Formal Methods in Software Development

Course Introduction Modelling Systems

Ivano Salvo

Computer Science Department

SAPIENZA
UNIVERSITÀ DI ROMA

Lesson **1**, October 5th, 2020

Lesson 0:

Course Presentation and Practical Information

About this course…

Classroom:

Monday, 16-19 prof. **Ivano Salvo – G50**
Wednesday, 12-14 prof. **Igor Melatti – Aula Alfa**
On-line: Zoom meetings

Main Topic: **Model Checking**
This part (**Monday**): mainly **theoretical** aspects

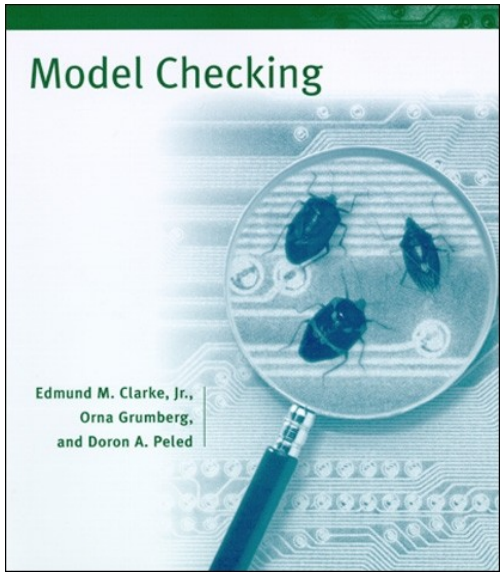
Prof. Melatti (**Wednesday**) introduce the use of several **model checkers** (murphi, nuSMV, SPIN etc.)

Website (**in progress**):
http://twiki.di.uniroma1.it/twiki/view/MFS/FormalMethodsInSoftwareDevelopment20202021

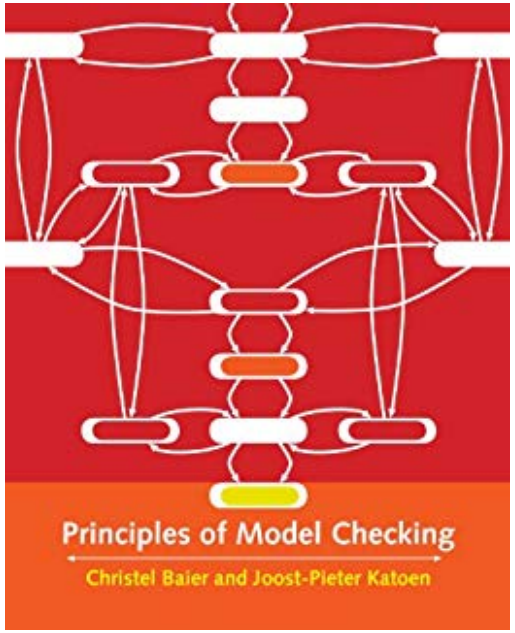
You can find course program, some additional material (slides), summary of lesson content, **previous exams**…

Course material

I will follow **mainly** the following books:



E. M. Clarke, O. Grumberg, and D. A. Peled
Model Checking
MIT press



C. Baier, J.-P. Katoen
Principles of Model Checking
MIT press

Final Examination

Written test: short questions and small exercises

+

Project/presentation:

- model and verify some toy system
- short presentation of a research paper

Lesson 0b

Course Introduction

The Need for Formal Methods

Reliance on ICT systems are growing quickly. We daily interact with hundreds of ICT systems. System errors may cause:

- Increase production costs
- Increase time-to-market
- Loss of money (**mission critical**)
- Threaten human life or environment (**safety critical**)

The reliability of ICT systems is a key issue in the system design process.

Program correctness: testing

Naïve approach: write a program and test if it produces the expected results

A bit of ingenuity:

test corner cases
try to provide significant test-set of inputs
…

Testing is a science itself! Tons of books on **generating** (automatically) significant test sets! part of Software Engineering…

Problem: **coverage** of possible program executions

Deductive Systems

Formal approaches: write a **specification** for a (sequential) program: $\forall x: Prec(x) \exists y. PostC(x, y)$

Prove formally that the program computes a function f such that: $\forall x: Prec(x). PostC(x, f(x))$

Several techniques: development of correct programs using **program assertions** (Dijkstra), **Hoare Logic**, …

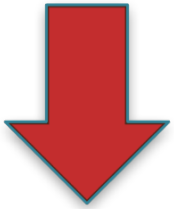
Problems: hard and **time-consuming**, requires **deep skills**, hard for large systems… even using software tools such **proof assistant** (Coq, Isabelle…)

Systems, not just Programs

ICT systems are much **more than just programs**

They consist of **many interacting components** (both **hardware** and **software**)

They interact with an **environment** (sensors, …)



The **verification problem** is quite **hard** and **system complexity** increase continuously

Model Checking

Modeling: find a **formal** **model** $\mathcal{M}$ of a system (usually via some abstract formalism, e.g. **Transition Systems**)

Specification: give a **formal** **specification** φ (first order logic is ok for sequential programs, but some kind of **Temporal Logic** is more suitable for concurrent or hybrid systems)

Verification: run a **formal** **verification** that the system $\mathcal{M}$ satisfies φ, $\mathcal{M} \models \varphi$ by **examining all states** in the computations of $\mathcal{M}$ (by means of efficient algorithms).

Result: **OK** or a **counterexample** useful to **refine** the model (or the specification).

Model Checking: Strength

- ✓ Quite **general** approach that is suitable for many applications.
- ✓ It supports **partial verification**, i.e. properties that can be checked individually
- ✓ It is **not vulnerable to expectation** on where an error can occur
- ✓ It provides **diagnostic information** (**counterexamples**) that helps debugging
- ✓ At least in principle: completely **automatic**
- ✓ It can be integrated in the development cycle and experimental studies support this.
- ✓ It is based on a **solid theory**: logics, graph algorithms

Model Checking: Weakness

- ✓ Adapt to **control intensive** applications (rather than data intensive). Example: **protocols**
- ✓ Some **decidability issues** (in particular for **infinite state systems**)
- ✓ It applies to **models** rather than systems
- ✓ It suffers from **state-explosion problem**: many systems are huge with respect to their description via a program
- ✓ **Expertise** on finding appropriate specifications and abstractions is **required** (not just **push the botton!**)
- ✓ **Does not allow generalizations**. Example: systems with an **arbitrary number of components**

Lesson 1a

Modeling Systems 1: Transition Systems

Concurrent Systems

A **concurrent system** is a **set of components** that execute together

They can evolve **independently** (**asynchronous** or **interleaved** executions) or evolve **synchronously** (all components evolve simultaneously)

Communication among components can take place via **shared variables** or by **exchanging messages** (handshaking)

(Labeled) Transition Systems

Let AP be a set of **atomic proposition**. A **Labeled Transition System** M over AP is a tuple $(S, A, S_0, \rightarrow, L)$, where:

- S is a set of **states**
- A is a set of **actions**
- $S_0 \subseteq S$ is the set of **initial states**
- $\rightarrow \subseteq S \times A \times S$ is the **transition relation**
- $L : S \rightarrow 2^{AP}$ is the **labeling function**

Modeling Concurrent Systems

We model concurrent systems by means of (**Labeled**) **Transition Systems** (**LTS**): **directed graphs** where **nodes** model **states** and **edges** model **transitions** (state changes)

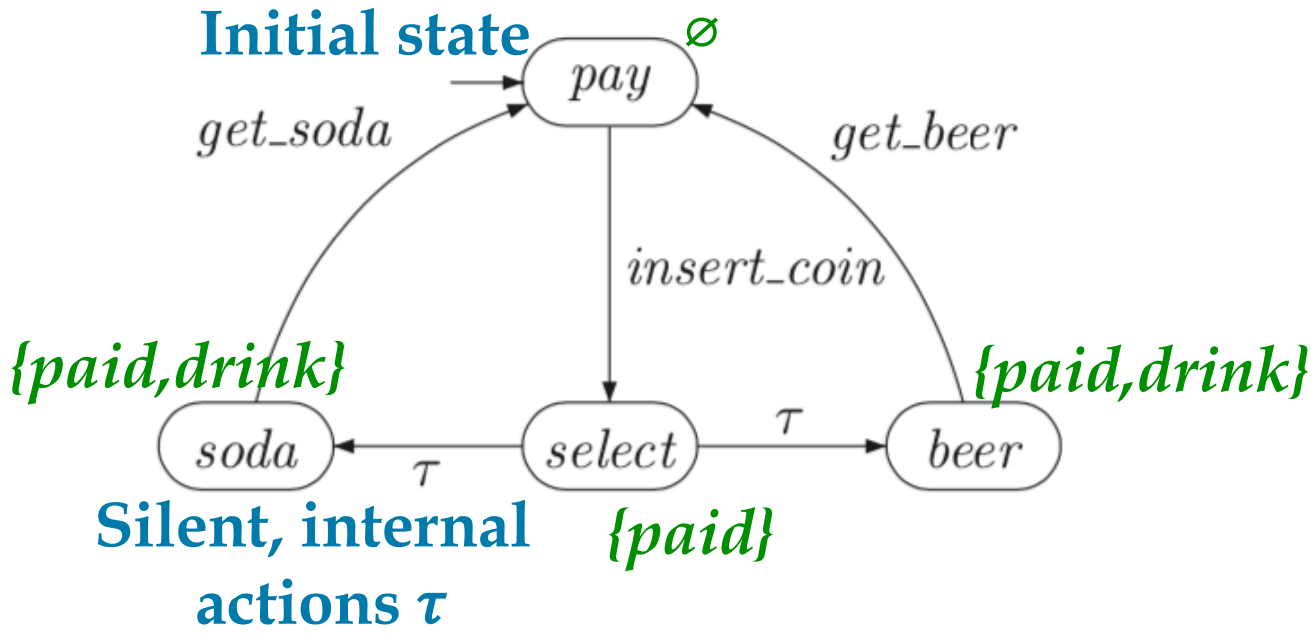
States record information about the system in **a certain moment**. **Transitions** (actions) specify **evolution of the system**

Question: Which are states and transitions of a traffic light? A program? A digital circuit? A chess game?

Action names are used mainly for **communication** between components of a system

Atomic propositions formalize **logical properties** of states (what **is really relevant** of a state wrt our verification task)

Ex.: Beverage Vending Machine



In this model, the machine **non-deterministically** delivers a soda or a beer.

One can prove **properties** such as: *"The vending machine only delivers a drink after inserting a coin"*

LTS: Semantics

A **path** (or **execution fragment**) from s in M is a sequence $\pi = s_0 a_1 s_1 a_2 s_2 \ldots a_n s_n$ such that $\mathbf{s_0 = s}$ and $\mathbf{s_i \rightarrow_{a_i} s_{i+1}}$

A path is **initial** if $s_0 \in S_0$ (i.e. it starts in an initial state). It is **maximal** if it is either **infinite** or the last state s_n has **no outgoing transitions**

An **execution of *M*** is an **initial** and **maximal** path

A state is **reachable** if it belongs to an execution of M

$\rightarrow$ is **total** if for each state s there exists always a, s' such that $s \rightarrow_a s'$ (shorthand for $(s, a, s') \in \rightarrow$)

Non-determinism

We define the set of **immediate successors** and **predecessors** of a state:

$Post(s, a) = \{s' \mid s \rightarrow_a s'\}$ and $Post(s) = \bigcup_{a \in A} Post(s, a)$

$Pred(s, a) = \{s' \mid s' \rightarrow_a s\}$ and $Pred(s) = \bigcup_{a \in A} Pred(s, a)$

A state s is **terminal** state if $Post(s) = \varnothing$

A system is (**action**) **deterministic** if $|Post(s, a)| \leq 1$ for all states s and for all actions a

Nondeterminism is a **matter of abstraction**!

- **Unpredictable** interleaving of concurrent processes
- **Underspecified** models
- Interaction with an **uncontrollable environment**
- …

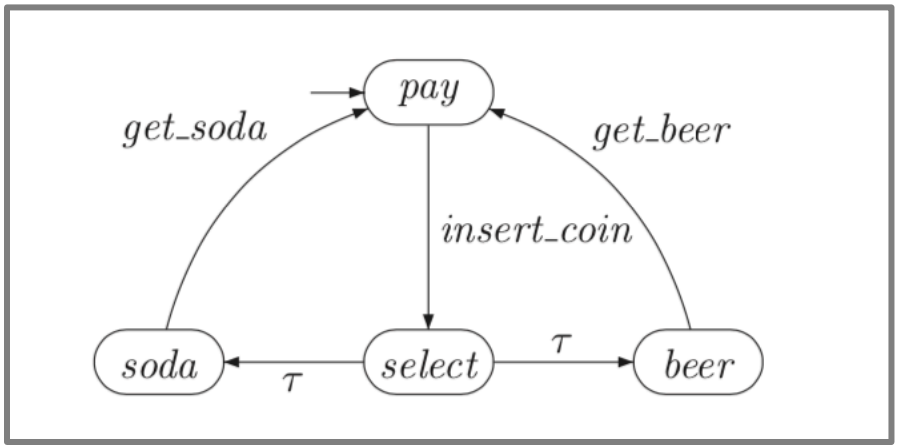
Examples from the Beverage Vending Machine

$\rightarrow$ is **total**: the machine **is always ready for interactions** (new input from the environment) only **infinite paths** are **maximal**, in this case

Initial paths start in the state *pay*

In the state *select* there are two **non-nodeterministic silent transitions** τ: silent transitions model internal (= **non observable**) evolutions of the system

All states are reachable



Modeling: not just input/output

Observation: differently from sequential programs, we are **not** interested **just** in the **input/output function defined by a system**

We are rather interested in properties that rely on:

- **Reachable** states
- **Sequence of actions** in some execution
- **Interactions** offered to other systems and or environment
- Fairness
- Liveness

…

Lesson 1b

Modeling Systems 2: Data Dependent Systems

Data Dependent Systems

Usually, systems are described by **kind of programs**, that in turn **depend on** (potentially infinite) **data**

Transitions can depend on some **conditions**: **this is not** in the framework of Transition Systems

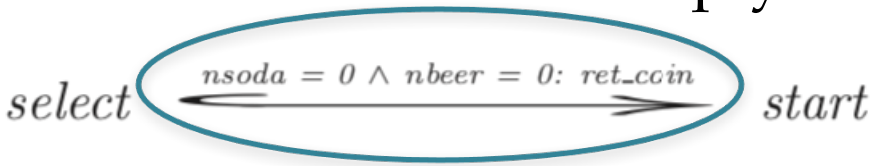
Conditional branching can be modeled by nondeterminism, but this can lead to **very abstract** (= **not useful**) models

In the following we see **programs that generate** a Labelled Transition System

For example, the SPIN model checker use the **ProMeLa** language to describe systems

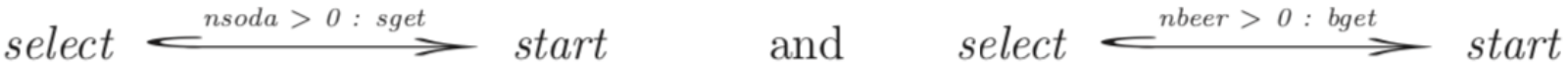
Bev. Vending Machine Reloaded

Extended **Beverage Vending Machine:** the model includes the **number of available beverages**: it returns the inserted coin when it is empty



conditional transitions

The machine has an action **refill** to **insert bottles**. One can get a bottle only if the machine is not empty



One can always **refill** or **insert coin**:

Action	Effect
refill	$nsoda := max;\ nbeer := max$
sget	$nsoda := nsoda - 1$
bget	$nbeer := nbeer - 1$

***max* is the maximum capacity**

Generalising: Program Graphs

A **program graph** *PG* over a set ***Var*** of **typed variables** is a tuple $(Loc, Act, Effect, \hookrightarrow, Loc_0, g_0)$, where:

- *Loc* is a set of **locations**
- *Act* is a set of **actions**
- $Effect: Act \times Eval(Var) \rightarrow Eval(Var)$
- $\hookrightarrow \subseteq Loc \times Cond(Var) \times Act \times Loc$ is the **conditional transition relation**
- $Loc_0 \subseteq Loc$ is the set of **initial** locations
- $g_0 \subseteq Cond(Var)$ is the **initial** condition

Eval(Var) is the set of variable evaluation

Cond(Var) is the set of conditional expressions over Var

Unfolding of a PG into a LTS

States are pairs of the form **(l, η)**, where **$l \in Loc$** and **η is an evaluation**

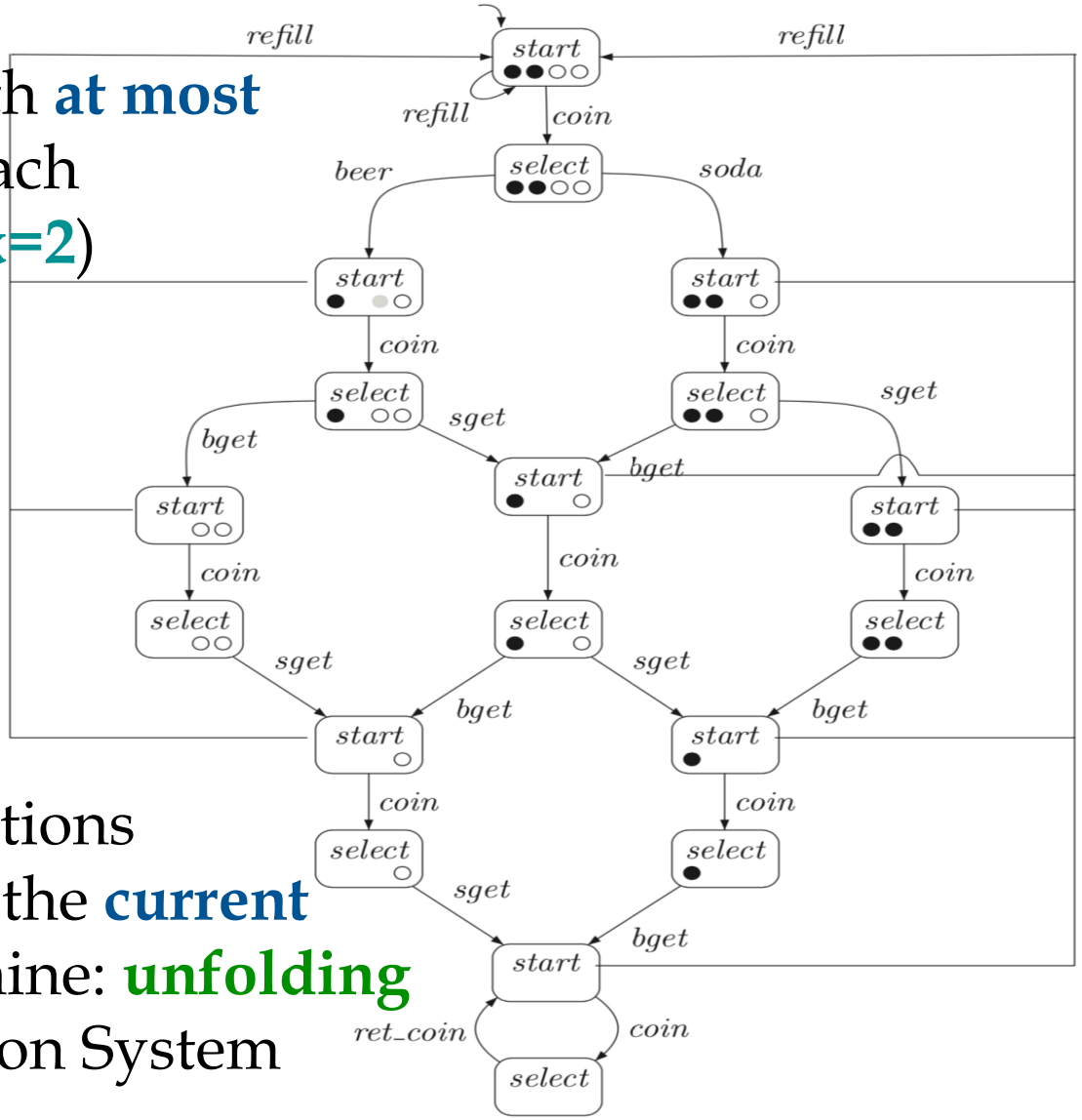
Initial states are **initial locations** that satisfy the **initial condition g_0**.

Atomic propositions are defined in terms of locations and values of variables (states)

The **transition relation $l \hookrightarrow_{g:a} l'$** produces transitions of the form $(l, \eta) \rightarrow_a (l', \eta')$, provided that **$g$ evaluates TRUE** in η and $\eta' = Effect(a, \eta)$

Bev. Vending Machine: unfolding

Here we show transitions with **at most 2 bottles** for each beverage (**max=2**)



Conditions in actions **are evaluated** in the **current state** of the machine: **unfolding** we get a Transition System

Vending Machine as PG

Var = {*nsoda, nbeer*}, whose domains are both {0,…, *max*}

Loc = {*start, select*} and Loc_0 = {*start*}.

We denote by η **evaluation** of variables.

Act = {*bget, sget, coin, ret_coin, refill*} with:

$Effect(coin, \eta) = \eta$

$Effect(ret\_coin, \eta) = \eta$

$Effect(bget, \eta) = \eta[nbeer := nbeer - 1]$

$Effect(sget, \eta) = \eta[nsoda := nsoda - 1]$

$Effect(refill, \eta) = [nsoda := max, nbeer := max]$

$g_0 \equiv nsoda = max \wedge nbeer = max$

Formally

Definition 2.15. Transition System Semantics of a Program Graph

The transition system $TS(PG)$ of program graph

$$PG = (Loc, Act, Effect, \hookrightarrow, Loc_0, g_0)$$

over set Var of variables is the tuple $(S, Act, \longrightarrow, I, AP, L)$ where

- $S = Loc \times Eval(Var)$
- $\longrightarrow \subseteq S \times Act \times S$ is defined by the following rule (see remark below):

$$\frac{\ell \overset{g:\alpha}{\hookrightarrow} \ell' \quad \wedge \quad \eta \models g}{\langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', Effect(\alpha, \eta) \rangle}$$

- $I = \{\langle \ell, \eta \rangle \mid \ell \in Loc_0, \eta \models g_0\}$
- $AP = Loc \cup Cond(Var)$
- $L(\langle \ell, \eta \rangle) = \{\ell\} \cup \{g \in Cond(Var) \mid \eta \models g\}$.

What you write inside a **model checker** is essentially a **program**. This definition shows you how to get a Transition System!

■

State Explosion Problem

As it is clear from this example, the **number of states** of the LTS is **huge** with respect to the size of the program graph

The number of states of a program graph is:

$$|Loc| \cdot \prod_{x \in Var} |dom(x)|$$

provided that ***dom(x)* is finite**

The number of states is **exponential in the number of variables**

Counteracting the state explosion problem is one of the **main research topic in Model Checking** (for example, implicit representation of states, etc.)

Lesson 1c

Modeling Systems 3: Composing Systems

Composition of Parallel Systems

Hard- and software systems are **parallel** in nature.

They are typically defined as the **parallel composition** of components that execute simultaneously:

$$M = M_1 \parallel M_2 \parallel \ldots \parallel M_n$$

Parallel composition can be used to **model systems hierarchically**: M_i can be in turn the parallel composition $M_{i,1} \parallel M_{i,2} \parallel \ldots \parallel M_{i,k}$

In the following, we briefly show different semantics of the operator $\parallel$ and how different systems can **communicate** (shared variables, handshaking etc.)

Interleaving Semantics

In **interleaving semantics**, concurrent components evolve **independently**, as they run on a **single-processor** machine with **unpredictable scheduling**

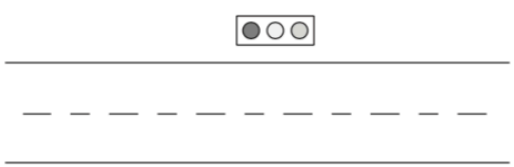
The composition contains **all possible interleaving sequences** of actions (abstracting from **scheduling)** policy)

No assumptions about order of execution (except for some **synchronization mechanism**, discussed later)

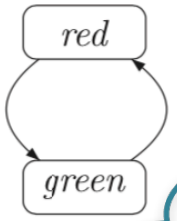
$$Effect(\alpha \,|||\, \beta, \eta) \;=\; Effect((\alpha\,;\,\beta) + (\beta\,;\,\alpha), \eta)$$

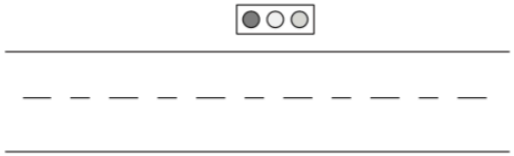
(where ; is sequential composition and + is nondeterministic choice)

Example: Independent Traffic Lights

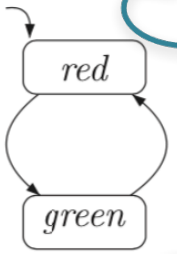


$TrLight_1$



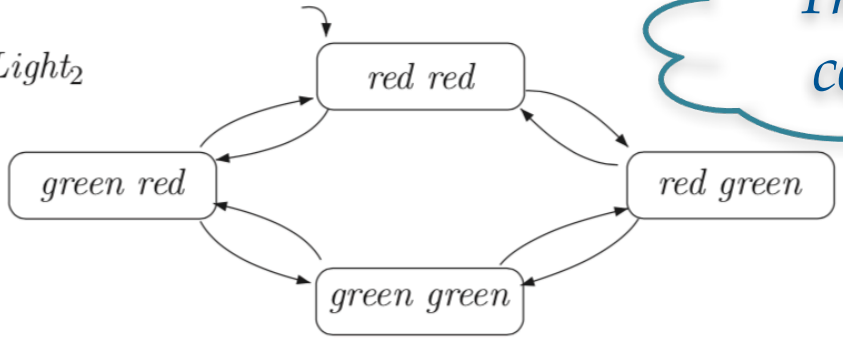


$TrLight_2$



Two LTSs

$TrLight_1 \;|||\; TrLight_2$



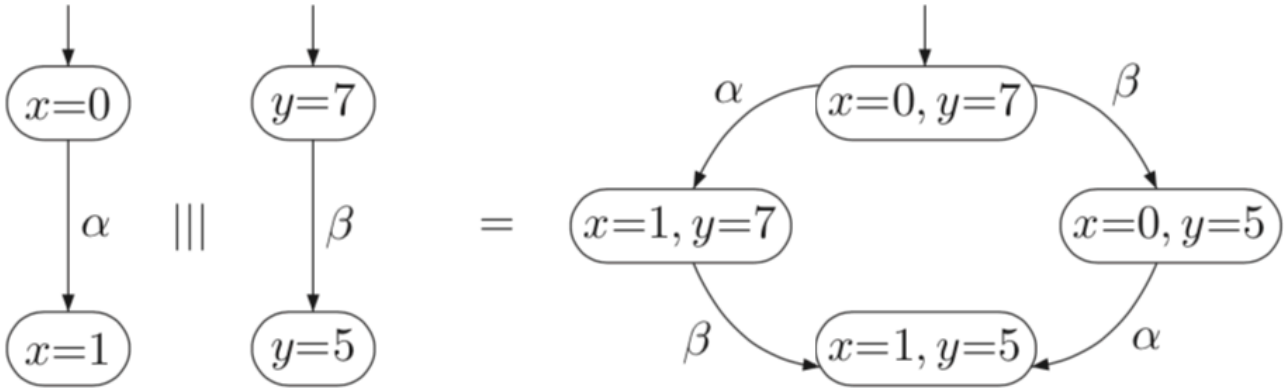
Their parallel composition

Example: Independent variables

Two processes modify **two independent** variables:

$$\underbrace{x := x + 1}_{=\alpha} \;|||\; \underbrace{y := y - 2}_{=\beta}.$$

All possible **executions** lead to the **same result**:



Interleaving of TS: definition

Definition 2.18. Interleaving of Transition Systems

Let $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP_i, L_i)$ $i{=}1,2$, be two transition systems. The transition system $TS_1 \,|||\, TS_2$ is defined by:

$$TS_1 \,|||\, TS_2 \;=\; (S_1 \times S_2, Act_1 \cup Act_2, \rightarrow, I_1 \times I_2, AP_1 \cup AP_2, L)$$

where the transition relation $\rightarrow$ is defined by the following rules:

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1'}{\langle s_1, s_2\rangle \xrightarrow{\alpha} \langle s_1', s_2\rangle} \quad \text{and} \quad \frac{s_2 \xrightarrow{\alpha}_2 s_2'}{\langle s_1, s_2\rangle \xrightarrow{\alpha} \langle s_1, s_2'\rangle}$$

and the labeling function is defined by $L(\langle s_1, s_2\rangle) = L(s_1) \cup L(s_2)$. ■

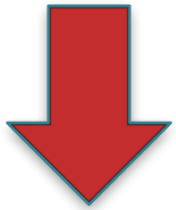
State Explosion Problem

Also **parallel composition** of systems is a **source** of **state explosion** problem.

The **state space** of the composed system is the **cartesian product** of state space of **its components**.

If $M = M_1 \parallel M_2 \parallel \ldots \parallel M_n$, then we have that:

$$|M| = \prod_{i=1,\ldots,n} |M_i|$$



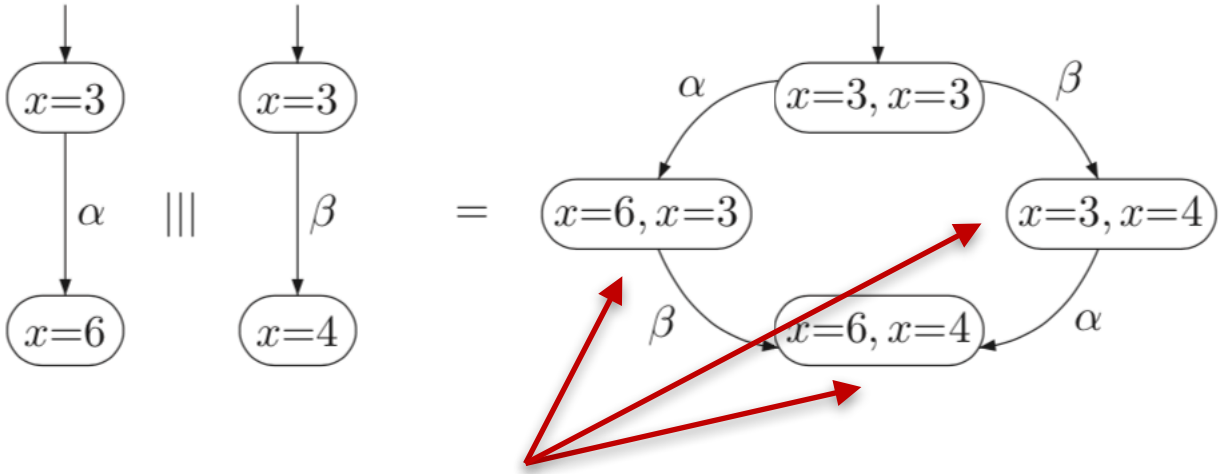
Therefore, the number of states **is exponential** in the **number of components**!

Communication: shared variables

Two processes modify the **same shared** variable:

$$\underbrace{x := 2{\cdot}x}_{\text{action } \alpha} \;|||\; \underbrace{x := x + 1}_{\text{action } \beta}$$

Interleaving is **too simplicistic** in this case!!!



Inconsistent states!

Interleaving, shared variables: def

The solution is to define the operator || at the **program graph level**, rather than transition systems.

Definition 2.21. Interleaving of Program Graphs

Let $PG_i = (Loc_i, Act_i, Effect_i, \hookrightarrow_i, Loc_{0,i}, g_{0,i})$, for $i=1,2$ be two program graphs over the variables Var_i. Program graph $PG_1 \,|||\, PG_2$ over $Var_1 \cup Var_2$ is defined by

$$PG_1 \,|||\, PG_2 \;=\; (Loc_1 \times Loc_2, Act_1 \uplus Act_2, Effect, \hookrightarrow, Loc_{0,1} \times Loc_{0,2}, g_{0,1} \wedge g_{0,2})$$

where $\hookrightarrow$ is defined by the rules:

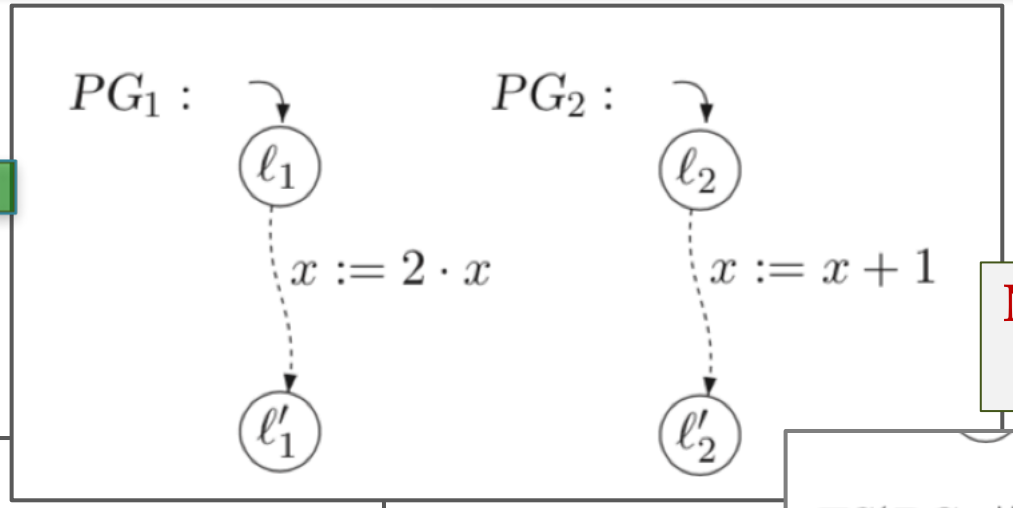
$$\frac{\ell_1 \overset{g:\alpha}{\hookrightarrow_1} \ell_1'}{\langle \ell_1, \ell_2 \rangle \overset{g:\alpha}{\hookrightarrow} \langle \ell_1', \ell_2 \rangle} \quad \text{and} \quad \frac{\ell_2 \overset{g:\alpha}{\hookrightarrow_2} \ell_2'}{\langle \ell_1, \ell_2 \rangle \overset{g:\alpha}{\hookrightarrow} \langle \ell_1, \ell_2' \rangle}$$

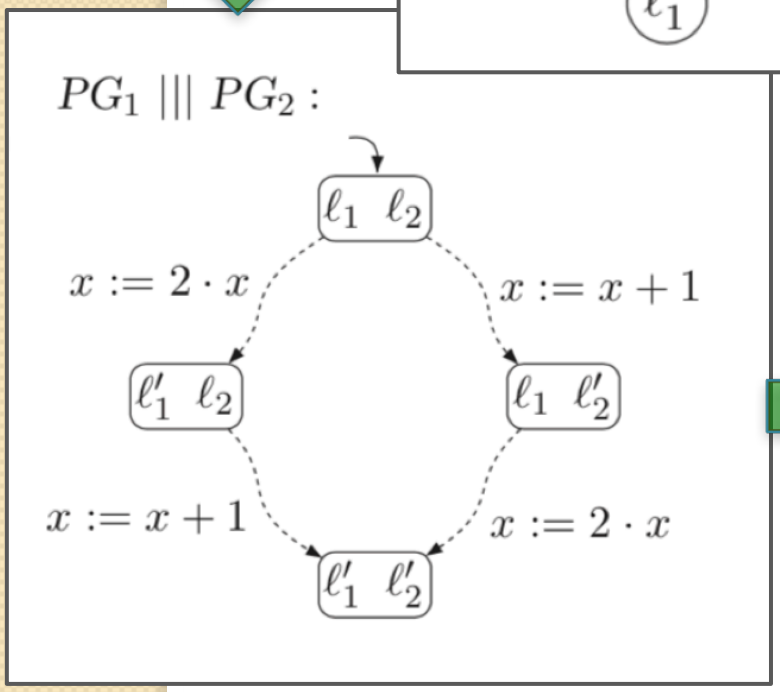
and $Effect(\alpha, \eta) \;=\; Effect_i(\alpha, \eta)$ if $\alpha \in Act_i$. ■

Effect changes **simultaneously** values of **shared variables**

Example: shared variables reloaded

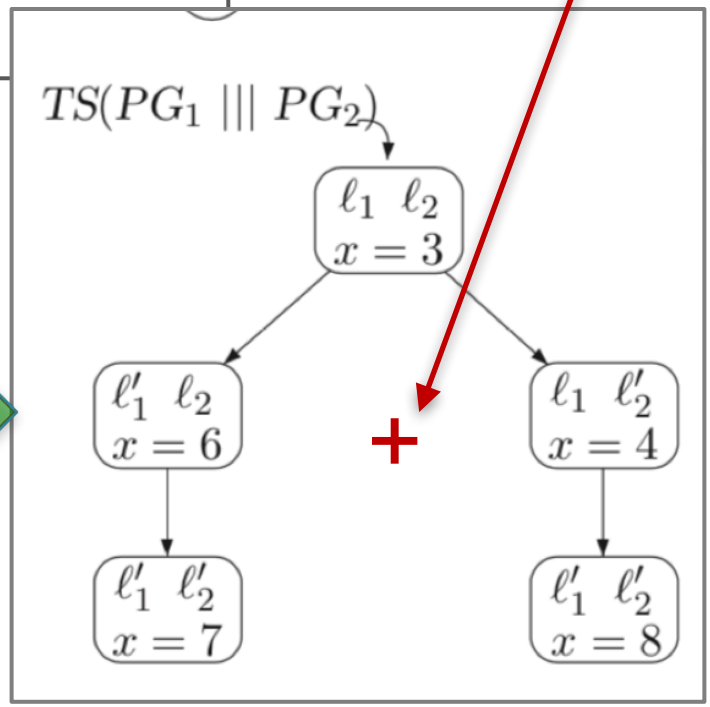
Parallel Composition





Unfolding

Nondeterministic choice



Modeling: Granularity

A **model** is always an **abstraction** of a real system.

Modeling is a critical issue.

Transitions must be **atomic: no observable** state must be ignored by the transition system.

if $x<10$ **then** $x=x+1 \parallel x=2*x$

Are $x=x+1$ or **if** $x<10$ **then** $x=x+1$ atomic? In a program **they correspond to several operations**!

Granularity:

- **too coarse**: some **errors** can be **ignored**
- **too fine**: model checking discover **spurious errors**

Granularity: Example

Let M_1 be the model described by two integer variables x and y, with two transitions:

$$\alpha: \; x := x + y \quad \text{and} \quad \beta: \; y := x + y$$

that can be executed concurrently.

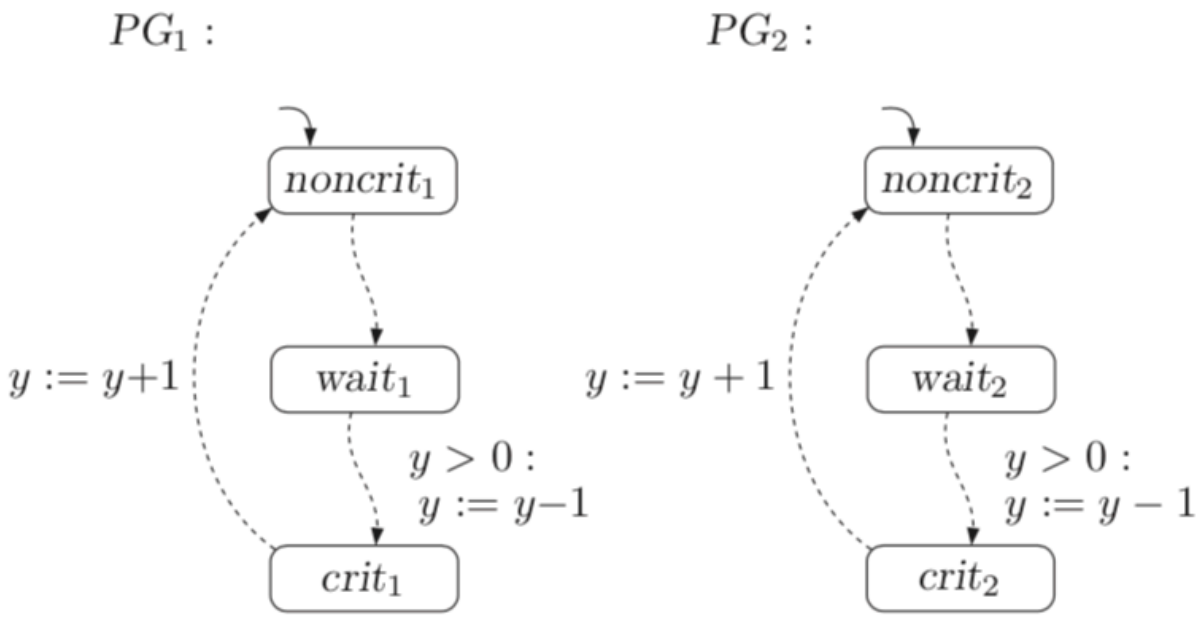
from $x=1 \wedge y=2$, the execution $\alpha\beta$ leads to $x=3 \wedge y=5$ and the execution $\beta\alpha$ leads to $x=4 \wedge y=3$.

Consider M_2 be the model of an **assembly-like implementation** of the **``same'' system** (R_i are registers):

α_0: load R_1 x	β_0: load R_2 y
α_1: add R_1 y	β_1: add R_2 x
α_2: store R_1 x	β_2: store R_2 y

In M_2, we have **more execution orders**, for example α_0 β_0 α_1 β_1 α_2 β_2 that leads to the state $x=3 \wedge y=3$.

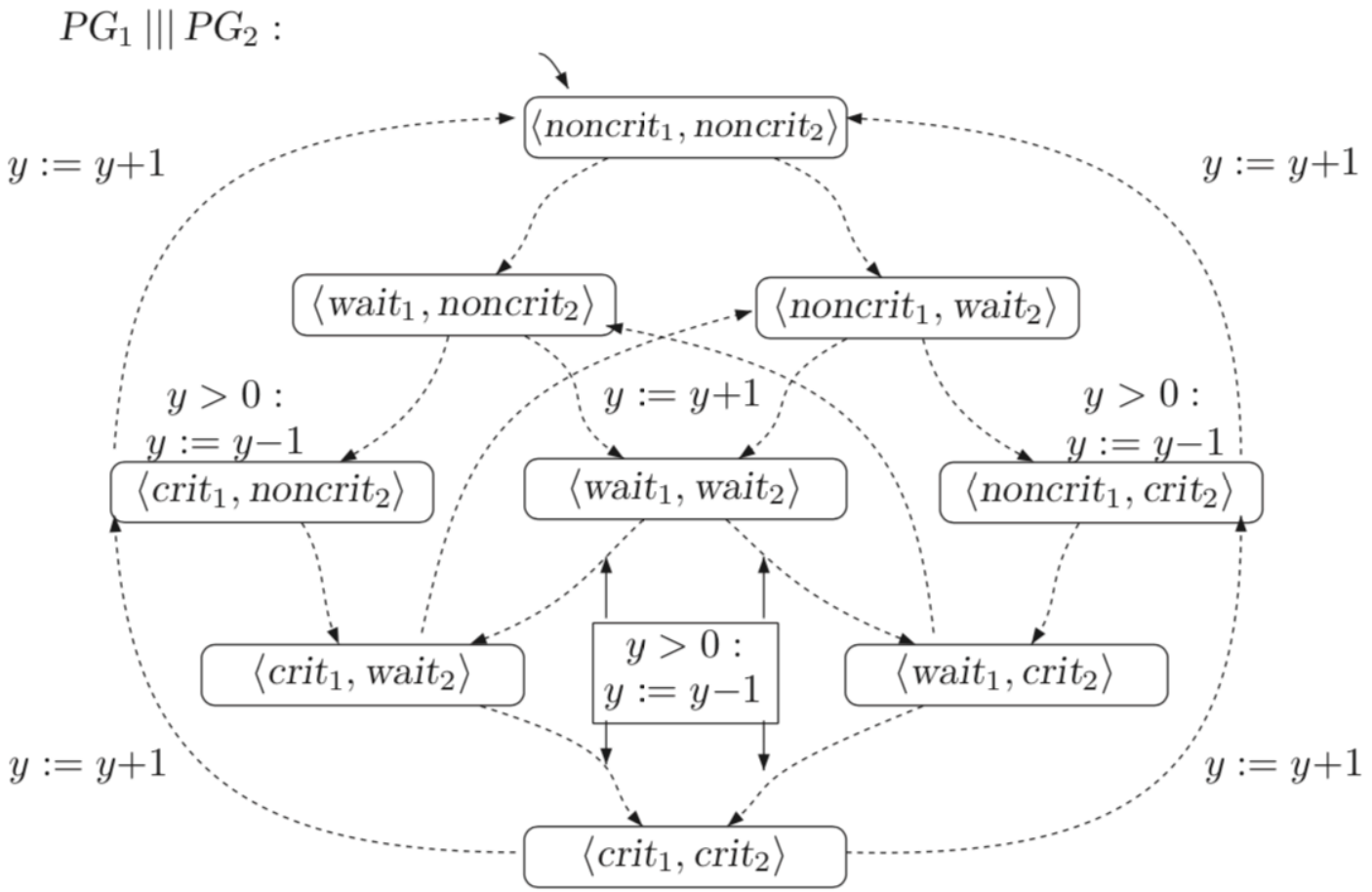
Mutual Exclusion via Semaphores



The **shared variable** y implements a **semaphore**, preventing both processes to enter the critical section simultaneously

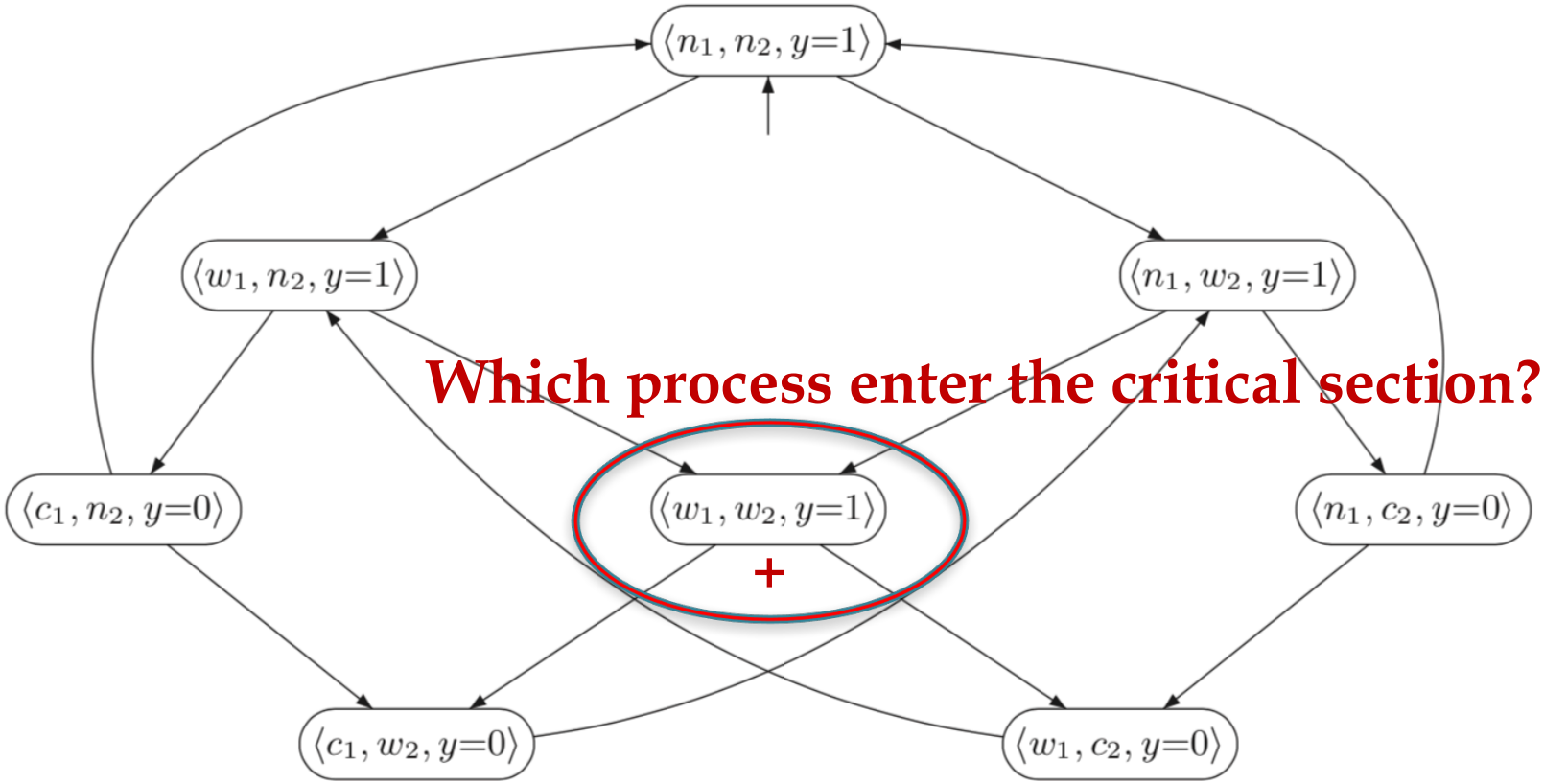
Observation: y := y-1 cannot be executed in parallel (**critical actions** involving shared variables)

Mutual Exclusion via Semaphores



Interleaving of **program graphs** of the **mutual exclusion protocol**.

Mutual Exclusion via Semaphores



Unfolding of the program graph $PG_1 \,|||\, PG_2$: some states, e.g.**<c_1, c_2, y=0>** are **not reachable**.

Communication: Handshaking

Another typical form of communication is via **exchanging messages**. Here, we see a synchronization mechanism where processes synchronize on some actions

H is a set of **synchronization actions**

- interleaving for $\alpha \notin H$:

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1'}{\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle s_1', s_2 \rangle} \qquad \frac{s_2 \xrightarrow{\alpha}_2 s_2'}{\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle s_1, s_2' \rangle}$$

- handshaking for $\alpha \in H$:

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1' \quad \wedge \quad s_2 \xrightarrow{\alpha}_2 s_2'}{\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle s_1', s_2' \rangle}$$

processes evolve simultaneously provided they are executing the same action.

Generalising to n processes

For **each pair** of processes P_i, P_j there exists a set $H_{i,j}$ of actions on which they **can synchronize**

- for $\alpha \in Act_i \setminus (\bigcup_{\substack{0<j\leqslant n \\ i\neq j}} H_{i,j})$ and $0 < i \leqslant n$:

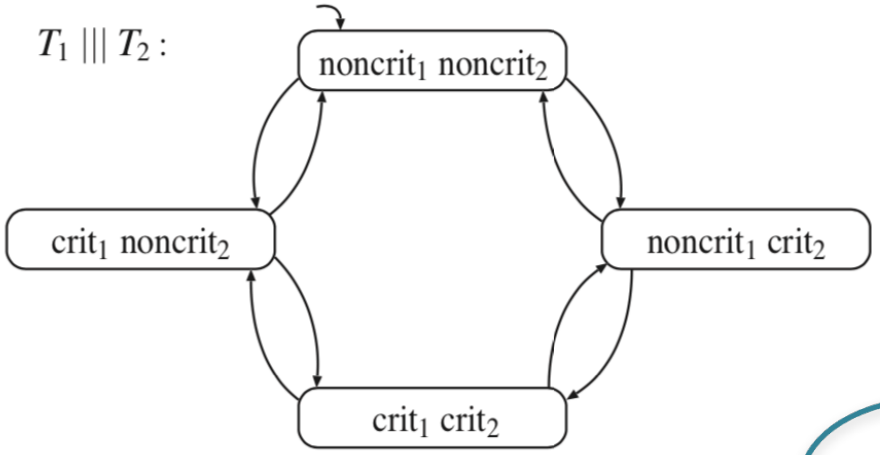
$$\frac{s_i \xrightarrow{\alpha}_i s_i'}{\langle s_1, \ldots, s_i, \ldots, s_n\rangle \xrightarrow{\alpha} \langle s_1, \ldots, s_i', \ldots s_n\rangle}$$

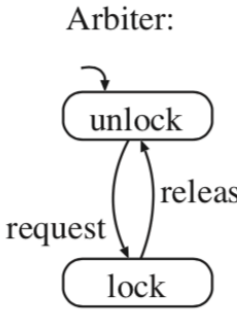
- for $\alpha \in H_{i,j}$ and $0 < i < j \leqslant n$:

$$\frac{s_i \xrightarrow{\alpha}_i s_i' \quad \wedge \quad s_j \xrightarrow{\alpha}_j s_j'}{\langle s_1, \ldots, s_i, \ldots, s_j, \ldots, s_n\rangle \xrightarrow{\alpha} \langle s_1, \ldots, s_i', \ldots, s_j', \ldots, s_n\rangle}$$

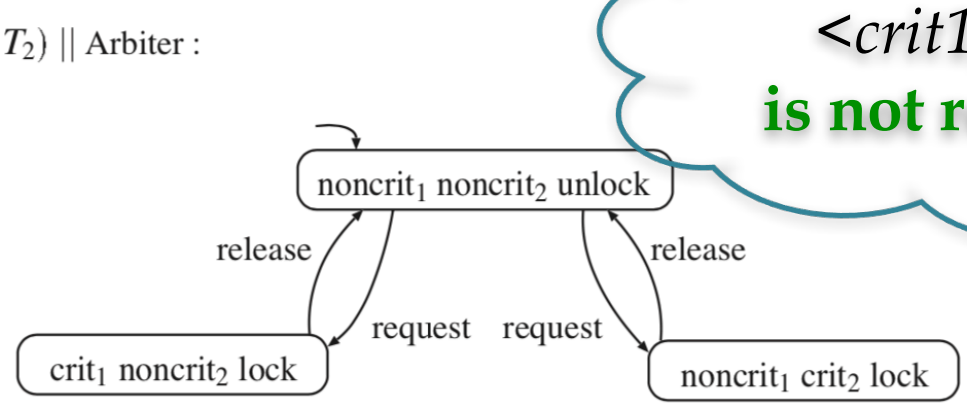
Mutual Exclusion: handshaking

Simplified version: process **just have two states**: ***noncrit, crit***. They synchronize with an **arbiter** on actions {*request, release*}:





It **works**!
<crit1, crit2>
is not reachable



That's all Folks!

Thanks for your attention…

…Questions?