# Concurrency Theory

Rob van Glabbeek

University of Edinburgh

University of New South Wales, Sydney, Australia

Stanford University

July 2025

# 4 hour course on concurrency theory

- Modelling Distributed systems
  - Labelled Transition Systems (Monday)
  - Petri nets (Thursday, if time allows, on whiteboard)
  - Process algebra – CCSP (Monday)

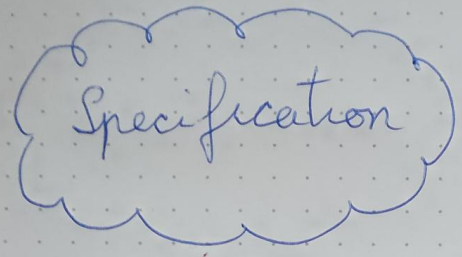
    (Denotational semantics on blackboard)
- When does an implementation meet a specification?
  - Semantic equivalences and preorders

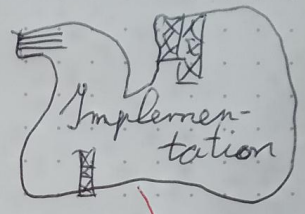
    (Monday, after the break, on whiteboard)
- Does a system have a property?
  - Safety and liveness properties (Monday & Tuesday)
  - Temporal logic (Tuesday)
  - Progress, Justness and Fairness assumptions (Tuesday & Thursday)
- Session types (Thursday)
- Mutual exclusion (Thursday)

30-hour course on this subject, with recorded lectures:
`https://cgi.cse.unsw.edu.au/~rvg/6752`

UNIVERSITÄT TRIER

www.uni-trier.de




what we want

What we made

Object
or
Expression
in
mathematical Model
or
Specification Language

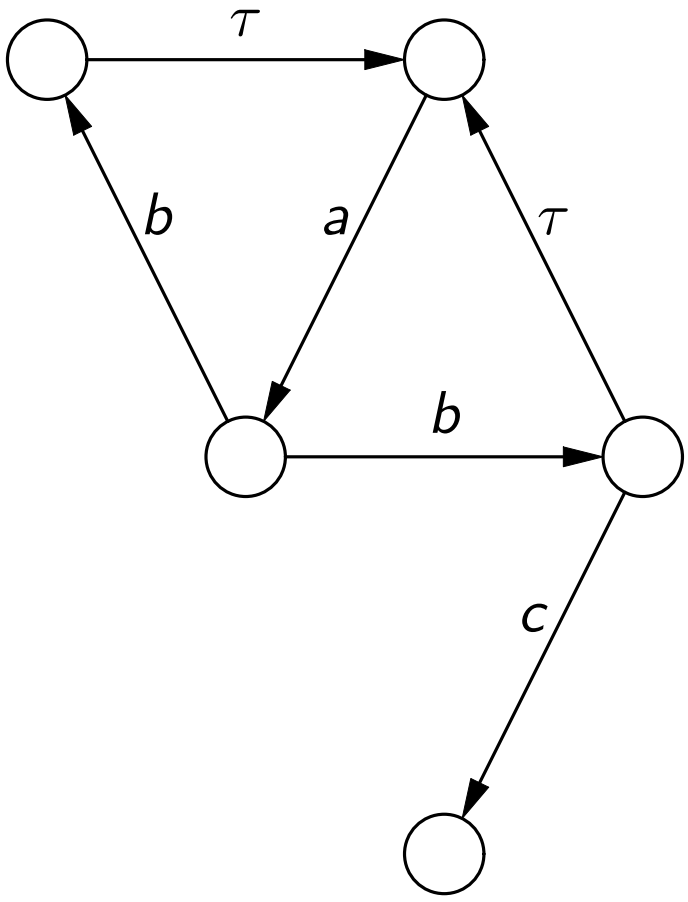
?
=

Object
or
Expression
in
math. Model
or
Language

Verification

Semantic Equivalence or preorder

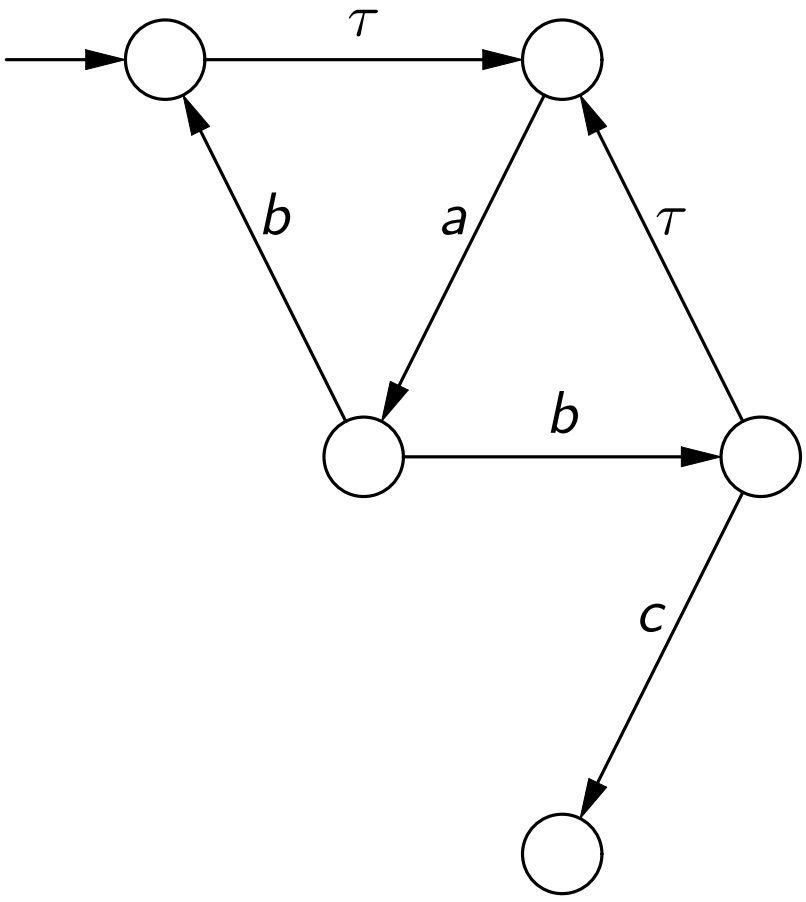
# Labelled Transition Systems




A *labelled transition system*

| | |
|---|---|
| $\tau$ | internal action |
| $a$ | insert coin in slot |
| $b$ | dispense pretzel |
| $c$ | close down machine |

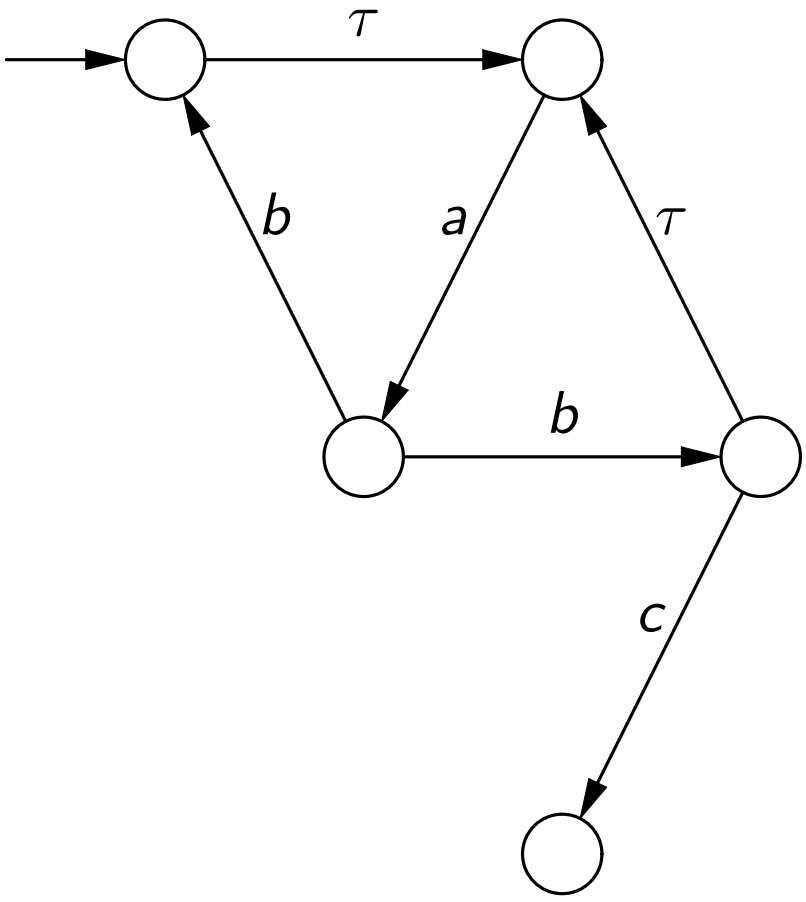
# Labelled Transition Systems




A *process graph*

| | |
|---|---|
| $\tau$ | internal action |
| $a$ | insert coin in slot |
| $b$ | dispense pretzel |
| $c$ | close down machine |

# Labelled Transition Systems




A *process graph*

| | |
|---|---|
| $\tau$ | internal action |
| $a$ | insert coin in slot |
| $b$ | dispense pretzel |
| $c$ | close down machine |

(completed) traces:
$(ab)^\infty \cup ab(ab)^*c$.

# Labelled Transition Systems

Let $Act$ be a set of *actions*.

A *Labelled Transition System* over $Act$ is tuple $(S, \rightarrow)$ with

- $S$ a set (of *states*), and
- $\rightarrow \subseteq S \times Act \times S$, the *transition relation*.

# Process graphs

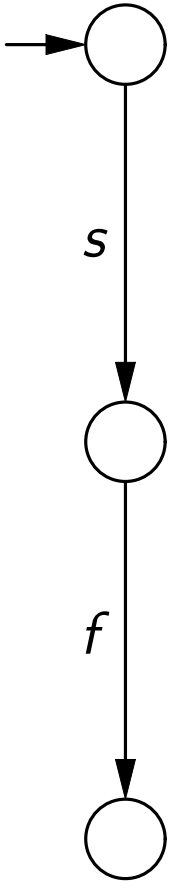
Let $Act$ be a set of *actions*.

A *Process graph* over $Act$ is tuple $(S, \rightarrow, I)$ with

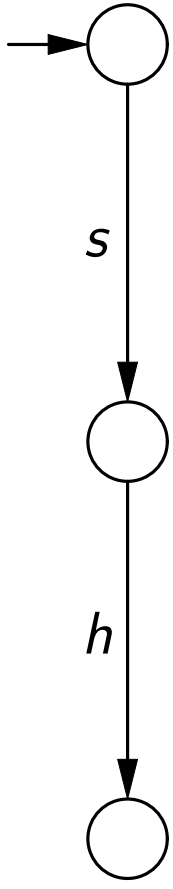
- $S$ a set (of *states*)
- $\rightarrow \subseteq S \times Act \times S$, the *transition relation*, and
- $I \in S$ the initial state.

# Relay Race

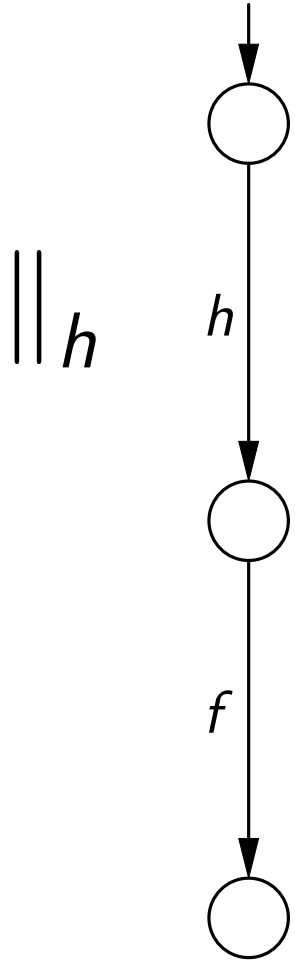
*Specification*




$s = start$
$h = handover$
$f = finish$

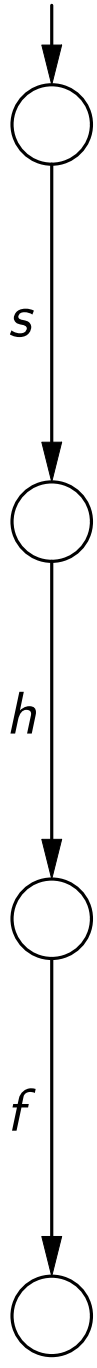



*Implementatation*



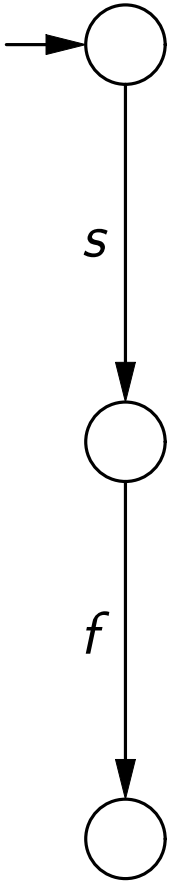

$\|_h$

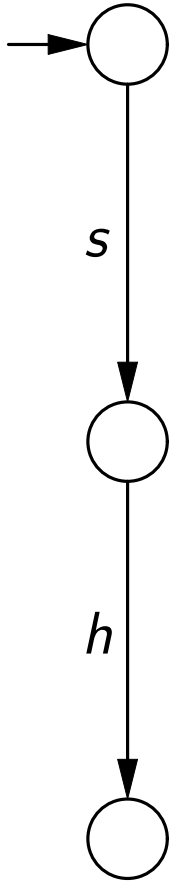
$=$



# Relay Race

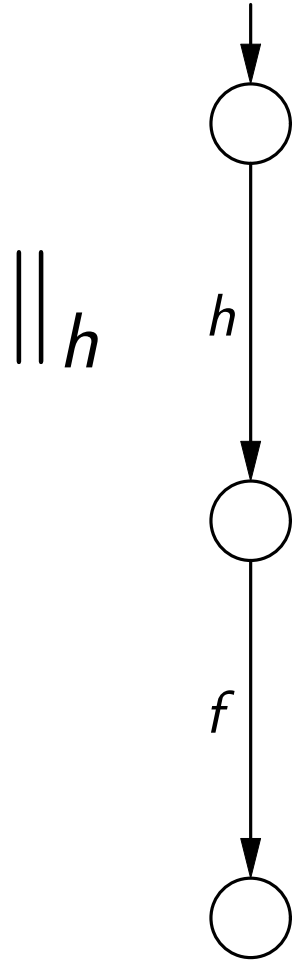
*Specification*



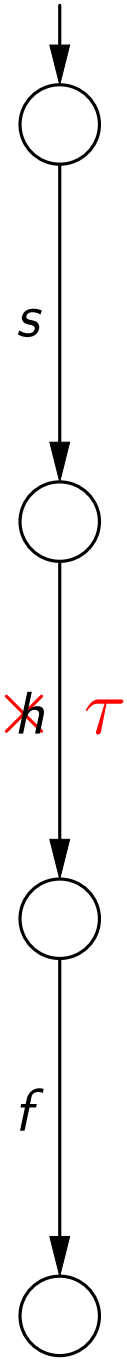

$s = start$
$h = handover$
$f = finish$

*Implementatation*




$\|_h$

$=$




Abstraction

# The process algebra CCSP

$$P, Q ::= \; 0 \;\Big|\; \alpha.P \;\Big|\; P + Q \;\Big|\; P \parallel_S Q \;\Big|\; \tau_I(P) \;\Big|\; \text{Relabel} \;\Big|\; \text{Recursion}$$

with $\alpha \in Act := A \uplus \{\tau\}$, $S, I \subseteq A$.

# The process algebra CCSP

$$P, Q ::= \; 0 \;\Big|\; \alpha.P \;\Big|\; P + Q \;\Big|\; P \parallel_S Q \;\Big|\; \tau_I(P) \;\Big|\; \text{Relabel} \;\Big|\; \text{Recursion}$$

with $\alpha \in Act := A \uplus \{\tau\}$, $S, I \subseteq A$.

*s.h.*0

# The process algebra CCSP

$$P, Q ::= \; 0 \;\Big|\; \alpha.P \;\Big|\; P + Q \;\Big|\; P \parallel_S Q \;\Big|\; \tau_I(P) \;\Big|\; \text{Relabel} \;\Big|\; \text{Recursion}$$

with $\alpha \in Act := A \uplus \{\tau\}$, $S, I \subseteq A$.

$s.h.0$

$coin.(chocolate.0 + pretzel.0)$

$VM := coin.(chocolate.VM + pretzel.VM)$

# The process algebra CCSP

$$P, Q ::= \; 0 \;\Big|\; \alpha.P \;\Big|\; P + Q \;\Big|\; P \parallel_S Q \;\Big|\; \tau_I(P) \;\Big|\; \text{Relabel} \;\Big|\; \text{Recursion}$$

with $\alpha \in Act := A \uplus \{\tau\}$, $S, I \subseteq A$.

$s.h.0$

$coin.(chocolate.0 + pretzel.0)$

$VM := coin.(chocolate.VM + pretzel.VM)$

$\tau_{\{h\}}(s.h \parallel_{\{h\}} h.f)$

# Operational semantics of CCSP

$$\alpha.P \xrightarrow{\alpha} P \qquad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'} \qquad \frac{P \xrightarrow{\alpha} P'}{\mathcal{R}(P) \xrightarrow{\beta} \mathcal{R}(P')} \begin{pmatrix} \alpha=\beta=\tau \vee \\ (\alpha,\beta) \in \mathcal{R} \end{pmatrix}$$

$$\frac{P \xrightarrow{\alpha} P'}{P \parallel_S Q \xrightarrow{\alpha} P' \parallel_S Q} (\alpha \notin S) \qquad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \parallel_S Q \xrightarrow{a} P' \parallel_S Q'} (a \in S) \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \parallel_S Q \xrightarrow{\alpha} P \parallel_S Q'} (\alpha \notin S)$$

$$\frac{P \xrightarrow{\alpha} P'}{\tau_I(P) \xrightarrow{\alpha} \tau_I(P')} (\alpha \notin I) \qquad \frac{P \xrightarrow{a} P'}{\tau_I(P) \xrightarrow{\tau} \tau_I(P')} (a \in I) \qquad \frac{P \xrightarrow{\alpha} P'}{X \xrightarrow{\alpha} P'} (X := P)$$

# Homework

Consider a process with the following set of completed traces: $\{bad, abd, adb, acd, adc\}$. Thus, in each run, the actions a, d and one out of c or b occur once each, subject to the following restrictions:

- b and c can never occur in the same trace.
- if c happens, it must be after a, and
- a happens before d.

Give a CCSP expression for this process that uses at most one $+$.

# Compositionality

$P_1 \| \cdots \| P_n$

$n$ parallel components of $k$ states each: $n^k$ states total.

*state explosion problem*

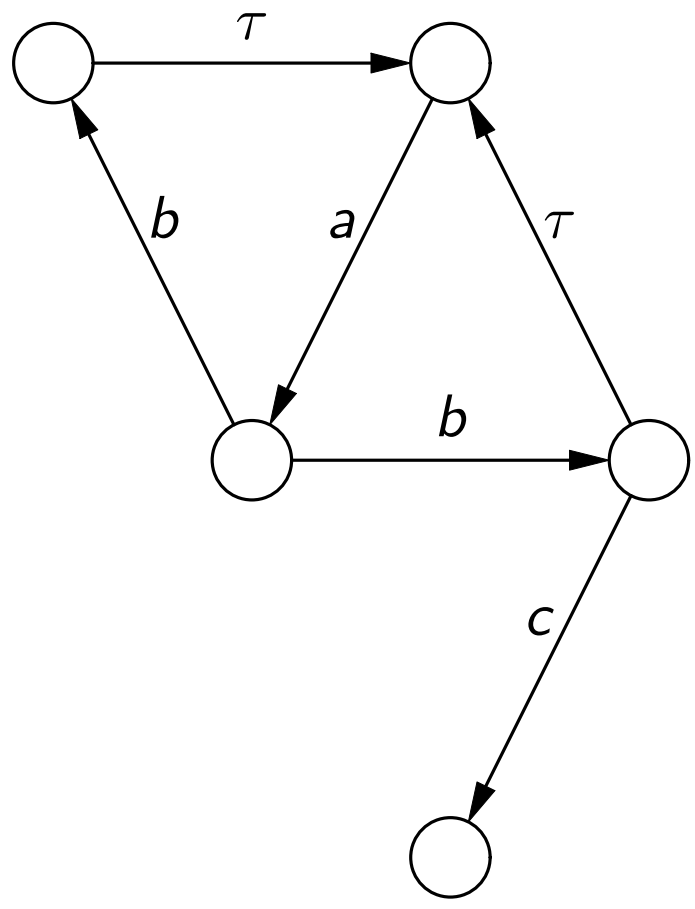
If we replace $P_i$ by an equivalent component $Q_i$, for $i = 1, \ldots, n$ then we want the new system $Q_1 \| \cdots \| Q_n$ to be equivalent to the old system $P_1 \| \cdots \| P_n$.

This is called *compositionality*.

Next to abstraction, this is the most important tool in combatting the state explosion problem.

# Safety and Liveness

| | |
|---|---|
| $\tau$ | internal action |
| $a$ | insert coin in slot |
| $b$ | dispense pretzel |
| $c$ | close down machine |




*Safety:*

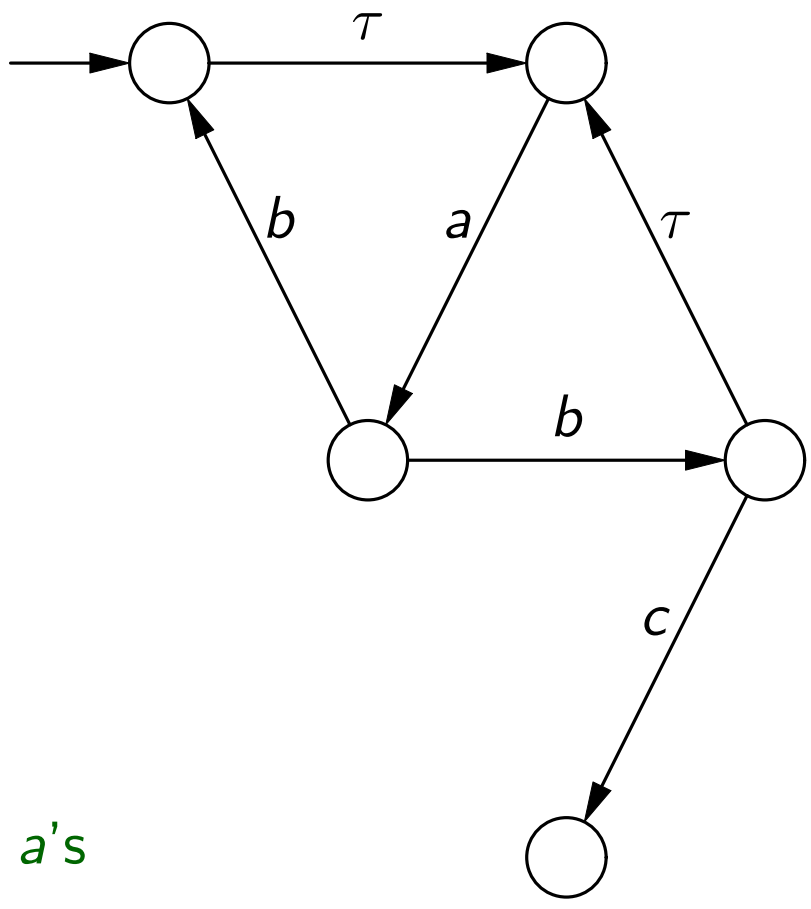
*nothing bad will ever happen*

*Liveness:*

*something good will ever happen eventually*

# Safety and Liveness

| | |
|---|---|
| $\tau$ | internal action |
| $a$ | insert coin in slot |
| $b$ | dispense pretzel |
| $c$ | close down machine |




*Safety:*

*nothing bad will ever happen*

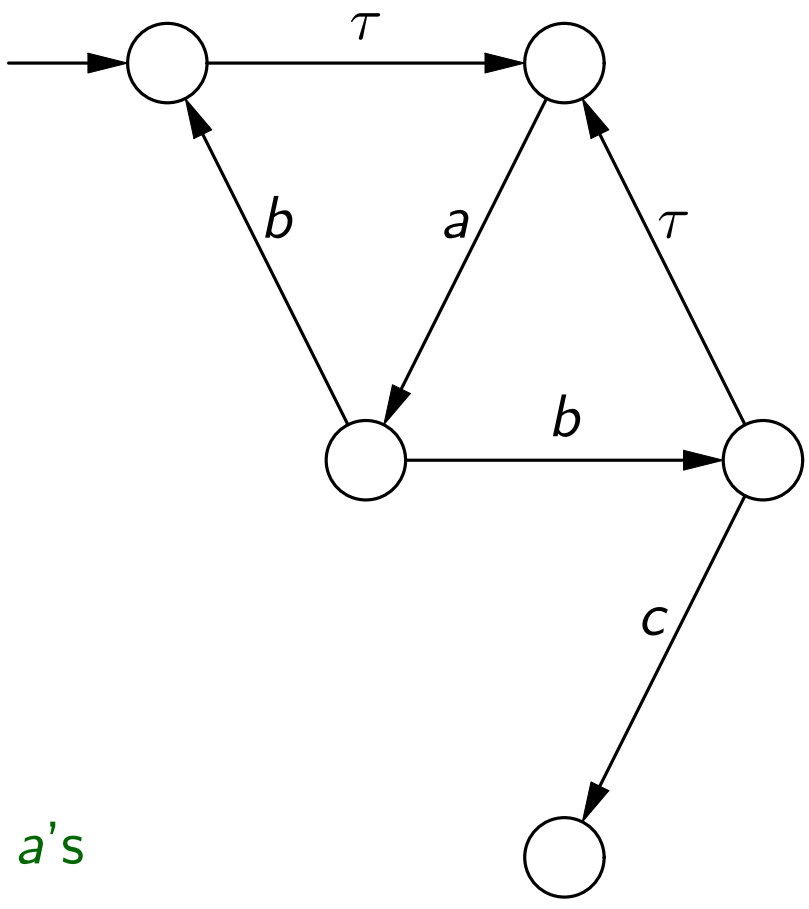
no run contains more $b$'s than $a$'s

*Liveness:*

*something good will ever happen eventually*

# Safety and Liveness

| | |
|---|---|
| $\tau$ | internal action |
| $a$ | insert coin in slot |
| $b$ | dispense pretzel |
| $c$ | close down machine |




*Safety:*

*nothing bad will ever happen*

no run contains more $b$'s than $a$'s

*Liveness:*

*something good will ever happen eventually*

each $a$ will eventually be followed by a $b$

$\mathsf{VM} = \mathsf{a}.\mathsf{VM}$

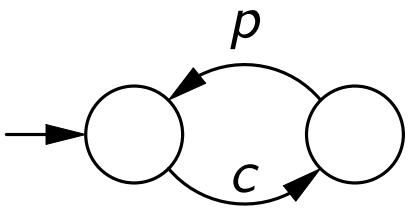
# Pretzels

Consider a vending machine that alternatingly accepts a coin ($c$) and produces a pretzel ($p$).

$$VM = c.p.VM$$

# Pretzels

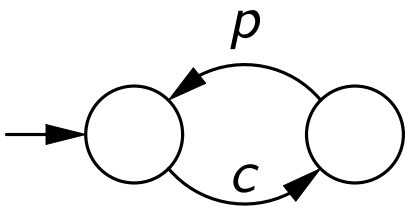
Consider a vending machine that alternatingly accepts a coin ($c$) and produces a pretzel ($p$).




$$VM = c.p.VM$$

# Pretzels

Consider a vending machine that alternatingly accepts a coin ($c$) and produces a pretzel ($p$).



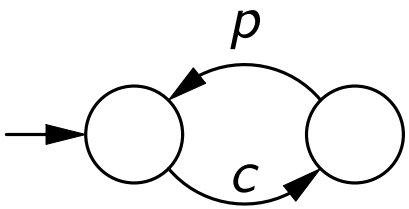

$$VM = c.p.VM$$

properties of systems.

*whenever a coin is inserted, eventually a pretzel is produced.*

# Pretzels

Consider a vending machine that alternatingly accepts a coin ($c$) and produces a pretzel ($p$).



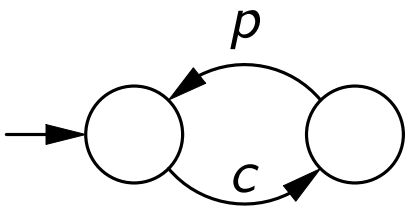

$$VM = c.p.VM$$

(desired) properties of systems.

*whenever a coin is inserted, eventually a pretzel is produced.*

# Pretzels

Consider a vending machine that alternatingly accepts a coin ($c$) and produces a pretzel ($p$).



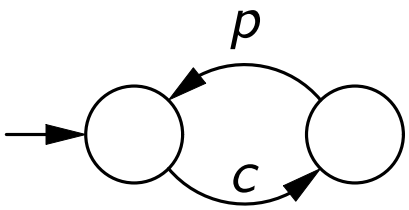

$$VM = c.p.VM$$

Temporal Logic is a formalism for specifying (desired) properties of systems.

*whenever a coin is inserted, eventually a pretzel is produced.*

# Pretzels

Consider a vending machine that alternatingly accepts a coin ($c$) and produces a pretzel ($p$).



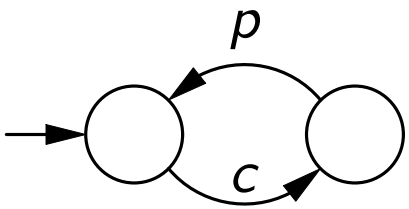

$$VM = c.p.VM$$

Temporal Logic is a formalism for specifying (desired) properties of systems. It can tell what should happen, and in which order, but not when.

*whenever a coin is inserted, eventually a pretzel is produced.*

# Pretzels

Consider a vending machine that alternatingly accepts a coin ($c$) and produces a pretzel ($p$).



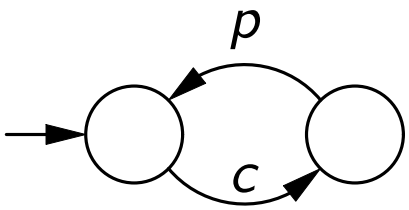

$$VM = c.p.VM$$

Linear-time Temporal Logic (LTL) is a formalism for specifying (desired) properties of systems. It can tell what should happen, and in which order, but not when. [Pnueli 1977]

*whenever a coin is inserted, eventually a pretzel is produced.*

# Pretzels

Consider a vending machine that alternatingly accepts a coin ($c$) and produces a pretzel ($p$).




$$VM = c.p.VM$$

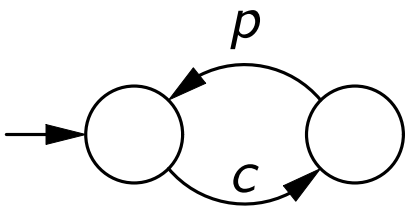
Linear-time Temporal Logic (LTL) is a formalism for specifying (desired) properties of systems. It can tell what should happen, and in which order, but not when. [Pnueli 1977]

In LTL we have $(\mathbf{c} \Rightarrow \mathbf{p})$. ← formula

*whenever a coin is inserted, eventually a pretzel is produced.*

# Pretzels

Consider a vending machine that alternatingly accepts a coin ($c$) and produces a pretzel ($p$).




$$VM = c.p.VM$$

Linear-time Temporal Logic (LTL) is a formalism for specifying (desired) properties of systems. It can tell what should happen, and in which order, but not when. [Pnueli 1977]

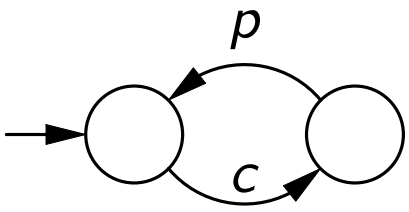
In LTL we have $(\mathbf{c} \Rightarrow \mathbf{Fp})$. ← a temporal formula

*whenever a coin is inserted, eventually a pretzel is produced.*

**F** (or $\diamond$) means eventually

# Pretzels

Consider a vending machine that alternatingly accepts a coin ($c$) and produces a pretzel ($p$).




$$VM = c.p.VM$$

Linear-time Temporal Logic (LTL) is a formalism for specifying (desired) properties of systems. It can tell what should happen, and in which order, but not when. [Pnueli 1977]

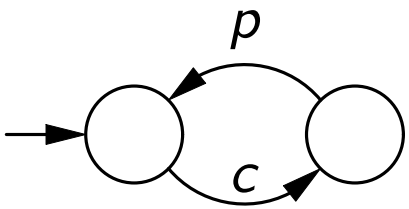
In LTL we have $\mathbf{G}(\mathbf{c} \Rightarrow \mathbf{F}\mathbf{p})$. ← a temporal formula

*whenever a coin is inserted, eventually a pretzel is produced.*

**F** (or $\diamond$) means eventually
**G** (or $\Box$) means globally, in each reachable state

# Pretzels

Consider a vending machine that alternatingly accepts a coin ($c$) and produces a pretzel ($p$).




$$VM = c.p.VM$$

Linear-time Temporal Logic (LTL) is a formalism for specifying (desired) properties of systems. It can tell what should happen, and in which order, but not when. [Pnueli 1977]

In LTL we have $VM \models \mathbf{G}(\mathbf{c} \Rightarrow \mathbf{F}\mathbf{p})$. $\leftarrow$ a temporal judgement

*whenever a coin is inserted, eventually a pretzel is produced.*

**F** (or $\diamond$) means eventually
**G** (or $\Box$) means globally, in each reachable state
$\models$ means satisfies.

# Kripke structures

Let $AP$ be a set of *atomic predicates*.

A *Kripke structure* over $AP$ is tuple $(S, \rightarrow, \models)$ with

- $S$ a set (of *states*),
- $\rightarrow \subseteq S \times S$, the *transition relation*, and
- $\models \subseteq S \times AP$.

$s \models p$ says that predicate $p \in AP$ *holds* in state $s \in S$.

# Kripke structures

Let $AP$ be a set of *atomic predicates*.

A *Kripke structure* over $AP$ is tuple $(S, \rightarrow, \models)$ with

- $S$ a set (of *states*),
- $\rightarrow \subseteq S \times S$, the *transition relation*, and
- $\models \subseteq S \times AP$.

$s \models p$ says that predicate $p \in AP$ *holds* in state $s \in S$.

A *path* in a Kripke structure is a nonempty finite or infinite sequence $s_0, s_1, \ldots$ of states that follows the transition relation.

# Kripke structures

Let $AP$ be a set of *atomic predicates*.

A *Kripke structure* over $AP$ is tuple $(S, \rightarrow, \models)$ with

- $S$ a set (of *states*),
- $\rightarrow \subseteq S \times S$, the *transition relation*, and
- $\models \subseteq S \times AP$.

$s \models p$ says that predicate $p \in AP$ *holds* in state $s \in S$.

A *path* in a Kripke structure is a nonempty finite or infinite sequence $s_0, s_1, \ldots$ of states that follows the transition relation.

If $\rho$ is a finite prefix of $\pi$, then $\pi \restriction \rho$ denotes the suffix of $\pi$ that remains after removing the prefix $\rho$, but not the last state of $\rho$.

# Kripke structures

Let $AP$ be a set of *atomic predicates*.

A *Kripke structure* over $AP$ is tuple $(S, \to, \models)$ with

- $S$ a set (of *states*),
- $\to \subseteq S \times S$, the *transition relation*, and
- $\models \subseteq S \times AP$.

$s \models p$ says that predicate $p \in AP$ *holds* in state $s \in S$.

A *path* in a Kripke structure is a nonempty finite or infinite sequence $s_0, s_1, \ldots$ of states that follows the transition relation.

If $\rho$ is a finite prefix of $\pi$, then $\pi \restriction \rho$ denotes the suffix of $\pi$ that remains after removing the prefix $\rho$, but not the last state of $\rho$.

The *length* of a path $\pi$, denoted $|\pi| \in \mathbb{N} \cup \{\infty\}$, is the number of transitions in $\pi$; for instance $|s_0 s_1 s_2 s_3| = 3$.

# Kripke structures

Let $AP$ be a set of *atomic predicates*.

A *Kripke structure* over $AP$ is tuple $(S, \to, \models)$ with

- $S$ a set (of *states*),
- $\to \subseteq S \times S$, the *transition relation*, and
- $\models \subseteq S \times AP$.

$s \models p$ says that predicate $p \in AP$ *holds* in state $s \in S$.

A *path* in a Kripke structure is a nonempty finite or infinite sequence $s_0, s_1, \ldots$ of states that follows the transition relation.

If $\rho$ is a finite prefix of $\pi$, then $\pi \restriction \rho$ denotes the suffix of $\pi$ that remains after removing the prefix $\rho$, but not the last state of $\rho$.
The *length* of a path $\pi$, denoted $|\pi| \in \mathbb{N} \cup \{\infty\}$, is the number of transitions in $\pi$; for instance $|s_0 s_1 s_2 s_3| = 3$.

A *completeness criterion* defines the *complete paths* as a subset of all paths. Complete paths model actual runs of the represented system.

# Kripke structures

Let $AP$ be a set of *atomic predicates*.

A *Kripke structure* over $AP$ is tuple $(S, \rightarrow, \models)$ with

- $S$ a set (of *states*),
- $\rightarrow \subseteq S \times S$, the *transition relation*, and
- $\models \subseteq S \times AP$.

$s \models p$ says that predicate $p \in AP$ *holds* in state $s \in S$.

A *path* in a Kripke structure is a nonempty finite or infinite sequence $s_0, s_1, \ldots$ of states that follows the transition relation.

If $\rho$ is a finite prefix of $\pi$, then $\pi \restriction \rho$ denotes the suffix of $\pi$ that remains after removing the prefix $\rho$, but not the last state of $\rho$.
The *length* of a path $\pi$, denoted $|\pi| \in \mathbb{N} \cup \{\infty\}$, is the number of transitions in $\pi$; for instance $|s_0 s_1 s_2 s_3| = 3$.

A *completeness criterion* defines the *complete paths* as a subset of all paths. Complete paths model actual runs of the represented system.

The default completeness criterion, called progress, classifies a path as complete iff it either is infinite or ends in a deadlock state.

# LTL

*Linear-time temporal logic* (LTL) is designed to formulate useful properties of systems. Its syntax is

$$\varphi, \psi ::= p \mid \neg\varphi \mid \varphi \wedge \psi \mid \mathbf{X}\varphi \mid \mathbf{F}\varphi \mid \mathbf{G}\varphi \mid \psi \mathbf{U} \varphi$$

with $p \in AP$ an atomic predicate. The propositional connectives $\Rightarrow$ and $\vee$ can be added as syntactic sugar. It is interpreted on paths.

# LTL

*Linear-time temporal logic* (LTL) is designed to formulate useful properties of systems. Its syntax is

$$\varphi, \psi ::= p \mid \neg\varphi \mid \varphi \wedge \psi \mid \mathbf{X}\varphi \mid \mathbf{F}\varphi \mid \mathbf{G}\varphi \mid \psi \mathbf{U} \varphi$$

with $p \in AP$ an atomic predicate. The propositional connectives $\Rightarrow$ and $\vee$ can be added as syntactic sugar. It is interpreted on paths.

$\pi \models \varphi$ says that the path $\pi$ *satisfies* the formula $\varphi$, or that $\varphi$ is *valid* on $\pi$.

# LTL

*Linear-time temporal logic* (LTL) is designed to formulate useful properties of systems. Its syntax is

$$\varphi, \psi ::= p \mid \neg\varphi \mid \varphi \wedge \psi \mid \mathbf{X}\varphi \mid \mathbf{F}\varphi \mid \mathbf{G}\varphi \mid \psi\mathbf{U}\varphi$$

with $p \in AP$ an atomic predicate. The propositional connectives $\Rightarrow$ and $\vee$ can be added as syntactic sugar. It is interpreted on paths.

$\pi \models \varphi$ says that the path $\pi$ *satisfies* the formula $\varphi$, or that $\varphi$ is *valid* on $\pi$. This is inductively defined by

- $\pi \models p$, with $p \in AP$, iff $s \models p$, where $s$ is the first state of $\pi$,
- $\pi \models \neg\varphi$ iff $\pi \not\models \varphi$,
- $\pi \models \varphi \wedge \psi$ iff $\pi \models \varphi$ and $\pi \models \psi$,
- $\pi \models \mathbf{X}\varphi$ iff $|\pi| > 0$ and $\pi_{+1} \models \varphi$,
- $\pi \models \mathbf{F}\varphi$ iff $\pi{\restriction}\rho \models \varphi$ for some finite prefix $\rho$ of $\pi$,
- $\pi \models \mathbf{G}\varphi$ iff $\pi{\restriction}\rho \models \varphi$ for each finite prefix $\rho$ of $\pi$, and
- $\pi \models \psi\mathbf{U}\varphi$ iff $\pi{\restriction}\rho \models \varphi$ for some finite prefix $\rho$ of $\pi$, and $\pi{\restriction}\zeta \models \psi$ for each proper prefix $\zeta$ of $\rho$.

# Interpreting LTL on states

Let $CC$ be a completeness criteria.

For $s$ a state in a Kripke structure, write
$P \models^{CC} \varphi$ iff $\pi \models \varphi$ for all complete paths starting in $s$.

# Interpreting LTL on states

Let $CC$ be a completeness criteria.

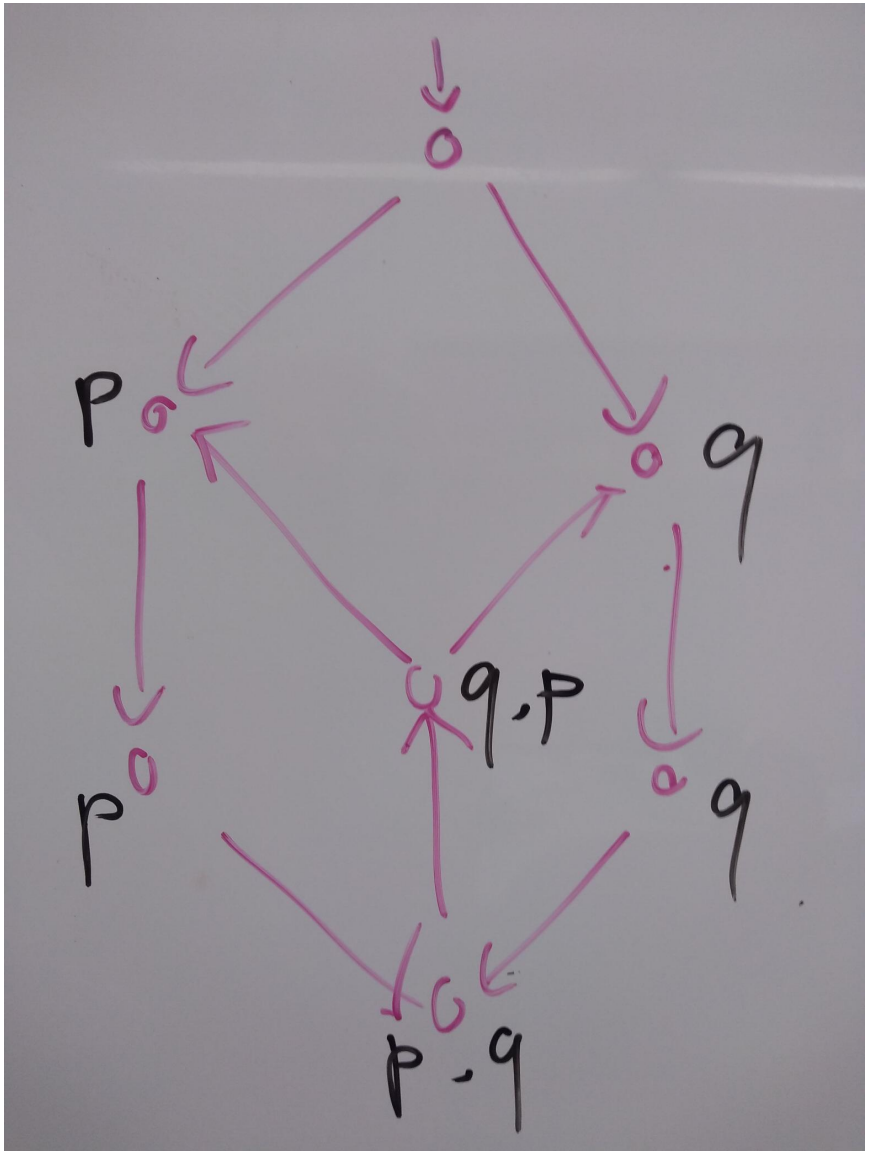
For $s$ a state in a Kripke structure, write
$P \models^{CC} \varphi$ iff $\pi \models \varphi$ for all complete paths starting in $s$.

We leave out the $CC$ and write $P \models \varphi$ when taking *progress* as $CC$.

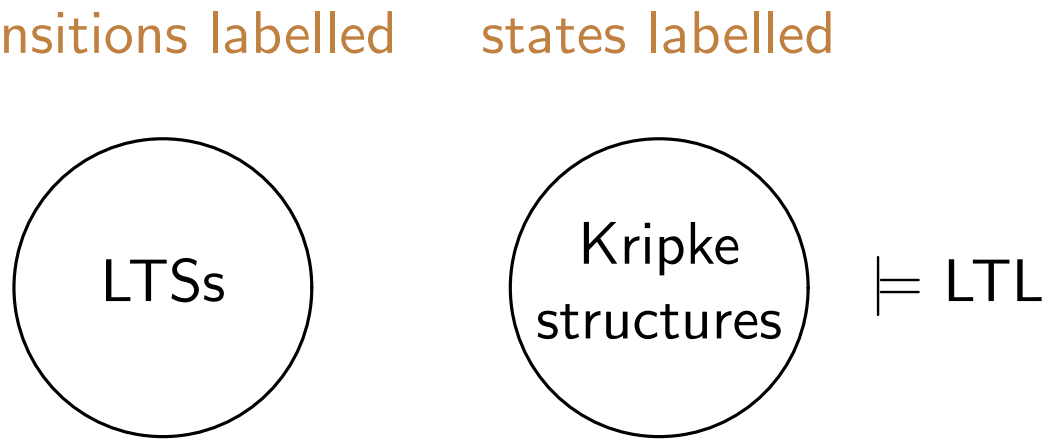
# Homework

Which of these LTL formulae hold in the initial state of this Kripke structure?

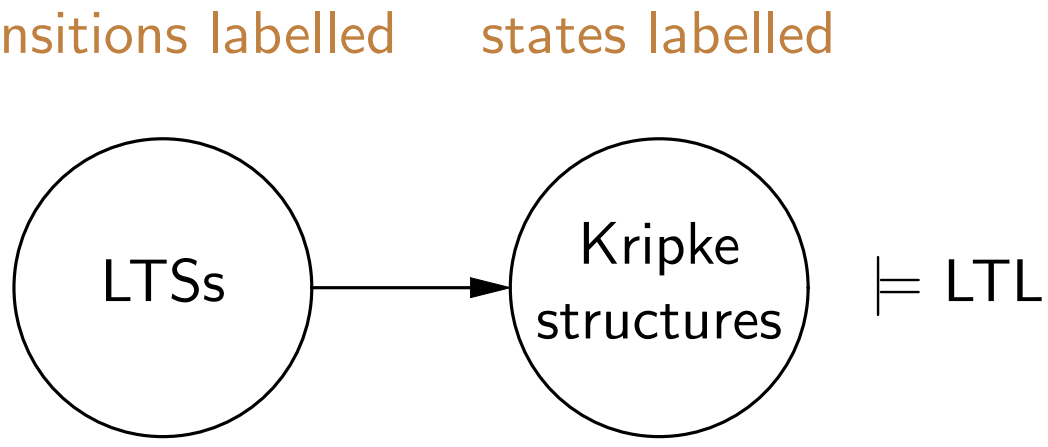
(a) $\mathbf{G}(p \vee q)$
(b) $\mathbf{FG}(p \vee q)$
(c) $\mathbf{F}(\mathbf{G}p \vee \mathbf{G}q)$
(d) $\mathbf{G}(q \rightarrow (q\mathbf{U}p))$
(e) $\mathbf{G}((p \wedge \neg q) \rightarrow \mathbf{X}q)$
(f) $\mathbf{G}((p \wedge q) \rightarrow \mathbf{X}(q\mathbf{U}p))$



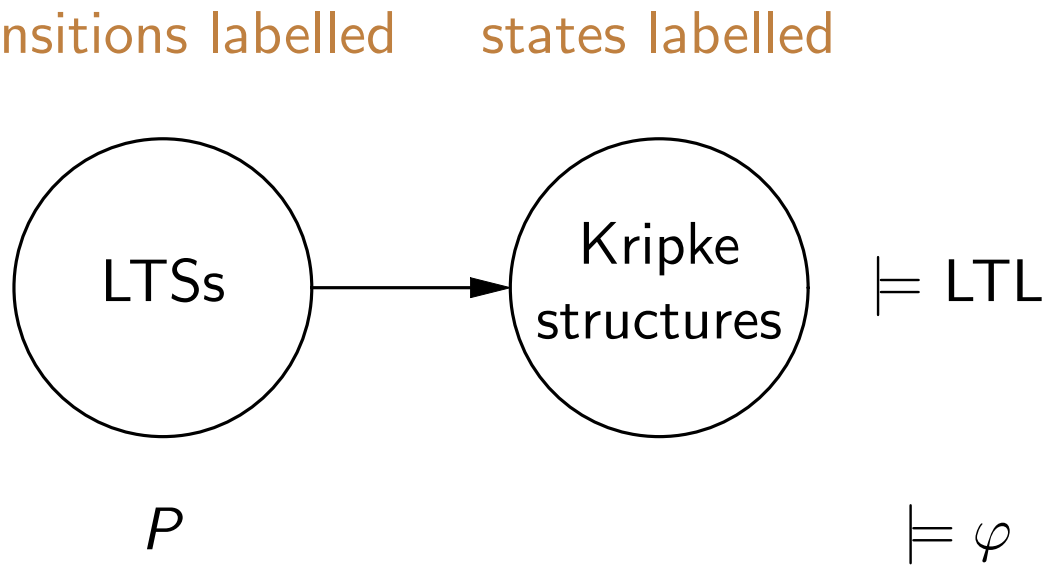
# De Nicola - Vaandrager translation



# De Nicola - Vaandrager translation

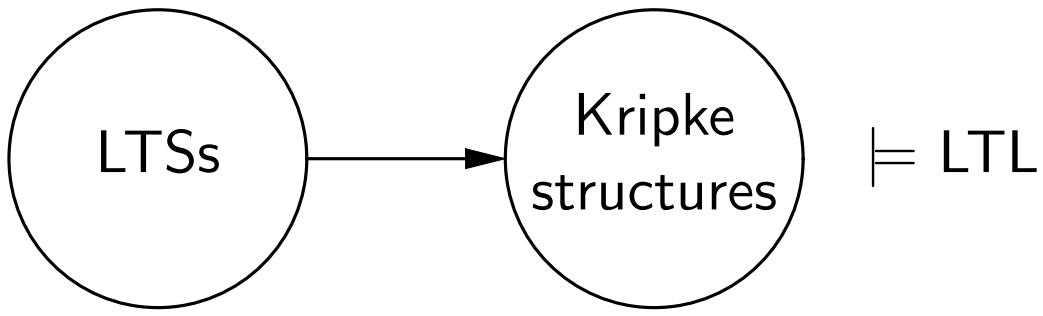


# De Nicola - Vaandrager translation



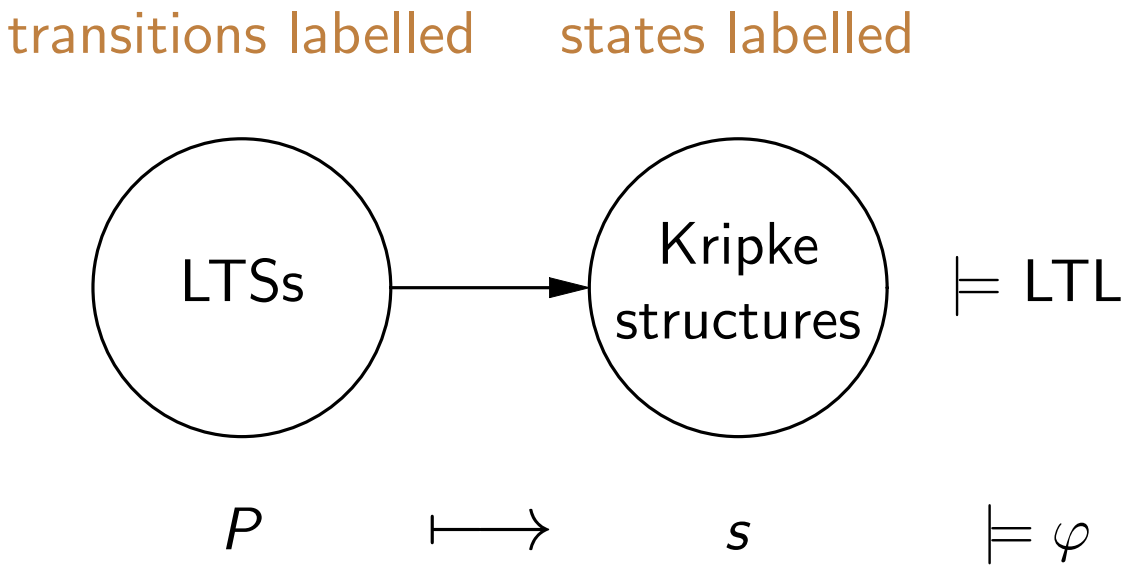
# De Nicola - Vaandrager translation

transitions labelled  states labelled



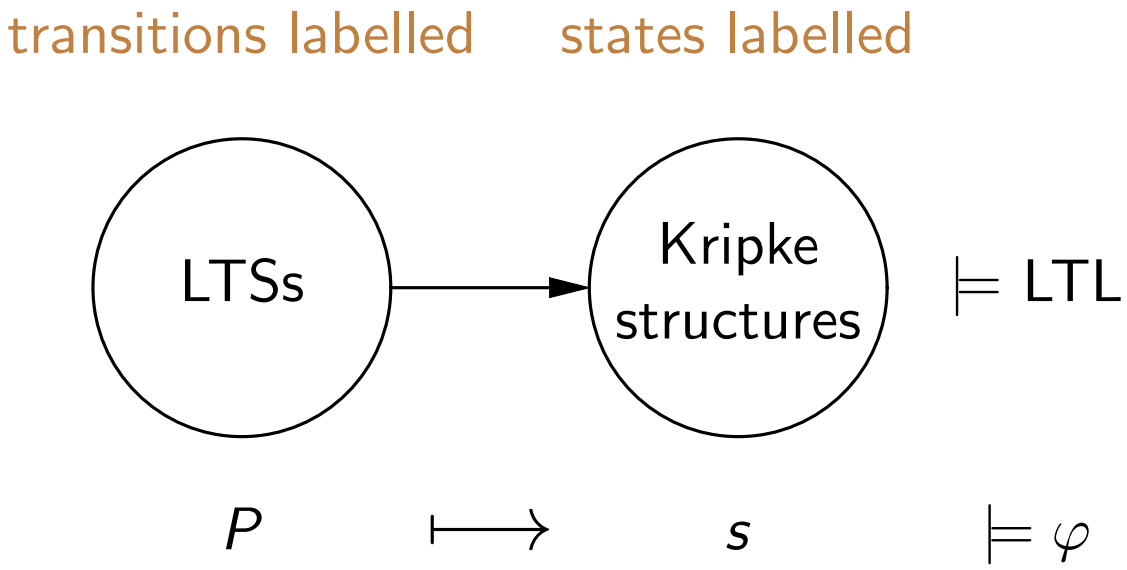

$P \quad \longmapsto \quad s \quad \models \varphi$

# De Nicola - Vaandrager translation




Create a new state halfway along any transition labelled by a visible action, and move the transition label to that state.

# De Nicola - Vaandrager translation




Create a new state halfway along any transition labelled by a visible action, and move the transition label to that state. $\tau$-transitions are not affected.

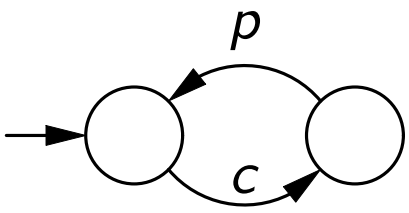
# Pretzels

Consider a vending machine that alternatingly accepts a coin ($c$) and produces a pretzel ($p$).

$$VM = c.p.VM$$

# Pretzels

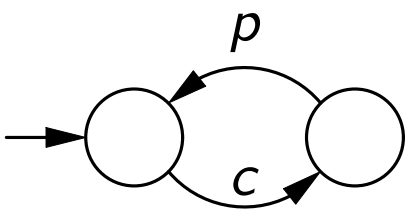
Consider a vending machine that alternatingly accepts a coin ($c$) and produces a pretzel ($p$).



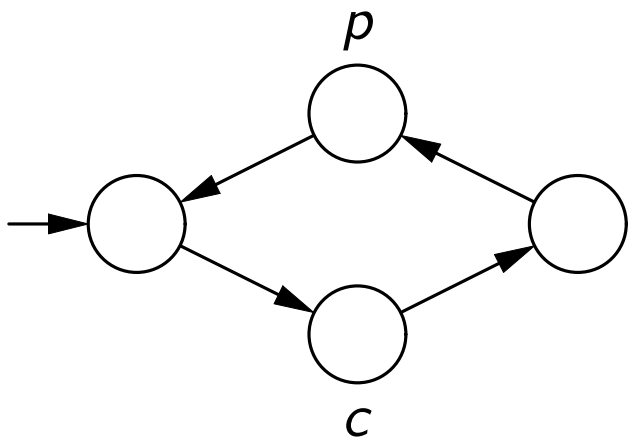

$$VM = c.p.VM$$

# Pretzels

Consider a vending machine that alternatingly accepts a coin ($c$) and produces a pretzel ($p$).

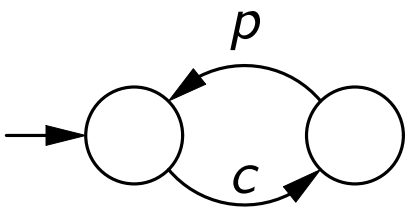



$$VM = c.p.VM$$

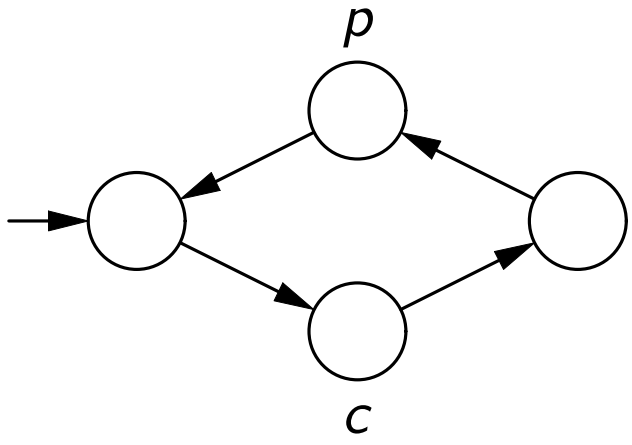


# Pretzels

Consider a vending machine that alternatingly accepts a coin ($c$) and produces a pretzel ($p$).




$$VM = c.p.VM$$

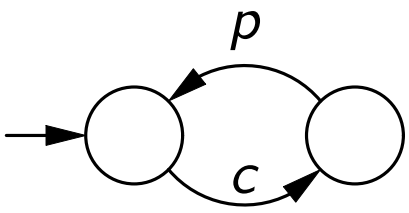



In LTL we have $VM \models \mathbf{G}(c \Rightarrow \mathbf{F}p)$.

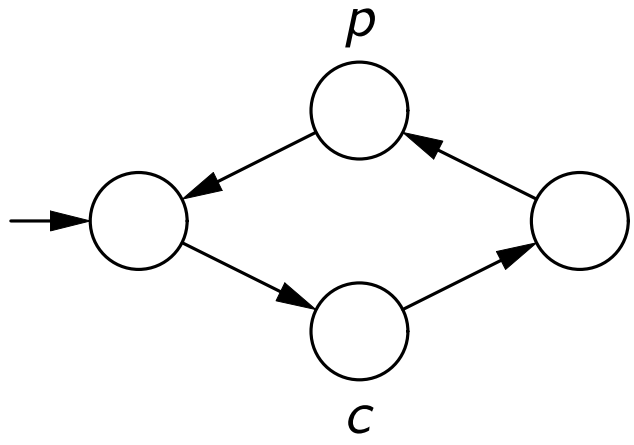
*whenever a coin is inserted, eventually a pretzel is produced.*

# Pretzels

Consider a vending machine that alternatingly accepts a coin ($c$) and produces a pretzel ($p$).




$VM = c.p.VM$




In LTL we have $VM \models \mathbf{G}(c \Rightarrow \mathbf{F}p)$.

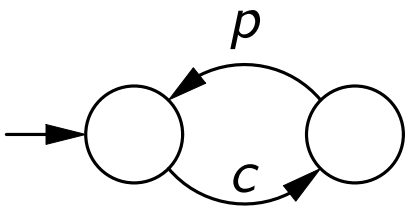
*whenever a coin is inserted, eventually a pretzel is produced.*

But we also have $VM \models \mathbf{G}(p \Rightarrow \mathbf{F}c)$.

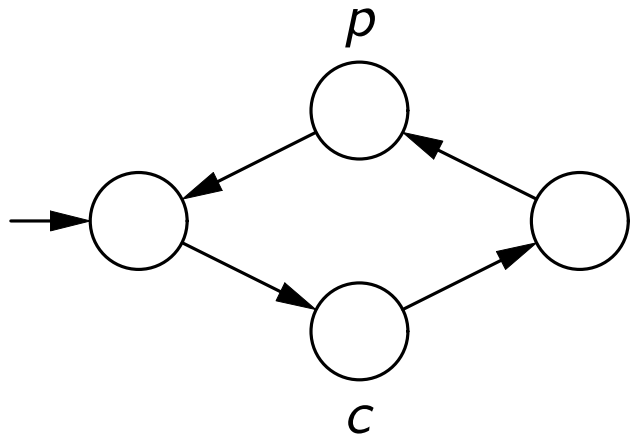
*whenever a pretzel is produced, eventually a new coin will be inserted.*

# Pretzels

Consider a vending machine that alternatingly accepts a coin ($c$) and produces a pretzel ($p$).




$$VM = c.p.VM$$




In LTL we have $VM \models \mathbf{G}(c \Rightarrow \mathbf{F}p)$.

*whenever a coin is inserted, eventually a pretzel is produced.*

But we also have $VM \models \mathbf{G}(p \Rightarrow \mathbf{F}c)$.

*whenever a pretzel is produced, eventually a new coin will be inserted.*

This example shows that standard LTL is not suitable to correctly describe the behaviour of this vending machine.

# Reactive Temporal Logic

Temporal judgements $P \models^{CC}_{B} \varphi$

$P$ is a process.

$\varphi$ is a temporal formula in LTL, CTL, or some other logic.

$B$ is a set of actions that may be blocked by the environment.

# Reactive Temporal Logic

Temporal judgements $P \models^{CC}_{B} \varphi$

$P$ is a process.

$\varphi$ is a temporal formula in LTL, CTL, or some other logic.

$B$ is a set of actions that may be blocked by the environment.

$CC$ is a *completeness criterion*, such as progress, justness, weak fairness or strong fairness.

In default temporal logic infinite paths are complete; finite paths not.

$B$ declares some of the finite paths as complete.

$CC$ declares some of the infinite paths as incomplete.

# Reactive Temporal Logic

Temporal judgements $P \models^{CC}_{B} \varphi$

$P$ is a process.

$\varphi$ is a temporal formula in LTL, CTL, or some other logic.

$B$ is a set of actions that may be blocked by the environment.

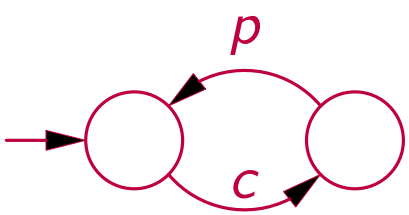
# Reactive Temporal Logic

Temporal judgements $P \models^{CC}_{B} \varphi$

$P$ is a process.

$\varphi$ is a temporal formula in LTL, CTL, or some other logic.

$B$ is a set of actions that may be blocked by the environment.




$$VM = c.p.VM$$

$$B = \{c\}$$

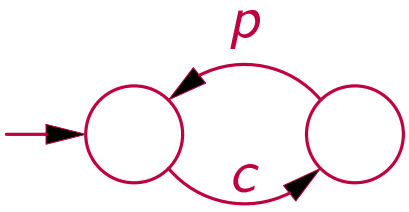
# Reactive Temporal Logic

Temporal judgements $P \models^{CC}_{B} \varphi$

$P$ is a process.

$\varphi$ is a temporal formula in LTL, CTL, or some other logic.

$B$ is a set of actions that may be blocked by the environment.




$VM = c.p.VM$

$B = \{c\}$

In LTL we had $VM \models \mathbf{G}(c \Rightarrow \mathbf{F}p)$.

*whenever a coin is inserted, eventually a pretzel is produced.*

But $VM \models \mathbf{G}(p \Rightarrow \mathbf{F}c)$.

*whenever a pretzel is produced, eventually a new coin will be inserted.*

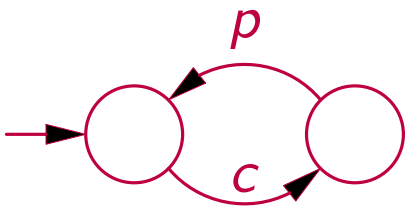
# Reactive Temporal Logic

Temporal judgements $P \models^{CC}_{B} \varphi$

$P$ is a process.

$\varphi$ is a temporal formula in LTL, CTL, or some other logic.

$B$ is a set of actions that may be blocked by the environment.




$$VM = c.p.VM$$
$$B = \{c\}$$

In reactive LTL we have $VM \models_B \mathbf{G}(c \Rightarrow \mathbf{F}p)$.

*whenever a coin is inserted, eventually a pretzel is produced.*

But $VM \not\models_B \mathbf{G}(p \Rightarrow \mathbf{F}c)$.

*whenever a pretzel is produced, eventually a new coin will be inserted.*

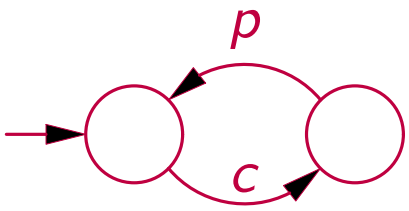
# Reactive Temporal Logic

Temporal judgements $P \models^{CC}_{B} \varphi$

$P$ is a process.

$\varphi$ is a temporal formula in LTL, CTL, or some other logic.

$B$ is a set of actions that may be blocked by the environment.




$$VM = c.p.VM$$
$$B = \{c\}$$

In reactive LTL we have $VM \models_B \mathbf{G}(c \Rightarrow \mathbf{F}p)$.

*whenever a coin is inserted, eventually a pretzel is produced.*

But $VM \not\models_B \mathbf{G}(p \Rightarrow \mathbf{F}c)$. We still have $VM \models_\emptyset \mathbf{G}(p \Rightarrow \mathbf{F}c)$.

*whenever a pretzel is produced, eventually a new coin will be inserted.*

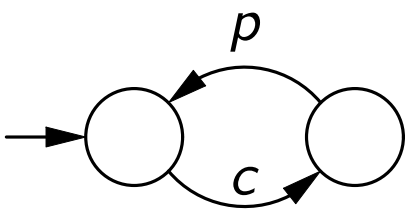
# Reactive LTL

LTL formula are traditionally interpreted on infinite paths.

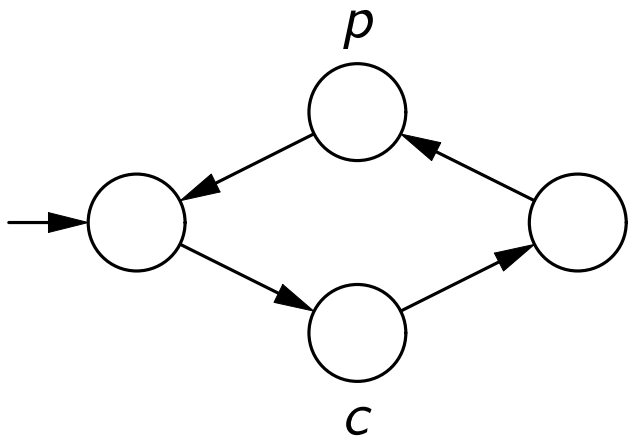
# Reactive LTL

LTL formula are ~~traditionally~~ interpreted on ~~infinite~~ paths.

# Reactive LTL

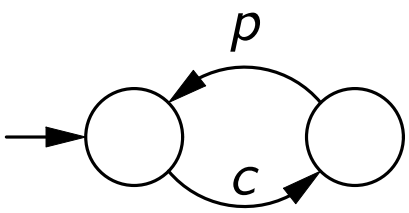



$VM = c.p.VM$

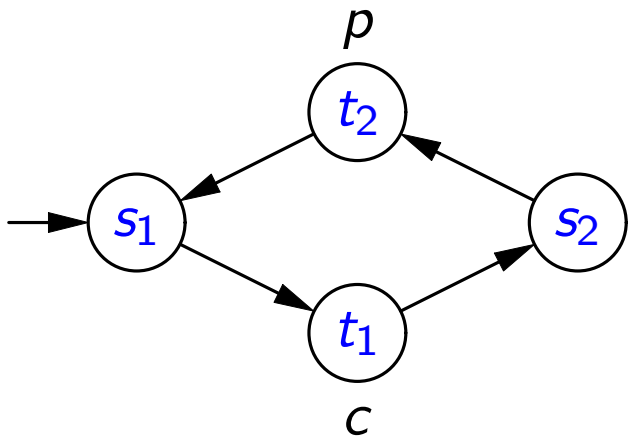



LTL formula are ~~traditionally~~ interpreted on ~~infinite~~ paths.

# Reactive LTL

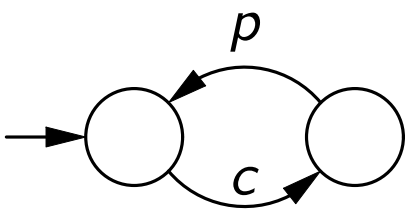



$VM = c.p.VM$

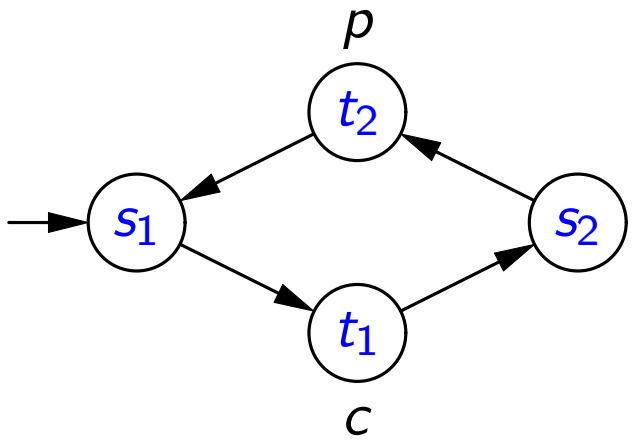



LTL formula are ~~traditionally~~ interpreted on ~~infinite~~ paths.

# Reactive LTL



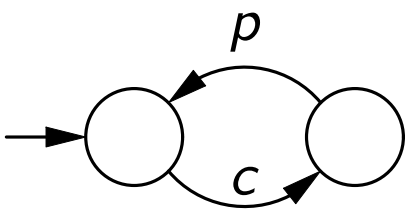

$VM = c.p.VM$



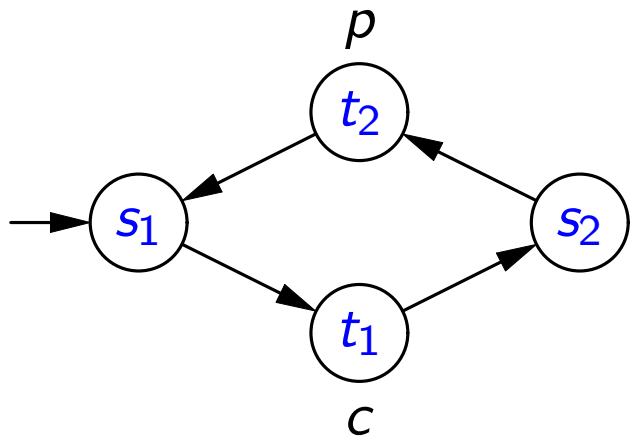

LTL formula are ~~traditionally~~ interpreted on ~~infinite~~ paths.

$s_1 t_1 s_2 t_2 s_1 t_1 s_2 t_2 s_1 \cdots \models \mathbf{G}(p \Rightarrow \mathbf{F} c)$

# Reactive LTL




$VM = c.p.VM$

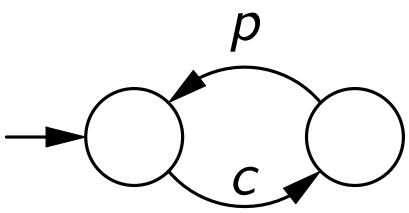



LTL formula are ~~traditionally~~ interpreted on ~~infinite~~ paths.

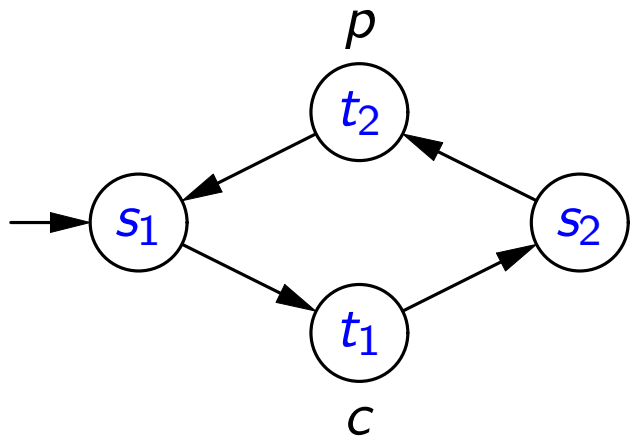
$s_1 t_1 s_2 t_2 s_1 t_1 s_2 t_2 s_1 \cdots \models \mathbf{G}(p \Rightarrow \mathbf{F}c)$

$s_1 t_1 s_2 t_2 s_1 t_1 s_2 t_2 s_1 \not\models \mathbf{G}(p \Rightarrow \mathbf{F}c)$ $\qquad VM \not\models \mathbf{G}(p \Rightarrow \mathbf{F}c)$

# Reactive LTL




$VM = c.p.VM$



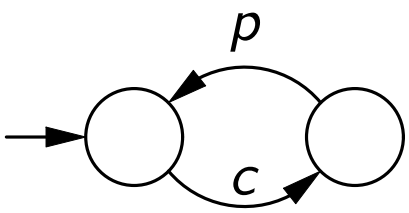

LTL formula are ~~traditionally~~ interpreted on ~~infinite~~ paths.

$s_1 t_1 s_2 t_2 s_1 t_1 s_2 t_2 s_1 \cdots \models \mathbf{G}(p \Rightarrow \mathbf{F}c)$

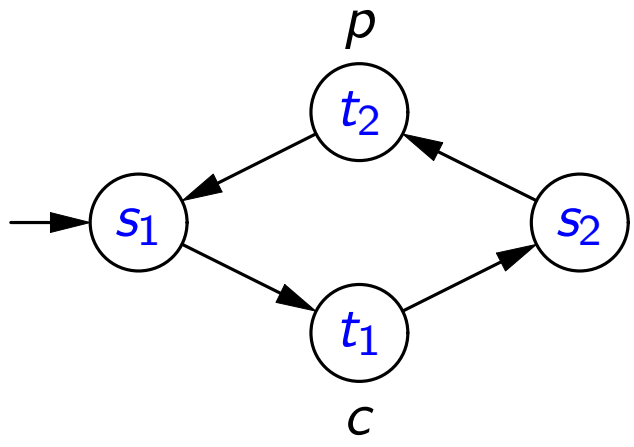
$s_1 t_1 s_2 t_2 s_1 t_1 s_2 t_2 s_1 \not\models \mathbf{G}(p \Rightarrow \mathbf{F}c)$ $\quad VM \not\models \mathbf{G}(p \Rightarrow \mathbf{F}c)$

$s_1 t_1 s_2 t_2 s_1 t_1 s_2 t_2 s_1 \models \mathbf{G}(c \Rightarrow \mathbf{F}p)$ $\quad VM \models \mathbf{G}(c \Rightarrow \mathbf{F}p)$

# Reactive LTL




$VM = c.p.VM$




LTL formula are ~~traditionally~~ interpreted on ~~infinite~~ paths.

$s_1 t_1 s_2 t_2 s_1 t_1 s_2 t_2 s_1 \cdots \models \mathbf{G}(p \Rightarrow \mathbf{F}c)$

$s_1 t_1 s_2 t_2 s_1 t_1 s_2 t_2 s_1 \not\models \mathbf{G}(p \Rightarrow \mathbf{F}c)$

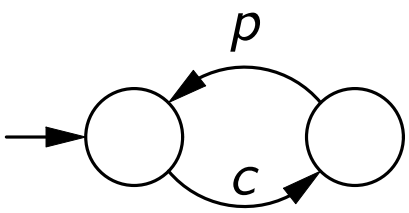
$s_1 t_1 s_2 t_2 s_1 t_1 s_2 t_2 s_1 \models \mathbf{G}(c \Rightarrow \mathbf{F}p)$

$s_1 t_1 s_2 t_2 s_1 t_1 s_2 \not\models \mathbf{G}(c \Rightarrow \mathbf{F}p)$

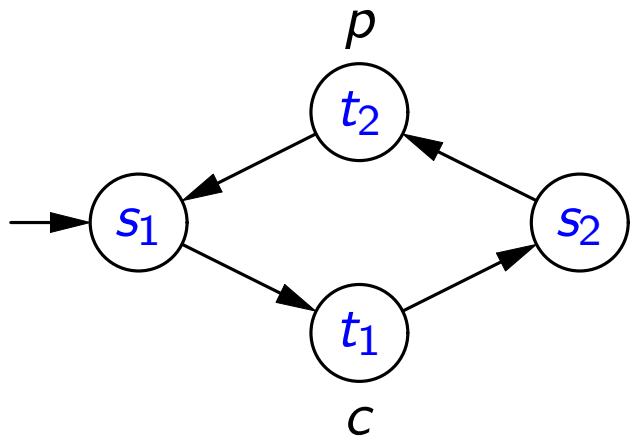
$VM \not\models \mathbf{G}(p \Rightarrow \mathbf{F}c)$

$VM \models \mathbf{G}(c \Rightarrow \mathbf{F}p)$

# Reactive LTL




$VM = c.p.VM$




LTL formula are ~~traditionally~~ interpreted on ~~infinite~~ paths.

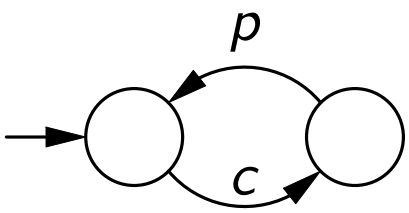
$s_1 t_1 s_2 t_2 s_1 t_1 s_2 t_2 s_1 \cdots \models \mathbf{G}(p \Rightarrow \mathbf{F}c)$ complete

$s_1 t_1 s_2 t_2 s_1 t_1 s_2 t_2 s_1 \not\models \mathbf{G}(p \Rightarrow \mathbf{F}c)$ complete $VM \not\models_B \mathbf{G}(p \Rightarrow \mathbf{F}c)$

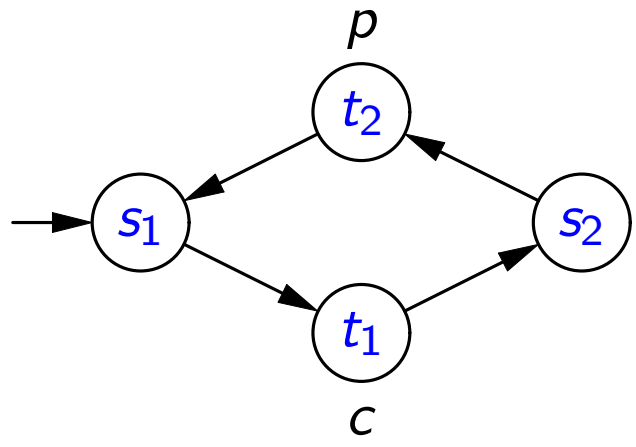
$s_1 t_1 s_2 t_2 s_1 t_1 s_2 t_2 s_1 \models \mathbf{G}(c \Rightarrow \mathbf{F}p)$ complete $VM \models_B \mathbf{G}(c \Rightarrow \mathbf{F}p)$

$s_1 t_1 s_2 t_2 s_1 t_1 s_2 \not\models \mathbf{G}(c \Rightarrow \mathbf{F}p)$ incomplete

# Reactive LTL




$VM = c.p.VM$




LTL formula are ~~traditionally~~ interpreted on ~~infinite~~ paths.

$s_1 t_1 s_2 t_2 s_1 t_1 s_2 t_2 s_1 \cdots \models \mathbf{G}(p \Rightarrow \mathbf{F}c)$ complete

$s_1 t_1 s_2 t_2 s_1 t_1 s_2 t_2 s_1 \not\models \mathbf{G}(p \Rightarrow \mathbf{F}c)$ complete $VM \not\models \mathbf{G}(p \Rightarrow \mathbf{F}c)$

$s_1 t_1 s_2 t_2 s_1 t_1 s_2 t_2 s_1 \models \mathbf{G}(c \Rightarrow \mathbf{F}p)$ complete $VM \models \mathbf{G}(c \Rightarrow \mathbf{F}p)$

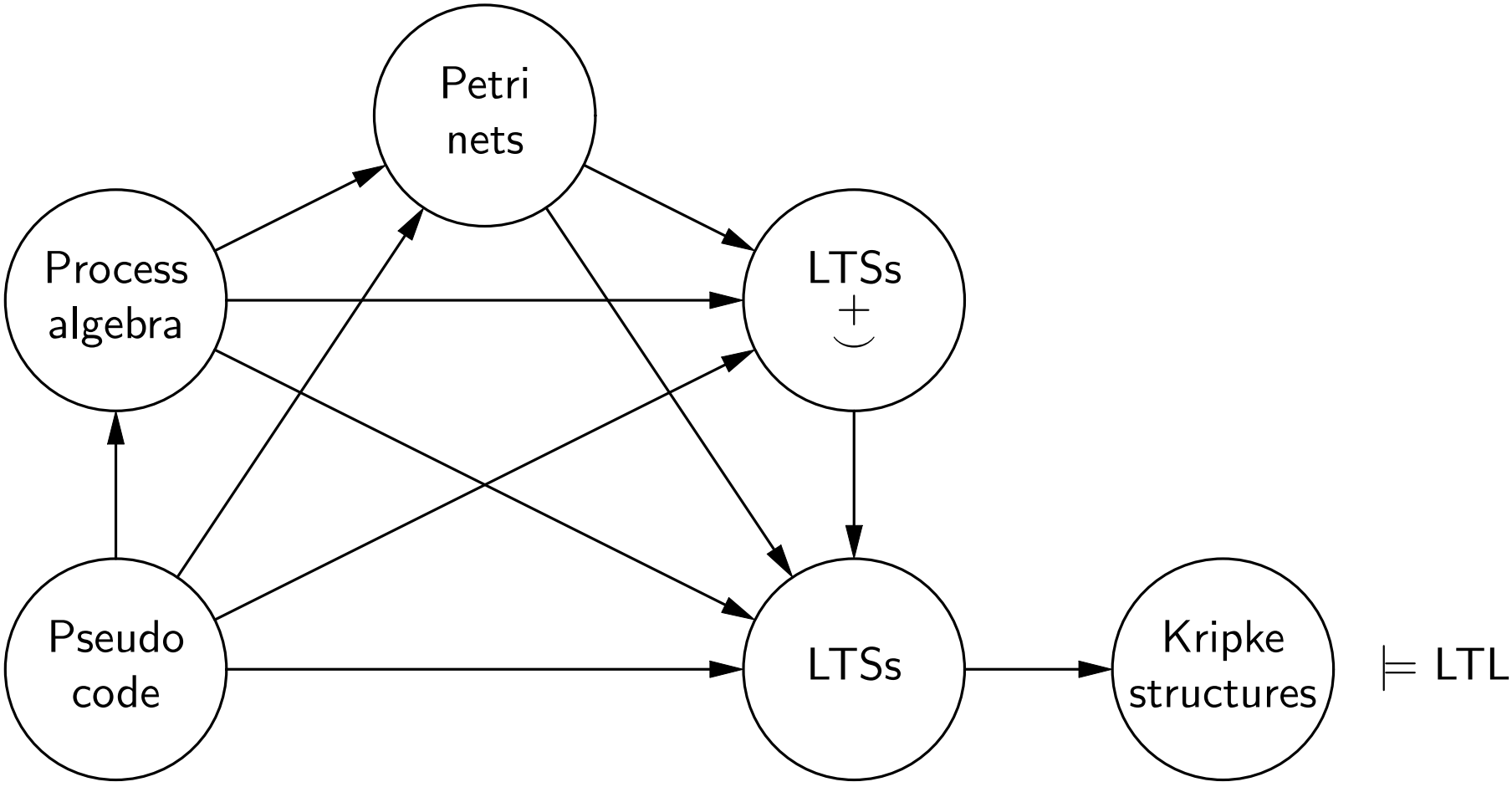
$s_1 t_1 s_2 t_2 s_1 t_1 s_2 \not\models \mathbf{G}(c \Rightarrow \mathbf{F}p)$ incomplete

$$B = \{c\}$$

$P \models \varphi$ iff $\pi \models \varphi$ for all complete paths $\pi$.

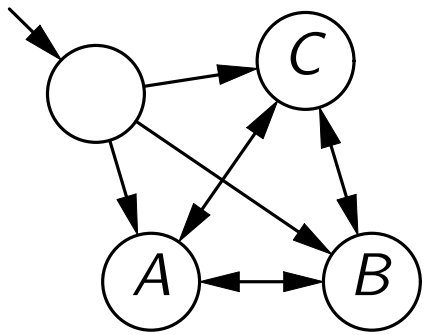
A path is complete iff either it is infinite, or its last state has outgoing transitions with labels from $B$ only.

# Models for distributed systems



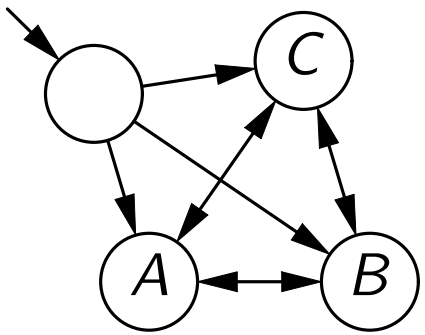
# Fairness

Alice, Bart and Cameron stand behind a bar, ordering and drinking beer.
As there is only one barman, they are served sequentially.
None of them is served twice in a row, but each of them is ready
for the next beer as soon as another person is served.



# Fairness

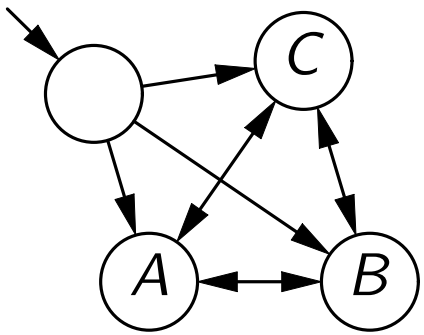
Alice, Bart and Cameron stand behind a bar, ordering and drinking beer.
As there is only one barman, they are served sequentially.
None of them is served twice in a row, but each of them is ready
for the next beer as soon as another person is served.




$\not\models \quad \mathbf{F}B$

# Fairness

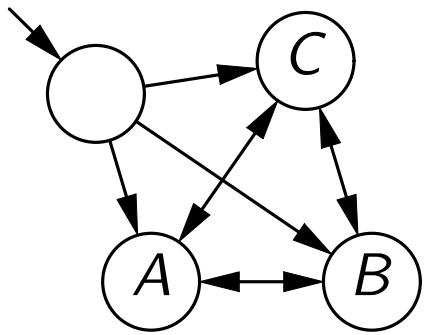
Alice, Bart and Cameron stand behind a bar, ordering and drinking beer.
As there is only one barman, they are served sequentially.
None of them is served twice in a row, but each of them is ready
for the next beer as soon as another person is served.




$\models^{WF} \mathbf{F}B$

# Fairness

Alice, Bart and Cameron stand behind a bar, ordering and drinking beer.
As there is only one barman, they are served sequentially.
None of them is served twice in a row, but each of them is ready
for the next beer as soon as another person is served.

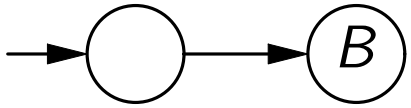



$\models^{WF} \mathbf{F}B$

Assuming fairness often leads to unwarranted conclusions.

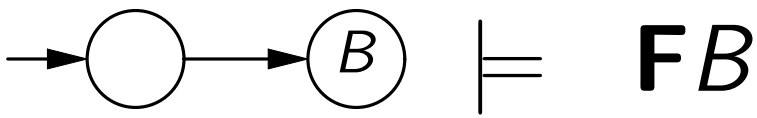
# Progress

Bart is the only customer in a bar in London, with a single barman.
He only wants one beer.



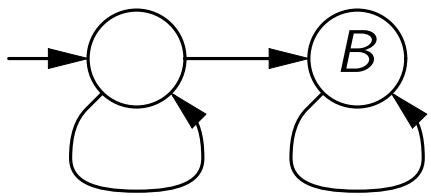
# Progress

Bart is the only customer in a bar in London, with a single barman.
He only wants one beer.



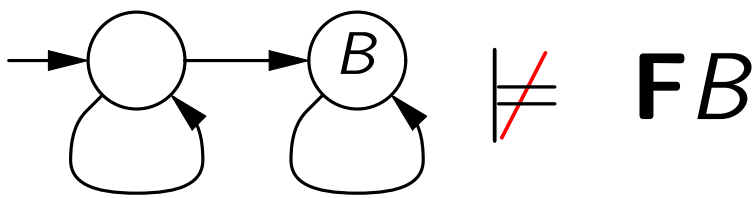
# Justness

Bart is the only customer in a bar in London, with a single barman.
He only wants one beer.
Alice and Cameron are in a bar in Tokyo. They drink a lot of beer.
There is no interaction of any kind between the two bars.
Yet, one may choose to model the drinking in these two bars as a single distributed system.



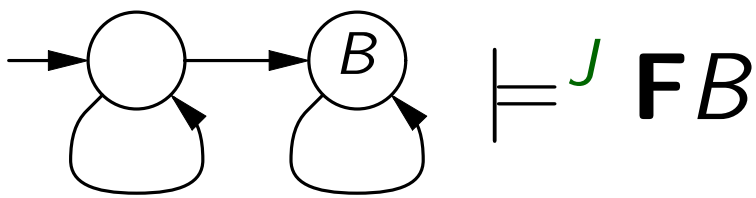
# Justness

Bart is the only customer in a bar in London, with a single barman.
He only wants one beer.
Alice and Cameron are in a bar in Tokyo. They drink a lot of beer.
There is no interaction of any kind between the two bars.
Yet, one may choose to model the drinking in these two bars as a single distributed system.



# Justness

Bart is the only customer in a bar in London, with a single barman.
He only wants one beer.
Alice and Cameron are in a bar in Tokyo. They drink a lot of beer.
There is no interaction of any kind between the two bars.
Yet, one may choose to model the drinking in these two bars as a single distributed system.




Each transition that is at some point enabled, will eventually occur, unless one of its necessary resources is used for another transition.

# Session types

Another process algebra:

$$
\begin{array}{rcl}
\mathrm{T} := & \textsc{ok} & \\
\mid & \bigoplus_{i \in I}\ p_i!\lambda_i; \mathrm{T}_i & \qquad \mathbb{N} := \quad p[\![\mathrm{T}]\!] \\
\mid & \sum_{i \in I}\ p_i?\lambda_i; \mathrm{T}_i & \qquad\quad \mid \quad 0 \\
\mid & X & \qquad\quad \mid \quad \mathbb{N} \parallel \mathbb{N} \\
\mid & \mu X.\mathrm{T} &
\end{array}
$$

Example:

$$
\begin{array}{rl}
 & \mathit{seller1}[\![\mu X.\mathit{buyer1}?; X]\!] \\
\parallel & \mathit{buyer1}[\![\mu Y.\mathit{seller1}!; Y]\!] \\
\parallel & \mathit{seller2}[\![\mu Z.\mathit{buyer2}?; Z]\!] \\
\parallel & \mathit{buyer2}[\![\mu W.\mathit{seller2}!; W]\!]
\end{array}
$$

Lock-freedom: Everyone wishing to trade eventually does so.

# Session types

Another process algebra:

$$
\begin{array}{rcl}
\mathrm{T} := & \textsc{ok} & \\
\mid & \bigoplus_{i \in I} \; p_i!\lambda_i; \mathrm{T}_i & \qquad \mathbb{N} := \quad p[\![\mathrm{T}]\!] \\
\mid & \sum_{i \in I} \; p_i?\lambda_i; \mathrm{T}_i & \qquad\quad \mid \quad 0 \\
\mid & X & \qquad\quad \mid \quad \mathbb{N} \parallel \mathbb{N} \\
\mid & \mu X.\mathrm{T} &
\end{array}
$$

Example:

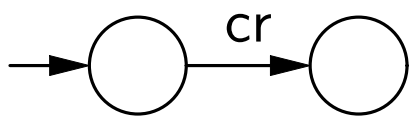
$$
\begin{array}{rl}
 & \mathit{seller1}[\![\mu X.\mathit{buyer1}?; X]\!] \\
\parallel & \mathit{buyer1}[\![\mu Y.\mathit{seller1}!; Y]\!] \\
\parallel & \mathit{seller2}[\![\mu Z.\mathit{buyer2}?; Z]\!] \\
\parallel & \mathit{buyer2}[\![\mu W.\mathit{seller2}!; W]\!]
\end{array}
$$

Lock-freedom: Everyone wishing to trade eventually does so.

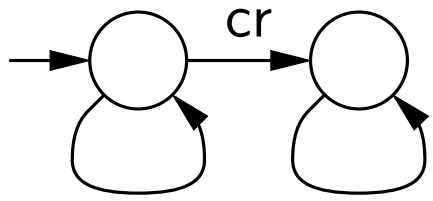
Theorem: $\mathcal{L}(J)$ holds iff network is well-typed and race-free.

# Justness

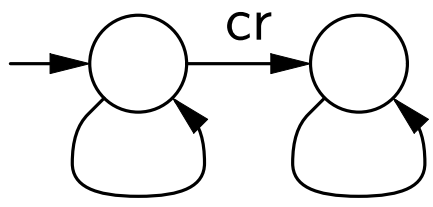
Cateline, eating a croissant in Paris:




In parallel with Alice, making an infinite sequence of phone calls in London:



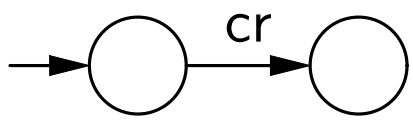

Cateline, making an infinite sequence of choices between eating her croissant and making yet another phone call:



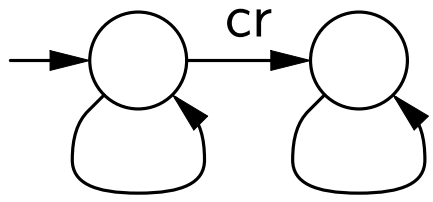

In Petri nets, these systems have totally different representations.

# Justness

Cateline, eating a croissant in Paris:

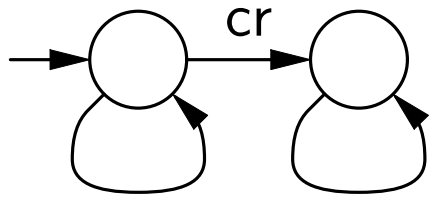



In parallel with Alice, making an infinite sequence of phone calls in London:




$\models^{J}_{\emptyset}$ $\mathbf{F}\,\mathrm{cr}$

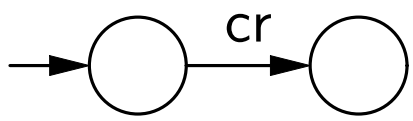
Cateline, making an infinite sequence of choices between eating her croissant and making yet another phone call:



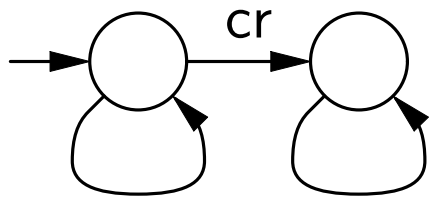

In Petri nets, these systems have totally different representations.

# Justness

Cateline, eating a croissant in Paris:

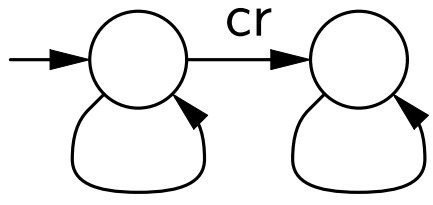



In parallel with Alice, making an infinite sequence of phone calls in London:




$\models^{J}_{\emptyset}$ **F** cr

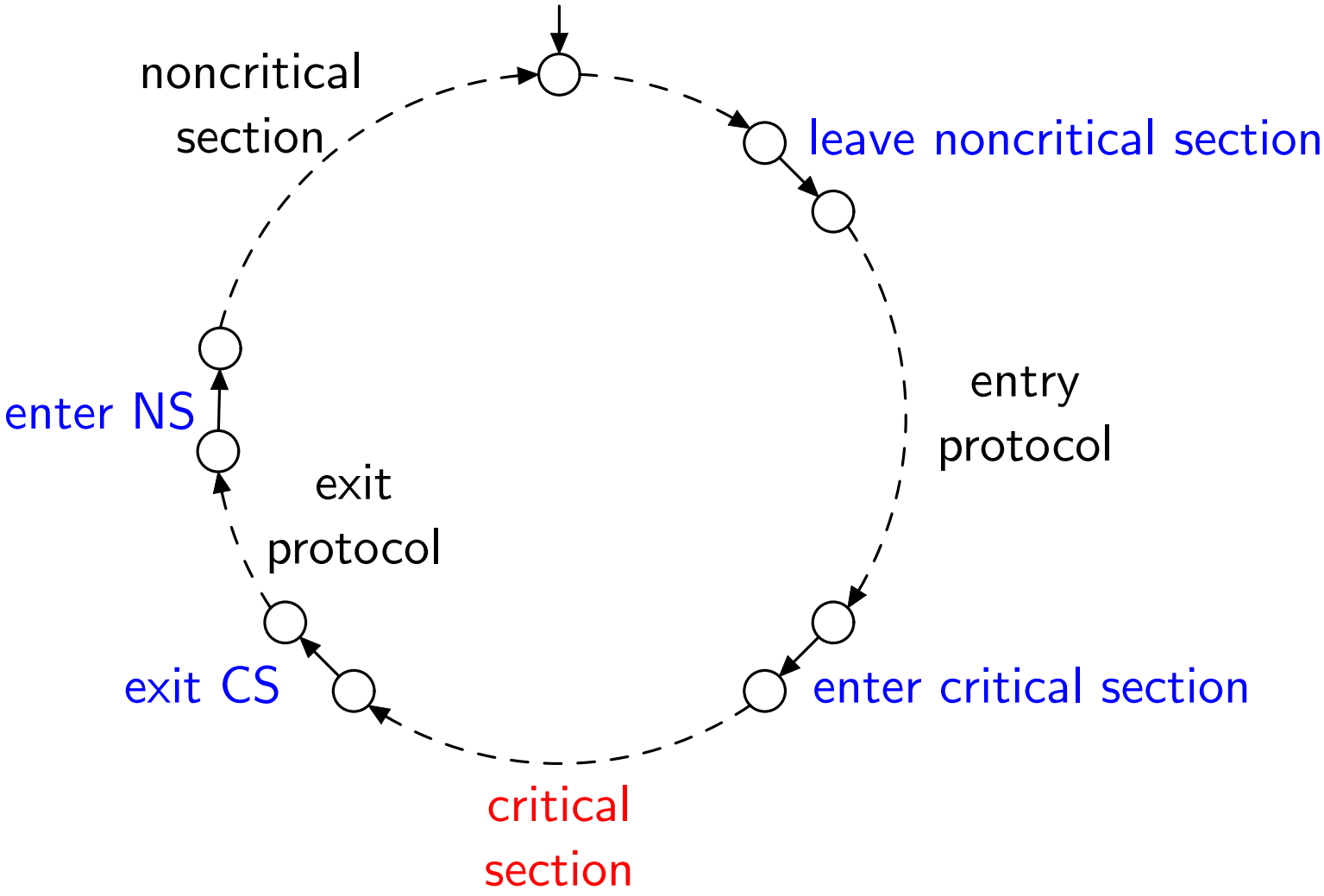
Cateline, making an infinite sequence of choices between eating her croissant and making yet another phone call:




$\not\models^{J}_{\emptyset}$ **F** cr

In Petri nets, these systems have totally different representations.

# Mutual exclusion



# Peterson's mutual exclusion algorithm

**Process A**

**repeat forever**

$\ell_1$ **leave noncritical section**
$\ell_2$ $readyA := true$
$\ell_3$ $turn := B$
$\ell_4$ **await** $(readyB = false \vee turn = A)$
$\ell_5$ **enter critical section**
$\ell_6$ **leave critical section**
$\ell_7$ $readyA := false$
$\ell_8$ **enter noncritical section**

**Process B**

**repeat forever**

$m_1$ **leave noncritical section**
$m_2$ $readyB := true$
$m_3$ $turn := A$
$m_4$ **await** $(readyA = false \vee turn = B)$
$m_5$ **enter critical section**
$m_6$ **leave critical section**
$m_7$ $readyB := false$
$m_8$ **enter noncritical section**

*Peterson's algorithm (pseudocode)*

# Peterson's mutual exclusion algorithm

**Process A**

**repeat forever**

$\ell_1$ **leave noncritical section**
$\ell_2$ $readyA := true$
$\ell_3$ $turn := B$
$\ell_4$ **await** $(readyB = false \vee turn = A)$
$\ell_5$ **enter critical section**
$\ell_6$ **leave critical section**
$\ell_7$ $readyA := false$
$\ell_8$ **enter noncritical section**

**Process B**

**repeat forever**

$m_1$ **leave noncritical section**
$m_2$ $readyB := true$
$m_3$ $turn := A$
$m_4$ **await** $(readyA = false \vee turn = B)$
$m_5$ **enter critical section**
$m_6$ **leave critical section**
$m_7$ $readyB := false$
$m_8$ **enter noncritical section**

*Peterson's algorithm (pseudocode)*

Mutual exclusion: Never are both processes together in the critical section.
Starvation-freedom: Any process that leaves its noncritical section, will enter its critical section.

# Peterson's mutual exclusion algorithm

**Process A**

**repeat forever**

$\ell_1$ **leave noncritical section**
$\ell_2$ $readyA := true$
$\ell_3$ $turn := B$
$\ell_4$ **await** $(readyB = false \vee turn = A)$
$\ell_5$ **enter critical section**
$\ell_6$ **leave critical section**
$\ell_7$ $readyA := false$
$\ell_8$ **enter noncritical section**

**Process B**

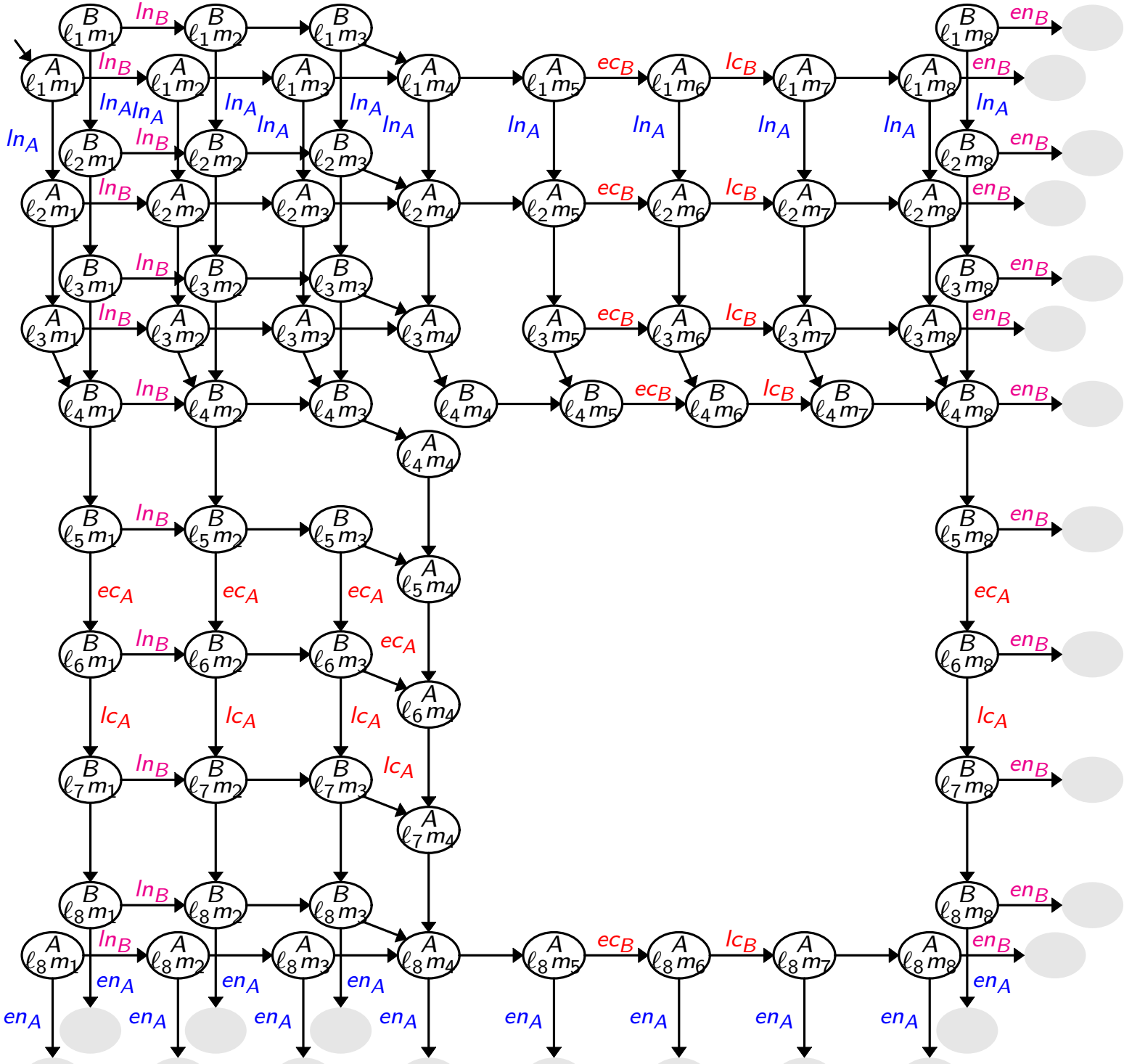
**repeat forever**

$m_1$ **leave noncritical section**
$m_2$ $readyB := true$
$m_3$ $turn := A$
$m_4$ **await** $(readyA = false \vee turn = B)$
$m_5$ **enter critical section**
$m_6$ **leave critical section**
$m_7$ $readyB := false$
$m_8$ **enter noncritical section**

*Peterson's algorithm (pseudocode)*

Mutual exclusion: Never are both processes together in the critical section.
Starvation-freedom: Any process that leaves its noncritical section, will enter its critical section.

Speed independence: the protocol should work regardless how long things take.
Atomicity: a memory cell cannot handle a read and a write at the same time. So a write will have to wait until a pending read is finished.


$B\ \ell_1 m_1$
$ln_B$
$B\ \ell_1 m_2$
$B\ \ell_1 m_3$
$B\ \ell_1 m_8$
$en_B$
$A\ \ell_1 m_1$
$A\ \ell_1 m_2$
$A\ \ell_1 m_3$
$A\ \ell_1 m_4$
$A\ \ell_1 m_5$
$ec_B$
$A\ \ell_1 m_6$
$lc_B$
$A\ \ell_1 m_7$
$A\ \ell_1 m_8$
$ln_A$
$B\ \ell_2 m_1$
$B\ \ell_2 m_2$
$B\ \ell_2 m_3$
$B\ \ell_2 m_8$
$A\ \ell_2 m_1$
$A\ \ell_2 m_2$
$A\ \ell_2 m_3$
$A\ \ell_2 m_4$
$A\ \ell_2 m_5$
$A\ \ell_2 m_6$
$A\ \ell_2 m_7$
$A\ \ell_2 m_8$
$B\ \ell_3 m_1$
$B\ \ell_3 m_2$
$B\ \ell_3 m_3$
$B\ \ell_3 m_8$
$A\ \ell_3 m_1$
$A\ \ell_3 m_2$
$A\ \ell_3 m_3$
$A\ \ell_3 m_4$
$A\ \ell_3 m_5$
$A\ \ell_3 m_6$
$A\ \ell_3 m_7$
$A\ \ell_3 m_8$
$B\ \ell_4 m_1$
$B\ \ell_4 m_2$
$B\ \ell_4 m_3$
$B\ \ell_4 m_4$
$B\ \ell_4 m_5$
$B\ \ell_4 m_6$
$B\ \ell_4 m_7$
$B\ \ell_4 m_8$
$A\ \ell_4 m_4$
$B\ \ell_5 m_1$
$B\ \ell_5 m_2$
$B\ \ell_5 m_3$
$B\ \ell_5 m_8$
$ec_A$
$A\ \ell_5 m_4$
$B\ \ell_6 m_1$
$B\ \ell_6 m_2$
$B\ \ell_6 m_3$
$B\ \ell_6 m_8$
$lc_A$
$A\ \ell_6 m_4$
$B\ \ell_7 m_1$
$B\ \ell_7 m_2$
$B\ \ell_7 m_3$
$B\ \ell_7 m_8$
$A\ \ell_7 m_4$
$B\ \ell_8 m_1$
$B\ \ell_8 m_2$
$B\ \ell_8 m_3$
$B\ \ell_8 m_8$
$A\ \ell_8 m_1$
$A\ \ell_8 m_2$
$A\ \ell_8 m_3$
$A\ \ell_8 m_4$
$A\ \ell_8 m_5$
$A\ \ell_8 m_6$
$A\ \ell_8 m_7$
$A\ \ell_8 m_8$
$en_A$

# Correctness properties of MUTEX protocols

$$\text{(ORD)} \quad P \models \big((\neg\, act_i)\mathbf{W}\, ln_i\big) \wedge \mathbf{G}\,\big(ln_i \Rightarrow \mathbf{Y}\big((\neg\, act_i)\mathbf{W}\, ec_i\big)\big) \wedge \mathbf{G}\,\big(ec_i \Rightarrow \mathbf{Y}\big((\neg\, act_i)\mathbf{W}\, lc_i\big)\big) \wedge \mathbf{G}\,\big(lc_i \Rightarrow \mathbf{Y}\big((\neg\, act_i)\mathbf{W}\, en_i\big)\big) \wedge \mathbf{G}\,\big(en_i \Rightarrow \mathbf{Y}\big((\neg\, act_i)\mathbf{W}\, ln_i\big)\big)$$

for $i = 1, \ldots, N$. Here $act_i := (ln_i \vee ec_i \vee lc_i \vee en_i)$.

This requirement says that in a mutex protocol the actions $ln_i$, $ec_i$, $lc_i$ and $en_i$ must occur in the right order.

# Correctness properties of MUTEX protocols

$$\text{(ORD)} \quad P \models \big((\neg\, act_i)\mathbf{W} ln_i\big) \wedge \mathbf{G}\,\big(ln_i \Rightarrow \mathbf{Y}\big((\neg\, act_i)\mathbf{W} ec_i\big)\big) \wedge \mathbf{G}\,\big(ec_i \Rightarrow \mathbf{Y}\big((\neg\, act_i)\mathbf{W} lc_i\big)\big) \wedge \mathbf{G}\,\big(lc_i \Rightarrow \mathbf{Y}\big((\neg\, act_i)\mathbf{W} en_i\big)\big) \wedge \mathbf{G}\,\big(en_i \Rightarrow \mathbf{Y}\big((\neg\, act_i)\mathbf{W} ln_i\big)\big)$$

for $i = 1, \ldots, N$. Here $act_i := (ln_i \vee ec_i \vee lc_i \vee en_i)$.

$$\text{(ME)} \quad P \models \mathbf{G}\,\big(ec_i \Rightarrow \big((\neg ec_j)\mathbf{W} lc_i\big)\big)$$

for all $i, j = 1, \ldots, N$ with $i \neq j$.

# Correctness properties of MUTEX protocols

$(\mathrm{EC}^J)$ $\quad P \models_B^J \mathbf{G}(\mathit{ln}_i \Rightarrow \mathbf{F}\mathit{ec}_i)$

for $i = 1, \ldots, N$.

# Correctness properties of MUTEX protocols

$(\mathrm{EC}^J)$ $\quad P \models^J_B \mathbf{G}(ln_i \Rightarrow \mathbf{F}\, ec_i)$

for $i = 1, \ldots, N$.

$(\mathrm{LC}^J)$ $\quad P \models^J_B \mathbf{G}(ec_i \Rightarrow \mathbf{F}\, lc_i)$

$(\mathrm{EN}^J)$ $\quad P \models^J_B \mathbf{G}(lc_i \Rightarrow \mathbf{F}\, en_i)$

$(\mathrm{LN}^J)$ $\quad P \models^J_{B \setminus \{ln_i\}} \mathbf{F}\, ln_i \wedge \mathbf{G}(en_i \Rightarrow \mathbf{F}\, ln_i)$

for $i = 1, \ldots, N$.

# Memory model

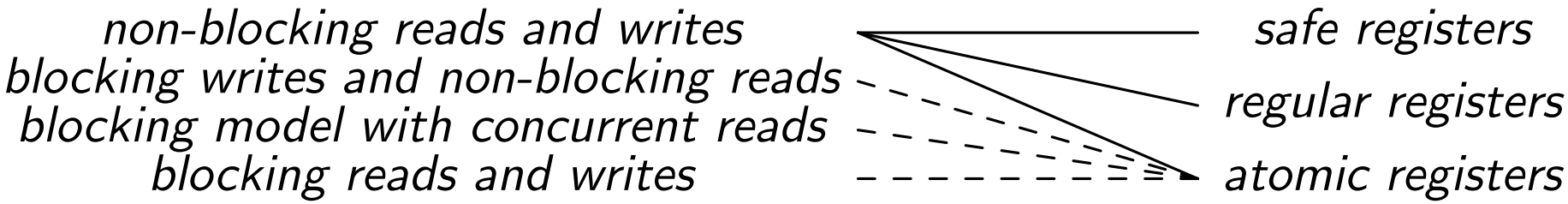
The processes in a mutex protocol communicate with each other by writing to and reading from shared registers.

# Memory model

The processes in a mutex protocol communicate with each other by writing to and reading from shared registers. Whether a mutex protocol is correct depends on the way these registers work.

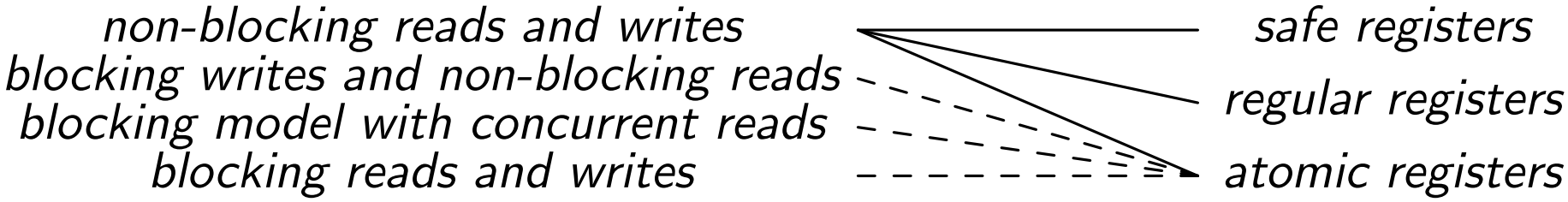
# Memory model

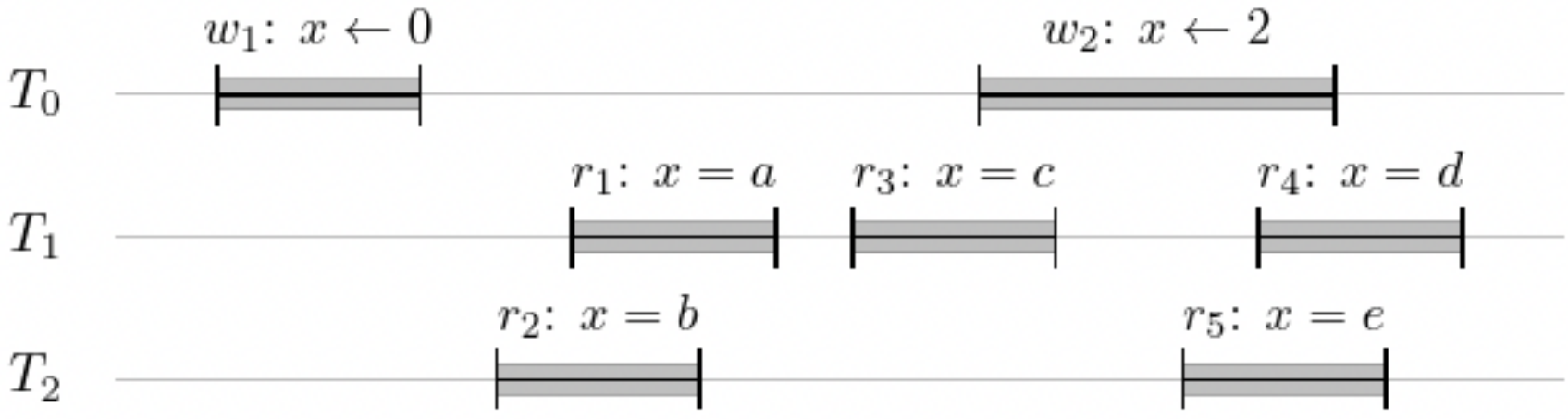
The processes in a mutex protocol communicate with each other by writing to and reading from shared registers. Whether a mutex protocol is correct depends on the way these registers work.



# Memory model

The processes in a mutex protocol communicate with each other by writing to and reading from shared registers. Whether a mutex protocol is correct depends on the way these registers work.



# Model checking

Traditional verification of mutual exclusion algorithms:
pen-and-paper proofs using behavioural reasoning.

> *"the behavioral reasoning used in our correctness proofs, and in most other published correctness proofs of concurrent algorithms, is inherently unreliable" [Lamport 1986]*

This is especially the case when dealing with non-atomic registers.

# Model checking

Traditional verification of mutual exclusion algorithms:
pen-and-paper proofs using behavioural reasoning.

> *"the behavioral reasoning used in our correctness proofs, and in most other published correctness proofs of concurrent algorithms, is inherently unreliable" [Lamport 1986]*

This is especially the case when dealing with non-atomic registers.

Alternatives: interactive theorem proving or model checking.

Model checking: Precise modelling requires great care,
the verification requires a mere button-push and some patience.

# Model checking

Traditional verification of mutual exclusion algorithms:
pen-and-paper proofs using behavioural reasoning.

> *"the behavioral reasoning used in our correctness proofs, and in most other published correctness proofs of concurrent algorithms, is inherently unreliable" [Lamport 1986]*

This is especially the case when dealing with non-atomic registers.

Alternatives: interactive theorem proving or model checking.

Model checking: Precise modelling requires great care,
the verification requires a mere button-push and some patience.

But only finite state spaces.
So no bakery protocol
and we can check only for a small number of threads.

# Results

| *Algorithm* | *# threads* | *Safe* | *Regular* | | *Atomic* | | |
|---|---|---|---|---|---|---|---|
| | | *T* | *T* | *T* | *S* | *I* | *A* |
| Anderson | 2 | S | S | S | S | M | M |
| Aravind BLRU | 3 | S | S | S | M | M | M |
| Aravind BLRU (alt.) | 3 | S | S | S | S | M | M |
| Attiya-Welch (orig.) | 2 | D | S | S | D | M | M |
| Attiya-Welch (orig., alt.) | 2 | S | S | S | D | M | M |
| Attiya-Welch (var.) | 2 | M | M | S | D | M | M |
| Attiya-Welch (var., alt.) | 2 | S | S | S | D | M | M |
| Burns-Lynch | 3 | D | D | D | D | M | M |
| Dekker | 2 | M | M | S | D | M | M |
| Dekker (alt.) | 2 | M | M | S | S | M | M |
| Dekker RW-safe | 2 | S | S | S | D | M | M |
| Dekker RW-safe (DFtoSF) | 2 | S | S | S | S | M | M |
| Dijkstra | 3 | M | D | D | M | M | M |

# Results (continued)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Kessels | 2 | X | X | S | S | M | M |
| Knuth | 3 | M | S | S | M | M | M |
| Lamport 1-bit | 3 | D | D | D | D | M | M |
| Lamport 1-bit (DFtoSF) | 3 | S | S | S | S | M | M |
| Lamport 3-bit | 3 | S | S | S | S | M | M |
| Peterson | 2 | X | X | S | S | M | M |
| Szymanski flag (int.) | 3 | X | X | S | S | M | M |
| Szymanski flag (bit) | 3 | X | X | X | X | X | X |
| Szymanski 3-bit lin. wait | 3 | X | X | X | X | X | X |
| Szymanski 3-bit lin. wait (alt.) | 2 | S | S | S | S | M | M |

# Speed independence

> Nothing may be assumed about the relative speed of the processes competing for access to the critical section. [Dijkstra'65]

# Speed independence

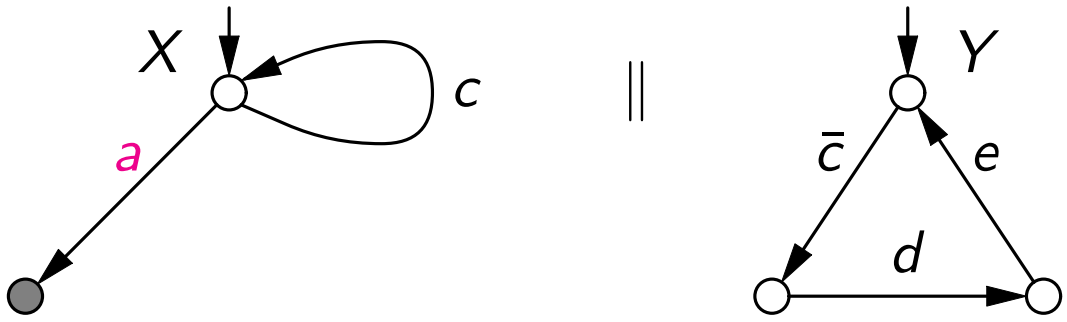
Nothing may be assumed about the relative speed of the processes competing for access to the critical section. [Dijkstra'65]

If two processes $A$ and $B$ are engaged in a race, and
$A$ has nothing else to do but performing the winning action,
whilst $B$ has a long list of tasks that must be done first,
it may still happen that $B$ wins.

# Speed independence

Nothing may be assumed about the relative speed of the processes competing for access to the critical section. [Dijkstra'65]

CCS process $(X|Y)\backslash c$ with $X \stackrel{def}{=} a.\mathbf{0} + c.X$ and $Y \stackrel{def}{=} \bar{c}.d.e.Y$.

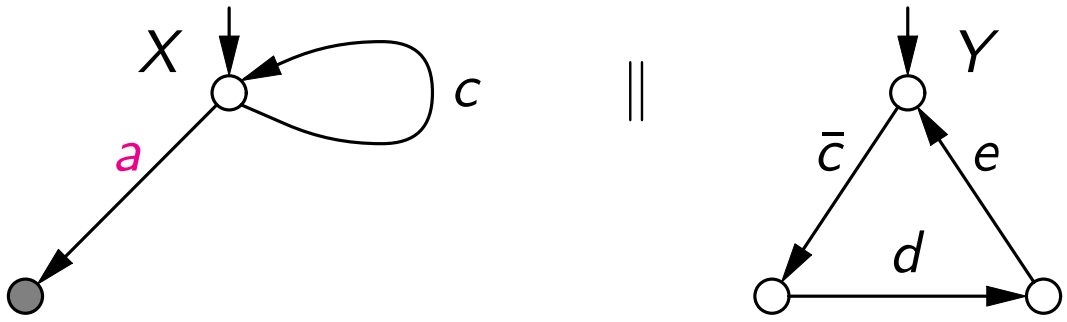



If two processes $A$ and $B$ are engaged in a race, and
$A$ has nothing else to do but performing the winning action,
whilst $B$ has a long list of tasks that must be done first,
it may still happen that $B$ wins.

# Speed independence

Nothing may be assumed about the relative speed of the processes competing for access to the critical section. [Dijkstra'65]

CCS process $(X|Y)\backslash c$ with $X \stackrel{def}{=} a.\mathbf{0} + c.X$ and $Y \stackrel{def}{=} \bar{c}.d.e.Y$.




Here it is possible that $a$ never happens.

If two processes $A$ and $B$ are engaged in a race, and
$A$ has nothing else to do but performing the winning action,
whilst $B$ has a long list of tasks that must be done first,
it may still happen that $B$ wins.

# Impossibility of speed ind. mutual excl. under atomicity

For mutual exclusion to be possible, there must be a variable *readyP* that is written by Proc. $P$ to request entry to CS.

# Impossibility of speed ind. mutual excl. under atomicity

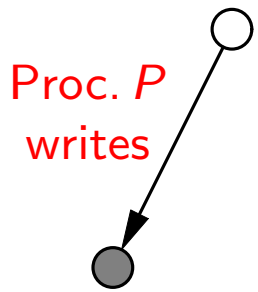
For mutual exclusion to be possible, there must be a variable *readyP* that is written by Proc. $P$ to request entry to CS.

*readyP* must be read by Proc. $Q$, before Proc. $Q$ can enter CS.

# Impossibility of speed ind. mutual excl. under atomicity

For mutual exclusion to be possible, there must be a variable *readyP* that is written by Proc. $P$ to request entry to CS.

*readyP* must be read by Proc. $Q$, before Proc. $Q$ can enter CS.

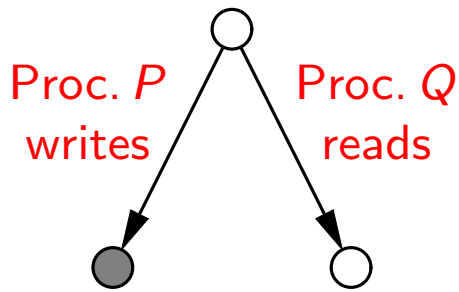



It suffices to present a scenario where Proc. $P$ is ready to write to *readyP* yet never succeeds in doing so, as that would violate EC or LN.

# Impossibility of speed ind. mutual excl. under atomicity

For mutual exclusion to be possible, there must be a variable *readyP* that is written by Proc. $P$ to request entry to CS.

*readyP* must be read by Proc. $Q$, before Proc. $Q$ can enter CS.

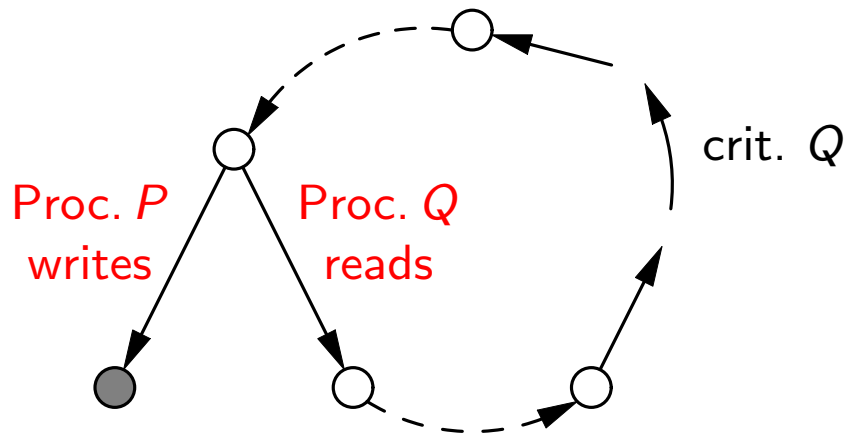



It suffices to present a scenario where Proc. $P$ is ready to write to *readyP* yet never succeeds in doing so, as that would violate EC or LN.

# Impossibility of speed ind. mutual excl. under atomicity

For mutual exclusion to be possible, there must be a variable *readyP* that is written by Proc. $P$ to request entry to CS.

*readyP* must be read by Proc. $Q$, before Proc. $Q$ can enter CS.




It suffices to present a scenario where Proc. $P$ is ready to write to *readyP* yet never succeeds in doing so, as that would violate EC or LN.