Faculty of Computer Systems & Software Engineering

Formal methods.

Verification by model checking

Vitaliy Mezhuyev

Introduction

- Verification: checking correctness of computer systems
  - hardware, software or its combination
- It is most principle in
  - safety-critical systems
  - mission critical
  - commercially critical

Verification techniques

- A formal verification techniques includes

1. A specification language
   - for describing the properties to be verified
2. A verification method
   - To check if the description of the system satisfies its specification
3. A framework for modeling
   - To support a user in system specification

Classification

- Approaches to verification can be classified in several ways:
  - Proof-based vs. model-based
  - Degree of automation
    - From fully automated to fully manual
  - Full- vs. property- verification
    - The specification may describe a single property of a system, or it may describe its full behavior
  - Domain of application
  - Pre- vs. post- use at development cycle

Intended domain of application

- Hardware, software
- Sequential, concurrent
- Reactive , terminating
  - Reactive: reacts to its environment, and is not meant to terminate (e.g. operating systems, embedded systems, computer hardware)

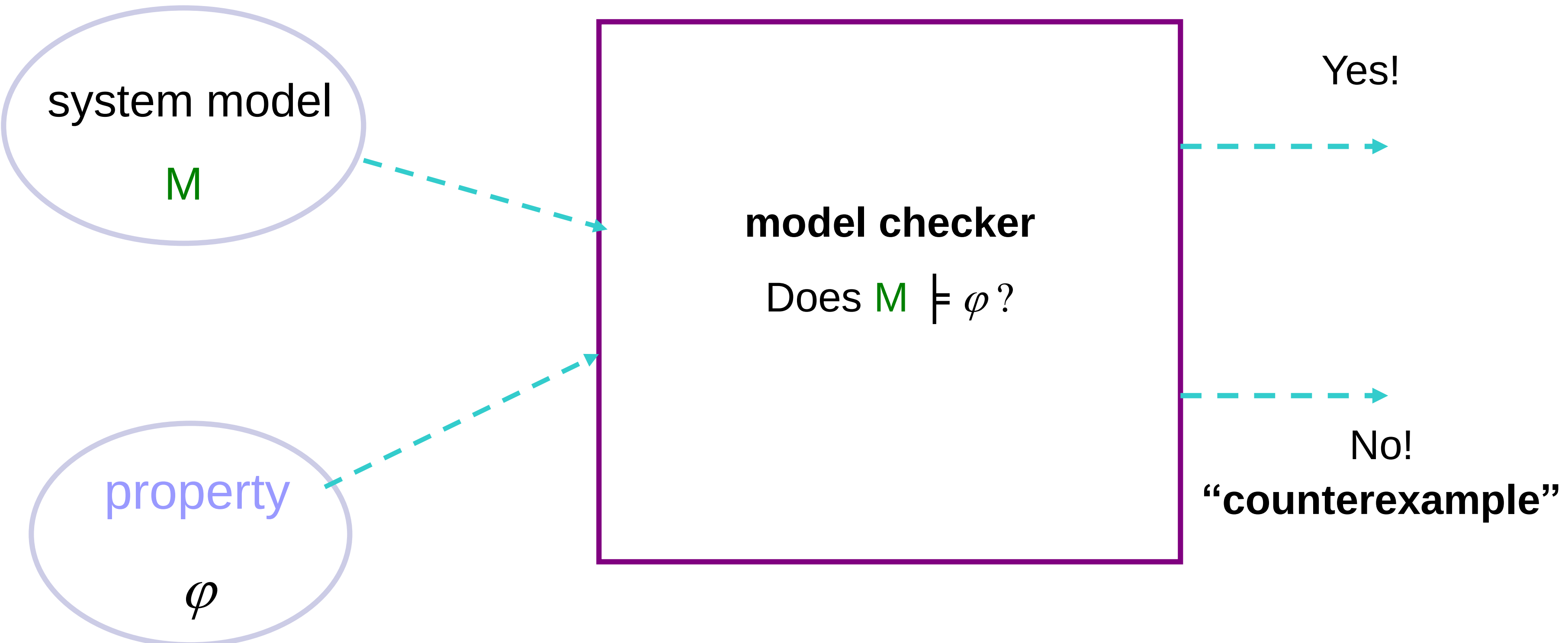
Proof-based verification

- A system description is a set of formulas Γ in some logic
- A specification is another formula φ
- The verification method is finding a proof that $\Gamma \vdash \varphi$
  - $\vdash$ means deduction
- Application of proof based verification needs high user expertise

Model-based verification

- The system is represented by a model $\mathcal{M}$ in some (appropriate) logic
- The specification is also represented by a formula φ
- The verification method consist of ***computing*** whether a model $\mathcal{M}$ satisfies φ
  - $\mathcal{M}$ satisfies φ : $\mathcal{M} \models \varphi$
- The computation is usually **automatic** for *finite* models

basic picture of model checking



Where do we get the system model?

hardware

e.g., Verilog or VHDL, source code

abstraction & other (semi-)automated transformations

software

e.g., C, C++ , Java, etc. source code

hand-built design models

state machine-based system model

Where do we get the properties?

requirements documentation

+

insights

formal properties

(typically based on temporal logic or state automata)

standard properties

e.g., deadlock, leavelock freedom

Model checking

- Model checking is an *automatic*, *model-based*, *property-verification* approach
- It is intended to be used for *concurrent* and *reactive* systems
  - The purpose of a reactive system is not necessarily to obtain a final result, but to maintain some interaction with its environment
- Concurrency bugs
  - non reproducible
  - not covered by test cases

Development of a system model: what do we want to model?

- systems have a **state** that evolves over time.
- they manipulate data, accessed through **variables**, whose values change as the state changes.
- **concurrency**: systems have interacting **processes**
  - asynchronous/synchronous,
  - message passing or shared data communication**.**
- dynamic memory allocation, process creation, procedure call stack, clocks and real time, etc.

Models have to be:

- show as many relevant aspects of real systems as possible
- be amenable to efficient algorithmic analysis.

Temporal Logic

- The idea is that a formula is not *statically* true or false in a model
- The models of temporal logic contain several states
  - a formula can be true in some and false in the others
- The static notion of truth is replaced by a *dynamic* one
  - the formulas may change their truth values as the system evolves from state to state

Why use temporal logic to specify properties?

- [Pnueli'77] and others recognized that correctness assertions for **reactive** systems are best phrased in terms of occurrence of events during the entire, potentially indefinite, execution of the system. Not just what it outputs when it halts.

- Indeed, systems like the *Windows OS* aren't really supposed to "halt and produce output". Rather, they should forever **react** to stimuli from their environment in a "*correct*" manner.

- What is temporal logic?
  - It is a language for describing relationships between the occurrence of events over time.
  - It comes is several dialects, e.g., Linear vs. Branching time. We will focus on propositional **Linear Temporal Logic** (**LTL**) .
  - **Example**:

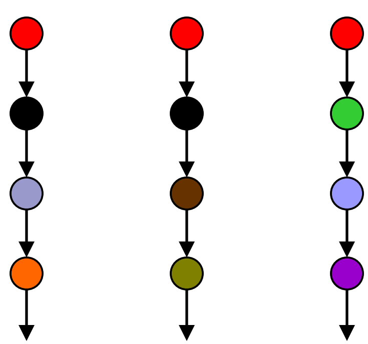
    "Until event stop occurs, every occurrence of event request is eventually followed by an occurrence of event response":

Linear vs. Branching

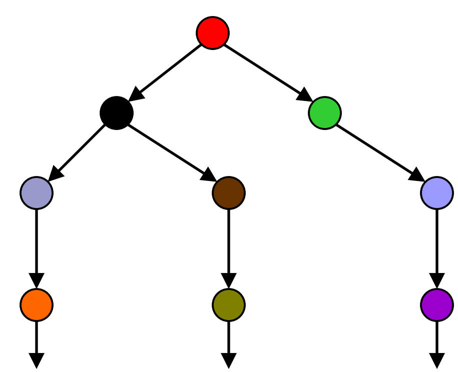
- Linear-time logics think of time as a **set of paths**
  - path is a sequence of time instances (states)
- Branching time logics represent time as a **tree**
  - it is rooted at the present moment and branches out into the future
- Many logics were suggested during last years that fit into one of above categories

Linear vs. Branching (cont.)

- Linear Time
  - Every moment has a unique successor
  - Infinite sequences (words)
  - Linear Time Temporal Logic (LTL)



- Branching Time
  - Every moment has several successors
  - Infinite tree
  - Computation Tree Logic (CTL)



LTL: **L**inear-time **T**emporal **L**ogic

- It models time as a sequence of states, extending infinitely to future
  - computation path
- The future is not determined, we should consider several paths for different futures
  - Any one of the paths can be the *actual path* that is realized

Models in Temporal Logic

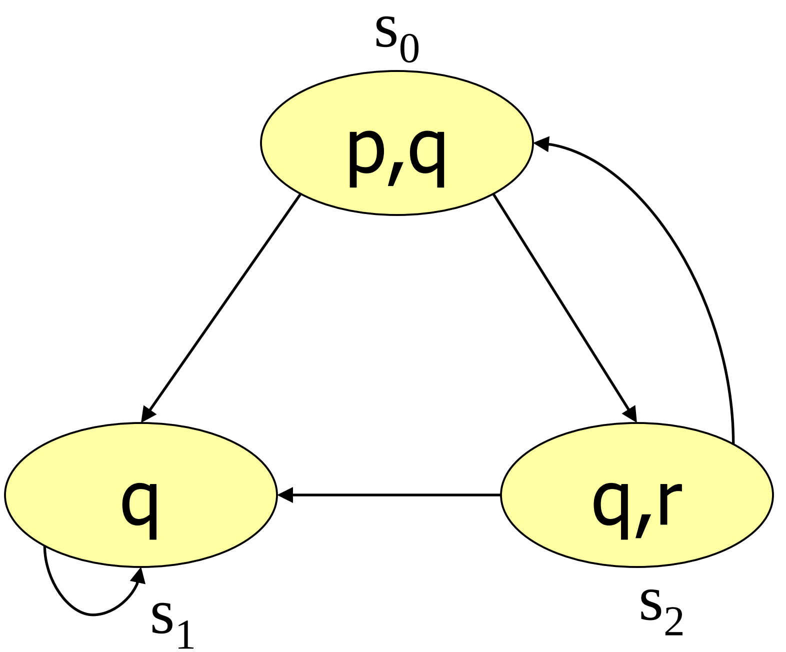
- In model checking:
  - The model M is a state transition system
    - e.g. HourClock
  - The properties φ are formulas in temporal logic
    - e.g. []HCini
- Model checking steps:
  1. Model M the system using some description language
  2. Write a property φ using the specification language
  3. Run the model checker with the inputs M and φ

Transition System

- A transition system is a structure $\mathcal{M} = (S, \rightarrow, L)$ where
  - S: a finite set of states
  - $\rightarrow$: a binary relation on S, such that every $s \in S$ has some $s' \in S$ with $s \rightarrow s'$
  - L : a labeling function $L: S \rightarrow \mathcal{P}(\mathit{Atoms})$
    - $\mathcal{P}(\mathit{Atoms})$ means the power set of Atoms

- The interpretation of the labeling function is that each state s has a set of atomic propositions $L(s)$ which are true at that particular state

Example



$S = \{s_0, s_1, s_2\}$

transitions = $s_0 \longrightarrow s_1$, $s_1 \longrightarrow s_1$, $s_2 \longrightarrow s_1$, $s_2 \longrightarrow s_0$, $s_0 \longrightarrow s_2$

$L(s_0) = \{p,q\}$

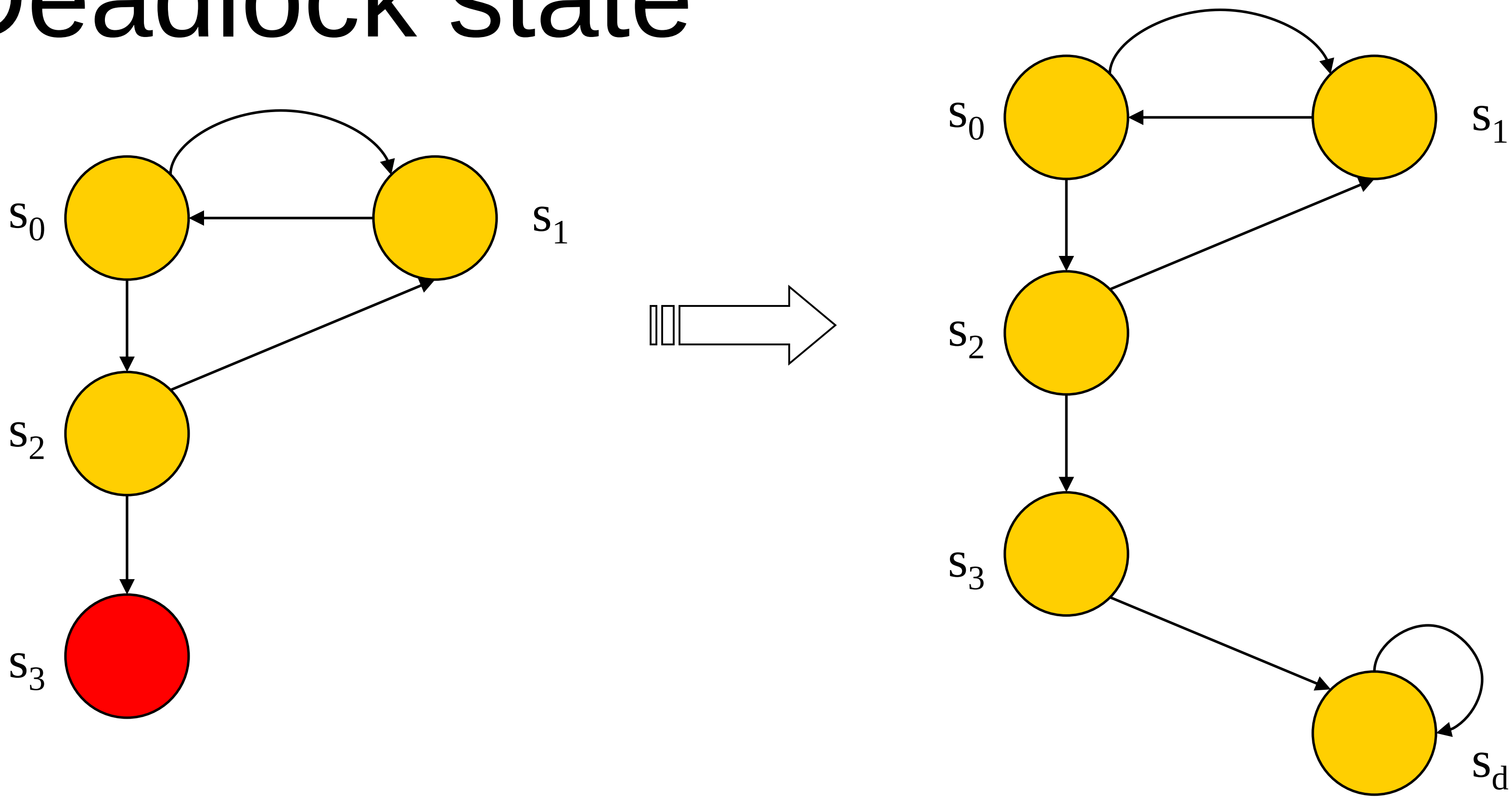
$L(s_1) = \{q\}$

$L(s_2) = \{q,r\}$

Deadlock

- Further, we will call a transition system simply a *model*
- According to the definition of a model, for each $s \in S$ there is at least one $s' \in S$
- I.e. there is should not be a deadlock state of a system
- If a system intentionally has a deadlock, lets add an extra state s_d representing it

Deadlock state



s_3 doesn't have any further transitions

adding a deadlock state s_d

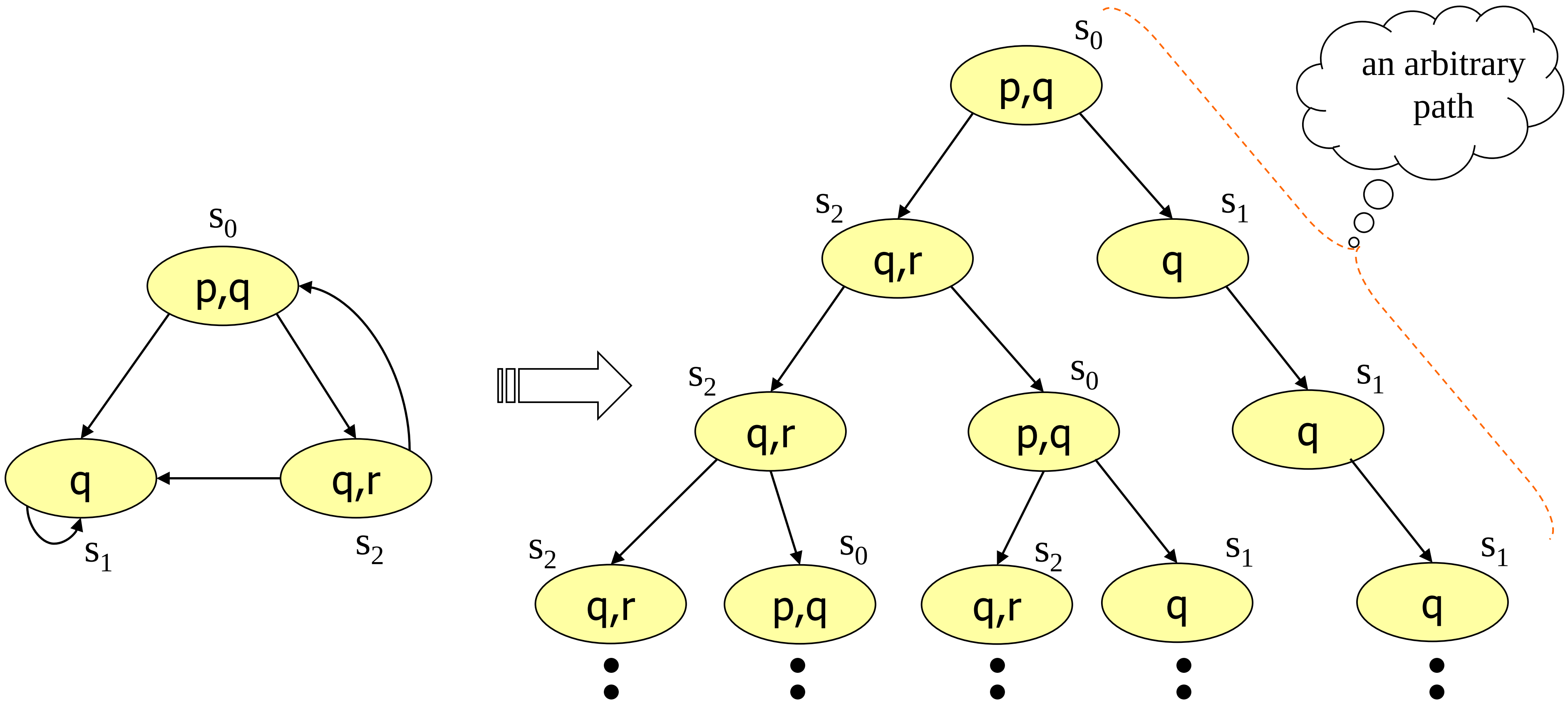
Paths and behavior

- A path in a model $\mathcal{M} = (S, \to, L)$ is an infinite sequence of states $s_1, s_2, s_3, \ldots$ in S such that, for each $i \geq 1$, $s_i \to s_{i+1}$.
- We write paths as $s_1 \to s_2 \to s_3 \to \ldots$
- Each arbitrary path (e.g. $\pi = s_1 \to s_2 \to \ldots$) represents a possible behavior of a system
    - first it is in s_1, then s_2 and so on

Building a computation tree (Unwinding)

- We can unwind the transition system to obtain an infinite computation tree
- The execution paths of a model M are explicitly represented in the tree obtained by unwinding the tree

Unwinding: example



Model checking example: Mutual exclusion

- The mutual exclusion problem (mutex)
  - Avoiding the simultaneous access to some kind of resources by use of the *critical sections* of concurrent processes
- The problem is to find a *protocol* for determining which process is allowed to enter its critical section
- Some expected properties for a correct protocol: Safety, Liveness, Non-blocking, No strict sequencing

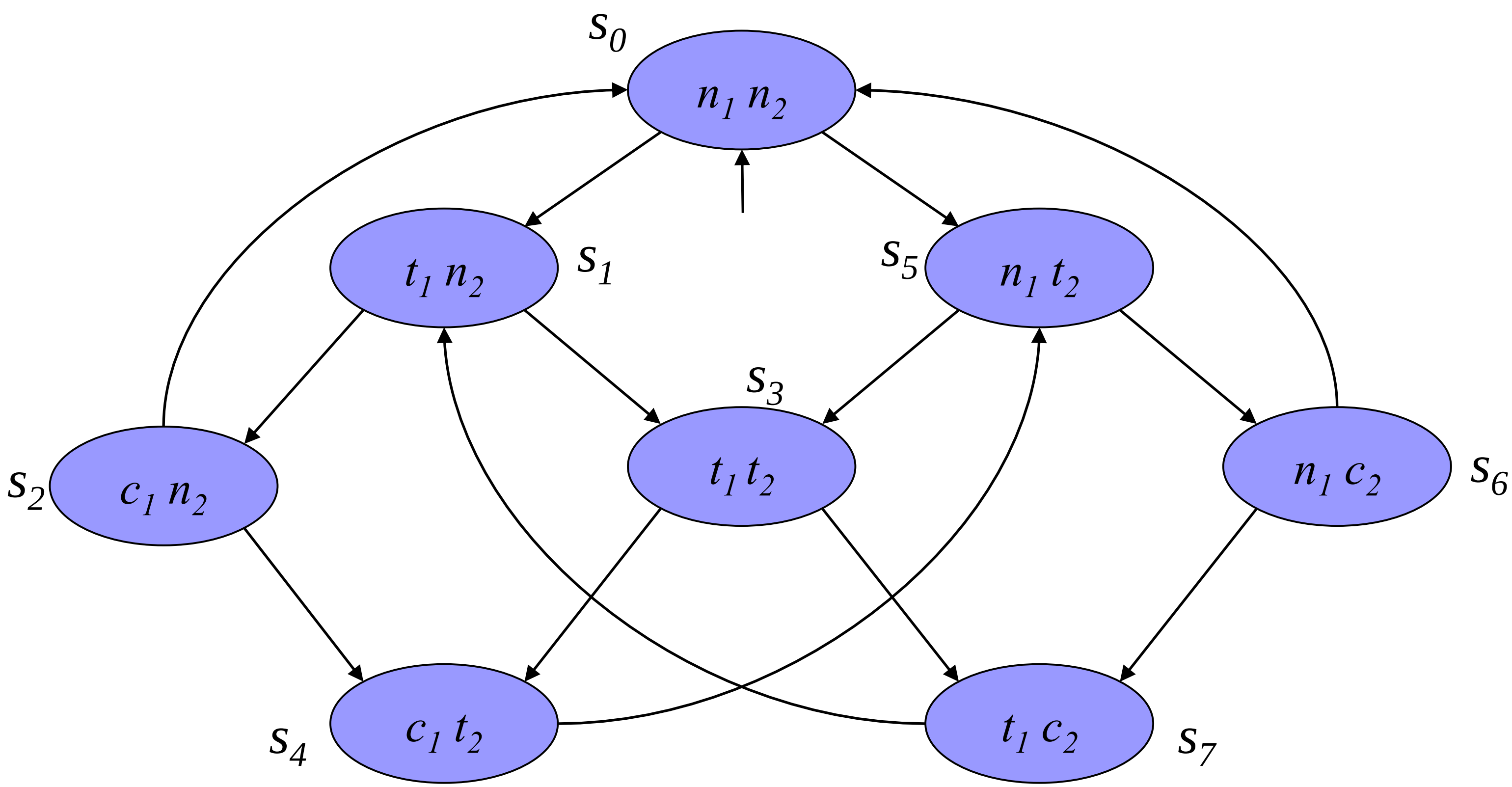
- Safety: Only one process is in its critical section at any time.
- Liveness: Whenever any process requests to enter its critical section, it will eventually be permitted to do so.
- Non-blocking: A process can always request to enter its critical section.
- No strict sequencing: Processes not need enter their critical section in a strict sequence.

Modeling mutex

- Consider each process to be either in its non-critical state *n*, trying to enter the critical section *t* or *c*
- Each individual process has this cycle:
  - $n \rightarrow t \rightarrow c \rightarrow n \rightarrow t \rightarrow c \rightarrow n \ldots$
- The processes phases are interleaved

2 process mutex

- The processes are *asynchronous interleaved*
  - one of the processes makes a transition while the other remains in its current state



- Safety: Only one process is in its critical section at any time:
  $[] \neg(c_1 \wedge c_2)$.
- Liveness: Whenever any process requests to enter its critical section, it will eventually be permitted to do so:
  $[] (t_1 \rightarrow \langle\rangle\, c_1)$.
- Non-blocking: A process can always request to enter its critical section:
  - (in CTL): $AG(n_1 \rightarrow EX\ t_1)$.
- No strict sequencing: Processes not need enter their critical section in strict sequence:
  - Expressing the negation in LTL: $G(c_1 \rightarrow c_1\ W(\sim c_1\ \&\ \sim c_1\ W\ c_2))$

Thank you for your attention!
Please ask questions