# Behavioural Equivalences and Abstraction Techniques

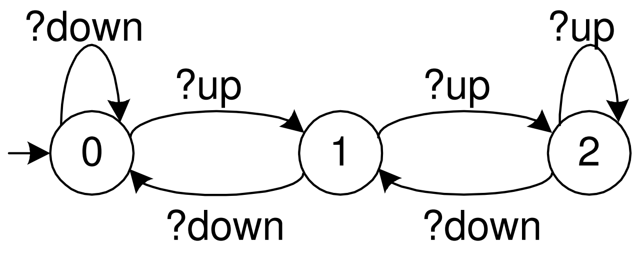
Natalia Sidorova

## Part 1: Behavioural Equivalences

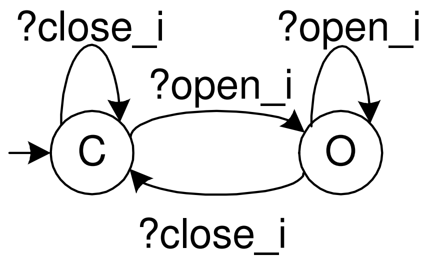
# The elevator example once more

How to compare this elevator model with some other?

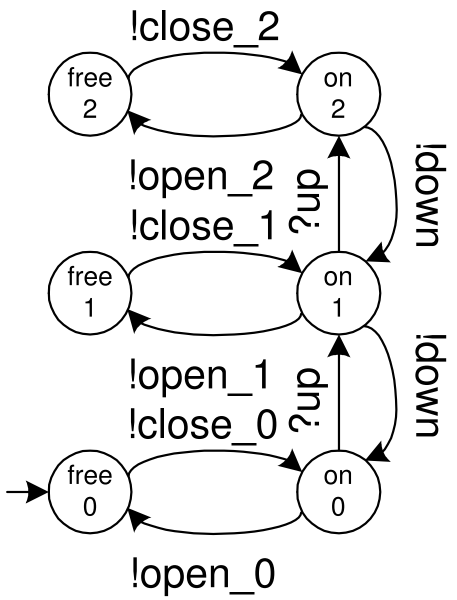
The cabin



The ith door



The controller



## System behaviour

How to compare different models?

Models can be described using state machines, known as Labelled Transition Systems.
A labelled transition system $T$ consists of:

- a set $S$ of states,
- a set $L$ of actions
- a set $\rightarrow$ of transitions of the form $s \xrightarrow{a} t$ with $s, t \in S$ and $a \in L$ or $a = \tau$,
- an initial state $s_0 \in S$.

We also write $T = (S, L, \rightarrow, s_0)$.

## From your model to LTS

Thus, you can build a single LTS for the Petri net, communicating finite state machines, etc. model you have.
Usually, you don’t do it manually: tools do it for you.

From now on we assume that the systems are given by LTSs and we compare them.

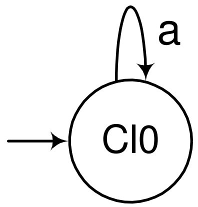
## The wish-list

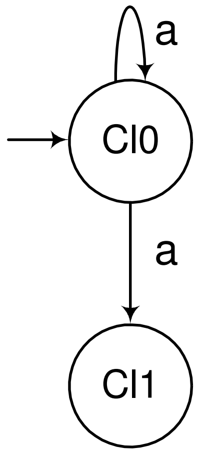
1. Behavioural equivalence should be a reflexive, symmetric, and transitive relation.
2. Processes that may terminate (deadlock) should not be equivalent to processes that may not terminate (deadlock).
3. If a component $Q$ of $P$ is replaced by an equivalent component $Q_0$ yielding $P_0$, then $P$ and $P_0$ should also be equivalent.
4. Two processes should be equivalent iff they satisfy exactly the same properties expressible in a nice modal or temporal logic.
5. Equivalence should abstract from silent actions.

# A first candidate: Trace equivalence

A trace of a process $P$ is a sequence $\sigma$ of actions such that $s_0 \xrightarrow{\sigma} s$ for some state $s$.

Two processes are trace-equivalent if they have the same traces. We will write $P \equiv_{tr} Q$.
This notion satisfies (1) but not (2). The following two clocks are trace-equivalent:





## 2: Completed-trace equivalence

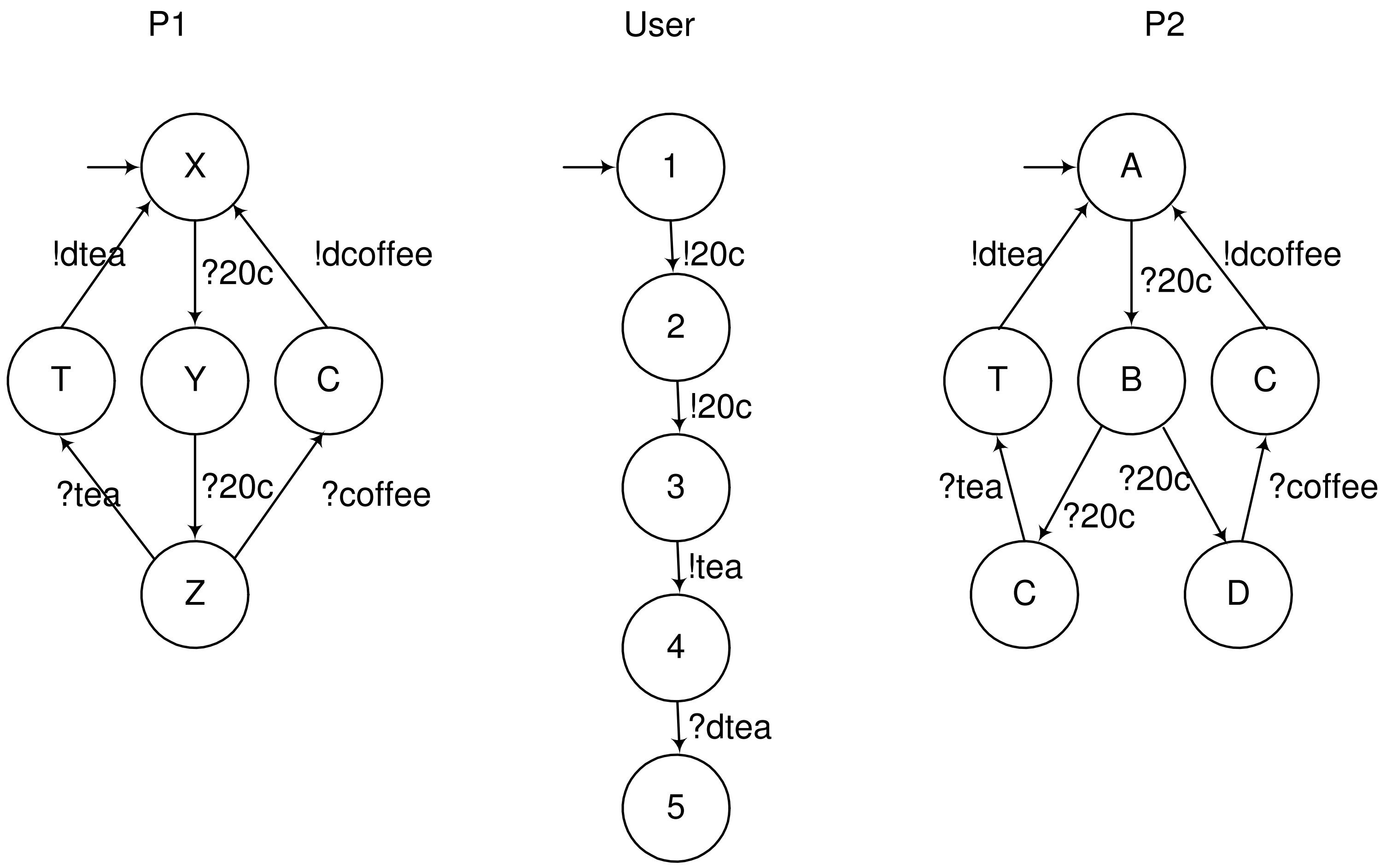
A completed trace of a process $P$ is a sequence $\sigma$ of actions such that $s_0 \xrightarrow{\sigma} s$ for some state $s$ at which no action is enabled.

Two processes $P$ and $Q$ are completed-trace equivalent if they are trace equivalent and have the same completed traces.

This notion satisfies (1) and (2), but not (3).

# 2: Completed-trace equivalence

Compare $P1 \| User$ with $P2 \| User$



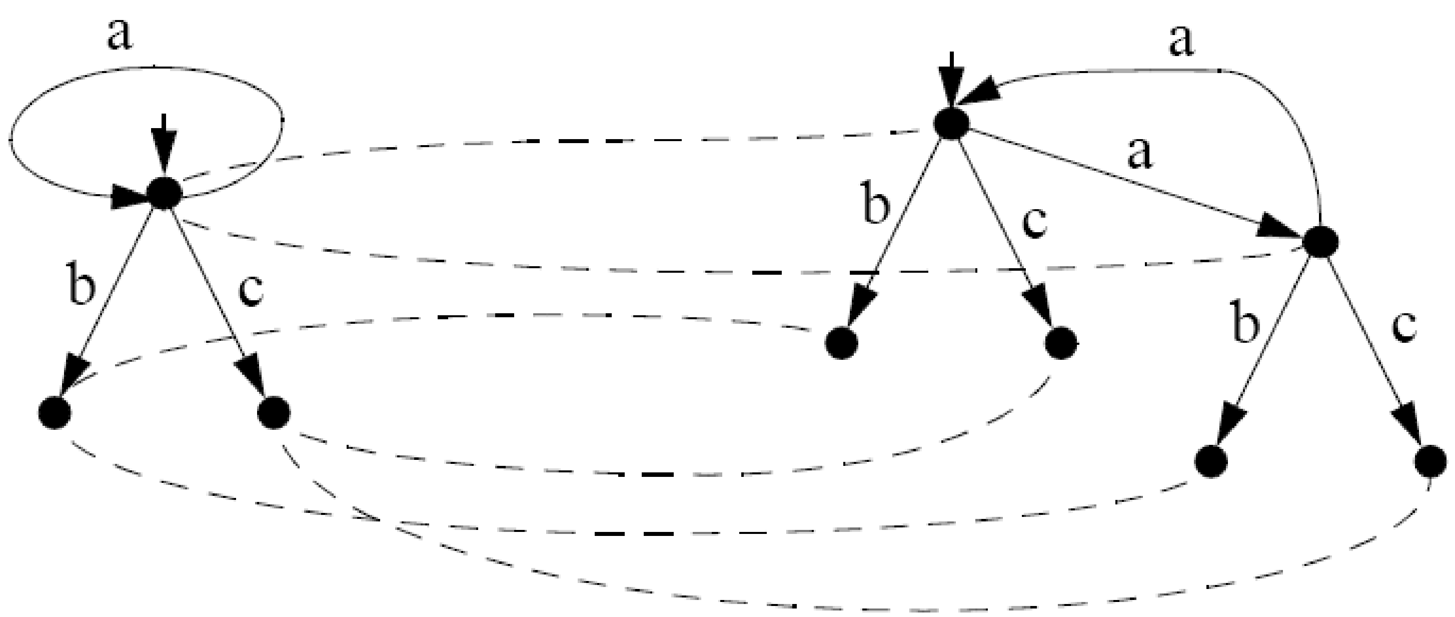
## 3: Bisimulation equivalence

A binary relation $B$ between states is a bisimulation provided that
whenever $(s, q) \in B$ and $a \in A$,

1. if $s \xrightarrow{a} s'$ then $q \xrightarrow{a} q'$ for some $q'$ such that $(s', q') \in B$, and
2. if $q \xrightarrow{a} q'$ then $s \xrightarrow{a} s'$ for some $s'$ such that $(s', q') \in B$.

Two processes $P$ and $Q$ with initial states $s_0$, $q_0$ are bisimilar if there is a bisimulation relation $B$ such that $(s_0, q_0) \in B$.
We write then $P \sim Q$.

## Example



## Game interpretation

Board: Transition systems of $P$ and $Q$.

Material: Two (identical) pebbles, initially on the initial states $s_0$ and $q_0$.

Players: R (refuter) and V (verifier), R and V take turns, R moves first.

R-move: Choose any of the two pebbles. Take any transition.

V -move: Choose the other pebble. Take any transition having the same label than the one chosen by R.

## Game interpretation (2)

R wins if: V cannot reply to his last move.

V wins if: R cannot move or the game goes on forever.
(i.e., a draw counts as a win for V ).

Theorem: R can force a win iff $P$ and $Q$ are not bisimilar (R refutes).
V can force a win iff $P$ and $Q$ are bisimilar (V verifies).

## Bisimilarity is an equivalence relation

Theorem: For all processes $E$, $E \sim E$.

Theorem: For all processes $E$ and $F$, if $E \sim F$ then $F \sim E$.

Theorem: For all processes $E, F$ and $G$, if $E \sim F$ and $F \sim G$, then $E \sim G$.

## Property preservation

Theorem: Let $E \equiv_{tr} F$. Then $E \models \phi$ iff $F \models \phi$, where $\phi$ is an $LTL$-property.

Theorem: Let $E \sim F$. Then $E \models \phi$ iff $F \models \phi$, where $\phi$ is a $CTL^*$-property.

## Equiv.-preserving transf. and Software Design

Design can start with a very abstract specification, representing the requirements.

Then, using equivalence-preserving transformations, this specification can be gradually transformed into an implementation-oriented specification.

This one can be transformed into code.

Maintenance may require to replace some components with others, while maintaining the same behaviour.

You may use behaviour-preserving transformations to make model checking, testing, etc. possible.

## Part 2: Abstraction Techniques

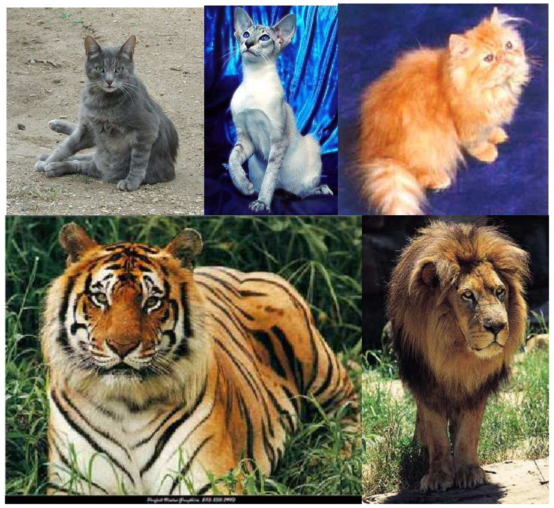
## Abstractions in our life

Cat: $\longleftarrow$

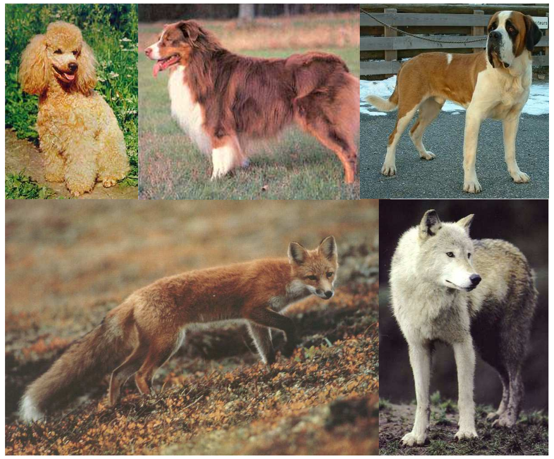
Dog: $\longleftarrow$

## Abstractions in our life

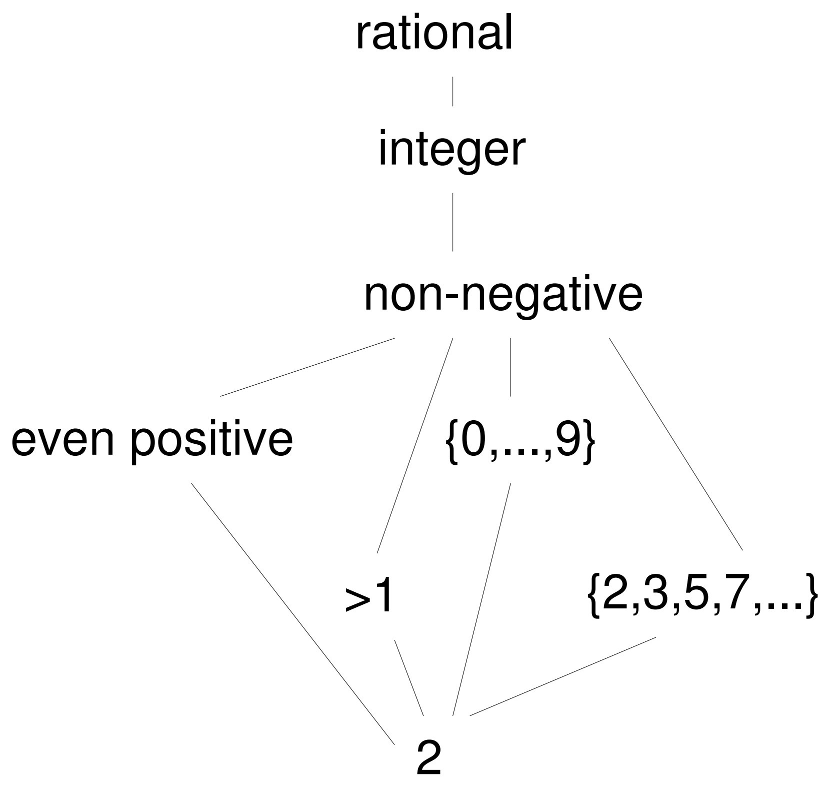
Feline: $\longleftarrow$



Canine: $\longleftarrow$



# Abstractions in computing



# Why abstractions?

- To cope with complexity
- To describe underspecified or uncertain things
  - Open models: under-specified or abstracted environment
  - A model can describe a set of implementations
    - flexibility for implementor
    - verification of a set of implementations
- Connecting interfaces of components makes global model more abstract through hiding internal computation
  - Weak (bi)simulations in process algebras [Milner'89].

## Abstract to verify

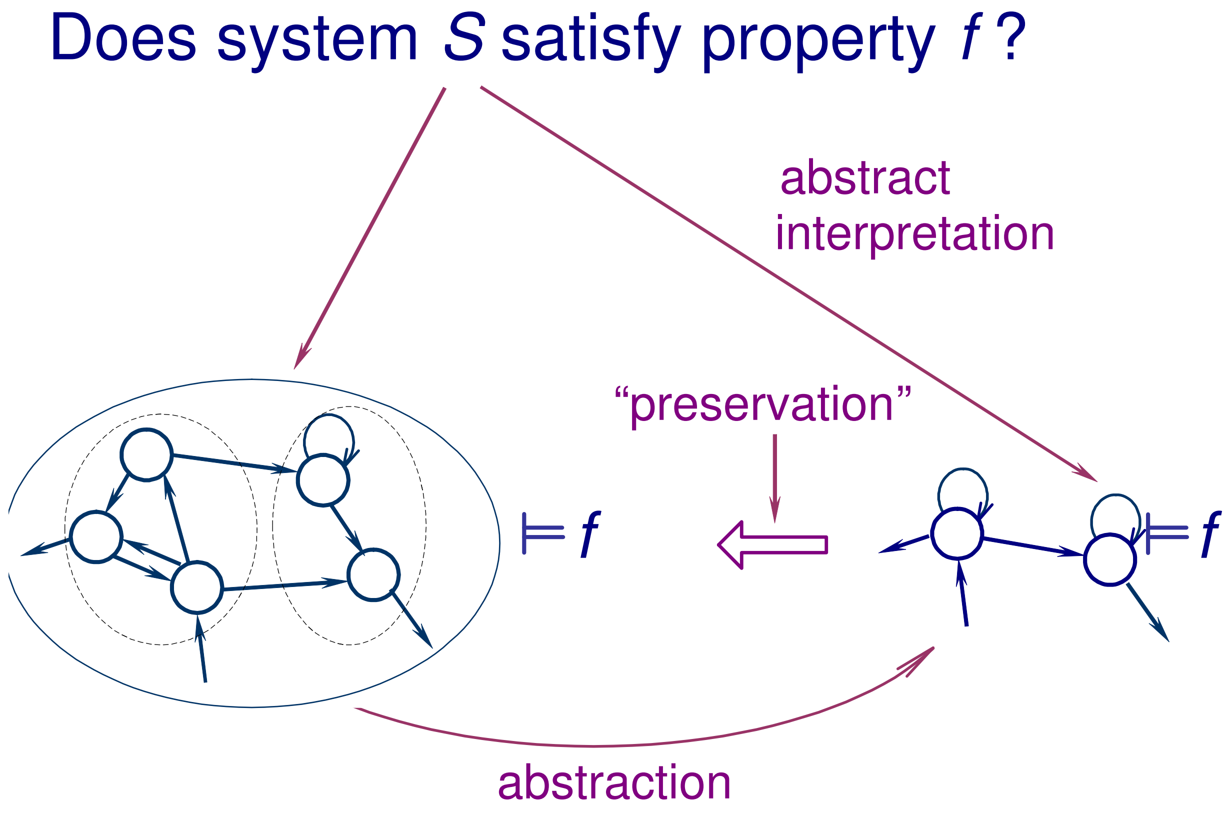
You can

- try to keep the behaviour of the system unchanged,
- overapproximate the behaviour of the system, or
- underapproximate the behaviour of the system

Potential problems:

- False negatives: your analyser will report the property violation while there is no any on the concrete system
- False positives: your analyser will report that the system is correct while it is wrong

# Abstraction



# Bisimulation and trace equivalence

Preservation results:

Theorem: Let $E \equiv_{tr} F$. Then $E \models \phi$ iff $F \models \phi$, where $\phi$ is an $LTL$-property.

Theorem: Let $E \sim F$. Then $E \models \phi$ iff $F \models \phi$, where $\phi$ is a $CTL^*$-property.

Bisimulation is too much to require;
trace equivalence is difficult to prove.

We might wish the preservation result in one direction only: from the abstract to the concrete system.

## Simulation preorder

Let $T_1 = (S_1, L_1, \rightarrow_1, s_{0_1})$ and $T_2 = (S_2, L_2, \rightarrow_2, s_{0_2})$ be two transition systems.

A binary relation $H \subseteq S_1 \times S_2$ is a simulation iff $(s_{0_1}, s_{0_2}) \in H$ and for every $(s_1, s_2) \in H$:

1. $s_1$ and $s_2$ satisfy the same atomic propositions
2. if $s_1 \longrightarrow_1 s_1'$ then $s_2 \longrightarrow_2 s_2'$ for some $s_2' \in S_2$ such that $(s_1', s_2') \in H$.

We write $s_1 \leq s_2$ when $(s_1, s_2) \in H$.

We write $T_1 \leq T_2$ when there exists a simulation $H$.

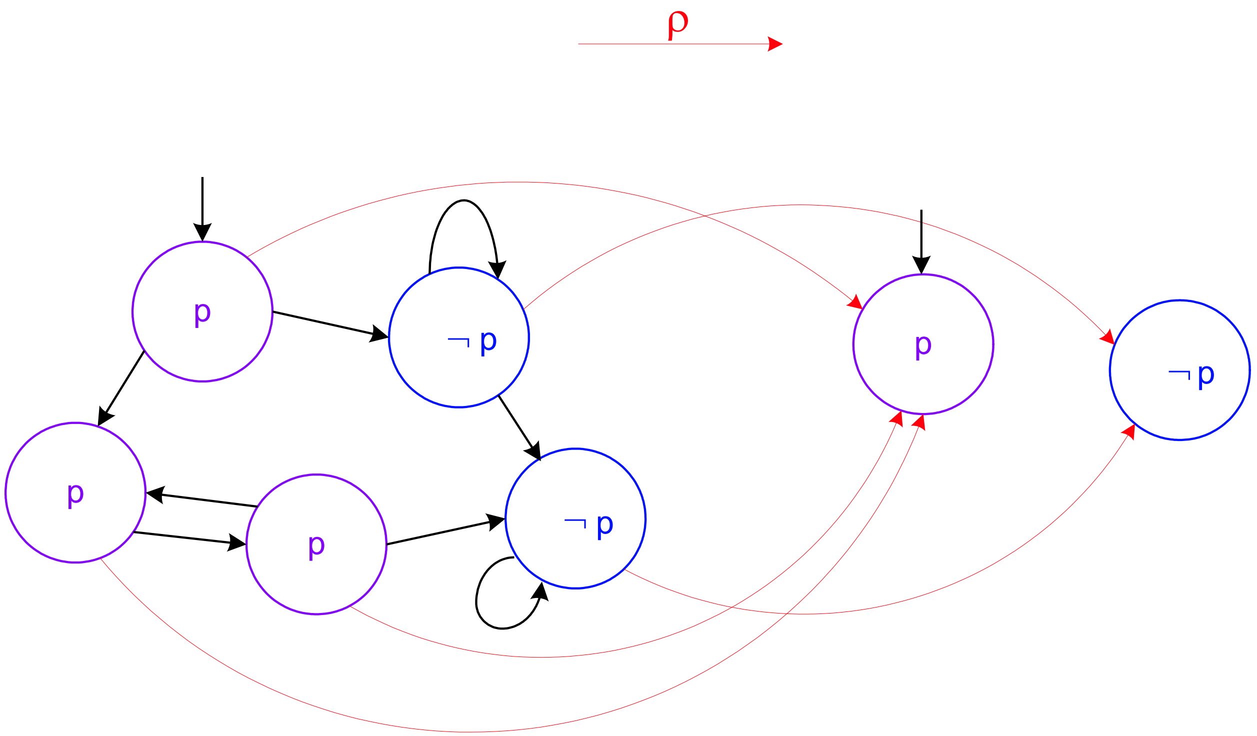
## Property preservation

If $T_2 \geq T_1$ then for every LTL formula $\phi$, $T_2 \models \phi$ implies $T_1 \models \phi$.

Thus our goal is to build for a given system $T$ its abstract version $T^\alpha$ such that $T^\alpha \geq T$.

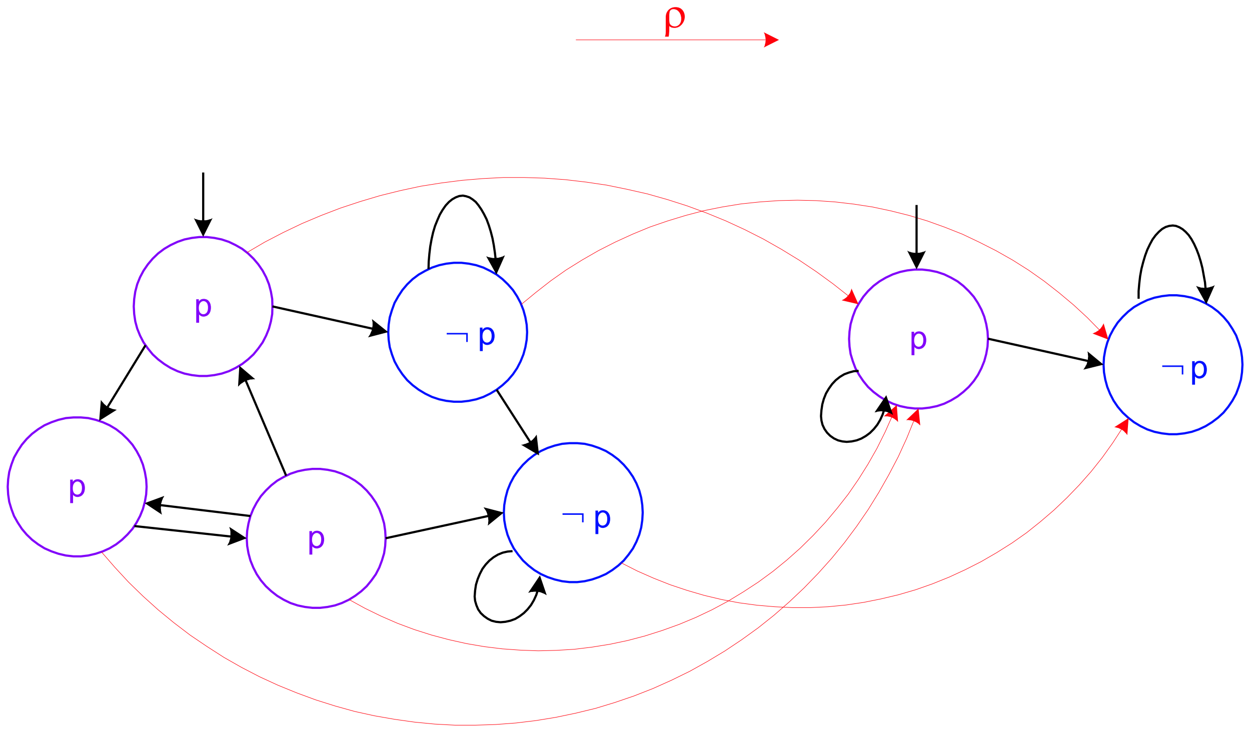
## Basic idea

Given a transition system $T$,
define some description function $\rho : S \longrightarrow S^{\alpha}$:



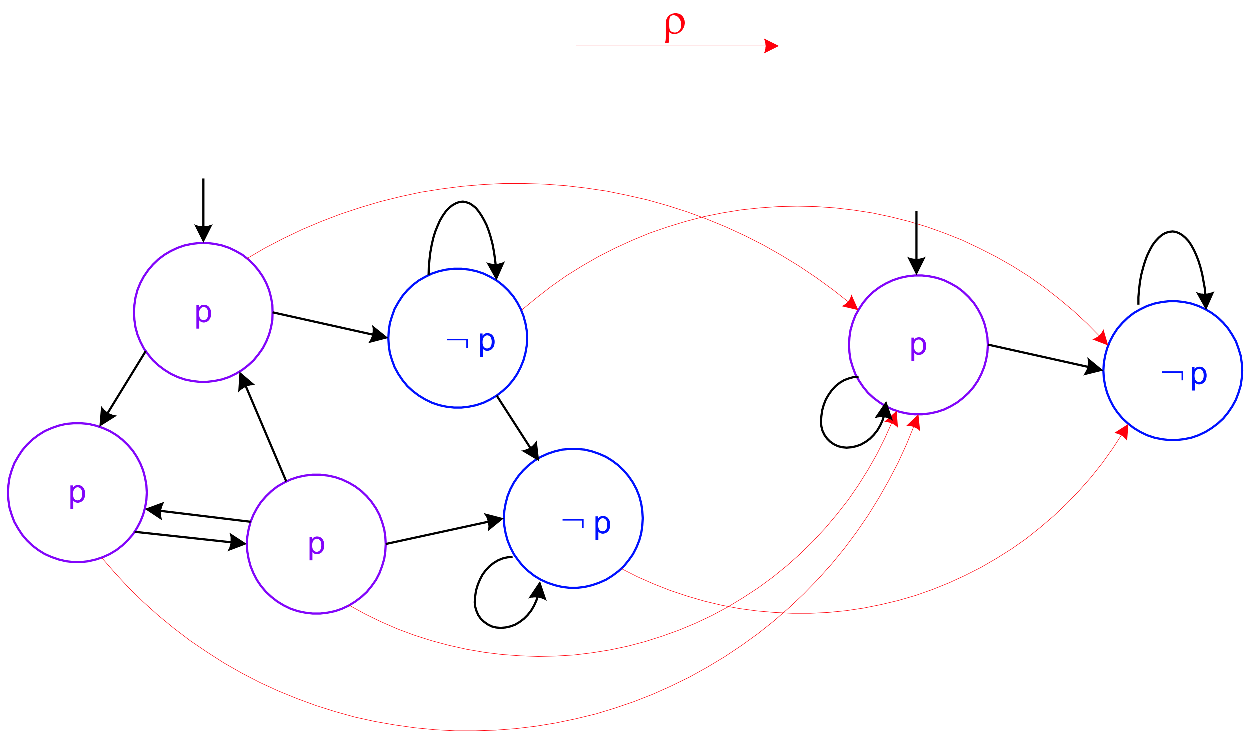
## Basic idea

Now define the abstract transition relation in such a way that whenever $s \longrightarrow s'$, we have $\rho(s) \longrightarrow \rho(s')$.



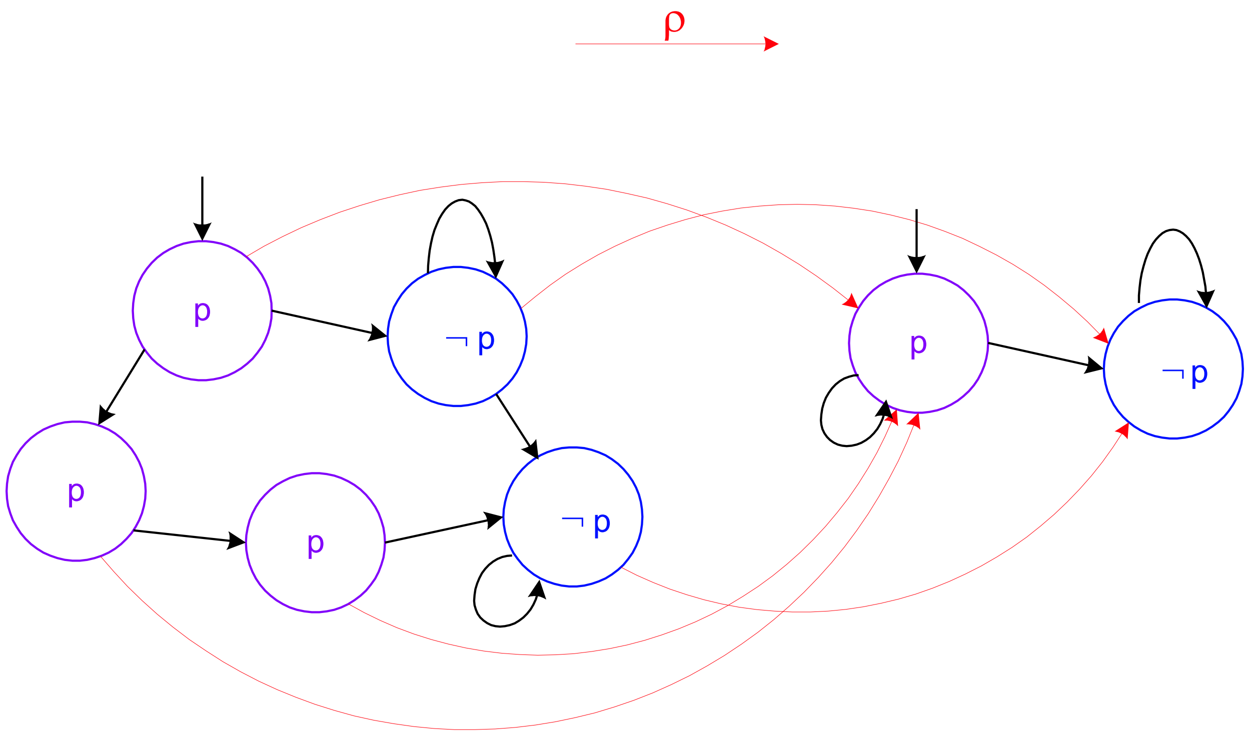
## Basic idea

Sometimes we get even bisimilar systems this way



. . . but not always.

# Another example



Can you formulate a property that holds on $T$ but not on $T^{\alpha}$ (thus, you get a false negative when you verify this property on $T^{\alpha}$)?

## Data abstractions

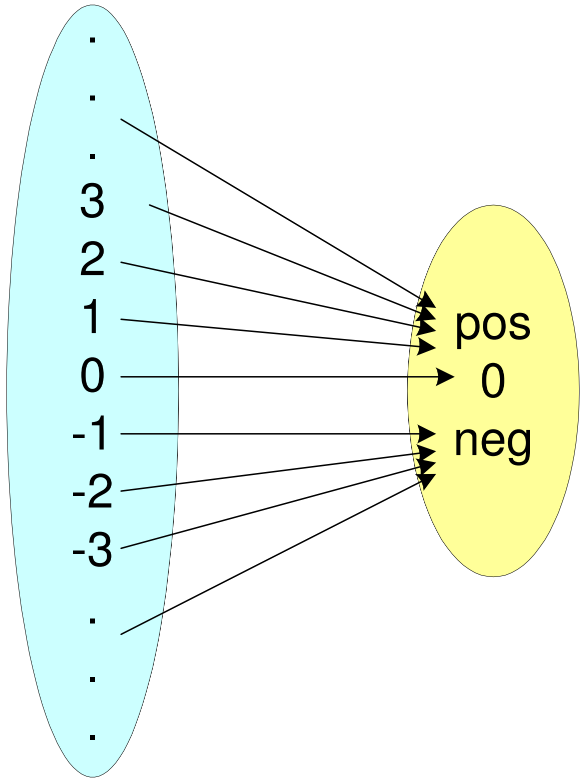
It is not always handy to define $\rho$ on states.

Idea: Replace data values by their descriptions (abstract values) and "mimic" operations on data with overapproximations. In general, abstract operations can be nondeterministic.

Then every LTL property checked to be true on the abstract system holds for the concrete system as well.

# An example of data abstraction

Consider the abstraction of integers into their signs:



$$\mathbf{pos} +^{\alpha} \mathbf{pos} = \{\mathbf{pos}\}$$
$$\mathbf{neg} +^{\alpha} \mathbf{neg} = \{\mathbf{neg}\}$$
$$\mathbf{pos} +^{\alpha} \mathbf{neg} = \{\mathbf{pos}, 0, \mathbf{neg}\}$$
$$\mathbf{neg} +^{\alpha} \mathbf{pos} = \{\mathbf{pos}, 0, \mathbf{neg}\}$$
$$\mathbf{pos} +^{\alpha} \mathbf{0} = \{\mathbf{pos}\}$$
$$\mathbf{0} +^{\alpha} \mathbf{pos} = \{\mathbf{pos}\}$$
$$\mathbf{neg} +^{\alpha} \mathbf{0} = \{\mathbf{pos}\}$$
$$\mathbf{0} +^{\alpha} \mathbf{neg} = \{\mathbf{neg}\}$$
$$\mathbf{0} +^{\alpha} \mathbf{0} = \{0\}$$

## Safety statement

The formal requirement for the abstraction of a binary function is:

$$\forall x \in D_x, y \in D_y \quad \exists z \in f_{abs}(\rho(x), \rho(y)) : \rho(f_{conc}(x, y)) = z$$

where $D_x, D_y$ are corresponding data domains.

## In practice

We will use nondeterminism to define abstract functions. For example,

```
#define sum(x,y,z) \
  if \
  :: x == pos && y == pos -> z = pos \
  :: x == pos && y == neg -> \
    if \
    :: z = pos \
    :: z = neg \
    :: z = 0 \
    fi \
  :: . . .
. . . . . . . . . . . .
  fi
```

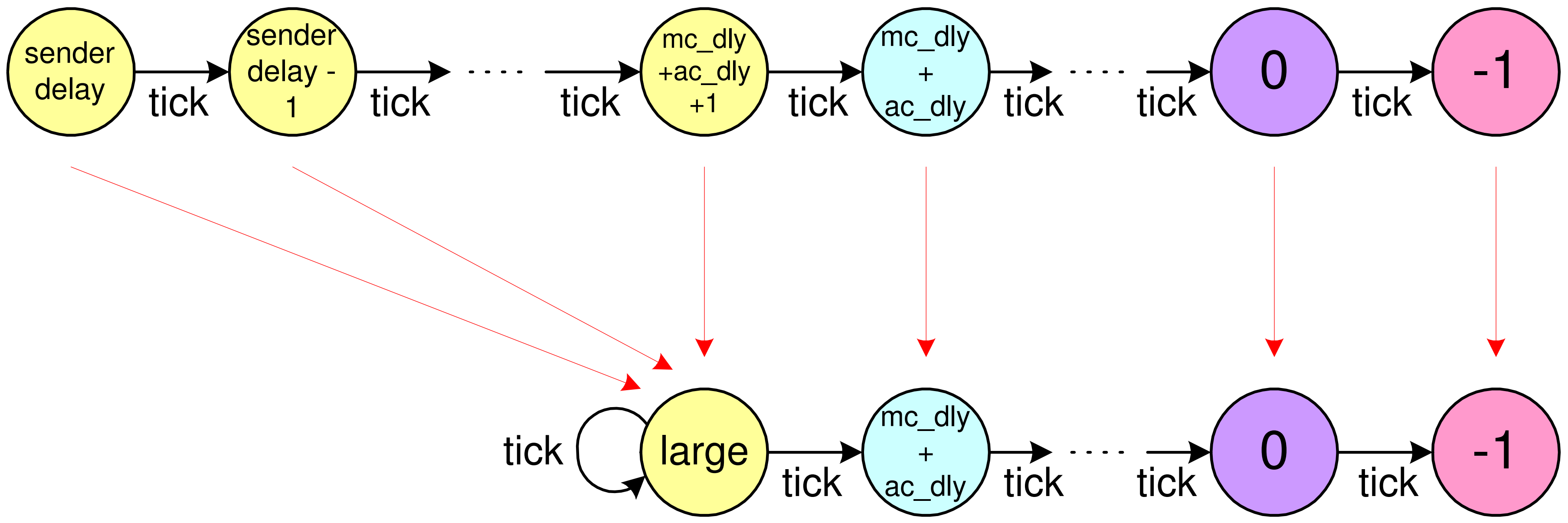
# Timer abstraction for PAR

Goal: Given message and acknowledgment channel delays mc_delay and ac_delay, prove that the protocol works correctly for any sender_delay such that sender_delay > mc_delay + ac_delay.

Problem: We can (in theory) perform checks for any value of sender_delay, but we cannot perform an infinite number of checks.

Solution: Use a timer abstraction for the timer at the sender side.

# Timer abstraction



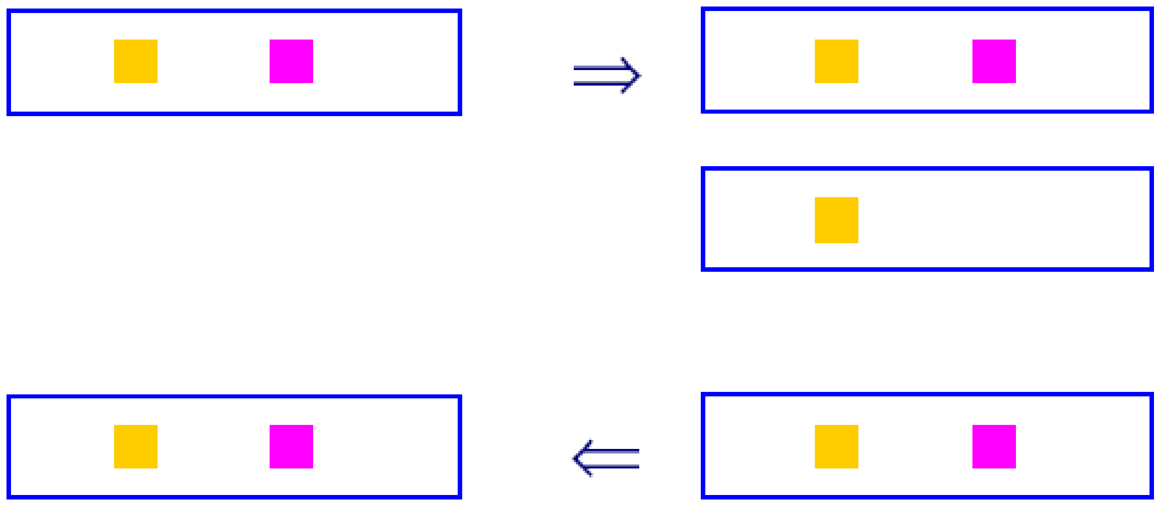
## Timer abstraction

```
typedef abstract_timer
{ short val = OFF;
bit plus = false;
}

#define abstract_set(tmr,value) \
tmr.val = value; tmr.plus = true
. . . . . .
#define abstract_tick(tmr) \
if \
:: tmr.plus==true -> skip; \
:: tmr.val != OFF -> tmr.val = tmr.val - 1; tmr.plus=false; \
:: tmr.val == OFF -> skip; \
fi
```

# PAR verification

Goal: to prove that PAR works correctly (when the delays meet the constraint) in any environment.
"Correctly" means "If the sender environment is transmitting a sequence $l$ of messages and the first element of $l$ arrives at least once at the receiver, then the sequence of elements that the receiving client gets ("the output list"), forms a prefix of $l$".



## Data independence

Problem: Any environment means that the environment is able to send any signals from any finite alphabet in any order, which requires an infinite number of checks when handled directly.

Solution: use data independence.

## Data independence

Intuitively, data independence means that the system just moves data around without looking at it. In particular, there are no conditionals that depend on the data values. All data values come from data sources.

More formally, a system is data-independent w.r.t. a variable if

- it only assigns values to/from that variable, or
- tests the identity of that variable;
- no other operation on this variable is allowed.

[Wolper 1986]

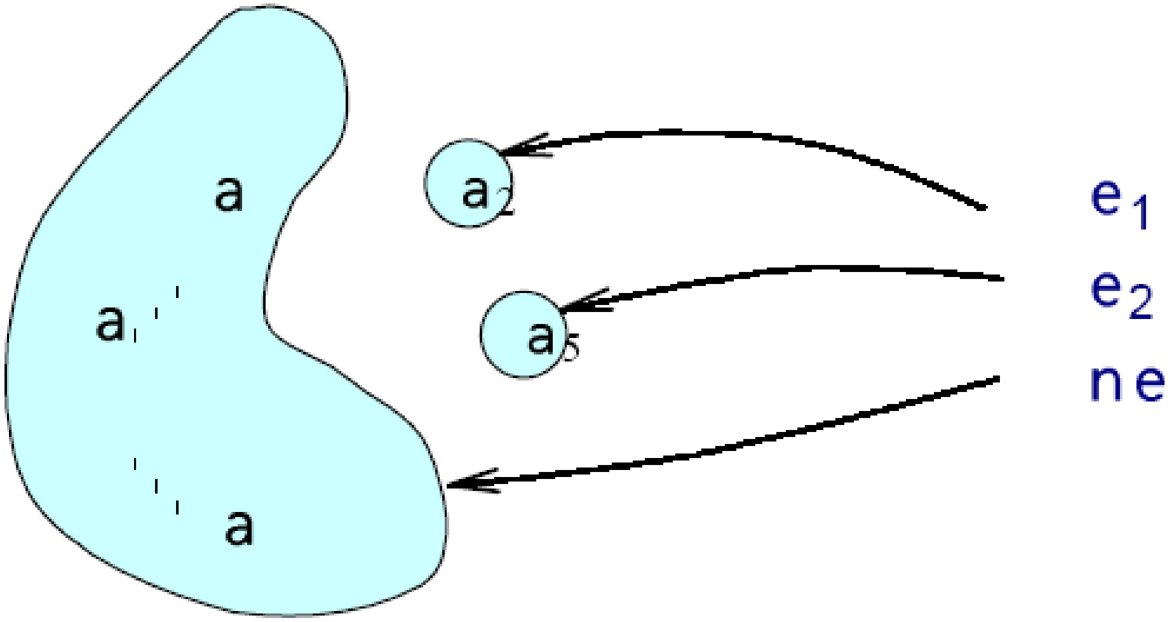
# PAR property

It can be shown that the prefix property holds if for any list of non-repeating naturals the following properties hold:

1. for any two values $e_1$ and $e_2$ on positions $i$ and $j$ resp. in the input list, with $i < j$, either $e_2$ does not occur in the output list, or $e_1$ and $e_2$ occur in the output list on positions $i'$, $j'$ resp., with $i' < j'$.
2. For any two values $e_1$ and $e_2$ on positions $i'$ and $j'$ resp. in the output list, with $i' < j'$, $e_1$ and $e_2$ occur on positions $i$ and $j$ resp. in the input list with $i < j$.

## Environment abstraction for PAR

That gives the following idea of an abstraction:
we distinguish two natural numbers $n1, n2$, which are abstracted into e1, e2 respectively, while all the other naturals are non-distinguishable and they are abstracted into an abstract element ne.



## Environment abstraction for PAR

Thus the abstract version of PAR can transmit messages e1, e2, ne.

The sender environment sends sequences ne*e1ne*e2ne*.

The receiver environment expects to receive sequences ne*e1ne*e2ne* and reports an error in case it gets an unexpected message.

## Assignement 2

Abstract the timer at the sender side in your Promela model of PAR, define an abstract environment, formulate the correctness property first in the textual form, then formalise it, check it with Spin for several timer settings.

## Assignement 2

List abstraction. We abstract naturals p1, p2 into e1, e2 resp., while all the other naturals are abstracted into ne.
An arbitrary list of naturals is represented by its abstracted head element, an abstract representation of the whole list, which is of the form epsl, e1l, e2l, e1e2l, e2e1l, error, and, the information whether it is an empty or a one-element list. Here, error is an abstraction for an "incorrect" list, i.e. a list with duplicated elements; epsl represents correct lists where no e1 and no e2 occur, e1l, e2l are representations of correct lists where only e1 (e2 resp.) occurs; e1e2l, e2e1l represent lists where both e1 and e2 occur, in the corresponding orders. Define a data type for abstract lists in Promela and abstract versions the operation of removing the head element from the list.