Transition Systems

Design of critical systems

- *Precision* and *rigor* are crucial in the development of a critical application
  - where "critical" means that failure of the application cannot be tolerated, because it might lead to loss of lives/money/resources...
- One way to achieve precision and rigor is through *formal techniques*
  - these are based on mathematical models of the system being designed
    - typically discrete models, such as automata, logic, etc.
  - formal models can be (at least in principle) *verified*, and the verification can often be carried out in an automated way
    - the user defines what needs to be verified, pushes a button, then an algorithm tells us whether the verification is successful or not

Overview of formal verification

- Suppose we are developing a critical system...
- We call **specification** (S) the high-level *model* of the system that we are developing
  - high-level = abstracting away from as many implementation details as possible
- We call **requirement** (R) the property that we want to hold for the system
  - it is the property that we would like to verify (i.e. we should be able to show that it holds for the developed application)

Requirements

- Requirements are often separated in two main categories: *functional* and *nonfunctional*
- Functional requirements deal with input/output relations
  - "if the input is x, then the output computed by the function should be y"
  - typically captured by pre-/post-conditions of functions/methods
- Nonfunctional requirements can be of various kinds:
  - ordering: "event A must always be followed by event B"
  - metric (real-time): "event A must always be followed, *within 3 seconds*, by event B"
  - probabilistic: "state S must be reached with probability 0.90"
  - real-time probabilistic: "there is a 0.8 probability of reaching state S within 3 seconds"

From specification to verification

- Once we have formalized R and S, we would like to be able to (formally) verify that the requirement R holds if the system behaves as modeled by S
- We write $S \models R$ to mean "property R holds for specification S"
  - then the ultimate goal is to determine whether $S \models R$ or not
- How do we actually formally verify whether $S \models R$ or not?
  - what techniques can we employ?
- It depends on the formalism(s) we used to describe R, S, etc.

Model Checking

- “Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.”

Model checking

- Formalisms:
  - for system models: finite state automata (or a variation thereof)
  - for properties to be verified: temporal logic
- Basic idea: exploration of a finite state space
- Advantages
  - highly automated
    - write the model, then "push button" to verify it
  - if verification fails, a counterexample is produced
    - the counterexample is a witness of the problem

Model checking (2)

- Drawbacks
  - computationally onerous
    - state space explosion
    - various techniques to circumvent (or at least alleviate) this problem
  - limited expressiveness
- References:
  - **C. Baier, J-P. Katoen, *Principles of Model Checking*, MIT Press, 2008**
  - C.A. Furia, D. Mandrioli, A. Morzenti, M. Rossi, Modeling Time in Computing, Springer, 2012

An operational model for concurrency

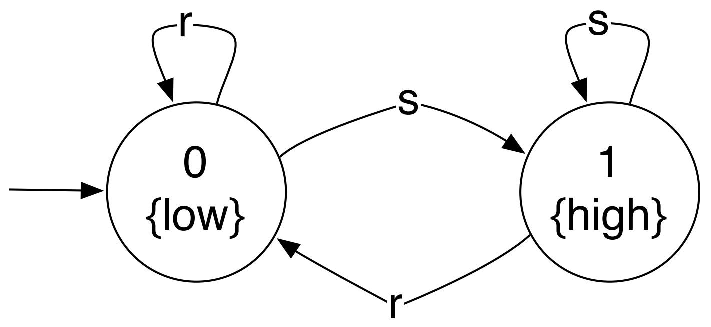
TRANSITION SYSTEMS

Transition systems

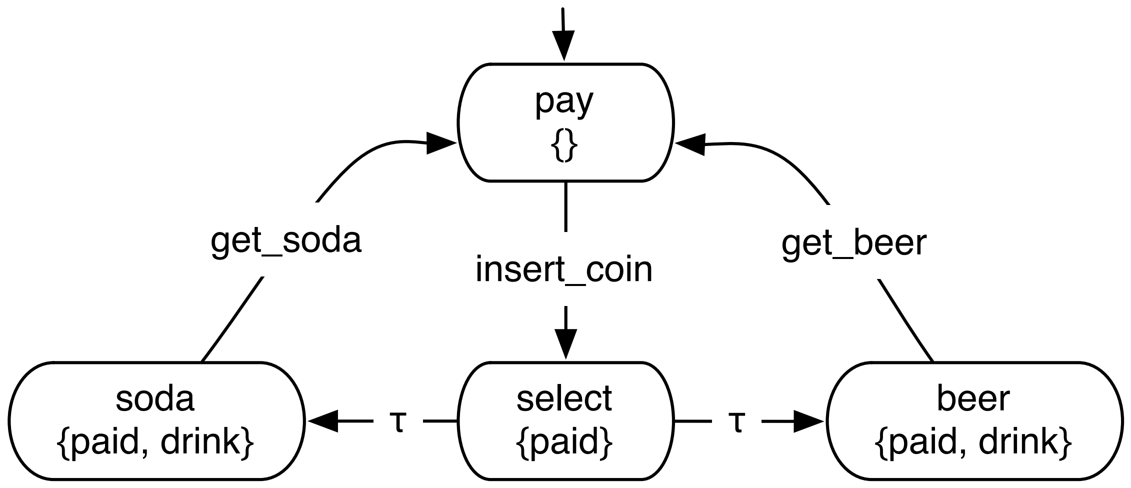
- A **transition system** is a tuple $\langle S, Act, \rightarrow, I, AP, L\rangle$:
  - *S* is a set of states
  - *Act* is a set of input symbols (aka *Actions*)
  - $\rightarrow \subseteq S \times Act \times S$ is a transition relation
  - $I \subseteq S$ is a nonempty set of initial states
  - *AP* is a set of atomic propositions, used as state labels
  - $L: S \rightarrow 2^{AP}$ is a labeling function
- Set of states, Act and AP can be finite or infinite
  - States can be denoted as s1, s2, etc.
  - A special action, τ, denotes an internal event (“silent action”)

Examples of transition system

(a)



(b)



Deterministic vs. nondeterministic

- A transition system is **deterministic** if:
  for all states *s* and input *i*, there is only one state *s*' such that $\langle s, i, s' \rangle \in \rightarrow$
- Otherwise, it is nondeterministic
  - Of the previous examples, (a) is deterministic, (b) is nondeterministic (because of the silent action τ)
- This is called action-determinism
  - There is also state-label determinism: given a state *s,* there exist only one state *s*' and one input *i* such that $\langle s, i, s' \rangle \in \rightarrow$

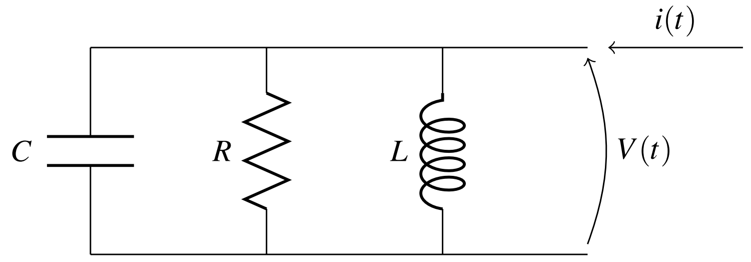
Examples of transition systems (2)

Label	Instruction	Comment
	READ 1	Store n into $M[1]$.
	LOAD= 1	If $n = 1$ then it is trivially prime
	SUB 1	and execution ends immediately
	JZ *yes*	with a positive answer.
	LOAD= 2	Initialize $M[2]$ to 2
	STORE 2	to start a loop of tests.
loop:	LOAD 1	When $M[2] = n$ exit the loop:
	SUB 2	all possible divisors have been tested
	JZ *yes*	hence the number is prime.
	LOAD 1	Assuming integer arithmetic, if
	DIV 2	$M[1]$ equals $(M[1]\ /\ M[2])\times M[2]$ then
	MULT 2	$M[2]$ contains a divisor of n,
	SUB 1	hence n is not prime:
	JZ *no*	exit loop and report negative answer.
	LOAD 2	Increment $M[2]$ by 1
	ADD=1	to test the next divisor.
	STORE 2	
	JUMP *loop*	Repeat loop.
yes:	WRITE= 1	Output positive answer.
	HALT	
no:	WRITE= 0	Output negative answer.
	HALT	

- State: contents the registers (PC, ACC), of the memory cells used and of the output
- Input symbols: naturals N (possible values read at line 0), plus ε (no input required)
- initial state: ⟨PC = 0, ACC = 0, M[0] = 0, M[1] = 0, U⟩
- Instruction 19 labelled "is_prime", Instruction 21 labelled "not_prime"
- $\langle s_1, i, s_2\rangle \in\rightarrow$ iff the execution of PC from configuration s_1 when input is i produces configuration s_2

- States and inputs are countable

Examples of transition systems (3)



$$\frac{\mathrm{d}}{\mathrm{d}t} i_L(t) = \frac{1}{L} V(t)$$
$$\frac{\mathrm{d}}{\mathrm{d}t} V(t) = \frac{1}{c}\left(i(t) - i_L(t) - \frac{V(t)}{R}\right)$$

- State space: $\mathcal{R} \times \mathcal{R}$
  - values of i_L, V
- Initial state: ⟨1,0⟩
- Input symbols: $\mathcal{R}_{>0}$
  - time distances between states
- ⟨1,0⟩ is labeled "i_high", ⟨0,1⟩ is labeled "i_low"
- $\langle\langle i_{L1},V_1\rangle, d, \langle i_{L2},V_2\rangle\rangle \in \rightarrow$ iff state $\langle i_{L1},V_1\rangle$ becomes $\langle i_{L2},V_2\rangle$ after d
- States and inputs are uncountable

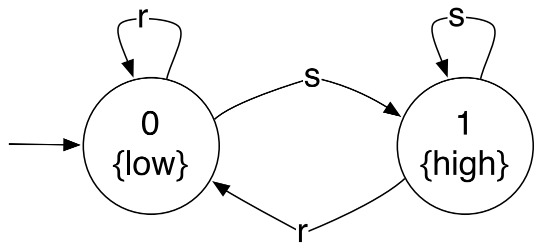
Runs of transition systems

- Given a (possibly infinite) sequence $\sigma = i_1 i_2 i_3, \ldots$ of input symbols in Act, a ***run*** r_σ for TS $=\langle S, Act, \rightarrow, I, AP, L\rangle$ is a sequence $s_0\ i_1\ s_1\ i_2\ s_2 \ldots$ where $s_0 \in I$, each $s_j \in S$ and for all $k \geq 0$, it is $\langle s_k, i_{k+1}, s_{k+1}\rangle \in \rightarrow$
  - If TS is nondeterministic, there can be many runs with the same input sequence
- A state s' is ***reachable*** if there exists a sequence $\sigma = i_1 i_2 \ldots i_k$ and a finite run $r_\sigma = s_0\ i_1\ s_1\ i_2\ s_2 \ldots i_k\ s'$
- Given a run r_σ its *trace* is the sequence of AP subsets $L(s_0)\ L(s_1)\ L(s_2), \ldots$, i.e., the sequence of (sets of) state labels.
  - sometimes trace is the name of the sequence σ of input symbols that has a run r_σ (called *input trace)*

Examples of runs

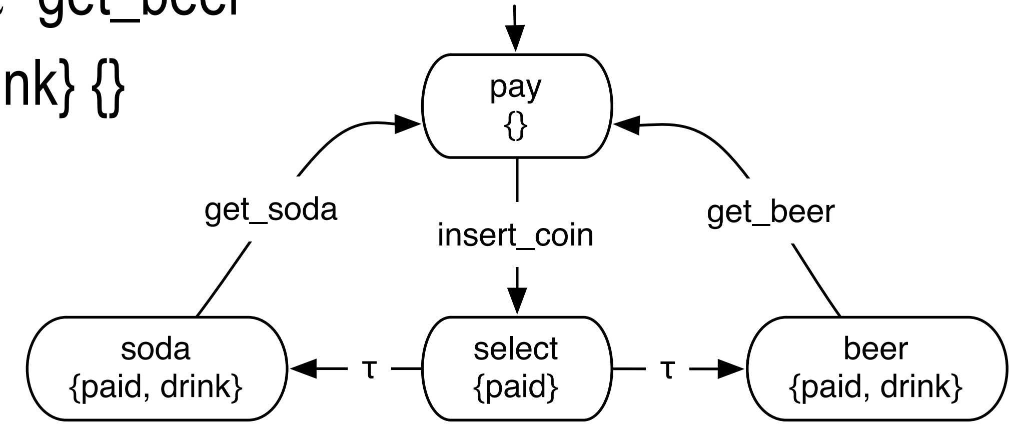
(a) 0 s 1 r 0 s 1 r 0 r 0 s 1

- trace of the inputs: s r s r r s
- trace of the labels: {low} {high} {low} {high} {low} {low} {high}



(b) pay insert_coin select τ soda get_soda pay insert_coin select τ beer get_beer pay

- input trace: insert_coin τ get_soda insert_coin τ get_beer
- label trace: {} {paid} {paid,drink} {} {paid} {paid,drink} {}



Infinite runs

- A run may be finite if it enters a terminal state (a state without any outgoing transition)
- To model reactive systems, often we consider **infinite runs**
  - The system is in principle non-terminating
  - Ex: KRC is non-terminating (trains may come forever)
  - Operating systems are a typical example, but also web servers, many games, etc.
  - In practice, every system has a finite lifetime, but we can abstract away this

What about automata or Petri Nets?

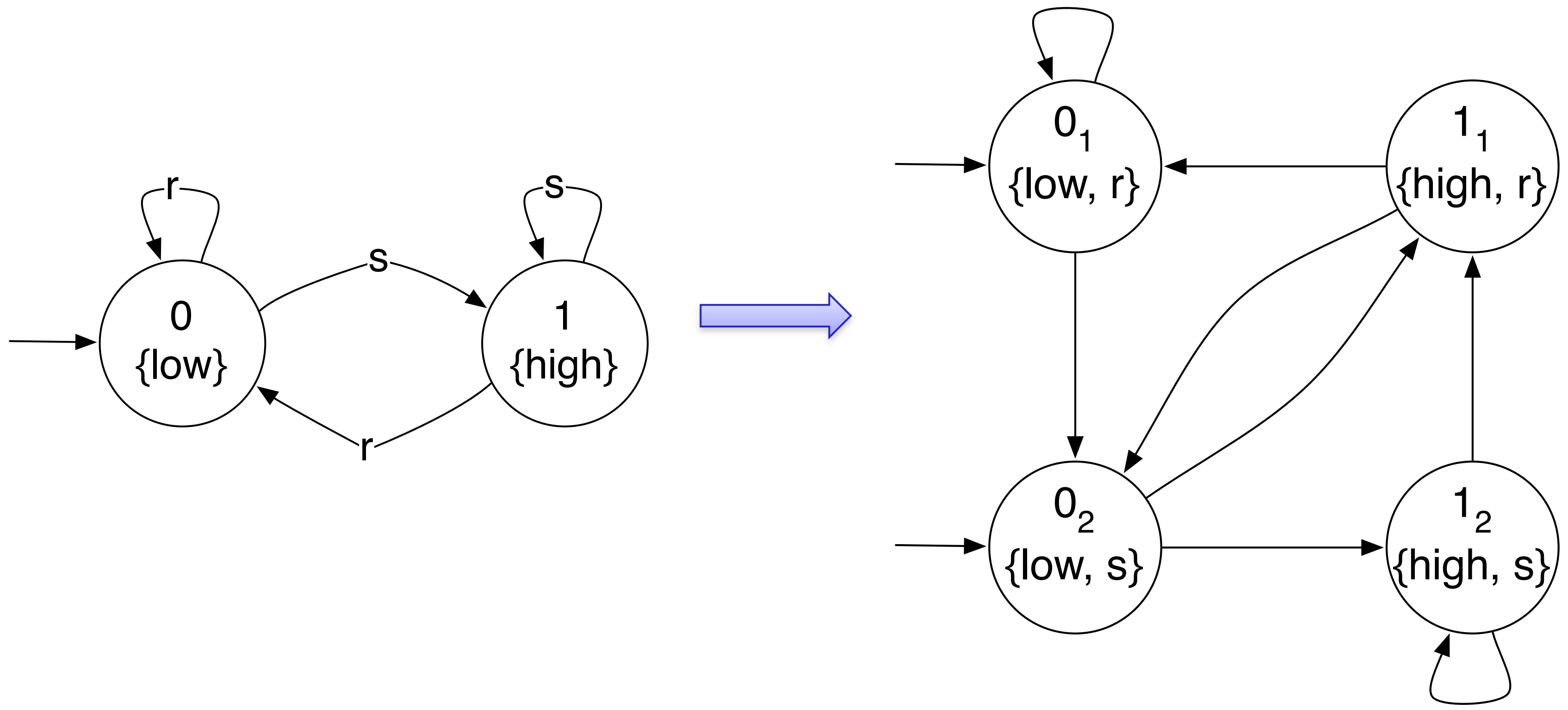
- A finite TS is just a finite automaton in disguise.
  - Finite TS are often called FSM as well.
  - Number of actual states is up to $|Q|\ |2^{AP}|$
  - But a TS may also be infinite…

- A TS is more suited to studying concurrency issues than FA.
  - Easy to define interacting, concurrent processes

- A Petri Net can be modeled by a TS
- Petri Nets are graphical appealing, but also TS have good graphical representation
- PNs are difficult to generalize and their decision problems have always exponential complexity (or more)
  - Reachability: NONELEMENTARY
  - Boundedness: $2^{O(n \lg n)}$

VARIANTS OF TS

Moving inputs to states



This transformation (external inputs become state labels) is very common.
Only «internal communications» are left as input.
It simplifies definitions and system analysis and we will also often use it.

Introducing variables

- Compare

$$\begin{tikzcd}
{} \arrow[r] & q_0 \arrow[loop above, "\tau"] \arrow[r, "s"] & q_s \arrow[r, "\tau"] & q_{w1} \arrow[r, "\tau"] & q_{w2} \arrow[r, "\tau"] & q_{w3} \arrow[r, "e"] & q_e \arrow[loop above, "\tau"] \arrow[lllll, bend right, "s"']
\end{tikzcd}$$

with

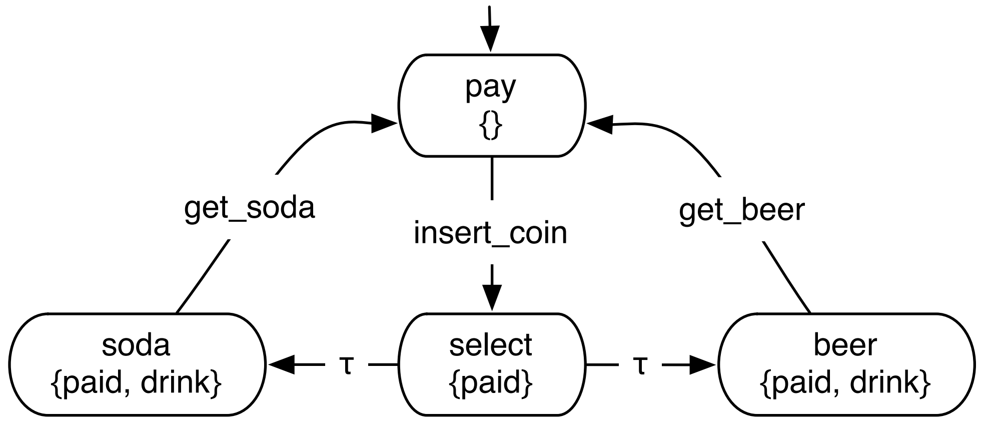
$$\begin{tikzcd}
{} \arrow[r] & q_0 \arrow[loop below, "\tau"] \arrow[r, "s / c := 0"] & q_s \arrow[loop below, "{[c < 3]\ \tau / c := c + 1}"] \arrow[r, "{[c = 3]\ e}"] & q_e \arrow[loop below, "\tau"] \arrow[l, bend right, "s / c := 0"']
\end{tikzcd}$$

- Also, what if the delay is K (with K a constant)?

Program graphs

- When using variables, TS are called ***Program Graphs***
- A PG has a set *Var* of variables, which are given a value in every state by an *Eval* function.
- Transitions may have conditions (“guards”) on the value of the variables
- An “Effect” function describes how the inputs affect the value of the variables (e.g., x:=0).
- “States” of PGs are usually called “Locations”.

Another example of program graph



- Add two Integer variables: numSoda and numBeer.
- Add guard numSoda>0 on transition soda ->pay, and guard numBeer>0 on transition beer->pay
- Add transition soda->select with guard numSoda=0, and similary beer->select
- get_soda effect: numSoda--; get_beer effect: numBeer--
- Add a refill state (numSoda=max, numBeer=max)

Program graphs as TS

- Program graphs may always be transformed into a (possibly infinite) TS
- TS have no guards and variables, but:
  - Each guard may be represented as a symbol of the AP set of atomic propositions.
  - The AP set must also include all locations of the program graph.
  - AP set becomes very large (although often only a small part is relevant for studying properties).

CONCURRENCY: PARALLEL COMPOSITION

INTERLEAVING OF TS AND PG

Interleaving ||| of independent TS

- $TS_1 = \langle S_1, Act_1, \rightarrow_1, I_1, AP_1, L_1 \rangle$, $TS_2 = \langle S_2, Act_2, \rightarrow_2, I_2, AP_2, L_2 \rangle$

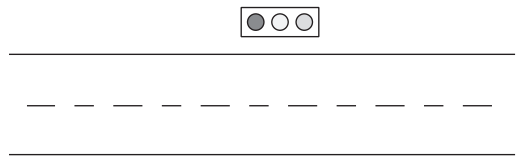
- Their interleaving $TS_1 \,|||\, TS_2$ is defined as:
  $\langle S_1 \times S_2, Act_1 \cup Act_2, \rightarrow, I_1 \times I_2, AP_1 \cup AP_2, L \rangle$

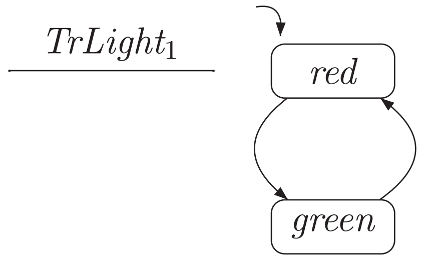
where L(<s1,s2>)= L(s1) U L(s2) and the transition relation $\rightarrow$ is:

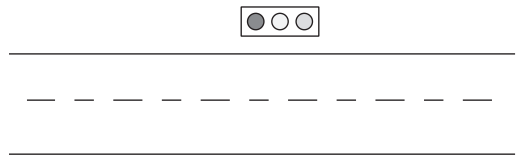
$$\frac{s_1 \xrightarrow{\alpha}_1 s_1'}{\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle s_1', s_2 \rangle} \quad \text{and} \quad \frac{s_2 \xrightarrow{\alpha}_2 s_2'}{\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle s_1, s_2' \rangle}$$

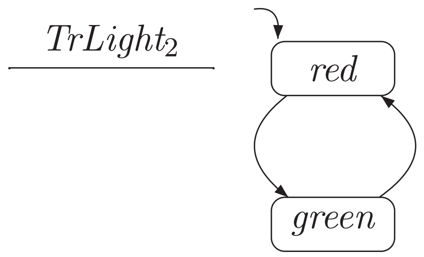
- In practice, the two TS proceed independently (alternating nondeterministically), but only one at a time is considered to be “active”. Similar to non-synchronizing threads: all possible interleavings are allowed.

Example: two independent traffic lights

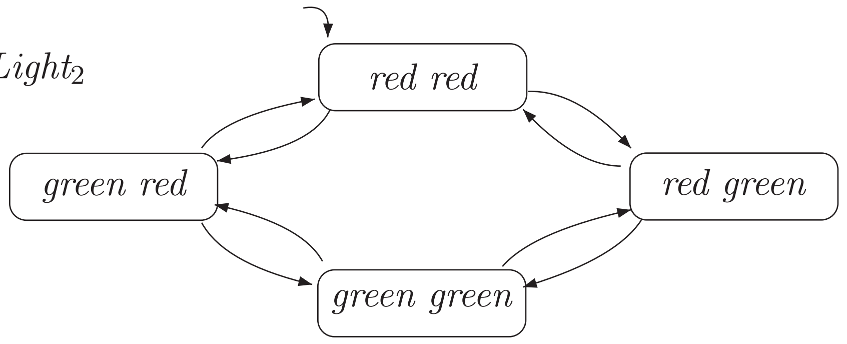








$TrLight_1 \; ||| \; TrLight_2$



Interleaving in program graphs with no shared variables

- Given two program graphs PG1, PG2, it seems immediate to define their interleaving as TS(PG1)||| TS(PG2)

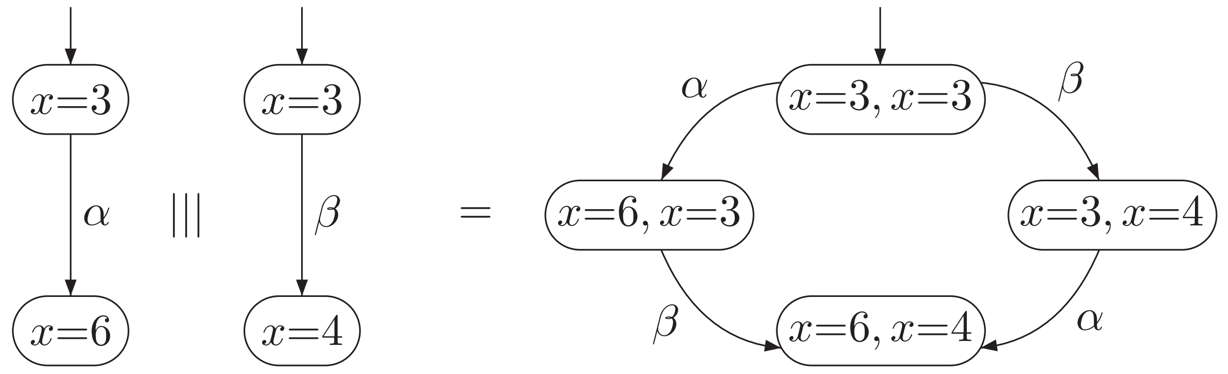
$\underbrace{x := x + 1}_{=\alpha} \;|||\; \underbrace{y := y - 2}_{=\beta}.$ initially $x = 0$ and $y = 7$,

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

Interleaving with shared variables

- Given two program graphs PG1, PG2, TS(PG1)||| TS(PG2) may not work if there are “shared variables”, since in this case some of the locations are *not* valid: the PGs access shared "critical" variables

$$\underbrace{x := 2 \cdot x}_{\text{action } \alpha} \;|||\; \underbrace{x := x + 1}_{\text{action } \beta} \quad \text{where we assume that initially } x = 3.$$



Interleaving example: critical actions cannot act in parallel

PG_1 :

$$\begin{tikzcd} {} \arrow[d] \\ \ell_1 \arrow[d, dashed, "x := 2 \cdot x"] \\ \ell_1' \end{tikzcd}$$

PG_2 :

$$\begin{tikzcd} {} \arrow[d] \\ \ell_2 \arrow[d, dashed, "x := x + 1"] \\ \ell_2' \end{tikzcd}$$

$PG_1 \mid\mid\mid PG_2$:

$$\begin{tikzcd} & {} \arrow[d] & \\ & \ell_1\ \ell_2 \arrow[dl, dashed, "x := 2 \cdot x"'] \arrow[dr, dashed, "x := x + 1"] & \\ \ell_1'\ \ell_2 \arrow[dr, dashed, "x := x + 1"'] & & \ell_1\ \ell_2' \arrow[dl, dashed, "x := 2 \cdot x"] \\ & \ell_1'\ \ell_2' & \end{tikzcd}$$

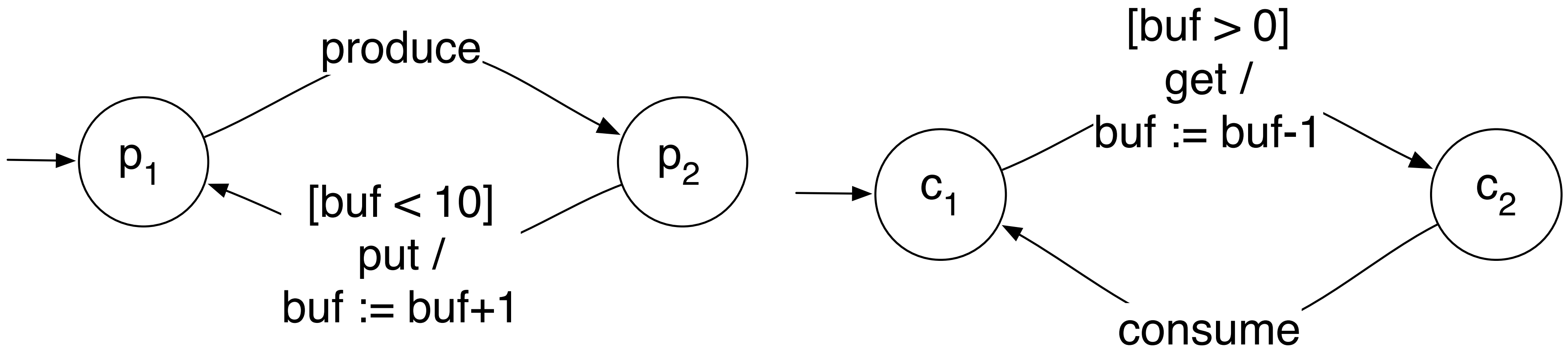
$TS(PG_1 \mid\mid\mid PG_2)$ where we assume that initially $x = 3$.

$$\begin{tikzcd} & {} \arrow[d] & \\ & \begin{matrix} \ell_1\ \ell_2 \\ x = 3 \end{matrix} \arrow[dl] \arrow[dr] & \\ \begin{matrix} \ell_1'\ \ell_2 \\ x = 6 \end{matrix} \arrow[d] & & \begin{matrix} \ell_1\ \ell_2' \\ x = 4 \end{matrix} \arrow[d] \\ \begin{matrix} \ell_1'\ \ell_2' \\ x = 7 \end{matrix} & & \begin{matrix} \ell_1'\ \ell_2' \\ x = 8 \end{matrix} \end{tikzcd}$$

assumption: **atomicity of the assignment statements**

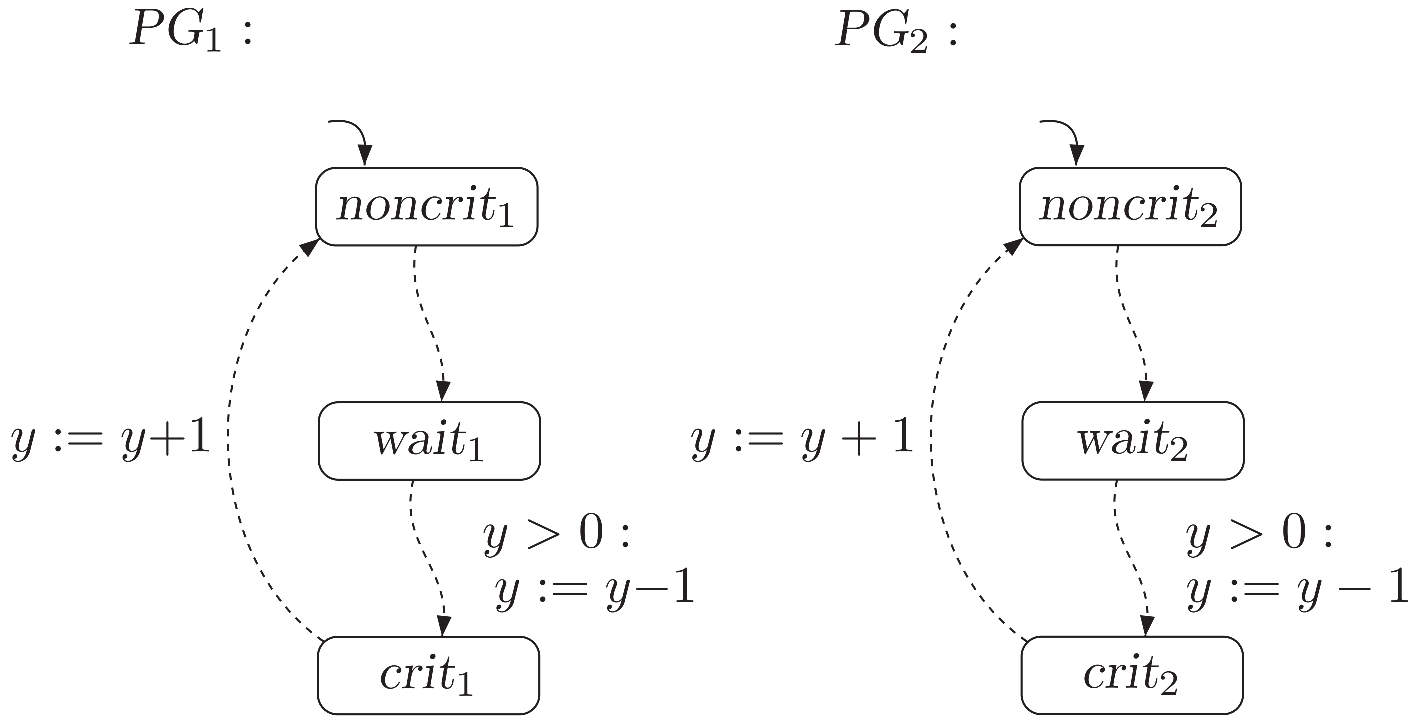
Synchronization via constraints on shared variables

- Components can also coordinate by imposing constraints on common variables



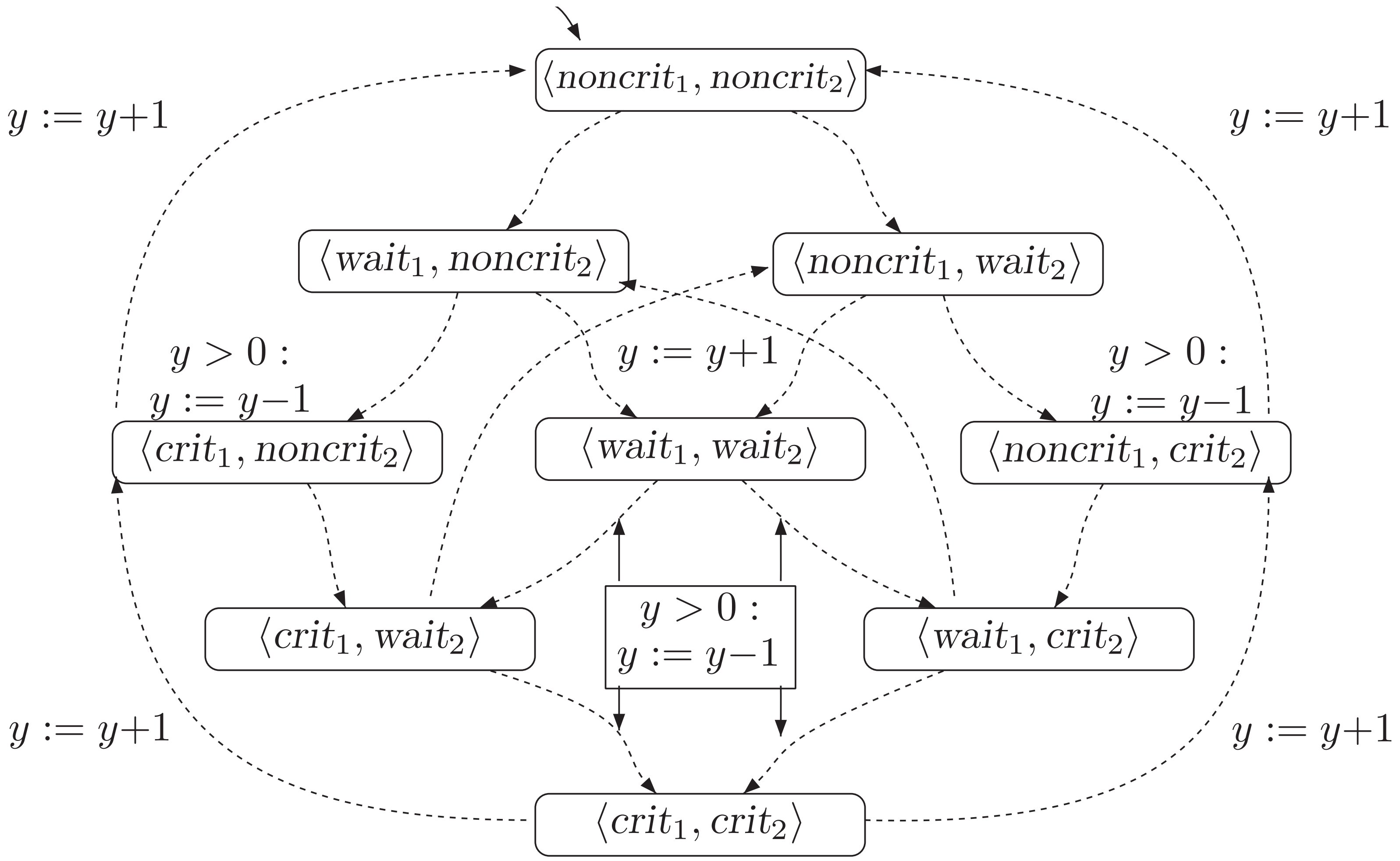
- Idea: The execution advances only if the conditions are satisfied in both transition systems

Example: Mutual exclusion with semaphores


$$\begin{tikzcd}[row sep=large] \arrow[d] & & \arrow[d] & \\ noncrit_1 \arrow[d, dashed] & & noncrit_2 \arrow[d, dashed] & \\ wait_1 \arrow[d, dashed, "{y > 0 : y := y-1}"] & & wait_2 \arrow[d, dashed, "{y > 0 : y := y-1}"] & \\ crit_1 \arrow[uu, dashed, bend left, "{y := y+1}"] & & crit_2 \arrow[uu, dashed, bend left, "{y := y+1}"] & \end{tikzcd}$$

y=0 indicates that the semaphore—the lock to get access to the critical section—is currently possessed by one of the processes. When y=1, the semaphore is free.
(NB: fundamental assumption: **atomicity of the assignment statements**)

PG1 ||| PG2



$\langle noncrit_1, noncrit_2\rangle$
$\langle wait_1, noncrit_2\rangle$
$\langle noncrit_1, wait_2\rangle$
$\langle crit_1, noncrit_2\rangle$
$\langle wait_1, wait_2\rangle$
$\langle noncrit_1, crit_2\rangle$
$\langle crit_1, wait_2\rangle$
$\langle wait_1, crit_2\rangle$
$\langle crit_1, crit_2\rangle$
$y := y+1$
$y := y+1$
$y > 0 :$ $y := y-1$
$y := y+1$
$y > 0 :$ $y := y-1$
$y > 0 :$ $y := y-1$
$y := y+1$
$y := y+1$

Unfolding $PG_1|||PG_2$ into $TS(PG_1|||PG_2)$

$$\begin{tikzcd}
 & & \langle n_1, n_2, y{=}1\rangle \arrow[dll] \arrow[drr] & & \\
 & & {} \arrow[u] & & \\
\langle w_1, n_2, y{=}1\rangle \arrow[d] \arrow[dr] & & & & \langle n_1, w_2, y{=}1\rangle \arrow[dl] \arrow[d] \\
\langle c_1, n_2, y{=}0\rangle \arrow[uuurr, bend left] \arrow[dr] & \langle w_1, w_2, y{=}1\rangle \arrow[d] \arrow[drr] & & & \langle n_1, c_2, y{=}0\rangle \arrow[uuull, bend right] \arrow[dl] \\
 & \langle c_1, w_2, y{=}0\rangle \arrow[uurrr, bend right] & & \langle w_1, c_2, y{=}0\rangle \arrow[uulll, bend left] &
\end{tikzcd}$$

It can be easily checked that the global state <crit1,crit2,y=...> is unreachable: the synchronization is correct

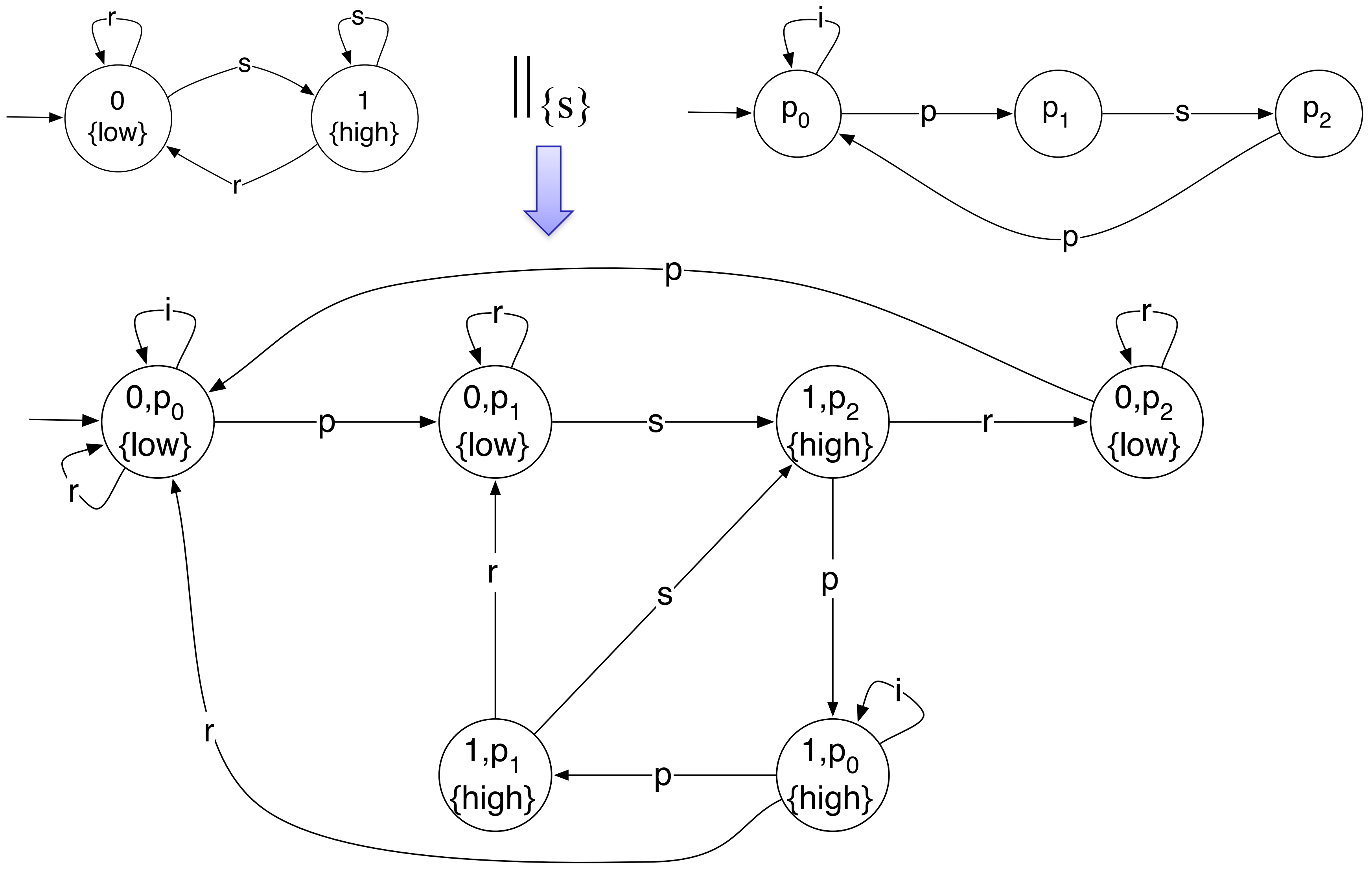
Remark: atomicity of statements

- For modeling a parallel system by means of the interleaving operator for program graphs it is decisive that the actions α ∈ Act are **indivisible**. The transition system representation only expresses the effect of the completely executed action α.
- Example: action α is: $x := x + 1;\ y := 2x + 1;\ \textbf{if}\ x < 12\ \textbf{then}\ z := (x - z)^2 * y\ \textbf{fi}$,

then an implementation is assumed which does *not* interlock the basic substatements $x := x+1$, $y := 2x+1$, the comparison "$x < 12$", and, possibly, the assignment $z := (x-z)^2 * y$ with other concurrent processes. In this case,

HANDSHAKING

Example of parallel composition, sync



TS: Parallel Composition || with Handshaking

- **$TS_1 \|_H TS_2$**: they synchronize on actions in a set H subset of $Act_1 \cap Act_2$
- They evolve independently (interleaving) on other events
  - Similar to firing a transition in petri nets
  - To synchronize, processes need “to shake hands”
  - HS also called **Synchronous Message Passing**
- If $Act_1 \cap Act_2$ is empty then handshaking is a special case of interleaving.

$$TS_1 \;\|_{\varnothing}\; TS_2 \quad = \quad TS_1 \;|||\; TS_2.$$

- If H= $Act_1 \cap Act_2$ then we just write ||

Rules for Handshaking

$TS_1 = \langle S_1, Act_1, \rightarrow_1, I_1, AP_1, L_1 \rangle$, $TS_2 = \langle S_2, Act_2, \rightarrow_2, I_2, AP_2, L_2 \rangle$

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 $\alpha \in H$:

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

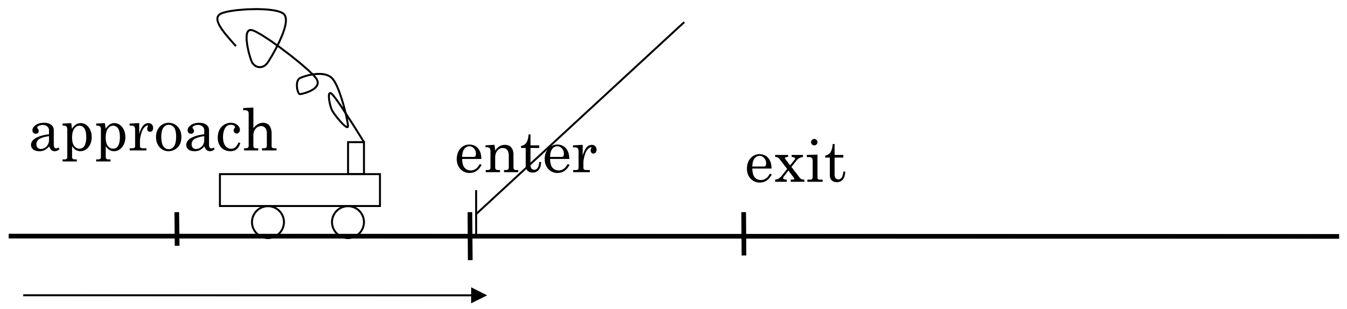
Broadcasting with n processes

- Consider a **fixed set H** of handshake actions, H subset of $Act_1 \cap Act_2 \ldots \cap Act_n$
- All processes can thus synchronize on the same actions.
- The operator $||_H$ is associative in this case:

$$TS = TS_1 \parallel_H TS_2 \parallel_H \ldots \parallel_H TS_n,$$

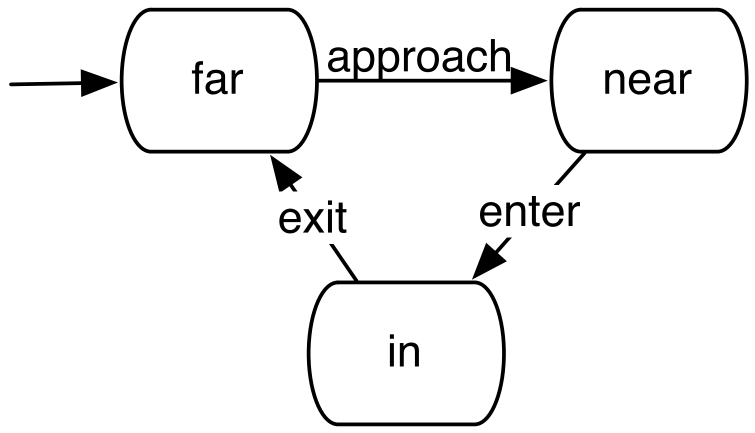
Example: Railroad Crossing Problem

- Widely used in the literature as an interesting benchmark for rigorous (formal) analysis methods

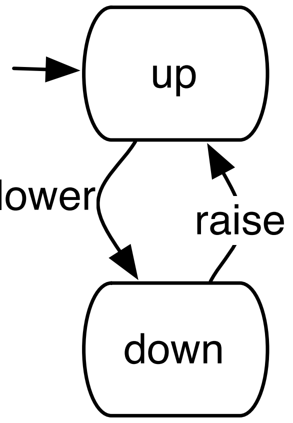


- Assumptions:
  - only one track;
  - only one train can approach the crossing at a time;
  - instantaneous opening and closing of the gate (ignore timing issues);
  - controller only receives approach and exit events, but no enter signal.
- Actions:
  - approach: A train is approaching the crossing
  - enter: the train is entering the section where the gate is located
  - exit: the train exits the crossing
  - lower: a command to lower the bar of the gate is issued
  - raise: a command to raise the bar of the gate is issued
- H={approach, exit, lower, raise}

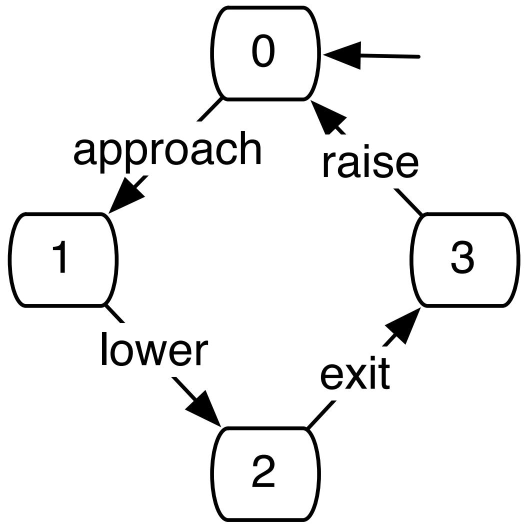
Railroad crossing example (no timing)



train



gate



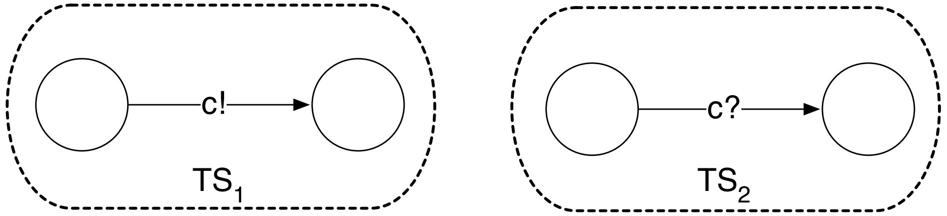
controller

train || gate || controller

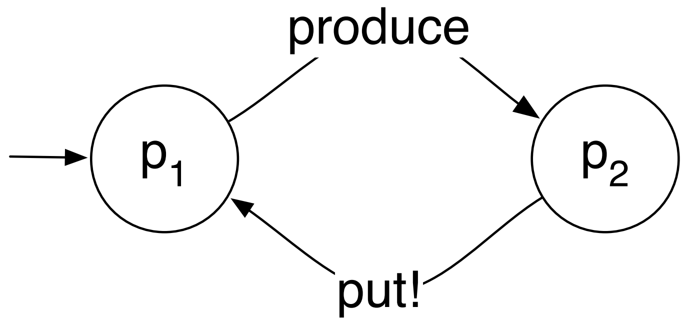
CHANNEL SYSTEMS

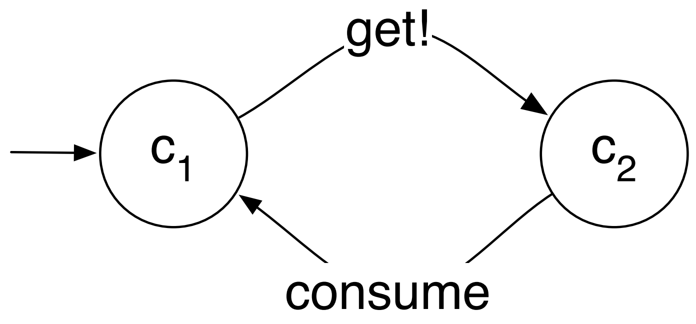
Composition through channels

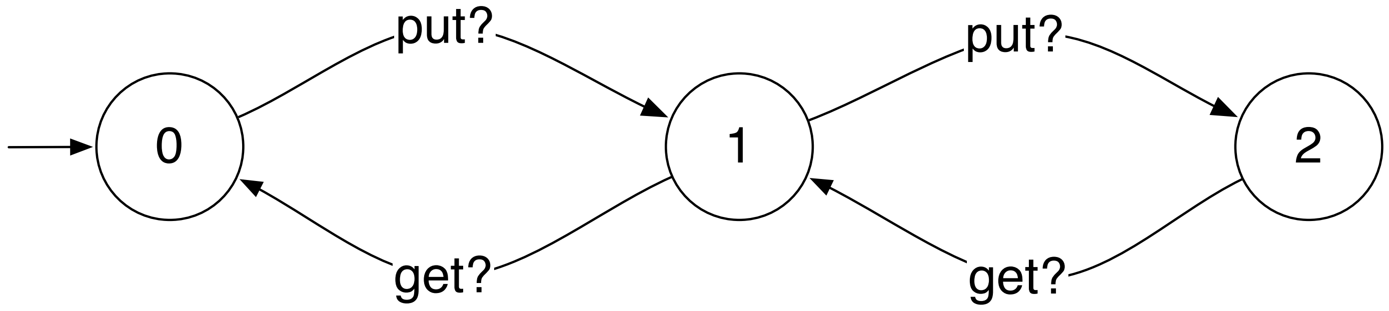
- Handshaking does not introduce a notion of *direction* in the messages that are exchanged
  - or, in other words, a cause-effect relationship in the synchronization
- However, a direction is a natural thing to model
  - one component issues a message, the other receives it
- We use fifo *channels* to represent this directionality
  - now it is $\rightarrow \subseteq S \times (Act \cup C! \cup C?) \times S$
  - C! is set of messages of the form c!x, C? of the form c?x.
  - c!x: component sends a message x through channel *c in C*
  - c?x: component receives a message x through channel *c*



Example of composition through channels







Channel capacity

- Capacity of a FIFO channel is the maximum number of events that can be stored in the channel (buffer)
- If capacity(c)=0 then this is still handshaking (**synchronous message passing**), but with a different syntax
- If capacity(c)>0 and the number of events in c is less than the capacity, then a TS may execute c!x *without waiting for a c?x*
  - If *c is full*, then the *TS is suspended* until a receiver does c?x
  - Receiver doing c?x is suspended until x is at the front of the channel
  - This is called **Asynchronous Message Passing**
  - **NB:** For formal definition of asynch. message passing see Baier&Katoen

EXAMPLE: THE NANO-PROMELA LANGUAGE OF SPIN

Nano-Promela

- TS are a mathematical basis to model and verify reactive systems
- In practice, we need more «user-friendly» specifications languages.
- The SPIN model checker has an interesting language, called Promela, to describe transition systems
- We describe here a subset, Nano-Promela

Syntax

- A Promela program is a set of interleaving processes communicating either synchronously or by finite FIFO channels
- Syntax of statements

$$\begin{array}{lll}
\mathsf{stmt} & ::= & \mathtt{skip} \mid x := \mathsf{expr} \mid c?x \mid c!\mathsf{expr} \mid \\
& & \mathsf{stmt}_1 \,;\, \mathsf{stmt}_2 \mid \mathtt{atomic}\{\mathsf{assignments}\} \mid \\
& & \mathbf{if} \quad :: g_1 \Rightarrow \mathsf{stmt}_1 \ \ldots \ :: g_n \Rightarrow \mathsf{stmt}_n \quad \mathbf{fi} \quad \mid \\
& & \mathbf{do} \quad :: g_1 \Rightarrow \mathsf{stmt}_1 \ \ldots \ :: g_n \Rightarrow \mathsf{stmt}_n \quad \mathbf{do}
\end{array}$$

  - Expr is an expression, whose syntax is not important here. NO SIDE EFFECT!
  - x:=expr is an (atomic) assignment

Intuitive meaning

- **skip** stands for a process that terminates in one step, without affecting the values of the variables or contents of the channels.
- **stmt1 ; stmt2** denotes sequential composition, i.e., stmt1 is executed first and after its termination stmt2 is executed.
- **atomic{stmt}** defines an atomic region.
  - The effect is that the execution of stmt is treated as an atomic step that cannot be interleaved with the activities of other processes.
  - Useful to avoid interference, but also to reduce verification overload by eliminating useless interleavings.

$$\textbf{if} \;::\; g_1 \Rightarrow \mathsf{stmt}_1 \;\ldots\; ::\; g_n \Rightarrow \mathsf{stmt}_n \;\textbf{fi}$$

- Stands for a nondeterministic choice between the statements stmt_i for which the guard gi is satisfied in the current state
- A test-and-set semantics
  - the choice between the enabled guarded commands and the execution of the first atomic step of the selected statement, are performed as an atomic unit that cannot be interleaved with the actions of concurrent processes.
- If none of the guards $g_1, \ldots, g_n$ is fulfilled in the current state, then the if–fi–command blocks.
  - other processes that run in parallel might abolish the blocking by changing the values of shared variables such that one or more of the guards may eventually evaluate to true.

$$\mathbf{do} \ :: g_1 \Rightarrow \mathsf{stmt}_1 \ \ldots \ :: g_n \Rightarrow \mathsf{stmt}_n \ \mathbf{od}$$

- Stands for the iterative execution of the nondeterministic choice among the guarded commands $g_i \Rightarrow stmt_i$, where guard g_i holds in the current configuration.
- Unlike conditional commands, do–od-loops do not block in a state if all guards are violated: the loop is just aborted.
- For instance, $\mathbf{do} \ :: g \Rightarrow \mathsf{stmt} \ \mathbf{od}$
- has the same effect of *while(g) do stmt*; in a programming language

The Vending Machine

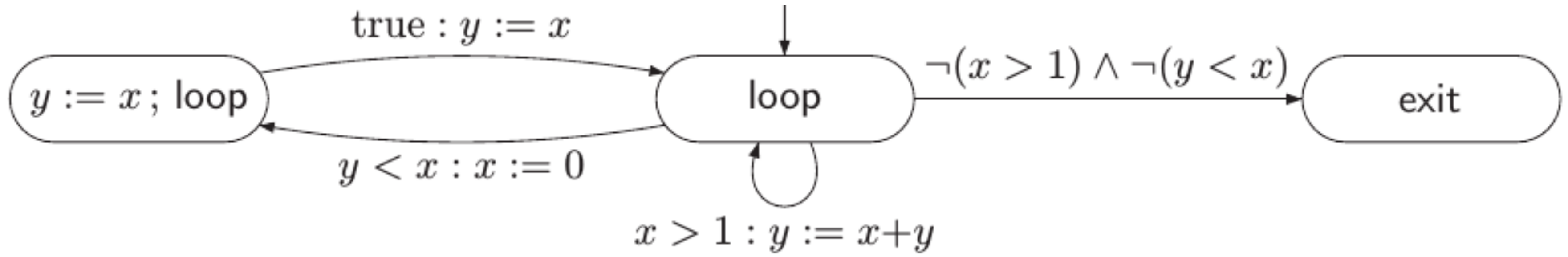
```
do  ::  true ⇒
          skip;
          if    ::  $nsoda > 0$  ⇒  $nsoda := nsoda - 1$
                ::  $nbeer > 0$  ⇒  $nbeer := nbeer - 1$
                ::  $nsoda = nbeer = 0$ ⇒ skip
          fi
    ::  true ⇒  atomic{$nbeer := max; nsoda := max$}
od
```

Other features

- Semantics can be given in terms of Program Graphs
- Promela provides many more features than nanoPromela, such as
  - atomic regions with more complex statements than sequences of assignments,
  - arrays and more data types
  - dynamic process creation.
  - We refer to the literature on the model checker SPIN

Example of semantics

$$\begin{array}{lcllcl} \mathsf{loop} & = & \mathbf{do} & :: \; x > 1 & \Rightarrow & y := x + y \\ & & & :: \; y < x & \Rightarrow & x := 0;\; y := x \\ & & \mathbf{od} & & & \end{array}$$



Find the error in this mutex algorithm

```
bit flag=0; /* signal entering/leaving the section */
proctype P(bit i) { /a process type */
            do ::flag==1 -> skip; od
            flag = 1;
crit:       skip;    /* critical section */)
            flag = 0;
}
init {
atomic { run P(0); run P(1);}
```

Safety Property to be verified: no 2 process are in location “crit” at the same time.

“Progress”: every process can enter crit. “infinitely many times”

A better version (Peterson's algorithm, 1981)

```
 bool flag[2]; //initially false, false
 bool turn=0;
 active [2] proctype user() {//two instances of the process are being run
            //process ready:
            flag[_pid] = true; //_pid is 0 or 1
            turn = _pid; /* select turn*/
            do
              ::(flag[1-_pid] == true && turn == _pid) -> skip;
            od
crit:       skip;      /* critical section */
            flag[_pid] = false /* release the resource */
  }
```

Proof of mutex and progress can be tricky, but verification in Spin is trivial.

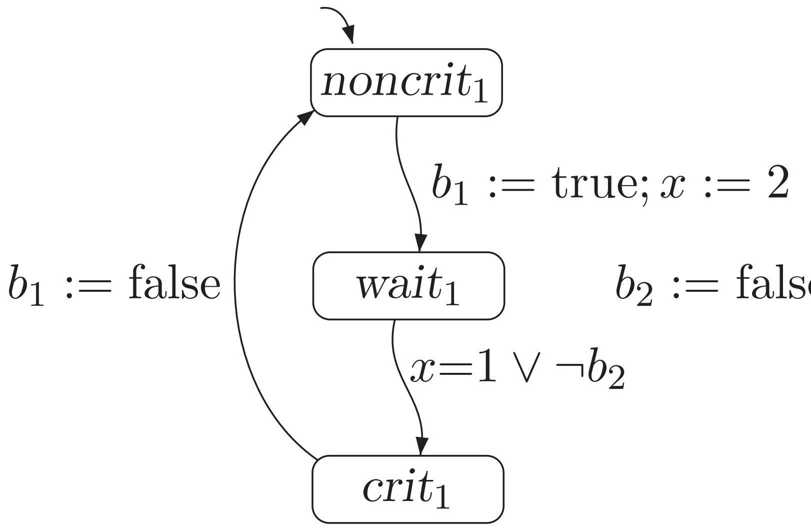
Abstract version of Peterson’s Algorithm and its PGs

<…> denotes an atomic region

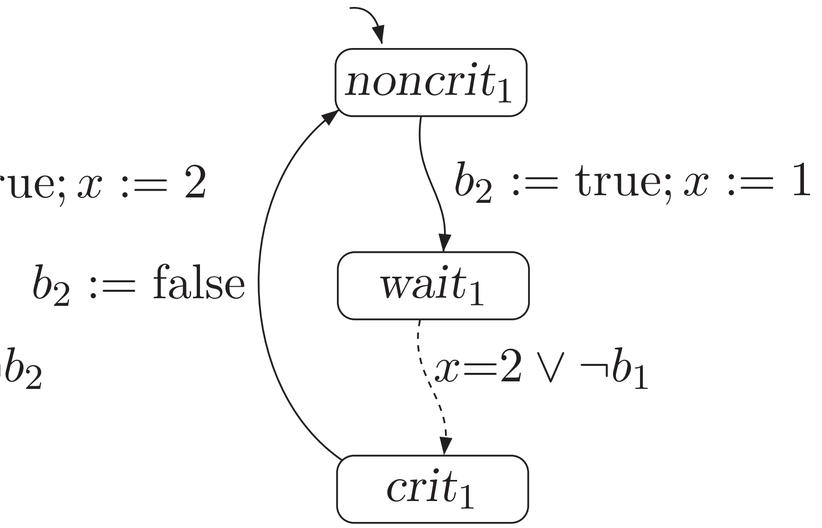
```
P₁   loop forever
       ⋮                                              (* noncritical actions *)
       ⟨$b_1$ := true; $x$ := 2⟩;                       (* request *)
       wait until ($x = 1 \lor \neg b_2$)
       do critical section od
       $b_1$ := false                                    (* release *)
       ⋮                                              (* noncritical actions *)
     end loop
```

P_2 is identical, with x:=1 and flag variable b_2

PG_1 :



PG_2 :



$noncrit_1$
$b_2 := \text{true}; x := 1$
$wait_1$
$b_2 := \text{false}$
$x{=}2 \lor \neg b_1$
$crit_1$