Modeling and Analyzing Concurrent Systems

Robert B. France

Overview

- Why model and analyze concurrent systems?
- How are concurrent systems modeled?
- How are concurrent systems analyzed?
- What tools are available for modeling and analyzing concurrent systems?

References

Principles of Model Checking

Christel Baier and Joost-Pieter Katoen, MIT Press

Some of the slides use diagrams and text extracted from the above book

Tool: UPPAAL model checker

http://www.uppaal.org/

Overview

- Why model and analyze concurrent systems?
- How are concurrent systems modeled?
- How are concurrent systems analyzed?

Why model concurrent systems?

- Distributed, concurrent systems are becoming commonplace, but they are notoriously difficult to develop
  - network applications, data communication protocols, multithreaded code, client-server applications
- Concurrency-specific errors: deadlock, livelock
  - A deadlock occurs when the system has reached a state in which no work is done but at least one process in the system needs to complete its tasks
  - A livelock occurs when the processes in a system are stuck in a repetitive task and make no progress towards their functional goals.
- These types of behavioral errors can be mechanically detected if the systems are properly modeled and analyzed

Common flaws in concurrent system modeling

- Underspecification: Model is incomplete, imprecise or allows behavior that should not be allowed (i.e., model is too permissive).
- Overspecification: Model disallows behavior that should be allowed, that is, model is to restrictive
- Violations of **safety** properties: A safety property is a property that must not be violated
  - "nothing bad happens"; a bad behavior should never occur
  - An invariant is an example of a safety property
  - Example 1: Mutual exclusion property – at most one process is in its critical section at any given time
  - Example 2: Absence of **deadlocks**
- Violations of **liveness** properties: Set of properties that a system must satisfy, i.e., properties that require desired events to eventually occur
  - "something good eventually happens"
  - Example 1: Starvation freedom, e.g., each process waiting to enter its critical section will eventually enter its critical section.
  - Example 2: Progress: A process will eventually perform a non-skip step

What is Model Checking?

- "Model checking is an automated technique that, given a **finite-state model** of a system and a **logical property**, systematically checks whether this property holds for (a given initial state in) that model." [Clarke & Emerson 1981]:
- Model checking tools automatically verify whether $M \models \varphi$, holds, where *M is a (finite-state) model of a system and* property φ is stated in some formal notation.

Model Checking process

1. Construct a model of the system (M)
2. Formalize the properties of the system that will be evaluated in the model (P)
3. Use a model checker to determine if M satisfies P. Three results are possible:
    1. The model M satisfies the property P, i.e. M |= P
    2. M does not satisfy P; in this case a counterexample is produced
    3. No conclusive result is produced by the model checker (model checker ran out of space or time)

What is meant by “model” in “model checker”?

- The term “model” as used in “model checker” is an assignment of values to variables in a logical formula that makes the formula true. Alternatively, a formula defines a family of “models” or instances (where an instance satisfies the formula)
  - For example, a model of a proposition is an assignment of truth values to the proposition variables that makes the proposition true (e.g., a line in a truth table is a model)
- A model checker checks whether a system model is an instance of the property
  - That is, it checks if the system model is an assignment of values to variables in the property that makes the property true.

Model of a formula: An example

- Some y: Year, All s:Student| advisedByRF(s) and completedThesis(s,y) implies academicPosition(s)
- Model
  - Student = {DT, DS, RR, ES, EG}
  - advisedByRf = {DS, ES, RR, EG}
  - completeThesis = {(DT,98), (DS, 08), (RR,08), (ES,08), (EG,03)}
  - academicPositions = {EG, DS, RR, ES}

Is the Alloy Analyzer a Model Checker?

- No! It is a Model Finder
- The Analyzer generates an instance that satisfies the constraints in signatures, facts and the condition in the predicates or assertions.

Use of the term “model” in this course

- We use the term “model” in this course to refer to an abstraction of a software system
- We’ll continue to use the term in this sense
- When model-checking a software model against a formally expressed property we’re checking that the software model is a mathematical model of the property.

Overview

- Why model and analyze concurrent systems?
- How are concurrent systems modeled?
- How are concurrent systems analyzed?

How can we describe a system so that it can be mechanically model-checked?

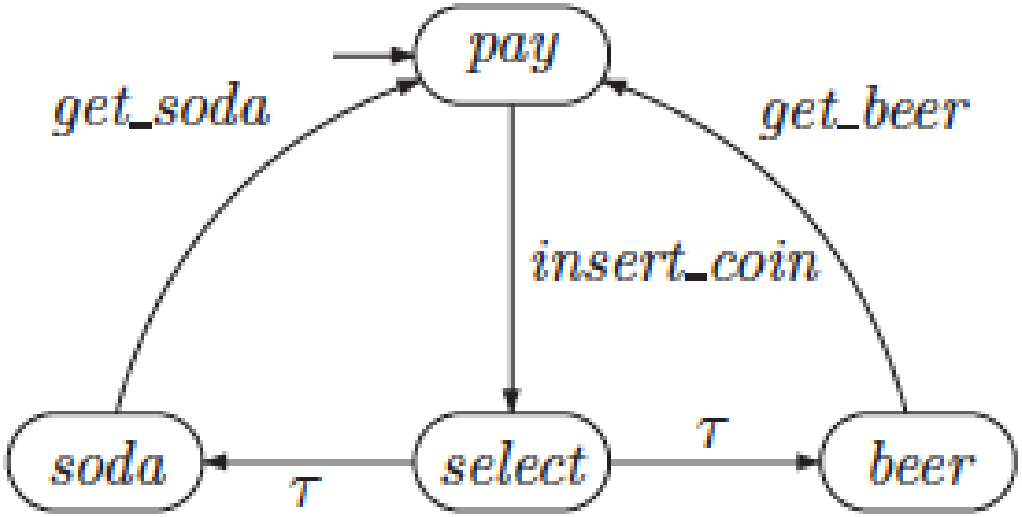
- Focus on linear temporal behavioral properties
  - Linear model of time; no branching in the timeline over which behaviors are observed
- Behaviors expressed in terms of Transition Systems that describe the effect of operations on the system's state.
- A **linear temporal (LT)** property characterizes a set of state transitions
- A model satisfies a linear temporal property if the state transitions it defines are all included in the transitions characterized by the LT property.

Transition systems

Using Transition Systems to model system behavior

- A **Transition System** (TS) is a directed graph where nodes represent states and edges represent transitions between states
- A **state** describes information about a system at a particular point in time (cf. state in Alloy)
  - E.g., the state of a traffic light indicates the color of the light that is illuminated at a point in time
- A **transition** describes the conditions under which a system moves from one state to another.

A (toy) example of a simple TS



Transitions are associated with action labels that indicate the actions that cause the transition.

- insert_coin is a user action
- get_soda, get_beer are actions performed by the machine
- τ denotes an activity that is not of interest to the modeler (e.g., it represents an internal activity of the vending machine)

Transition System (TS): Formal Definition

A *transition system TS* is a tuple (*S, Act,→, I,AP, L*) where

- *S* is a set of states,
- *Act* is a set of actions,
- -> $\subseteq S \times Act \times S$ is a transition relation (the first element in the triplet is the source state, the second element is an action and the third element is the target state of the transition)
- $I \subseteq S$ is a set of initial states,
- *AP* is a set of atomic propositions, and
- $L : S \rightarrow 2^{AP}$ is a labeling function (2^{AP} is the power set of AP)

TS is called *finite* if *S*, *Act*, and *AP* are finite.

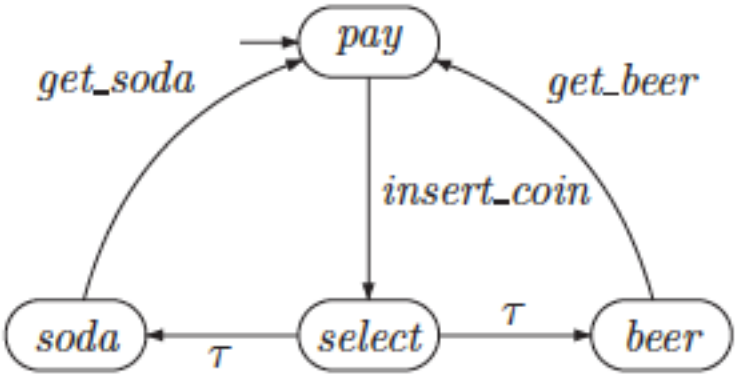
(s, act, s') in -> is written as s -^{act}-> s'

L(s) are the atomic propositions in AP that are satisfied in state s.

Given a formula, f, a state s satisfies f (i.e., is a model of f) if and only if f can be derived from the atomic propositions associated with state s via the labeling function L, that is:

s |= f iff L(s) |= f

Toy example again



S = {pay, select, soda, beer}

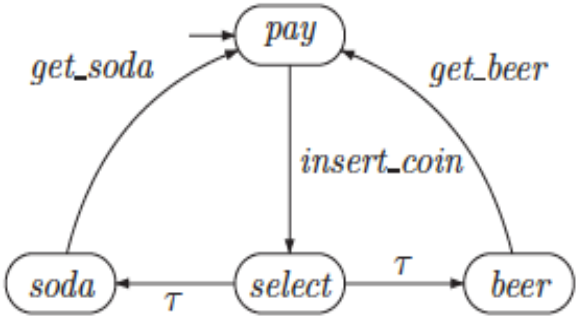
I = {pay}

Act = {insert_coin, get_soda, get_beer, T}

-> = {(pay, insert_coin, select), (beer, get_beer, pay), (soda, get_soda, pay), (select, τ , soda), (select, τ , beer)}

Atomic propositions in the toy Example

The atomic propositions in a transition system are chosen based on the properties the modeler wants to check.



Example property to verify: The vending machine only delivers a drink after the user pays (inserts a coin).

Relevant atomic propositions: AP = {paid, delivered}

Appropriate Labeling function:

L(pay) = empty set

L(soda)=L(beer)={paid, delivered}

L(select)={paid}

Using non-determinism to under-specify a problem

- The toy model is non-deterministic: When the system enters the “select” state, the transition system non-deterministically chooses to dispense beer or soda (i.e., it makes a choice that cannot be determined beforehand by examining the model).
- In this case the model deliberately abstracts over the mechanism a vending machine customer uses to select soda or beer
  - In other words, the modeler is leaving open the choice of how this is done; someone implementing the model needs to resolve this non-determinism to make the system deterministic
- Non-deterministic choice is also used to model concurrent (parallel) behavior as we will see later.

Action-Deterministic TS

- TS = (S, Act, ->, I, AP, L) is action-deterministic if
  - There is at most one initial state
    - #(I) <= 1 (# returns the number of elements in its set argument)
  - For all states s in S and actions act in Act, there is at most one transition labeled with the action act that leaves the state s, i.e.
    - #(Post(s,act)) <= 1, where Post(s,act) are all the target states associated with s via transitions labeled with act; i.e., Post(s,act) = {s': State | s –^{act}-> s'}

AP-Deterministic TS

- TS = (S, Act, ->, I, AP, L) is AP-deterministic if
  - There is at most one initial state
    - #(I) <= 1
  - For all states s in S and proposition A in 2^{AP}, there is at most one next state s' in which A holds
    - For all states s in S, and A in 2^{AP}, #(Post(s) intersect {s':State | L(s') = A}) <= 1, where Post(s) consists of all the target states associated with s via transitions; i.e., Post(s) = $U_{act\ in\ Act}$ Post(s,act)

Observable behavior

- Often useful to have behavior that is observable by external agents be deterministic
- Two observable views
  - Action-based view: only the actions are observable
  - State-based view: only the states, via the propositions associated with them, are observable

The two notions of deterministic behavior discussed in the previous slides support these views.

Executions of a TS

- TS Executions formalize the notion of behavior in a modeled system
- A **finite execution fragment** of a TS is a sequence of state transitions.
  - For example, s0-act1->s1, s1-act2->s3, is written as an alternating sequence of states and actions that ends in a state, s0,act1,s1,act2,s3
- An **infinite execution fragment** is an infinite sequence of transitions
- A **maximal execution fragment** is either a finite execution fragment that ends in a final state, or an infinite execution fragment.
  - An execution fragment is called initial if it starts in an initial state.
- An execution of a transition system is an initial maximal execution fragment

Executions of the vending machine

$$\rho_1 = pay \xrightarrow{coin} select \xrightarrow{\tau} soda \xrightarrow{sget} pay \xrightarrow{coin} select \xrightarrow{\tau} soda \xrightarrow{sget} \ldots$$

$$\rho_2 = select \xrightarrow{\tau} soda \xrightarrow{sget} pay \xrightarrow{coin} select \xrightarrow{\tau} beer \xrightarrow{bget} \ldots$$

$$\varrho = pay \xrightarrow{coin} select \xrightarrow{\tau} soda \xrightarrow{sget} pay \xrightarrow{coin} select \xrightarrow{\tau} soda \ .$$

Execution fragments ρ_1 and ϱ are initial, but ρ_2 is not. ϱ is not maximal as it does not end in a terminal state. Assuming that ρ_1 and ρ_2 are infinite, they are maximal. ■

Reachability of states: A state in a transition system is reachable if there is an initial finite execution fragment that ends in s.

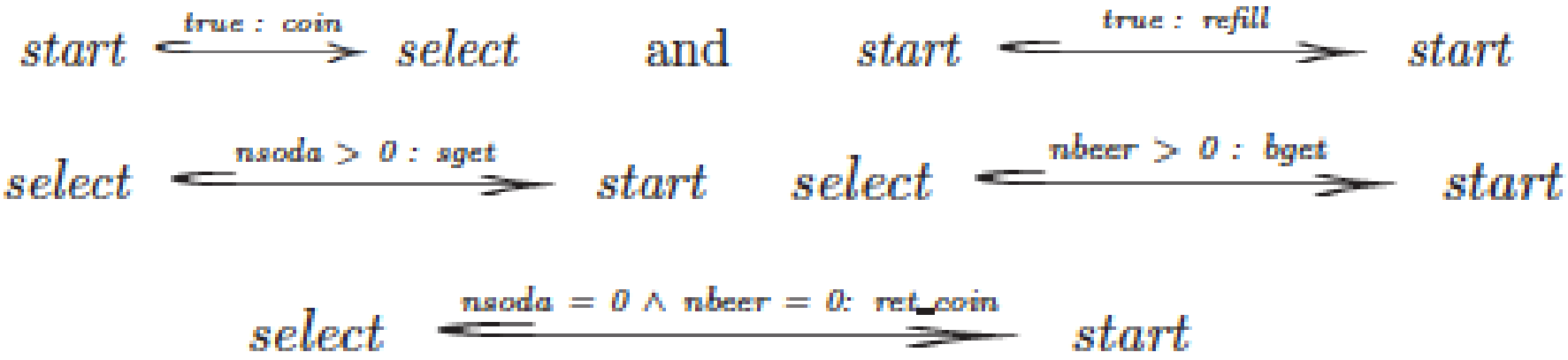
Modeling concurrent systems that manipulate data

- In software the transition from one state to another often depends on conditions expressed in terms of data
  - **Conditional transitions** are higher-level constructs used to describe actions that are performed only under certain conditions
- Models with conditional transitions are called **program graphs**
  - Program graphs are "higher-level" in that they can be transformed into TSs (*Note: TSs do not have conditional transitions*) via a process called **unfolding**

Extended vending machine example

- Vending machine extended to:
  - to maintain information on number of beers and soda in machine
    - nsoda: variable that stores number of soda in vending machine at a particular time
    - nbeer: variable that stores number of beer in vending machine at a particular time
  - return coins entered by user if product is not available
    - ret_coin: represents the return coin action

Program graph of the extended vending machine



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

select and *start* are called **locations**

nsoda, and *nbeer* are **variables**

coin, refill, *sget*, *bget, ret_coin* are **actions**

A simple text representation of the vending machine PG

start:

coin; go to select

refill{nsoda := max; nbeer := max}; go to start

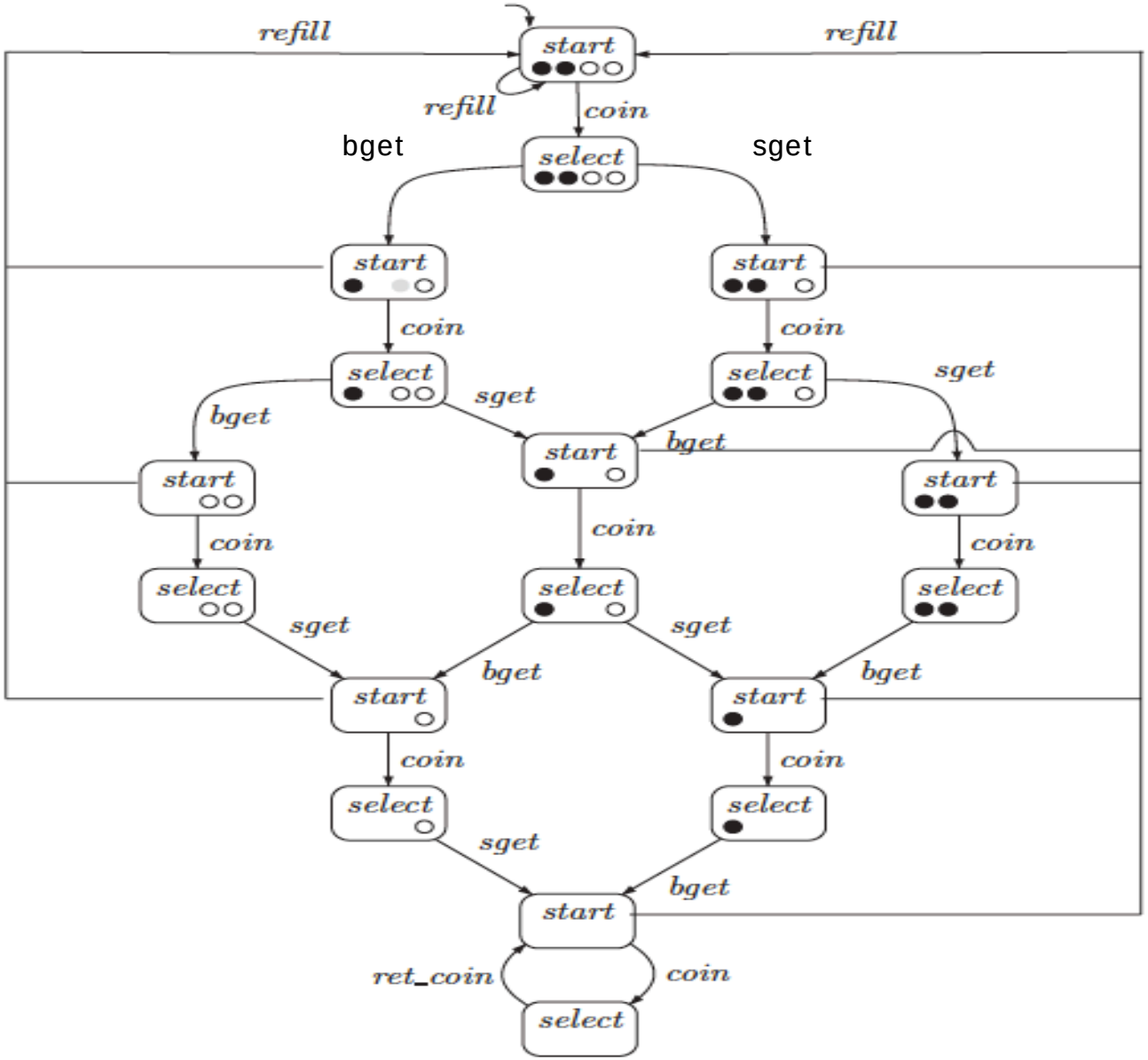
select:

nsoda > 0:: sget{nsoda := nsoda -1}; go to start

nbeer > 0:: bget{nbeer := nbeer-1}; go to start

nsoda = 0 and nbeer = 0:: ret_coin; go to start

Unfolding the vending machine PG



Program Graphs

- A program graph over a set of typed variables, *Var*, consists of nodes representing locations and edges representing conditional transitions
  - In the vending machine example Var = {nsoda, nbeer}
- A program graph also defines effects of actions on the variables
  - An effect is a function that takes an action and an assignment of values to variables and returns a new assignment of values to variables (the new assignment is the effect of the action)

$$Effect : Act \times Eval(Var) \rightarrow Eval(Var)$$

Program Graph (PG): Formal Definition

A *program graph PG* over set *Var* of typed variables is a tuple (*Loc, Act, Effect,->, Loc*0*, g*0) where

- *Loc* is a set of locations and *Act* is a set of actions,
- *Effect* : *Act* $\times$ *Eval*(*Var*) --> *Eval*(*Var*) is the effect function,
  - Eval(Var) is the set of assignments of values to variables in Var, e.g.,{ <nbeer:= 10, nsoda:=20>, <nbeer:= 1, nsoda:=20>, <nbeer:=0, nsoda:=4>, …} is the set of assignments when Var = {nbeer, nsoda}
- -> $\subseteq$ *Loc* $\times$ *Cond*(*Var*) $\times$ *Act* $\times$ *Loc* is the conditional transition relation,
  - Cond(Var) is the set of all Boolean conditions (propositions) over Var
- $Loc_0 \subseteq Loc$ is a set of initial locations,
- $g_0 \in Cond(Var)$ is the initial condition.

Vending machine program graph

Loc = {start, select}

Var = {nsoda, nbeer}

Act = {bget, sget, coin, ret_coin, refill}

Effect(coin, η) = η

Effect(ret_coin, η) = η

Effect(sget, η) = η [nsoda' = nsoda - 1]

Effect(bget, η) = η [nbeer' = nbeer - 1]

Effect(refill, η) = η [nsoda' = max, nbeer' = max]

In the above η is an assignment of values to variables in Var

η**[v' = f(v)]** means that the new assignment to variable v is a function, f, of the previous assignment of v and all other variable assignments are unchanged

TS semantics of program graphs

- The TS is produced by unfolding the program graph
  - You can think of unfolding as a representation of the execution of a program described by a PG
- A state consists of a location (a point in the program) and an assignment of values to variables: $\langle l, \eta \rangle$
- An initial state consists of an initial location and an assignment that satisfies the condition g_0 defined in the PG
  - $\langle l_0, \eta \rangle$ is an initial state if l_0 is an initial location and $\eta \models g_0$
- The propositions consists of the locations together with Cond(Var)
  - The proposition loc is true in any state of the form $\langle loc, \eta \rangle$, and false otherwise

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

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

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

Structured Operational Semantics

- The semantics defined previously is an example of SOS
- The semantics uses inference rules of the form

$$\frac{\text{premise}}{\text{conclusion}}$$

Using transition systems to model concurrent behavior

Concurrent systems

- A concurrent (parallel) system consists of multiple processes executing concurrently (in parallel).
- If a concurrent system consists of n processes, in which each process, $proc_i$, is modeled by a transition system TS_i, the concurrent system can be modeled by a transition system $TS = TS_1 \,||\, TS_2 \,||\, \ldots \,||\, Ts_n$
  - where || is a parallel composition operator

Types of parallel composition operators

- **Interleaving**
  - Actions of concurrent processes are interleaved in a non-deterministic manner
  - Used to model processes whose behaviors are completely independent (*asynchronous* system of processes)
- **Communication via shared variables**
  - A process can influence the behavior of another process by changing the value of a variable that is shared with the process
- **Handshaking**
  - Two processes that want to interact must synchronize their actions such that they take part in the interaction at the same time
- **Channel systems**
  - In a channel system processes interact by reading from and writing to channels connecting them

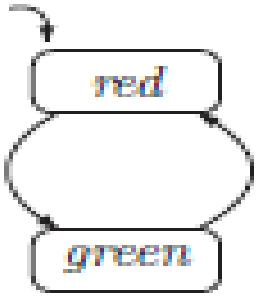
Interleaving

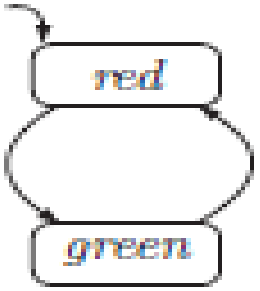
Interleaving of processes

- When processes can execute in a completely independent manner (with no interactions) one can view the system of processes as one system consisting of the actions of each process merged (interleaved) in an arbitrary manner
  - In this system concurrency means that the order in which the actions are performed does not affect the final result; i.e., *P1.act1;P2.act2* produces the same result as *P2.act2;P1.act1*, where *Pi.acti* is an action performed by process *Pi* (i=1 or i =2)
- The interleaving view is an abstraction in which only one processor is assumed available to execute the processes

Interleaving of Traffic Light Transition Systems

Consider a system with two traffic lights, each modeled by a transition system

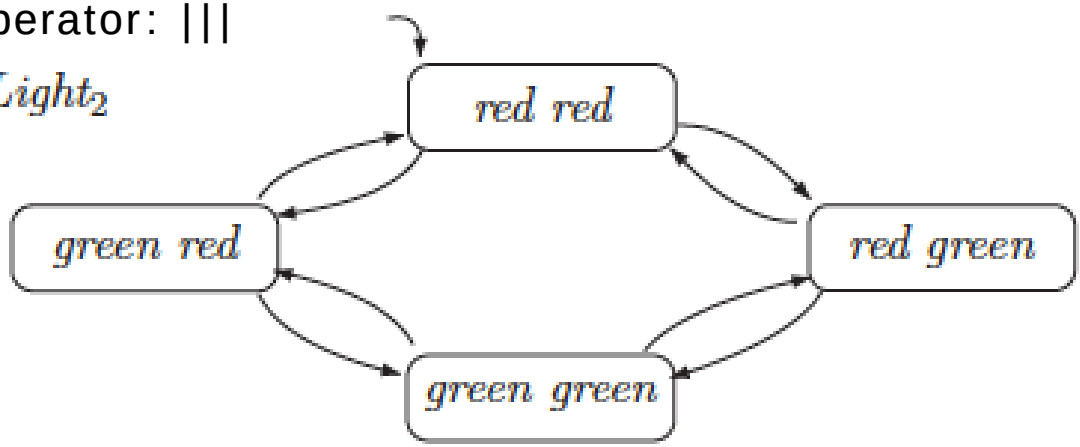




Interleaved System

Interleaving operator: |||

$TrLight_1 \;|||\; TrLight_2$



Effect of an interleaving operator

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

The above states that the order in which the actions α, β are performed does not matter.

||| is the interleaving operator

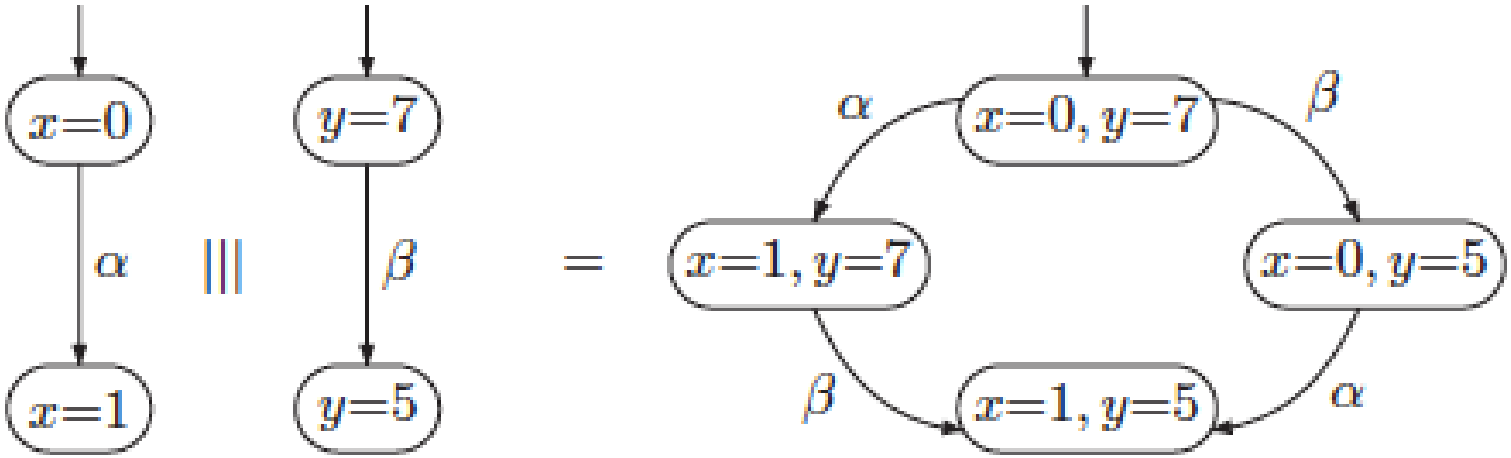
; is sequential composition

+ represents non-deterministic choice

Effect of an interleaving operator: An example

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

Note that variables are not shared across processes



Formal definition of interleaving operator

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 $L(\langle s_1, s_2\rangle) = L(s_1) \cup L(s_2)$. ■

Communication via Shared Variables

Modeling non-asynchronous systems

- Interleaving operator requires that processes are completely independent
- What happens if processes access data that is globally accessible (global data)?
- See example on next slide

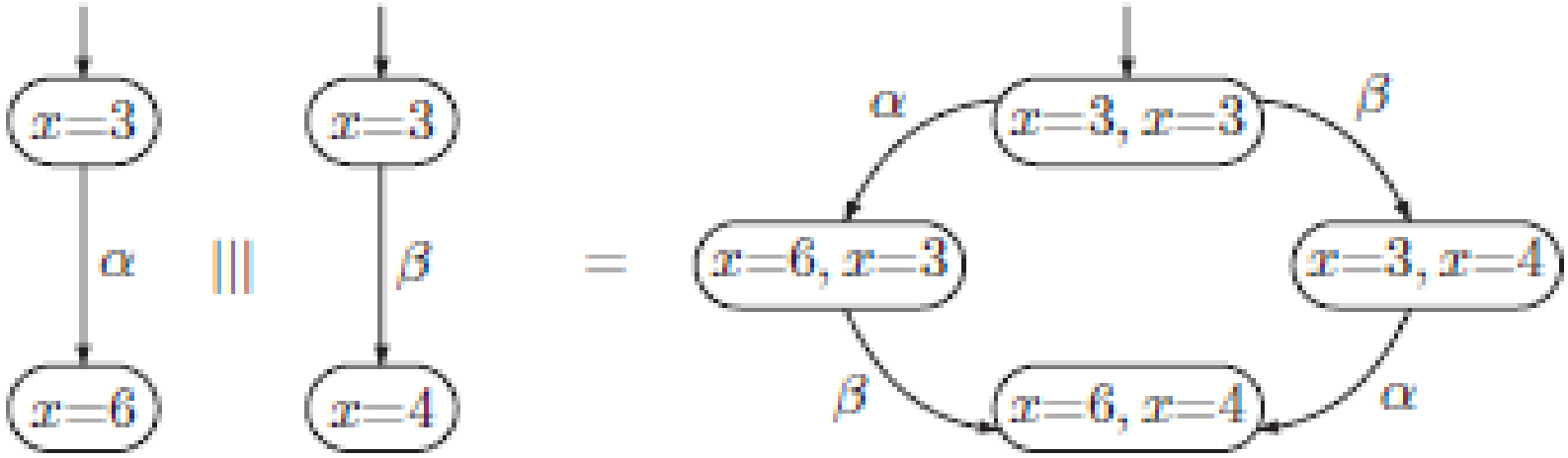
Interleaving in the presence of shared variables

Consider the program graph describing 2 actions from 2 processes, PG1, PG2, that access a global variable x (locations are omitted to simplify the presentations)

α: x := 2 * x

β: x := x + 1

(α ||| β) = (x := 2 * x ||| x := x + 1)

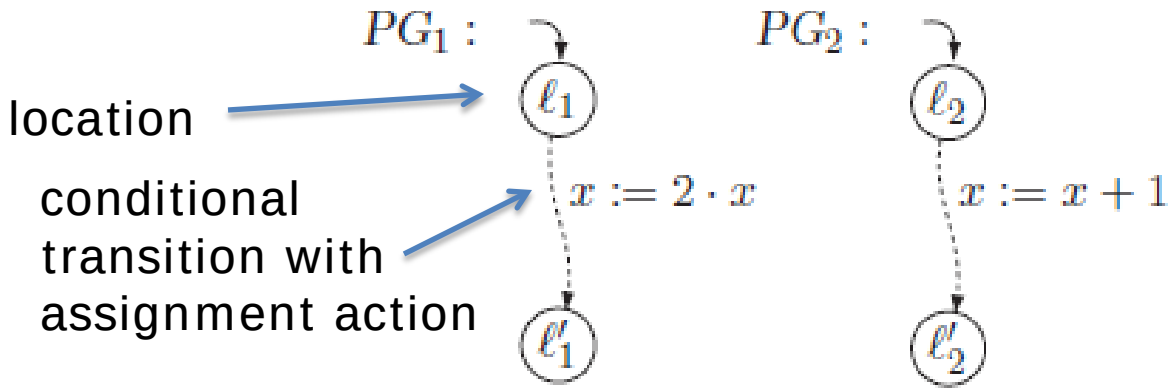


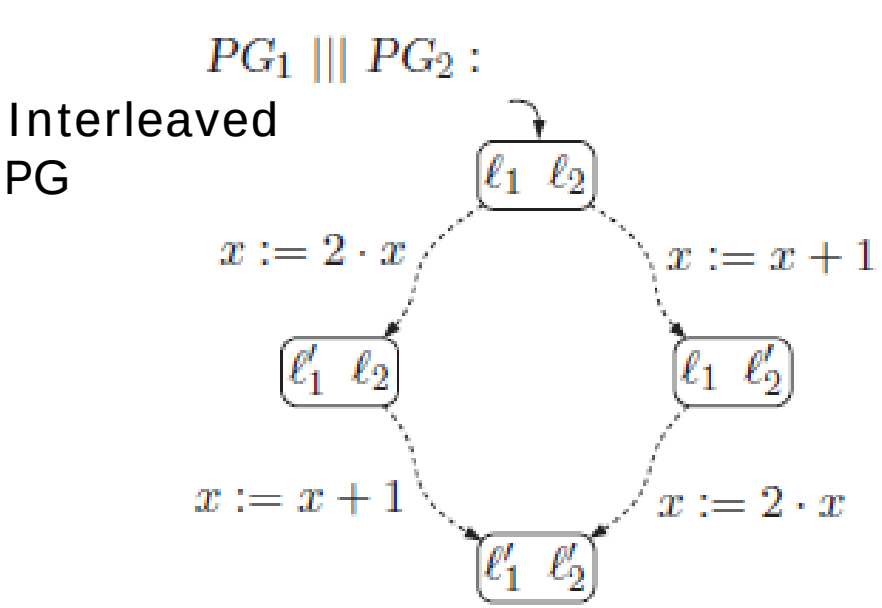
TS(PG1) TS(PG2) TS(PG1)|||TS(PG2)

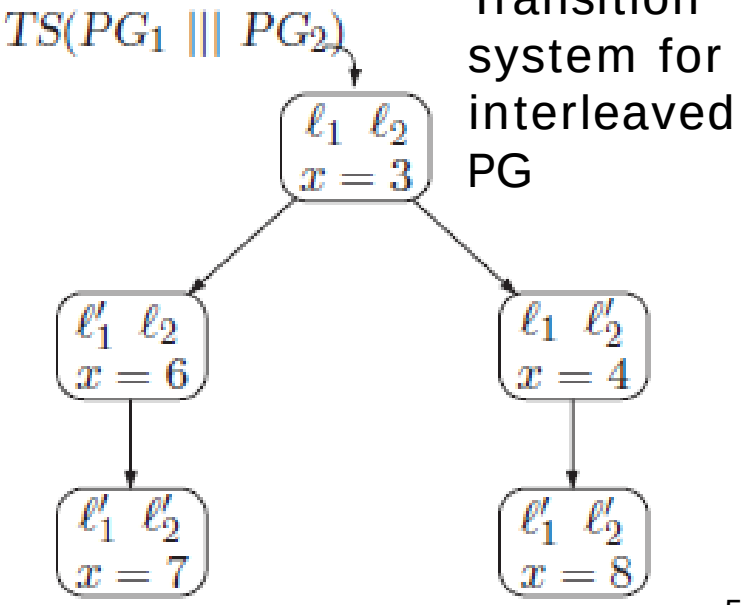
Modeling processes that access global variables

- An interleaving operator, |||, on program graphs (rather than transition systems) is used
  - PG1 ||| PG2
- TS(PG1 ||| PG2) describes a TS that treats shared variables appropriately
- In general,
  - TS(PG1 ||| PG2) ≠ TS(PG1) ||| TS(PG2)

Interleaving of Two Example Program Graphs







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 $\alpha \in Act_i$.

Non-determinism

- Non-determinism in a state of a TS produced by a interleaved PG can be interpreted in 3 ways:
  1. As an internal non-deterministic choice made in the PG
  2. As an interleaving of actions that access variables that are not shared (referred to as **non-critical actions**)
  3. As the resolution of a contention between actions of PG1 and PG2 that access global variables (referred to as **critical actions**)

Accessing global variables

- Critical actions are those that access global variables
- Access to global variables needs to be controlled
  - Only one critical action can access a global variable at any time
  - How do we ensure this? The mutual exclusion problem

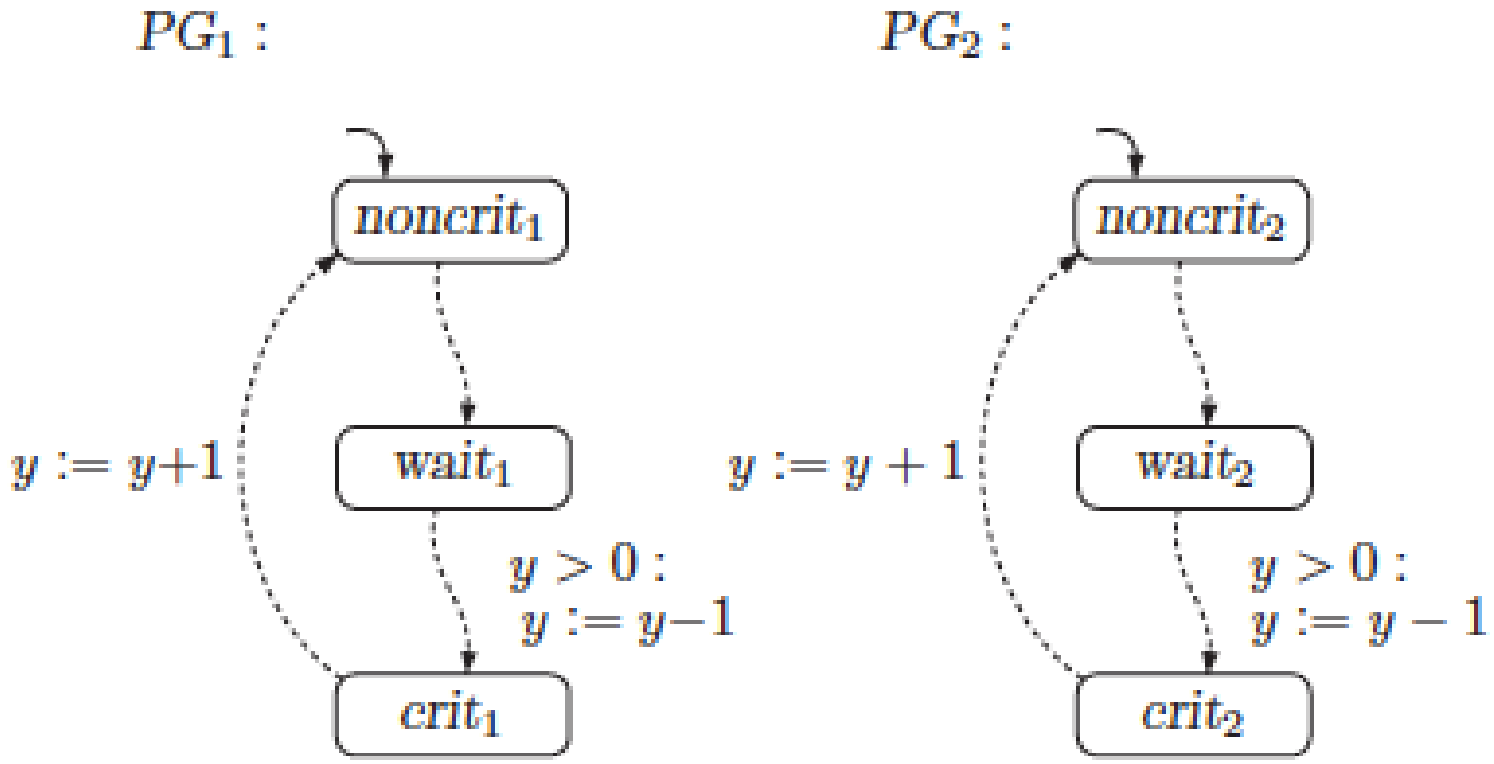
Mutual exclusion using semaphores

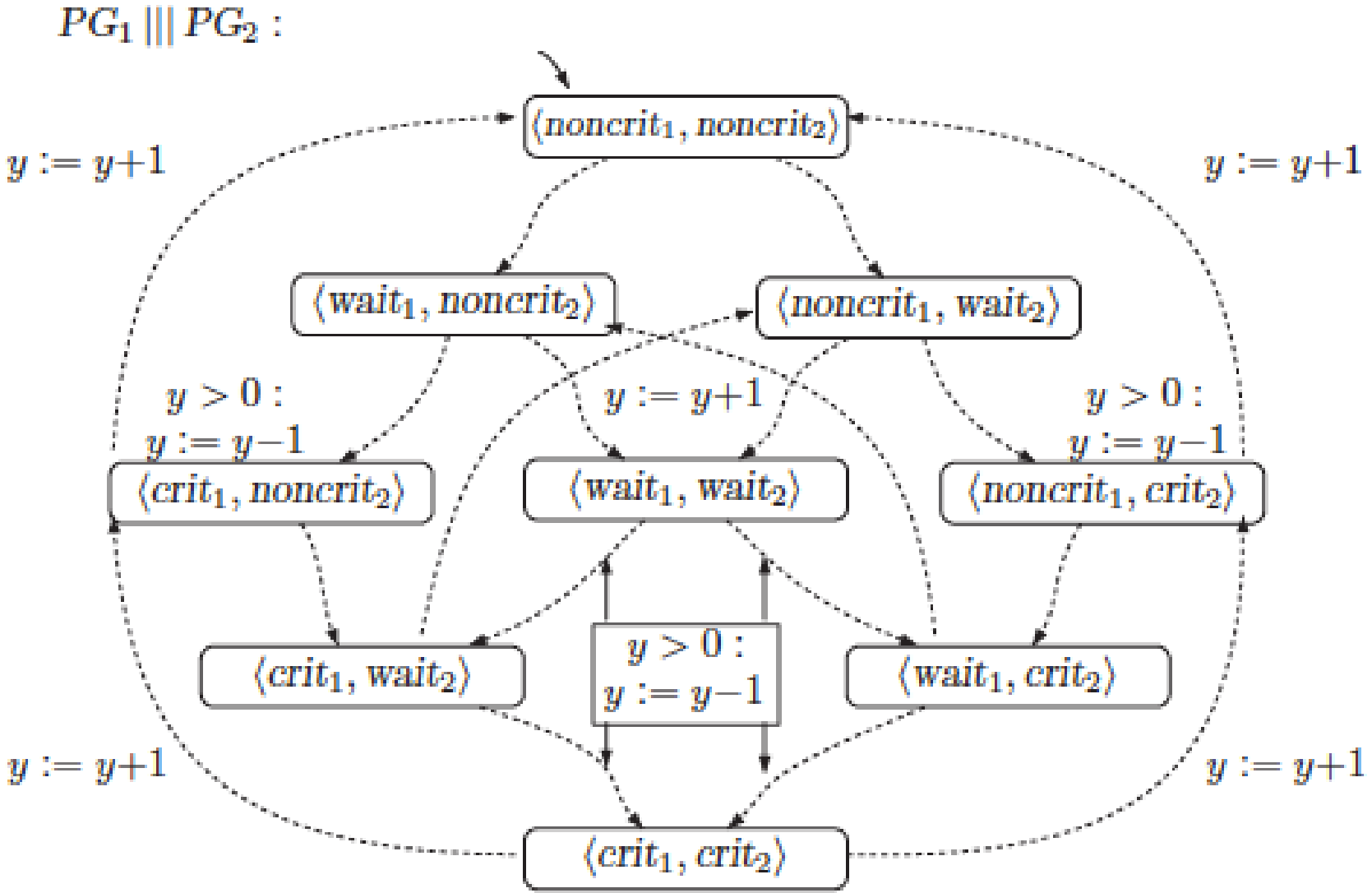
- Two processes with critical actions use a shared variable, y, called a semaphore to determine when they can perform their critical actions, i.e., enter their critical sections.
  - y = 0 indicates that one process is executing its critical actions (i.e., is in its critical section), and thus the other cannot execute its critical actions; The process that is executing its critical section in essence locks access to the global variables.
  - y = 1 indicates that none of the processes are in their critical sections (access to the global variables is unlocked)

Critical vs. non-critical sections

```
P_i   loop forever
      :                          (* noncritical actions *)
      request
      critical section
      release
      :                          (* noncritical actions *)
      end loop
```

Program graphs for semaphore-based mutual exclusion

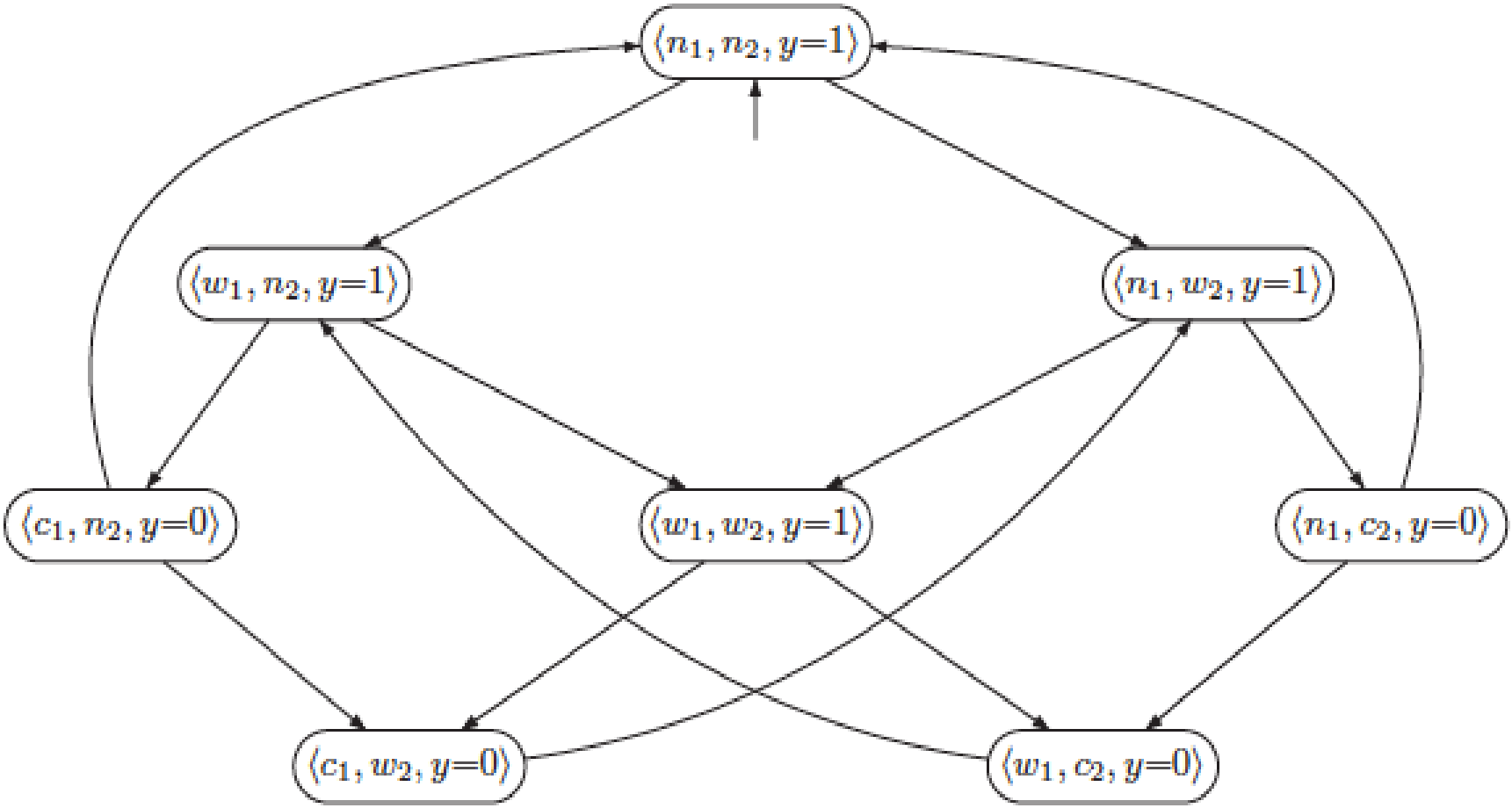




Reachable states

$\langle noncrit_1, noncrit_2, y = 1 \rangle$
$\langle wait_1, noncrit_2, y = 1 \rangle$
$\langle noncrit_1, crit_2, y = 0 \rangle$
$\langle wait_1, crit_2, y = 0 \rangle$
$\langle noncrit_1, wait_2, y = 1 \rangle$
$\langle wait_1, wait_2, y = 1 \rangle$
$\langle crit_1, noncrit_2, y = 0 \rangle$
$\langle crit_1, wait_2, y = 0 \rangle$

TS(PG1|||PG2)



$\langle noncrit_1, noncrit_2, y = 1\rangle$
$\langle wait_1, noncrit_2, y = 1\rangle$
$\langle noncrit_1, crit_2, y = 0\rangle$
$\langle wait_1, crit_2, y = 0\rangle$

$\langle noncrit_1, wait_2, y = 1\rangle$
$\langle wait_1, wait_2, y = 1\rangle$
$\langle crit_1, noncrit_2, y = 0\rangle$
$\langle crit_1, wait_2, y = 0\rangle$

Peterson's mutual exclusion algorithm

- In the semaphore approach the choice of which process enters its critical section is made non-deterministically
  - That is, it is up to the implementer of the model to determine how the next process to enter its critical section is selected
- Peterson's algorithm makes an explicit choice
- Uses variables b_1, b_2, and x
  - b_1:Boolean - true if P1 is waiting to enter its critical section or is in its critical section (i.e., $b_1 = wait_1$ or $crit_1$)
  - b_2:Boolean - true if P2 is waiting to enter its critical section or is in its critical section
  - x:{1,2} - if x = 1 then P1 can enter its critical section; else (x = 2) P2 can enter its critical section

P_1 **loop forever**

$\vdots$ (* noncritical actions *)

$\langle b_1 := \text{true}; x := 2 \rangle$; (* request *)

wait until $(x = 1 \ \vee \ \neg b_2)$

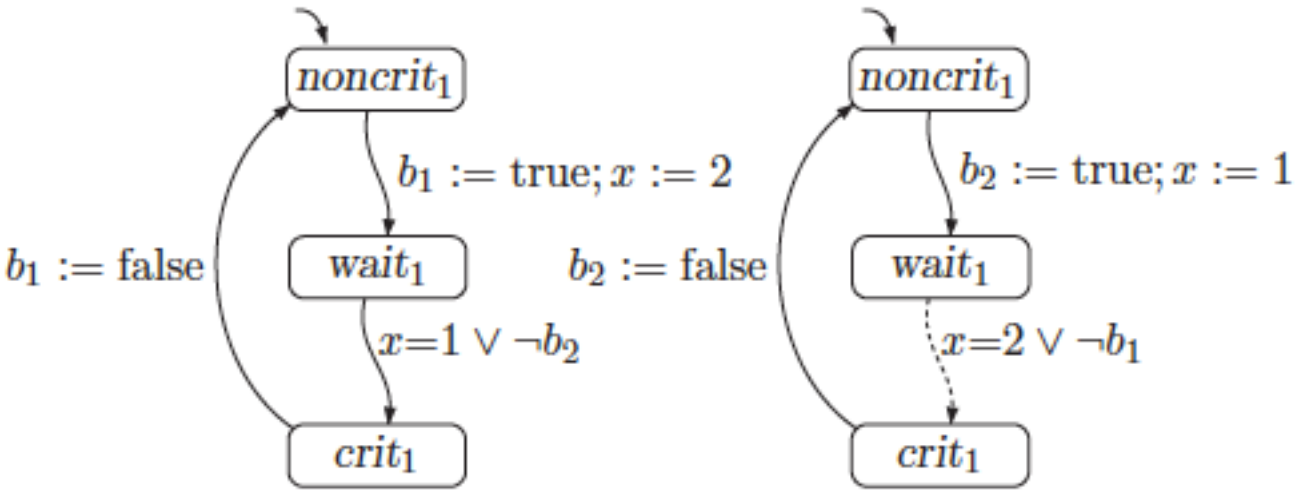
do critical section **od**

$b_1 := \text{false}$ (* release *)

$\vdots$ (* noncritical actions *)

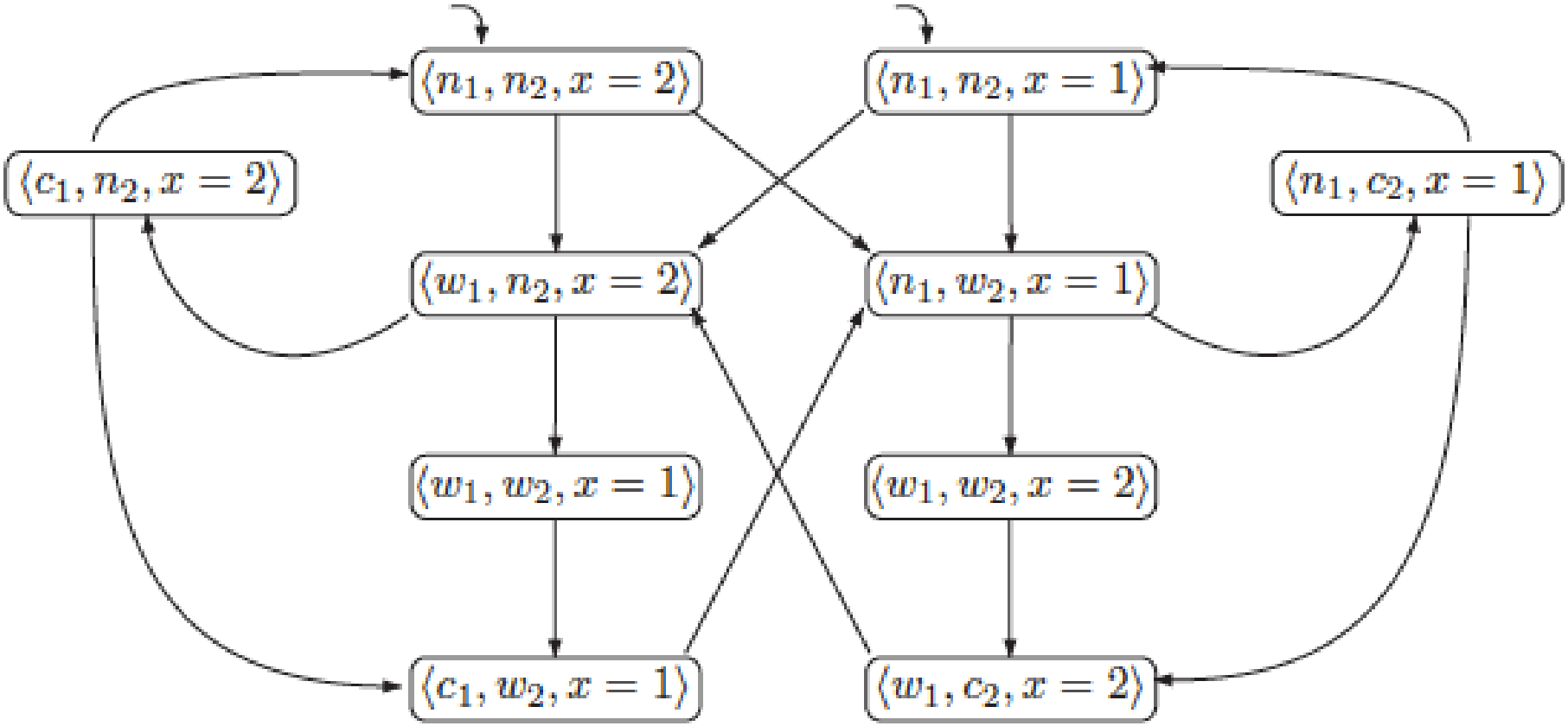
end loop

PG_1 : PG_2 :



Transition System

As an exercise draw the **interleaved program graph** used to produce the transition system shown below.
Notational shortcuts:
n1, n2: noncrit1, noncrit2; w1, w2: wait1, wait2; c1, c2: critical1, critical2



Atomicity

- The assignment group (b_i:=true; x:=i), where i = 1 or 2, are **atomic**, i.e., together they are treated as a single action; the individual assignments cannot be interleaved with other actions
- This is not essential for Petersen's algorithm to work
  - Mutual exclusion can also be ensured when the processes perform these actions in the given order
  - Mutual exclusion is NOT guaranteed if the operations are performed in reverse order, i.e., (x:=i,; b_i:=true)

Example of violation of mutual exclusion

$$
\begin{array}{rrlll}
\langle noncrit_1, & noncrit_2, & x = 1, & b_1 = \text{false}, & b_2 = \text{false}\rangle \\
\langle noncrit_1, & req_2, & x = 1, & b_1 = \text{false}, & b_2 = \text{false}\rangle \\
\langle req_1, & req_2, & x = 2, & b_1 = \text{false}, & b_2 = \text{false}\rangle \\
\langle wait_1, & req_2, & x = 2, & b_1 = \text{true}, & b_2 = \text{false}\rangle \\
\langle crit_1, & req_2, & x = 2, & b_1 = \text{true}, & b_2 = \text{false}\rangle \\
\langle crit_1, & wait_2, & x = 2, & b_1 = \text{true}, & b_2 = \text{true}\rangle \\
\langle crit_1, & crit_2, & x = 2, & b_1 = \text{true}, & b_2 = \text{true}\rangle
\end{array}
$$

Handshaking

Synchronous interactions

- Processes can also interact through a set of synchronizing actions, H, called **handshake actions**
- Processes interact only if they all can perform the same handshake action at the same time
  - i.e., the models must “shake hands” for the interaction to take place
- These actions may involve the transfer of data
  - This transfer will be ignored in the models we consider, i.e., we are interested only in the occurrence of the handshake and not in the data that is exchanged

Handshaking (Synchronous Message Passing)

Let $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP_i, L_i)$, i=1, 2 be transition systems and $H \subseteq Act_1 \cap Act_2$ with $\tau \notin H$. The transition system $TS_1 \parallel_H TS_2$ is defined as follows:

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

where $L(\langle s_1, s_2 \rangle) \;=\; L_1(s_1) \cup L_2(s_2)$

Notation: $TS_1 \parallel TS_2$ abbreviates $TS_1 \parallel_H TS_2$ for $H = Act_1 \cap Act_2$.

- 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 $\alpha \in H$:

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

Handshaking forms

$$TS_1 \parallel_{\varnothing} TS_2 \quad = \quad TS_1 \mid\mid\mid TS_2.$$

Empty set of handshake actions reduces to interleaving

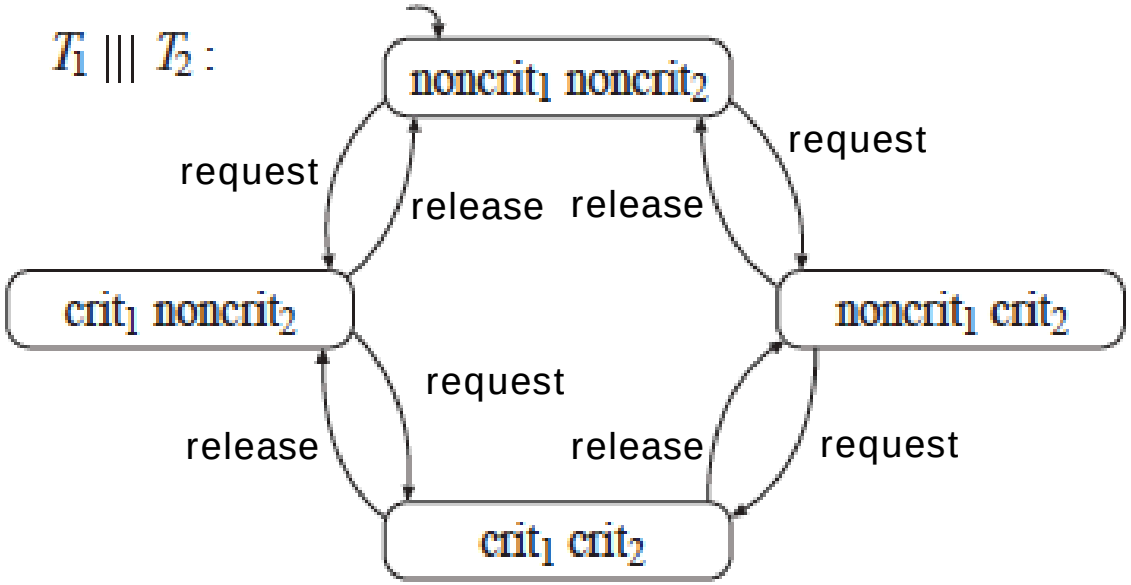
$$TS \;=\; TS_1 \parallel_H TS_2 \parallel_H \ldots \parallel_H TS_n$$

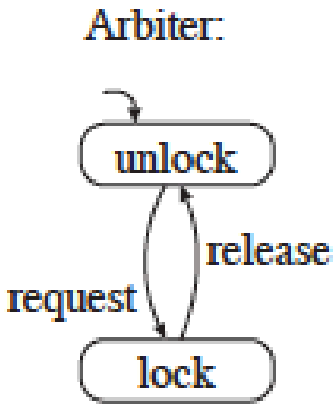
Models broadcasting communication

Mutual exclusion using an Arbiter process

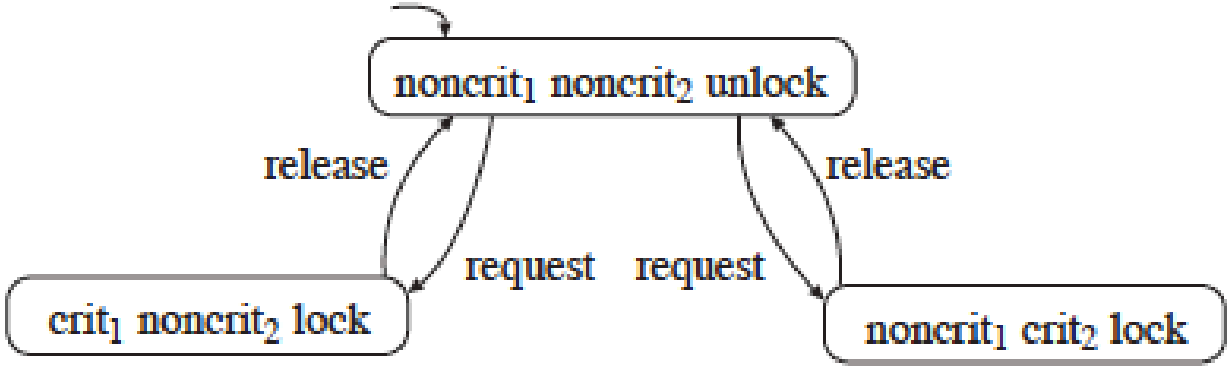
- Model the semaphore as a separate process, called an Arbiter
- Example: TS1 and TS2 are the transition systems of the parallel processes and Arbiter is the semaphore process

$$TS_{Arb} = (TS_1 \;|||\; TS_2) \parallel Arbiter$$



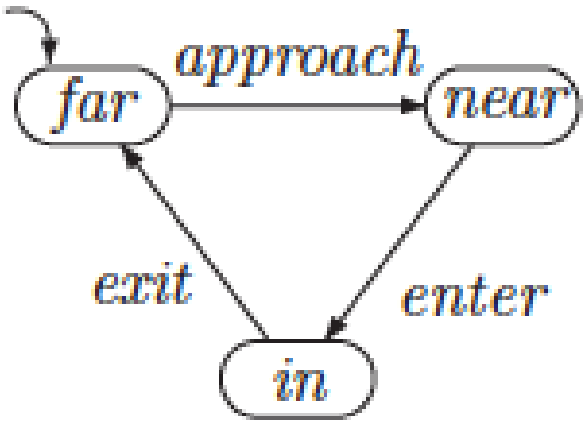


$(T_1 \,|||\, T_2) \,\|\, \text{Arbiter}$:

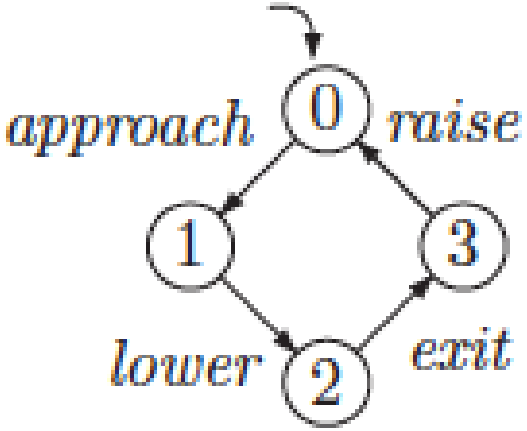


Railroad crossing example

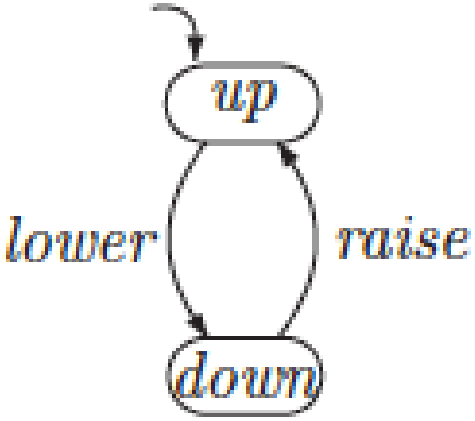
- Three processes in the system: Train, Gate, Controller.
- When the Controller receives a signal that a train is approaching it closes the gate
- The gate is opened only after the train has sent a signal to the Controller indicating it has crossed the road.



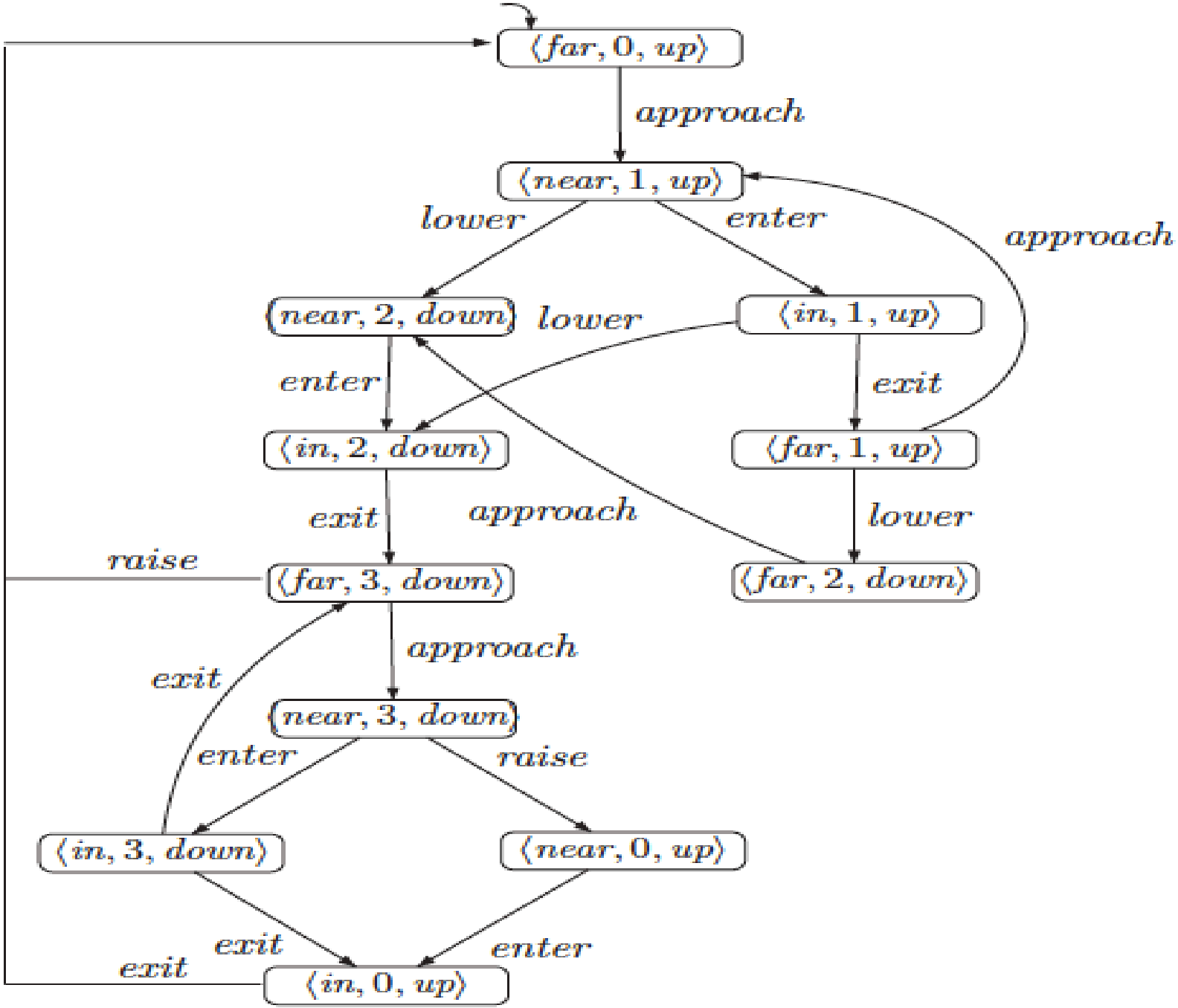
Train



Controller



Gate


⟨far, 0, up⟩
approach
⟨near, 1, up⟩
lower
enter
approach
⟨near, 2, down⟩
lower
⟨in, 1, up⟩
enter
exit
⟨in, 2, down⟩
⟨far, 1, up⟩
exit
approach
lower
raise
⟨far, 3, down⟩
⟨far, 2, down⟩
approach
exit
⟨near, 3, down⟩
enter
raise
⟨in, 3, down⟩
⟨near, 0, up⟩
exit
enter
exit
⟨in, 0, up⟩

Channel Systems

Asynchronous message passing

- Processes interact by passing information to each other via **channels** of finite or infinite capacity
  - A channel is like a buffer
- System thus consists of processes and channels
- If channel capacity > 0 the processes do not need to wait for a response from receiver when sending a message
- If channel capacity is 0 then this form of interaction reduces to handshaking
- Each channel can accept messages of a specified type only

Communication actions

Processes can perform the following communication actions:

$c!v$	transmit the value v along channel c,
$c?x$	receive a message via channel c and assign it to variable x.

Formal definition

A program graph over $(Var, Chan)$ is a tuple

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

$$\hookrightarrow \subseteq Loc \times (Cond(Var) \times (Act \cup Comm) \times Loc.$$

A *channel system* CS over $(Var, Chan)$ consists of program graphs PG_i over $(Var_i, Chan)$ (for $1 \leqslant i \leqslant n$) with $Var = \bigcup_{1 \leqslant i \leqslant n} Var_i$. We denote

$$CS = [PG_1 \mid \ldots \mid PG_n] .$$

The transition relation $\hookrightarrow$ of a program graph over $(Var, Chan)$ consists of two types of conditional transitions. As before, conditional transitions $\ell \overset{g:\alpha}{\hookrightarrow} \ell'$ are labeled with guards and actions. These conditional transitions can happen whenever the guard holds. Alternatively, conditional transitions may be labeled with communication actions. This yields conditional transitions of type $\ell \overset{g:c!v}{\hookrightarrow} \ell'$ (for sending v along c) and $\ell \overset{g:c?x}{\hookrightarrow} \ell'$ (for receiving a message along c).

Enabling communication actions

- *Handshaking.* If $cap(c) = 0$, then process P_i can transmit a value v over channel c by performing

$$\ell_i \overset{c!v}{\hookrightarrow} \ell_i'$$

only if another process P_j, say, "offers" a complementary receive action, i.e., can perform

$$\ell_j \overset{c?x}{\hookrightarrow} \ell_j'.$$

P_i and P_j should thus be able to perform $c!v$ (in P_i) and $c?x$ (in P_j) simultaneously.

- *Asynchronous message passing.* If $cap(c) > 0$, then process P_i can perform the conditional transition

$$\ell_i \overset{c!v}{\hookrightarrow} \ell_i'$$

if and only if channel c is not full, i.e., if less than $cap(c)$ messages are stored in c. In this case, v is stored at the rear of the buffer c. Channels are thus considered as first-in, first-out buffers. Accordingly, P_j may perform

$$\ell_j \overset{c?x}{\hookrightarrow} \ell_j'$$

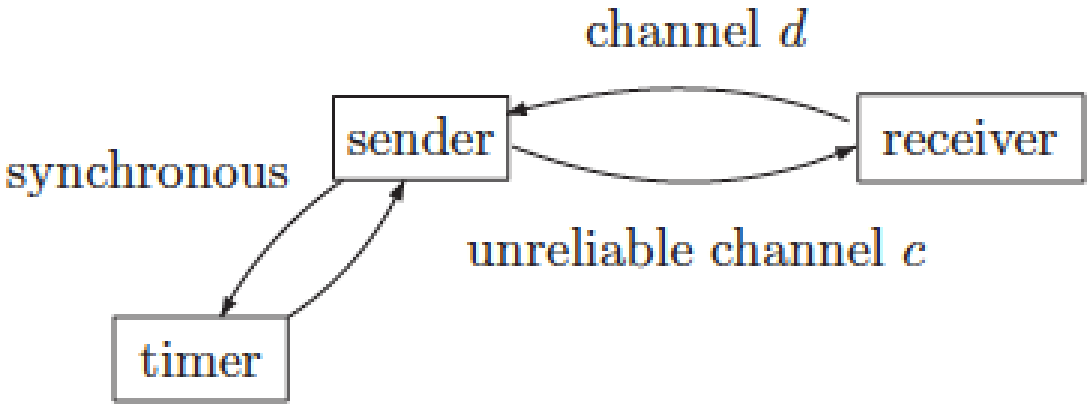
if and only if the buffer of c is not empty. In this case, the first element v of the buffer is extracted and assigned to x (in an atomic manner). This is summarized in Table 2.1.

	executable if ...	effect
$c!v$	c is not "full"	$Enqueue(c, v)$
$c?x$	c is not empty	$\langle x := Front(c)\,;\, Dequeue(c)\rangle$;

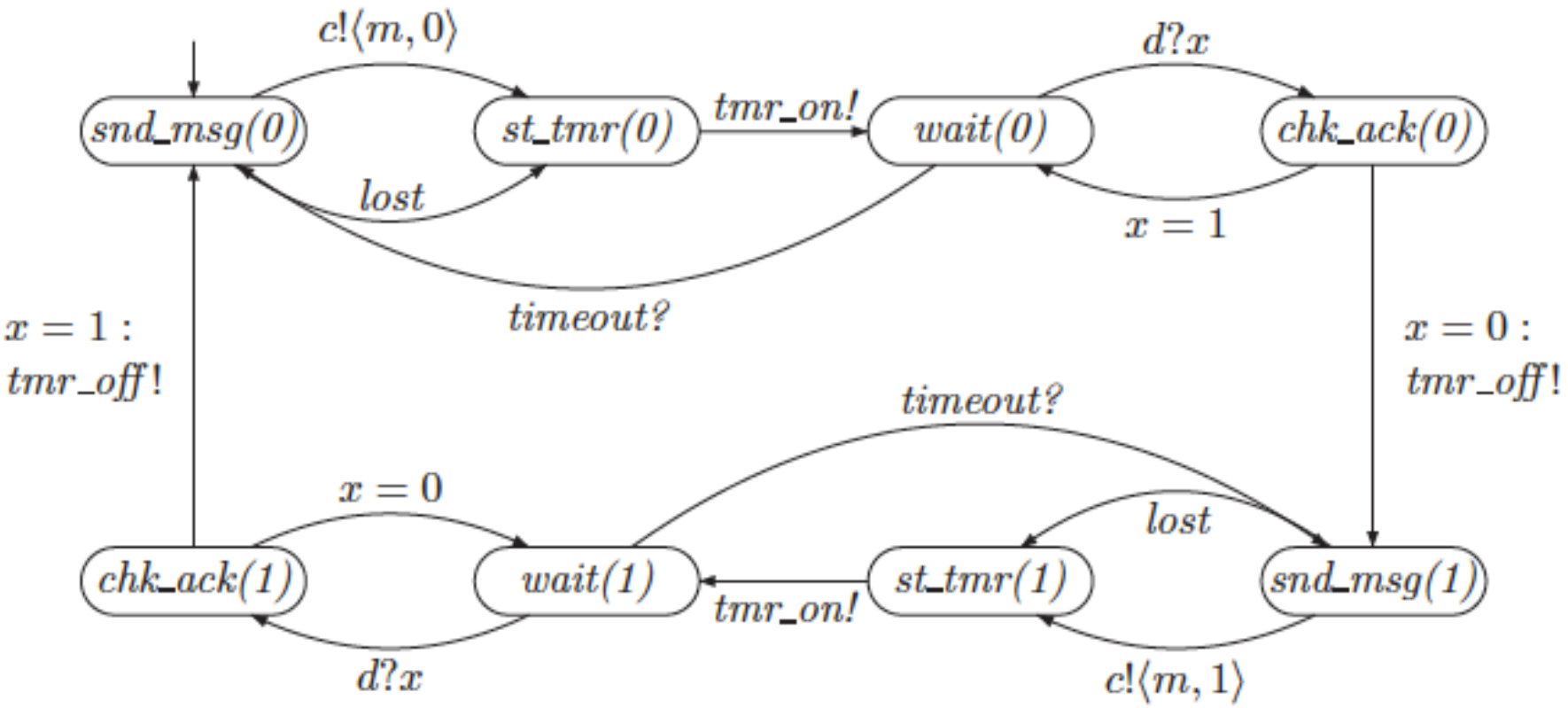
Table 2.1: Enabledness and effect of communication actions if $cap(c) > 0$.

Example: Alternating Bit Protocol

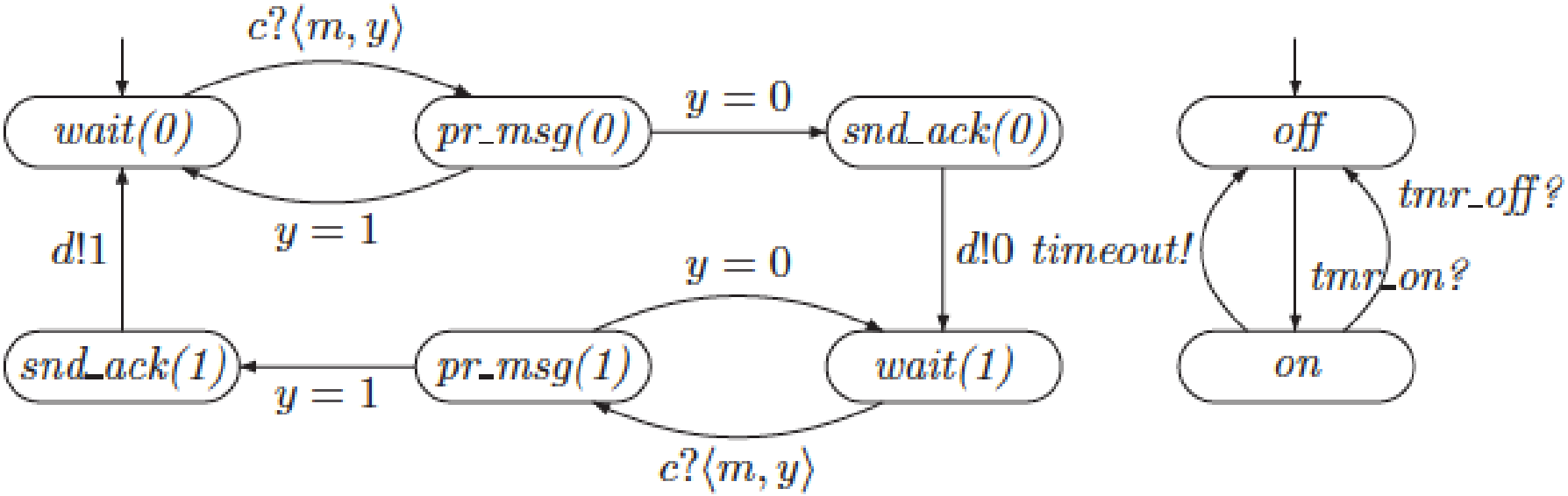
- System consists of two processes, S (sender), R (receiver) that communicate over two channels, c, d
- Channel c is unreliable ("lossy") in that it can lose messages during transmission; channel d is perfect
- The goal of the design is to ensure that data units (datums) transmitted by S are received by R
  - S sends data of the form <m,b>, where m is a message and b is a control bit that cab be either 0 or 1
  - S transmits a message and waits for R to acknowledge receipt; if an acknowledgement is not received within a given time S retransmits the message
  - If R receives the message then it sends an acknowledgement consisting of the control bit it received



PG for Sender



PG for Receiver, Timer



TS Semantics

Let $CS = [PG_1 \mid \ldots \mid PG_n]$ be a channel system over $(Chan, Var)$ with

$$PG_i = (Loc_i, Act_i, Effect_i, \hookrightarrow_i, Loc_{0,i}, g_{0,i}), \quad \text{for } 0 < i \leqslant n.$$

The transition system of CS, denoted $TS(CS)$, is the tuple $(S, Act, \rightarrow, I, AP, L)$ where:

- $S = (Loc_1 \times \ldots \times Loc_n) \times Eval(Var) \times Eval(Chan)$,
- $Act = \biguplus_{0<i\leqslant n} Act_i \uplus \{\tau\}$,
- $\rightarrow$
- $I = \left\{ \langle \ell_1, \ldots, \ell_n, \eta, \xi_0 \rangle \mid \forall 0 < i \leqslant n.\ (\ell_i \in Loc_{0,i} \wedge \eta \models g_{0,i}) \right\}$,
- $AP = \biguplus_{0<i\leqslant n} Loc_i \uplus Cond(Var)$,
- $L(\langle \ell_1, \ldots, \ell_n, \eta, \xi \rangle) = \{\ell_1, \ldots, \ell_n\} \cup \{g \in Cond(Var) \mid \eta \models g\}$.

- interleaving for $\alpha \in Act_i$:

$$\frac{\ell_i \overset{g:\alpha}{\hookrightarrow} \ell_i' \quad \wedge \quad \eta \models g}{\langle \ell_1, \ldots, \ell_i, \ldots, \ell_n, \eta, \xi \rangle \xrightarrow{\alpha} \langle \ell_1, \ldots, \ell_i', \ldots, \ell_n, \eta', \xi \rangle}$$

  where $\eta' = \mathit{Effect}(\alpha, \eta)$.

- asynchronous message passing for $c \in Chan$, $cap(c) > 0$:
  - receive a value along channel c and assign it to variable x:

$$\frac{\ell_i \overset{g:c?x}{\hookrightarrow} \ell_i' \ \wedge \ \eta \models g \ \wedge \ len(\xi(c)) = k > 0 \ \wedge \ \xi(c) \ = \ v_1 \ldots v_k}{\langle \ell_1, \ldots, \ell_i, \ldots, \ell_n, \eta, \xi \rangle \xrightarrow{\tau} \langle \ell_1, \ldots, \ell_i', \ldots, \ell_n, \eta', \xi' \rangle}$$

    where $\eta' = \eta[x := v_1]$ and $\xi' = \xi[c := v_2 \ldots v_k]$.
  - transmit value $v \in dom(c)$ over channel c:

$$\frac{\ell_i \overset{g:c!v}{\hookrightarrow} \ell_i' \ \wedge \ \eta \models g \ \wedge \ len(\xi(c)) = k < cap(c) \ \wedge \ \xi(c) \ = \ v_1 \ldots v_k}{\langle \ell_1, \ldots, \ell_i, \ldots, \ell_n, \eta, \xi \rangle \xrightarrow{\tau} \langle \ell_1, \ldots, \ell_i', \ldots, \ell_n, \eta, \xi' \rangle}$$

    where $\xi' = \xi[c := v_1 \, v_2 \ldots v_k \, v]$.

- synchronous message passing over $c \in Chan$, $cap(c) = 0$:

$$\frac{\ell_i \overset{g_1:c?x}{\hookrightarrow} \ell_i' \ \wedge \ \eta \models g_1 \ \wedge \ \eta \models g_2 \ \wedge \ \ell_j \overset{g_2:c!v}{\hookrightarrow} \ell_j' \ \wedge \ i \neq j}{\langle \ell_1, \ldots, \ell_i, \ldots, \ell_j, \ldots, \ell_n, \eta, \xi \rangle \xrightarrow{\tau} \langle \ell_1, \ldots, \ell_i', \ldots, \ell_j', \ldots, \ell_n, \eta', \xi \rangle}$$

  where $\eta' = \eta[x := v]$.

Coming up

- Why model and analyze concurrent systems?
- How can concurrent systems be modeled?
- How can concurrent systems be analyzed?