Semantics of processes: behavioural equivalences and proof techniques

Davide Sangiorgi

University of Bologna

Email: Davide.Sangiorgi@cs.unibo.it
http://www.cs.unibo.it/~sangio/

July 17, 2006

The semantics of processes:

- usually **operational**: (Labelled Transitions Systems, behavioural equivalences)
- alternative approach could be the **denotational** one: a structure-preserving function would map processes into elements of a given semantic domain.
  Problem: it has often proved very hard to find appropriate semantic domains for these languages

Thse lectures: An introduction to the meaning of behavioural equivalence

We especially discuss bisimulation, as an instance of the co-induction proof method

Outline

- From functions to processes
- Bisimulation
- Induction and Co-induction
- Weak bisimulation
- Other equivalences: failures, testing, trace ...

From processes to functions

Processes?

We can think of sequential computations as mathematical objects, namely **functions**.

Concurrent program are not functions, but **processes**. But what is a process?

No universally-accepted mathematical answer.

Hence we do not find in mathematics tools/concepts for the denotational semantics of concurrent languages, at least not as successful as those for the sequential ones.

Processes are not functions

A sequential imperative language can be viewed as a function from states to states.

These two programs denote the same function from states to states:

$$\mathtt{X} := 2 \qquad \text{and} \qquad \mathtt{X} := 1;\ \mathtt{X} := \mathtt{X} + 1$$

But now take a context with parallelism, such as $[\cdot] \mid \mathtt{X} := 2$. The program

$$\mathtt{X} := 2 \mid \mathtt{X} := 2$$

always terminates with $\mathtt{X} = 2$. This is not true (why?) for

$$(\mathtt{X} := 1;\ \mathtt{X} := \mathtt{X} + 1) \mid \mathtt{X} := 2$$

Therefore: Viewing processes as functions gives us a notion of equivalence that is not a **congruence**. In other words, such a semantics of processes as functions would not be **compositional**.

Furthermore:

- A concurrent program may not terminate, and yet perform meaningful computations (examples: an operating system, the controllers of a nuclear station or of a railway system).
  In sequential languages programs that do not terminate are undesirable; they are 'wrong'.

- The behaviour of a concurrent program can be non-deterministic.
  Example:
  $$(\; \mathtt{X} := 1; \; \mathtt{X} := \; \mathtt{X} + 1 \,) \mid \; \mathtt{X} := 2$$
  In a functional approach, non-determinism can be dealt with using powersets and powerdomains.

This works for pure non-determinism, as in $\lambda x. \, (3 \oplus 5)$

But not for parallelism.

What is a process?

When are two processes behaviourally equivalent?

These are basic, fundamental, questions; they have been at the core of the research in concurrency theory for the past 30 years. (They are still so today, although remarkable progress has been made)

Fundamental for a model or a language on top of which we want to make proofs ...

We shall approach these questions from a simple case, in which interactions among processes are just synchronisations, without exchange of values.

Interaction

In the example at page 5

$$\mathtt{X} := 2 \qquad \text{and} \qquad \mathtt{X} := 1;\ \mathtt{X} := \ \mathtt{X} + 1$$

should be distinguished because they interact in a different way with the memory.

Computation is **interaction**. Examples: access to a memory cell, interrogating a data base, selecting a programme in a washing machine, ...

The participants of an interaction are **processes** (a cell, a data base, a washing machine, ...)

The **behaviour** of a process should tell us **when** and **how** a process can interact with its environment

How to represent interaction: labelled transition systems

Definition 1 A **labeled transition system** (LTS) is a triple $(\mathcal{P}, \mathtt{Act}, \mathcal{T})$ where

- $\mathcal{P}$ is the set of **states**, or **processes**;
- $\mathtt{Act}$ is the set of **actions**; (NB: can be infinite)
- $\mathcal{T} \subseteq (\mathcal{P}, \mathtt{Act}, \mathcal{P})$ is the **transition relation**.

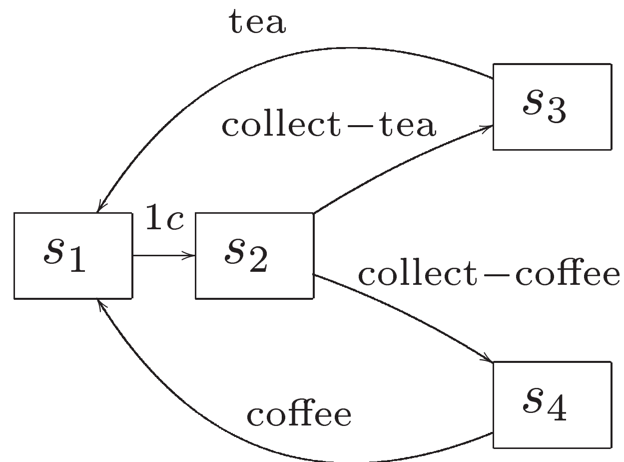
We write $P \xrightarrow{\mu} P'$ if $(P, \mu, P') \in \mathcal{T}$. Meaning: process P accepts an interaction with the environment where P performs action μ and then becomes process P'.

P' is a **derivative** of P if there are $P_1, \ldots, P_n, \mu_1, \ldots, \mu_n$ s.t. $P \xrightarrow{\mu_1} P_1 \ldots \xrightarrow{\mu_n} P_n$ and $P_n = P'$.

Example

A vending machine, capable of dispensing tea or coffee for 1 coin (1c).

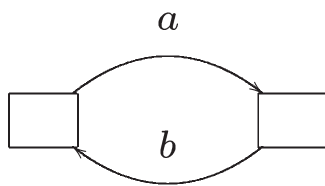
The behaviour of the machine is what we can observe, by interacting with the machine. We can represent such a behaviour as an LTS:

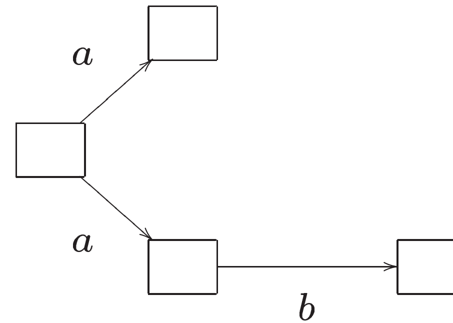


(where s_1 is the initial state)

Other examples of LTS

(we omit the name of the states)





Equivalence of processes

An LTS tells us what is the behaviour of processes. When should two behaviours be considered equal? ie, what does it mean that two processes are equivalent?

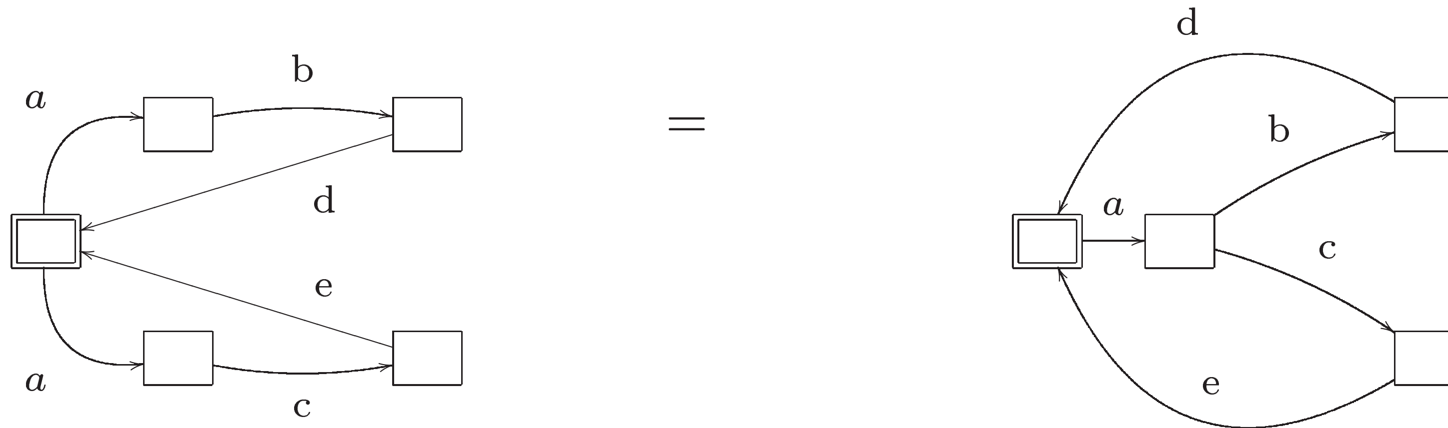
Two processes should be equivalent if we cannot distinguish them by interacting with them.

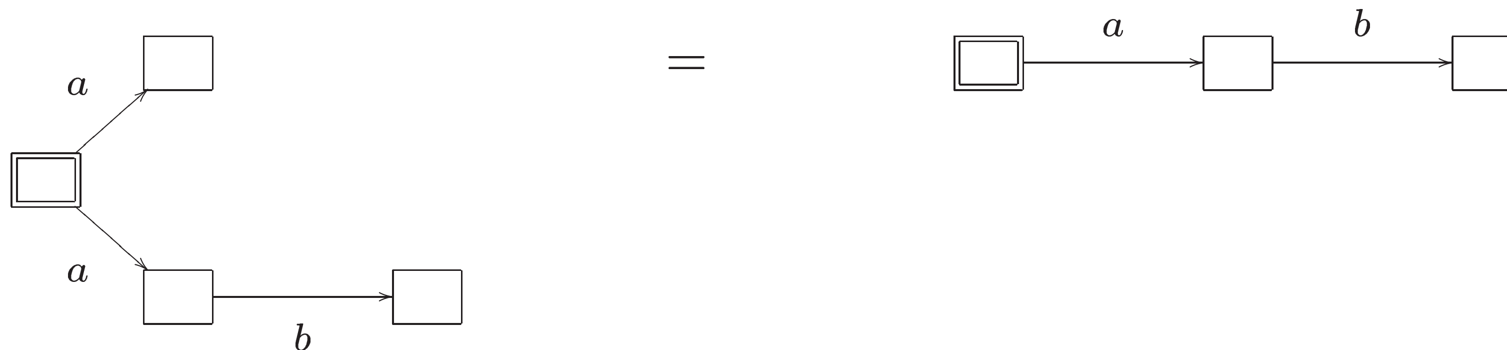
Example (where indicates the processes we are interested in):

This shows that **graph isomorphism** as behavioural equivalence is too strong.

A natural alternative (from automata theory): **trace equivalence**.

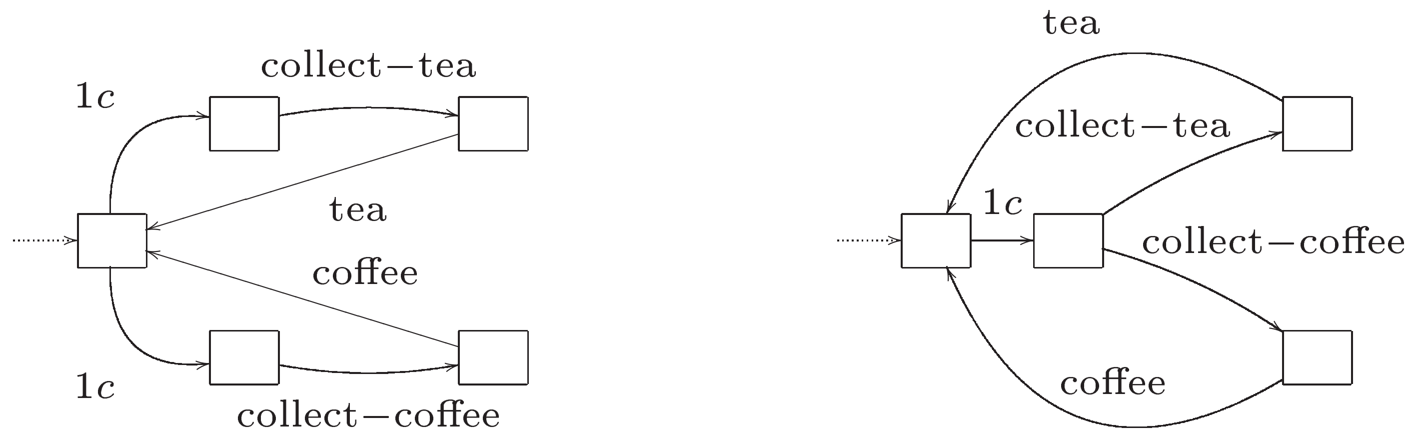
Examples of trace-equivalent processes:





These equalities are OK on automata. But they are not on processes: in one case interacting with the machine can lead to deadlock!

For instance, you would not consider these two vending machines ‘the same’:



Trace equivalence (also called language equivalence) is still important in concurrency.

Examples: confluent processes; liveness properties such as termination

These examples suggest that the notion of equivalence we seek:

- should imply a tighter correspondence between transitions than language equivalence,
- should be based on the informations that the transitions convey, and not on the shape of the diagrams.

Intuitively, what does it mean for an observer that two machines are equivalent?

If you do something with one machine, you must be able to the same with the other, and on the two states which the machines evolve to the same is again true.

This is the idea of equivalence that we are going to formalise; it is called **bisimilarity**.

Bisimulation

References:

Robin Milner, *Communication and Concurrency*, Prentice Hall, 1989.

We define bisimulation on a single LTS, because: the union of two LTSs is an LTS; we will often want to compare derivatives of the same process.

Definition 2 (bisimulation) A relation $\mathcal{R}$ on the states of an LTS is a **bisimulation** if whenever $P \mathcal{R} Q$:

1. if $P \xrightarrow{\mu} P'$, then there is Q' such that $Q \xrightarrow{\mu} Q'$ and $P' \mathcal{R} Q'$.
2. if $Q \xrightarrow{\mu} Q'$, then there is P' such that $P \xrightarrow{\mu} P'$ and $P' \mathcal{R} Q'$.

P and Q are **bisimilar**, written $P \sim Q$, if $P \mathcal{R} Q$, for some bisimulation $\mathcal{R}$.

The bisimulation diagram:

$$\begin{array}{ccc} P & \mathcal{R} & Q \\ \mu \downarrow & & \mu \downarrow \\ P' & \mathcal{R} & Q' \end{array}$$

Exercises

Exercise 3 Prove that the processes at page 12 are bisimilar. Are the processes at page 13 bisimilar?

Proposition 4 1. $\sim$ is an equivalence relation, i.e. the following hold:

1.1. $p \sim p$ (reflexivity)
1.2. $p \sim q$ implies $q \sim p$ (symmetry)
1.3. $p \sim q$ and $q \sim r$ imply $p \sim r$ (transitivity);

2. $\sim$ itself is a bisimulation.

Proposition 4(2) suggests an alternative definition of $\sim$:

Proposition 5 $\sim$ is the largest relation among the states of the LTS such that $P \sim Q$ implies:

1. if $P \xrightarrow{\mu} P'$, then there is Q' such that $Q \xrightarrow{\mu} Q'$ and $P' \sim Q'$.
2. if $Q \xrightarrow{\mu} Q'$, then there is P' such that $P \xrightarrow{\mu} P'$ and $P' \sim Q'$.

Exercise 6 Prove Propositions 4-5
(for 4.2 you have to show that

$$\cup\{\mathcal{R} \ : \ \mathcal{R} \text{ is a bisimulation } \}$$

is a bisimulation).

We write $P \sim\!\mathcal{R}\!\sim Q$ if there are P', Q' s.t. $P \sim P'$, $P' \mathrel{\mathcal{R}} Q'$, and $Q' \sim Q$ (and alike for similar notations).

Definition 7 (bisimulation up-to $\sim$) A relation $\mathcal{R}$ on the states of an LTS is a *bisimulation up-to* $\sim$ if $P \mathrel{\mathcal{R}} Q$ implies:

1. if $P \xrightarrow{\mu} P'$, then there is Q' such that $Q \xrightarrow{\mu} Q'$ and $P' \sim\!\mathcal{R}\!\sim Q'$.
2. if $Q \xrightarrow{\mu} Q'$, then there is P' such that $P \xrightarrow{\mu} P'$ and $P' \sim\!\mathcal{R}\!\sim Q'$.

Exercise 8 If $\mathcal{R}$ is a bisimulation up-to $\sim$ then $\mathcal{R} \subseteq \sim$. (Hint: prove that $\sim \mathcal{R} \sim$ is a bisimulation.)

Definition 9 (simulation) A relation $\mathcal{R}$ on the states of an LTS is a *simulation* if $P \mathcal{R} Q$ implies:

1. if $P \xrightarrow{\mu} P'$, then there is Q' such that $Q \xrightarrow{\mu} Q'$ and $P' \mathcal{R} Q'$.

P is *simulated by* Q, written $P < Q$, if $P \mathcal{R} Q$, for some simulation $\mathcal{R}$.

Exercise* 10 Does $P \sim Q$ imply $P < Q$ and $Q < P$? What about the converse? (Hint for the second point: think about the 2nd equality at page 13.)

- Bisimulation has been introduced in Computer Science by Park (1981) and made popular by Milner.

- Bisimulation is a robust notion: characterisations of bisimulation have been given in terms of non-well-founded-sets, modal logic, final coalgebras, open maps in category theory, etc.

- But the most important feature of bisimulation is the associated **co-inductive** proof technique.

Induction and co-induction

Co-inductive definitions and co-inductive proofs

Consider this definition of $\sim$ (Proposition 5):

$\sim$ is the largest relation such that $P \sim Q$ implies:

1. if $P \xrightarrow{\mu} P'$, then there is Q' such that $Q \xrightarrow{\mu} Q'$ and $P' \sim Q'$.
2. if $Q \xrightarrow{\mu} Q'$, then there is P' such that $P \xrightarrow{\mu} P'$ and $P' \sim Q'$.

It is a circular definition; does it make sense?

We claimed that we can prove $(P, Q) \in \sim$ by showing that $(P, Q) \in \mathcal{R}$ and $\mathcal{R}$ is a *bisimulation relation*, that is a relation that satisfies the same clauses as $\sim$. Does such a proof technique make sense?

Contrast all this with the usual, familiar *inductive definitions* and *inductive proofs*.

The above definition of $\sim$, and the above proof technique for $\sim$ are examples of *co-inductive definition* and of *co-inductive proof technique*.

Bisimulation and co-induction: what are we talking about?
Has *co*-induction anything to do with induction?

An example of an inductive definition: reduction to a value in the λ-calculus

The set Λ of λ-terms (an inductive def!)

$$e ::= x \;\Big|\; \lambda x.\, e \;\Big|\; e_1(e_2)$$

Consider the definition of $\Downarrow_n$ in λ-calculus (convergence to a value):

$$\frac{}{\lambda x.\, e \Downarrow_n \lambda x.\, e} \qquad \frac{e_1 \Downarrow_n \lambda x.\, e_0 \qquad e_0\{^{e_2}/_x\} \Downarrow_n e'}{e_1(e_2) \Downarrow_n e'}$$

$\Downarrow_n$ is the *smallest* relation on λ-terms that is closed forwards under these rules; i.e., the smallest subset C of $\Lambda \times \Lambda$ s.t.

- $\lambda x.\, e C \lambda x.\, e$ for all abstractions,
- if $e_1 \; C \; \lambda x.\, e_0$ and $e_0\{^{e_2}/_x\} \; C \; e'$ then also $e_1(e_2) \; C \; e'$.

An example of a co-inductive definition: divergence in the λ-calculus

Consider the definition of $\Uparrow^n$ (divergence) in λ-calculus :

$$\frac{e_1 \Uparrow^n}{e_1(e_2) \Uparrow^n} \qquad\qquad \frac{e_1 \Downarrow_n \lambda x.\, e_0 \qquad e_0\{^{e_2}/_x\} \Uparrow^n}{e_1(e_2) \Uparrow^n}$$

$\Uparrow^n$ is the *largest* predicate on λ-terms that is closed backwards under these rules; i.e., the largest subset D of Λ s.t. if $e \in D$ then

- either $e = e_1(e_2)$ and $e_1 \in D$,
- or $e = e_1(e_2)$, $e_1 \Downarrow_n \lambda x.\, e_0$ and $e_0\{^{e_2}/_x\} \in D$.

Hence: to prove e is divergent it suffices to find $E \subseteq \Lambda$ that is closed backwards and with $e \in E$ (co-induction proof technique).

What is the smallest predicate closed backwards?

An example of an inductive definition: finite lists over a set A

$$\frac{}{\texttt{nil} \in \mathcal{L}} \qquad\qquad \frac{\ell \in \mathcal{L} \qquad a \in A}{\texttt{cons}(a, \ell) \in \mathcal{L}}$$

Finite lists: the set generated by these rules; i.e., the smallest set closed forwards under these rules.

Inductive proof technique for lists: Let $\mathcal{P}$ be a predicate (a property) on lists. To prove that $\mathcal{P}$ holds on all lists, prove that

– $\texttt{nil} \in \mathcal{P}$;

– $\ell \in \mathcal{P}$ implies $\texttt{cons}(a, \ell) \in \mathcal{P}$, for all $a \in A$.

An example of an co-inductive definition: finite and infinite lists over a set A

$$\frac{}{\texttt{nil} \in \mathcal{L}} \qquad \frac{\ell \in \mathcal{L} \qquad a \in A}{\texttt{cons}(a, \ell) \in \mathcal{L}}$$

Finite and infinite lists: the largest set closed backwards under these rules.

To explain finite and infinite lists as a set, we need *non-well-founded sets* (Forti-Honsell, Aczel).

- – * An inductive definition tells us what are the *constructors* for generating all the elements (cf: closure forwards).
    * A co-inductive definition tells us what are the *destructors* for decomposing the elements (cf: closure backwards).
    The destructors show what we can *observe* of the elements (think of the elements as black boxes; the destructors tell us what we can do with them; this is clear in the case of infinite lists).
- – When a definition is give by means of some rules:
    * if the definition is inductive, we look for the smallest universe in which such rules live.
    * if it is co-inductive, we look for the largest universe.

The duality

constructors	destructors
inductive defs	co-inductive defs
induction technique	co-inductive technique
congruence	bisimulation
least fixed-points	greatest fixed-points

(The dual of a *bisimulation* is a *congruence*: intuitively: a bisimulation is a relation “closed backwards”, a congruence is “closed forwards”)

- In what sense are $\Downarrow_n, \Uparrow^n, \sim$ fixed-points?
- What is the co-induction proof technique?
- In what sense is co-induction dual to the familiar induction technique?

What follows answers these questions. It is a simple application of fixed-point theory.

To make things simpler, we work on *powersets*. (It is possible to be more general, working with universal algebras or category theory.)

For a given set S, the powerset of S, written $\wp(S)$, is

$$\wp(S) \triangleq \{T : T \subseteq S\}$$

$\wp(S)$ is a *complete lattice*.

Complete lattices are “dualisable” structures: reverse the arrows and you get another complete lattice.

From Knaster-Tarski's theorem for complete lattices, we know that if $\mathcal{F} : \wp(S) \to \wp(S)$ is monotone, then $\mathcal{F}$ has a least fixed-point (lfp), namely:

$$\mathcal{F}_{\mathrm{lfp}} \triangleq \bigcap \{A \;:\; \mathcal{F}(A) \subseteq A\}$$

As we are on a complete lattice, we can dualise the statement:

If $\mathcal{F} : \wp(S) \to \wp(S)$ is monotone, then $\mathcal{F}$ has a greatest fixed-point (gfp), namely:

$$\mathcal{F}^{\mathrm{gfp}} \triangleq \bigcup \{A \;:\; A \subseteq \mathcal{F}(A)\}$$

These results give us proof techniques for $\mathcal{F}_{\mathrm{lfp}}$ and $\mathcal{F}^{\mathrm{gfp}}$:

$$\text{if } \mathcal{F}(A) \subseteq A \text{ then } \mathcal{F}_{\mathrm{lfp}} \subseteq A \tag{1}$$

$$\text{if } A \subseteq \mathcal{F}(A) \text{ then } A \subseteq \mathcal{F}^{\mathrm{gfp}} \tag{2}$$

– Inductive definitions give us lfp's (precisely: an inductive definition tells us how to construct the lfp). Co-inductive definitions give us gfp's.

– On inductively-defined sets (1) is the same as the familiar induction technique (cf: example of lists). (2) gives us the co-inductive proof technique.

$\Downarrow_n$ and $\Uparrow^n$ as fixed-points

A set R of rules on a set S give us a function $\mathcal{R}$: $\wp(S) \to \wp(S)$, so defined:

$$\mathcal{R}\ (A) \triangleq \{a \ : \ \text{there are } a_1, \ldots, a_n \in A \quad \text{and a rule in } R \text{ so that using } a_1, \ldots, a_n \text{ as premises in the rule we can derive } a\}$$

$\mathcal{R}$ is monotone, and therefore (by Tarsky) has lfp and gfp.

In this way, the definitions of $\Downarrow_n$ and $\Uparrow^n$ can be formulated as lfp and gfp of functions.

For instance, in the case of $\Uparrow^n$, take $S = \Lambda$. Then

$$\mathcal{R}(A) = \{e_1(e_2) \ : \ e_1 \in A, \quad \text{or } e_1 \Downarrow_n \lambda x.\, e_0 \text{ and } e_0\{^{e_2}\!/\!_x\} \in A\}.$$

The co-induction proof technique for $\Uparrow^n$ mentioned at page 27 is just an instance of (2).

Bisimulation as a fixed-point

Let $(\mathcal{P}, \mathtt{Act}, \mathcal{T})$ be an LTS. Consider the function $\mathcal{F} : \wp(\mathcal{P} \times \mathcal{P}) \rightarrow \wp(\mathcal{P} \times \mathcal{P})$ so defined.

$\mathcal{F}(R)$ is the set of all pairs (P, Q) s.t.:

1. if $P \xrightarrow{\mu} P'$, then there is Q' such that $Q \xrightarrow{\mu} Q'$ and $(P', Q') \in \mathcal{R}$.
2. if $Q \xrightarrow{\mu} Q'$, then there is P' such that $P \xrightarrow{\mu} P'$ and $(P', Q') \in \mathcal{R}$.

Proposition 11 1. $\mathcal{F}$ is monotone;

2. $\sim = \mathcal{F}^{\text{gfp}}$;

3. R is a bisimulation iff $R \subseteq \mathcal{F}(\mathcal{R})$.

The induction technique as a fixed-point technique: the example of finite lists

Let $\mathcal{F}$ be this function (from sets to sets):

$$\mathcal{F}(S) \triangleq \{\texttt{nil}\} \cup \{\texttt{cons}(a, s) \; : \; a \in A, s \in S\}$$

$\mathcal{F}$ is monotone, and $\mathrm{finLists} = \mathcal{F}_{\mathrm{lfp}}$. (i.e., $\mathrm{finLists}$ is the smallest set solution to the equation $\mathcal{L} = \texttt{nil} + \texttt{cons}(A, \mathcal{L})$).

From (1), we infer: Suppose $\mathcal{P} \subseteq \mathrm{finLists}$. If $\mathcal{F}(\mathcal{P}) \subseteq \mathcal{P}$ then $\mathcal{P} \subseteq \mathrm{finLists}$ (hence $\mathcal{P} = \mathrm{finLists}$).

Proving $\mathcal{F}(\mathcal{P}) \subseteq \mathcal{P}$ requires proving

- $\texttt{nil} \in \mathcal{P}$;
- $\ell \in \mathrm{finLists} \cap \mathcal{P}$ implies $\texttt{cons}(a, \ell) \in \mathcal{P}$, for all $a \in A$.

This is the same as the familiar induction technique for lists (page 28).

Note: $\mathcal{F}$ is defined the class of all sets, rather than on a powerset; the class of all sets is not a complete lattice (because of paradoxes such as Russel's), but the constructions that we have seen for lfp and gfp of monotone functions apply.

Continuity

Another important theorem of fixed-point theory: if $\mathcal{F} : \wp(S) \rightarrow \wp(S)$ is continuous, then

$$\mathcal{F}_{\mathrm{lfp}} = \bigcup_n \mathcal{F}^n(\bot)$$

This has, of course, a dual, for gfp (also the definition of continuity has to be dualised), but: the function $\mathcal{F}$ of which bisimilarity is the gfp may not be continuous! (This is usually the case for *weak* bisimilarity, that we shall introduce later.)

It is continuous only if the LTS is finite-branching, meaning that for all P the set $\{P' \; : \; P \xrightarrow{\mu} P', \text{ for some } \mu\}$ is finite.

On a finite branching LTS, it is indeed the case that

$$\sim \; = \bigcap_n \mathcal{F}^n(\mathcal{P} \times \mathcal{P})$$

where $\mathcal{P}$ is the set of all processes.

Stratification of bisimilarity

Continuity, operationally:

Consider the following sequence of equivalences, inductively defined:

$\sim_0 \triangleq \mathcal{P} \times \mathcal{P}$

$P \sim_{n+1} Q \triangleq$:

1. if $P \xrightarrow{\mu} P'$, then there is Q' such that $Q \xrightarrow{\mu} Q'$ and $P' \sim_n Q'$.
2. if $Q \xrightarrow{\mu} Q'$, then there is P' such that $P \xrightarrow{\mu} P'$ and $P' \sim_n Q'$.

Then set:

$\sim_\omega \triangleq \bigcap_n \sim_n$

Theorem 12 On processes that are image-finite: $\sim = \sim_\omega$

Image-finite processes :

each reachable state can only perform a finite set of transitions.

Abbreviation: $a^n \triangleq a. \ldots a. 0$ (n times)

Example: $\sum_{1 \le i \le n} a^n$ (note: n is fixed)

Non-example: $P \triangleq \sum_{1 \le i < \omega} a^n$

In the theorem, image-finiteness is necessary:

$P \sim_\omega P + a^\omega$ but $P \not\sim P + a^\omega$

The stratification of bisimilarity given by continuity is also the basis for **algorithms** for mechanically checking bisimilarity and for minimisation of the state-space of a process

These algorithms work on processes that are **finite-state** (ie, each process has only a finite number of possible derivaties)

They proceed by progressively refining a partition of all processes

In the initial partition, all processes are in the same set

Bisimulation: P-complete

[Alvarez, Balcazar, Gabarro, Santha, '91]

With m transitions, n states:

O(m log n) time and O(m + n) space **[Paige, Tarjan, '87]**

Trace equivalence, testing: PSPACE-complete

[Kannelakis, Smolka, '90; Huynh, Tian, 95]

Weak bisimulation

Consider the processes

$$\tau.\,\overline{a}.\,0 \qquad \text{and} \qquad \overline{a}.\,0$$

They are not strongly bisimilar.

But we do want to regard them as behaviourally equivalent! τ-transitions represent internal activities of processes, which are not visible.

(Analogy in functional languages: $(\lambda x.\, x)3$ and 3 are semantically the same.)

Internal work (τ-transitions) should be ignored in the bisimulation game. Define:

(i) $\Longrightarrow$ as the reflexive and transitive closure of $\xrightarrow{\tau}$.

(ii) $\stackrel{\mu}{\Longrightarrow}$ as $\Longrightarrow\xrightarrow{\mu}\Longrightarrow$ (relational composition).

(iii) $\stackrel{\widehat{\mu}}{\Longrightarrow}$ is $\Longrightarrow$ if $\mu = \tau$; it is $\stackrel{\mu}{\Longrightarrow}$ otherwise.

Definition 13 (weak bisimulation, or observation equivalence) A process relation $\mathcal{R}$ is a *weak bisimulation* if $P \mathcal{R} Q$ implies:

1. if $P \stackrel{\mu}{\Longrightarrow} P'$, then there is Q' s.t. $Q \stackrel{\widehat{\mu}}{\Longrightarrow} Q'$ and $P' \mathcal{R} Q'$;
2. the converse of (1) on the actions from Q.

P and Q are *weakly bisimilar*, written $P \approx Q$, if $P \mathcal{R} Q$ for some weak bisimulation $\mathcal{R}$.

Why did we study strong bisimulation?

- $\sim$ is simpler to work with, and $\sim \subseteq \approx$; (cf: exp. law)
- the theory of $\approx$ is in many aspects similar to that of $\sim$;
- the differences between $\sim$ and $\approx$ correspond to subtle points in the theory of $\approx$

Are the processes $\tau.\, 0 + \tau.\, \overline{a}.\, 0$ and $\overline{a}.\, 0$ weakly bisimilar ?

Examples of non-equivalence:

$$a + b \not\approx a + \tau.\, b \not\approx \tau.\, a + \tau.\, b \not\approx a + b$$

Examples of equivalence:

$$\tau.\, a \approx a \approx a + \tau.\, a$$

$$a.\, (b + \tau.\, c) \approx a.\, (b + \tau.\, c) + a.\, c$$

These are instances of useful algebraic laws, called the τ *laws*:

Lemma 14 1. $P \approx \tau.\, P$;

2. $\tau.\, N + N \approx N$;

3. $M + \alpha.\, (N + \tau.\, P) \approx M + \alpha.\, (N + \tau.\, P) + \alpha.\, P$.

In the clauses of Definition 13, the use of $\stackrel{\mu}{\Longrightarrow}$ on the challenger side can be heavy.

For instance, take the CCS process $K \triangleq \tau.\,(a \mid K)$; for all n, we have $K \Longrightarrow (a \mid)^n \mid K$, and all these transitions have to be taken into account in the bisimulation game.

The following definition is much simpler to use (the challenger makes a single move):

Definition 15 A process relation $\mathcal{R}$ is a *weak bisimulation* if $P \mathcal{R} Q$ implies:

1. if $P \xrightarrow{\mu} P'$, then there is Q' s.t. $Q \stackrel{\widehat{\mu}}{\Longrightarrow} Q'$ and $P' \mathcal{R} Q'$;
2. the converse of (1) on the actions from Q (ie, the roles of P and Q are inverted).

Proposition 16 The definitions 13 and 15 of weak bisimulation coincide.

Proof: A useful exercise. ■

Weak bisimulations "up-to"

Definition 17 (weak bisimulation up-to $\sim$) A process relation $\mathcal{R}$ is a *weak bisimulation up-to* $\sim$ if $P \; \mathcal{R} \; Q$ implies:

1. if $P \xrightarrow{\mu} P'$, then there is Q' s.t. $Q \stackrel{\widehat{\mu}}{\Longrightarrow} Q'$ and $P' \sim\mathcal{R}\sim Q'$;
2. the converse of (1) on the actions from Q.

Exercise 18 If $\mathcal{R}$ is a weak bisimulation up-to $\sim$ then $\mathcal{R} \subseteq \approx$.

Definition 19 (weak bisimulation up-to $\approx$) A process relation $\mathcal{R}$ is a *weak bisimulation up-to* $\approx$ if $P \; \mathcal{R} \; Q$ implies:

1. if $P \stackrel{\mu}{\Longrightarrow} P'$, then there is Q' s.t. $Q \stackrel{\widehat{\mu}}{\Longrightarrow} Q'$ and $P' \approx\mathcal{R}\approx Q'$;
2. the converse of (1) on the actions from Q.

Exercise 20 If $\mathcal{R}$ is a weak bisimulation up-to $\approx$ then $\mathcal{R} \subseteq \approx$.

Enhancements of the bisimulation proof method

- The forms of “up-to” techniques we have seen are examples of **enhancements** of the bisimulation proof method
- Such enhancements are **extremely useful**
  * They are **essential** in π-calculus-like languages, higher-order languages
- **Various forms** of enhancement (“up-to techniques”) exist (up-to context, up-to substitution, etc.)
- They are **subtle**, and not well-understood yet

Example: up-to bisimilarity that fails

In Definition 17 we cannot replace $\sim$ with $\approx$:

$$
\begin{array}{ccc}
\tau.\,a.\,0 & \mathcal{R} & 0 \\
\downarrow & & \Downarrow \\
a.\,0 & & 0 \\
\\
\approx & & \approx \\
\\
\tau.\,a.\,0 & \mathcal{R} & 0
\end{array}
$$

The success of bisimulation

and co-induction

Bisimulation in Computer Science

– One of the most important contributions of concurrency theory to CS

– It has spurred the study of coinduction

– In concurrency: probably the most studied equivalence

  * ... in a plethora of equivalences **(see van Glabbeek 93)**
  * Why?

Bisimulation in concurrency

- **Clear** meaning of equality
- **Natural**
- The **finest** extensional equality

  Extensional: – "whenever it does an output at b it will also do an input at a"

  Non-extensional: – "Has 8 states"
  – "Has an Hamiltonian circuit"
- An associated powerful **proof technique**
- **Robust**

  Characterisations: logical, algebraic, set-theoretical, categorical, game-theoretical, ...
- Several **separation results** from other equivalences

Bisimulation in concurrency, today

- To **define equality** on processes (fundamental !!)
- To **prove equalities**
  * even if bisimilarity is not the chosen equivalence
    - trying bisimilarity first
    - coinductive characterisations of the chosen equivalence
- To **justify algebraic laws**
- To **minimise** the state space
- To **abstract** from certain details

Coinduction in programming languages

- **Bisimilarity in functional languages and OO languages**

  [Abramsky, Ong]

  A major factor in the movement towards operationally-based techniques in PL semantics in the 90s

- **Program analysis** **(see Nielson, Nielson, Hankin 's book)**

  Noninterference (security) properties

- **Verification tools**: algorithms for computing gfp (for modal and temporal logics), tactics and heuristics

- **Types** **[Tofte]**
  * type soundness
  * coinductive types and definition by corecursion
    Infinite proofs in Coq **[Coquand, Gimenez]**
  * recursive types (equality, subtyping, ...)
    A coinductive rule:

$$\frac{\Gamma, \langle p_1, q_1 \rangle \sim \langle p_2, q_2 \rangle \vdash p_i \sim q_i}{\Gamma \vdash \langle p_1, q_1 \rangle \sim \langle p_2, q_2 \rangle}$$

- **Recursively defined data types and domains** **[Fiore, Pitts]**
- **Databases** **[Buneman]**
- **Compiler correctness** **[Jones]**

Other equivalences

Concurrency theory: models of processes

- LTS
- Petri Nets
- Mazurkiewikz traces
- Event structures
- I/O automata

Process calculi

- CCS [$\rightarrow$ π-calculus $\rightarrow$ Join]
- CSP
- ACP
- Additional features: real-time, probability,...

Behavioural equivalences (and preorders)

- traces
- bisimilarity (in various forms)
- failures and testing
- non-interleaving equivalences (in which parallelism cannot be reduced to non-determinism, cf. the expansion law)
  [causality, location-based]

Depending on the desired level of abstraction or on the tools available, an equivalence may be better than an other.

van Glabbeek, in '93, listed more than 50 forms of behavioural equivalence, today the listing would be even longer

Rob J. van Glabbeek: The Linear Time - Branching Time Spectrum II, LNCS 715, 1993

Failure equivalence

In CSP equivalence, it is intended that the observations are those obtained from all possible finite experiments with the process

A failure is a pair (μ^+, A), where μ^+ is a trace and A a set of actions. The failure (μ^+, A) belongs to process P if

- $P \stackrel{\mu^+}{\longrightarrow} P'$, for some P'
- not $P' \stackrel{\tau}{\longrightarrow}$
- not $P' \stackrel{a}{\longrightarrow}$, for all $a \in A$

Example: $P \triangleq a.(b.c.0 + b.d.0)$ has the following failures:

- (ϵ, A) for all A with $a \notin A$.
- (a, A) for all A with $b \notin A$.
- (ab, A) for all A with $\{c, d\} \not\subseteq A$.
- (abc, A) and (abd, A), for all A

Two processes are failure-equivalent if they possess the same failures

Advantages of failure equivalence:

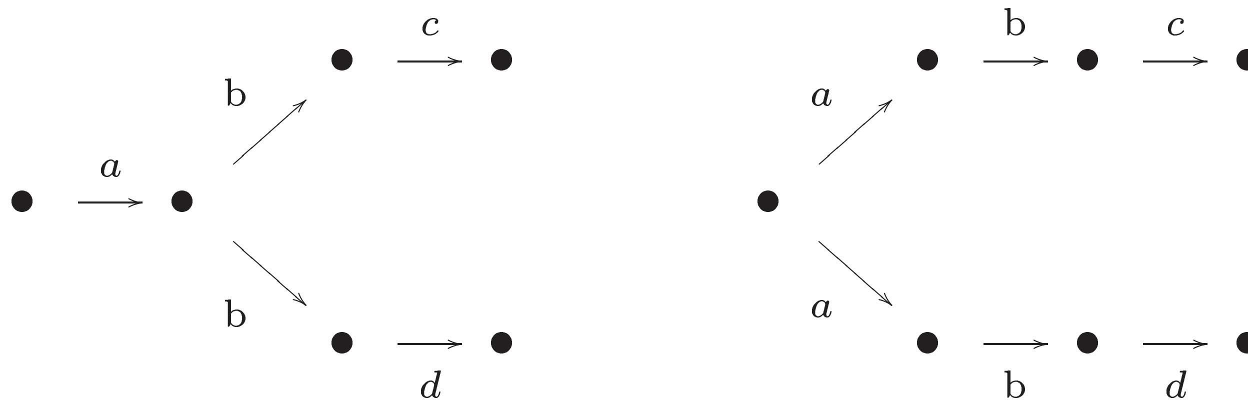
- the coarsest equivalence sensitive to deadlock
- characterisation as testing equivalence

Advantages of bisimilarity:

- the co-inductive technique
- the finest reasonable behavioural equivalence for processes
- robust mathematical characterisations

Failure is not preserved, for instance, by certain forms of priority

These processes are failure equivalent but not bisimilar



A law valid for bisimilarity but not for failure:

$$a.\,(b.\,P + b.\,Q) = a.\,b.\,P + a.\,b.\,Q$$

Testing

References:

Matthew Hennessy: Observing processes. REX Workshop 1988, LNCS 354, 1989.

Rocco De Nicola, Matthew Hennessy: Testing Equivalences for Processes. Theor. Comput. Sci. 34 (1984)

Matthew Hennessey. Algebraic Theory of Processes. MIT Press, 1988.

Samson Abramsky, Observation equivalence as a testing equivalence, Theoretical Computer Science, v.53 n.2-3, 1987

- The testing theme:

  Processes should be equivalent unless there is some test that can tell them apart

- We first show how to capture bisimilarity this way

- Then we will notice that there are other reasonable ways of defining the language of tests, and these may lead to different semantic notions.

- In this section: processes are (image-finte) LTSs (ie, finitely-branching labelled trees), with labels from a given alphabet of actions `Act`

Bisimulation in a testing scenario

Language for testing:

$$T ::= \mathrm{SUCC} \;\big|\; \mathrm{FAIL} \;\big|\; a.\,T \;\big|\; \widetilde{a}.\,T \;\big|\; T_1 \wedge T_2 \;\big|\; T_1 \vee T_2 \;\big|\; \forall T \;\big|\; \exists T$$

($a \in \mathtt{Act}$)

The outcomes of an **experiment**, testing a process P with a test T:

$$\mathcal{O}(T, P) \subseteq \{\top, \bot\}$$

$\top$:success

$\bot$: lack of success (failure, or success is never reached)

Notation:

$P\ \mathtt{ref}(a) \triangleq P$ cannot perform a (ie, there is no P' st $P \xrightarrow{a} P'$)

Outcomes

$$\mathcal{O}(\mathrm{SUCC}, P) = \top$$

$$\mathcal{O}(\mathrm{FAIL}, P) = \bot$$

$$\mathcal{O}(a.T, P) = \begin{cases} \{\bot\} & \text{if } P\ \mathtt{ref}(a) \\ \bigcup\{\mathcal{O}(T, P') \ : \ P \xrightarrow{a} P'\} & \text{otherwise} \end{cases}$$

$$\mathcal{O}(\tilde{a}.T, P) = \begin{cases} \{\top\} & \text{if } P\ \mathtt{ref}(a) \\ \bigcup\{\mathcal{O}(T, P') \ : \ P \xrightarrow{a} P'\} & \text{otherwise} \end{cases}$$

$$\mathcal{O}(T_1 \wedge T_2, P) = \mathcal{O}(T_1, P) \wedge^\star \mathcal{O}(T_1, P)$$

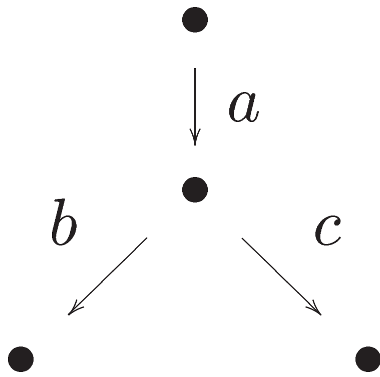
$$\mathcal{O}(T_1 \vee T_2, P) = \mathcal{O}(T_1, P) \vee^\star \mathcal{O}(T_1, P)$$

$$\mathcal{O}(\forall T, P) = \begin{cases} \{\top\} & \text{if } \bot \notin \mathcal{O}(T, P) \\ \{\bot\} & \text{otherwise} \end{cases}$$

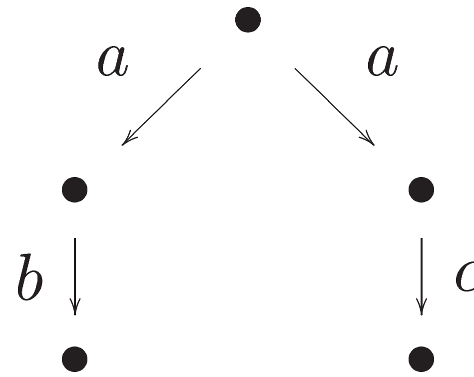
$$\mathcal{O}(\exists T, P) = \begin{cases} \{\top\} & \text{if } \top \in \mathcal{O}(T, P) \\ \{\bot\} & \text{otherwise} \end{cases}$$

where $\wedge^\star$ and $\vee^\star$ are the pointwise extensions of $\wedge$ and $\vee$ to powersets

Examples (a)



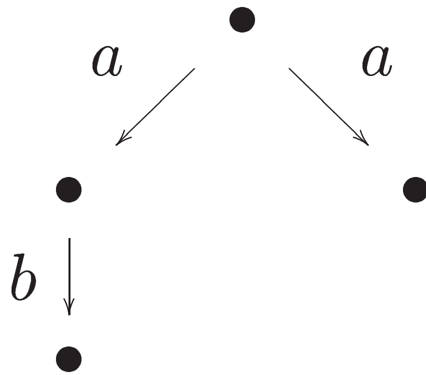
P_1



P_2

For $T_1 = a.\,(b.\,\mathrm{SUCC} \wedge c.\,\mathrm{SUCC})$, we have $\mathcal{O}(T_1, P_1) = \{\top\}$ and $\mathcal{O}(T_1, P_2) = \{\bot\}$

Examples (b)

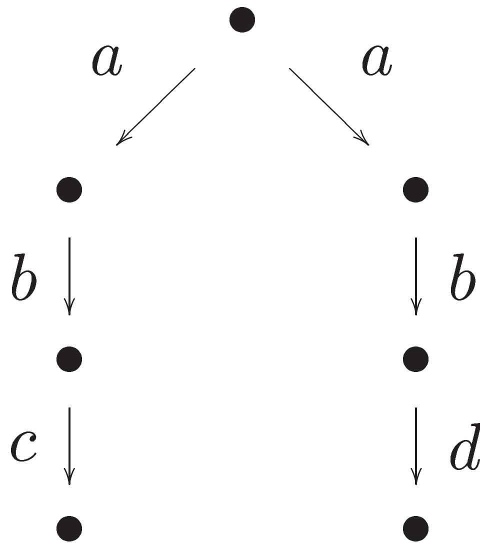


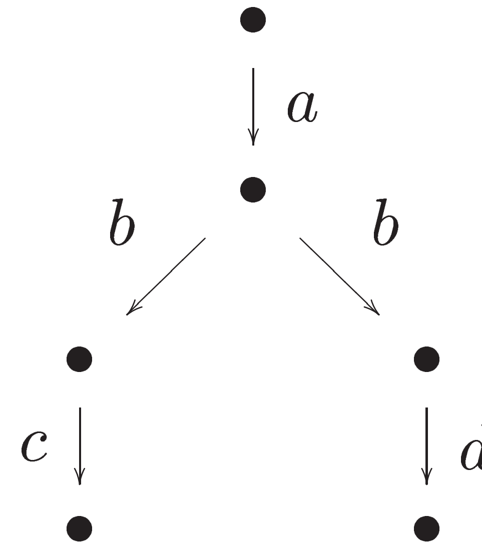
P_3 P_4

For $T_3 = a.\, b.\, \mathrm{SUCC}$, we have $\mathcal{O}(T_3, P_3) = \{\bot, \top\}$ and $\mathcal{O}(T_3, P_4) = \{\top\}$

For $T_4 = a.\, \widetilde{b}.\, \mathrm{FAIL}$, we have $\mathcal{O}(T_4, P_3) = \{\bot, \top\}$ and $\mathcal{O}(T_4, P_4) = \{\bot\}$

Examples (c)



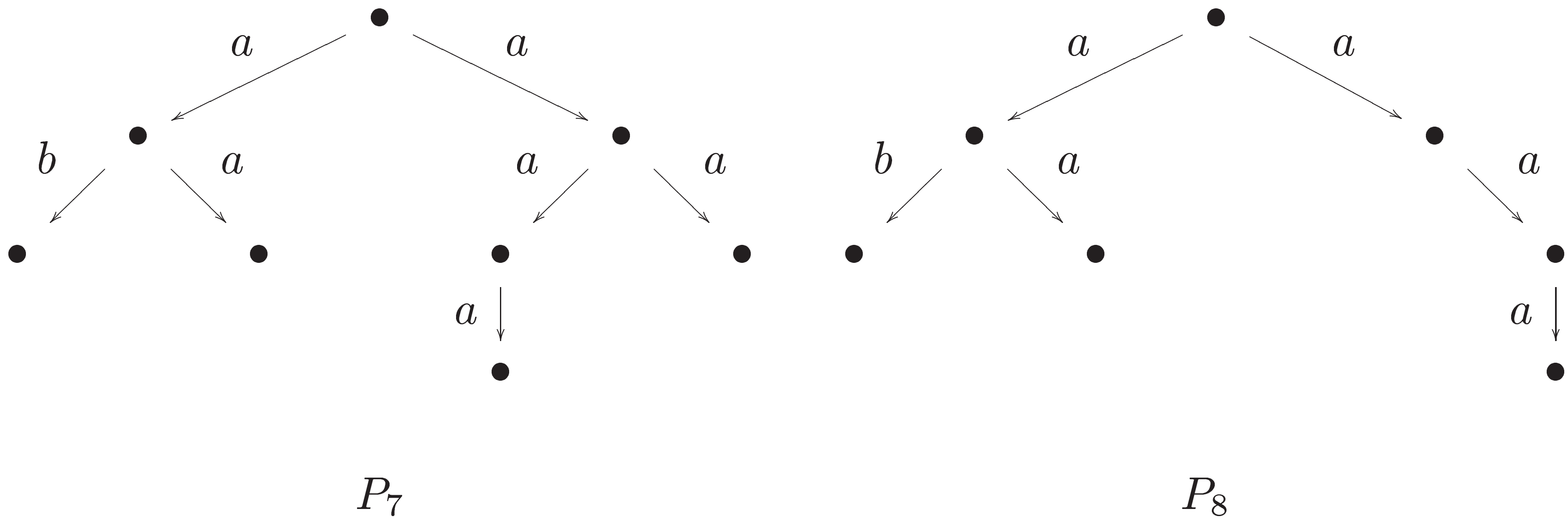


P_5 P_6

For $T = \exists a.\, \forall b.\, c.\, \mathrm{SUCC}$, we have $\mathcal{O}(T, P_5) = \{\top\}$ and $\mathcal{O}(T, P_6) = \{\bot\}$

Exercise: define other tests that distinguish between P_5 and P_6.

Examples (d)



Exercise 21 Define tests that distinguish between P_7 and P_8.

Note: Every test has an inverse:

$$
\begin{array}{rcl}
\overline{\mathrm{SUCC}} & = & \mathrm{FAIL} \\
\overline{\mathrm{FAIL}} & = & \mathrm{SUCC} \\
\overline{a.T} & = & \widetilde{a}.\overline{T} \\
\overline{\widetilde{a}.T} & = & a.\overline{T} \\
\overline{T_1 \wedge T_2} & = & \overline{T_1} \vee \overline{T_2} \\
\overline{T_1 \vee T_2} & = & \overline{T_1} \wedge \overline{T_2} \\
\overline{\forall T} & = & \exists \overline{T} \\
\overline{\exists T} & = & \forall \overline{T}
\end{array}
$$

We have:

1. $\bot \in \mathcal{O}(T, P)$ iff $\top \in \mathcal{O}(\overline{T}, P)$
2. $\top \in \mathcal{O}(T, P)$ iff $\bot \in \mathcal{O}(\overline{T}, P)$

The equivalence induced by these tests:

$P \sim_{\mathrm{T}} Q \triangleq$ for all T, $\mathcal{O}(T, P) = \mathcal{O}(T, Q)$.

Theorem 22 $\sim = \sim_{\mathrm{T}}$

- The proof is along the lines of the proof of characterisation of bisimulation in terms of modal logics (Hennessy-Milner's logics and theorem)
- A similar theorem holds for weak bisimilarity (with internal actions, the definition of the tests may need to be refined)

Testing equivalence

- The previous testing scenario requires considerable control over the processes (eg: the ability to copy their state at any moment)
  One may argue that this is too strong
- An alternative: the tester is a process of the same language as the tested process (in our case: an LTS)
- Performing a test : the two processes attempt to communicate with each other.
- Thus most of the constructs in the previous testing language are no longer appropriate (for instance, because they imply the ability of copying a process)
- To signal success, the tester process uses a special action $w \notin \mathtt{Act}$

Outcomes of running a test

Experiments:

$$E ::= \langle T, P \rangle \ \Big| \ \top$$

A *run* for a pair $\langle T, P \rangle$: a (finite or infinite) sequence of esperiments E_i such that

1. $E_0 = \langle T, P \rangle$
2. a transition $E_i \xrightarrow{a} E_{i+1}$ is defined by the following rules:

$$\frac{T \xrightarrow{a} T' \qquad P \xrightarrow{a} P'}{\langle T, P \rangle \longrightarrow \langle T', P' \rangle}$$

$$\frac{T \xrightarrow{w} T'}{\langle T, P \rangle \longrightarrow \top}$$

3. the last element of the sequence, say E_k, is such that there is no E' such that $E_k \longrightarrow E'$.

We now set:

$\top \in \mathcal{O}(T, P)$ if $\langle T, P\rangle$ has a run in which $\top$ appears (ie, $\langle T, P\rangle \Longrightarrow \top$)

$\perp \in \mathcal{O}(T, P)$ if there is a run for $\langle T, P\rangle$ in which $\top$ never appears

Testing equivalence ($\simeq$): the equivalence on processes so obtained

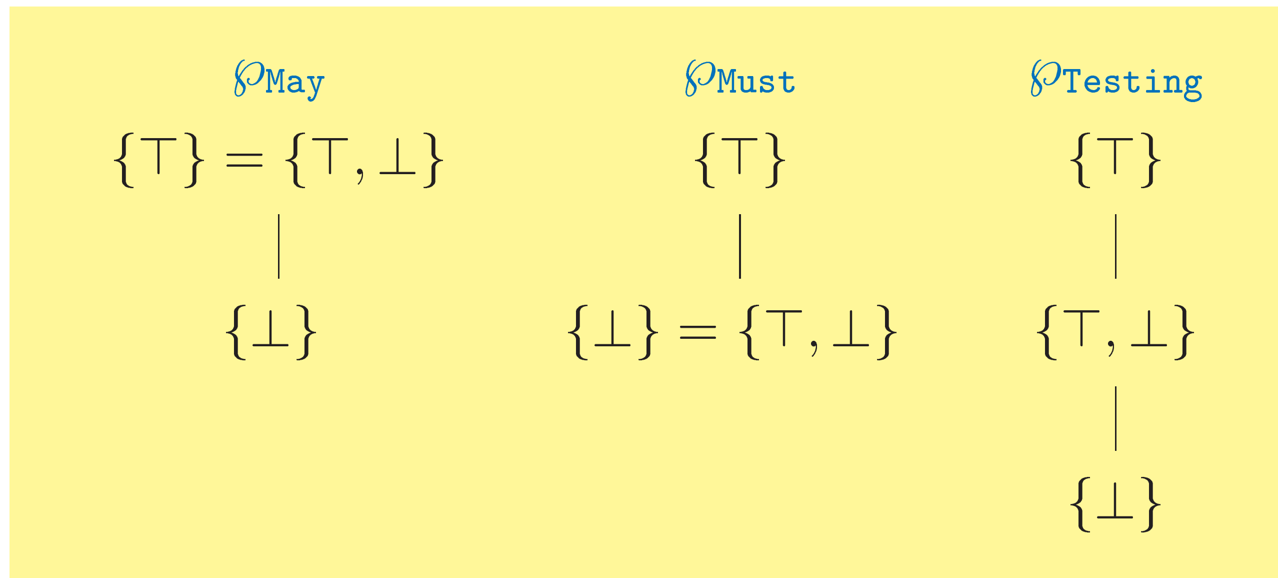
Note: If processes could perform internal actions, then other rules would be needed:

$$\frac{T \xrightarrow{\tau} T'}{\langle T, P\rangle \longrightarrow \langle T', P\rangle}$$

$$\frac{P \xrightarrow{\tau} P'}{\langle T, P\rangle \longrightarrow \langle T, P'\rangle}$$

$\mathcal{O}(T, P)$ is a non-empty subset of the 2-point lattice $\top$ — $\bot$

However, there are 3 ways of lifting such lattice to its non-empty subsets:



$\wp_{\texttt{May}}$: the possibility of success is essential

$\wp_{\texttt{Must}}$: failure is disastrous

The resulting equivalences are $\simeq_{\texttt{May}}$ (**may testing**) and $\simeq_{\texttt{Must}}$ (**must testing**)

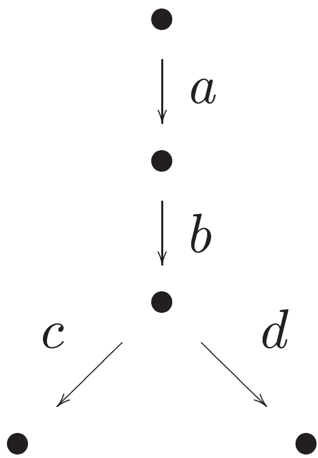
Note: $\simeq_{\texttt{Testing}}$ is $\simeq$

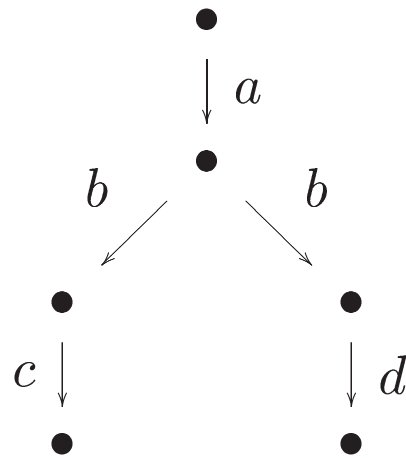
Theorem 23 1. $\simeq = (\simeq_{\texttt{May}} \cap \simeq_{\texttt{Must}})$

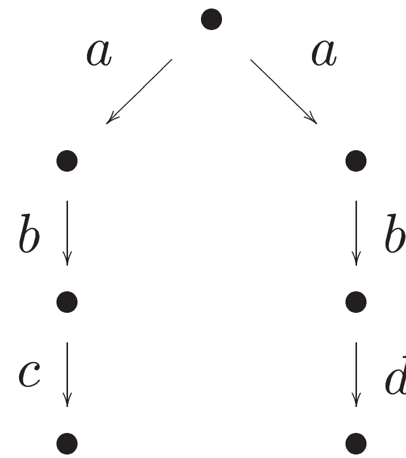
2. $\simeq_{\texttt{May}}$ coincides with trace equivalence

3. $\simeq$ coincides with failure equivalence

Example







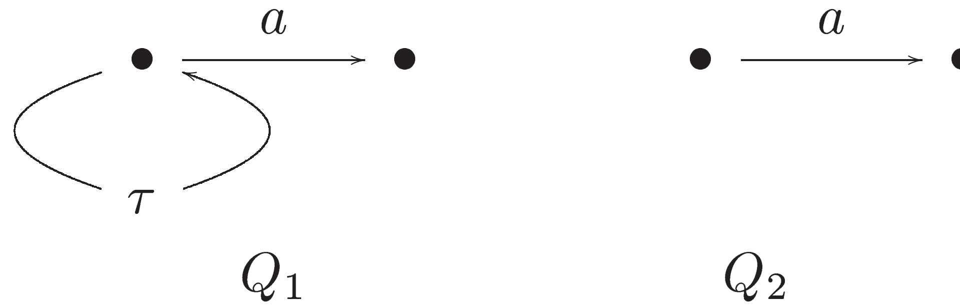
P_9 P_6 P_5

$$P_9 \simeq_{\mathtt{May}} P_5 \simeq_{\mathtt{May}} P_6$$

$$P_9 \not\simeq_{\mathtt{Must}} P_5 \simeq_{\mathtt{Must}} P_6$$

$$P_9 \not\simeq P_5 \simeq P_6$$

$$P_9 \not\sim P_5 \not\sim P_6$$



In CCS: $Q_1 = \tau^\omega \mid a$, and $Q_2 = a.\,0$

Q_1 and Q_2 are **weakly bisimilar**, but **not testing equivalent**

Justification for testing: **bisimulation is insensitive to divergence**

Justification for bisimulation: **testing is not "fair"**

(notions of fair testing have been proposed, and then bisimulation is indeed strictly included in testing)

All equivalences discussed in these lectures reduce parallelism to interleaving, in that

$$a.\,0 \mid b.\,0 \text{ is the same as } a.\,b.\,0 + b.\,a.\,0$$

Not discussed in these lectures: equivalences that refuse the above equality (called true-concurrency, or non-interleaving)