LTS-Based Testing
Part II

Mohammad Mousavi

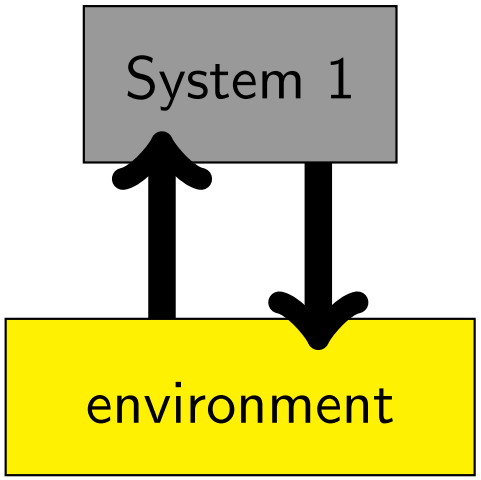
Eindhoven University of Technology, The Netherlands

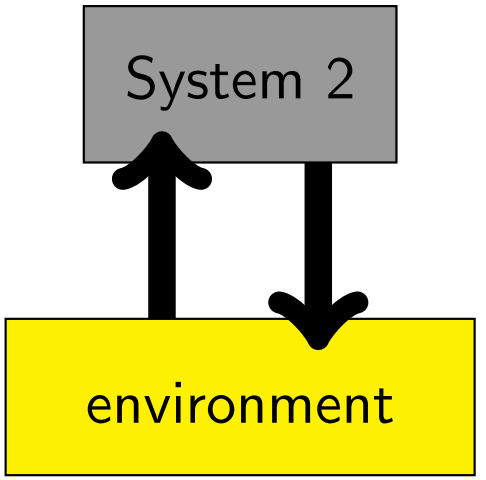
Software Testing, 2008

Overview

- Exercises
- pre-orders:
  - trace,
  - completed trace,
  - testing equivalence,
  - refusal equivalence
- Conformance relation conf
- Input/Output Transition Systems
- Conformance relation ioco
- Automated test generation for ioco

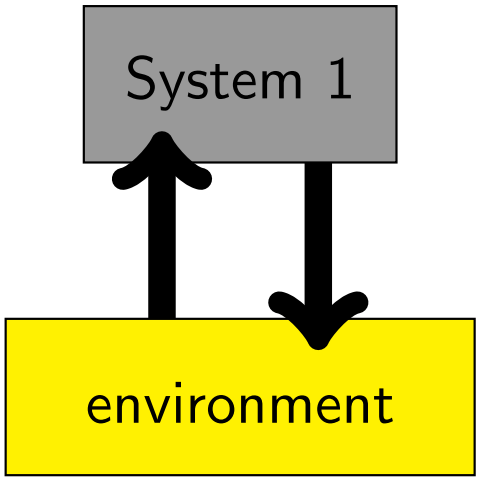
Comparing Transition Systems

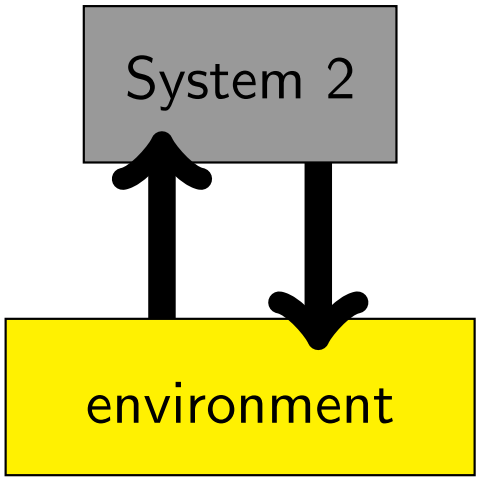




Equivalences are often not interesting in testing.
Why?

Comparing Transition Systems

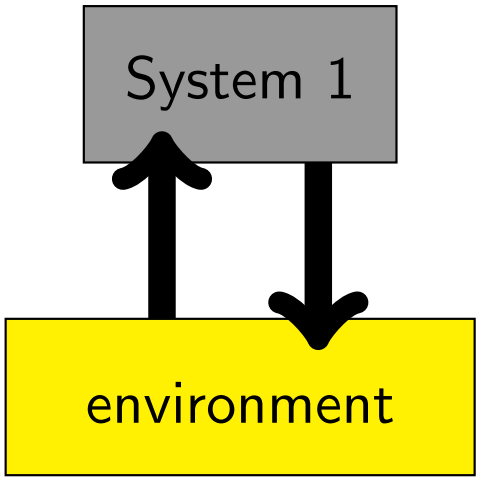


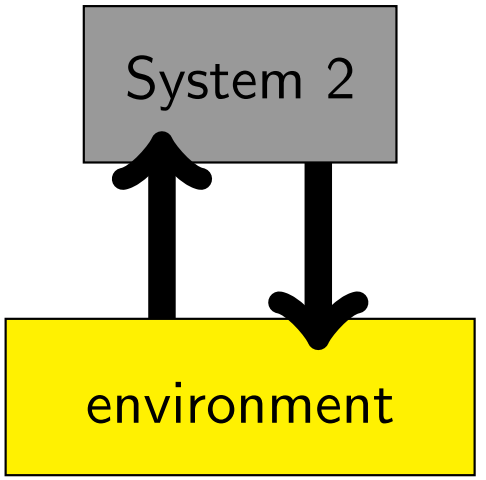


Equivalences are often not interesting in testing.
Try giving a full specification of:

- Your i-Pod,
- Amazon's webshop,
- Your favourite operating system,
- ...

Comparing Transition Systems

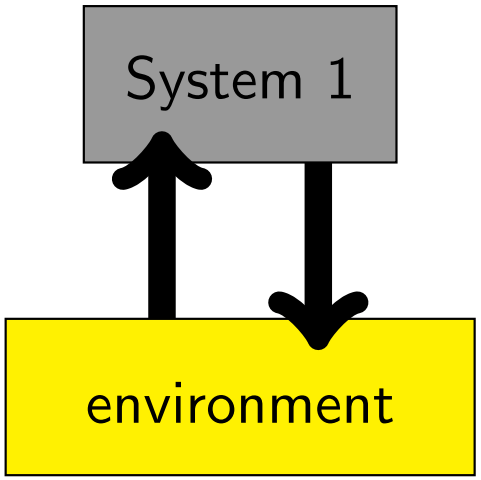


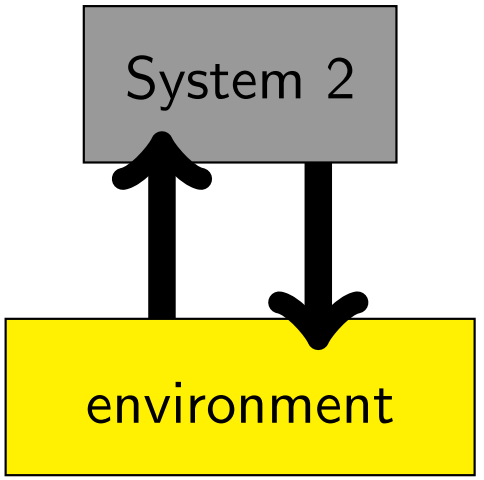


Pre-orders: a relation $\sqsubseteq$ satisfying:

- reflexivity: $x \sqsubseteq x$ for all elements x, and
- transitivity: if $x \sqsubseteq y$ and $y \sqsubseteq z$, then also $x \sqsubseteq z$

Comparing Transition Systems

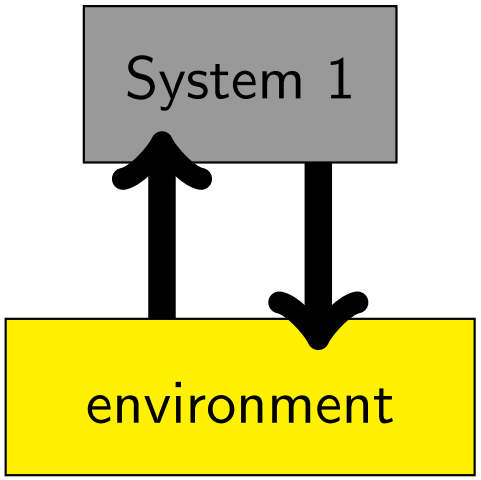


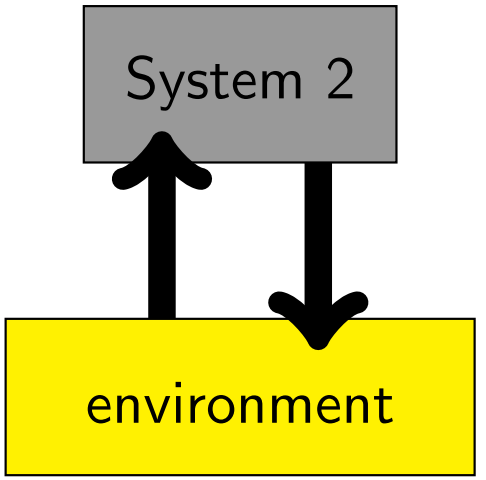


Extensional characterisations:

- Testing pre-order: $S_1 \sqsubseteq_{te} S_2$ iff for all environments e:
  $c - traces(e \,||\, S_1) \subseteq c - traces(e \,||\, S_2) \wedge$
  $traces(e \,||\, S_1) \subseteq traces(e \,||\, S_2)$
- Refusal pre-order (ext): $S_1 \sqsubseteq_{rf} S_2$ iff for all environments e:
  $c - traces(e \,||\, S_1) \subseteq c - traces(e \,||\, S_2) \wedge$
  $traces(e \,||\, S_1) \subseteq traces(e \,||\, S_2)$

Comparing Transition Systems

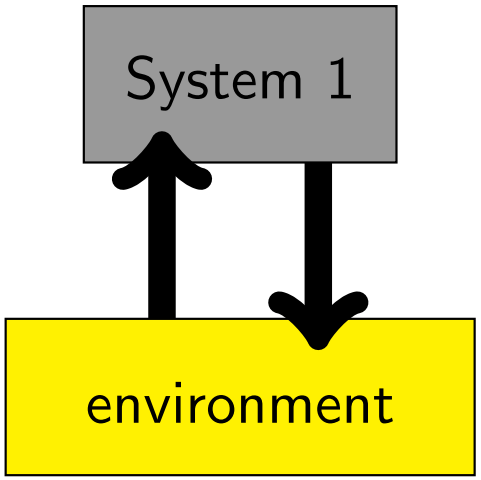


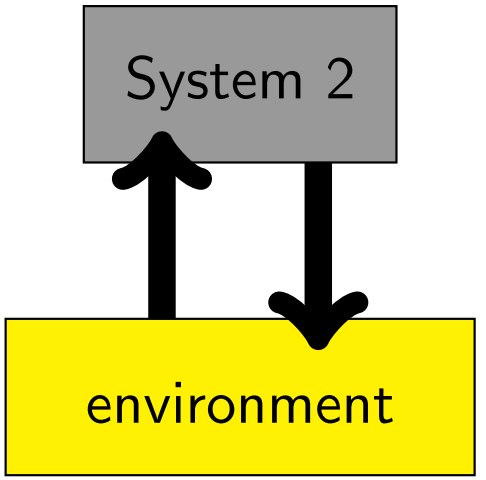


Intensional characterisations:

- Testing pre-order (int): $S_1 \sqsubseteq_{te} S_2$ iff f-pairs$(S_1) \subseteq$ f-pairs(S_2)
- Refusal pre-order (int): $S_1 \sqsubseteq_{rf} S_2$ iff f-traces$(S_1) \subseteq$ f-traces(S_2)

Comparing Transition Systems

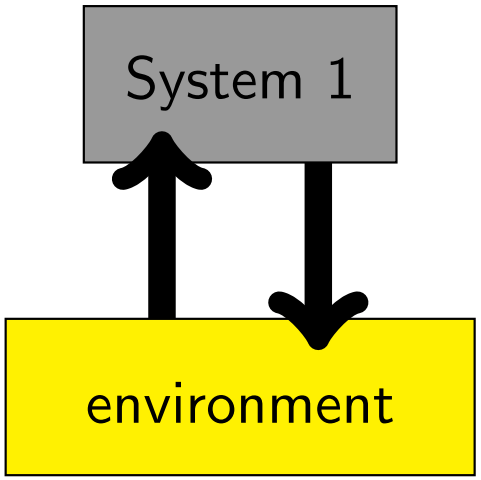


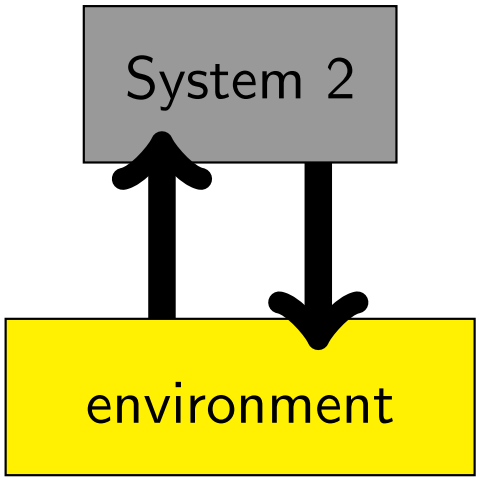


Advantages of pre-orders:

- high-level specifications (several solutions for a problem); implementation can choose one
- abstraction ("don't care" can be specified);

Comparing Transition Systems

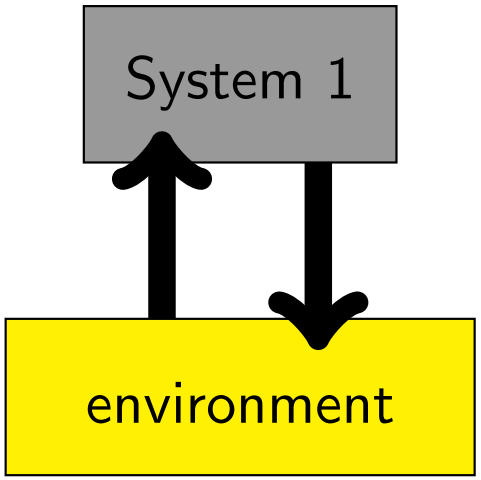


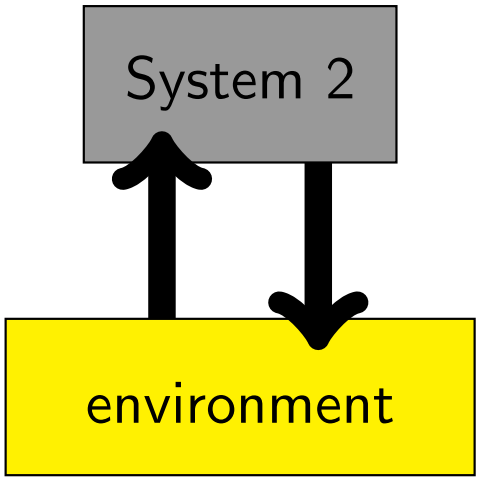


Conformance relation Conf:

Instead of testing all behaviors, test the specified behaviors

Comparing Transition Systems



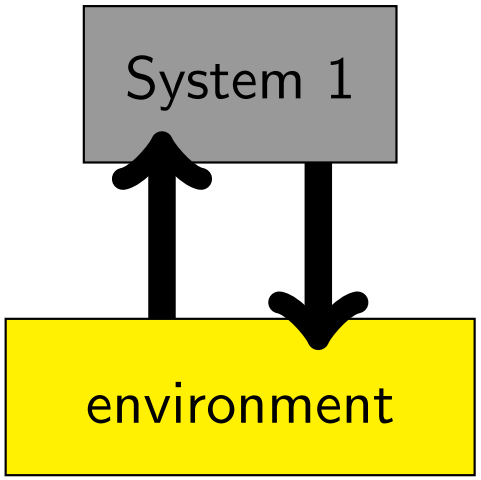


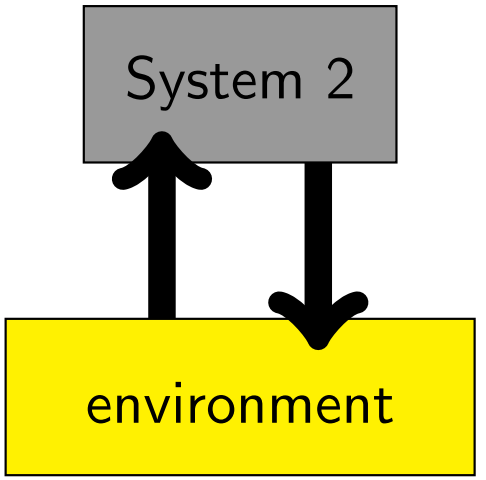
Conformance relation Conf (extensionally):

S_1 *conf* S_2 iff for all environments $e \in LTS(\mathcal{A})$:

$c - traces(e \,||\, S_1) \cap traces(S_2) \subseteq c - traces(e \,||\, S_2) \wedge$
$traces(e \,||\, S_1) \cap traces(S_2) \subseteq traces(e \,||\, S_2)$

Comparing Transition Systems





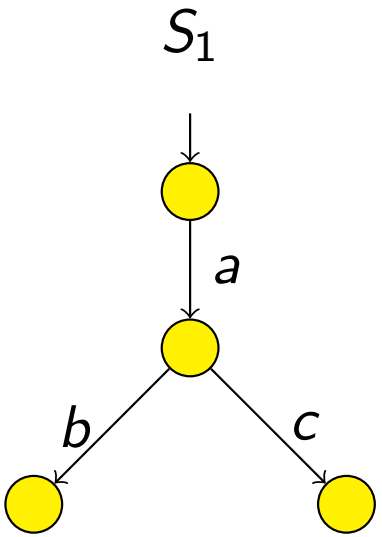
Conformance relation Conf (intensionally):

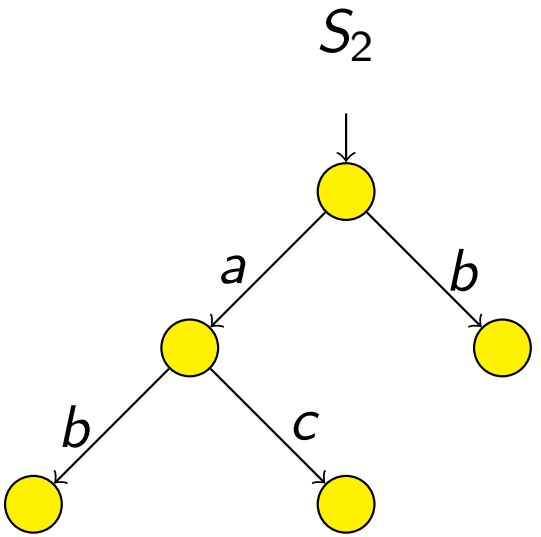
S_1 *conf* S_2 iff

$\forall \sigma \in traces(S_2) : \ \forall A \subseteq \mathcal{A} :$
if $(S_1$ *after* $\sigma)$ *refuses* A *then* $(S_2$ *after* $\sigma)$ *refuses* A

S_1 *conf* S_2 is read as:
implementation S_1 *conforms to* specification S_2.

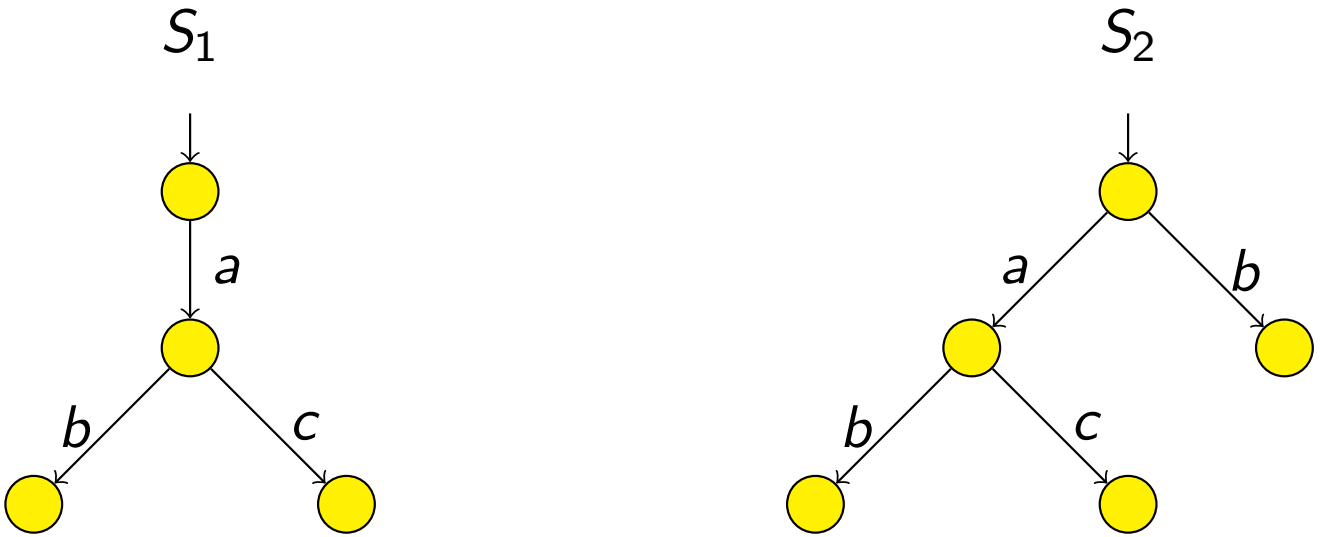
Conformance relation Conf





Question: S_1 *conf* S_2?

Conformance relation Conf

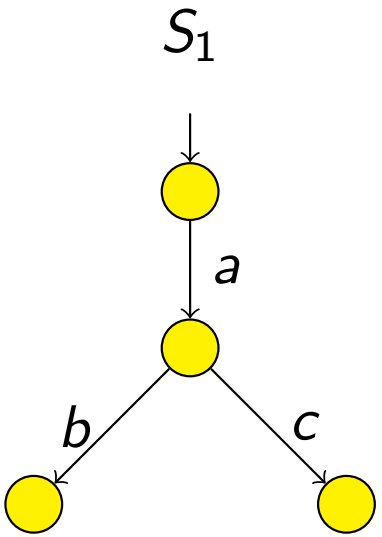


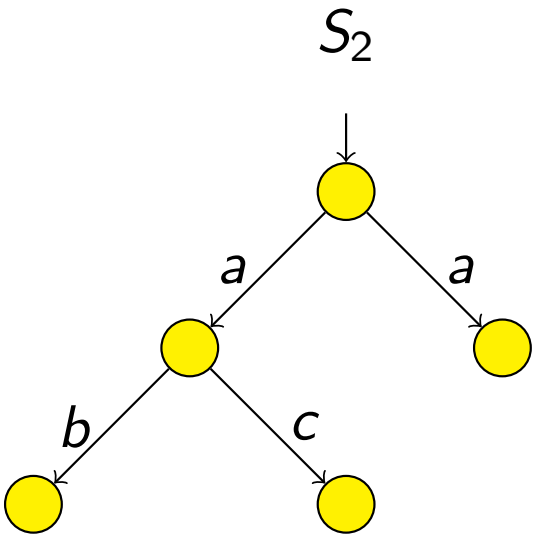
Question: S_1 *conf* S_2?

$traces(S_2) = \{\epsilon, a, b, ab, ac\}$
$(S_1$ *after* $\epsilon)$ *refuses* $\{b\}$, but not $(S_2$ *after* $\epsilon)$ *refuses* $\{b\}$

Hence: S_1 *not conf* S_2.

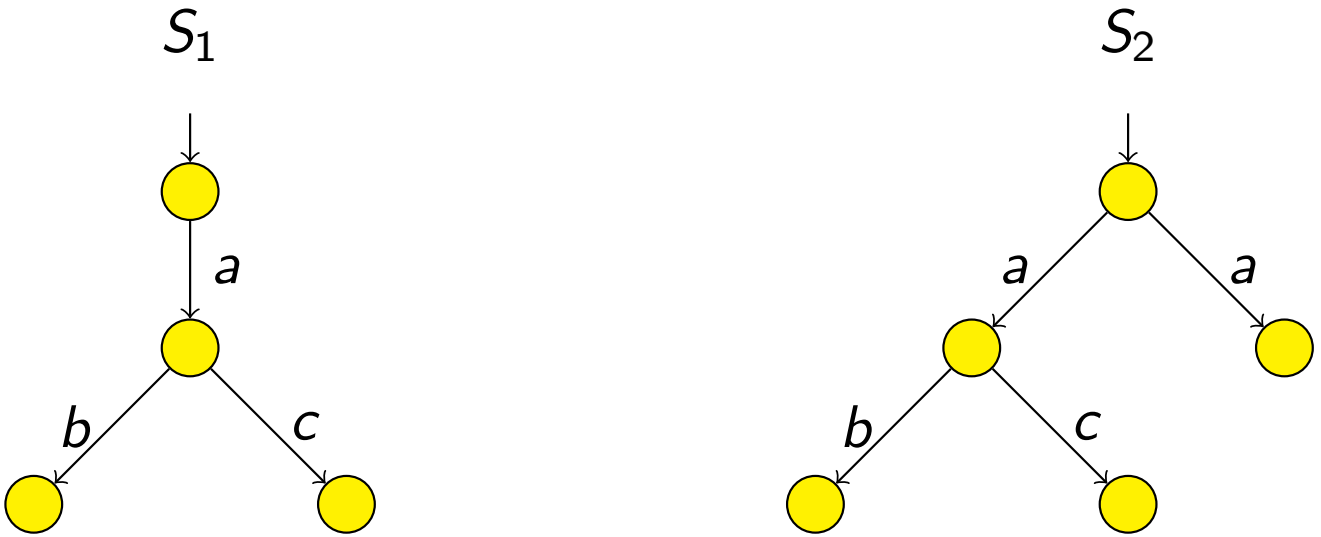
Conformance relation Conf





Question: S_1 *conf* S_2?

Conformance relation Conf



Question: S_1 *conf* S_2?

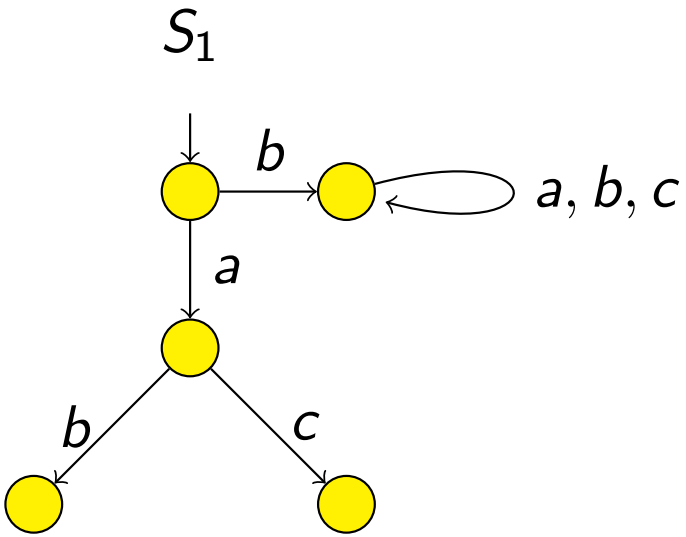
(maximal) failure-pairs w.r.t. $traces(S_2) = \{\epsilon, a, ab, ac\}$:

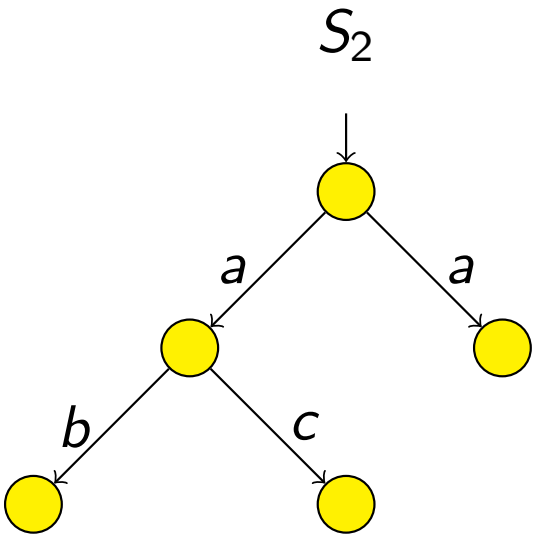
$f\text{-}pairs(S_1) : \{(\epsilon, \{b, c\}),\ (a, \{a\}),\ (ab, \{a, b, c\}),\ (ac, \{a, b, c\})\}$
$f\text{-}pairs(S_2) : \{(\epsilon, \{b, c\}),\ (a, \{a, b, c\}), (ab, \{a, b, c\}), (ac, \{a, b, c\})\}$

Note: $(a, \{a\})$ in S_1 is subsumed by $(a, \{a, b, c\})$ in S_2

Hence: S_1 *conf* S_2.

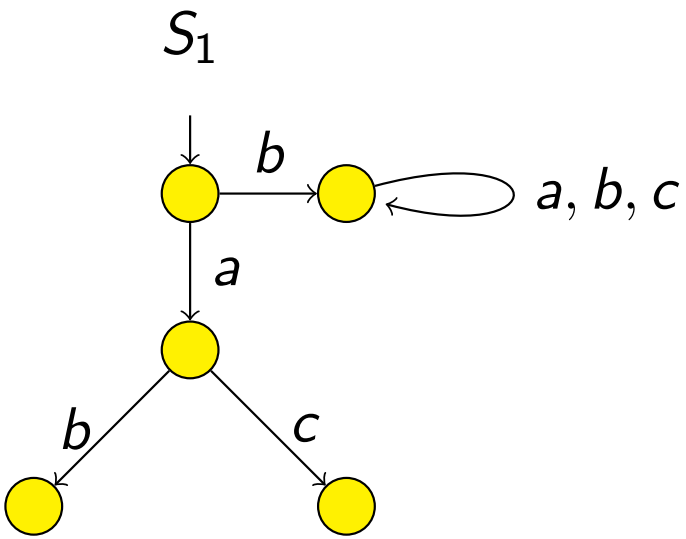
Conformance relation Conf

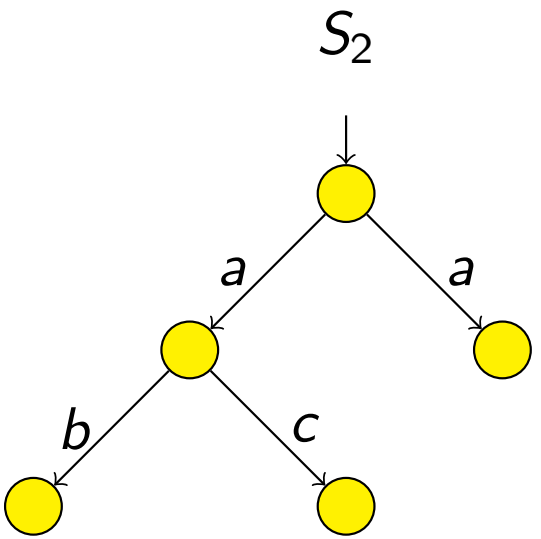




Question: S_1 *conf* S_2?

Conformance relation Conf





Question: S_1 *conf* S_2?

maximal failure-pairs w.r.t. $traces(S_2) = \{\epsilon, a, ab, ac\}$:

$f\text{-}pairs(S_1) : \{(\epsilon, \{c\}),\ (a, \{a\}),\ (ab, \{a, b, c\}),\ (ac, \{a, b, c\})\}$
$f\text{-}pairs(S_2) : \{(\epsilon, \{b, c\}),\ (a, \{a, b, c\}), (ab, \{a, b, c\}), (ac, \{a, b, c\})\}$

Hence: S_1 *conf* S_2.

Testing using LTSs

- literature mainly describes test derivation algorithms for conformance relation *conf*.
- *Conf* not used in practice.
- Ideas from *conf* and refusal pre-order used as a basis for more advanced testing theory *ioco*.

A refinement of LTS: IOTS

Most SUTs do not interact "symmetrically" with their environments but have a notion of initiative:

- Input (e.g. method call, sending messages, pressing a button)
- Output (e.g. response to a method call, retrieving of a message, playing a sound)

Distinguishing between *Inputs* and *Outputs* in LTSs leads to Input/Output Transition Systems

A refinement of LTS: IOTS

An IOTS S_1 is a five-tuple $\langle S, s_0, \mathcal{A}_I, \mathcal{A}_U, \rightarrow \rangle$:

- S is a countable set of *states*,
- $s_0 \in S$ is the initial state,
- $\mathcal{A}_I$ is a finite set of *input actions*,
- $\mathcal{A}_U$ is a finite set of *output actions*,
  - the set of all actions is denoted $\mathcal{A} = \mathcal{A}_I \cup \mathcal{A}_U$,
  - inputs and outputs are disjoint: $\mathcal{A}_I \cap \mathcal{A}_U = \emptyset$,
  - τ is neither input nor output; it's still unobservable.
- $\rightarrow \subseteq S \times \mathcal{A}_\tau \times S$ is the *transition relation*.

Note: we often write S_1 when we mean s_0, and vice versa.

Test hypothesis

1. input-enabled: all input-actions are "soon" enabled
2. realistic: add a self-loop, for disabled input

An IOTS $L = \langle S, s_0, \mathcal{A}_I, \mathcal{A}_U, \rightarrow \rangle$ is input-enabled iff:
$\forall \sigma \in \mathcal{A}^* : \forall s \in (s_0 \ \textit{after} \ \sigma) : \forall a \in \mathcal{A}_I : \ (s \ \textit{after} \ a) \neq \emptyset$

So: input-enabledness is about the *reachable states* only.

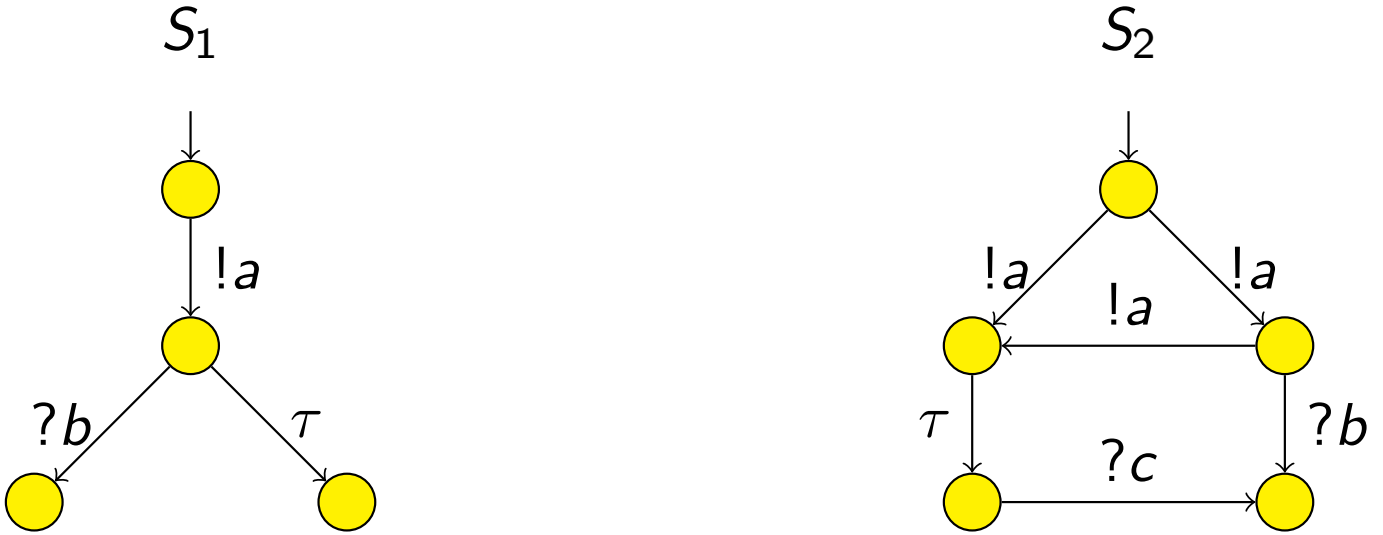
Observable behavior

Similar to refusal testing: observers detect the absence of output.

Common notation:

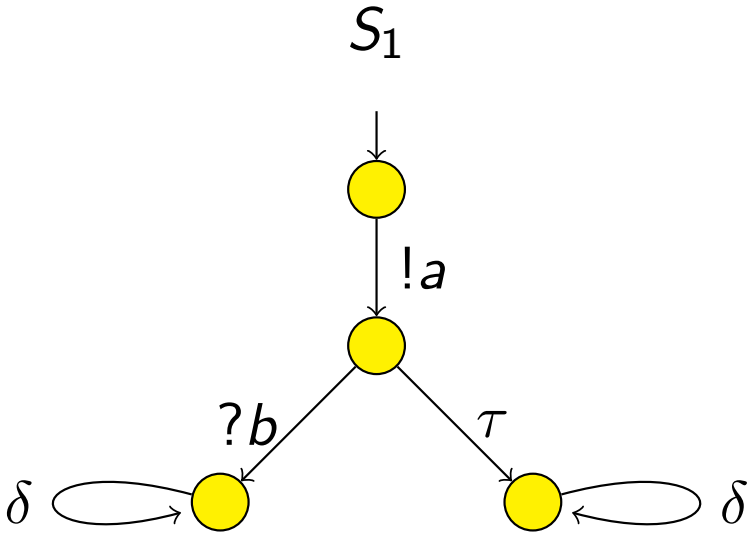
- suspension loop: $s \xrightarrow{\delta} s$ iff $\forall a \in \mathcal{A}_U \cup \{\tau\} : s \overset{a}{\not\rightarrow}$.
  A state s is quiescent iff $s \xrightarrow{\delta}$.
  Quiescence: silence, refusing to produce output.
- for a state s: $out(s) = \{a \in \mathcal{A}_U \cup \{\delta\} \mid s \xrightarrow{a}\}$;
- for a set of states S: $out(S) = \cup_{s \in S} : out(s)$
- suspension traces of state s:
  $S\text{-}traces(s) = \{\sigma \in (\mathcal{A} \cup \{\delta\})^* \mid s \ after \ \sigma \neq \emptyset\}$

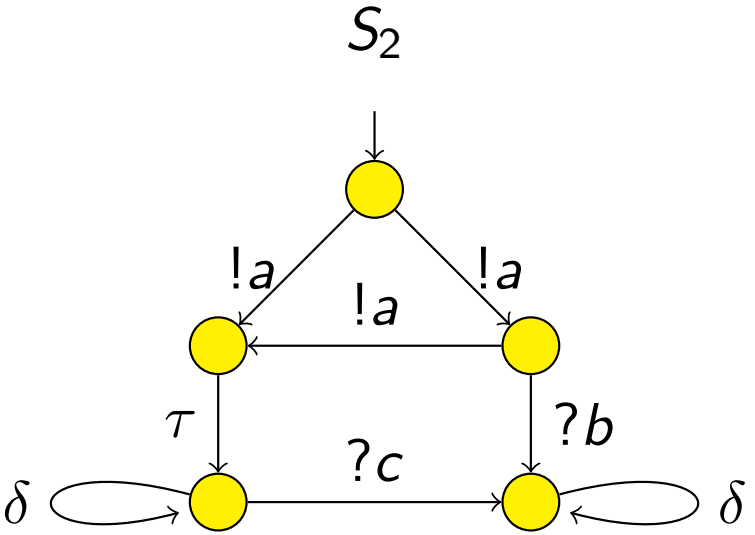
Example:



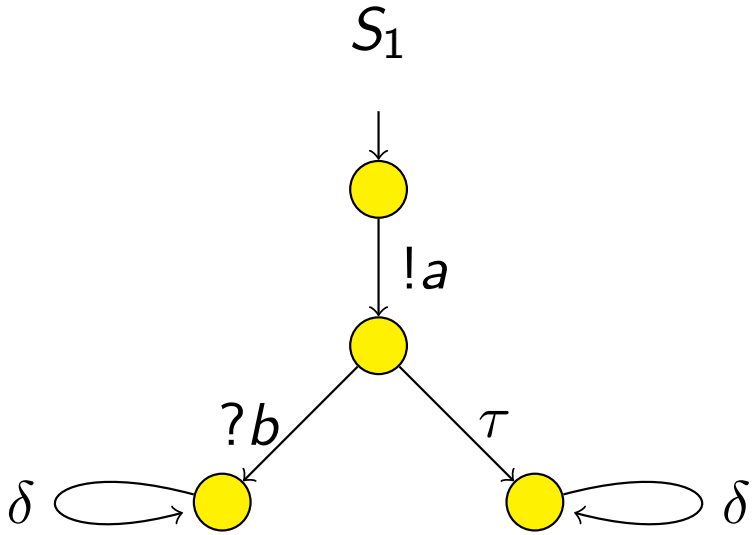
Question: Which states are *quiescent* (can be given a δ-loop)?

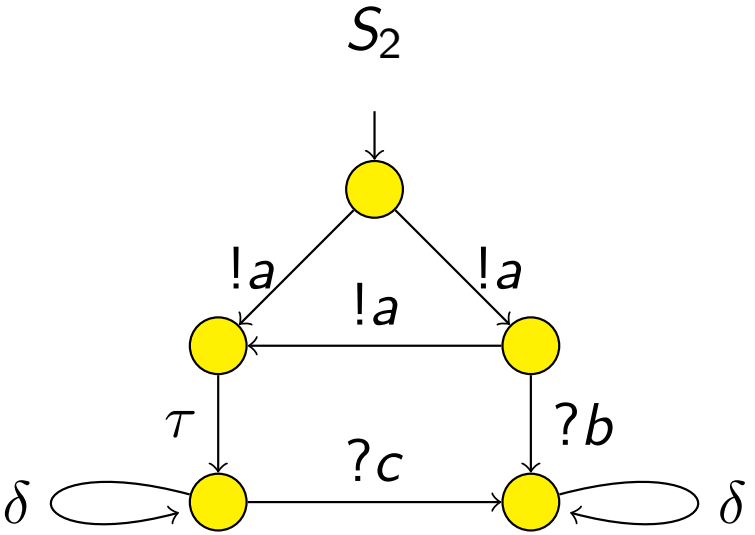
Example:





Example:

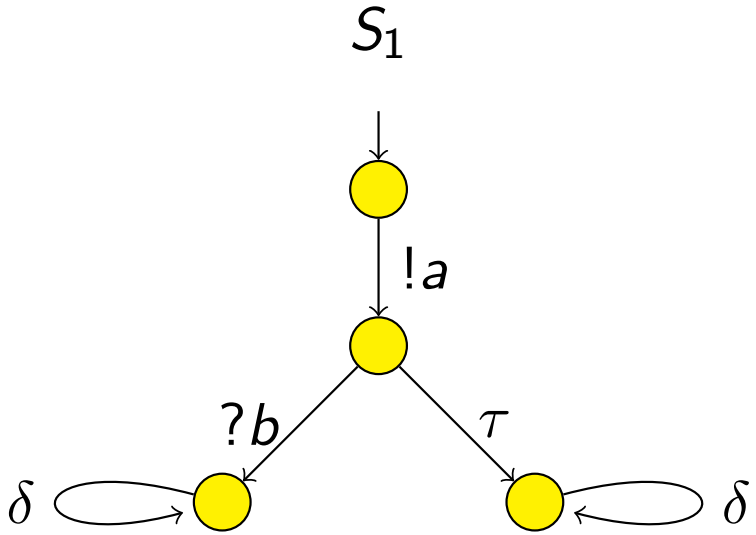


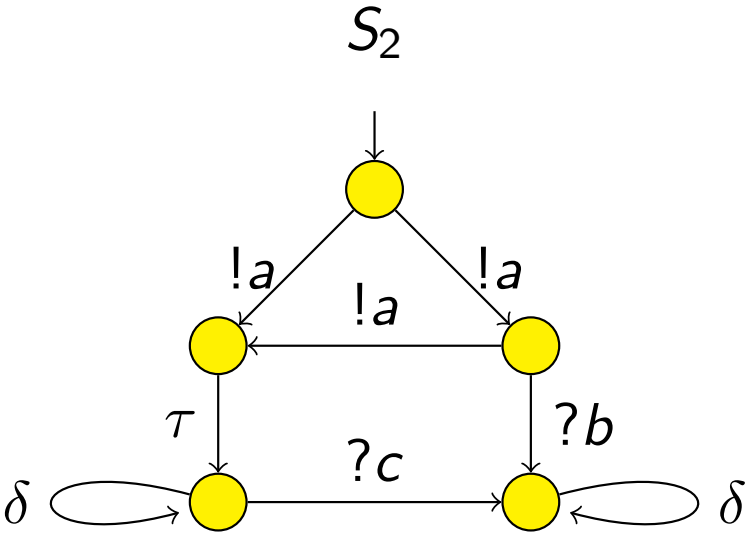


Question: Give the outcome for:

- $out(S_1 \ after \ !a) = \ ?$
- $out(S_1 \ after \ (!a \ ?b)) = \ ?$
- $out(S_2 \ after \ !a) = \ ?$
- $out(S_2 \ after \ (!a \ \delta)) = \ ?$

Example:

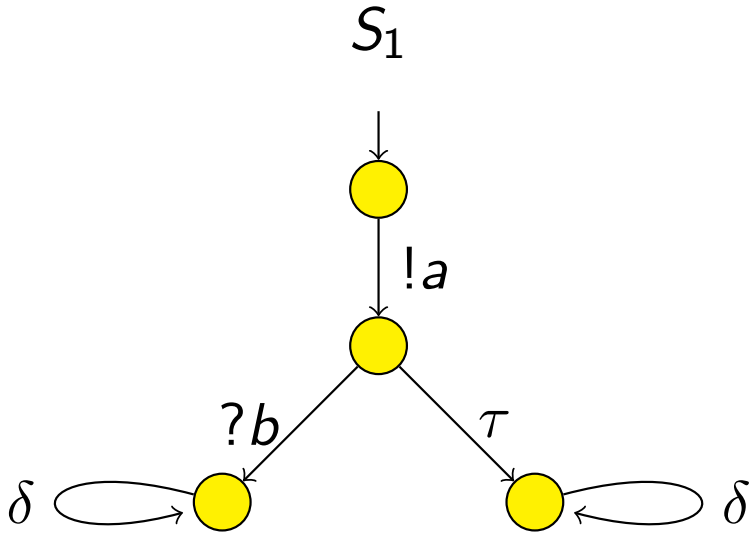


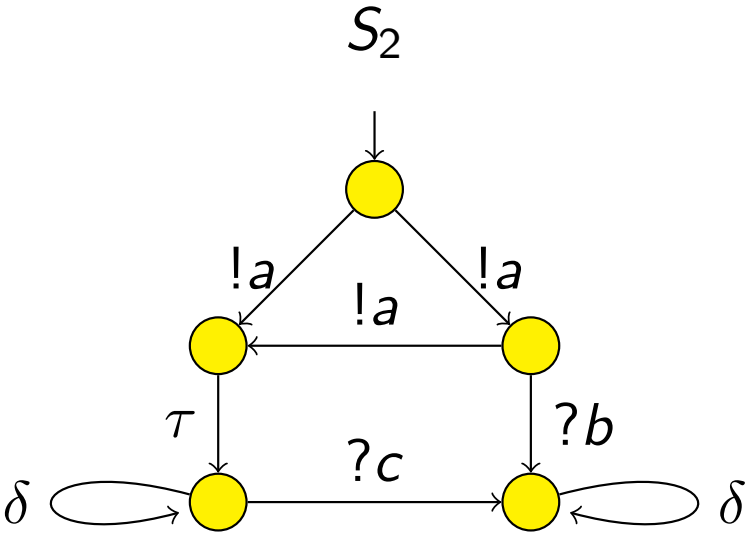


Question: Give the outcome for:

- $out(S_1 \ after \ !a) = \{\delta\}$
- $out(S_1 \ after \ (!a \ ?b)) = \{\delta\}$
- $out(S_2 \ after \ !a) = \{\delta, !a\}$
- $out(S_2 \ after \ (!a \ \delta)) = \{\delta\}$

Example:

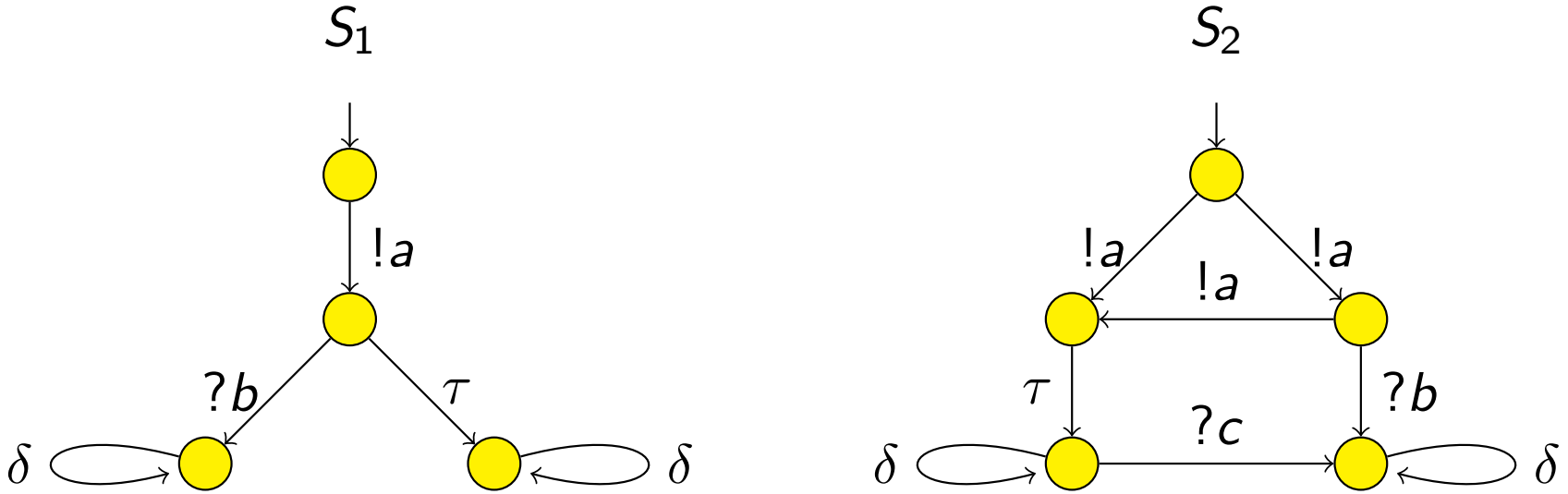




Question: Give the outcome for:

- S-traces(S_1) = ?
- S-traces(S_2) = ?

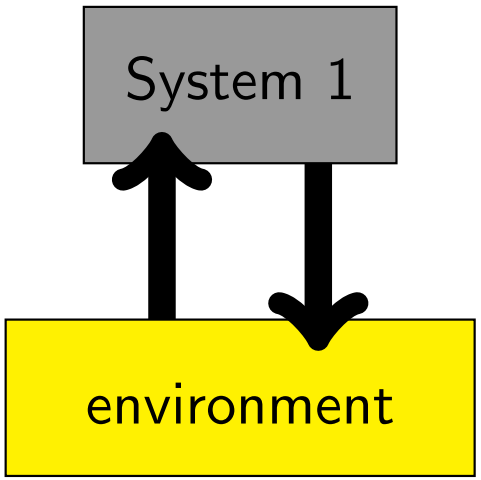
Example:

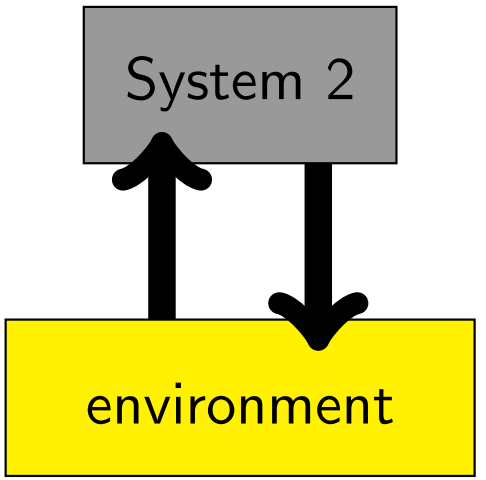


Question: Give the outcome for:

- S-traces(S_1) $= \{\epsilon\} \cup !a\ \delta^* \cup !a\ ?b\ \delta^*$
- S-traces(S_2) $=$
  $\{\epsilon\} \cup !a\ \delta^* \cup !a\ !a\ \delta^* \cup !a\ ?b\ \delta^* \cup !a\ \delta^*\ ?c\ \delta^* \cup !a\ !a\ \delta^*\ ?c\ \delta^*$

Conformance Relation: ioco





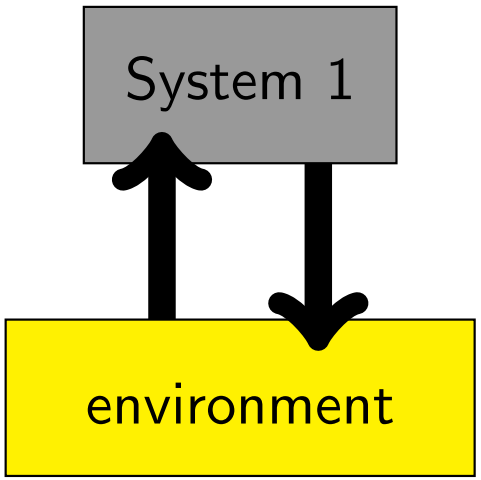
The conformance relation **ioco** takes inspiration from:

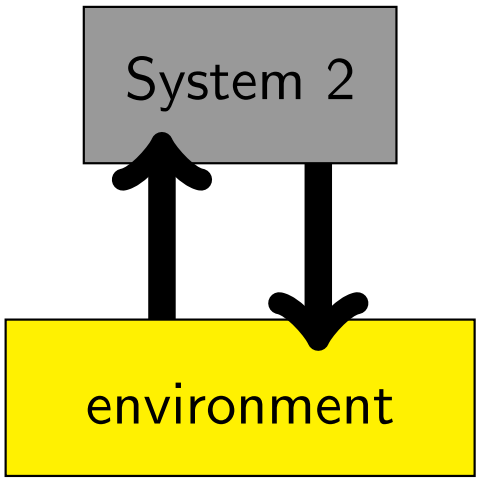
- conformance relation *conf* ,
- *refusal pre-order*: observer can see absence of output

  failure-traces of an LTS $\in (\mathcal{A} \cup 2^{\mathcal{A}})^*$.
  failure-traces of an IOTS $\in (\mathcal{A}_I \cup \mathcal{A}_U \cup 2^{\mathcal{A}_I \cup \mathcal{A}_U})^*$.

  Failure $\{\mathcal{A}_U\}$ is replaced by the special symbol δ.

Conformance Relation: ioco





Intensional characterisation:

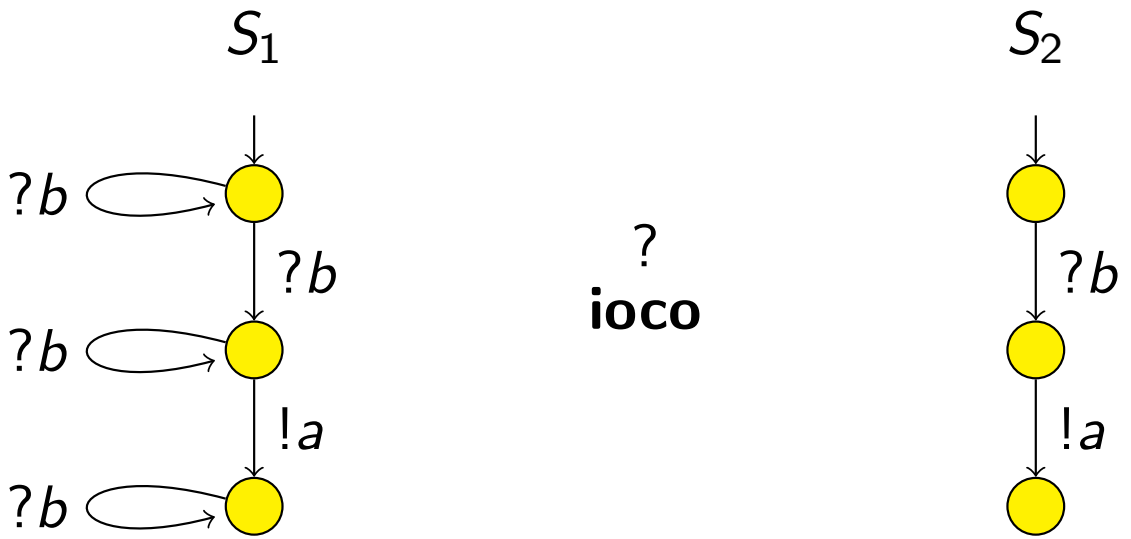
Implementation S_1 is **ioco**-conform to specification S_2 iff:

$\forall \sigma \in \text{S-traces}(S_2) : \; out(S_1 \; after \; \sigma) \subseteq out(S_2 \; after \; \sigma)$

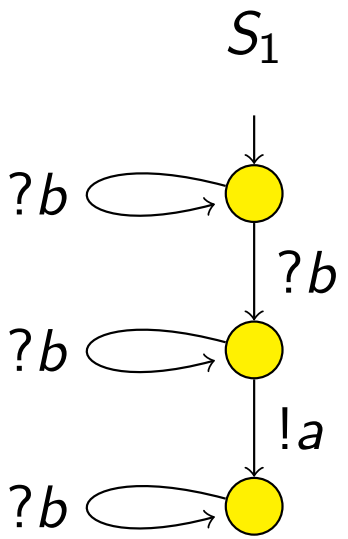
Take care:

- implementation S_1 is assumed to behave as an *input-enabled* IOTS.
- specification S_2 can be an arbitrary IOTS.

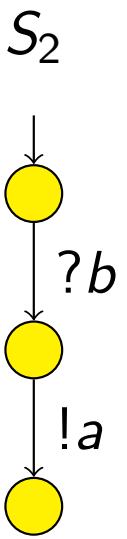
Example



Example



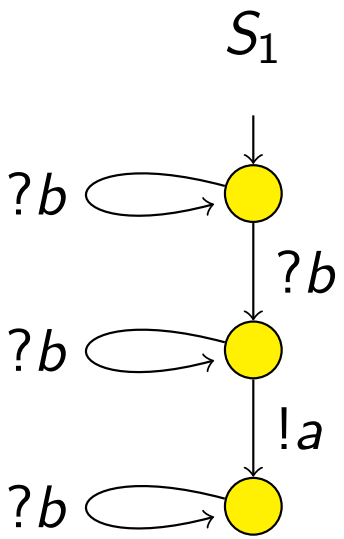
not ioco



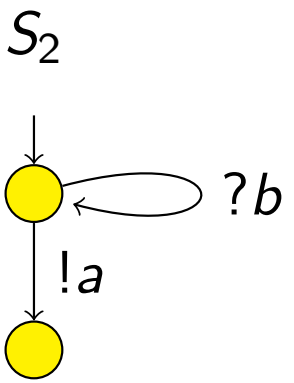
Counterexample:

$out(S_1 \ after \ ?b) = \{\delta, !a\} \not\subseteq \{!a\} = out(S_2 \ after \ ?b)$

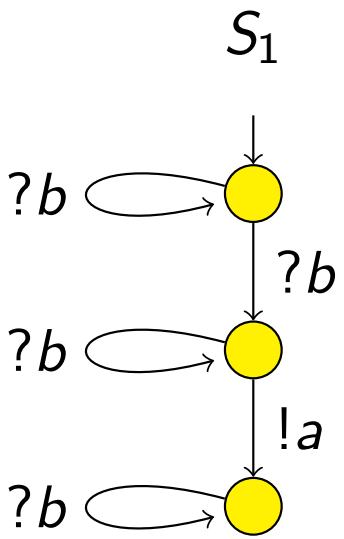
Example



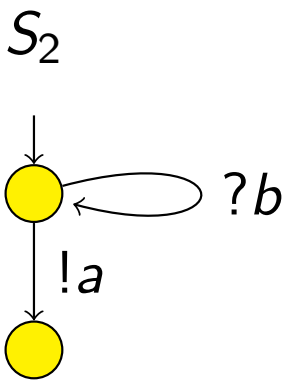
?
ioco



Example



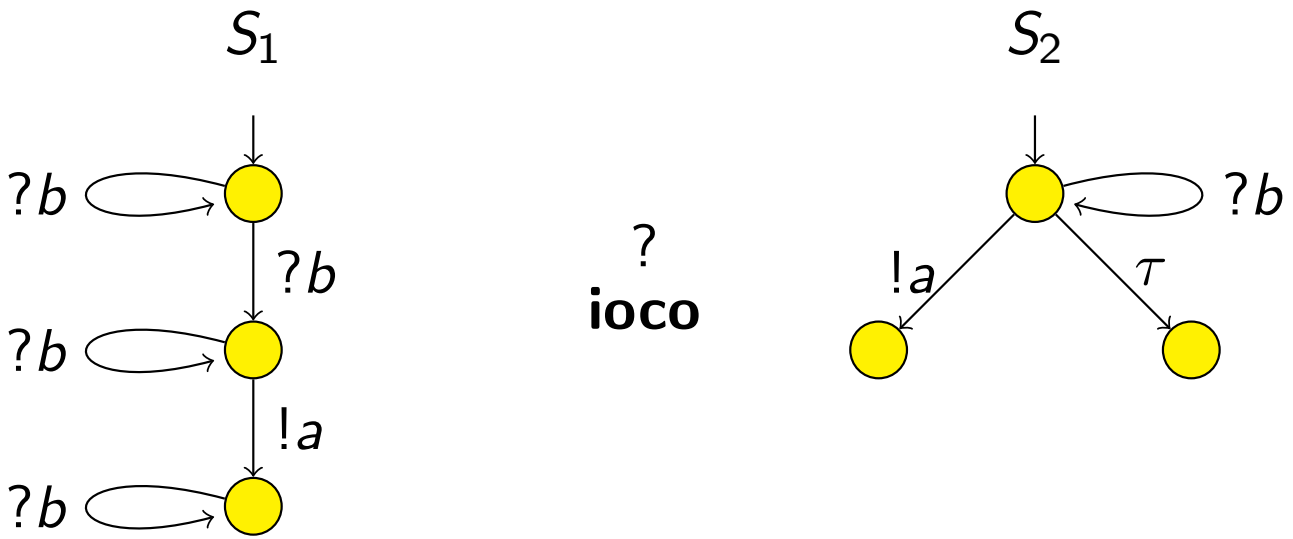
not ioco



Counterexample:

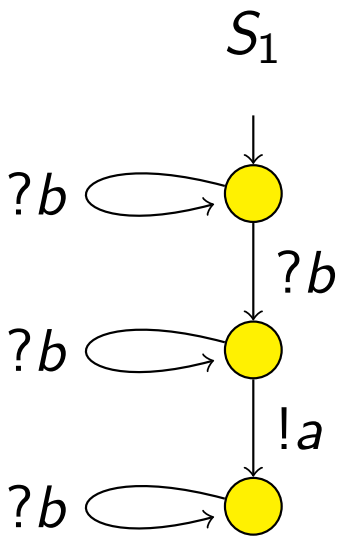
$out(S_1 \text{ after } \epsilon) = \{\delta\} \not\subseteq \{!a\} = out(S_2 \text{ after } \epsilon)$

Example

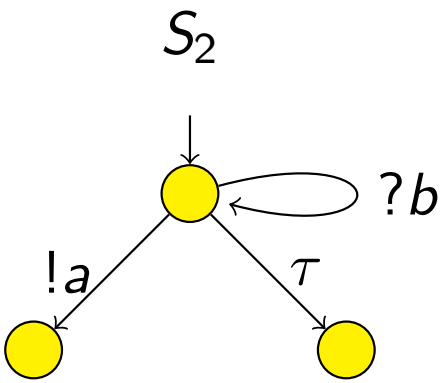


Suspension traces of S_2: $\{\epsilon\} \cup ?b\ \delta^* \cup ?b^*\ !a\ \delta^*$

Example



ioco



Suspension traces of S_2: $\{\epsilon\} \cup ?b\ \delta^* \cup ?b^*\ !a\ \delta^*$

$out(S_1\ after\ \epsilon) = \{\delta\}$	$out(S_2\ after\ \epsilon) = \{!a, \delta\}$
$out(S_1\ after\ ?b) = \{!a, \delta\}$	$out(S_2\ after\ ?b) = \{!a, \delta\}$
$out(S_1\ after\ ?b\ \delta^*) = \{\delta\}$	$out(S_2\ after\ ?b\ \delta^*) = \{\delta\}$
$out(S_1\ after\ !a\ \delta^*) = \emptyset$	$out(S_2\ after\ !a\ \delta^*) = \{\delta\}$
$out(S_1\ after\ ?b\ !a\ \delta^*) = \{\delta\}$	$out(S_2\ after\ ?b\ !a\ \delta^*) = \{\delta\}$

Black-box testing for ioco

- Wasn't the system-under-test black-box?

Black-box testing for ioco

- Wasn't the system-under-test black-box?
- So why can we verify that the SUT is **ioco**-correct?

Black-box testing for ioco

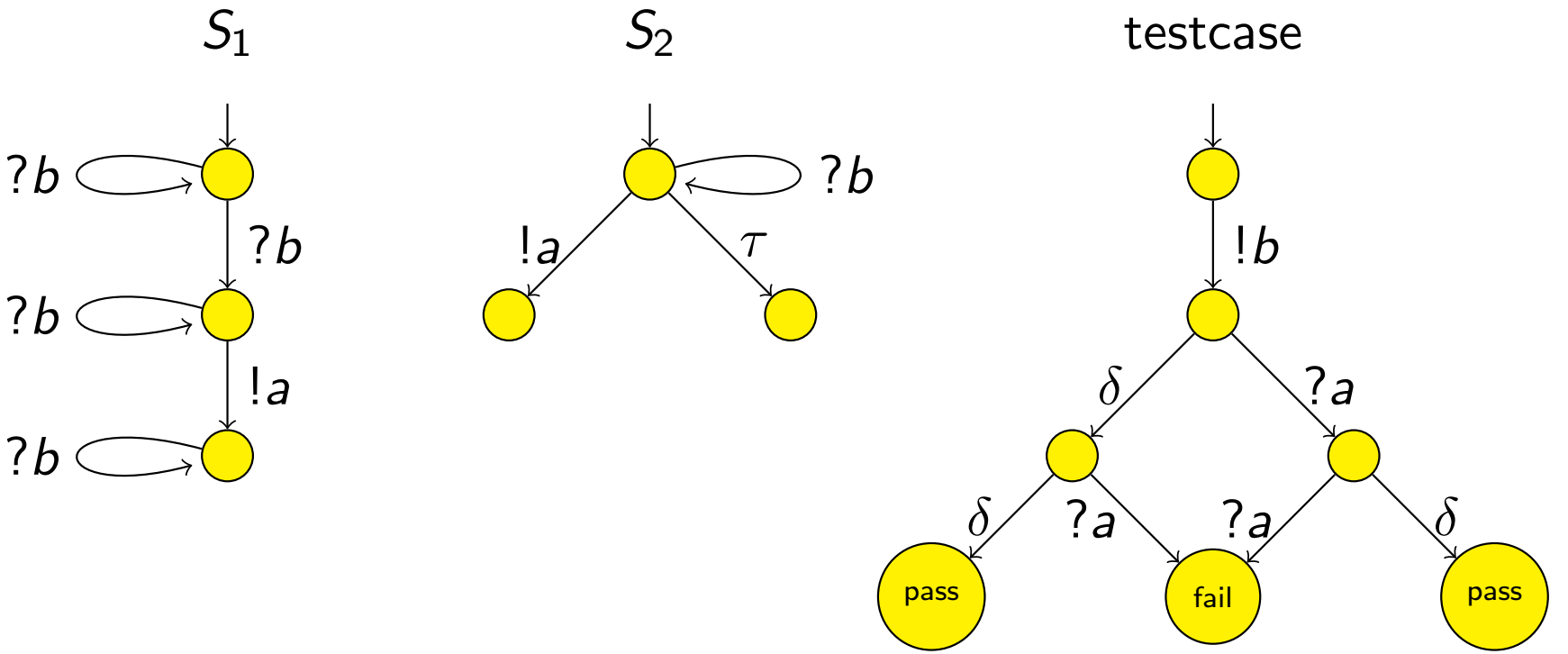
- Wasn't the system-under-test black-box?
- So why can we verify that the SUT is **ioco**-correct?
- Answer: we can't verify, so we validate by means of testing

Black-box testing for ioco

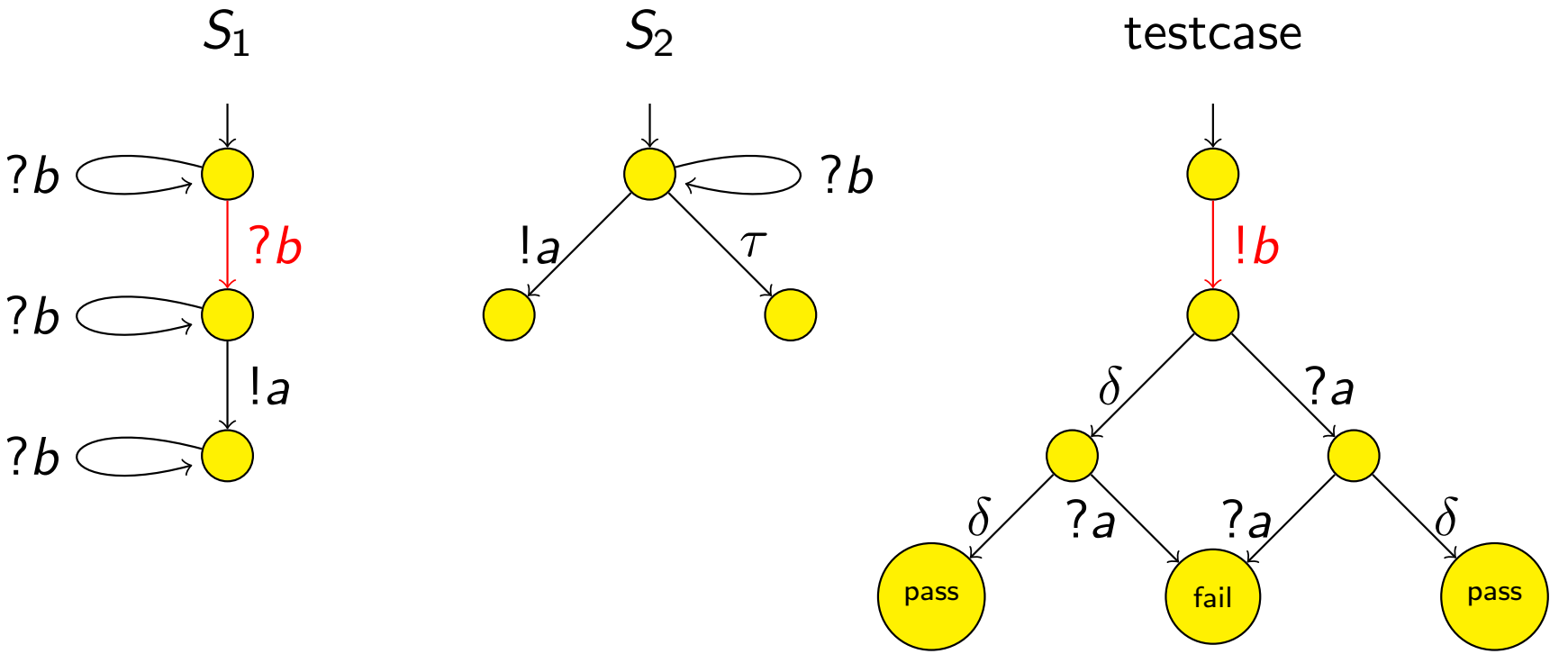
Formalising test-cases for **ioco**:

- Input-Output Transition Systems with terminal states (verdicts) pass and fail,
- Inputs of the implementation are the outputs of the test cases, outputs of the implementation are the inputs of the test cases,
- Special input label δ for synchronization with output δ,
- All completed traces end in states pass or fail,
- Deterministic, so a test run does not yield ambiguous verdicts,
- Finite behavior, so a test run always terminates.

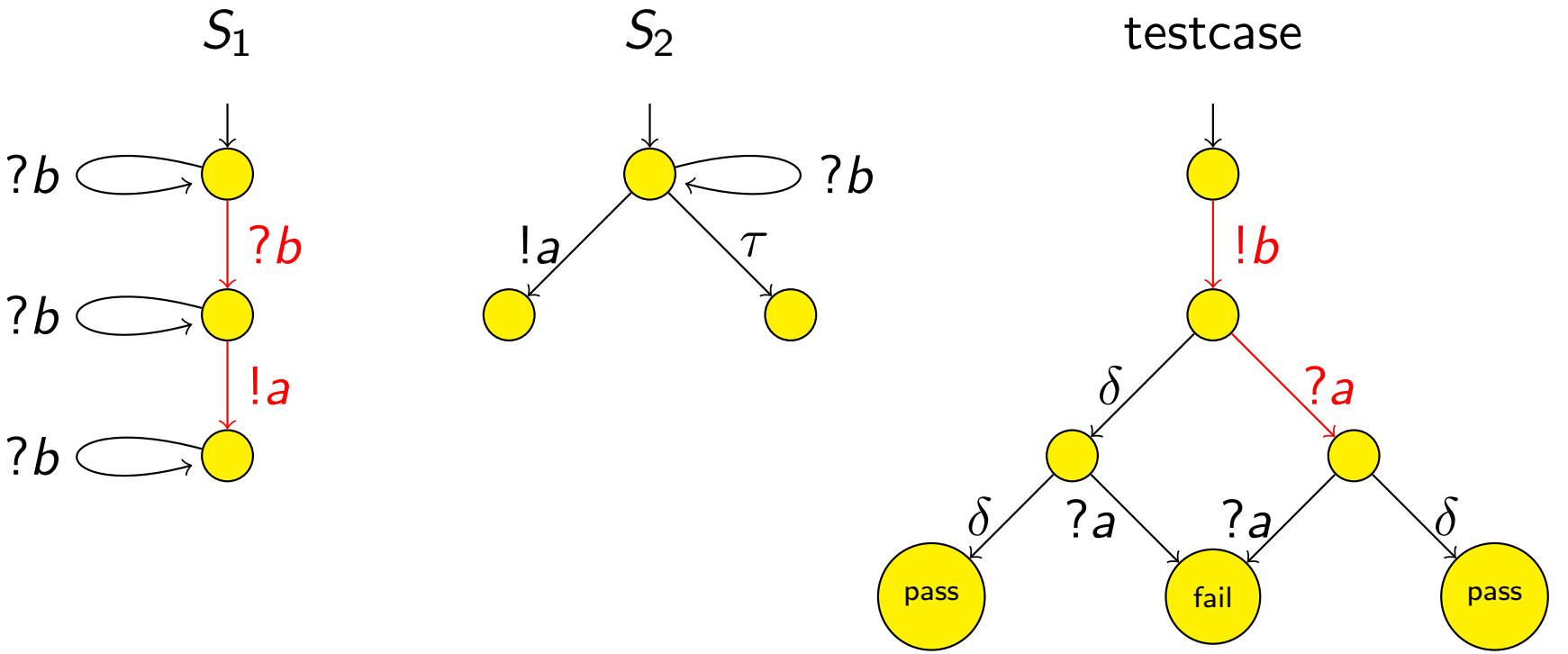
Example



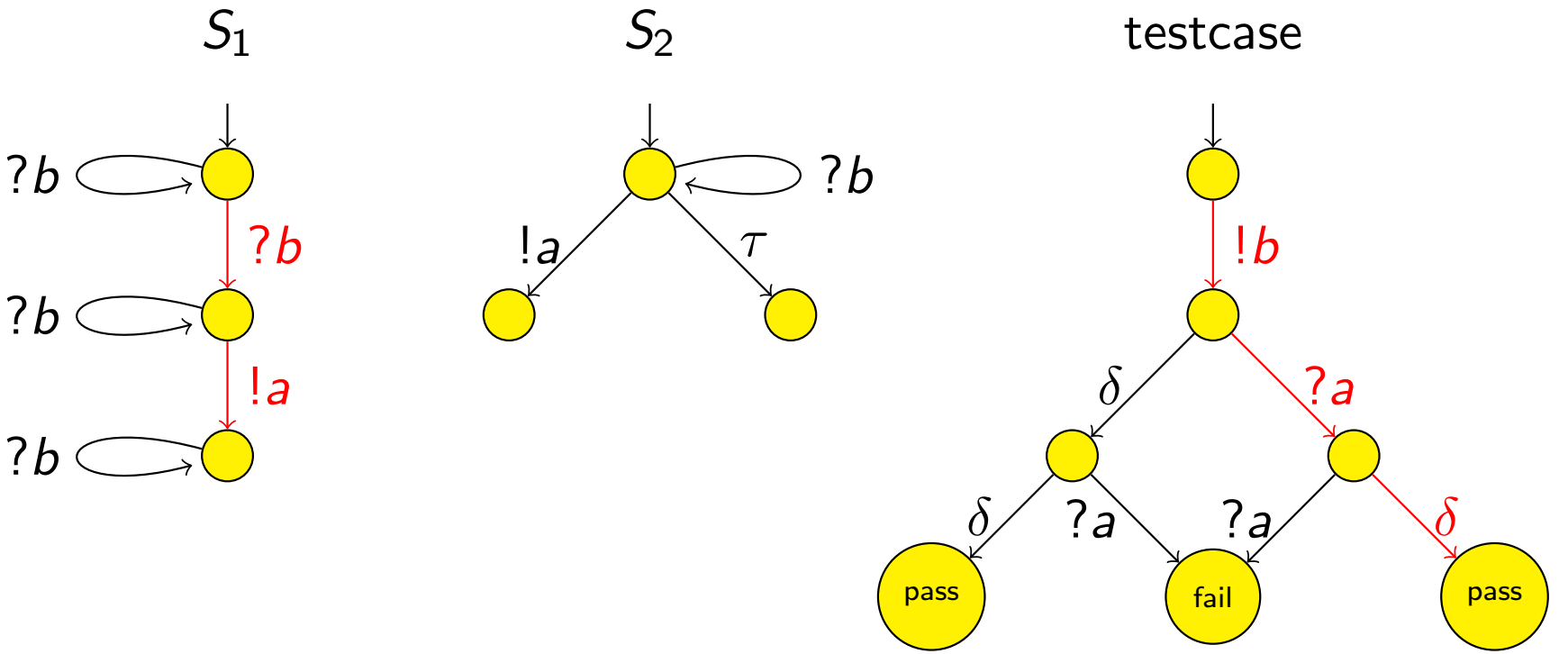
Example



Example



Example



Black-box testing for ioco

Terminology:

- a test suite is a set of test cases.
- an implementation S_1 passes a test case t:
  S_1 *passes* $t \stackrel{def}{=} \forall \sigma \in \mathcal{A}_\delta^* : \forall S_1' : (t||S_1) \stackrel{\sigma}{\not\Longrightarrow} (fail||S_1')$
- an implementation S_1 passes a test suite T:
  S_1 *passes* $T \stackrel{def}{=} \forall t \in T : S_1$ *passes* t
- an implementation S_1 fails a test suite T:
  S_1 *fails* $T \stackrel{def}{=} \exists t \in T : S_1$ *not passes* t

Black-box testing for ioco

Instead of "proving" **ioco**-conformance between a white-box IOTS implementation S_1 and an LTS specification S_2, we test for **ioco**-conformance by generating a sound and complete test suite T:

$$S_1 \textbf{ ioco } S \quad \text{iff} \quad S_1 \textit{ passes } T$$

Recursive Test Generation Algorithm

Given a specification $S_2 = \langle S, s_0, \mathcal{A}_I, \mathcal{A}_U, \rightarrow \rangle$.
$Gen(S' : 2^S)$ generates, recursively, a single test case from S_2.

INIT: $Gen(s_0$ *after* $\epsilon)$
RECURSION (roughly):

At each point in the recursion,
choose non-deterministically between:

1. Stopping the recursion,
2. Supplying an input,
3. Observing an output

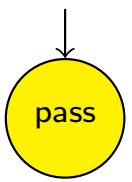
Note: S' is the set of states the specification can be in after a sequence of observable actions (initially: observation ϵ)

Recursive Test Generation Algorithm

Given a specification $S_2 = \langle S, s_0, \mathcal{A}_I, \mathcal{A}_U, \rightarrow \rangle$.
$Gen(S' : 2^S)$ generates, recursively, a single test case from S_2.

INIT: $Gen(s_0$ *after* $\epsilon)$
RECURSION: Let t be the initial state of $Gen(S')$.

1. Stop recursion. Set $t :=$ *pass*.

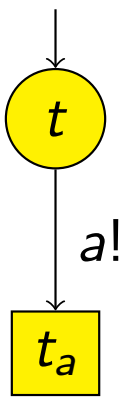


Recursive Test Generation Algorithm

Given a specification $S_2 = \langle S, s_0, \mathcal{A}_I, \mathcal{A}_U, \rightarrow \rangle$.
$Gen(S' : 2^S)$ generates, recursively, a single test case from S_2.

INIT: $Gen(s_0 \ \textit{after} \ \epsilon)$
RECURSION: Let t be the initial state of $Gen(S')$.

2. Input: create a transition $t \xrightarrow{!a} t_a$, where
   - $?a$ is an input satisfying $(S' \ \textit{after} \ ?a) \neq \emptyset$, and,
   - t_a is the initial state of IOTS $Gen(S' \ \textit{after} \ ?a)$.

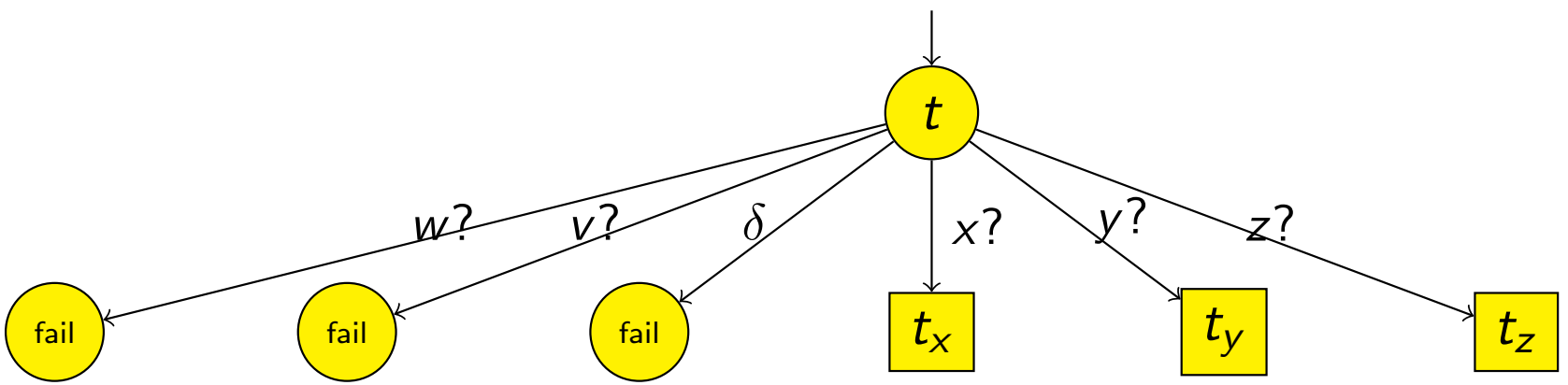


Recursive Test Generation Algorithm

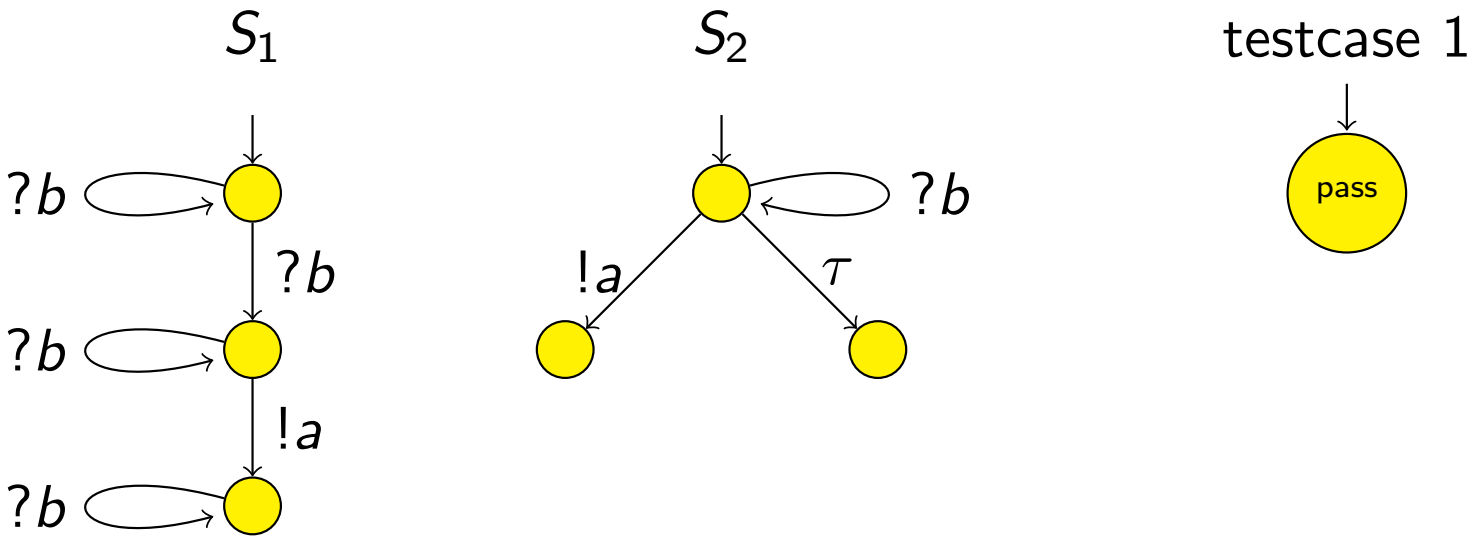
Given a specification $S_2 = \langle S, s_0, \mathcal{A}_I, \mathcal{A}_U, \rightarrow \rangle$.
$Gen(S' : 2^S)$ generates, recursively, a single test case from S_2.

INIT: $Gen(s_0$ *after* $\epsilon)$
RECURSION: Let t be the initial state of $Gen(S')$.

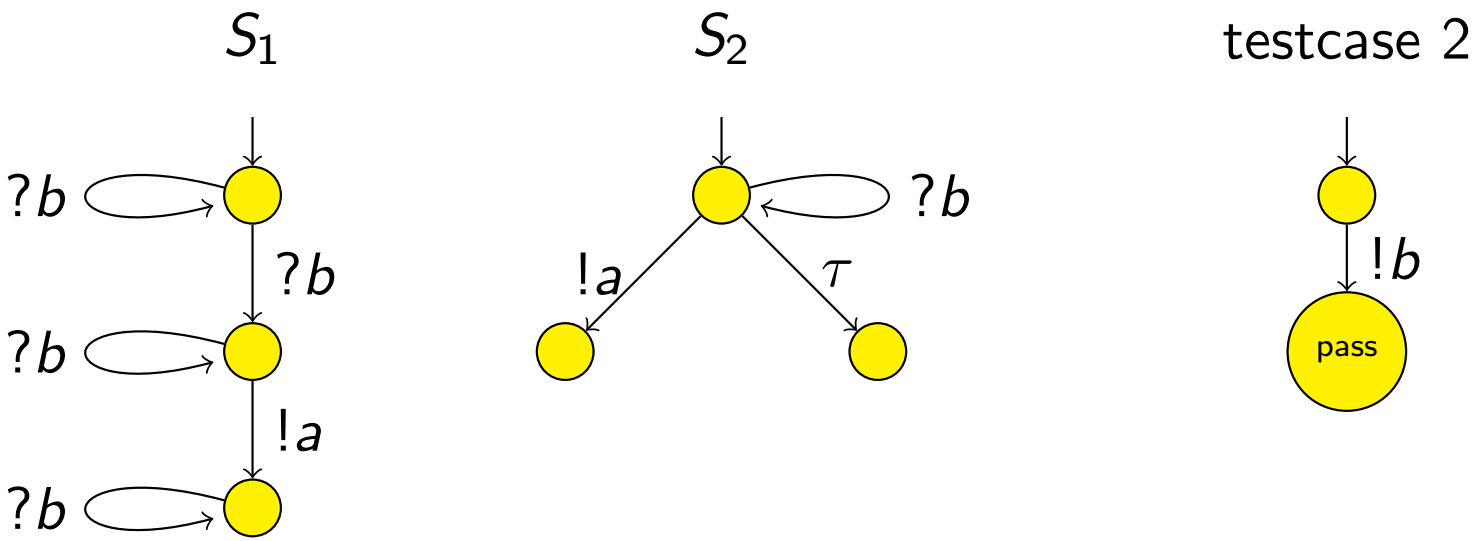
3. Output: create a transition $t \xrightarrow{?x} t_x$ for all $x \in \mathcal{A}_U \cup \{\delta\}$:
   - t_x is state fail if $!x \notin out(S')$, and
   - t_x is the initial state of IOTS $Gen(S'$ *after* $!x)$ otherwise.



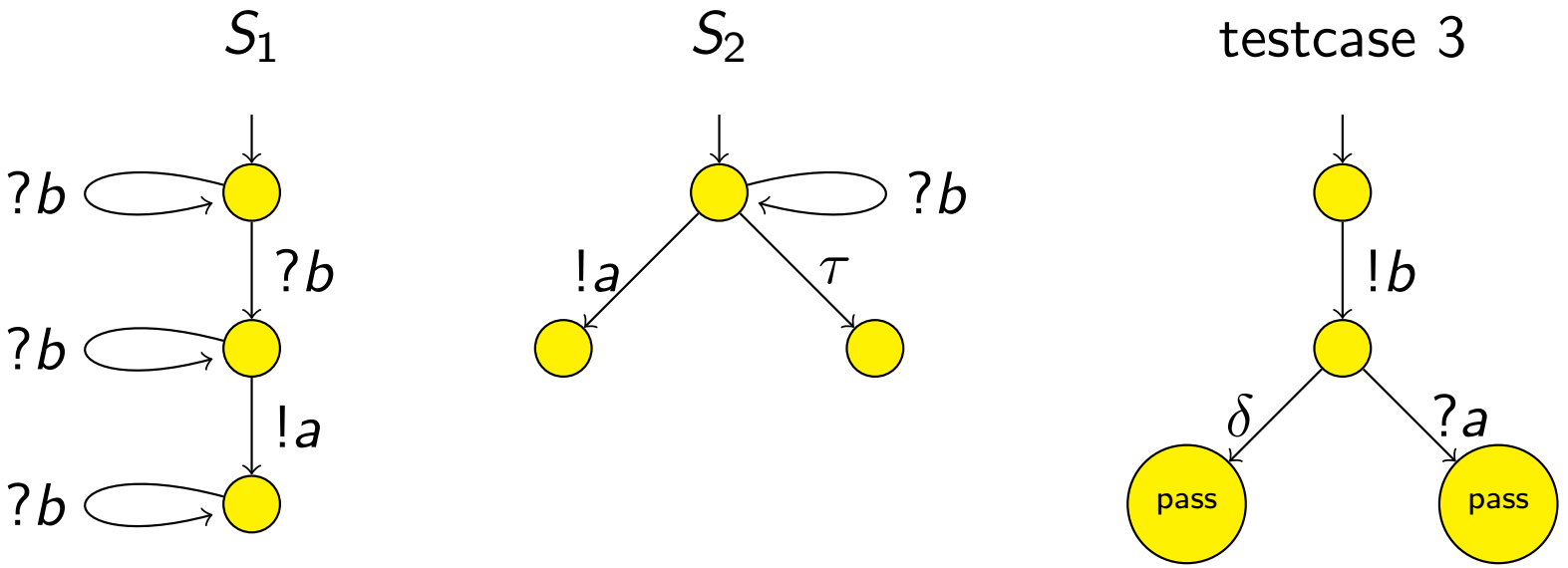
Example



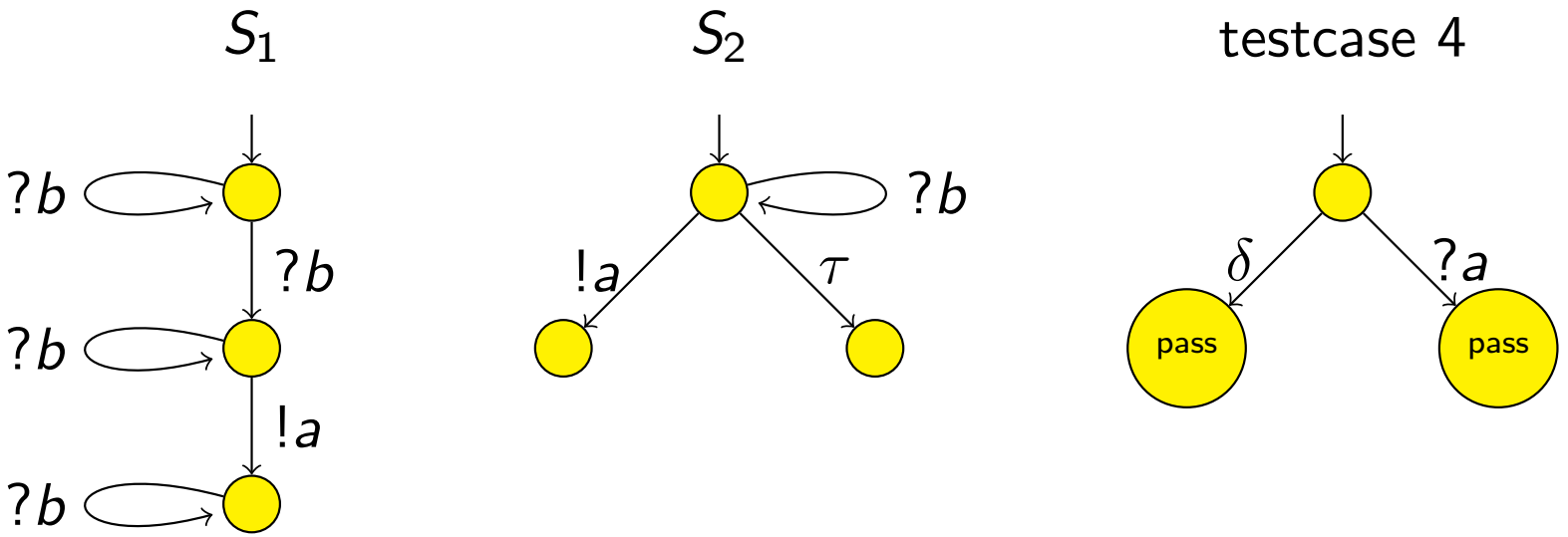
Example



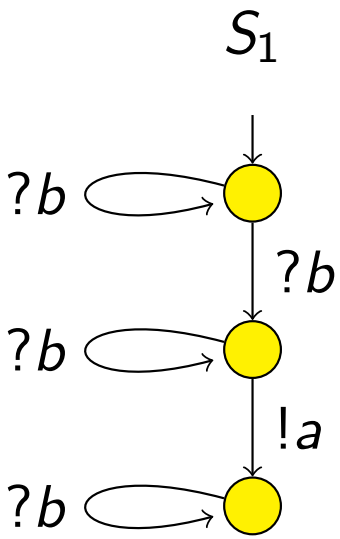
Example

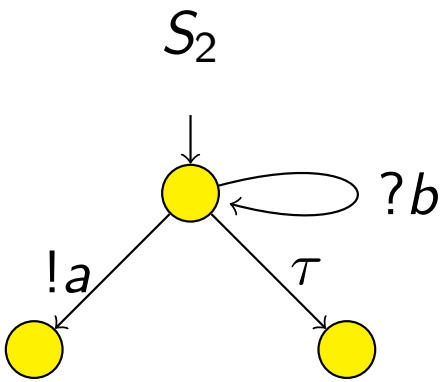


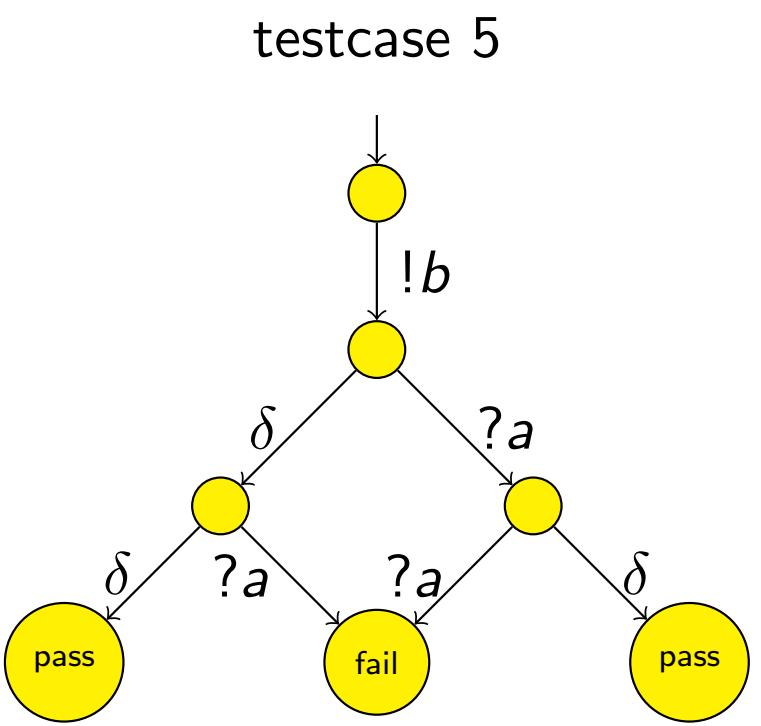
Example



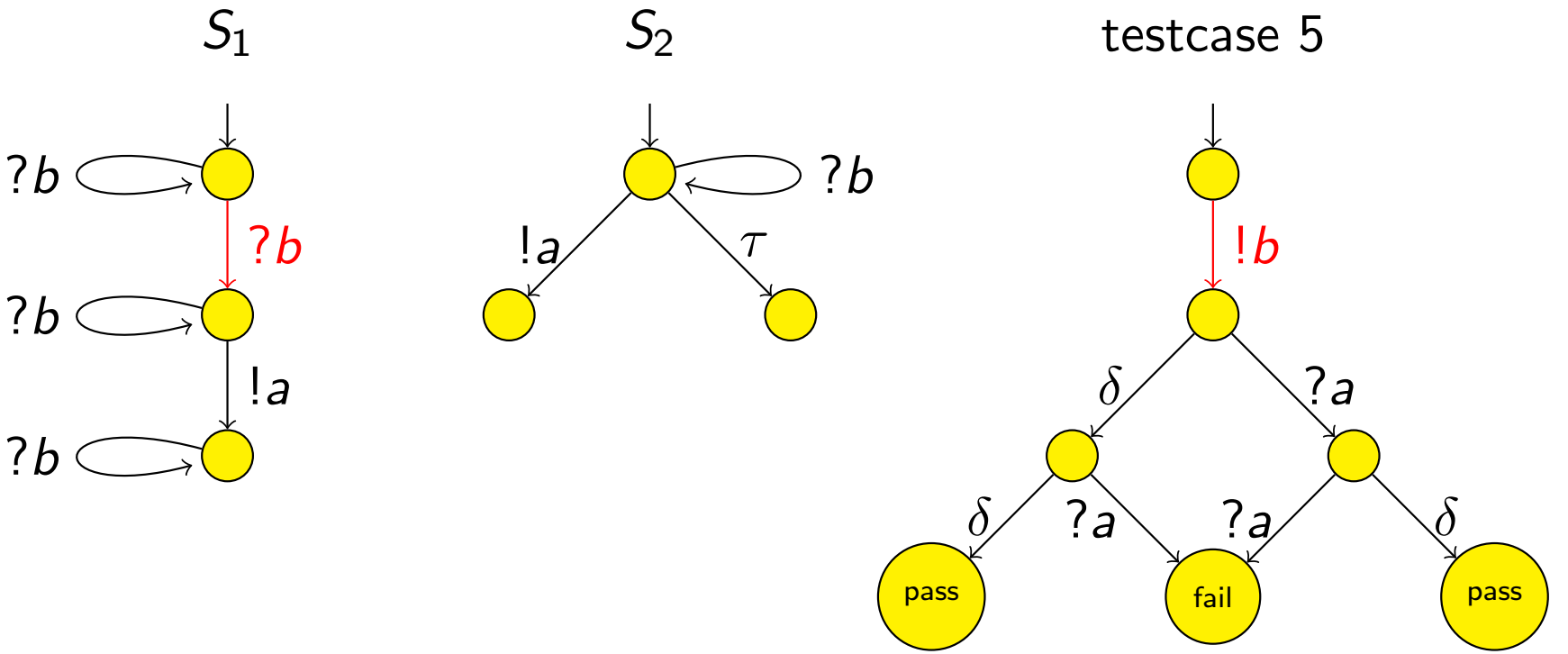
Example



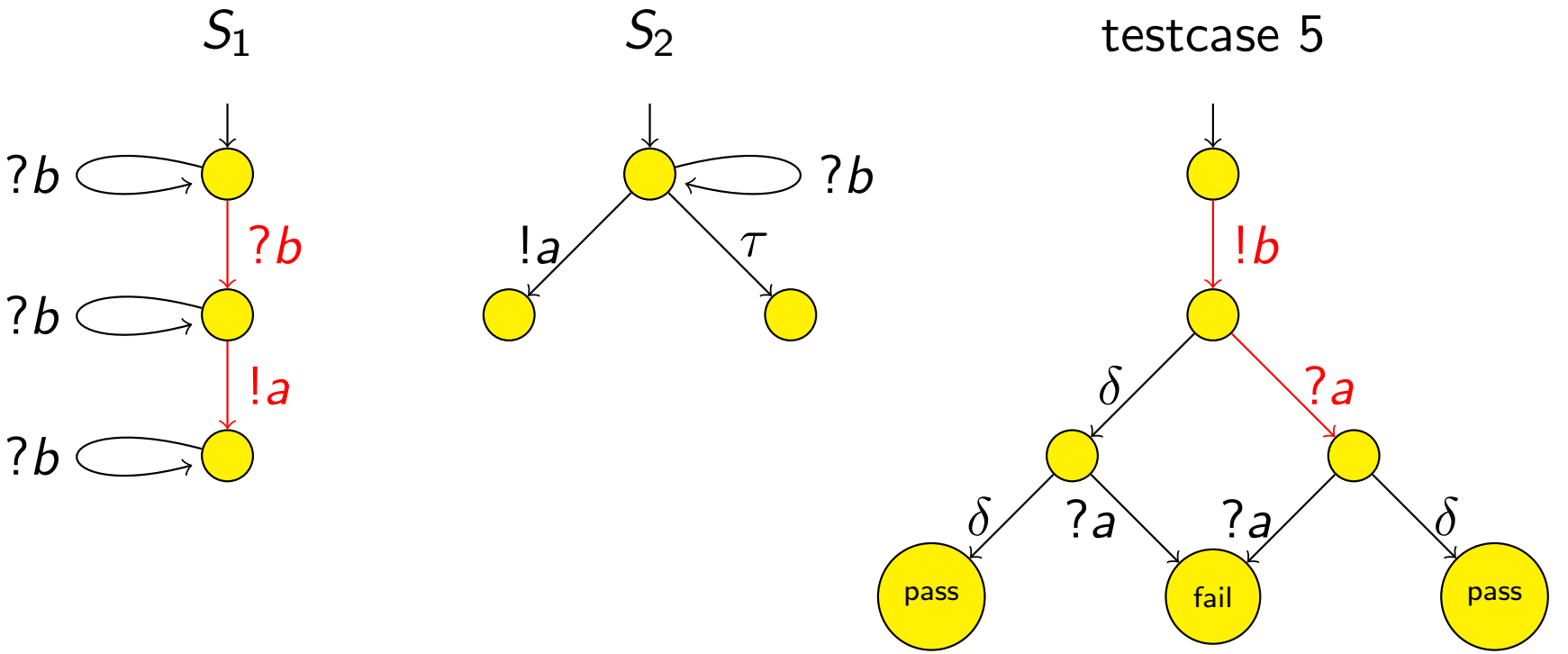




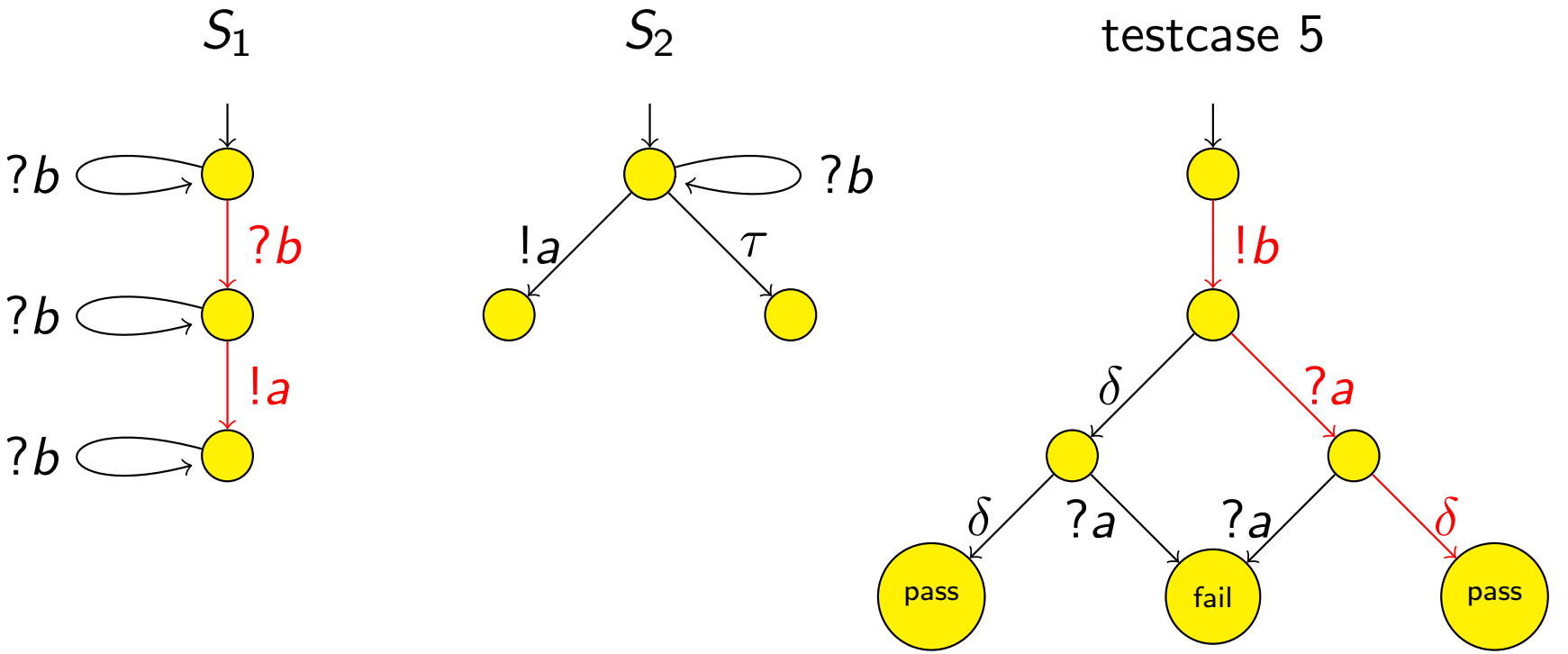
Example



Example



Example



Properties of Test Generation Algorithm

Given a specification $S_2 = \langle S, s_0, \mathcal{A}_I \cup \mathcal{A}_U, \rightarrow \rangle$.
Given an implementation S_1.

- $Gen(S')$ is a tree-like IOTS,
- (Soundness) for each test cases t obtained by running algorithm $Gen(s_0$ *after* $\epsilon)$, we have:

  S_1 **ioco** S_2 implies S_1 *passes* t
- (Completeness) algorithm $Gen(s_0$ *after* $\epsilon)$ can generate a test-case, such that:

  S_1 **not ioco** S_2 implies S_1 *fails* t

Conformance testing in practice

ioco

-testing on real, black-box implementations:

BFdV+99 A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw and L. Heerink, *Formal Test Automation: A Simple Experiment*. In *Proceedings of TestCom'99*.

dVT00 R.G. de Vries, J. Tretmans, *On-the-fly Conformance Testing using SPIN*. In *STTT 2(4)*.

dVBF02 R.G. de Vries, A. Belinfante, J. Feenstra, *Automated Testing in Practice: The Highway Tolling System*. In *Proceedings of TestCom 2002*.

How to read the handout

- **stop** there: deadlock,
  $\cdot$ there: ; here,
  $\Box$ there : $+$ here,
  $\sum$ there: a generalization of $+$ (as in mCRL2)
- $\delta(s)$ there: $s \xrightarrow{\delta}$ here
- $\leq_{iot}$ and $\leq_{ior}$ defined in the handouts but not here
- *suspension automata* in the handout (deterministic version of LTS with δ-loops) not presented here
- θ there (input for δ): δ here