Formal Methods in Software Development

Ivano Salvo
and Igor Melatti

Computer Science Department

Sapienza Università di Roma

Lesson 1, September 23rd, 2019

Lesson 0:

Course Presentation and Information

About this course…

Classroom:

Tuesday, 14:00-17:00 prof. Ivano Salvo
Wednesday, 14:00-16:00 prof. Igor Melatti

Main Topic: **Model Checking**

This part (**Tuesday**): mainly theoretical aspects

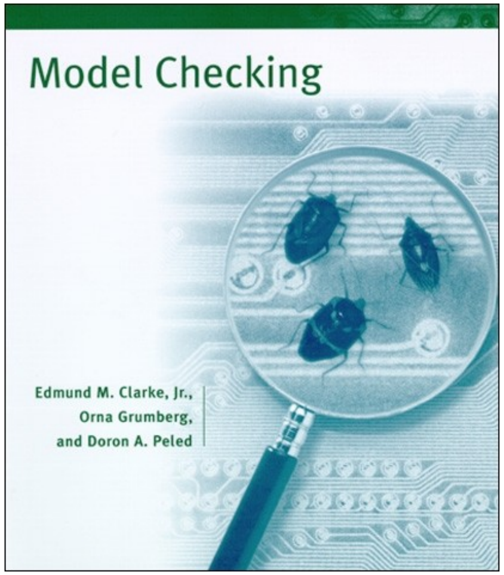
Prof. Melatti (**Wednesday**) will introduce the use of several model checkers (murphi, nuSMV, SPIN etc.)

Website (under construction):
http://twiki.di.uniroma1.it/twiki/view/MFS/FormalMethodsInSoftwareDevelopment20192020

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

Course material

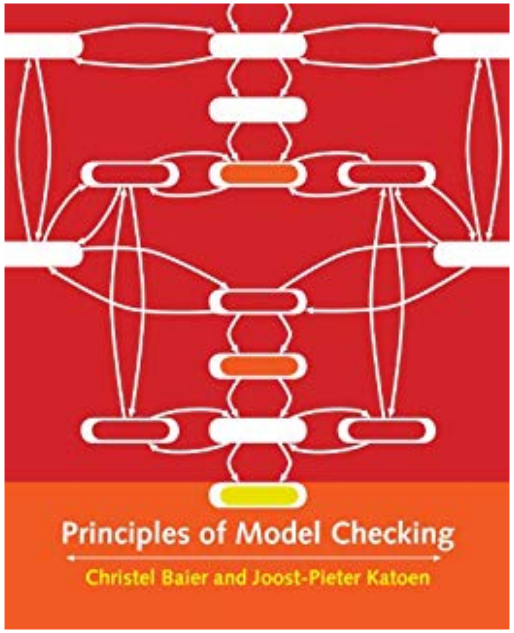
I will follow mainly the following book:



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

I will take some examples (expecially for the modeling part) from:

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


Principles of Model Checking
Christel Baier and Joost-Pieter Katoen

Final Examination

Written test: short questions and exercises

+

Project / presentation:

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

Lesson 0′:

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:

$$\forall x\colon Prec(x) \exists y. P(x,y)$$

Prove formally that the program computes a function f such that:

$$\forall x\colon Prec(x). P(x, f(x))$$

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:

Formally describe a model $\mathcal{M}$ of a system (**modeling**, usually via some abstract formalism, eg. **Transition Systems**).

Specification:

Formally describe **specification** φ (First order logic is ok for sequential programs, but some kind of **Temporal Logic** is needed for concurrent or hybrid systems).

Verification:

Formal **verification** that the system $\mathcal{M}$ satisfies φ, $\mathcal{M} \vDash \varphi$ 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

Model Checking: Strength

- Quite **general** approach that applies to many applications.
- It support **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 etc

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
- Does not allow **generalizations**. Example: systems with an arbitrary number of components.

Lesson 1:

Modeling 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)

Modeling Concurrent Systems

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

States record information about a system in **a certain moment.**

Transitions, or 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 in our verification task).

(Labeled) Transition Systems

Let AP be a set of atomic proposition. A **Labeled Transition System** M over AP is a 5-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.

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

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

Modeling: not just input/output

Observation 1: 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
- Interaction offered with other systems

…

Kripke structures

For verification purposes, we usually **drop action labels.**

Let AP be a set of atomic proposition. A **Kripke structure** M over AP is a 4-tuple (S, S_0, R, L), where:

- S is a finite set of ***states***;
- $S_0 \subseteq S$ is the set of **initial states**;
- $\mathrm{R} \subseteq S \times S$ is the **transition relation**;
- $L : \mathrm{S} \rightarrow 2^{AP}$ is the labeling function.

R must be **total**, i.e. for each state s there exists always s' such that $R(s, s')$

A **path** (or **execution**) from s in M is a sequence $\pi = s_0 s_1 s_2 \ldots$ such that $s_0 = s$ and $R(\mathrm{s}_i, \mathrm{s}_{i+1})$

Representing Kripke structures

Kripke structures can be represented by **first order formulas**.

Let $V=\{v_1, \ldots, v_n\}$ be a set of **variables**. Values of variables range in a **domain** D. A state s is just a **valuation** $s\colon V \to D$.

Each state is a tuple of values in D.

The set S_0 of initial states can be represented by a predicate $I(V)$ such that $I(s)$ holds if $s(v) \in S_0$.

Let $V' = \{v' \mid v \in V\}$. V' is called **next state** variables, whereas V is the set of **present state** variables. The transition relation R can be represented as a predicate $T(V,V')$, such that $R(s,s')$ if and only if $T(s(v), s'(v'))$.

Modeling: Example

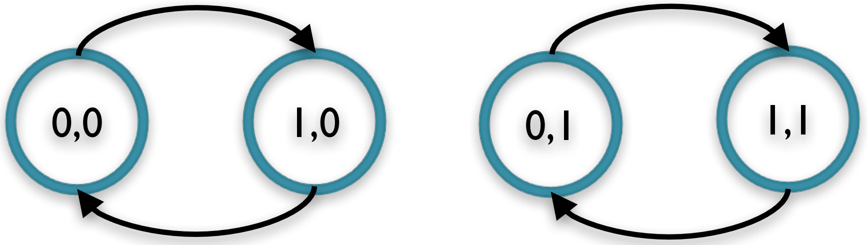
Consider the system S described by two boolean variables x and y. A state is a pair (d_1,d_2) in D={0,1}×{0,1}.

We consider the initial states (0,0) and (1,1) that can be represented by the predicate $x=y$.

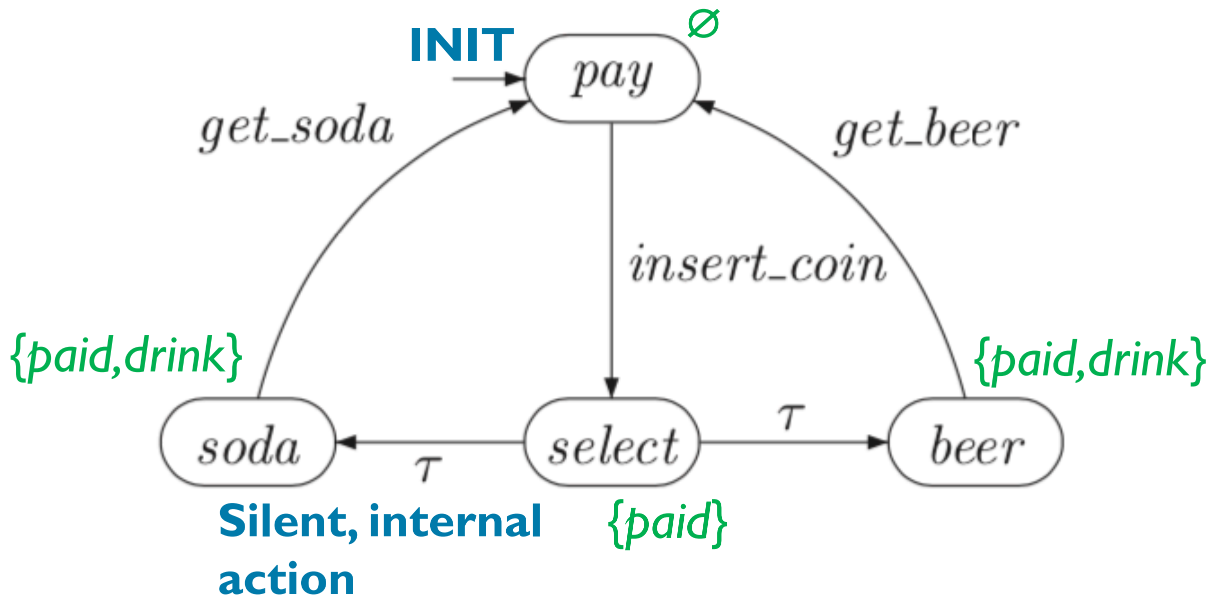
The transition is $x:=x+1$ mod 2, that can be represented by the predicate $T(x, y, x', y') \equiv x' = x+1 \bmod 2 \wedge y'=y$.

Execution paths are (0,0), (1,0), (0,0)… and (1,1), (0,1), (1,1)…

Observe that (1,1) is not reachable from (0,0)



Ex.: Beverage Vending Machine



In this model, the machine chooses **non-deterministically** to deliver a soda or a beer.

One can prove properties such as:

“The vending machine only delivers a drink after inserting a coin”

Non-determinism

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

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

The state s is a **terminal** state if $\mathrm{post}(s) = \varnothing$.

A system is deterministic if $|\mathrm{post}(s, a)| \leq 1$ 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
- …

LTS: executions

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

n is the **length** of π.

A path is **maximal** either if it is infinite or if its final state s_n is a terminal state. It is **initial** if s_0 is an initial state.

An **execution path** of a transition system is a maximal, initial execution fragment.

A state s is **reachable** if there exists an execution path that contains s.

Data Dependent Systems

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

Transitions can depend on some condition.
Conditional branching can be modeled by nondeterminism, but this can lead to very abstract (not useful) models.

In the following we see how a program generate a Labelled Transition System

Bev. Vending Machine Reloaded

Let us consider again the **Beverage Vending Machine** with a given number of beverages.
The machine returns the inserted coin when it is empty.

$$select \xrightarrow{nsoda\ =\ 0\ \wedge\ nbeer\ =\ 0:\ ret\_coin} start$$

conditional transition

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

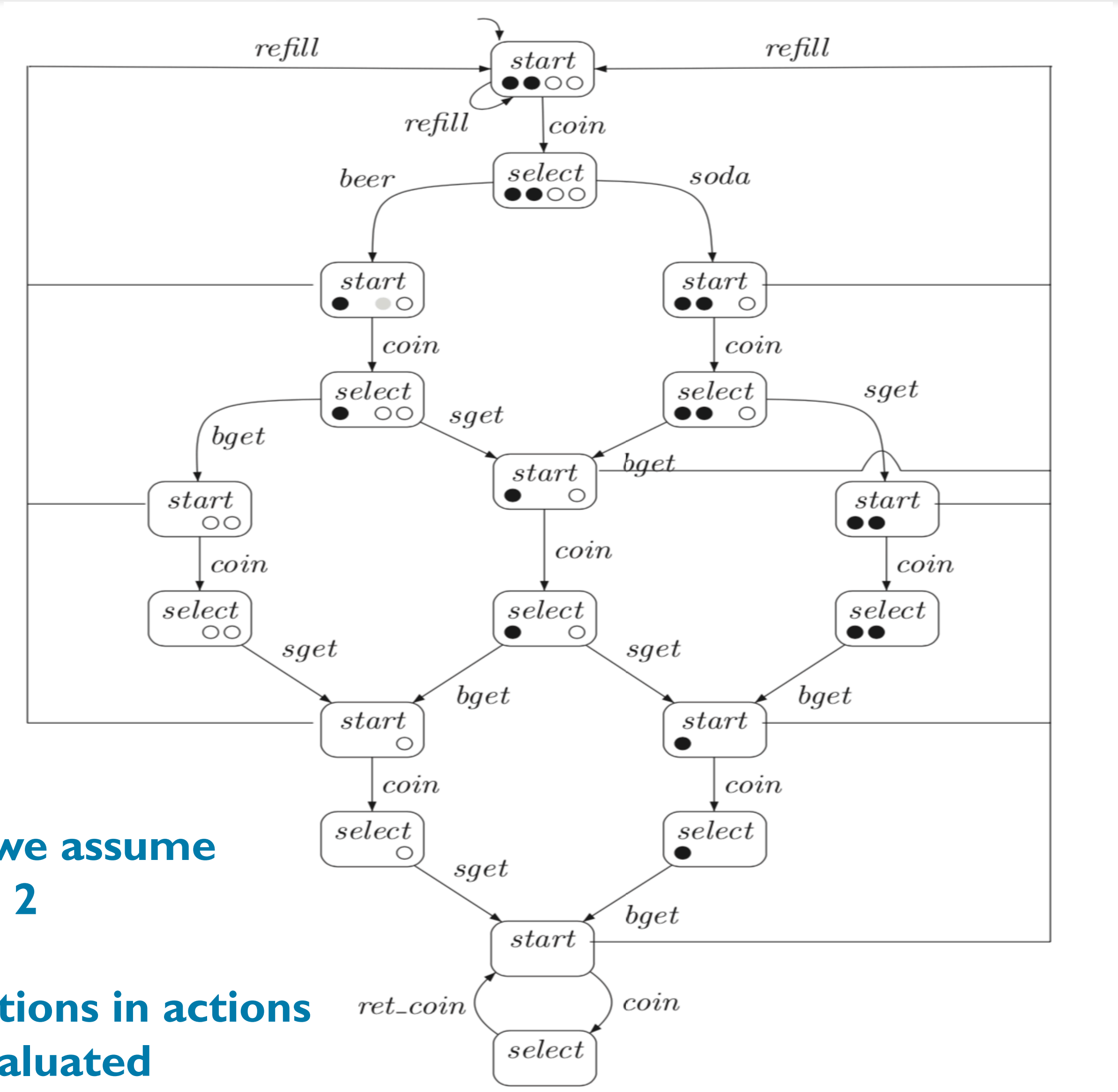
$$select \xrightarrow{nsoda\ >\ 0\ :\ sget} start \qquad \text{and} \qquad select \xrightarrow{nbeer\ >\ 0\ :\ bget} start$$

One can always refill or insert coin:

$$start \xrightarrow{true\ :\ coin} select \qquad \text{and} \qquad start \xrightarrow{true\ :\ refill} start$$

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

Bev. Vending Machine: unfolding



Here we assume max = 2

Conditions in actions are evaluated

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** and 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.

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:

$$Effect(coin, \eta) = \eta$$
$$Effect(ret\_coin, \eta) = \eta$$
$$Effect(bget, \eta) = \eta[nbeer := nbeer - 1]$$
$$Effect(sget, \eta) = \eta[nsoda := nsoda - 1]$$
$$Effect(refill, \eta) = [nsoda := max, nbeer := max]$$

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

Unfolding of a PG into a LTS

States are pairs of the form (l, η), where $l \in Loc$ and η 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.

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

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):

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

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

■

State Explosion Problem

As it is clear from this example, the number of states of the LTS is **huge** with respect to a program describing a concurrent system.

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.)

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.

$$M = M_1 \,//\, M_2 \,//\, \ldots \,//\, M_n$$

In the following, we briefly show how the operator $//$ can be defined and how different systems can communicate.

Interleaving Semantics

In **interleaving semantics**, the idea is that concurrent components evolve independently, as they run on a **single-processor** machine.

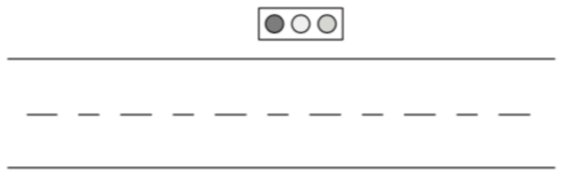
The composition contains all possible interleaving of actions (abstract from an **unknown scheduling** policy).

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

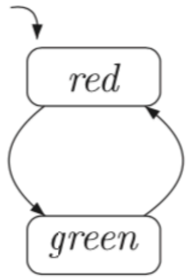
$$\mathit{Effect}(\alpha \,|||\, \beta, \eta) \;=\; \mathit{Effect}((\alpha \,;\, \beta) + (\beta \,;\, \alpha), \eta)$$

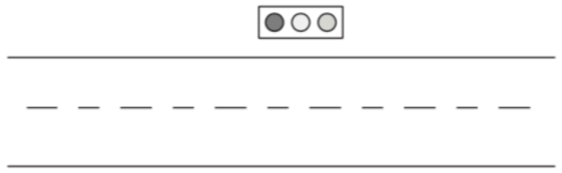
; is sequential composition and + is nondeterministic choice.

Ex.: Independent Traffic Lights

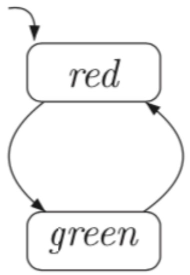


$TrLight_1$

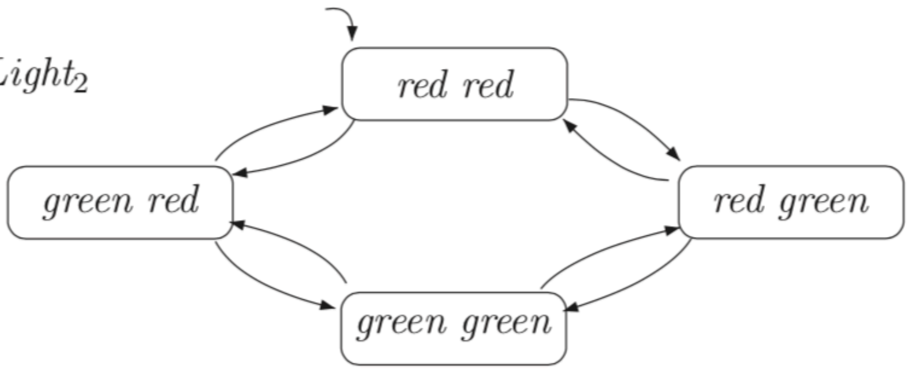




$TrLight_2$



$TrLight_1 \;|||\; TrLight_2$



Ex.: Independent variables

Two processes modify two independent variables:

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

All possible execution lead to the same results:

$$\begin{tikzcd}
x{=}0 \arrow[dd, "\alpha"] & & y{=}7 \arrow[dd, "\beta"] & & & x{=}0, y{=}7 \arrow[dl, bend right, "\alpha"'] \arrow[dr, bend left, "\beta"] & \\
 & ||| & & = & x{=}1, y{=}7 \arrow[dr, bend right, "\beta"'] & & x{=}0, y{=}5 \arrow[dl, bend left, "\alpha"] \\
x{=}1 & & y{=}5 & & & x{=}1, y{=}5 &
\end{tikzcd}$$

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 system is a source of state explosion.

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

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

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

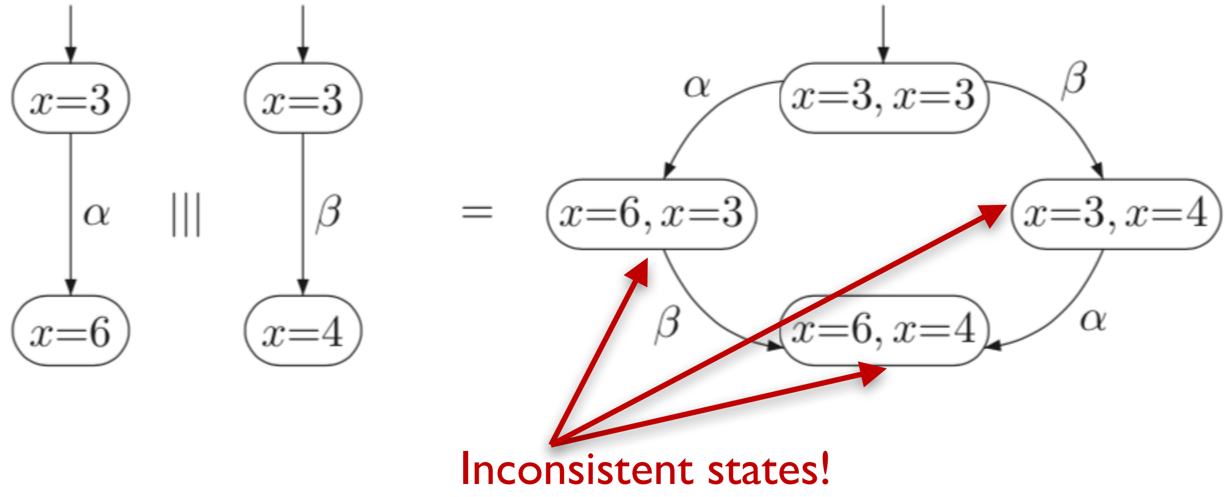
Therefore, the number of states **is exponential** in the **number of components**!

Ex.: shared variables

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

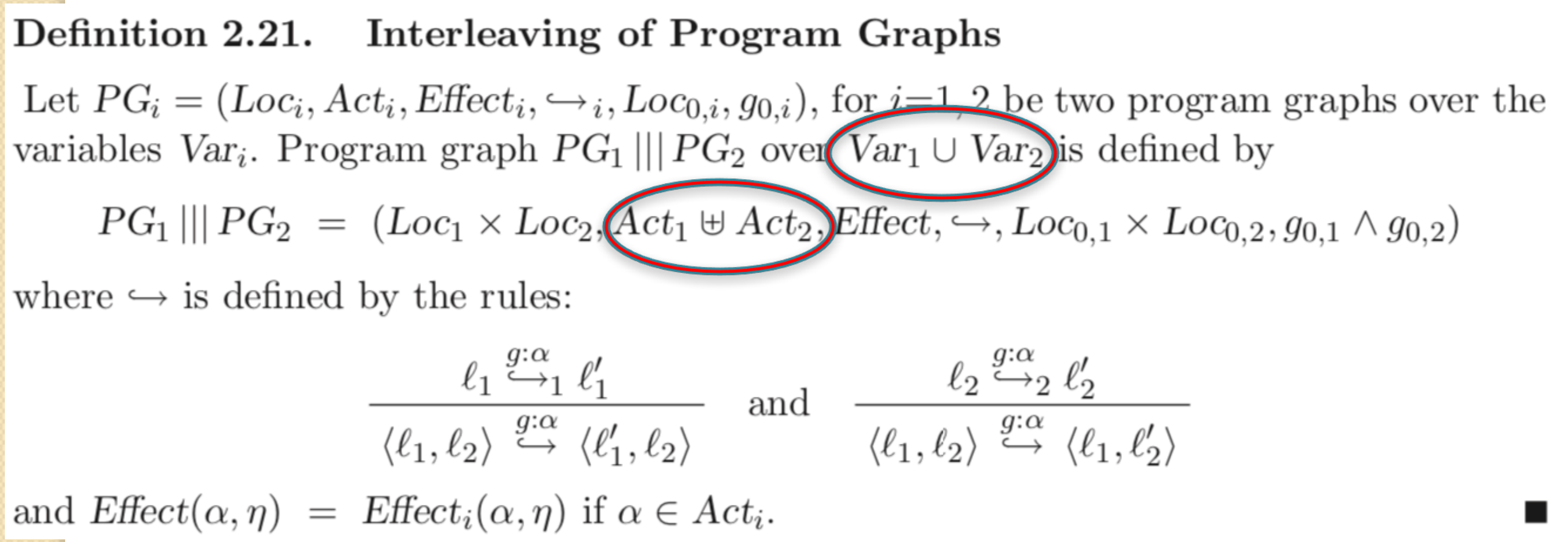
$$\underbrace{x := 2\cdot x}_{\text{action } \alpha} \;|||\; \underbrace{x := x + 1}_{\text{action } \beta}$$

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



Interleaving (shared vars): 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

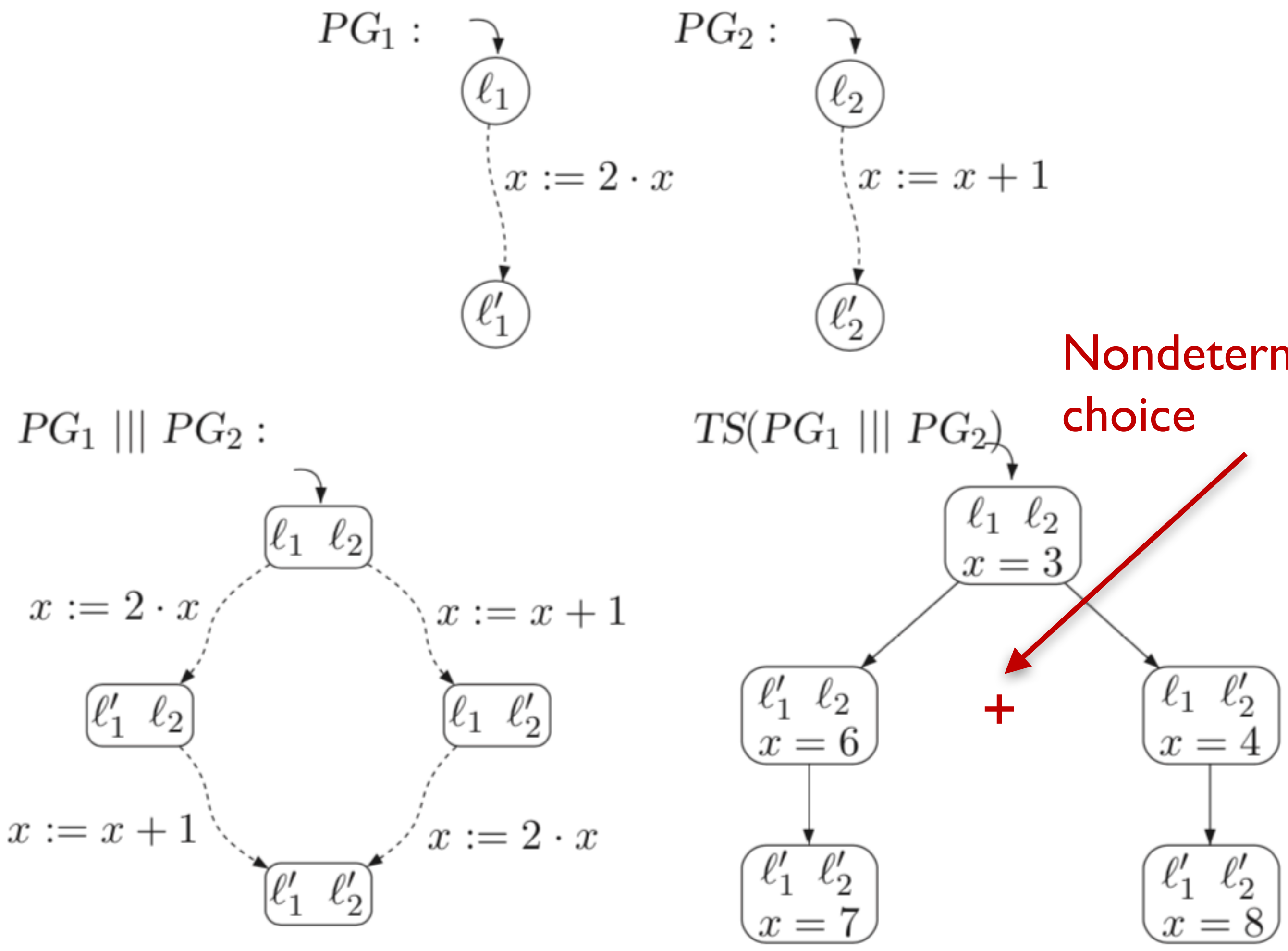
$$PG_1 \,|||\, PG_2 \;=\; (Loc_1 \times Loc_2, Act_1 \uplus Act_2, Effect, \hookrightarrow, Loc_{0,1} \times Loc_{0,2}, g_{0,1} \wedge g_{0,2})$$

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 $\alpha \in Act_i$. ■

Ex.: shared variables reloaded



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 is ignored by the transition system.

Granularity:

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

Granularity: Example

Consider the model M_1 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.

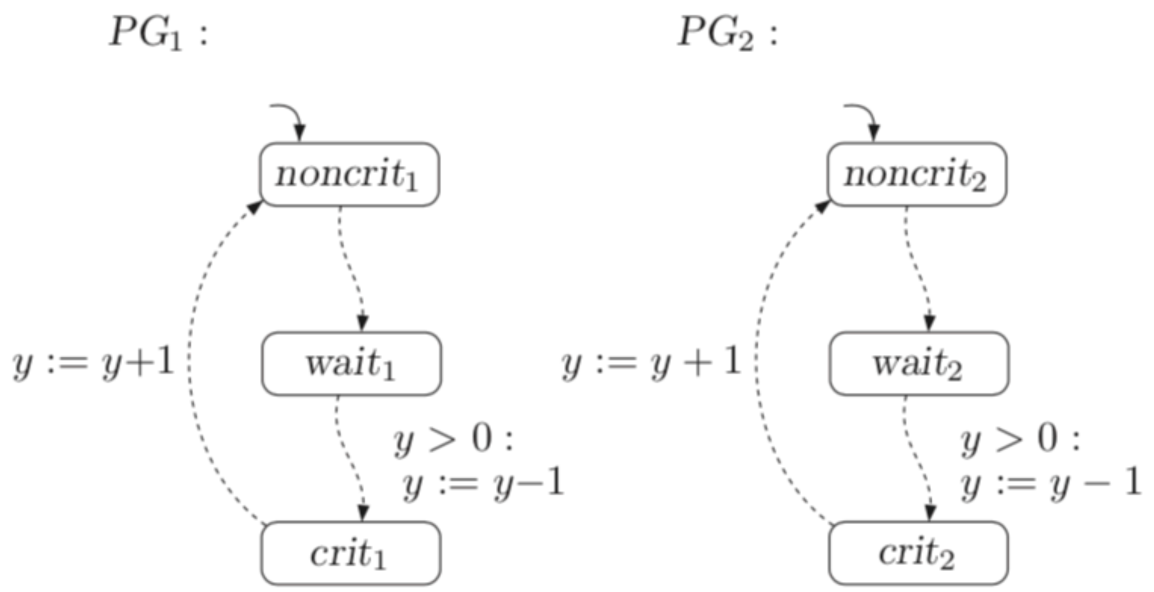
Consider the model M_2 an **assembly like implementation** of the ``same'' system (R_i are registers):

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

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

In M_2, we can have different execution orders, for example $\alpha_0\ \beta_0\ \alpha_1\ \beta_1\ \alpha_2\ \beta_2$ that leads to the state $x=3 \wedge y=3$.

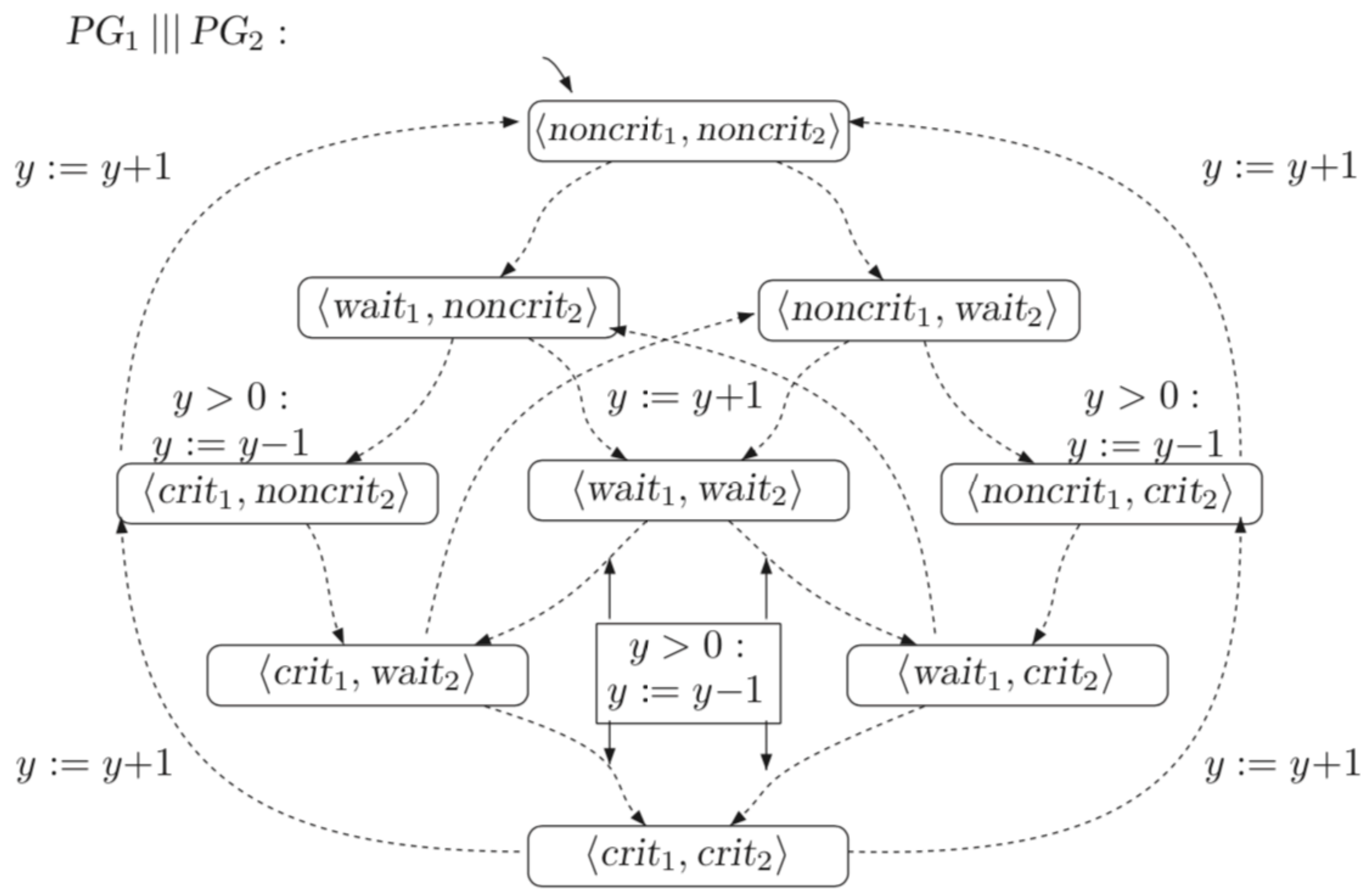
Mutual Exclusion via semaphores



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

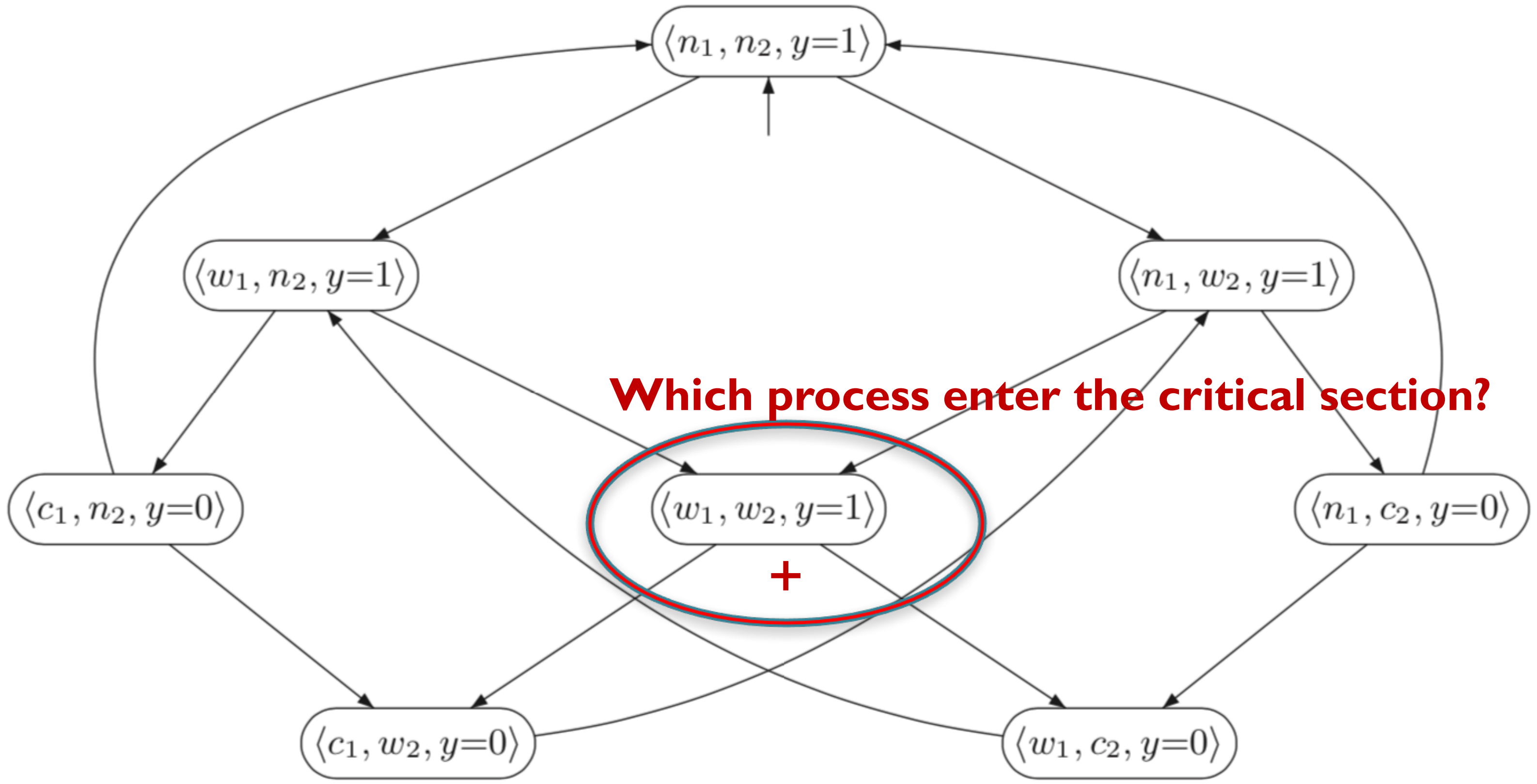
Observation: *y* := *y*-1 cannot executed in parallel (**critical actions** involving shared variables)

Mutual Exclusion via semaphores


$PG_1 \mid\mid\mid PG_2$:
$\langle noncrit_1, noncrit_2 \rangle$
$y := y{+}1$
$y := y{+}1$
$\langle wait_1, noncrit_2 \rangle$
$\langle noncrit_1, wait_2 \rangle$
$y > 0 :$
$y := y{-}1$
$y := y{+}1$
$y > 0 :$
$y := y{-}1$
$\langle crit_1, noncrit_2 \rangle$
$\langle wait_1, wait_2 \rangle$
$\langle noncrit_1, crit_2 \rangle$
$\langle crit_1, wait_2 \rangle$
$y > 0 :$
$y := y{-}1$
$\langle wait_1, crit_2 \rangle$
$y := y{+}1$
$y := y{+}1$
$\langle crit_1, crit_2 \rangle$

Interleaving of program graphs of the mutual exclusion protocol.

Mutual Exclusion via semaphores


$\langle n_1, n_2, y{=}1 \rangle$
$\langle w_1, n_2, y{=}1 \rangle$
$\langle n_1, w_2, y{=}1 \rangle$
Which process enter the critical section?
$\langle c_1, n_2, y{=}0 \rangle$
$\langle w_1, w_2, y{=}1 \rangle$
$\langle n_1, c_2, y{=}0 \rangle$
+
$\langle c_1, w_2, y{=}0 \rangle$
$\langle w_1, c_2, y{=}0 \rangle$

Unfolding of the program graph $PG_1 \parallel\!| PG_2$: $<c_1, c_2, y=0>$ is **not reachable**.

Communication: Handshaking

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

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

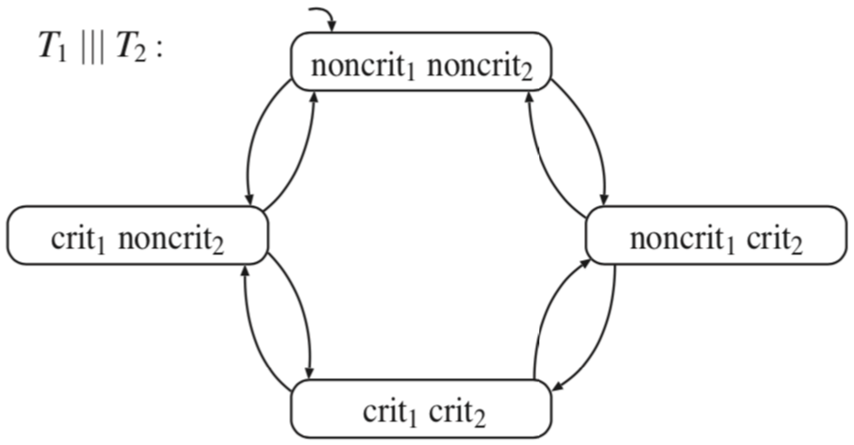
$$\frac{s_i \xrightarrow{\alpha}_i s_i'}{\langle s_1, \ldots, s_i, \ldots, s_n \rangle \xrightarrow{\alpha} \langle s_1, \ldots, s_i', \ldots s_n \rangle}$$

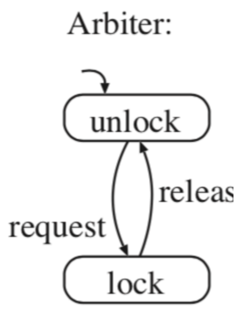
- for $\alpha \in H_{i,j}$ and $0 < i < j \leqslant n$:

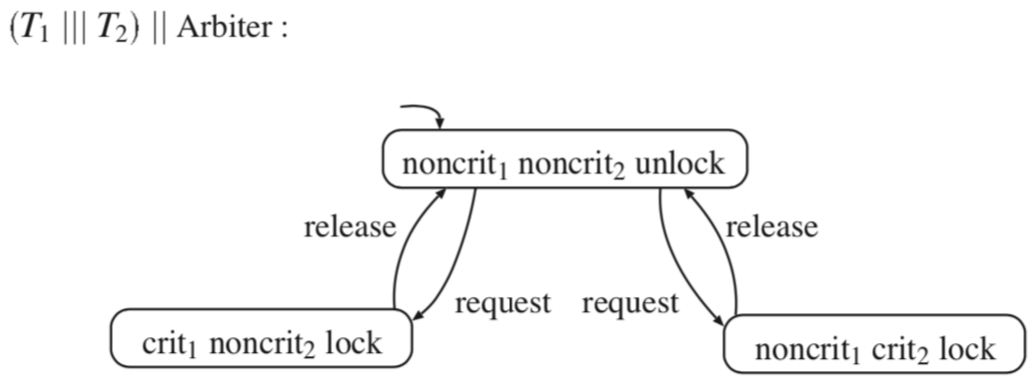
$$\frac{s_i \xrightarrow{\alpha}_i s_i' \quad \wedge \quad s_j \xrightarrow{\alpha}_j s_j'}{\langle s_1, \ldots, s_i, \ldots, s_j, \ldots, s_n \rangle \xrightarrow{\alpha} \langle s_1, \ldots, s_i', \ldots, s_j', \ldots, s_n \rangle}$$

Mutual Exclusion: handshaking

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






$(T_1 ||| T_2) ||$ Arbiter :
noncrit$_1$ noncrit$_2$ unlock
release
release
request
request
crit$_1$ noncrit$_2$ lock
noncrit$_1$ crit$_2$ lock