Chapter 9

Labelled Transition Systems

Systems and Processes

Remember: **Abstractly, what is a Process?**

- **Processes** are subsets of the events occurring in a system.
- In a **sequential process**, the events are fully ordered in time.

Therefore:

- A system specification is **decomposed** into process specifications.
- A system implementation is **composed** from process implementations.

System Composition

- A system specification is **decomposed** into process specifications.
- A system implementation is **composed** from process implementations.

- **Sequential composition:** every event in P_1 occurs before every event in P_2
- **Concurrent composition:** No such clear ordering imposed a priori.
- Sequential processes are basic building blocks.

Processes, Actions, Events

- A **process** is a subset of the events occurring in a system.
- The simplest possible process: empty set of events, called STOP.
- More interesting processes have events, which can also be interpreted as **actions**.
- We assume that all actions can be decomposed into **atomic actions**.
- In a system, each event belongs to **at least one** process.
- Events can be **shared** between processes — several processes can **together** engage in a single action.

Processes and State

- Processes perform **state transitions** — in different states, a processs will be able to engage in different sets of actions.

  — After some action, the set of possible continuing actions may be different from before.

- Atomic actions induce **indivisible state changes**.

- A system composed of several processes has a state that is composed from the states of the individual processes.

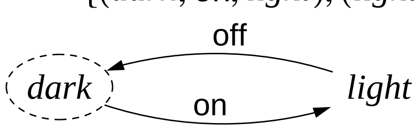
Labelled Transition Systems (LTSs)

Definition: A **labelled transition system** (S, s_0, L, δ) consists of

– a set S of *states*
– an *initial state* $s_0 : S$
– a set L of *action labels*
– a *transition relation* $\delta : \mathbb{P}(S \times L \times S)$.

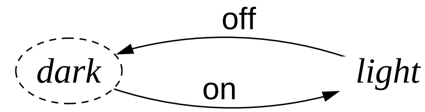
Example:

$$LightSwitch_1 = (\{dark, light\}, dark, \{\mathsf{on}, \mathsf{off}\}, \{(dark, \mathsf{on}, light), (light, \mathsf{off}, dark)\})$$

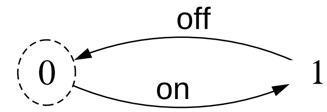


Another LTS …

$$LightSwitch_1 = (\{dark, light\}, dark, \{\mathsf{on}, \mathsf{off}\}, \{(dark, \mathsf{on}, light), (light, \mathsf{off}, dark)\})$$



$$LightSwitch_2 = (\{0, 1\}, 0, \{\mathsf{on}, \mathsf{off}\}, \{(0, \mathsf{on}, 1), (1, \mathsf{off}, 0)\})$$



Different, but **isomorphic**, where the isomorphism preserves action labels and the transition relation.

— *The identity of the states does not matter.*

Traces

Definition: A **trace** of an LTS is a sequence (finite or infinite) of action labels that results from a maximal path (with respect to the prefix ordering) starting at the initial state.

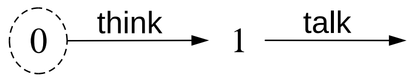
Example:

- Sequences of action labels that result from finite paths starting at the initial state:
  on
  on, off
  on, off, on
  on, off, on, off
- $LightSwitch_1$ has only **one infinite trace**:
  on, off, on, off, on, off, …
- $LightSwitch_2$ has the same set of traces as $LightSwitch_1$ — they are **behaviourally equivalent**.

Concurrent Composition

- A system composed of several processes has a state that is composed from the states of the individual processes.

Converse =

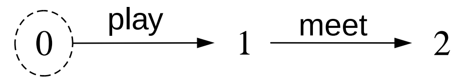


Itch =

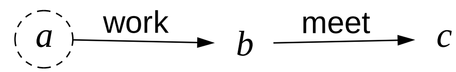
Converse || *Itch* =

Shared Actions

Bill =



Ben =



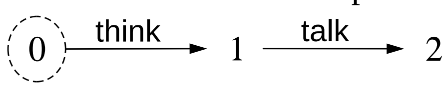
In the composition *Bill* || *Ben*,

- play and work are *concurrent actions* — the order in which they are observed does not matter.
- The **shared** action meet *synchronizes* the execution of the two constituent processes.
- Traces of the composition: play, work, meet
work, play, meet

Concurrent Composition

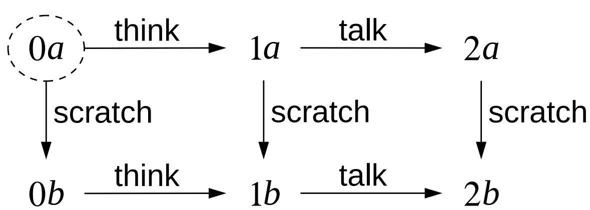
- A system composed of several processes has a state that is composed from the states of the individual processes.

Converse =



Itch =

Converse || *Itch* =



While *Converse* and *Itch* have only one trace each, their composition has three, representing *arbitrary interleaving*.

Concurrent Composition of LTSs

Definition: For $P_1 = (S_1, s_1, L_1, \delta_1)$ and $P_2 = (S_2, s_2, L_2, \delta_2)$, the **concurrent composition** $P_1 \,||\, P_2$ is the LTS

$$(S_1 \times S_2,\ (s_1, s_2),\ L_1 \cup L_2,\ \delta)$$

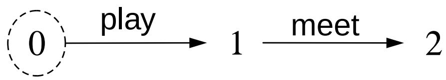
where

$$((x_1, x_2), a, (y_1, y_2)) \in \delta$$

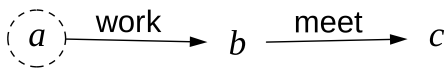
$$\Leftrightarrow \begin{cases} (x_1, a, y_1) \in \delta_1 & \wedge & x_2 = y_2 & \wedge & a \in L_1 - L_2 \\ & & \vee & & \\ x_1 = y_1 & \wedge & (x_2, a, y_2) \in \delta_2 & \wedge & a \in L_2 - L_1 \\ & & \vee & & \\ (x_1, a, y_1) \in \delta_1 & \wedge & (x_2, a, y_2) \in \delta_2 & \wedge & a \in L_1 \cap L_2 \end{cases}$$

Composition with Shared Actions

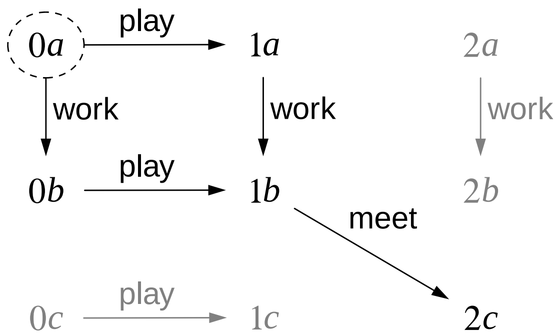
Bill =



Ben =



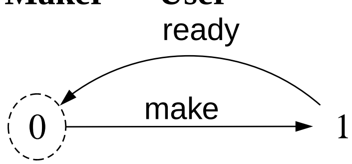
Bill || *Ben* =



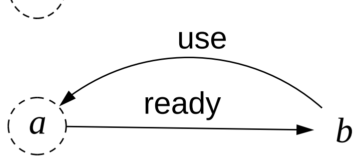
Unreachable states do not influence the behaviour!

Maker — User

Maker =



User =

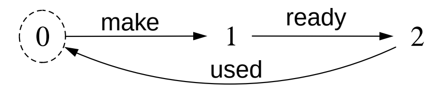


Maker || *User* =

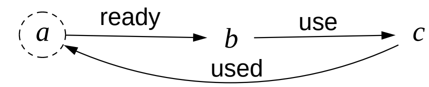
How many traces do these processes have?

Maker — User 2

$Maker_2$ =



$User_2$ =



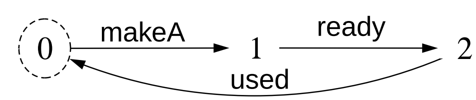
$Maker_2 \mid\mid User_2$ =

How many traces do these processes have?

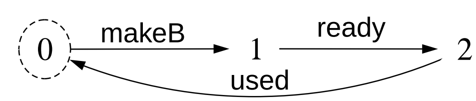
Factory

Factory = *MakerA* || *MakerB* || *Assemble*

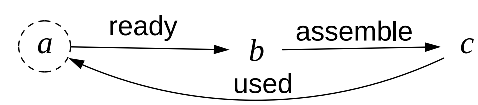
MakerA =



MakerB =



Assemble =

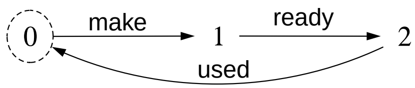


Factory =

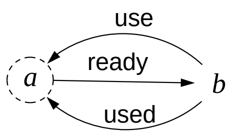
How many **states** does *Factory* have?

Maker — User 3

$Maker_2$ =



$User_3$ =



$Maker_2 \,||\, User_3$ =

How many traces do these processes have?

Deadlock

- **Deadlock** occurs in a **system** when **all** its constituent **processes** are blocked.
- A system is **deadlocked** if there are no actions it can perform.
- A **deadlock state** in an LTS is a reachable state with no outgoing transitions.
- An LTS has a deadlock state iff it has a **finite trace**.

- A terminating constituent process introduces “atypical” deadlock.
- **“Typical”** deadlocks occur in **concurrent compositions** of processes that individually are deadlock-free.

Liveness and Safety Properties

A **safety property** asserts:

“something **bad** will **never** happen”

A **liveness property** asserts:

“something **good** will **eventually** happen”

Safety

A **safety** property asserts:

“something **bad** will **never** happen”

Important safety conditions:

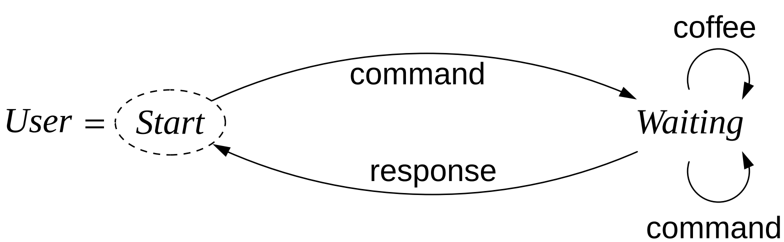
- **Partial correctness**
  - *State predicate:* If in a proper termination state, then postcondition is satisfied.
- **Invariants**
  - If in a certain kind of state, or before or after a certain kind of action, then the invariant holds for the current state.
- **Safe access sequences to resources**
  - Certain actions happen only conforming to a fixed pattern.

Such properties are often formulated using **temporal logic**.

Safe Access Sequences

- Given a system modelled as an LTS $P = (S, s, L, \delta)$, accesses to some resource (set) involve actions of a subset $A \subseteq L$.
- For every trace t of P, only its **projection** on A is considered, i.e., the sequence of those elements of t that are in A.
- These projections need to satisfy some **predicate.**
- **Conveniently:** These projections have to be traces of some (simpler) LTS

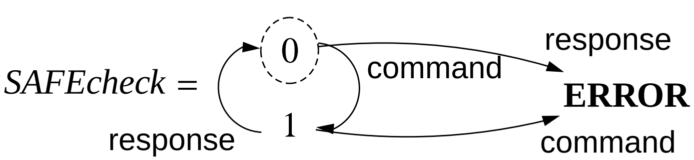
Example: $SAFE$ = command → response → $SAFE$

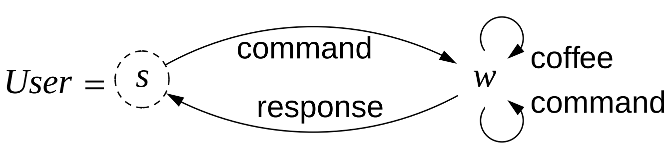


Checking Safe Access Sequences using ||

$SAFE$ = command → response → $SAFE$

Add **catch-all error state**:





User || *SAFEcheck* =

Safety

Ideally, a software system will be safe if it satisfies its specification.

— However, the specification may not guarantee safety.

Safety is a greater concern in a concurrent software system because the order of events is harder to control

Fundamental Safety Failure: An action by a process or thread that is ***intended to be atomic*** is breached by another process or thread.

- The code that implements the atomic action is called a **critical section**
- The breach of the atomic action may be unpredictable due to **race conditions**

Liveness

A **liveness** property asserts:

> "*no matter when we start to look*,
> something **good** will **eventually** happen"

Example: "Philosopher i cannot starve at the table."

- *No matter when we start to look, if philosopher i is at the table, he will eventually be eating*
- **This can be expressed in terms of traces:**

  Philosopher phil.i "cannot starve at the table" **iff** for every trace t and every position m such that t_m = phil.i.sitdown there is a position n with $n > m$ such that t_n = phil.i.eat.

Liveness

Ideally, a software system will be live if it satisfies its specification.

— However, the specification may not guarantee liveness.

Fundamental Liveness Failure:

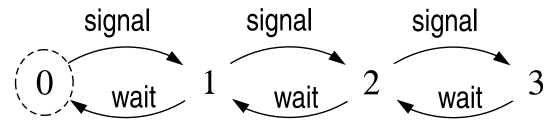
A process (thread) waits for an event that will never happen.

Examples:

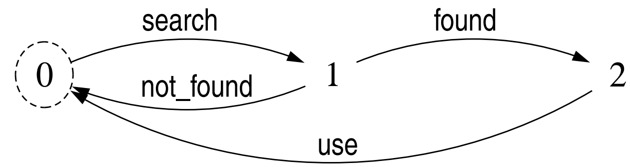
- Deadlock
- Missed signals
- Nested monitor lockouts
- Livelock
- Starvation
- Resource exhaustion
- Distributed failure

Reactive Choice

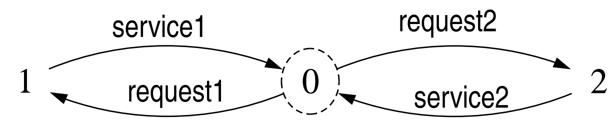
*Sema*3 =



Lookup =



Server =



Branching Transitions

A state of a process from which several transitions exist usually models one of the following:

– In this state, the process is prepared to **react** to different environmental stimuli

– In this state, the process **acts** by making a (non-deterministic) choice

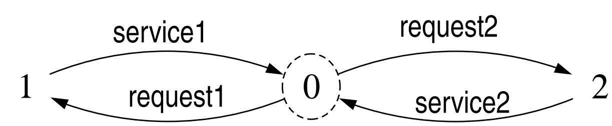
- non-determinism could be intended
- non-determinism could be the result of abstraction

LTSs do not differentiate between **action** and **reaction**!

Active Non-Deterministic Choice

$$
\begin{aligned}
Client1 &= \mathsf{request}\ 1 \rightarrow \mathsf{service}\ 1 \rightarrow \mathsf{sleep} \rightarrow Client1 \\
Client2 &= \mathsf{request}\ 2 \rightarrow \mathsf{service}\ 2 \rightarrow \mathsf{work} \rightarrow Client2 \\
Clients &= Client1 \parallel Client2 \\
System &= Clients \parallel Server
\end{aligned}
$$

Server =



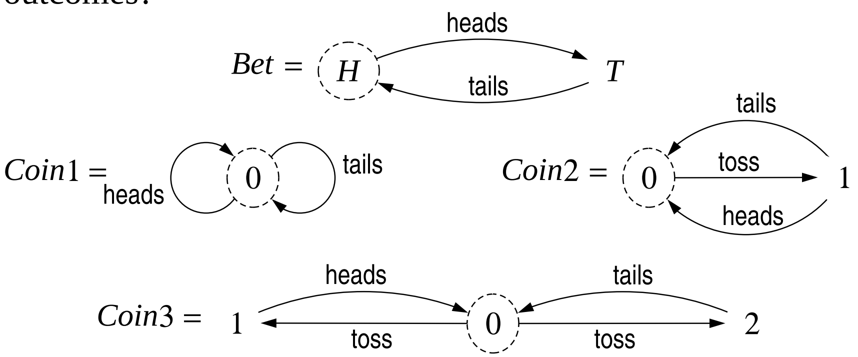
Concurrency is a good source of non-determinism!

Distribution is one of the best sources of non-determinism!

Modelling Real Non-Deterministic Choice

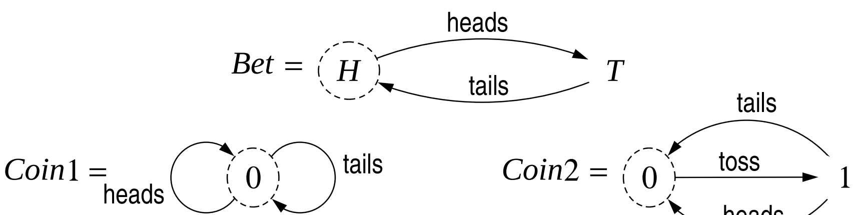
How should we model a process that repeatedly tosses a coin?

How should we model a process that bets on alternating outcomes?



Consider the compositions with *Bet*!

Betting Must Not Influence the Coin …

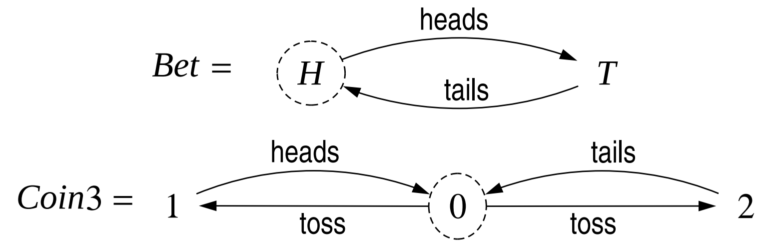


---

*Coin*1 || *Bet* =

*Coin*2 || *Bet* =

Betting Introduces Deadlock



*Coin*3 || *Bet* =

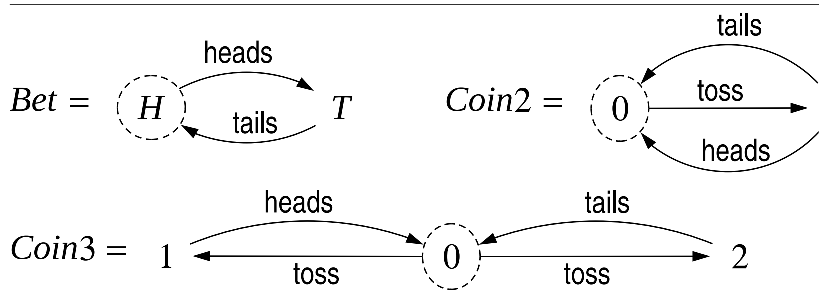
Non-Deterministic Choice, Traces, and Composition

*Coin*2 and *Coin*3 have the same trace set!

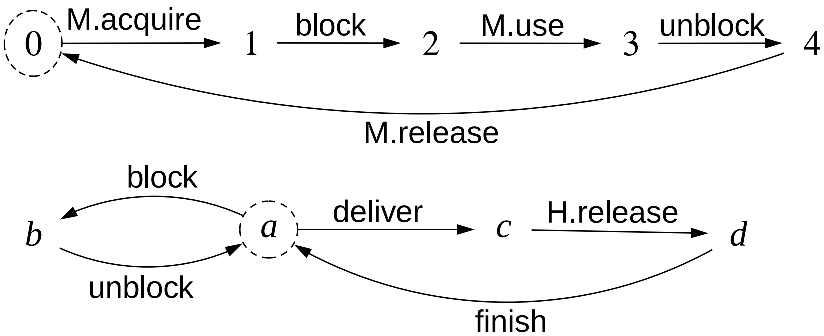
But, *Coin*2 || *Bet* and *Coin*3 || *Bet* have **different** trace sets!

⇒ Two LTSs P_1 and P_2 are **equivalent** iff for every LTS Q, the compositions $P_1 \parallel Q$ and $P_2 \parallel Q$ have the same trace set.

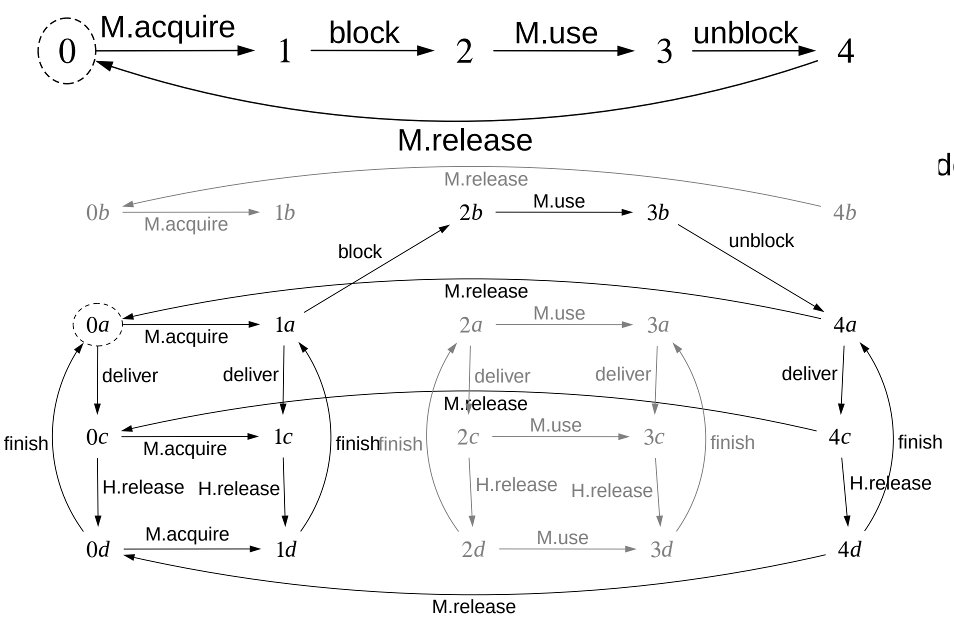
This is a *black-box* view: "No context enables distinction."



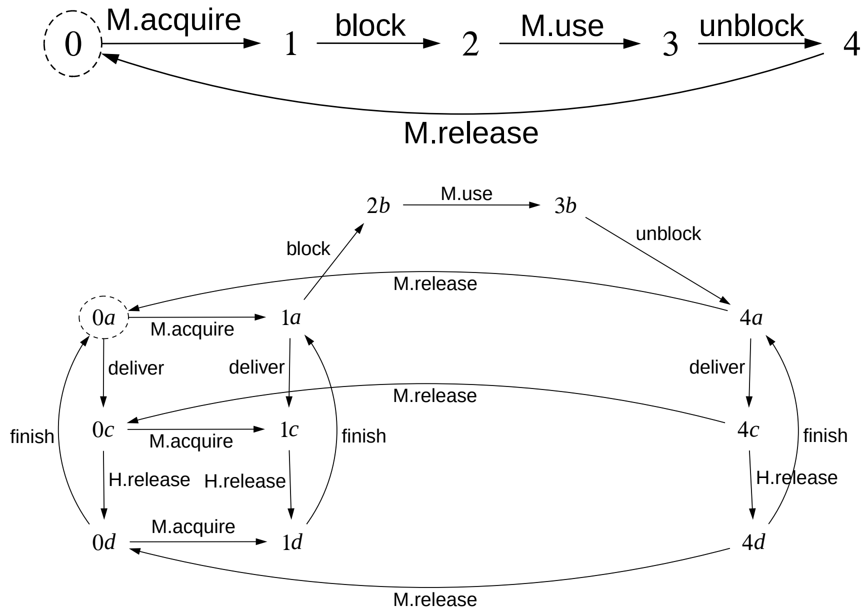
How Not to Model Signal Handling



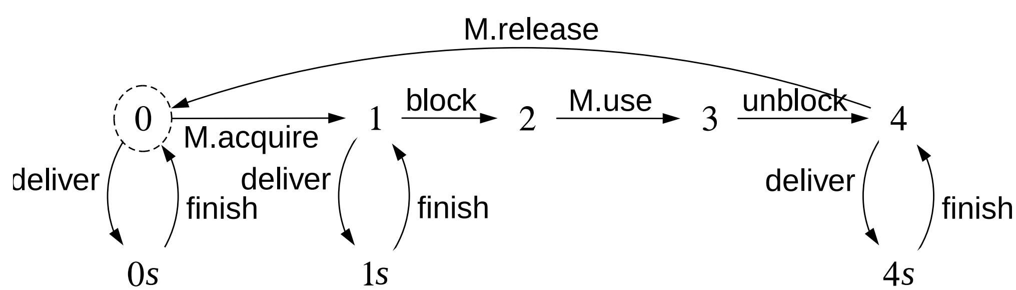
How Not to Model Signal Handling



How Not to Model Signal Handling

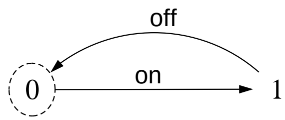


Modelling Signal Handling

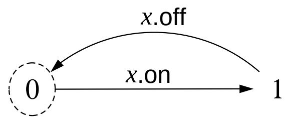


Labelling: Switches

Switch =



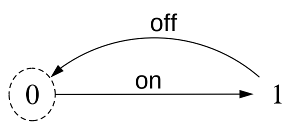
x:Switch =



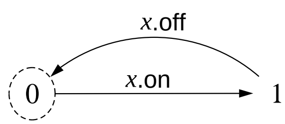
a:*Switch* || b:*Switch* =

Labelling: Switches

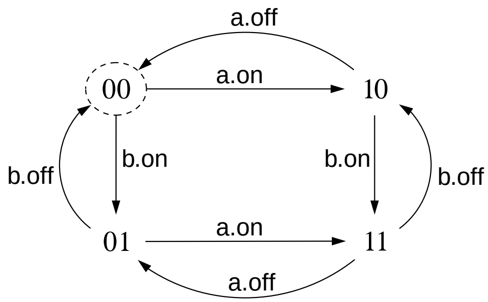
Switch =



x:Switch =



a:*Switch* || b:*Switch* =



Labelling and Sharing

Definition: For an action label set L and a label set A, we let $A{::}L$ denote the following set of **labelled actions**:

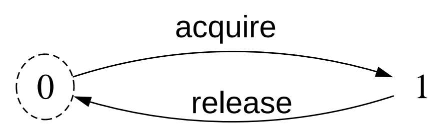
$$F{::}L = \{f : F; q : L \bullet f.q\}$$

For an LTSs $P = (S, s_0, L, \delta)$, we define:

- The LTS P **labelled** with a label f is $f{:}P = (S, s_0, \{f\}{::}L, \delta_f)$, where
  $$(x, a, y) \in \delta_f \iff \exists a_0 : L \bullet a = f.a_0 \wedge (x, a_0, y) \in \delta.$$
- The LTS P **shared** among a label set F is $F{::}P = (S, s_0, F{::}L, \delta_F)$, where
  $$(x, a, y) \in \delta_F \iff \exists f : F; a_0 : L \bullet a = f.a_0 \wedge (x, a_0, y) \in \delta.$$

Sharing: Resources

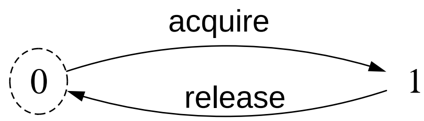
Resource =



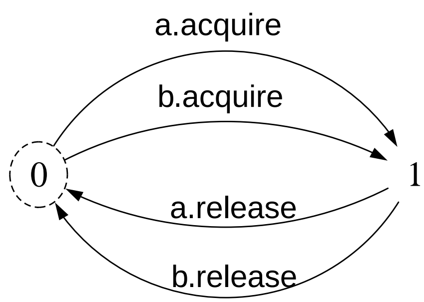
{a, b}::*Resource* =

Sharing: Resources

Resource =



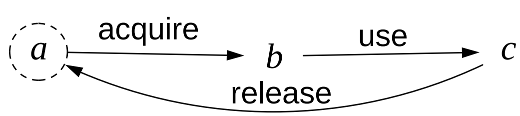
{a, b}::*Resource* =



Sharing Resources

ResSharing = a: User || b: User || {a, b}::*Resource*

User =

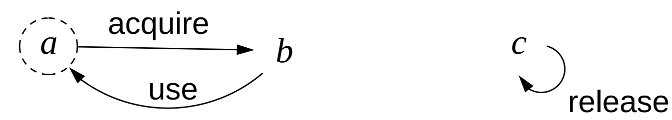


ResSharing =

Blocking Resources

ResBlocking = a:*Abuser* || b:*User* || {a, b}::*Resource*

Abuser =

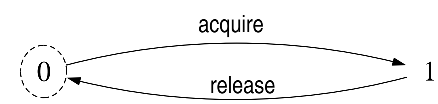


ResBlocking =

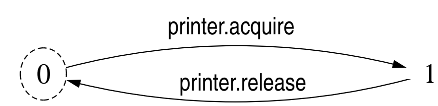
How many traces do these processes have?

Sharing a Labelled Resource

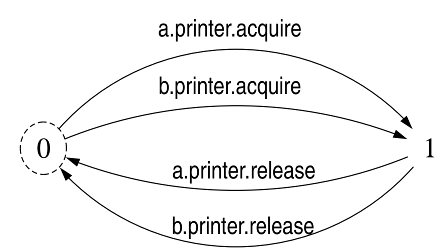
Resource =



printer:*Resource* =

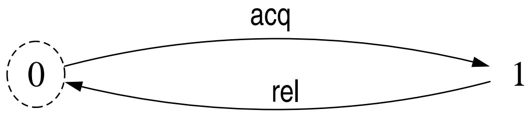


{a, b}::printer:*Resource* =

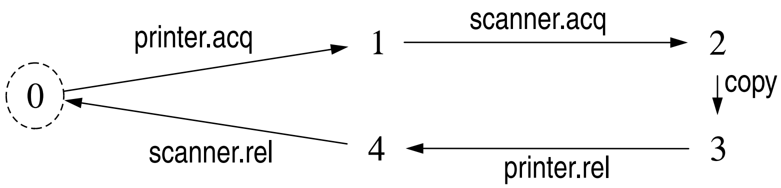


An Alternative Way of Defining Primitive Processes

Resource =



A =



Process Calculus Notation:

$$Resource = \mathsf{acq} \rightarrow \mathsf{rel} \rightarrow Resource$$

$$A = \mathsf{printer.acq} \rightarrow \mathsf{scanner.acq} \rightarrow \mathsf{copy} \rightarrow \mathsf{printer.rel} \rightarrow \mathsf{scanner.rel} \rightarrow A$$

Sharing Two Resources

$$Resource = \mathsf{acq} \rightarrow \mathsf{rel} \rightarrow Resource$$

$$A = \mathsf{pr.acq} \rightarrow \mathsf{sc.acq} \rightarrow \mathsf{copy} \rightarrow \mathsf{pr.rel} \rightarrow \mathsf{sc.rel} \rightarrow A$$

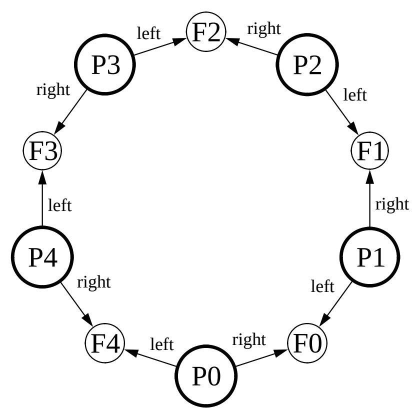
$$B = \mathsf{sc.acq} \rightarrow \mathsf{pr.acq} \rightarrow \mathsf{copy} \rightarrow \mathsf{sc.rel} \rightarrow \mathsf{pr.rel} \rightarrow B$$

$$Sys = \mathsf{a}{:}A \parallel \{\mathsf{a},\mathsf{b}\}{::}\mathsf{pr}{:}Resource \parallel \{\mathsf{a},\mathsf{b}\}{::}\mathsf{sc}{:}Resource \parallel \mathsf{b}{:}B$$

The Dining Philosophers

- Five philosophers live together in a house.
- The live of a philosopher essentially consists of alternating phases of thinking and eating.
- For eating, there is a round table with five seats and a large bowl of spaghetti on it; between adjacent seats there is always one fork.
- Each philosopher needs two forks in order to be able to eat.
- When hungry, each philosopher will sit down on a free chair, take up the fork to his left, take up the fork to his right, eat, put down the forks, and leave for more thinking.
- *Is it possible that the philosophers all starve to death?*

The Dining Philosophers



$$Fork = \mathsf{get} \rightarrow \mathsf{put} \rightarrow Fork$$

$$Phil = \mathsf{sitdown} \rightarrow \mathsf{right.get} \rightarrow \mathsf{left.get} \rightarrow \mathsf{eat} \rightarrow \mathsf{left.put} \rightarrow \mathsf{right.put} \rightarrow \mathsf{arise} \rightarrow Phil$$

Let $N = 5$

Let $succ_N(i) = (i + 1)\%N$

$$Diners = \Big\Vert_{i=0}^{N-1} \left(\mathsf{phil}{:}i{:}Phil \parallel \{\mathsf{phil}.i.\mathsf{right}, \mathsf{phil}.succ_N(i).\mathsf{left}\}{::}Fork\right)$$

Model-Checking the Dining Philosophers Using LTSA

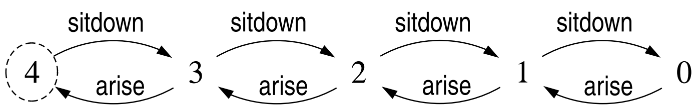
```
PHIL = (sitdown->right.get
  ->left.get->eat->left.put
  ->right.put->arise->PHIL).

FORK = (get -> put -> FORK).

||DINERS(N=5)=
   forall [i:0..N-1]
   (phil[i]:PHIL
   ||{phil[i].right,
     phil[(i+1)%N].left}::FORK).
```

```
Trace to DEADLOCK:
  phil.0.sitdown
  phil.0.right.get
  phil.1.sitdown
  phil.1.right.get
  phil.2.sitdown
  phil.2.right.get
  phil.3.sitdown
  phil.3.right.get
  phil.4.sitdown
  phil.4.right.get
```


Solutions to the Dining Philosophers Problem

Original solution: Introduce a **butler** who restricts the maximum number of sitting philosophers to 4.

Butler =



The butler is a counting semaphore!

Some other solutions:

- Have some philosophers pick up the left fork first.
- Make picking up both forks atomic.
- Have all philosophers decide randomly which fork to pick up, and give priority to “hungrier” neighbours.