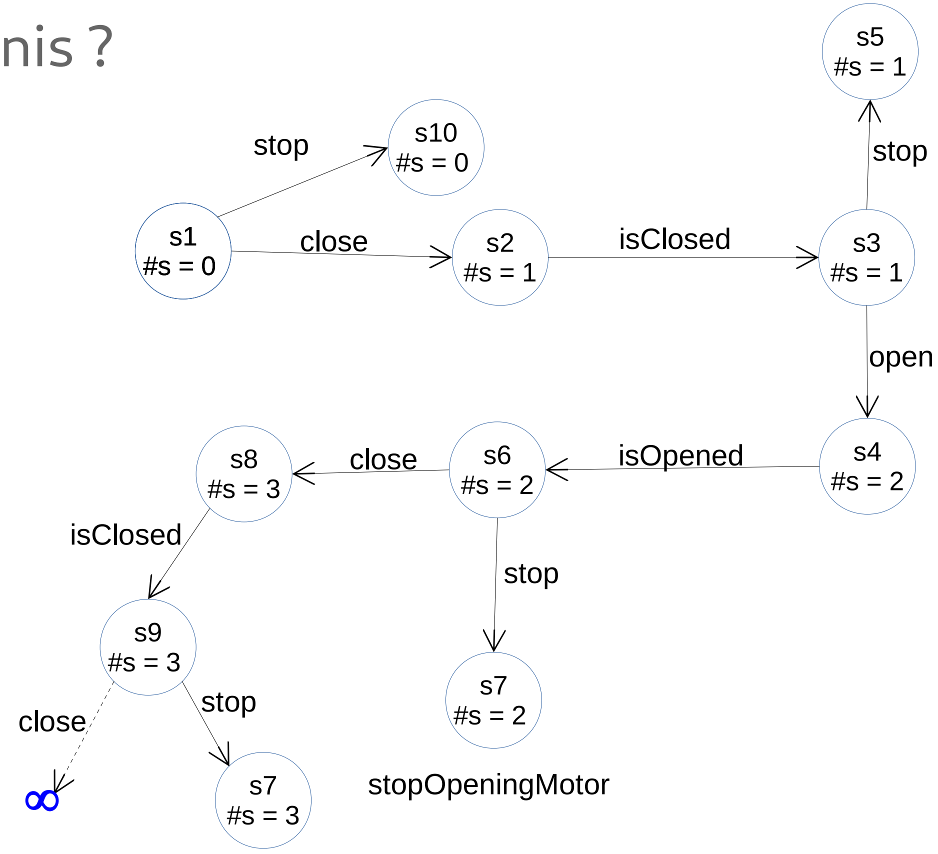




V&V ?

- ensemble de chemins d’exécutions finis ?



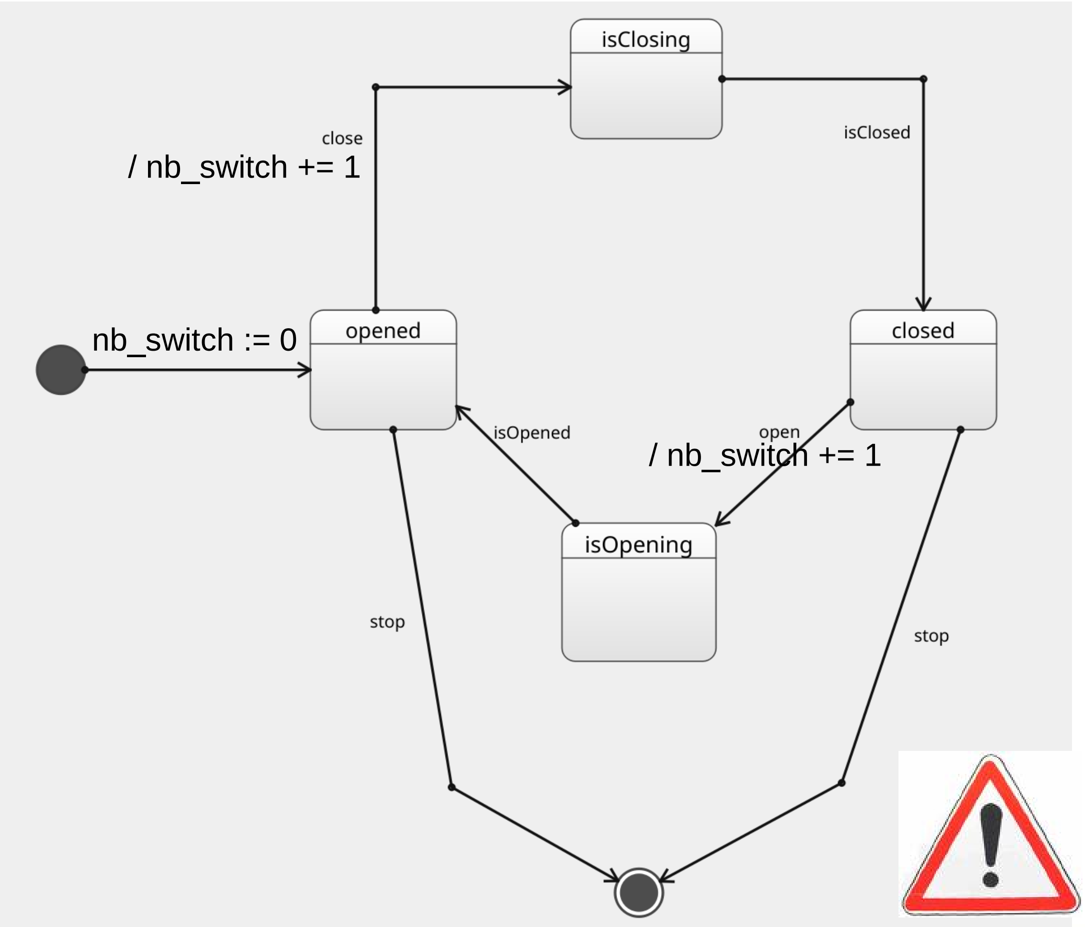


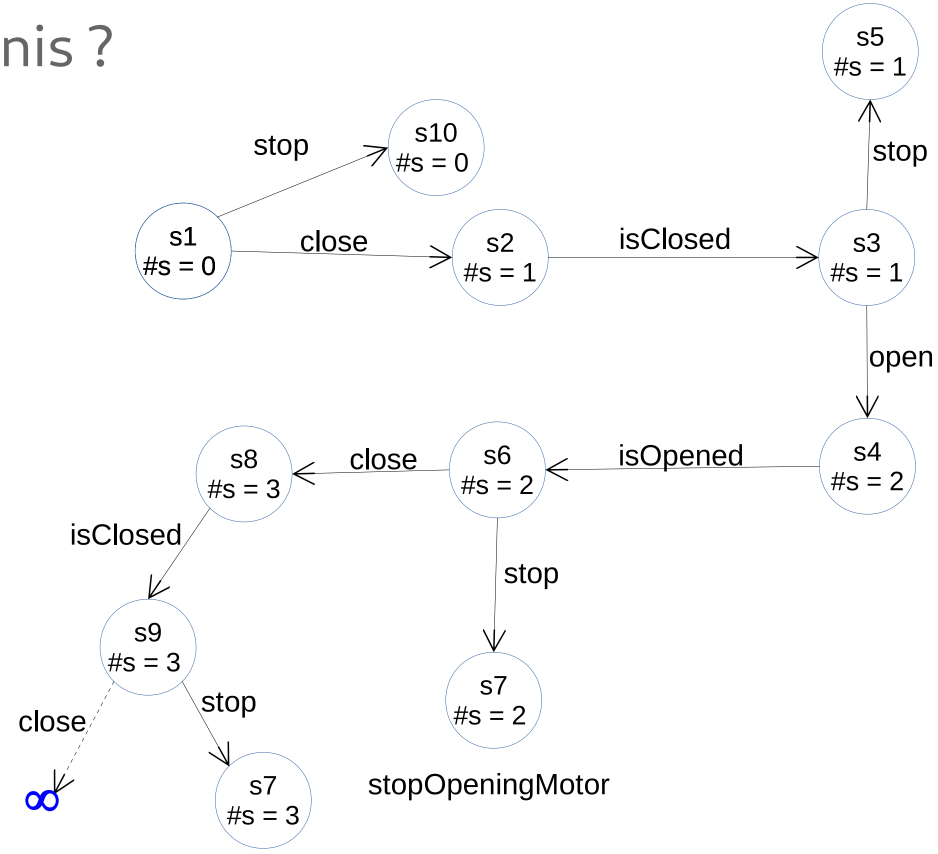
Tout ce qui est dans la state machine est considéré dans la construction de l’espace d’état.

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 ?

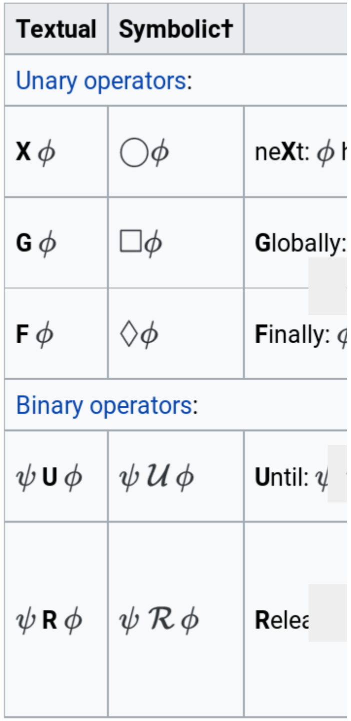




- Determine what should be observable events / conditions / execution states

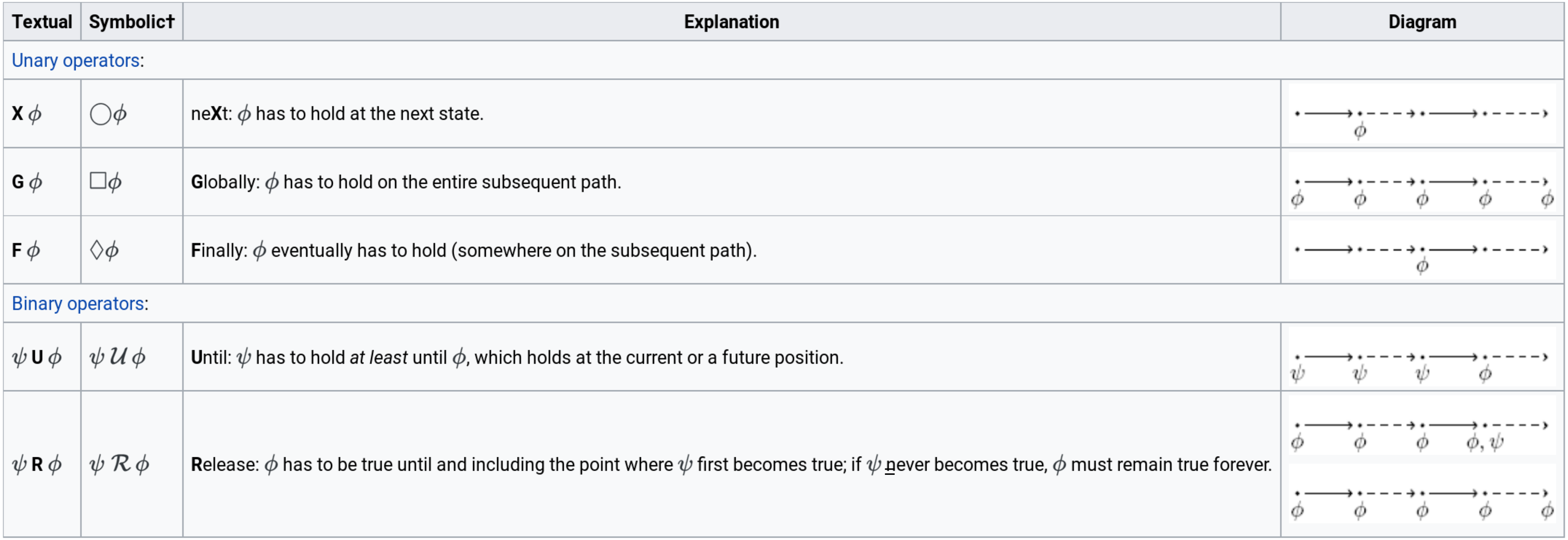
Logiques temporelles

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



Textual	Symbolic†	
Unary operators:		
X ϕ	$\bigcirc\phi$	ne**X**t: ϕ h
G ϕ	$\Box\phi$	**G**lobally:
F ϕ	$\Diamond\phi$	**F**inally: ϕ
Binary operators:		
ψ **U** ϕ	$\psi\,\mathcal{U}\,\phi$	**U**ntil: ψ
ψ **R** ϕ	$\psi\,\mathcal{R}\,\phi$	**R**elea

LTL



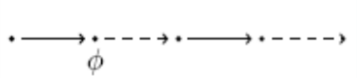
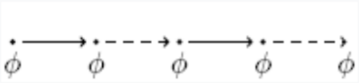
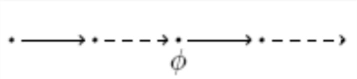
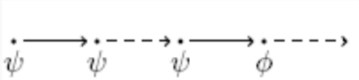
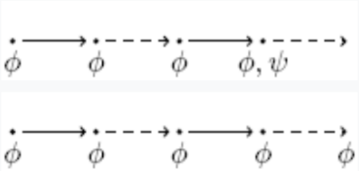
Textual	Symbolic†	Explanation	Diagram
Unary operators:			
X ϕ	$\bigcirc\phi$	ne**X**t: ϕ has to hold at the next state.	ϕ
G ϕ	$\Box\phi$	**G**lobally: ϕ has to hold on the entire subsequent path.	ϕ ϕ ϕ ϕ ϕ
F ϕ	$\Diamond\phi$	**F**inally: ϕ eventually has to hold (somewhere on the subsequent path).	ϕ
Binary operators:			
ψ **U** ϕ	$\psi\,\mathcal{U}\,\phi$	**U**ntil: ψ has to hold *at least* until ϕ, which holds at the current or a future position.	ψ ψ ψ ϕ
ψ **R** ϕ	$\psi\,\mathcal{R}\,\phi$	**R**elease: ϕ has to be true until and including the point where ψ first becomes true; if ψ never becomes true, ϕ must remain true forever.	ϕ ϕ ϕ ϕ,ψ ϕ ϕ ϕ ϕ ϕ

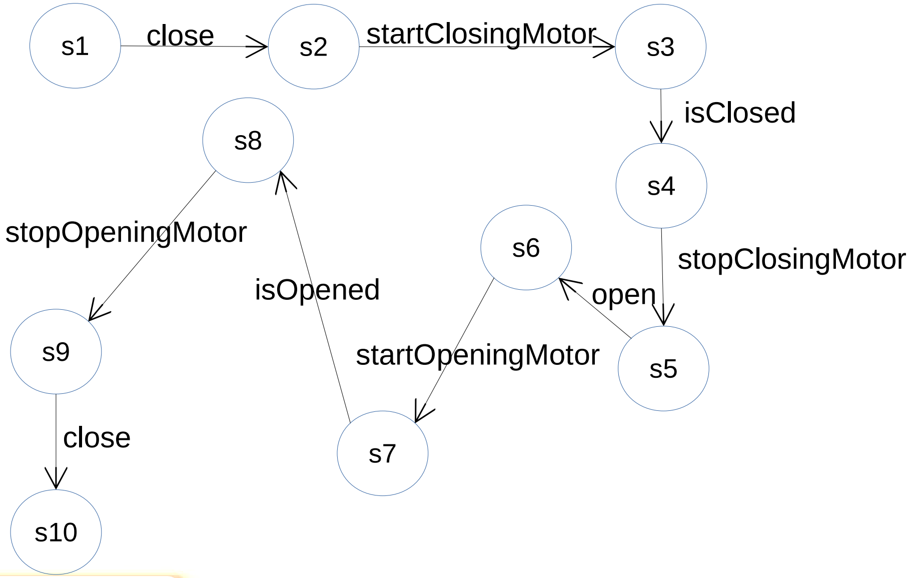
Model checking problem:

Given a model M and an LTL formula φ, all traces of M must satisfy φ

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

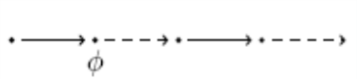
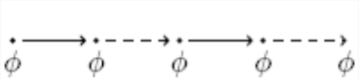
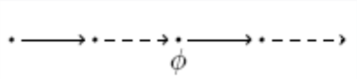
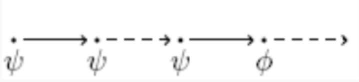
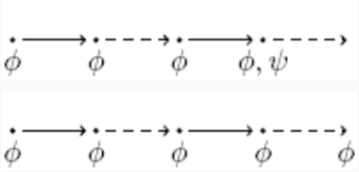
LTL

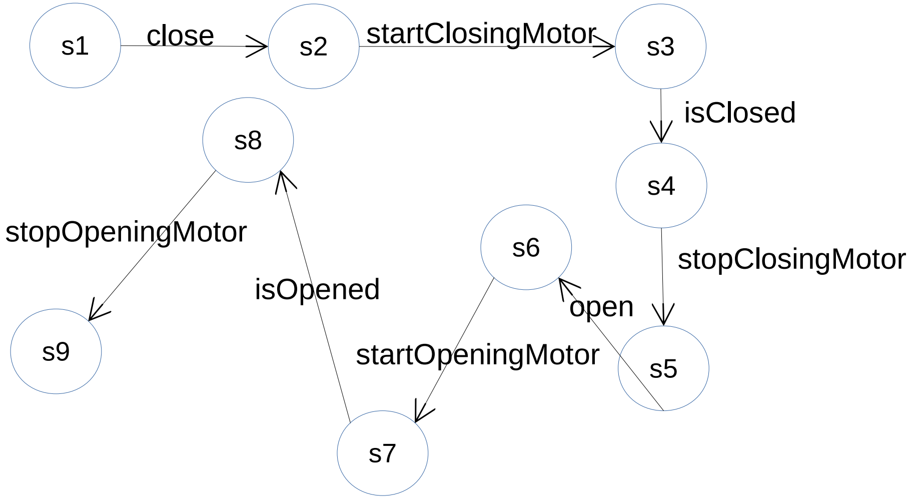
Textual	Symbolic†	Explanation	Diagram
Unary operators:			
X ϕ	$\bigcirc\phi$	ne**X**t: ϕ has to hold at the next state.	
G ϕ	$\square\phi$	**G**lobally: ϕ has to hold on the entire subsequent path.	
F ϕ	$\lozenge\phi$	**F**inally: ϕ eventually has to hold (somewhere on the subsequent path).	
Binary operators:			
ψ **U** ϕ	$\psi\,\mathcal{U}\,\phi$	**U**ntil: ψ has to hold *at least* until ϕ, which holds at the current or a future position.	
ψ **R** ϕ	$\psi\,\mathcal{R}\,\phi$	**R**elease: ϕ has to be true until and including the point where ψ first becomes true; if ψ never becomes true, ϕ must remain true forever.	



$\models \quad \square\,(close \Rightarrow \lozenge\, isClosed)$?

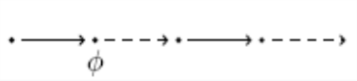
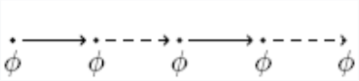
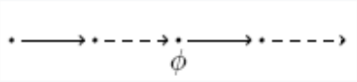
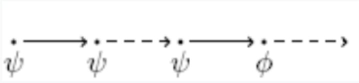
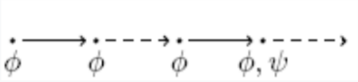
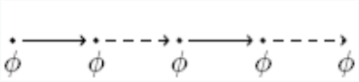
LTL

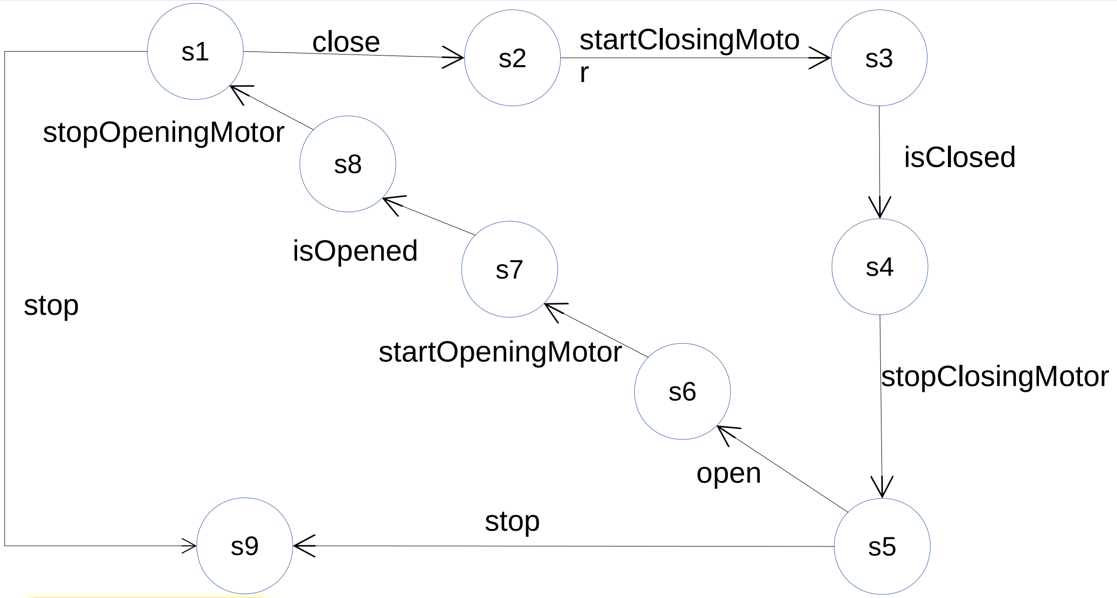
Textual	Symbolic†	Explanation	Diagram
Unary operators:			
X ϕ	$\bigcirc\phi$	ne**X**t: ϕ has to hold at the next state.	
G ϕ	$\Box\phi$	**G**lobally: ϕ has to hold on the entire subsequent path.	
F ϕ	$\Diamond\phi$	**F**inally: ϕ eventually has to hold (somewhere on the subsequent path).	
Binary operators:			
ψ **U** ϕ	$\psi\,\mathcal{U}\,\phi$	**U**ntil: ψ has to hold *at least* until ϕ, which holds at the current or a future position.	
ψ **R** ϕ	$\psi\,\mathcal{R}\,\phi$	**R**elease: ϕ has to be true until and including the point where ψ first becomes true; if ψ never becomes true, ϕ must remain true forever.	 ϕ ϕ ϕ ϕ, ψ ϕ ϕ ϕ ϕ ϕ



$$\models \quad \Box\,(close \Rightarrow \Diamond isClosed)\ ?$$

LTL

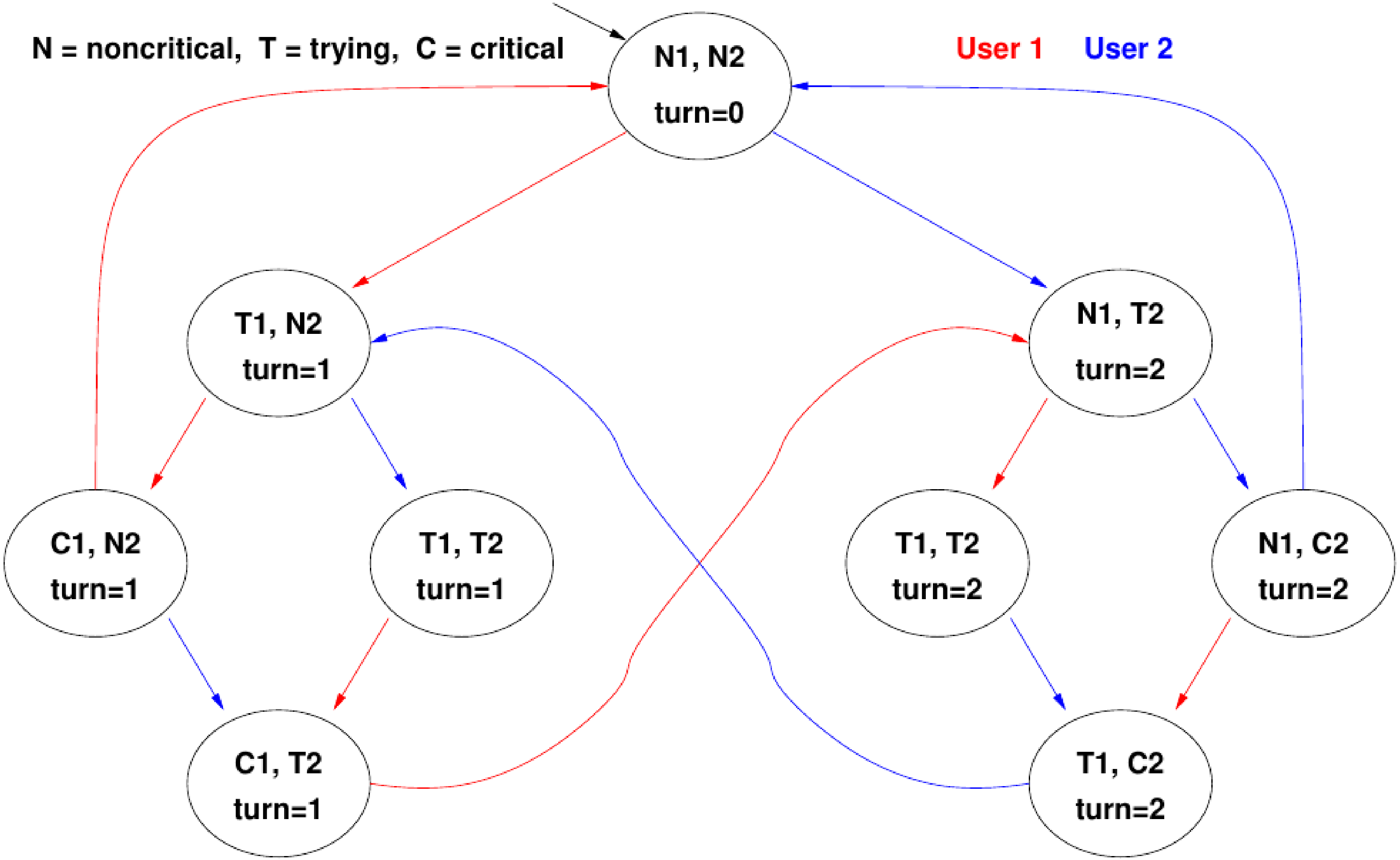
Textual	Symbolic†	Explanation	Diagram
Unary operators:			
X ϕ	$\bigcirc\phi$	ne**X**t: ϕ has to hold at the next state.	
G ϕ	$\Box\phi$	**G**lobally: ϕ has to hold on the entire subsequent path.	
F ϕ	$\Diamond\phi$	**F**inally: ϕ eventually has to hold (somewhere on the subsequent path).	
Binary operators:			
ψ **U** ϕ	$\psi\,\mathcal{U}\,\phi$	**U**ntil: ψ has to hold *at least* until ϕ, which holds at the current or a future position.	
ψ **R** ϕ	$\psi\,\mathcal{R}\,\phi$	**R**elease: ϕ has to be true until and including the point where ψ first becomes true; if ψ never becomes true, ϕ must remain true forever.	 



$$\models \quad \Box\,(\mathit{close} \Rightarrow \Diamond\,\mathit{isClosed})\ ?$$

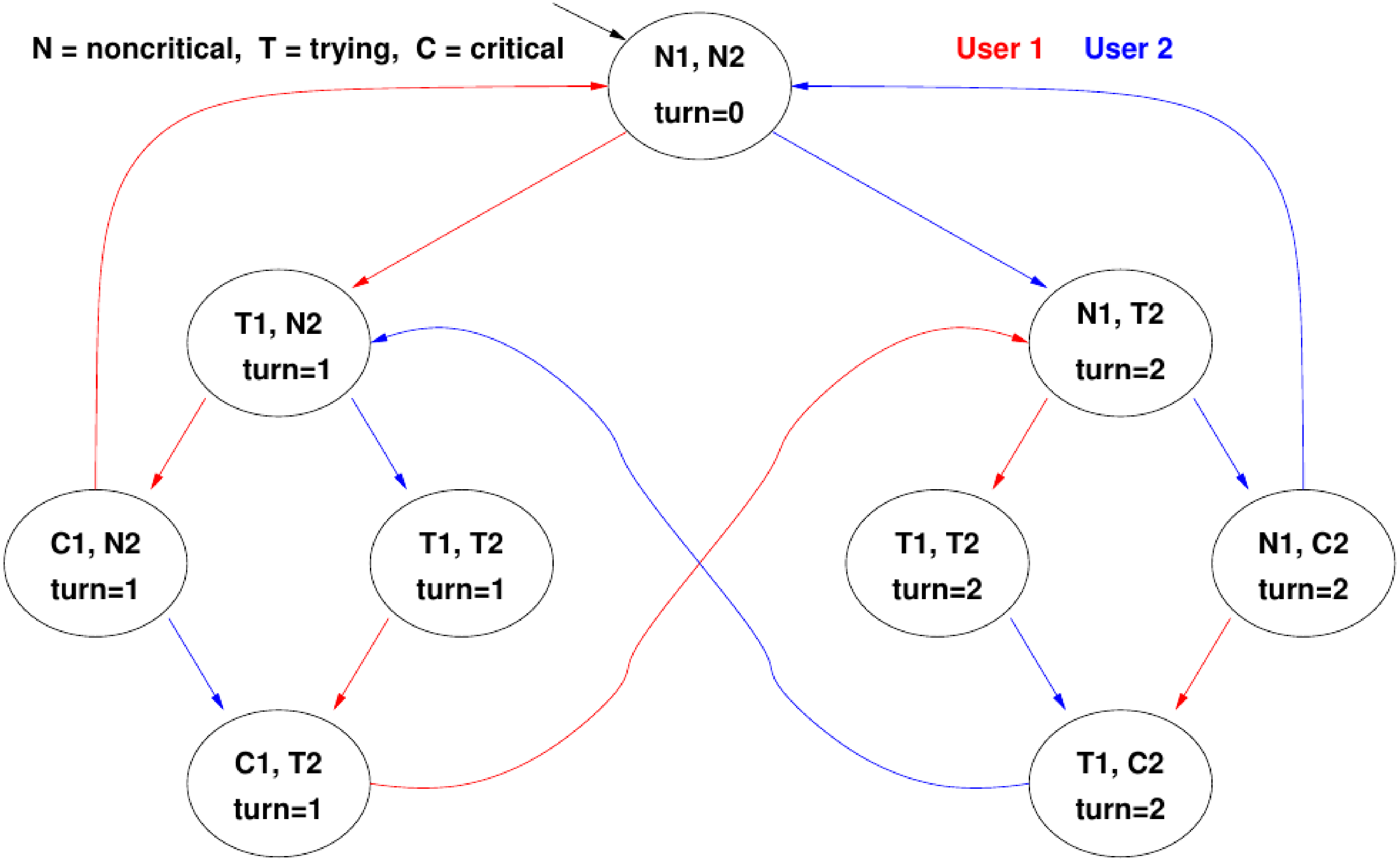
Mutual exclusion system

taken from lectures of Alessandro Artale



Mutual exclusion system

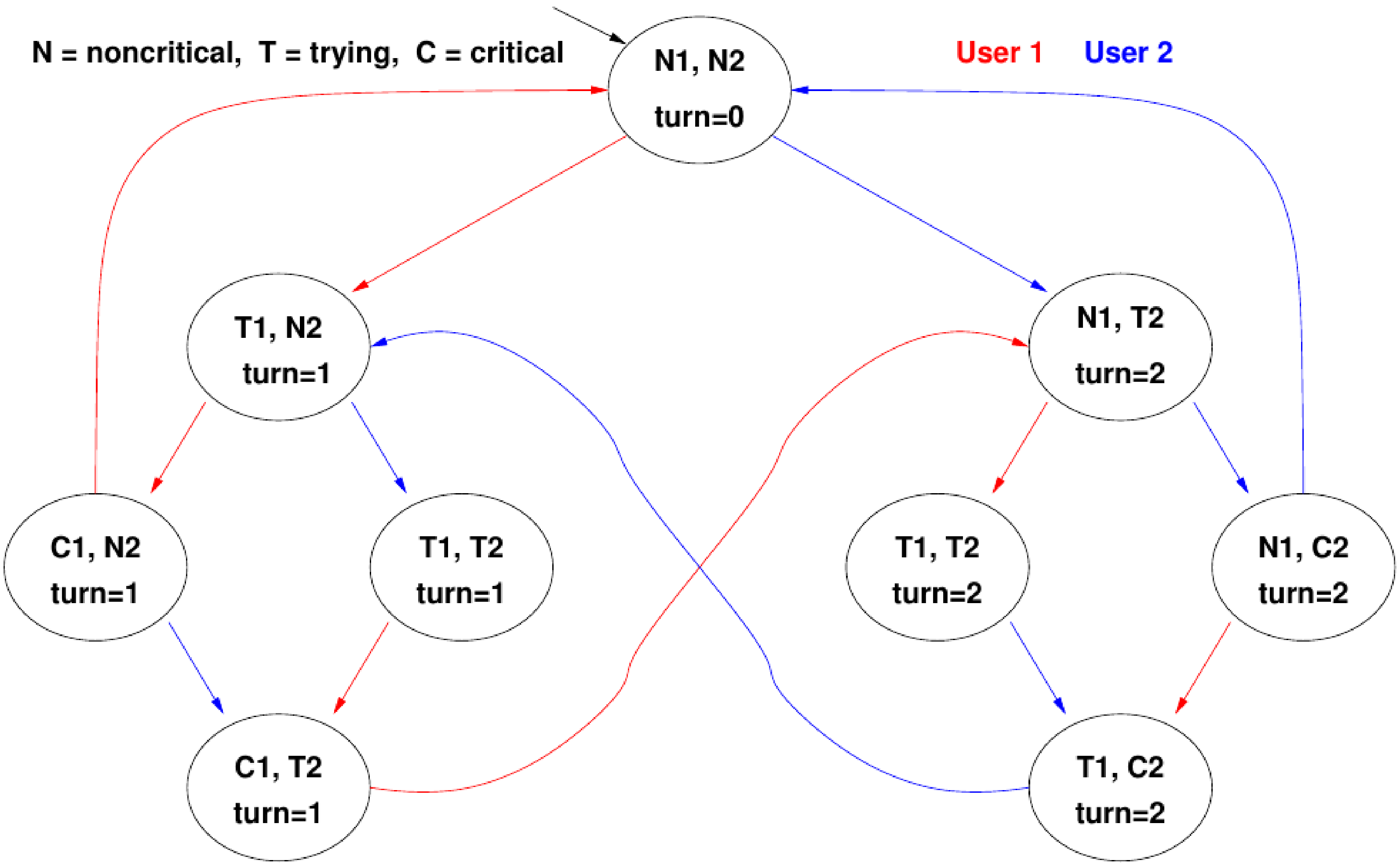
taken from lectures of Alessandro Artale



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

Mutual exclusion system

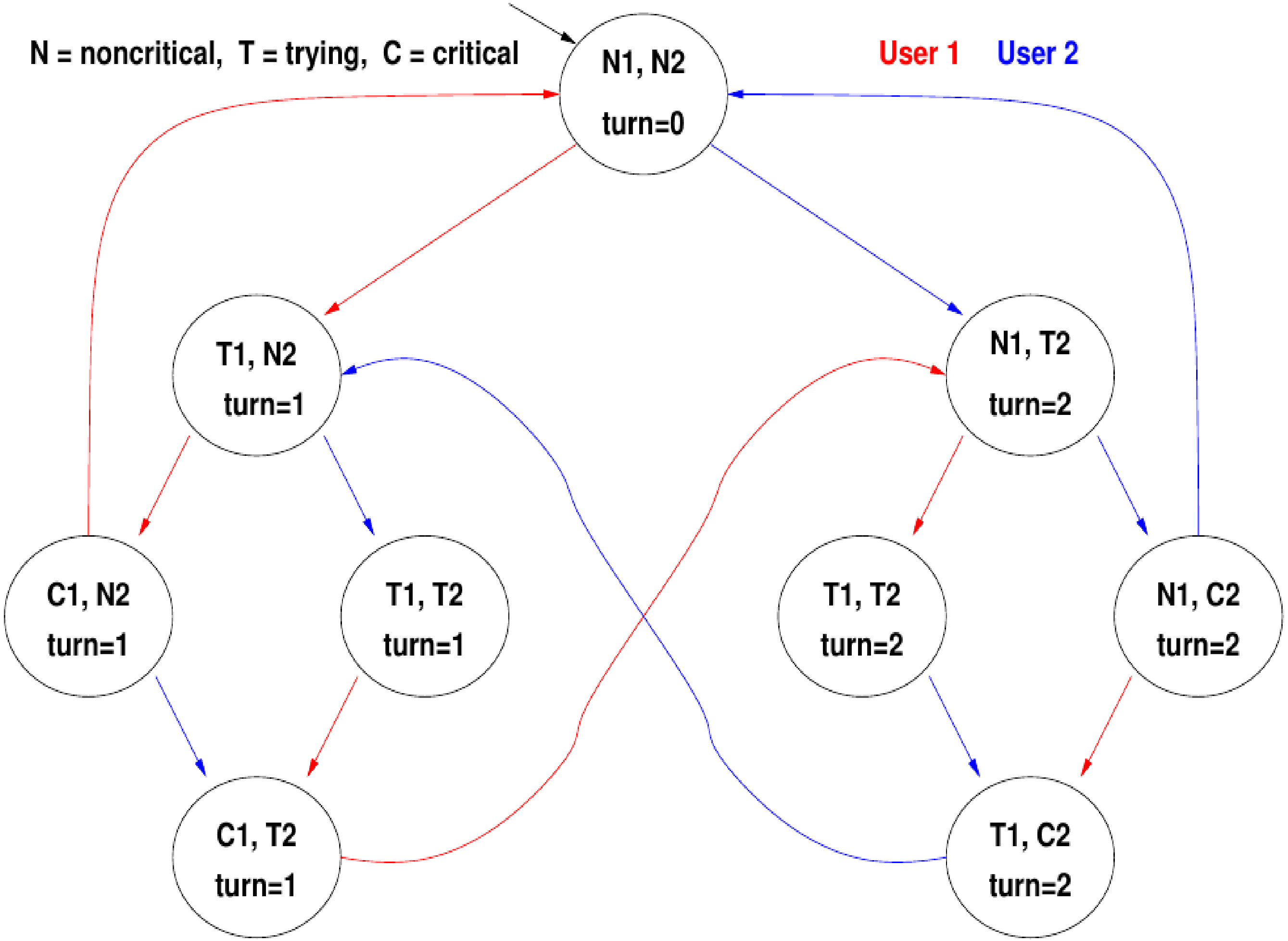
taken from lectures of Alessandro Artale



$$\models \Diamond C_1 \ ?$$

Mutual exclusion system

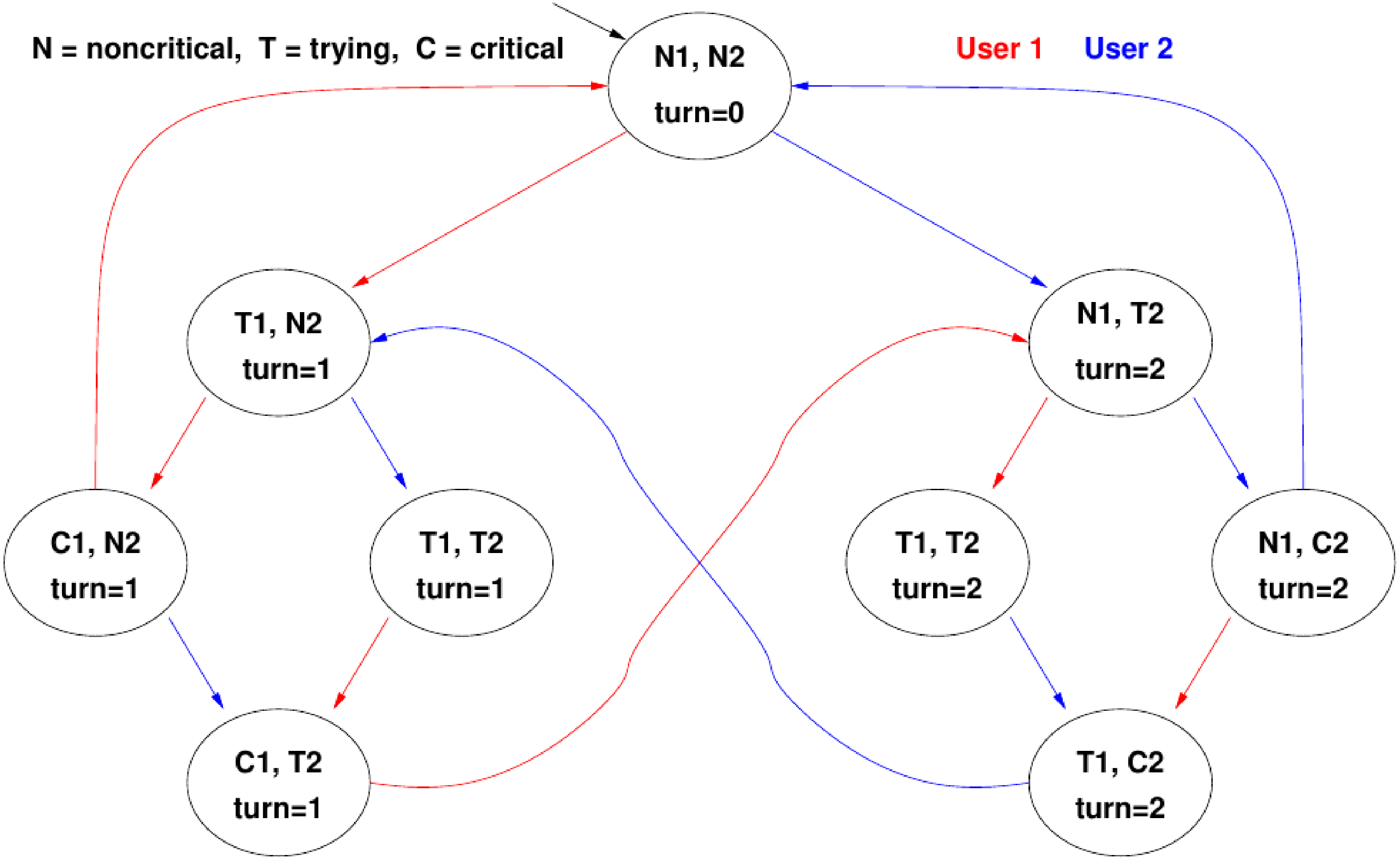
taken from lectures of Alessandro Artale



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

Mutual exclusion system

taken from lectures of Alessandro Artale



Il existe de nombreuse forme en langage naturel contraint des logiques temporelles (pattern de Dwyer, de Cheng, de Dhaussy)