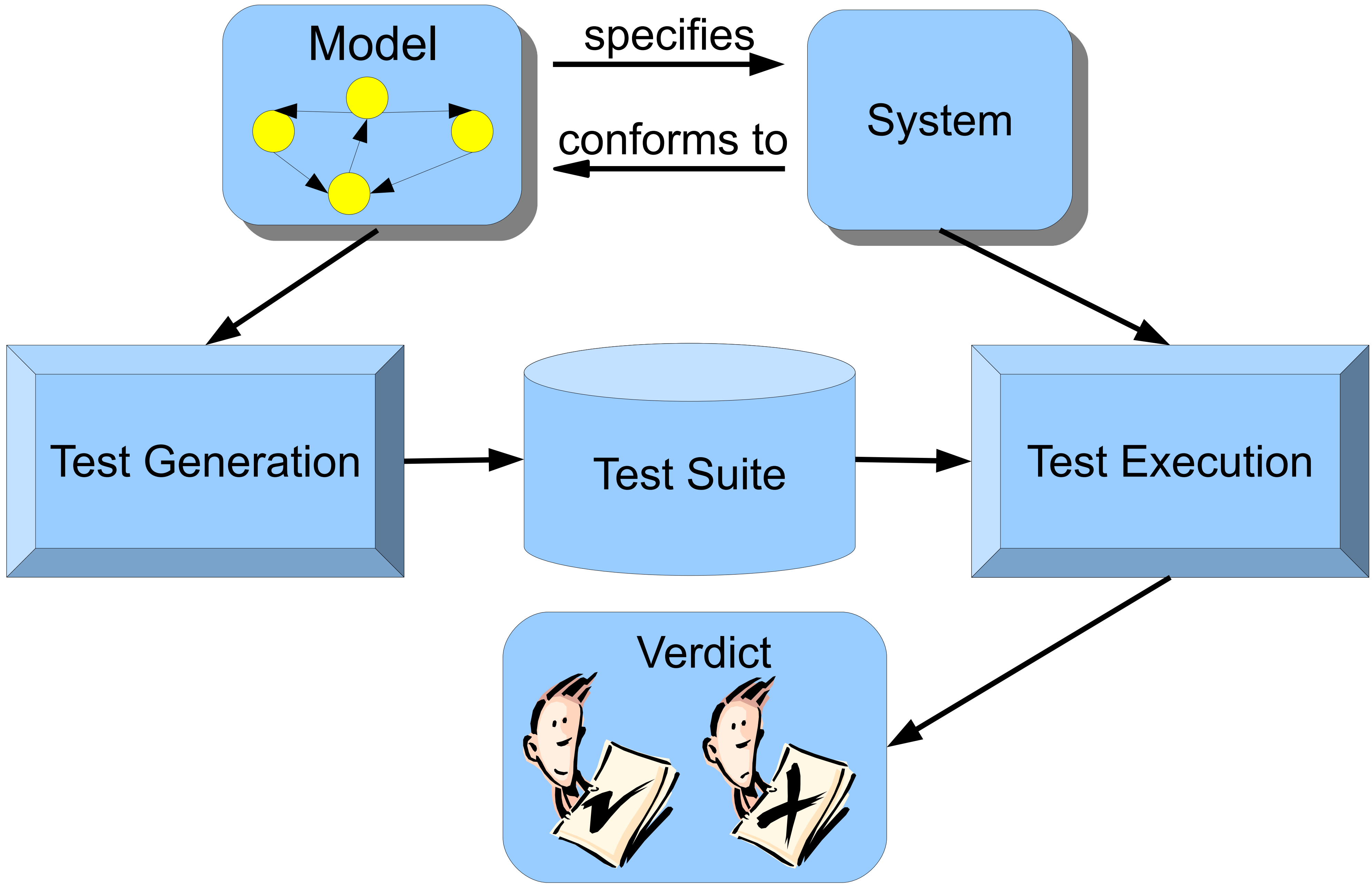
Model-based Testing

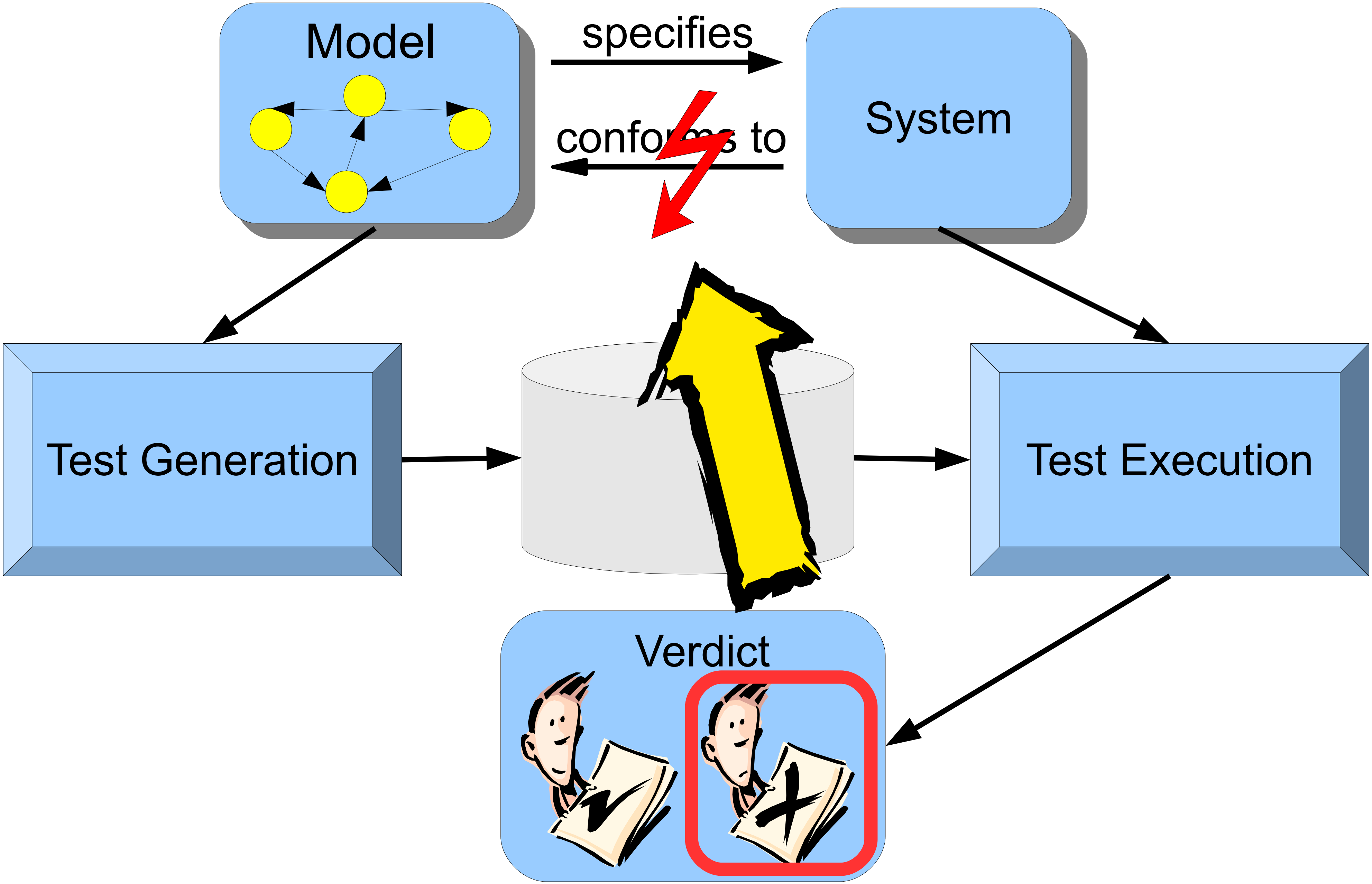
Gordon Fraser · Saarland University

Slides by Lars Frantzen

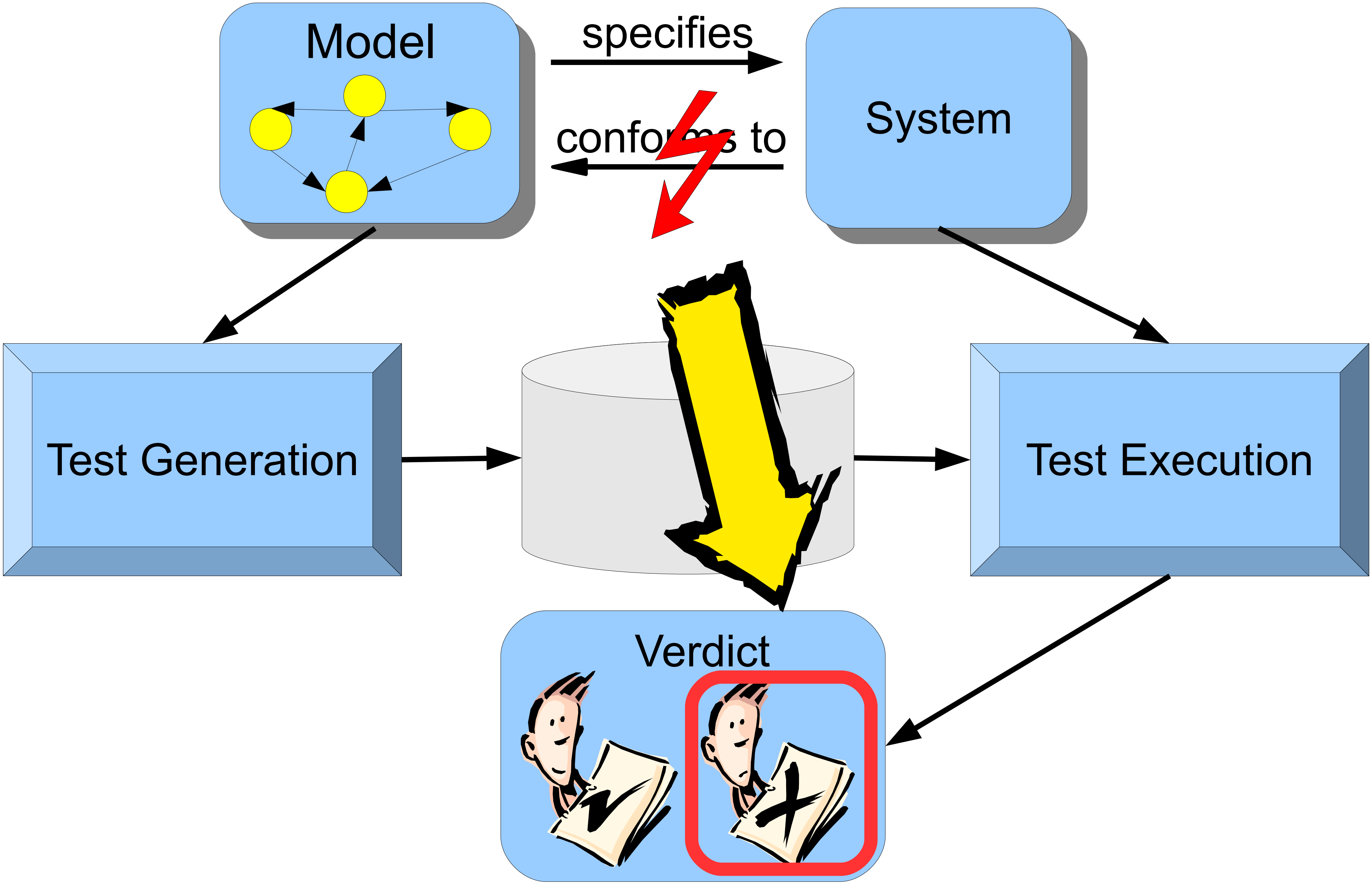
Conformance Testing



Soundness of Conformance Testing



Completeness of Conformance Testing



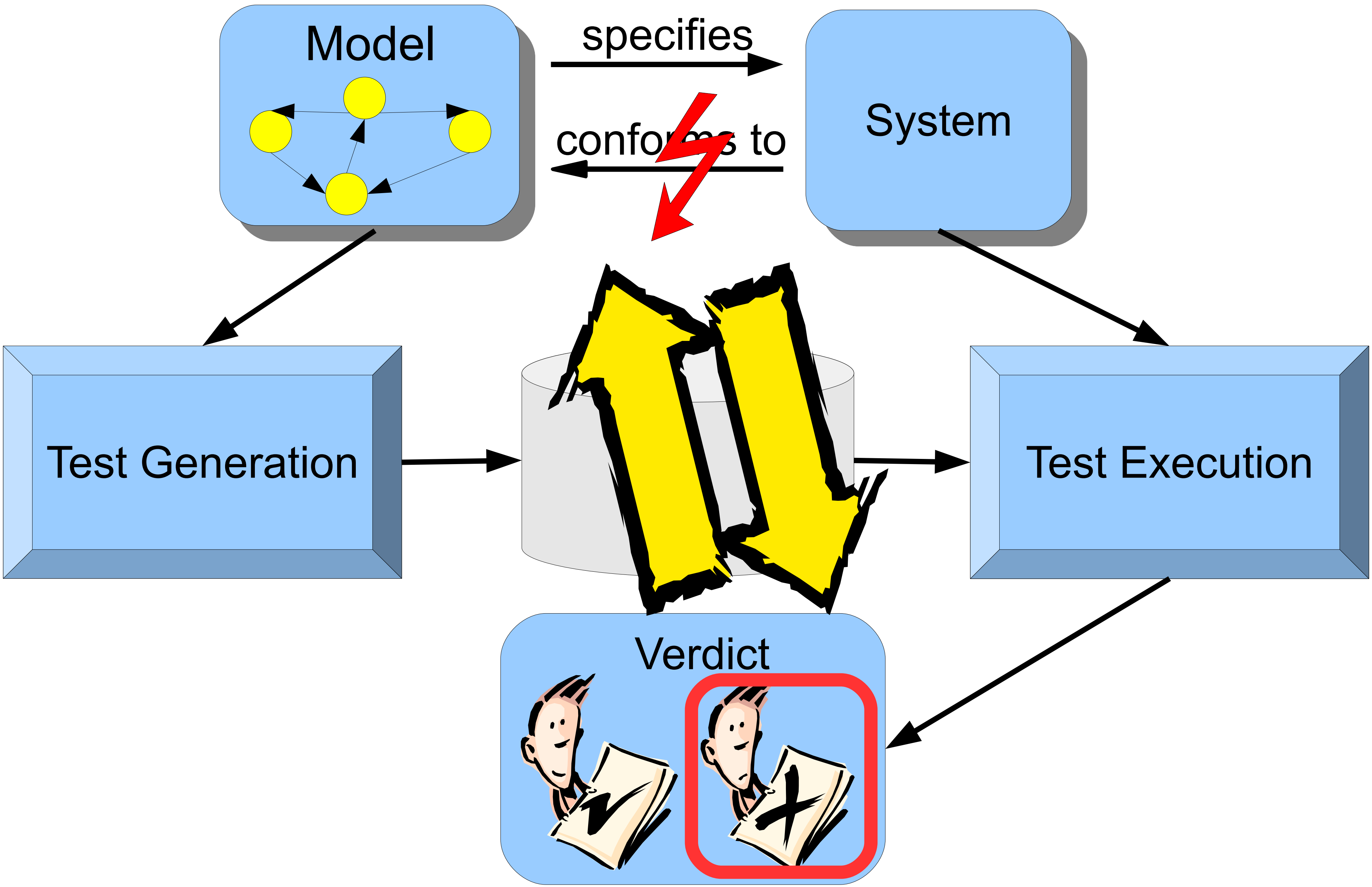
Remarks on Soundness and Completeness

- A test system, which always says  is sound.

- A test system, which always says  is complete.

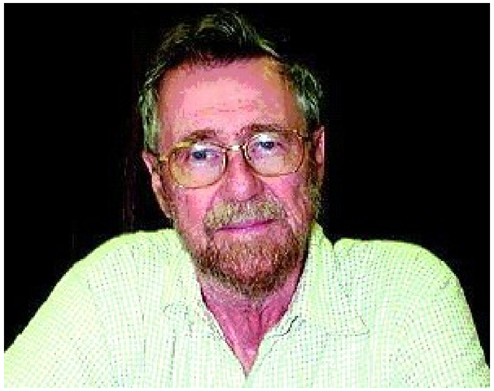
- We want test systems which are **sound and complete**!

Soundness and Completeness of Conformance Testing



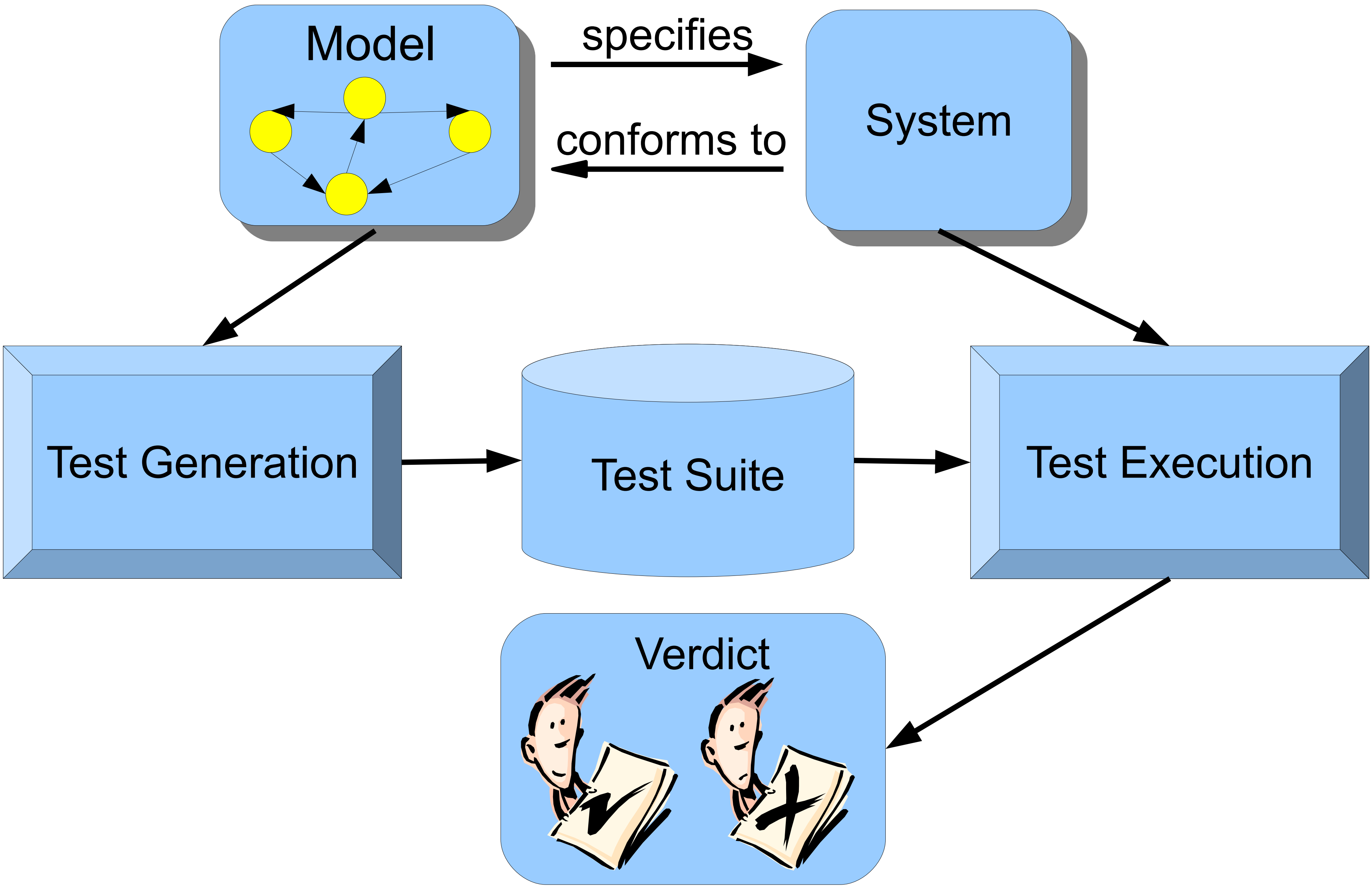
More Remarks on Soundness and Completeness

Testing can never be sound and complete!

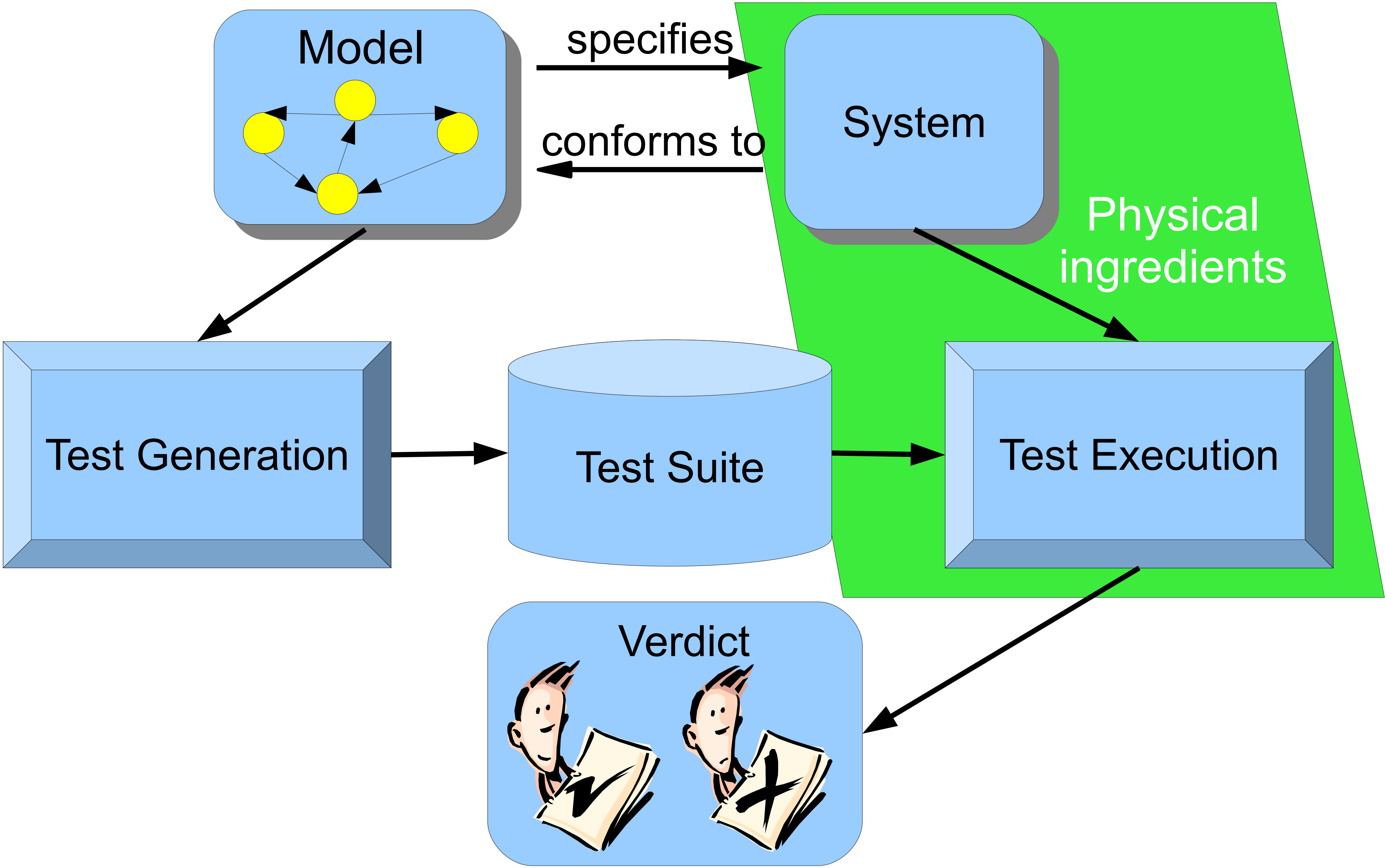


Edsger W. Dijkstra

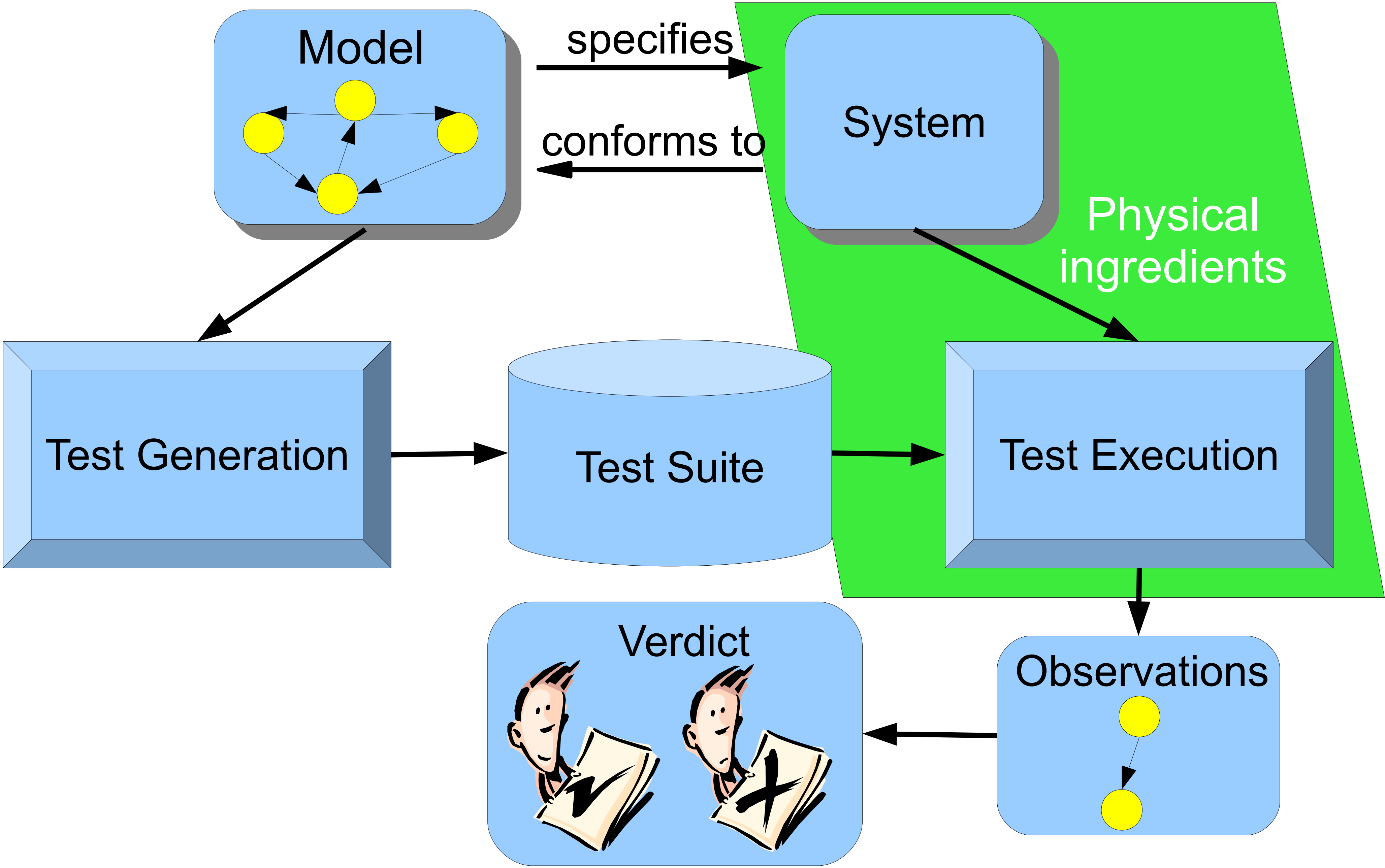
Proving Soundness and Completeness



Proving Soundness and Completeness

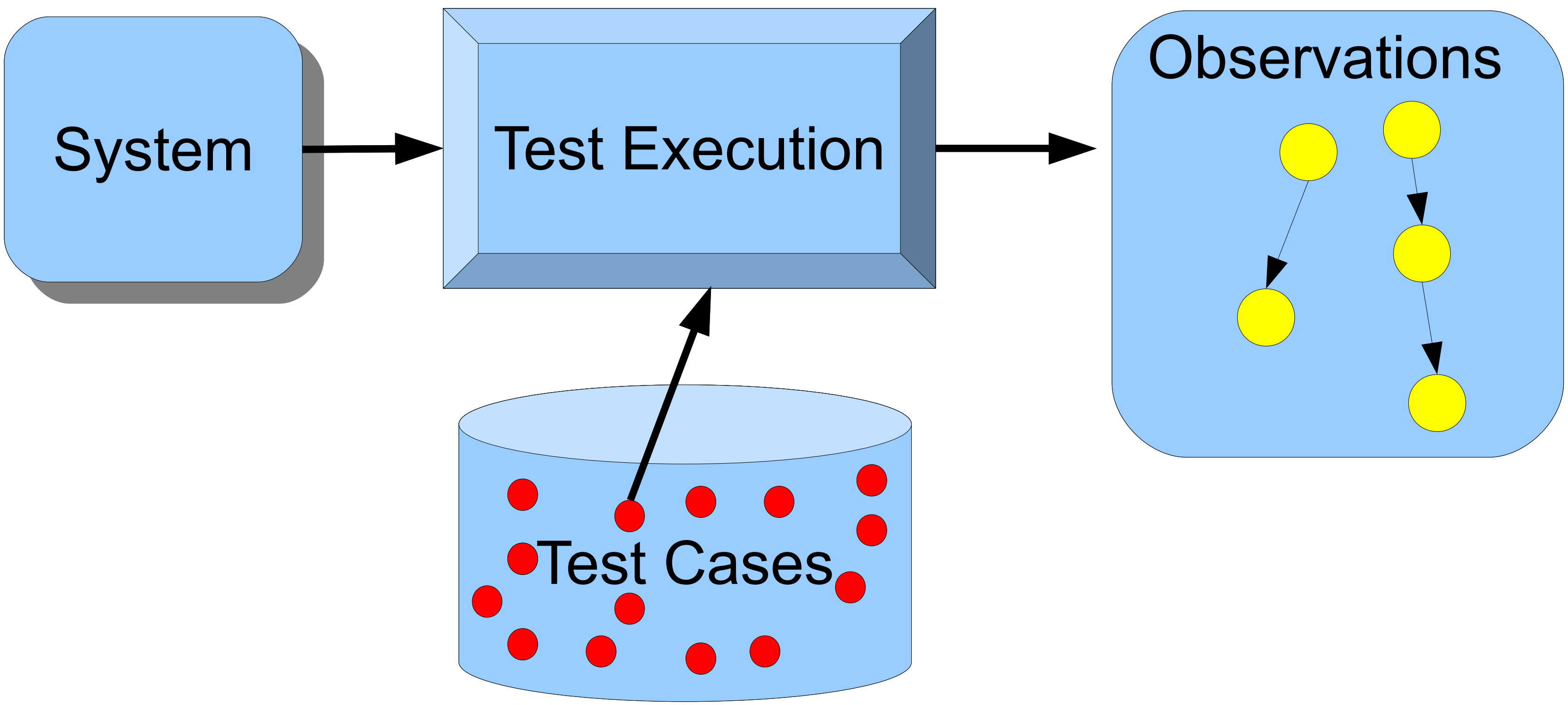


Proving Soundness and Completeness



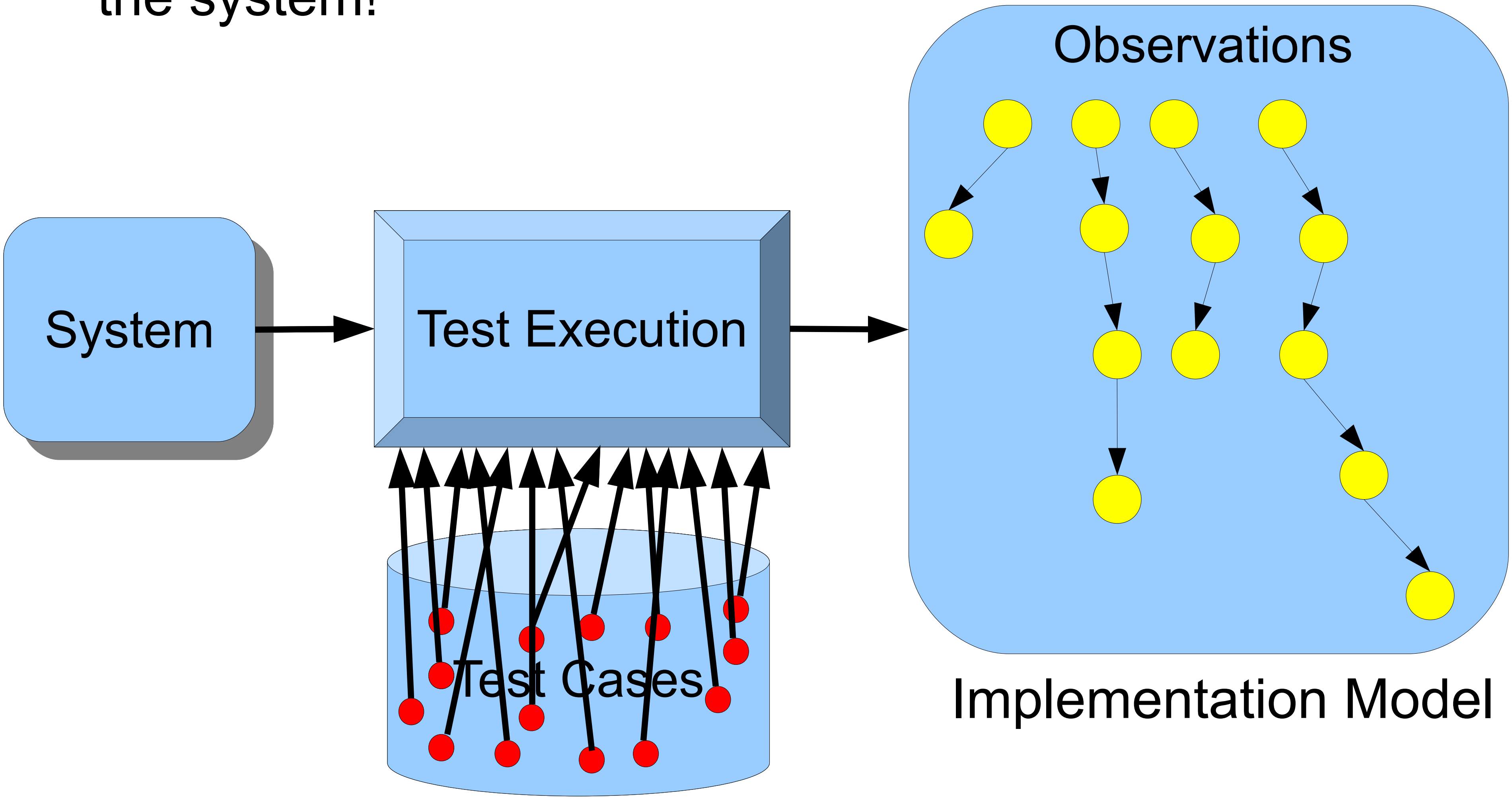
Observations

- Executing a test case on the system yields a set of observations.
- Every observation represents a part of the **implementation model** of the system, i.e. the model describing how the real system behaves.



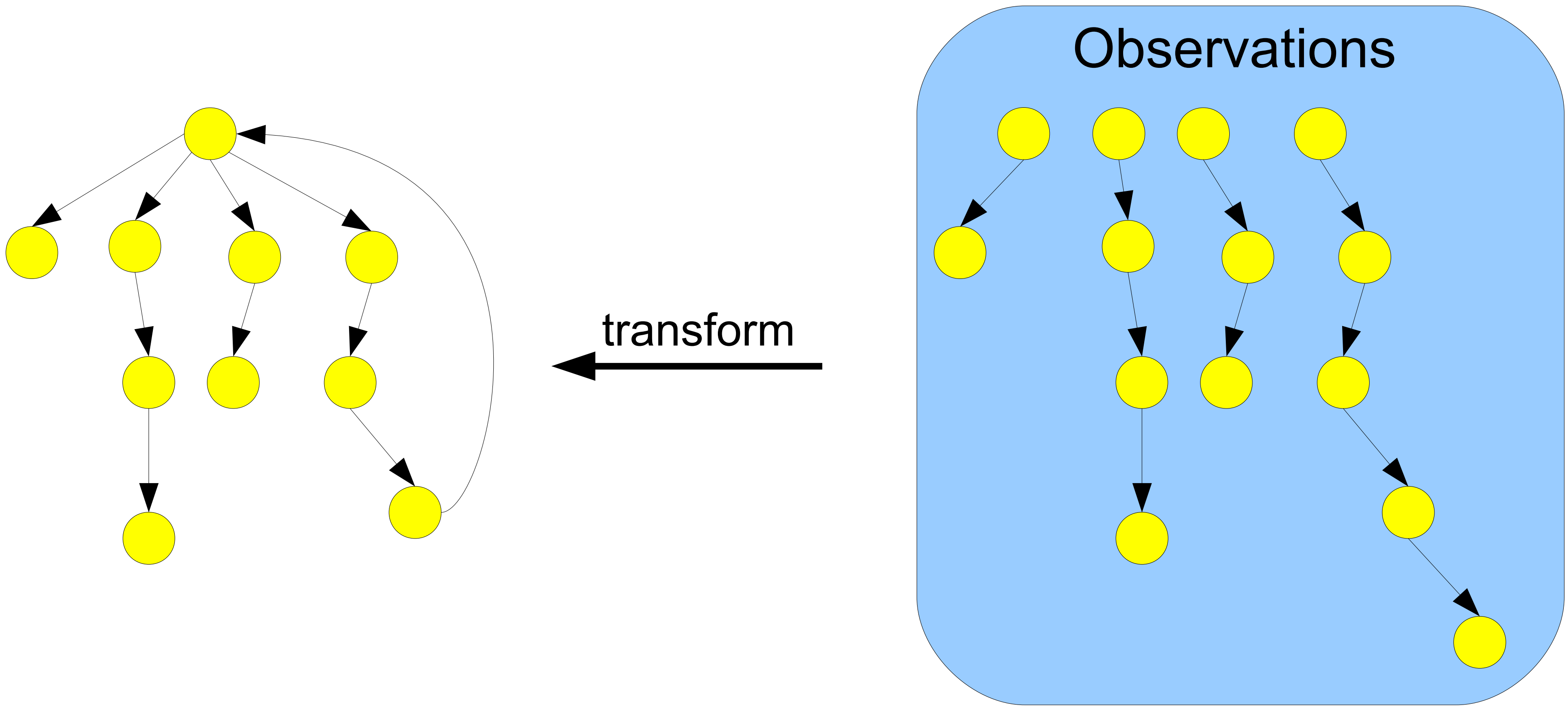
Implementation Models

- The set of all observations made with <u>all possible test cases</u> represents the complete **implementation model** of the system!



Implementation Models

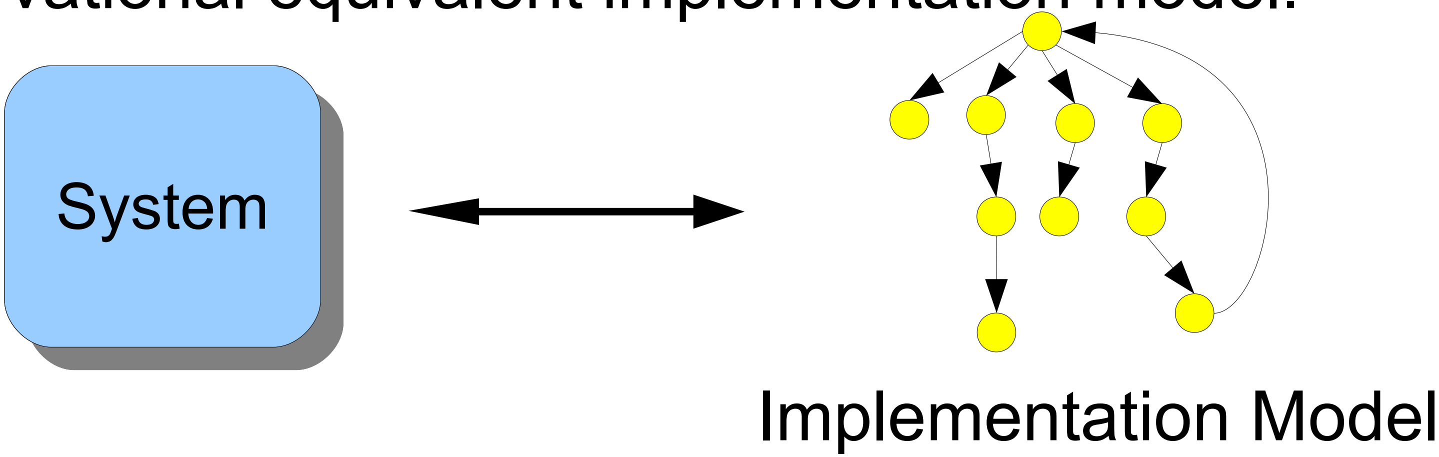
- Depending on the chosen class of implementation models, the observations might have to be transformed, first.



Implementation Model

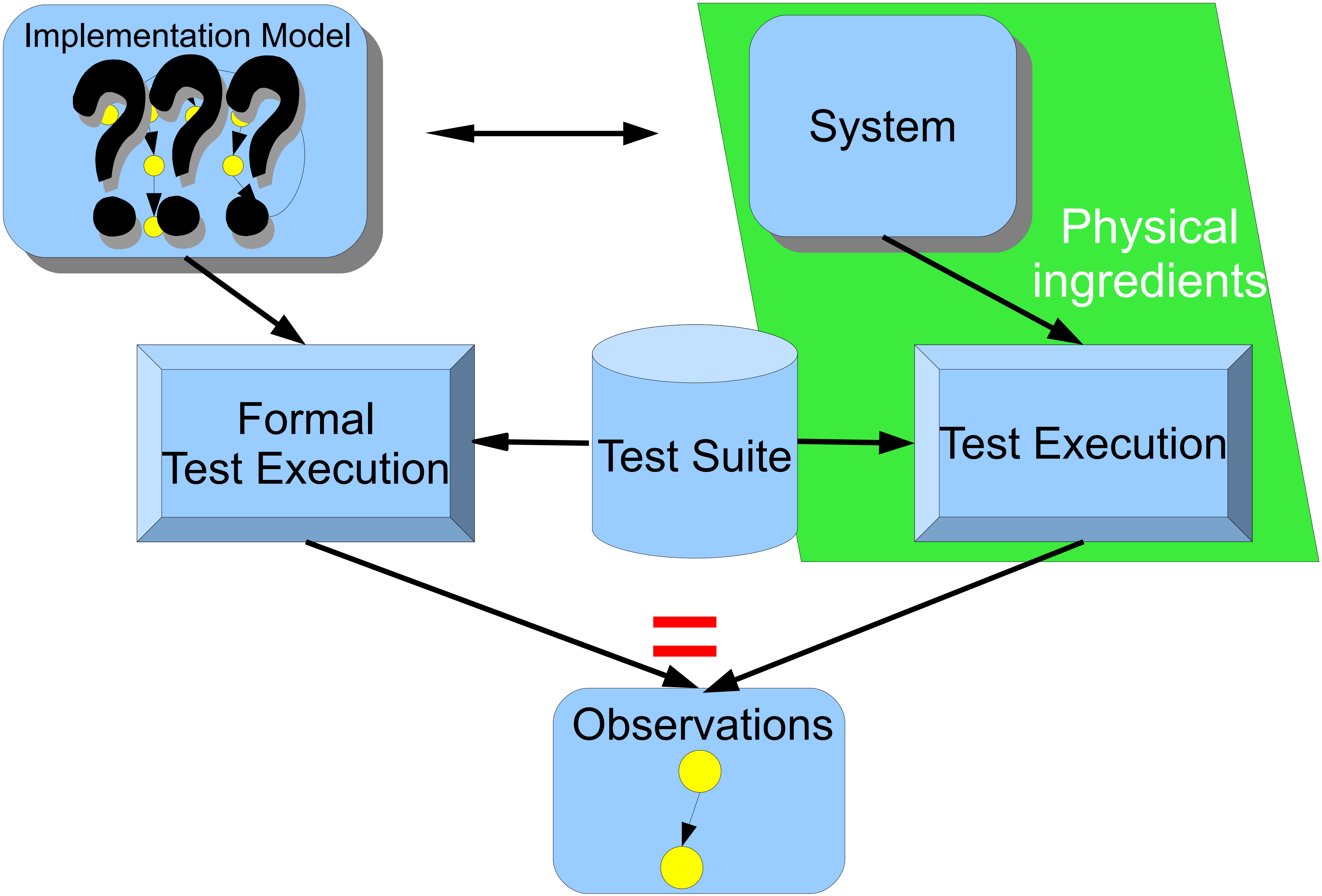
Implementation Models

- Assuming from now on the validity of the test hypothesis, we know that for every system there is a corresponding observational equivalent implementation model.

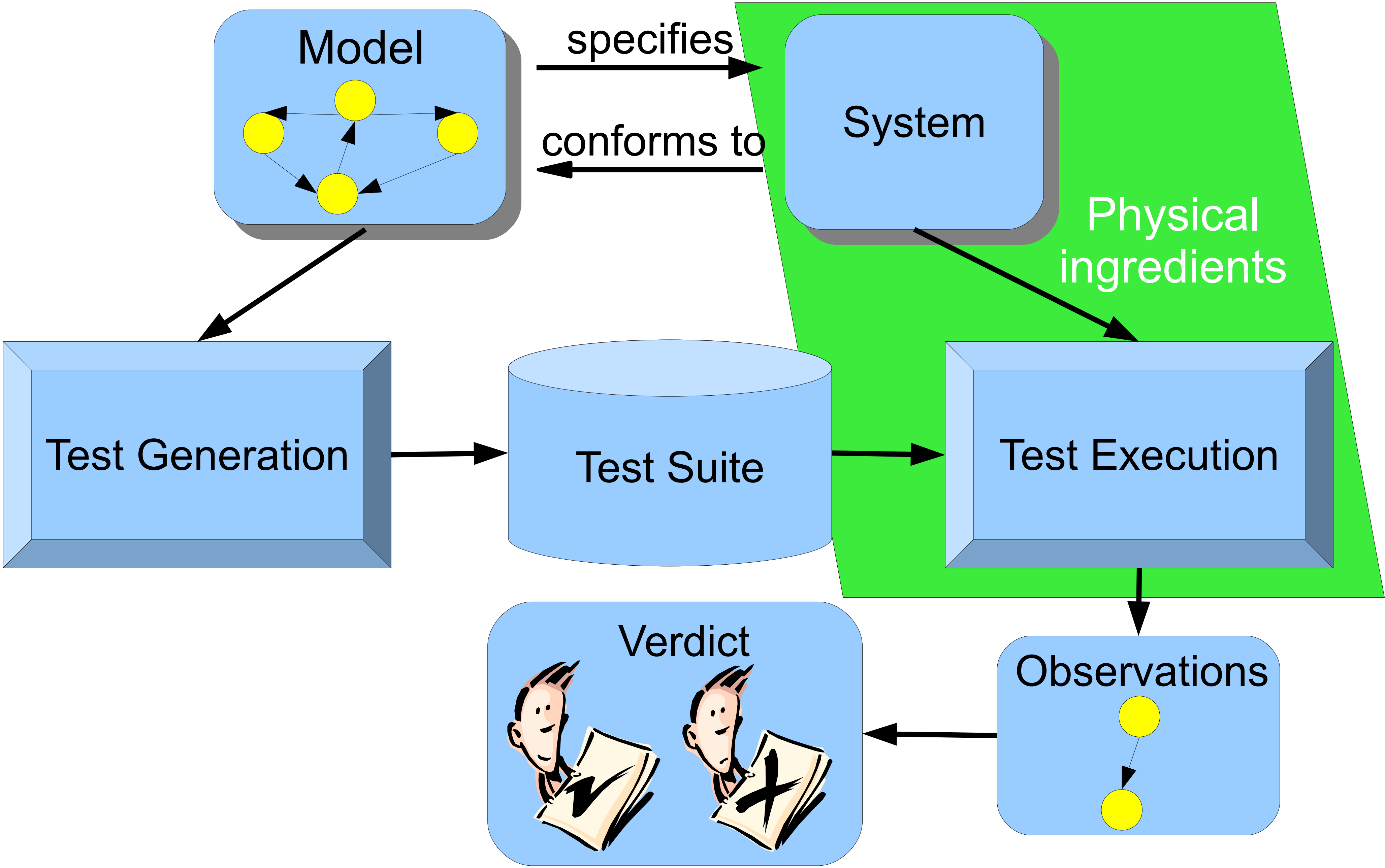


- This implementation model is **unknown** since in practice we cannot execute all possible test cases at the system.
- But since we know it exists, we can now define formally what conformance means!

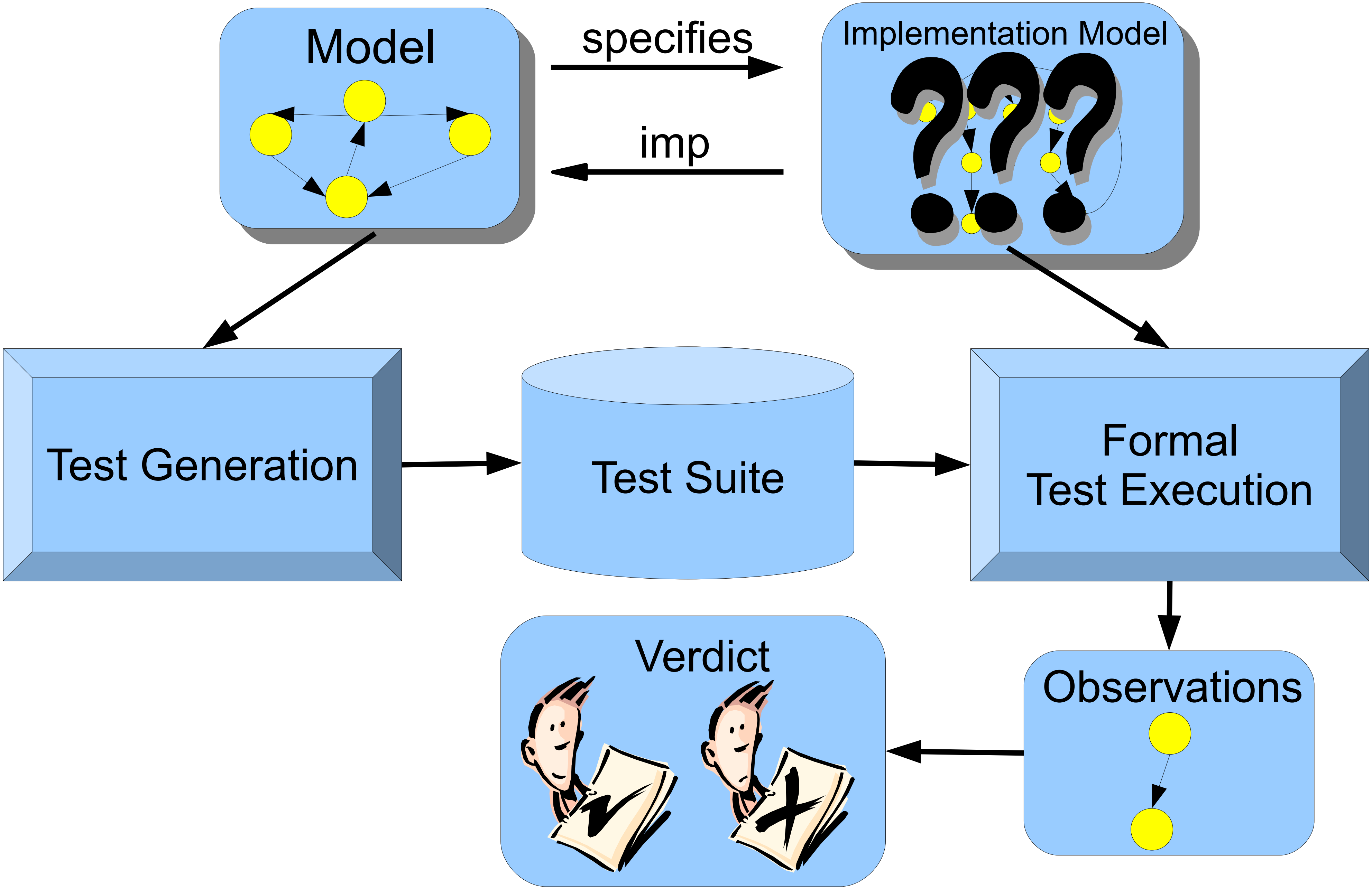
Observational Equivalence



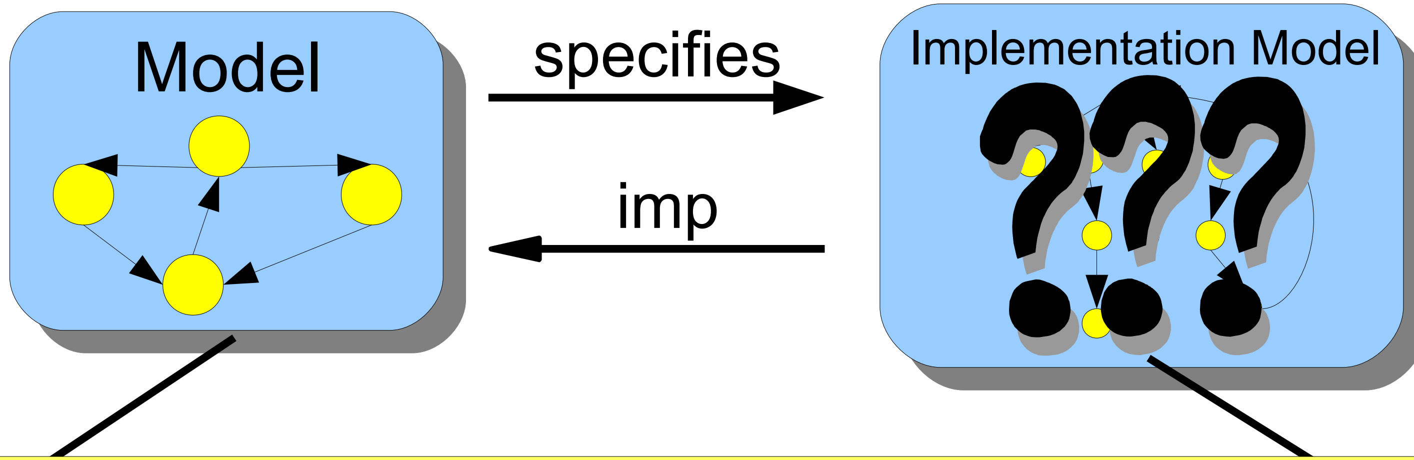
Proving Soundness and Completeness



Proving Soundness and Completeness



Proving Soundness and Completeness

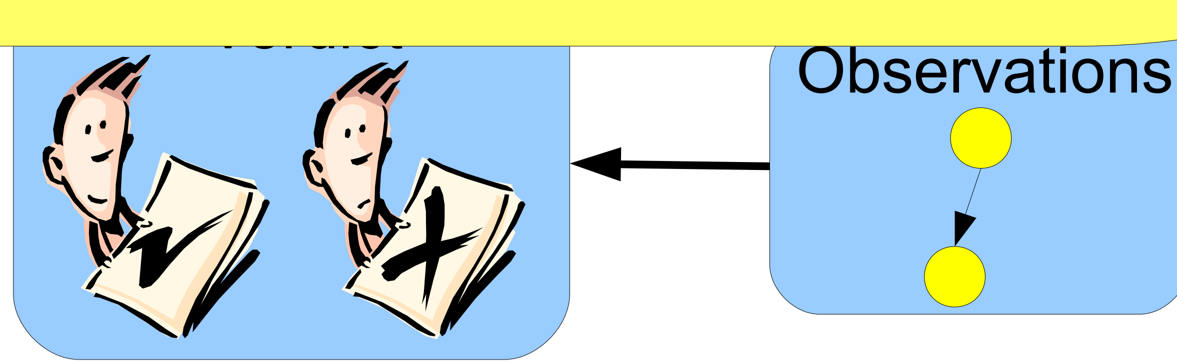


Now we can define:

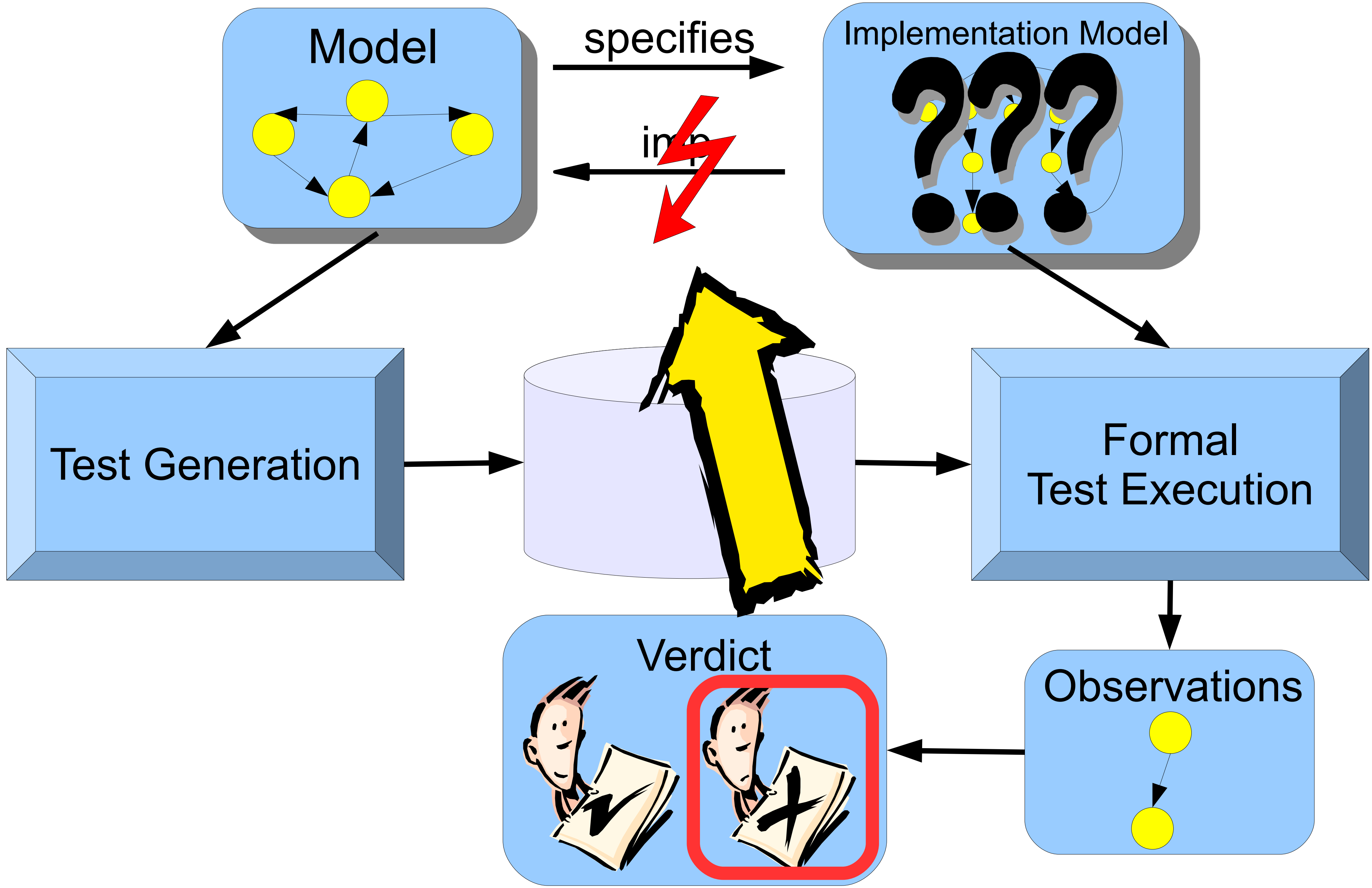
System **conforms-to** the specification model

⇤

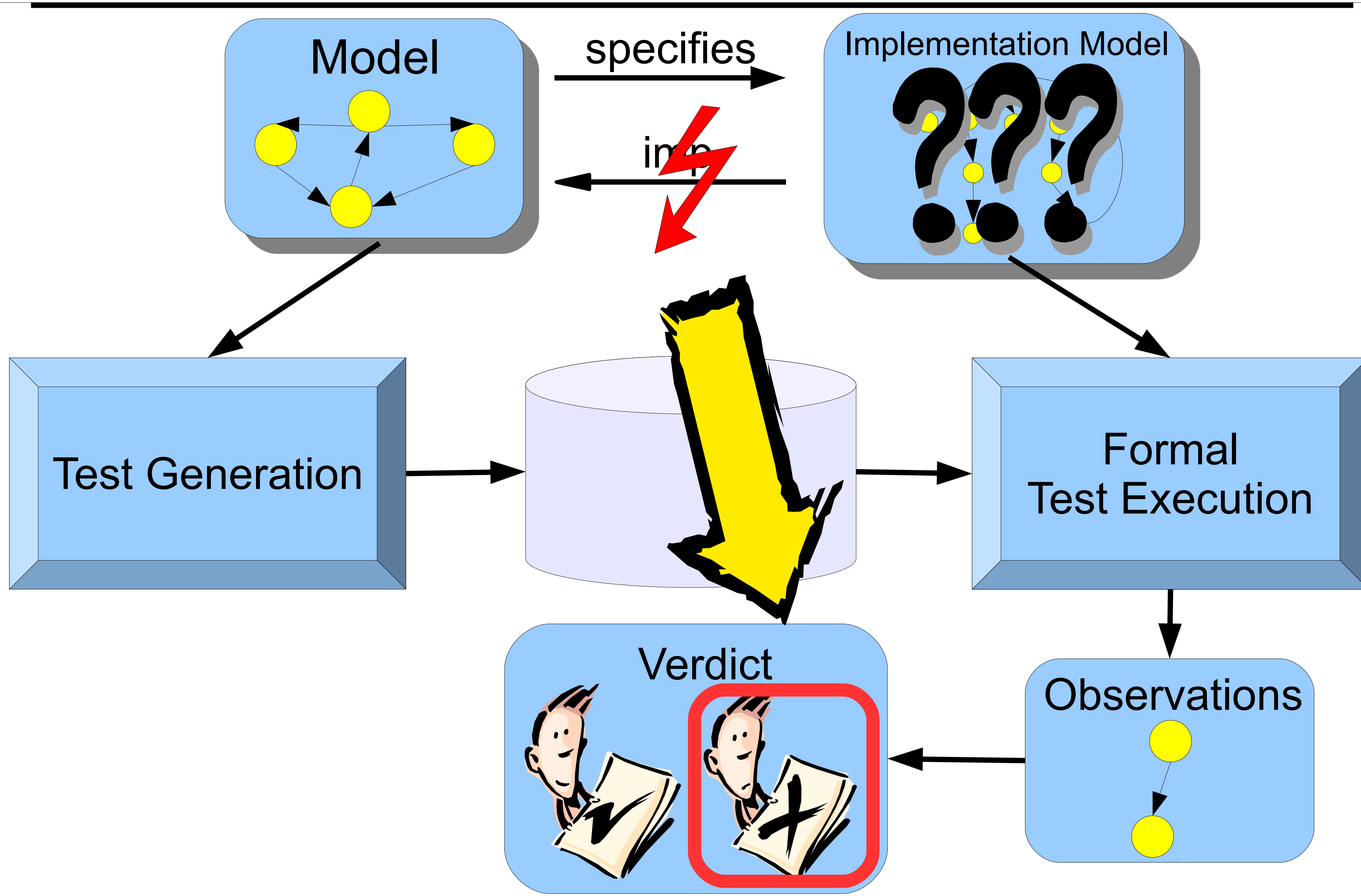
the implementation model is **imp-correct** to the specification model



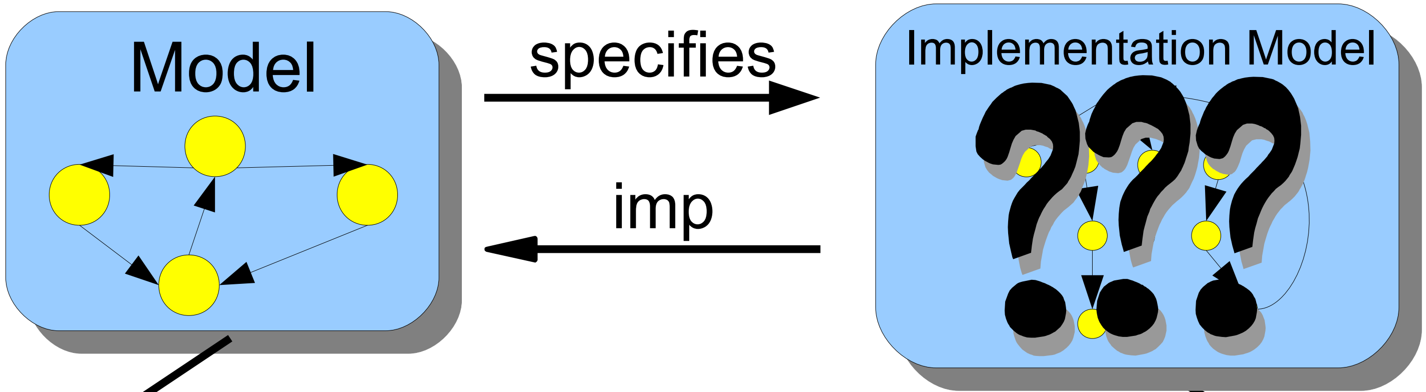
Proving Soundness



Proving Completeness



Proving Soundness and Completeness



The **proof-obligation** to show the soundness and completeness of a test generation algorithm <u>w.r.t. an implementation relation imp</u> is:

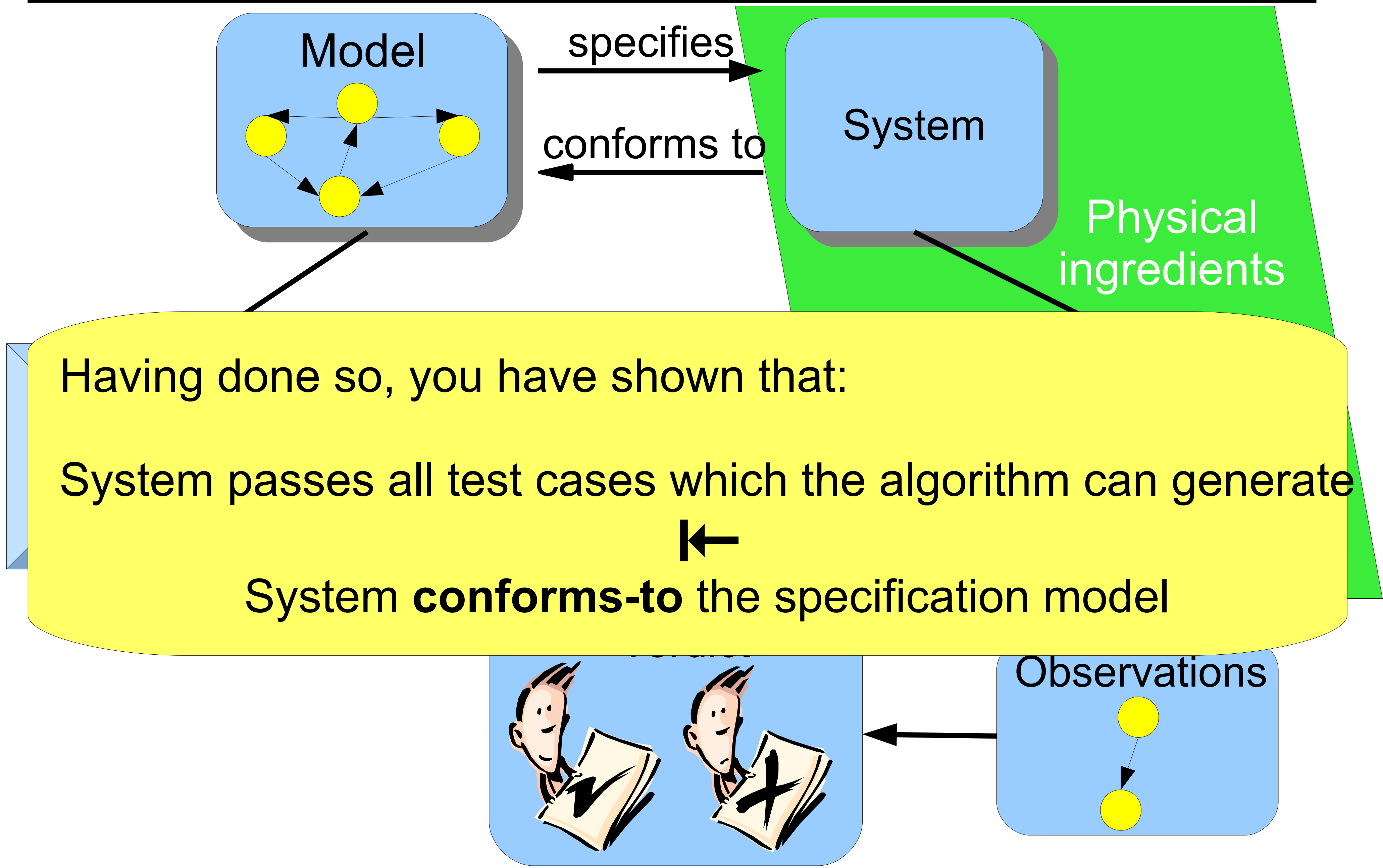
Show **for all implementation models**:

implementation model M is **imp-correct** to the specification model

⇤

M passes all test cases which the algorithm can generate

Proving Soundness and Completeness



Having done so, you have shown that:

System passes all test cases which the algorithm can generate

$\Leftarrow$

System **conforms-to** the specification model

Summary

- We want test generation algorithms to be **sound and complete** for the **conforms-to** relation.
- Every system has an underlying **implementation model** consisting of all possible observations one can make with all possible test cases.
- To restrict the class of systems, **assumptions** are made on the test execution.
- Based on these assumptions, one has to **prove** that an implementation model exists which is **observational equivalent** to the system.

Summary

- Now the implementation model can be substituted for the real system (aka the **test hypothesis**).
- Between the implementation model and the specification model **implementation relations** can be defined.
- **Conformity** of a system to a model is then defined by the **imp-correctness** of its underlying implementation model.
- The main **proof obligation** is to show the soundness and completeness of the test generation algorithm w.r.t. the chosen implementation relation.

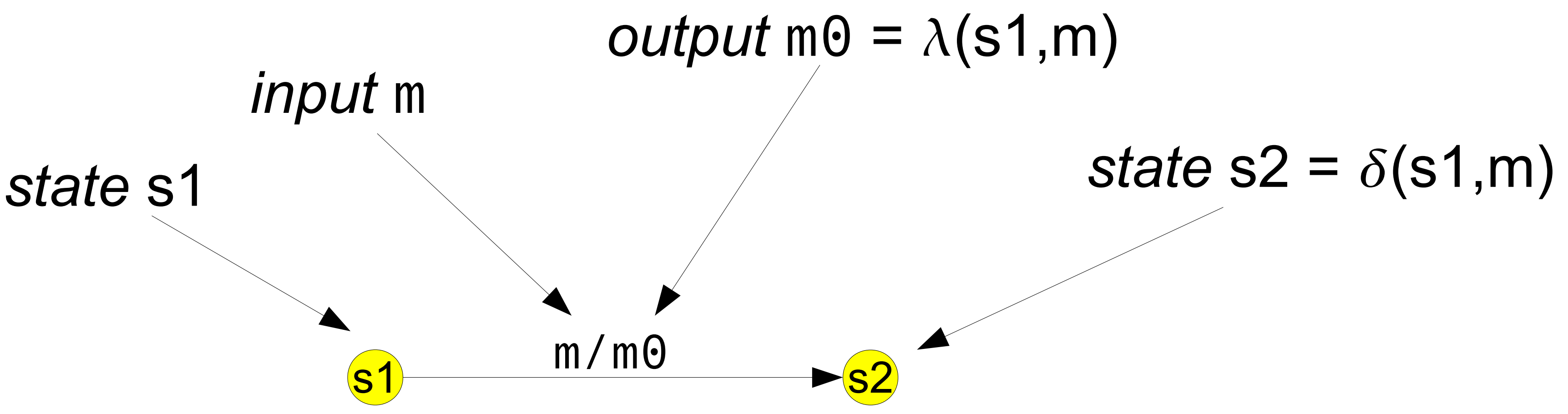
Finite State Machines

- Original domains:
  - sequential circuits
  - communication protocols

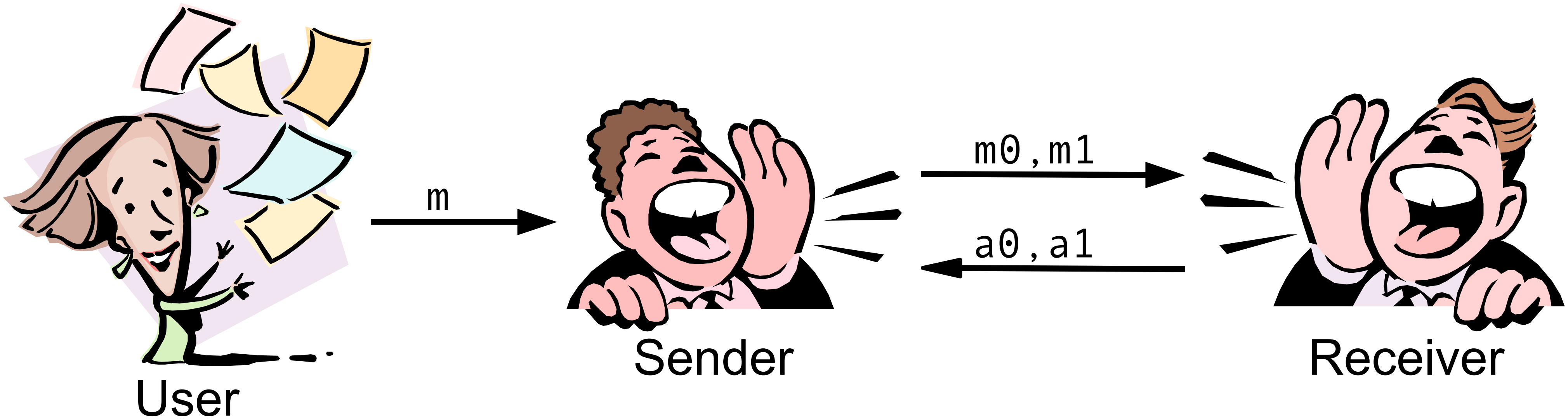
- Two types of **Finite State Machines (FSM)** matter for testing:
  - Mealy Machines
  - Moore Machines
- Commonly, FSM is identified with **Mealy Machine**.

Mealy Machines

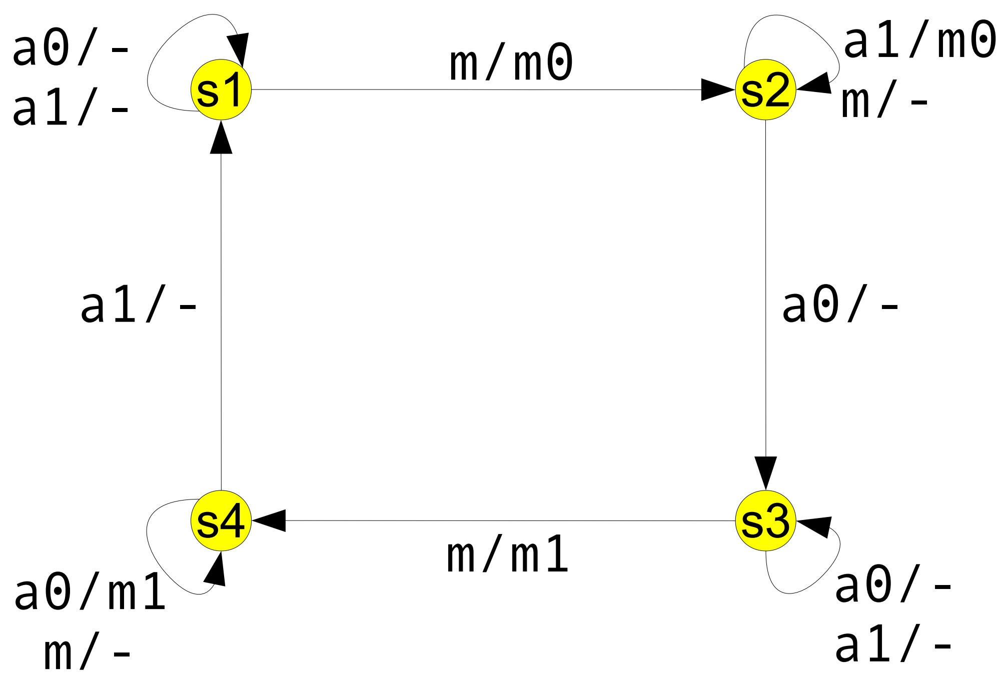
- **Definition 1 (Mealy Machine).** *A* Mealy Machine M *is a quintuple* $M = (I, O, S, \delta, \lambda)$ *where*
  - I *is a finite and nonempty set of* **inputs**
  - O *is a finite and nonempty set of* **outputs**
  - S *is a finite and nonempty set of* **states**
  - $\delta : S \times I \to S$ *is the* **state transition function**
  - $\lambda : S \times I \to O$ *is the* **output function**



Alternating Bit Protocol

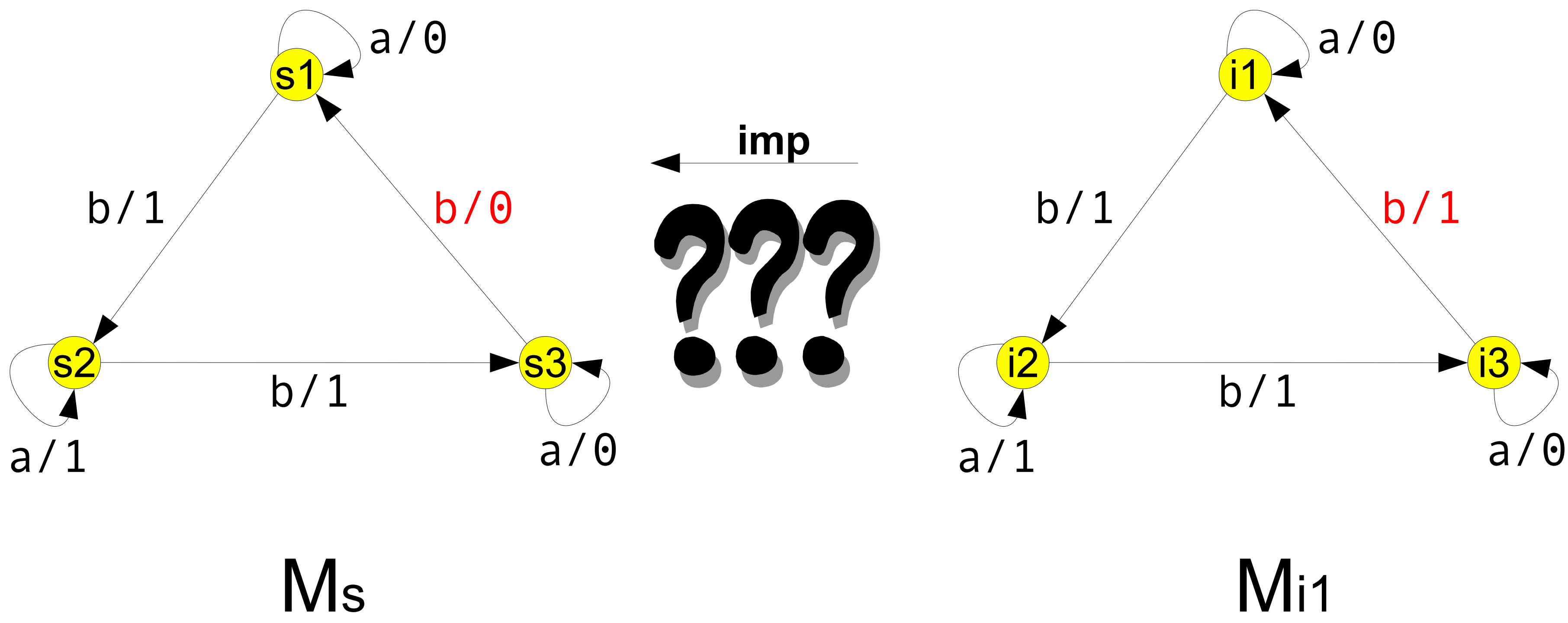


The Sender as a Mealy Machine:



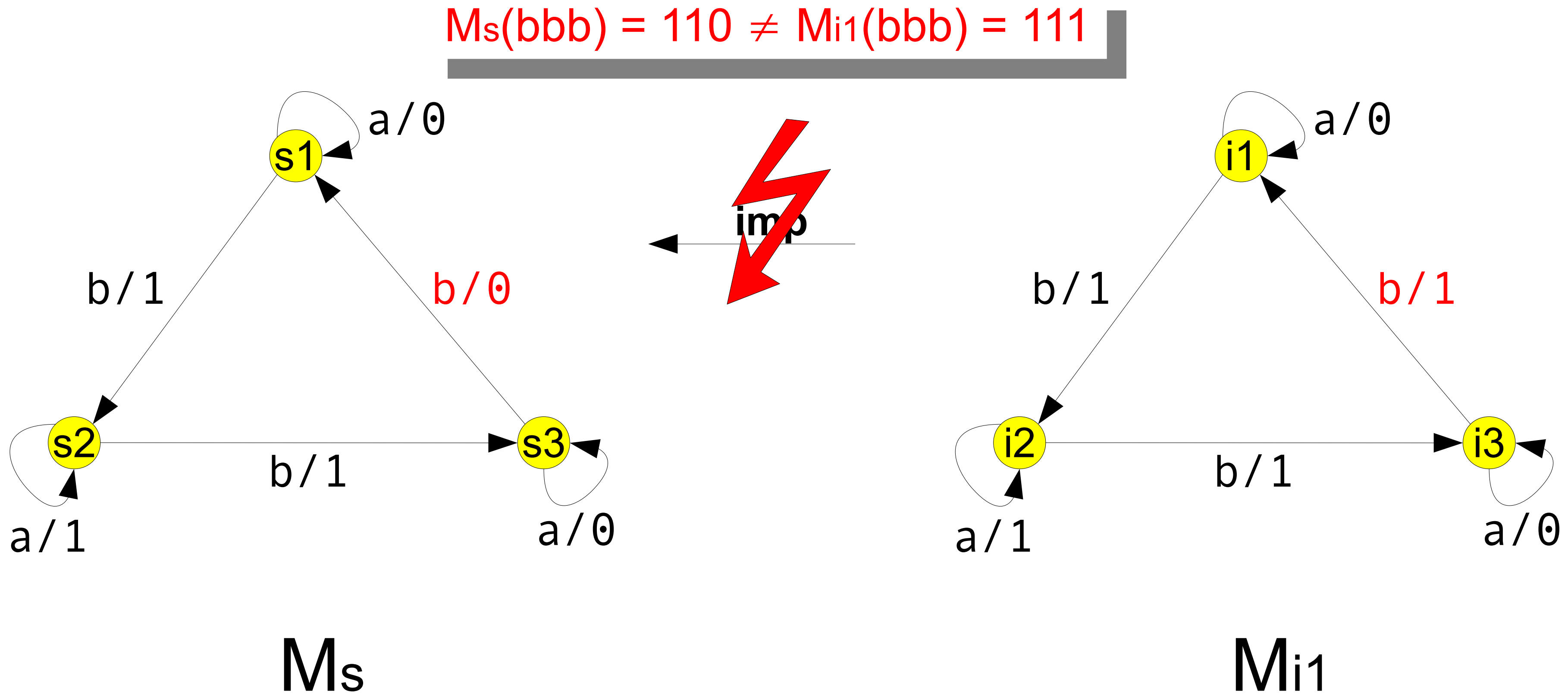
Conformance

- Specification models and implementation models are Mealy Machines.
- What does conformance mean here?



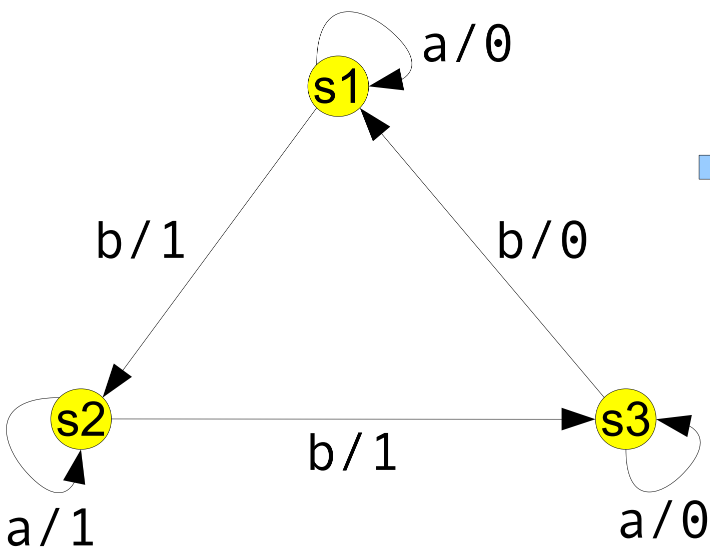
Conformance

- We have M_i **imp** M_s ⇔ M_i is **equivalent** to M_s
- Two FSM are **equivalent** iff for every input sequence they produce the same output sequence.



Test Cases

- A **test case** is an input sequence together with its output sequence, derived from the specification model.



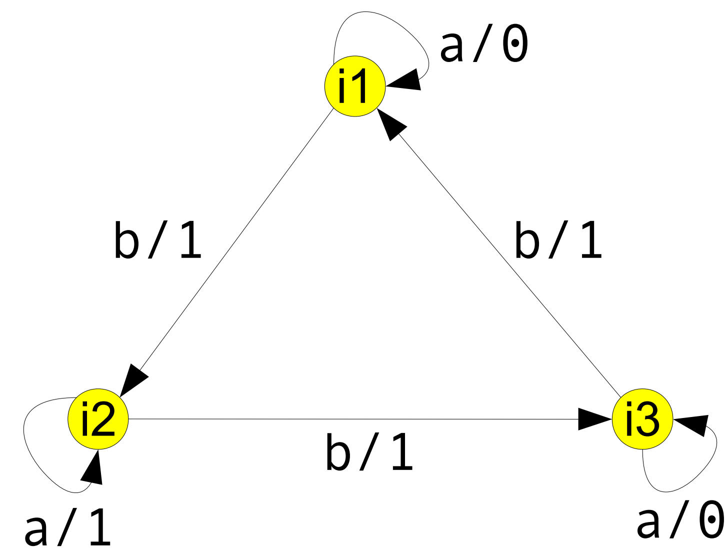
M_s

Test case: input: `bbb`
output: `110`

Formal Test Execution

- A **test case** is an input sequence together with its output sequence, derived from the specification model.
- **Formally executing** a test case means giving the input sequence to the implementation model.

Test case: input: bbb
output: 110



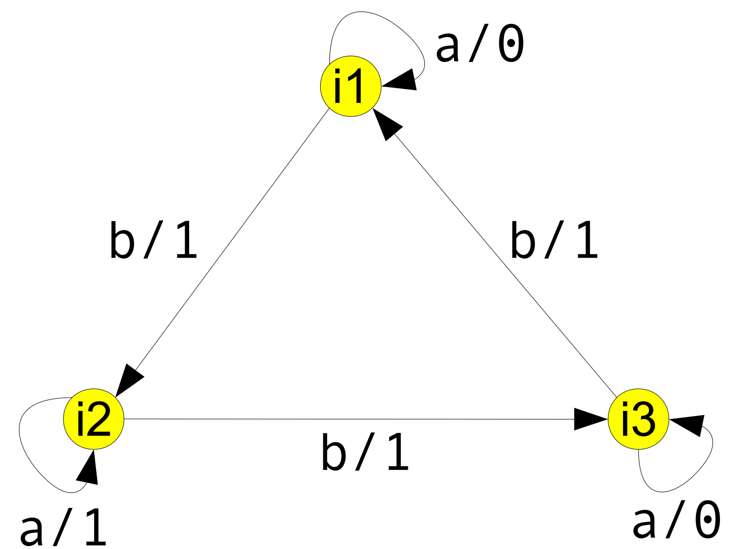
M_{i1}

Observations

- A **test case** is an input sequence together with its output sequence, derived from the specification model.
- **Formally executing** a test case means giving the input sequence to the implementation model, and **observing** the corresponding output sequence.

Test case: input: bbb
output: 110

Observation: input: bbb
output: 111



M_{i1}

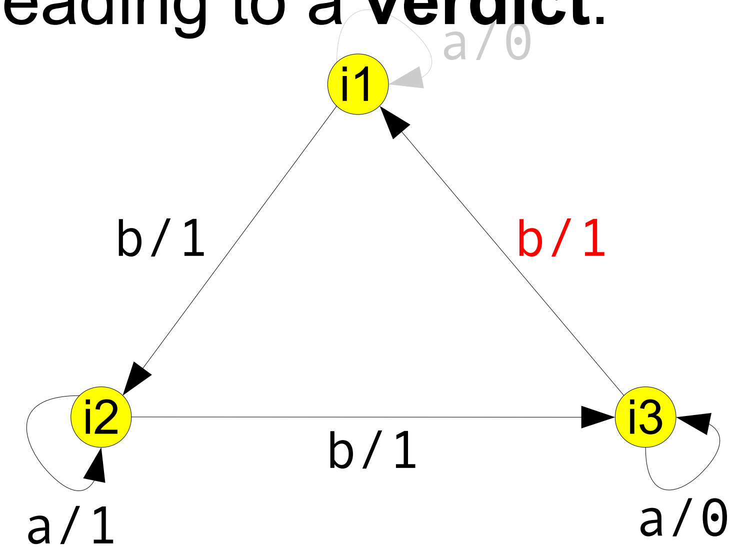
Verdicts

- A **test case** is an input sequence together with its output sequence, derived from the specification model.
- **Formally executing** a test case means giving the input sequence to the implementation model, and **observing** the corresponding output sequence - leading to a **verdict**.

Test case: input: bbb
output: 110

+

Observation: input: bbb
output: 111

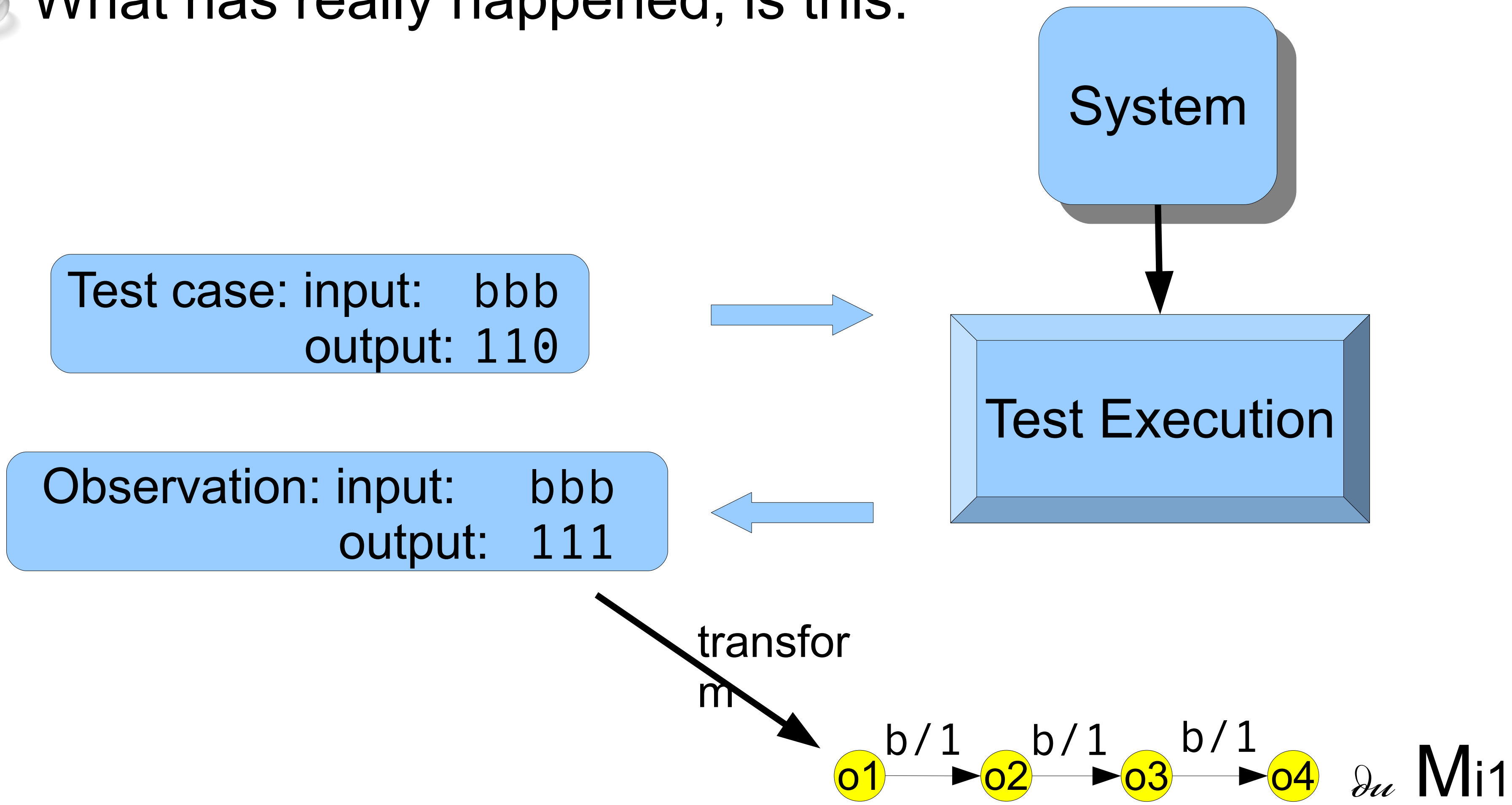


M_{i1}

= 

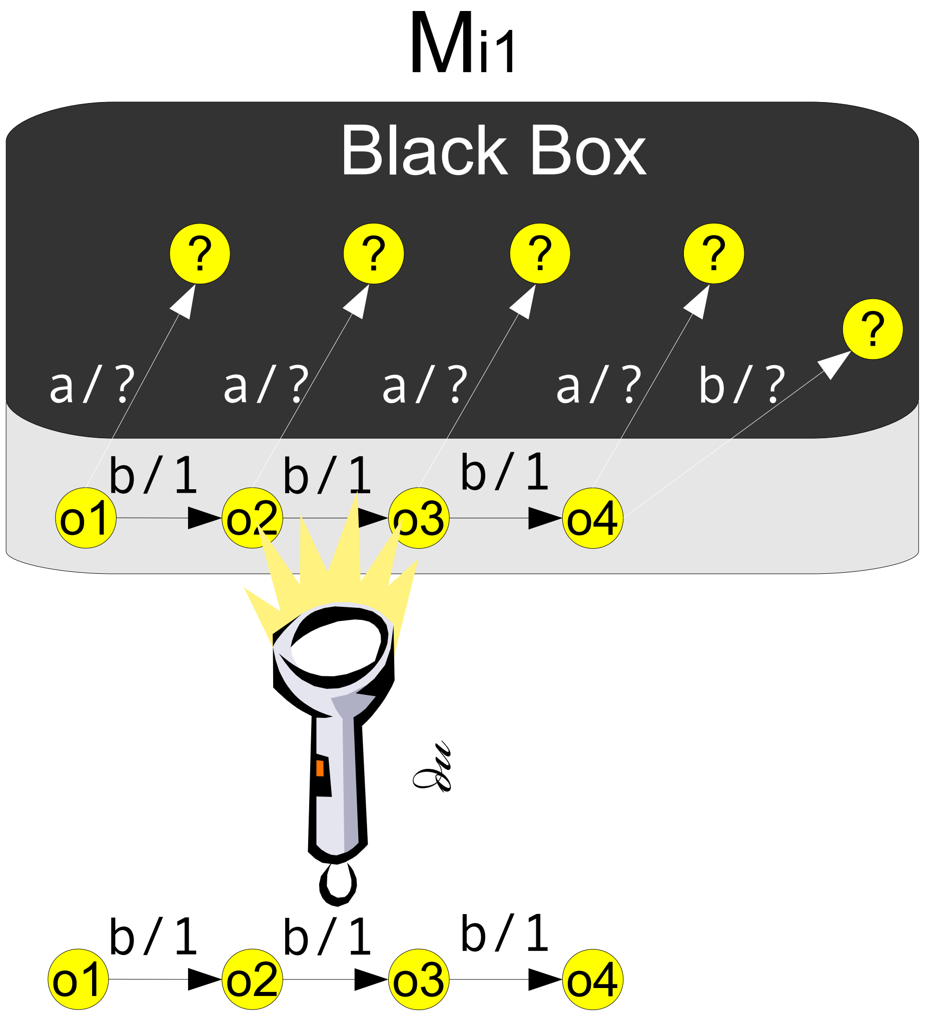
Implementation Models

- But we don't know the implementation model a priori, we have just executed a single test case!
- What has really happened, is this:



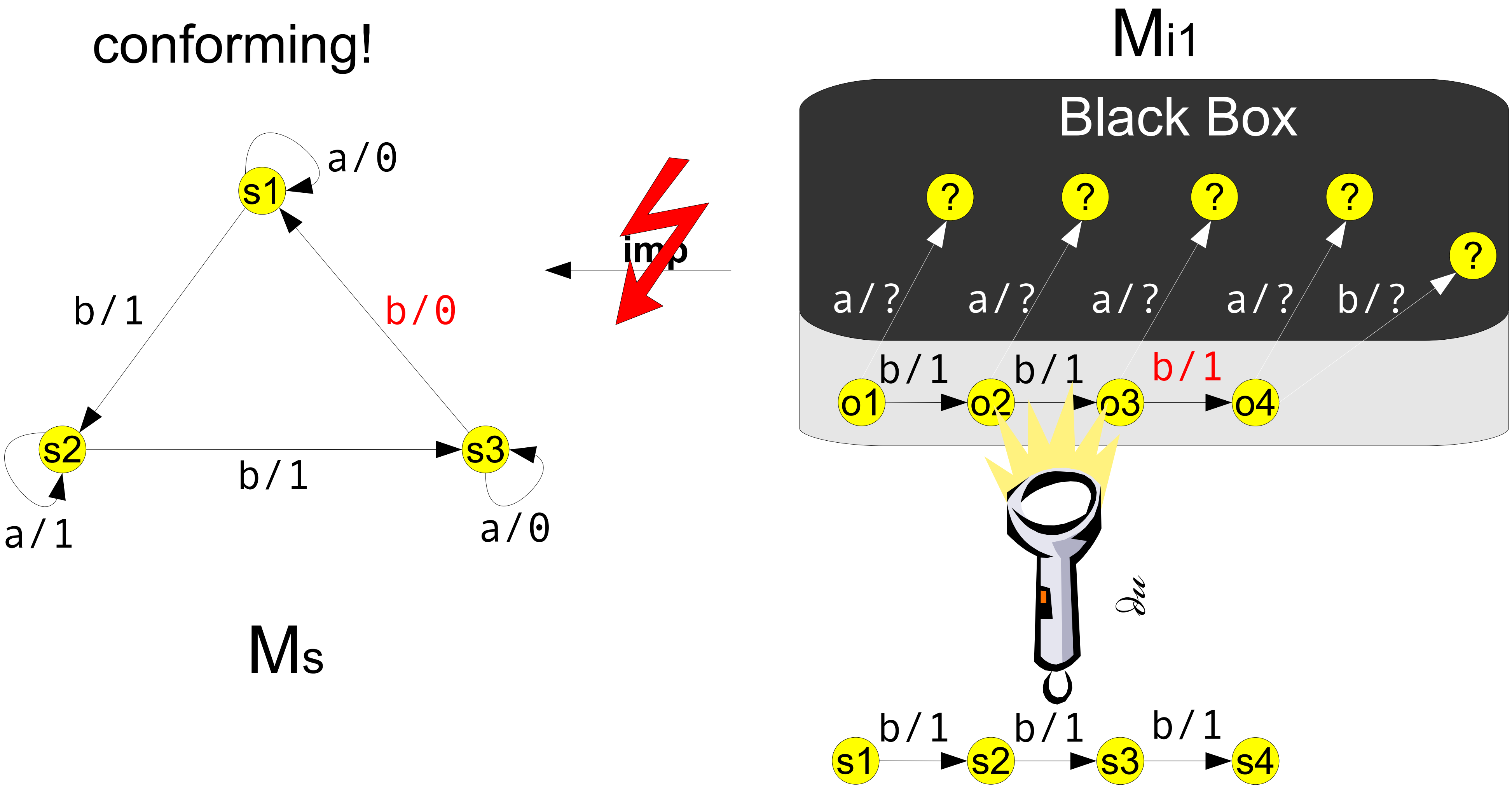
Implementation Models

- All we know is a little puzzle-piece from M_{i1}.



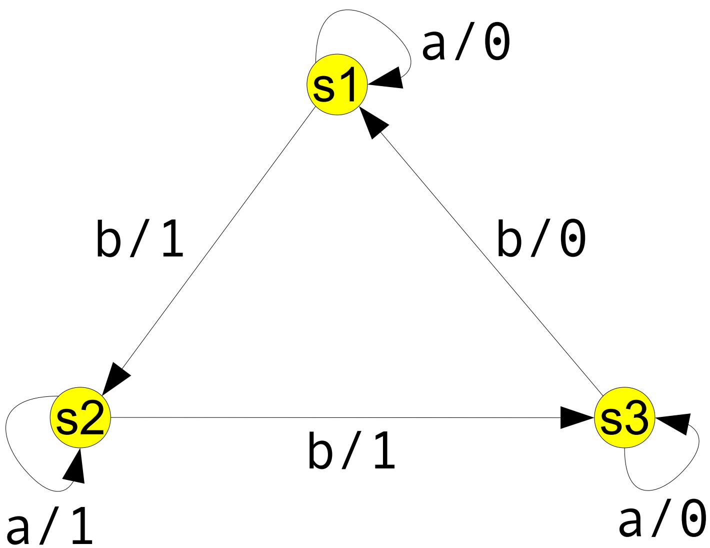
Implementation Models

- But this is sufficient to observe non-conformity, since **all possible** completions of the Black Box are non-conforming!



Test Generation

- A sound and **complete** test generation algorithm must generate **all possible test cases**.



Ms

Test Cases

Input	Output
a	0
b	1
ab	01
aab	001
bbb	110
abababab	01110001

...

Dijkstra Revisited

- A sound and **complete** test generation algorithm must generate **all possible test cases**.

Edsger W. Dijkstra
Edsger W. Dijkstra

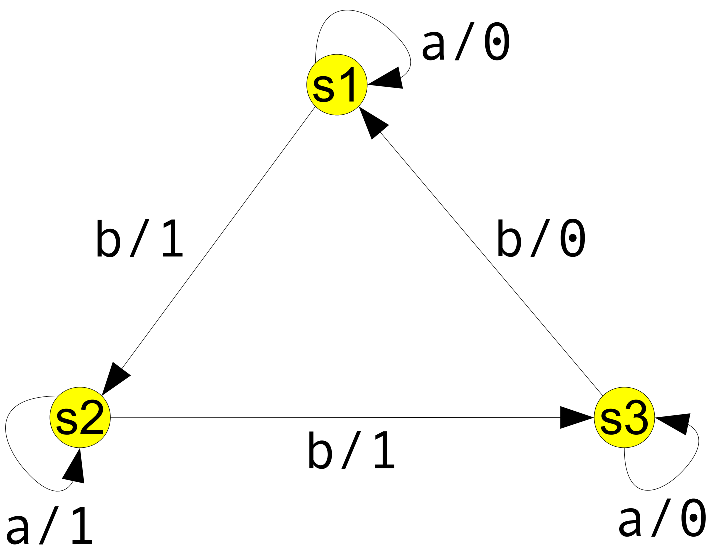
Test Cases

Input	Output
a	0
b	1
ab	01
aab	001
bbb	110
abababab	01110001

...

Test Generation

- A sound and **complete** test generation algorithm must generate **all possible test cases**.



Ms

Test Cases

Input	Output
~~a~~	~~0~~
~~b~~	~~1~~
~~ab~~	~~01~~
aab	001
bbb	110
abababab	01110001

...

Dijkstra Revisited

- **When** should we stop testing?
- **Which** test cases shall we select?

  ⇒ How to deal with the practical incompleteness of testing?

1) Accept it, and focus on **heuristics** like code coverage, model coverage, timing constraints, randomness, test purposes, etc.

2) Try to find further **assumptions**, which makes testing complete in practice, i.e., leading to a **finite** sound and complete test suite.

Dijkstra Revisited

- **When** should we stop testing?
- **Which** test cases shall we select?

  ⇒ How to deal with the practical incompleteness of testing?

1) Accept it, and focus on **heuristics** like code coverage, model coverage, timing constraints, randomness, test purposes, etc.

2) Try to find further **assumptions**, which makes testing complete in practice, i.e., leading to a **finite** sound and complete test suite.

Remember...

- *Dijkstra is right (of course).*
- *He refers to the fact, that the number of test cases in a sound and complete test suite is usually infinite (or at least too big).*
- *If that would not be the case, testing could **<u>prove</u>** the conformity of the system to the model (given some **<u>assumptions</u>** on the system).*

2) Try to find further **assumptions**, which makes testing complete in practice, i.e., leading to a **finite** sound and complete test suite.

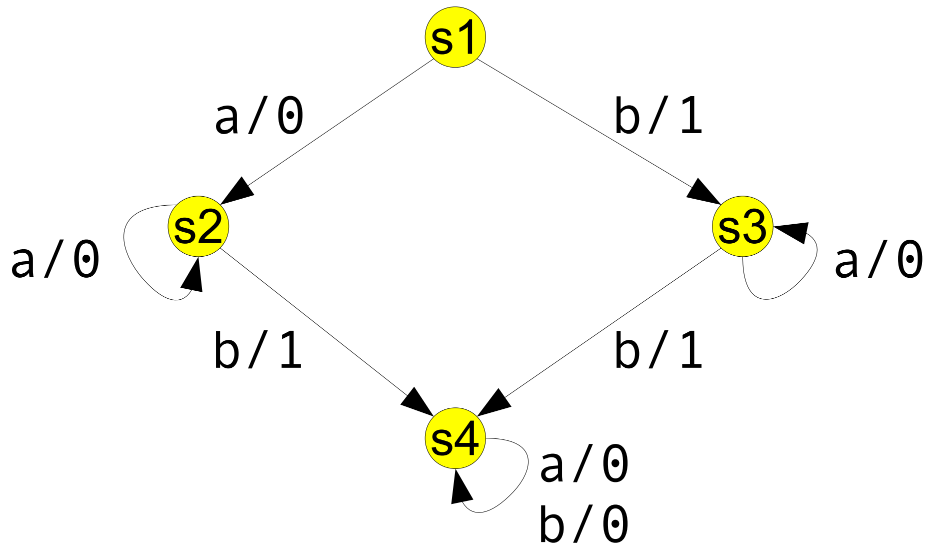
Checking Sequences

- A **checking sequence** for M_S is an input sequence that distinguishes the class of machines **equivalent** to M_S from all other machines.

- The length of this sequence can be used to compare the time complexity of the several algorithms.

Mandatory Assumptions

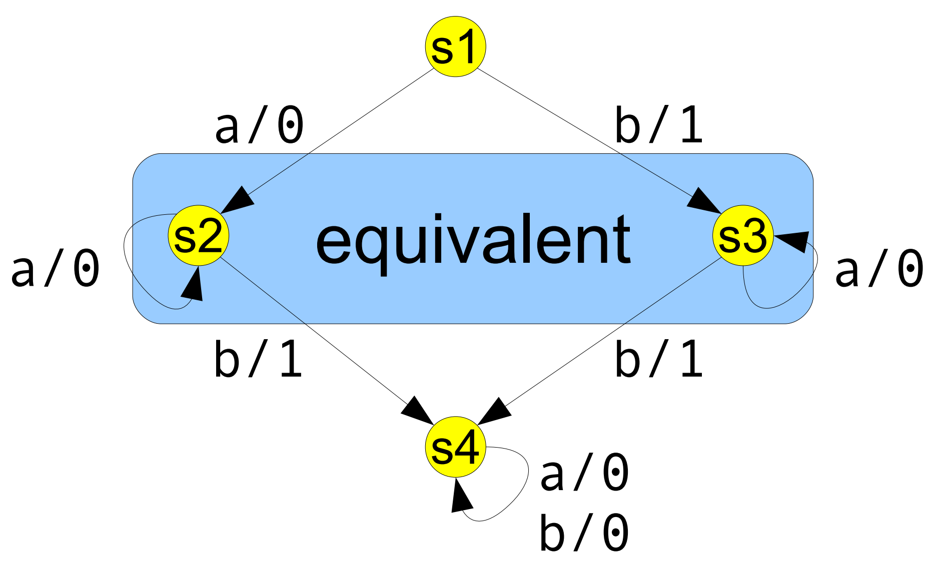
(1) M_s is **minimized**, meaning that M_s has no equivalent states. Equivalent states produce the same output sequence for every input sequence.



M_s

Mandatory Assumptions

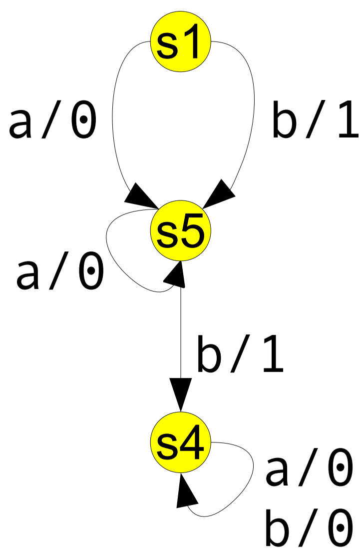
(1) M_S is **minimized**, meaning that M_S has no equivalent states. Equivalent states produce the same output sequence for every input sequence.



M_S

Mandatory Assumptions

(1) M_S is **minimized**, meaning that M_S has no <u>equivalent states</u>. Equivalent states produce the same output sequence for every input sequence.

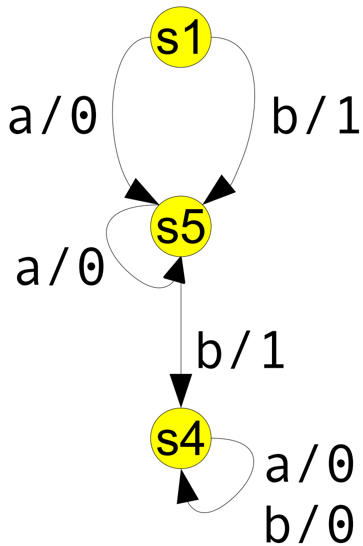


M_S (minimized)

Mandatory Assumptions

(1) M_S is **minimized**.

(2) M_S is **strongly connected**, meaning every state can reach every other state.



Ms is **not** strongly connected!

Ms (minimized)

Mandatory Assumptions

(1) M_S is **minimized**.

(2) M_S is **strongly connected**.

(3) M_I has the **same inputs and outputs** as M_S, and **does not change** during runtime.

Mandatory Assumptions

(1) M_S is **minimized**.

(2) M_S is **strongly connected**.

(3) M_I has the **same inputs and outputs** as M_S, and **does not change** during runtime.

Mandatory

Mandatory and Additional Assumptions

(1) M_S is **minimized**.

(2) M_S is **strongly connected**.

(3) M_I has the **same inputs and outputs** as M_S, and **does not change** during runtime.

(4) M_S and M_I have an **initial state**.

Mandatory and Additional Assumptions

(1) M_S is **minimized**.

(2) M_S is **strongly connected**.

(3) M_I has the **same inputs and outputs** as M_S, and **does not change** during runtime.

(4) M_S and M_I have an **initial state**.

(5) M_S and M_I have a **reset message** that from any state of the machine causes a transition which ends in the initial state, and produces no output.

Mandatory and Additional Assumptions

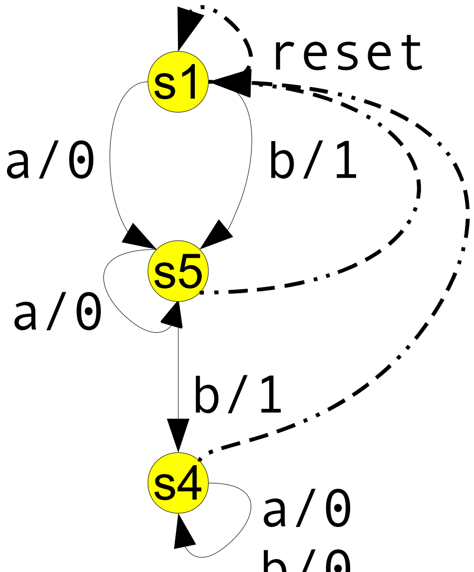
(1) M_S is **minimized**.

(2) M_S is **strongly connected**.

(3) M_I has the **same inputs and outputs** as M_S, and **does not change** during runtime.

(4) M_S and M_I have an **initial state**.

(5) M_S and M_I have a **reset message**.



Having a **reset message**, Ms **is** strongly connected!

Ms (minimized)

Mandatory and Additional Assumptions

(1) M_S is **minimized**.

(2) M_S is **strongly connected**.

(3) M_I has the **same inputs and outputs** as M_S, and **does not change** during runtime.

(4) M_S and M_I have an **initial state**.

(5) M_S and M_I have a **reset message**.

(6) M_I has **the same number of states** than M_S.

Mandatory and Additional Assumptions

(1) M_S is **minimized**.

(2) M_S is **strongly connected**.

(3) M_I has the **same inputs and outputs** as M_S, and **does not change** during runtime.

(4) M_S and M_I have an **initial state**.

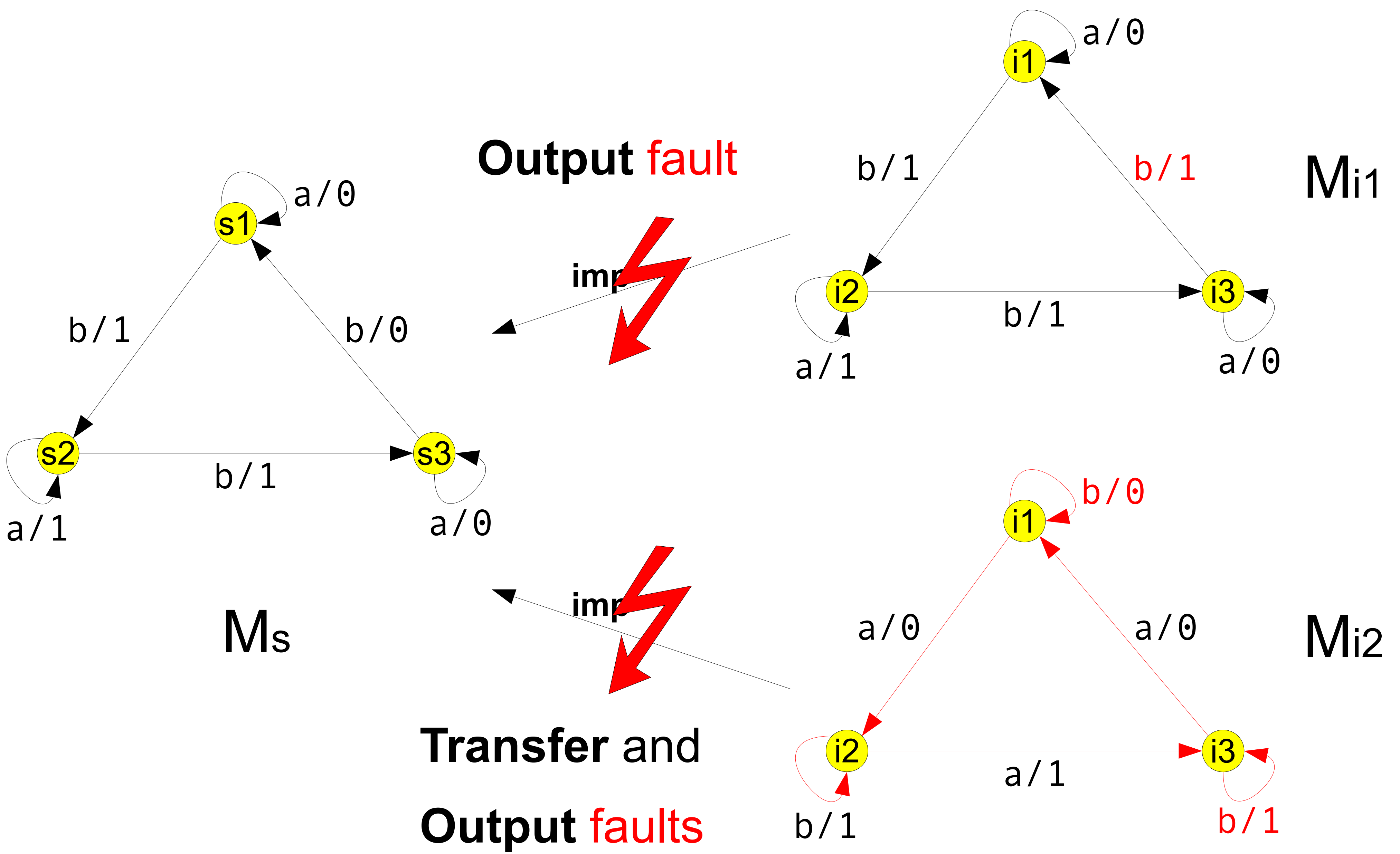
(5) M_S and M_I have a **reset message**.

(6) M_I has **the same number of states** than M_S. Under this assumption, two types of faults can be present in M_I:

Output faults: a transition produces a wrong output

Transfer faults:a transition goes to a wrong state

Output- and Transfer Faults



Mandatory and Additional Assumptions

(1) M_S is **minimized**.

(2) M_S is **strongly connected**.

(3) M_I has the **same inputs and outputs** as M_S, and **does not change** during runtime.

(4) M_S and M_I have an **initial state**.

(5) M_S and M_I have a **reset message**.

(6) M_I has **the same number of states** than M_S.

Mandatory and Additional Assumptions

(1) M_S is **minimized**.

(2) M_S is **strongly connected**.

(3) M_I has the **same inputs and outputs** as M_S, and **does not change** during runtime.

(4) M_S and M_I have an **initial state**.

(5) M_S and M_I have a **reset message**.

(6) M_I has **the same number of states** than M_S.

(7) M_S and M_I have a **status message**. Giving a particular input *"status"*, the output uniquely defines the current state.

Mandatory and Additional Assumptions

(1) M_S is **minimized**.

(2) M_S is **strongly connected**.

(3) M_I has the **same inputs and outputs** as M_S, and **does not change** during runtime.

(4) M_S and M_I have an **initial state**.

(5) M_S and M_I have a **reset message**.

(6) M_I has **the same number of states** than M_S.

(7) M_S and M_I have a **status message**.

(8) M_S and M_I have a **set message**. From the initial state the system can be transferred to every other state `s` by giving the input `set(s)`. No output is produced while doing so.

Mandatory and Additional Assumptions

(1) M_S is **minimized**.

(2) M_S is **strongly connected**.

(3) M_I has the **same inputs and outputs** as M_S, and **does not change** during runtime.

(4) M_S and M_I have an **initial state**.

(5) M_S and M_I have a **reset message**.

(6) M_I has **the same number of states** than M_S.

(7) M_S and M_I have a **status message**.

(8) M_S and M_I have a **set message**.

Additional

A Sound and Complete Algorithm

For all states s and all inputs a do:

1. Apply the **reset message** to bring M_i to the initial state.
2. Apply a **set message** to transfer M_i to state s.
3. Apply the input a.
4. Verify that the output conforms to the specification M_S.
5. Apply the **status message** and verify that the final state conforms to the specification M_S.

This algorithm is **sound and complete** given that all assumptions (1) – (8) hold.

The length of the **checking sequence** is 4 * |I| * |S|

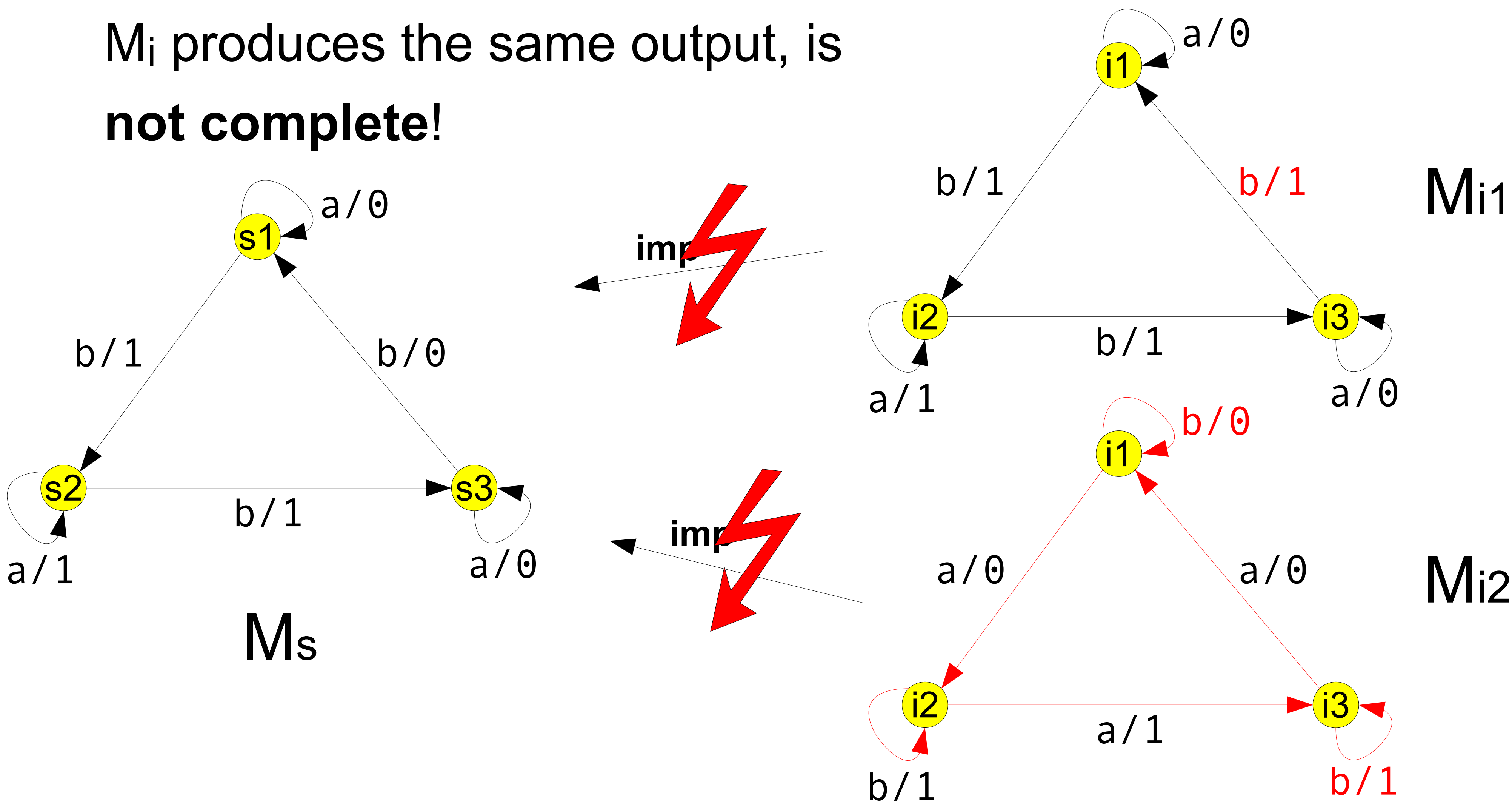
Transition Tours

- To get rid of the **set message**, and possibly shorten the test suite, we can build a sequence that visits every state and every transition at least once – a **transition tour**.
- The shortest transition tour visits each transition exactly once, and is called an **Euler tour**. It only exists for **symmetric** FSM (every state is the start state and end state of the same number of transitions).
- An Euler Tour can be computed in linear time w.r.t. the number of transitions.
- In non-symmetric FSM finding the shortest tour is referred to as the **Chinese Postman Problem**. It can be solved in polynomial time.

Transition Tours

- **Problem:**

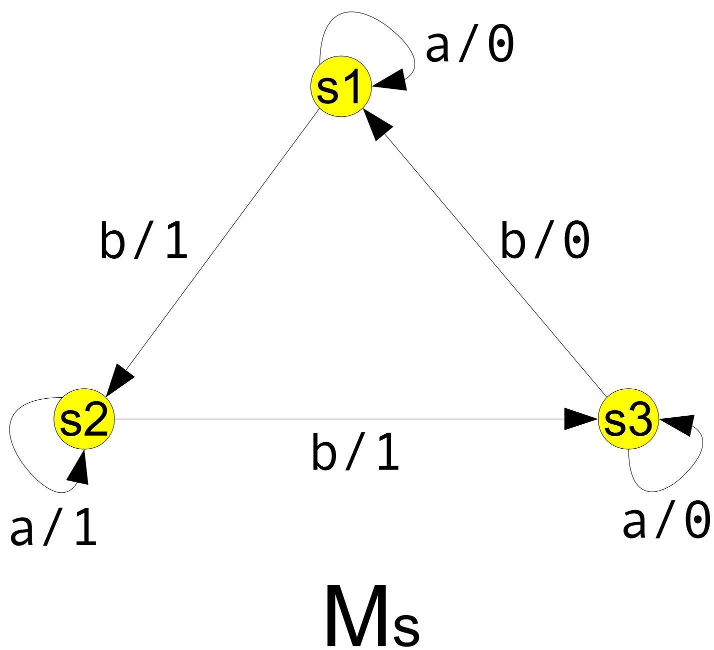
  Covering all transitions of M_s, and checking whether M_i produces the same output, is **not complete**!

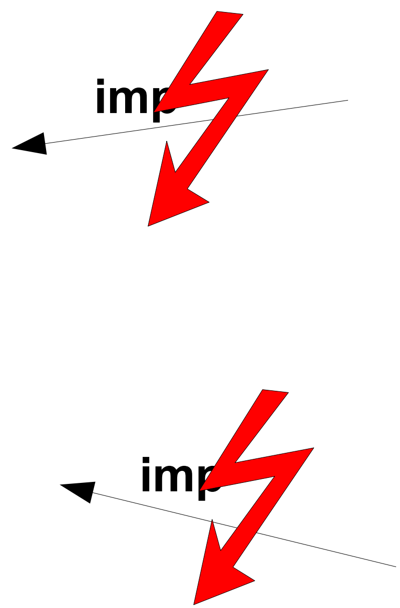


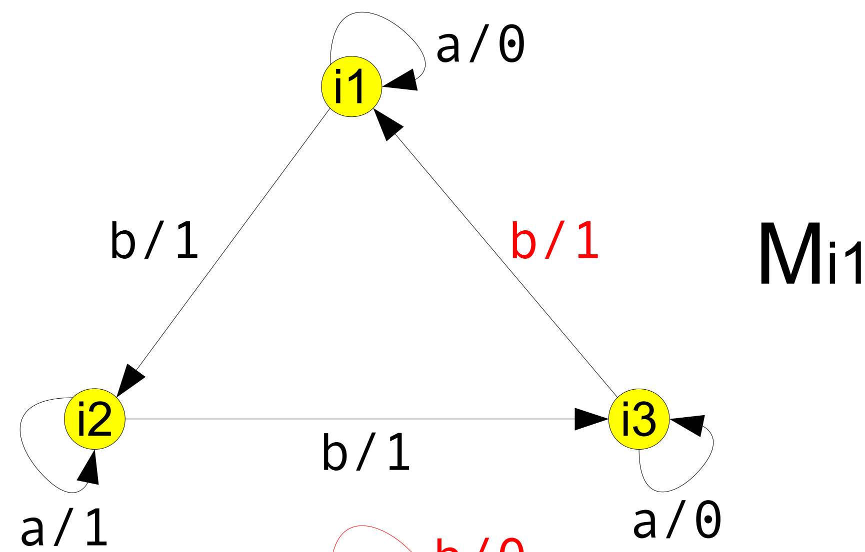
Transition Tours

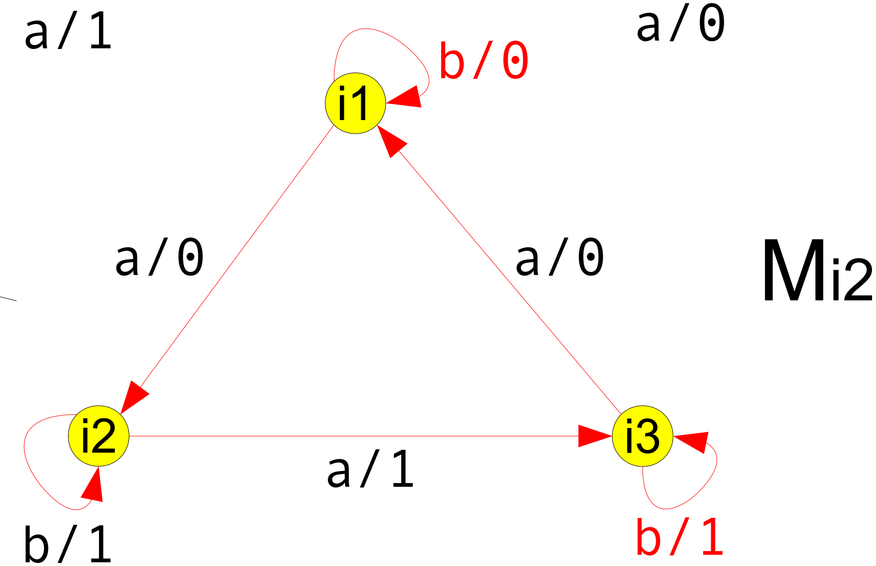
- **Problem:**

  Covering all transitions of M_s, and checking whether M_i produces the same output, is **not complete**!









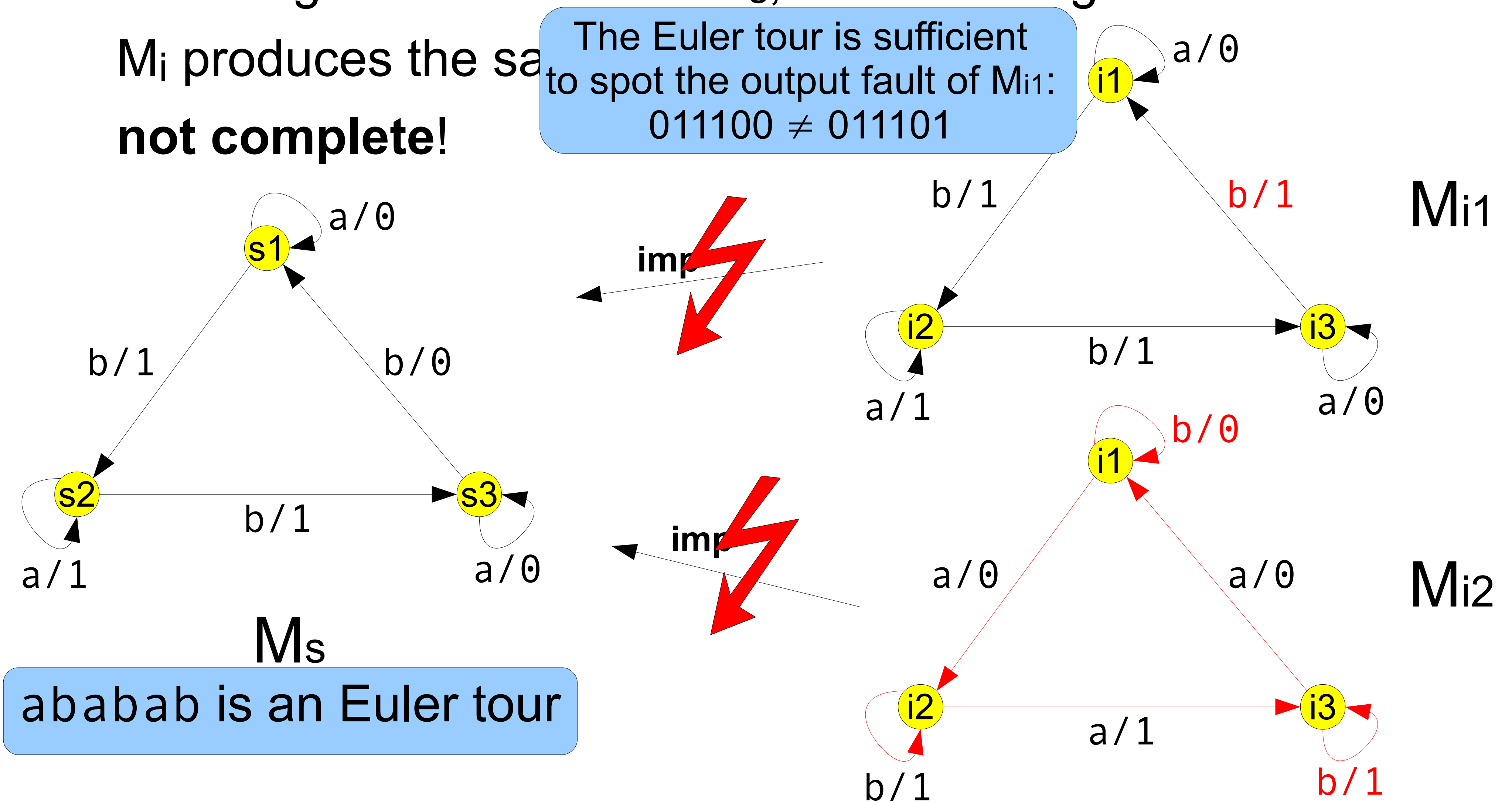
ababab is an Euler tour

Transition Tours

- **Problem:**

  Covering all transitions of M_s, and checking whether M_i produces the sa

  not complete!

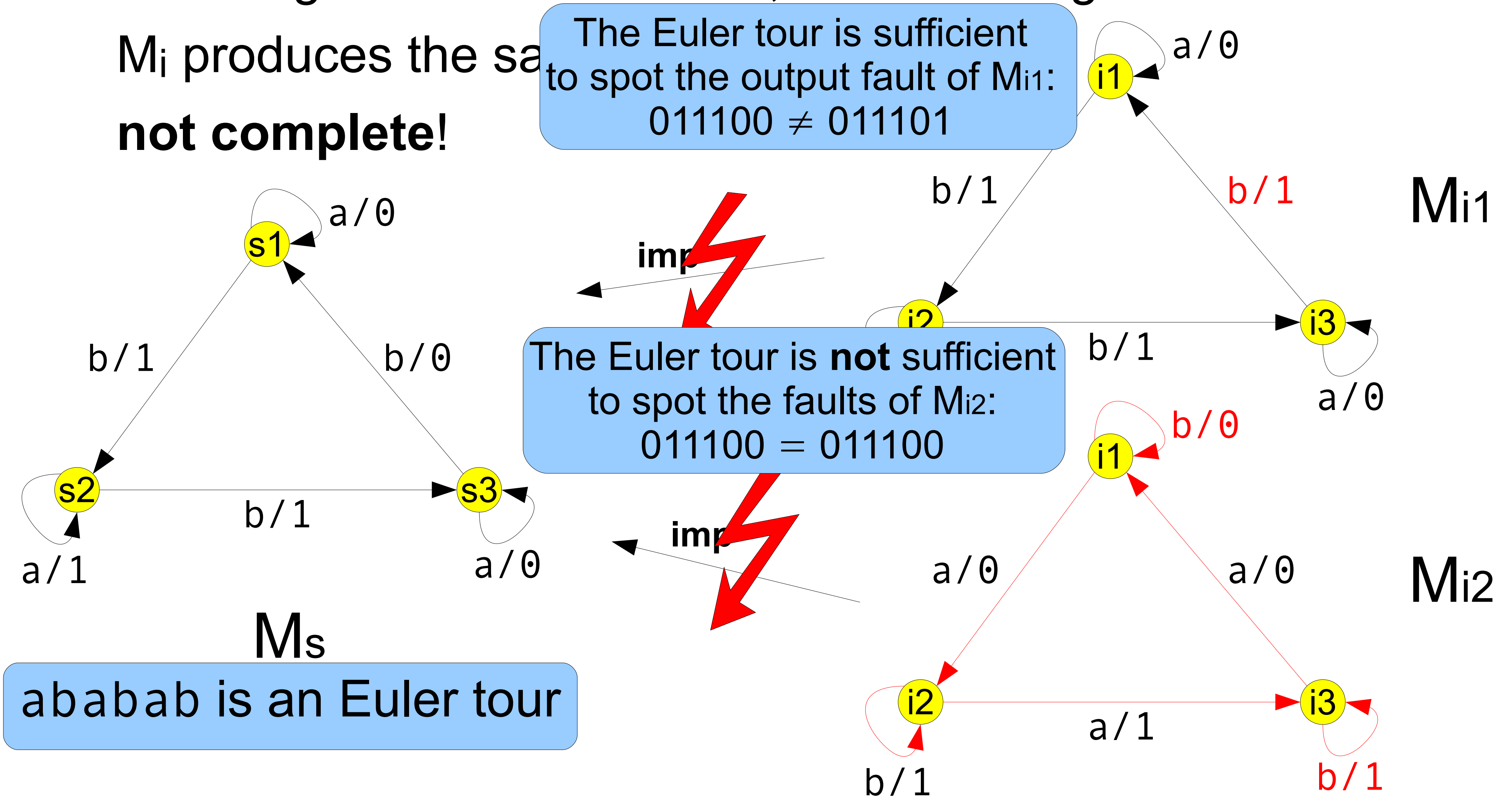


Transition Tours

- **Problem:**

  Covering all transitions of M_s, and checking whether M_i produces the sa

  not complete!



State Identification and Verification

- **Solution 1:**

  Use the **status message** to **verify the states** while doing the transition tour.

- **Solution 2:**

  If the status message does not exists, use **separating sequences** instead. Examples are:

  - Characterizing set (W Method)
  - Identification set (Wp Method)
  - UIO sequence (UIO Methods)
  - Distinguishing sequence (Distinguishing Sequence Method)
  - ...

- Not all separating sequences are guaranteed to exists.
- Not all of these methods are complete.

Where's the Reset Button?

- When even a **reset message** is not available, more can be done...

...transfer sequences...

...using distinguishing sequences without reset...

...using identifying sequences instead of distinguishing sequences...

...adaptive distinguishing sequences...

...homing sequence...

The General Procedure

- Every method follows the same scheme:

For all states s and all inputs a do:

1. Bring M_i to the state s.
2. Apply the input a.
4. Verify that the output conforms to the specification M_S.
5. Verify that the final state conforms to the specification M_S.

Summary

- The test hypothesis for FSM-based testing makes some **general assumptions** regarding the system to be tested:
  - The system has **finite state**.
  - The system is **deterministic**.
  - The system communicates in a **synchronous** manner (input / output).
- FSM-based testing focused on testing for **equivalence**.
- Based on a given set of **further mandatory and additional assumptions**, the FSM algorithms can give a **finite sound and complete** test suite.
- In other words, these algorithm can **prove** the equivalence.
- Most of the theoretical problems have been solved.

Summary

- FSM-based testing can be the underlying testing model of several other formalisms, like UML state machines, Abstract State Machines, RPC-like systems, etc.
- Tools related to FSM-based testing are for instance:
  - Conformance Kit, PHACT, TVEDA, Autofocus, AsmL Test Tool,...
- Results regarding other types of state machines have shown that there is no hope that feasible algorithms can yield a **finite** sound and complete test suite, for instance:
  - Nondeterministic machines
  - Probabilistic machines
  - Symbolic machines
  - Real-Time machines
  - Hybrid machines

Labeled Transition Systems

- Original domains:
  - sequential and concurrent programs
  - hardware circuits
- Several formalisms have an underlying **Labeled Transition System (LTS)** semantics, for instance:
  - Statecharts
  - Process Algebras

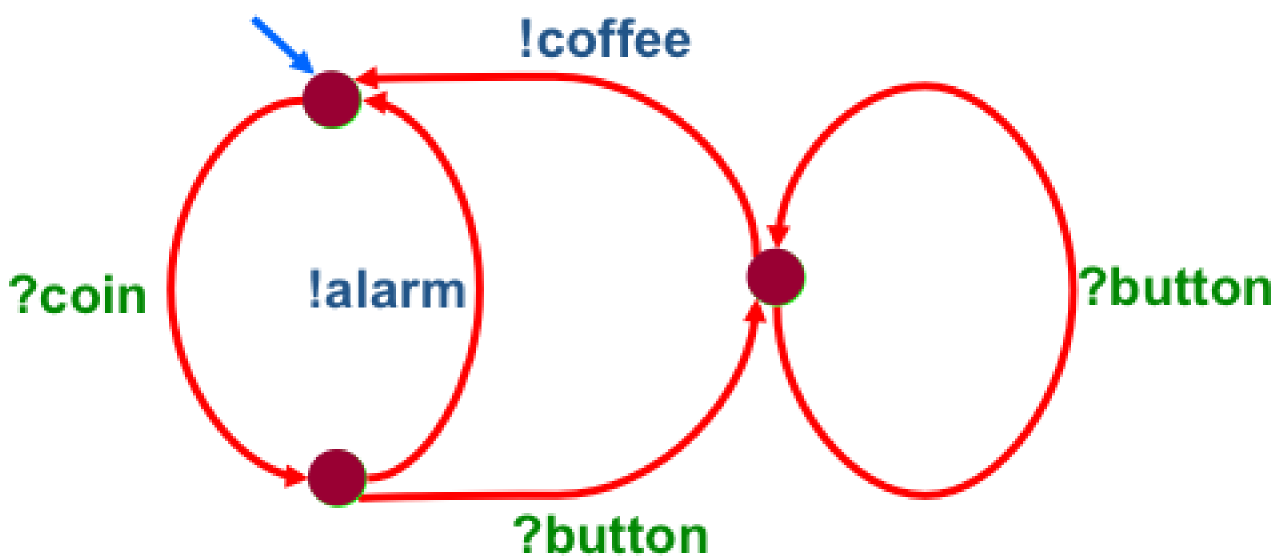
Models: Labelled Transition Systems

Labelled Transition System: $< S, L_I, L_U, T, s_0 >$

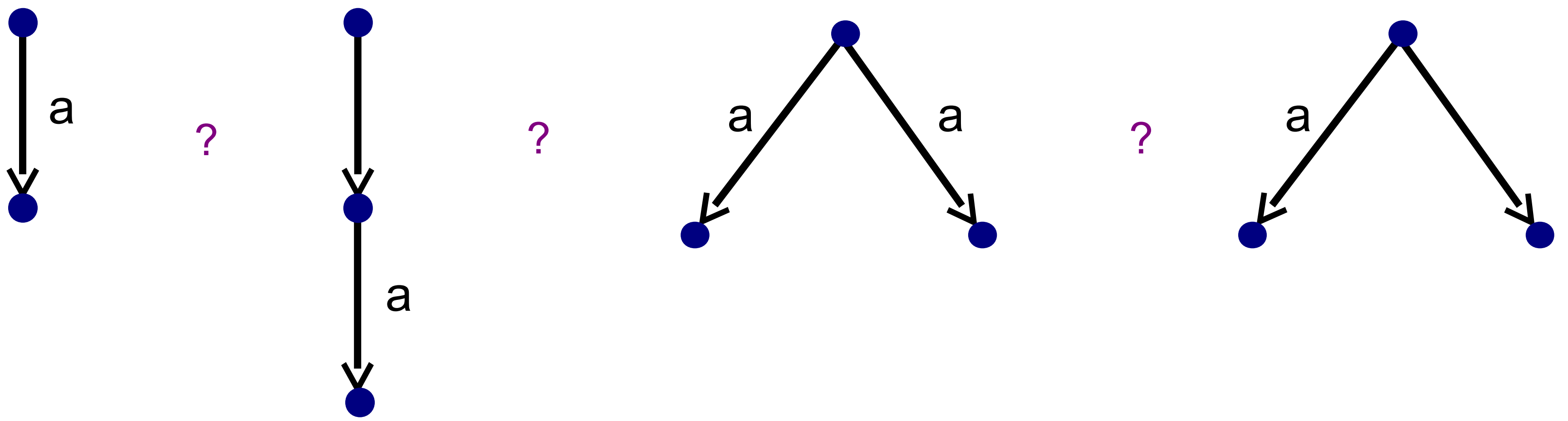
- S: states
- L_I: input actions
- L_U: output actions
- T: transitions
- s_0: initial state

? = input

! = output



Observable Behaviour



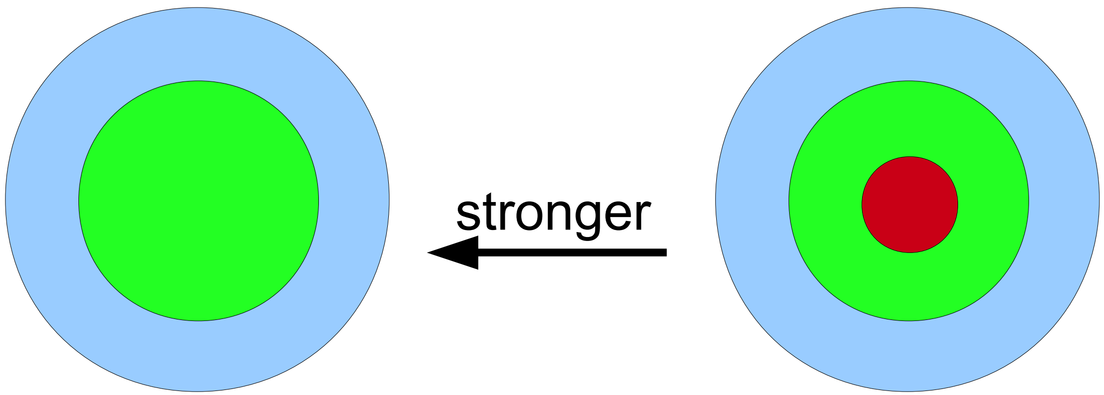
“ Some systems are more equal than others “

Conformance

- Relating two LTS can be done in a variety of manners, e.g.:
- **Equivalence relations:**
  - Isomorphism, Bisimulation, Trace Equivalence, Testing Equivalence, Refusal Equivalence, ...
- **Preorder relations:**
  - Observation Preorder, Trace Preorder, Testing Preorder, Refusal Preorder, ...
- **Input-Output relations:**
  - Input-Output Testing
  - Input-Output Refusal
  - *ioconf*
  - *ioco*
- ...

Conformance

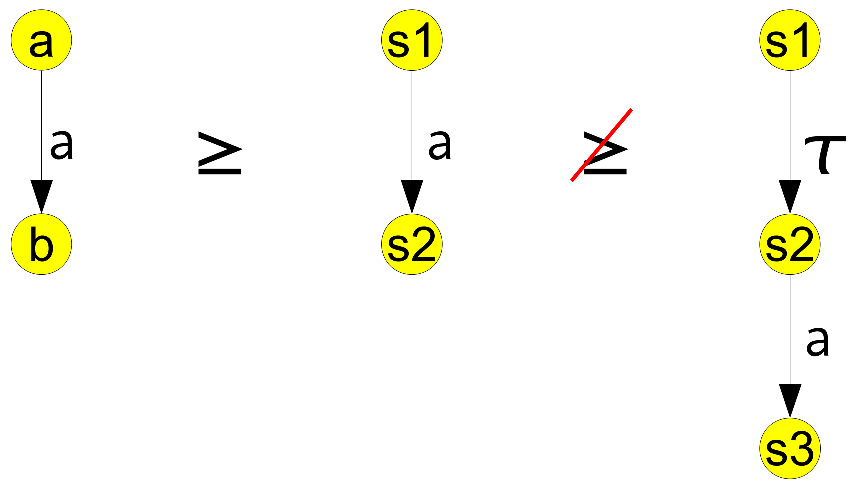
- An implementation relation is called **stronger** than another, if the classes of related LTS are more differentiated.



- Implementation relations may also be incomparable.
- We want an implementation relation to
  - relate systems we **naturally consider** as being conforming
  - be **applicable** in practice, i.e., having a feasible testing scenario
  - be **as strong as possible**

Isomorphism

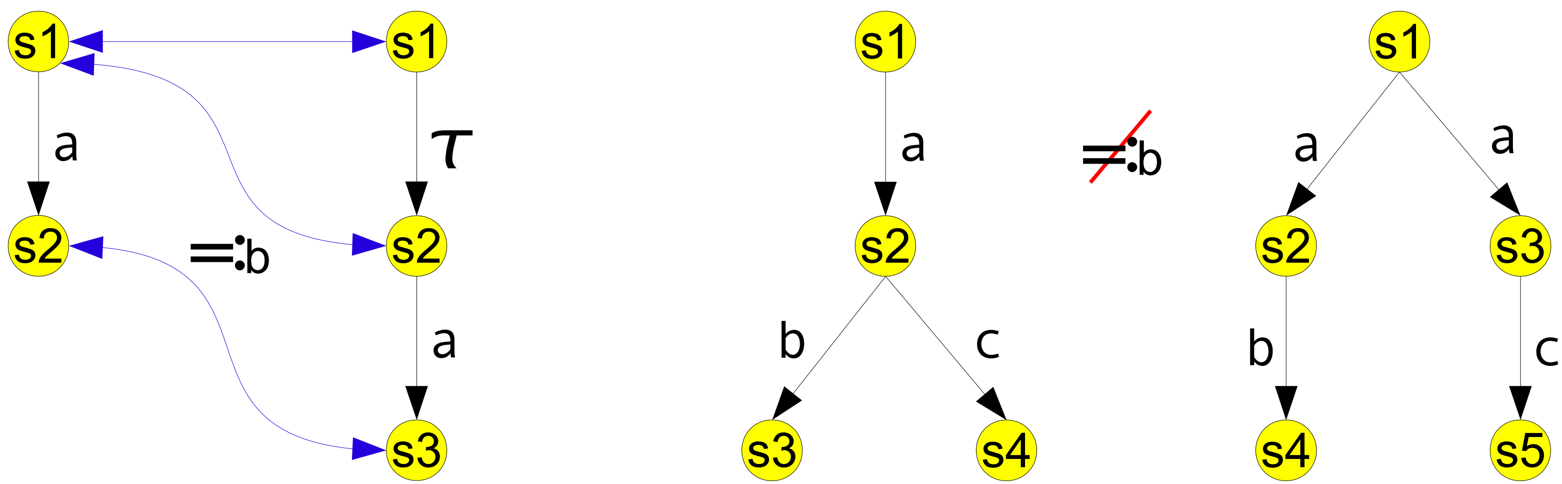
- Two LTS are **isomorph** (or: equivalent) iff they are exactly the same modulo state names.



- Isomorphism is the strongest notion of conformance.
- Isomorphism is <u>not suited for testing</u> since we cannot observe the unobservable τ action!

Bisimulation

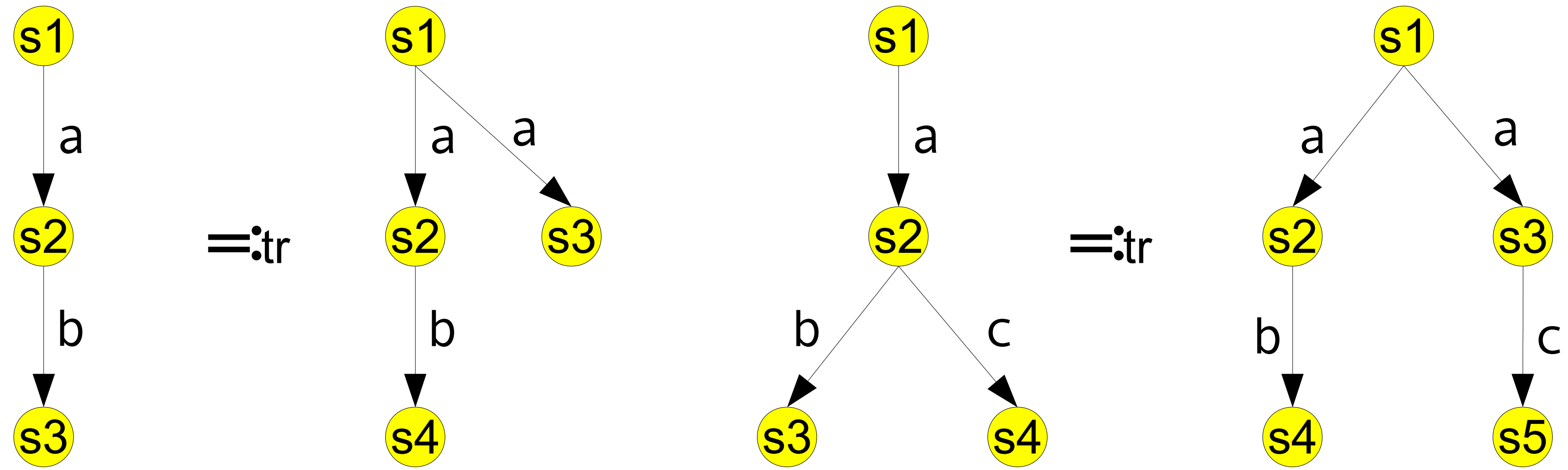
- Two LTS are (weak) **bisimular** iff they simulate each other and go to states from where they can simulate each other again.



- Bisimulation is <u>not suited for testing</u> since its testing scenario comprises means which are infeasible in practice.

Trace Equivalence

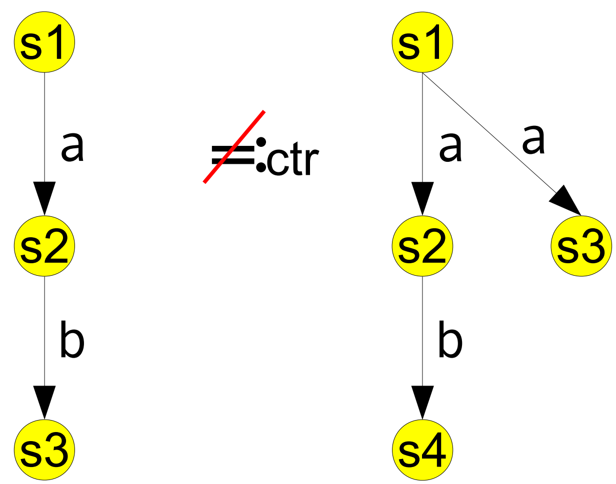
- A **trace** is an <u>observable</u> sequence of actions.
- Two LTS are **trace equivalent** iff they have the same traces.



- Trace equivalence is the weakest notion of conformance.
- For testing purposes it is usually considered <u>too weak</u>.
- isomorphism $\subset$ bisimularity $\subset$ trace equivalence

Completed Trace Equivalence

- A **completed trace** is a trace leading to a state **refusing** all actions – a final state.
- Two LTS are **completed trace equivalent** iff they are trace equivalent, and also share the same completed traces.



- Here we need to be able to observe the absence of **all** actions, i.e., **deadlocks**.

More Relations

- **Testing equivalence** is stronger than completed trace equivalence, and demands a test scenario which can observe the refusal of actions.
- ***conf*** is a modification of testing equivalence restricting the observations to only those traces contained in the specification (***conf*** is not transitive).
- **Refusal equivalence** is stronger than testing equivalence, and demands a test scenario which can continue the test after observing the refusal of actions.

- Tool: **Cooper** for the Co-Op method for ***conf***

Preorder Relations

- **i imp s** means that implementation model **i** implements specification model **s**.
- Do we want **imp** to be
  - **reflexive** — s **imp** s ✔
  - **symmetric** — i **imp** s ⇒ s **imp** i ✘
  - **transitive** — i **imp** s ∧ s **imp** t ⇒ i **imp** t ✔
  - **anti-symmetric** — i **imp** s ∧ s **imp** i ⇒ i = s ✘
  - **total** — i **imp** s ∨ s **imp** i ✘
  - **congruent** — i **imp** s ⇒ f(i) **imp** f(s) ✔

- An **equivalence** is reflexive, symmetric and transitive.
- A **preorder** is just reflexive and transitive.

Preorder Relations

- The motivation for preorder relations is to simplify the testing scenario.
- For almost every equivalence $\approx$ a corresponding preorder $\leq$ can be defined such that

$$p \approx q \;\Leftrightarrow\; p \leq q \wedge q \leq p$$

- **Trace preorder**:

  $i \leq_{tr} s \Leftrightarrow traces(i) \subseteq traces(s)$
- In the same way **testing preorder** $\leq_{te}$ and **refusal preorder** $\leq_{rf}$ can be defined.

Input-Output Labeled Transition Systems

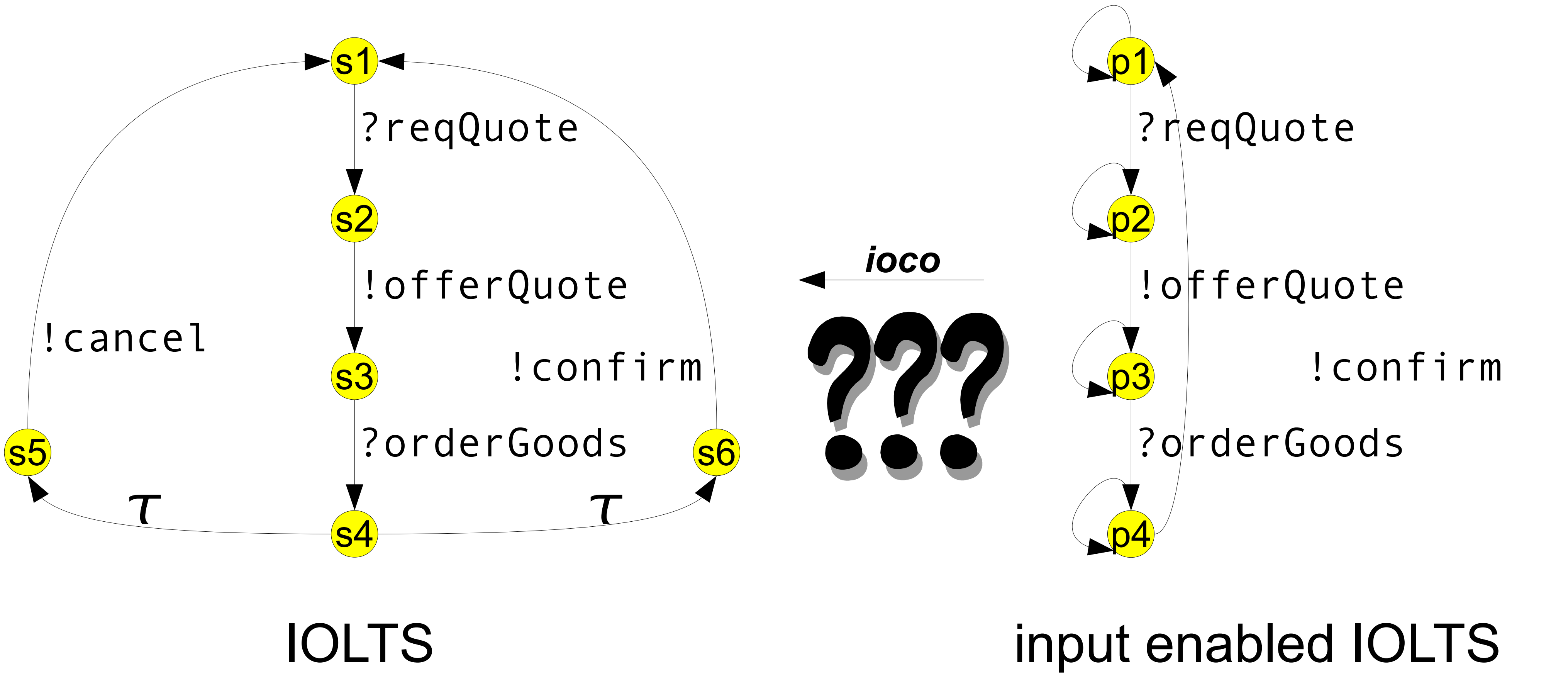
- In Input-Output relations, the set of action labels is partitioned into **input actions** and **output actions**, leading to an **Input-Output Labeled Transition System (IOLTS)**.

- Compared to **FSM**, **IOLTS** differ in
  - having **asynchronous** transitions (either input or output)
  - having potentially an **infinite** number of states
  - being potentially **nondeterministic**
  - being **not** necessarily **completely specified** for all inputs
  - being **compositional**

- An IOLTS, which is completely specified for all inputs is called an **input enabled IOLTS**.

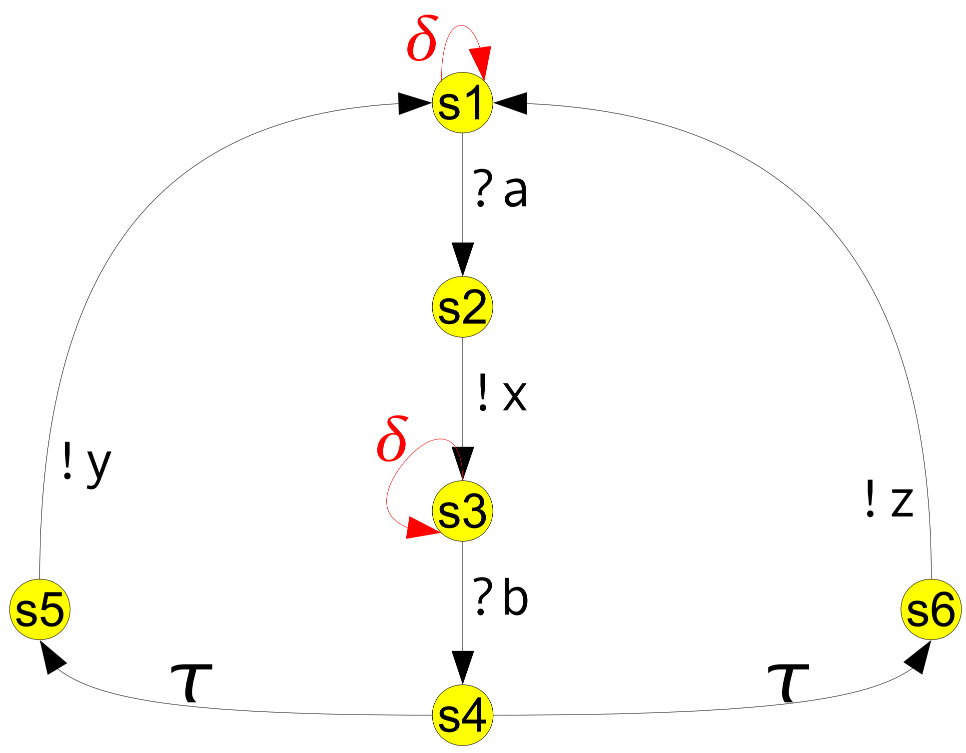
ioco - Conformance

- Specification models are IOLTS
- Implementation models are input-enabled IOLTS
- What does *ioco*-conformance mean?



Quiescence

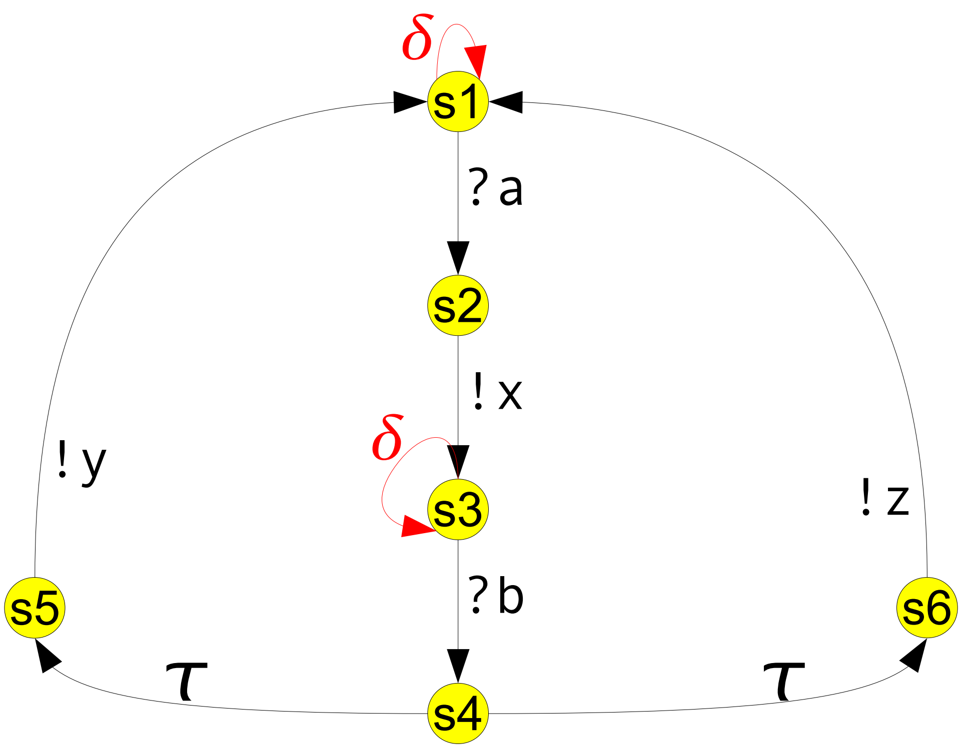
Definition 3 (Quiescence). *A state* $s \in S$ *in* $\mathcal{L}$ *is* quiescent, *denoted* $\delta(s)$, *iff* $\forall \mu \in \Sigma_U \cup \{\tau\}$: $s \stackrel{\mu}{\not\rightarrow}$. *Let* δ *be a constant not part of any action label set;* Σ_δ *abbreviates* $\Sigma_I \cup \Sigma_U \cup \{\delta\}$



after

Definition 4 (after). *Let $s \in S$, $C \subseteq S$ and $\sigma \in \Sigma_\delta^*$. We define*

$$s\ \textbf{after}\ \sigma\ =_{def} \{s' \in S \mid s \xRightarrow{\sigma}_\delta s'\}$$



s1 after δ = {s1}

s1 after ?a = {s2}

s2 after !x?b = {s4,s5,s6}

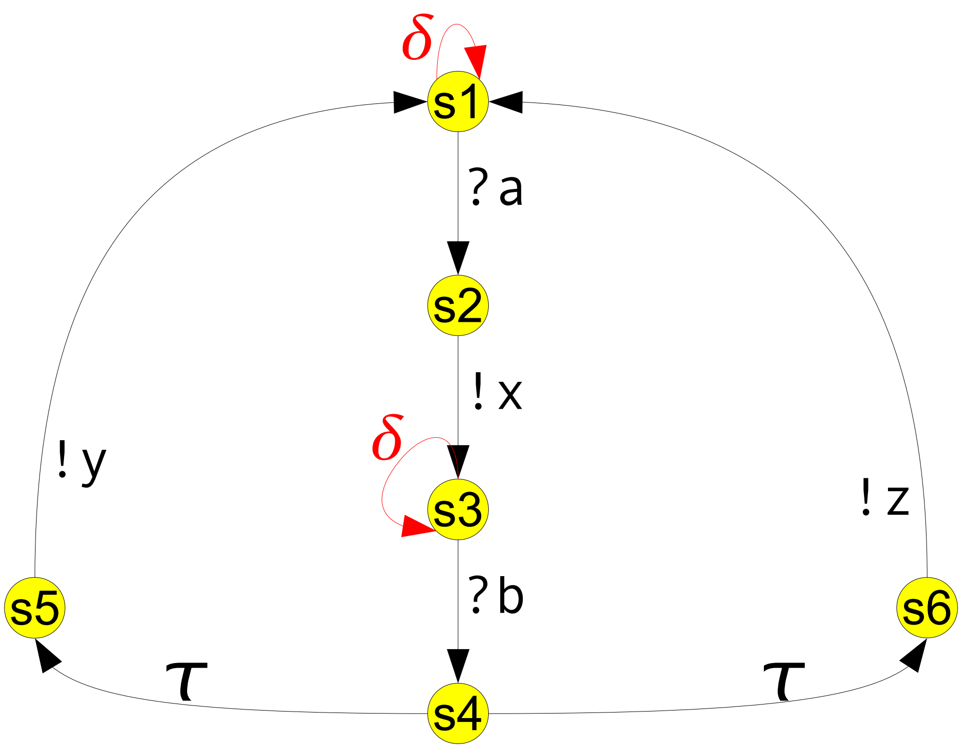
s2 after !xδ?b = {s4,s5,s6}

s1 after ?a!x$\delta\delta$?b!zδ = {s1}

out

Definition 5 (out). *Let $s \in S$ and $C \subseteq S$. We define*

- $out(s) =_{def} \begin{cases} \{\delta\} & if\ \delta(s) \\ \{a \in \Sigma_U \mid s \xrightarrow{a}\} & otherwise \end{cases}$
- $out(C) =_{def} \bigcup_{s \in C} out(s)$



out(s1) = {δ}
out(s2) = {!x}
out(s3) = {δ}
out(s4) = $\emptyset$
out(s5) = {!y}
out({s1,s2}) = {δ,!x}

out(s1 after δ?a) = {!x}
out(s1 after ?a!x?b) = {!y,!z}

ioco

Definition 6. *Let* $\mathcal{S} = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{S}} \rangle$ *be an IOLTS, and let* $\mathcal{F} \subseteq \Sigma_\delta^*$. *An input-enabled IOLTS* $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{P}} \rangle$ *is* $\mathbf{ioco}_{\mathcal{F}}$*-conform to* $\mathcal{S}$*, denoted by* $\mathcal{P}$ $\mathbf{ioco}_{\mathcal{F}}$ $\mathcal{S}$*, iff*

$$\forall \sigma \in \mathcal{F} : out(\, p_1 \ \mathbf{after}\ \sigma \,) \subseteq out(\, s_1 \ \mathbf{after}\ \sigma \,)$$

ioco

Definition 6. *Let* $\mathcal{S} = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_\mathcal{S} \rangle$ *be an IOLTS, and let* $\mathcal{F} \subseteq \Sigma_\delta^*$. *An input-enabled IOLTS* $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_\mathcal{P} \rangle$ *is* $\mathbf{ioco}_\mathcal{F}$*-conform to* $\mathcal{S}$*, denoted by* $\mathcal{P}$ $\mathbf{ioco}_\mathcal{F}$ $\mathcal{S}$*, iff*

$$\forall \sigma \in \mathcal{F} : out(\, p_1 \ \mathbf{after}\ \sigma\,) \subseteq out(\, s_1 \ \mathbf{after}\ \sigma\,)$$

Definition 7 (Suspension Traces). *The set of* suspension traces *is defined as* $Straces(s) =_{def} \{\sigma \in \Sigma_\delta^* \mid s \stackrel{\sigma}{\Longrightarrow}_\delta\}$

Just writing *ioco* abbreviates *ioco*$_F$ with F = Straces(s_0).

ioco

Definition 6. *Let $\mathcal{S} = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{S}} \rangle$ be an IOLTS, and let $\mathcal{F} \subseteq \Sigma_\delta^*$. An input-enabled IOLTS $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{P}} \rangle$ is* $\mathbf{ioco}_{\mathcal{F}}$*-conform to $\mathcal{S}$, denoted by $\mathcal{P}$* $\mathbf{ioco}_{\mathcal{F}}$ *$\mathcal{S}$, iff*

$$\forall \sigma \in \mathcal{F} : out(\, p_1 \ \mathbf{after}\ \sigma \,) \subseteq out(\, s_1 \ \mathbf{after}\ \sigma \,)$$

Intuition:

i *ioco* **s**, iff

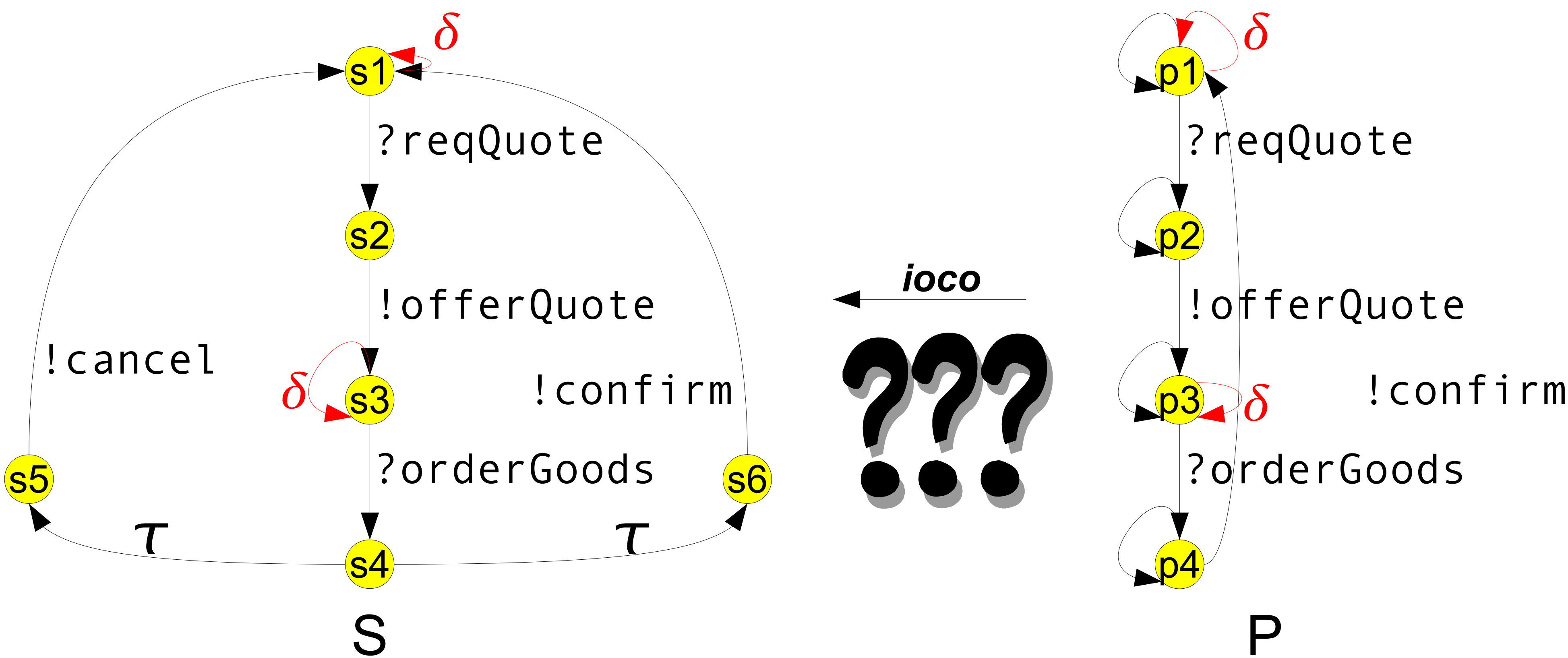
- if **i** produces output x after trace σ,
  then **s** can produce x after trace σ.

- if **i** cannot produce any output after trace σ,
  then **s** cannot produce any output after trace σ (quiescence).

ioco

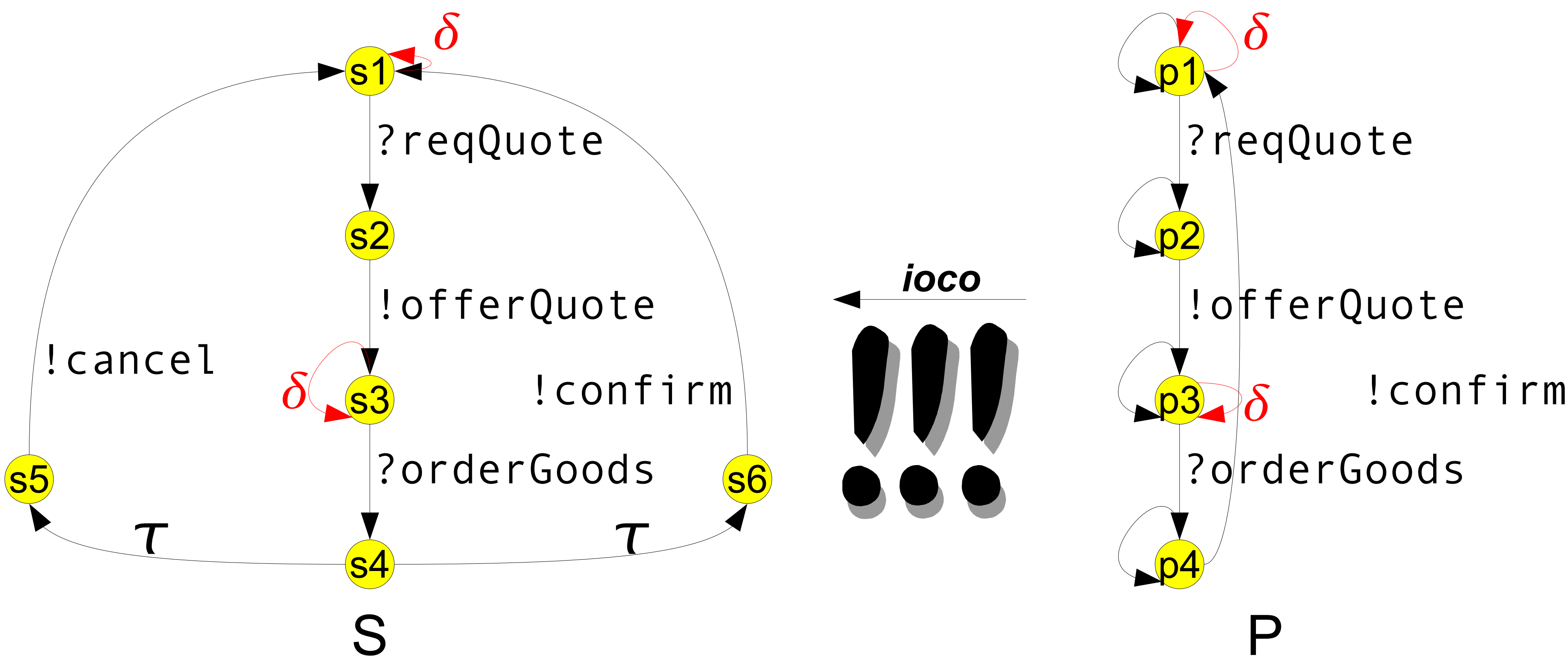
Definition 6. *Let $\mathcal{S} = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{S}} \rangle$ be an IOLTS, and let $\mathcal{F} \subseteq \Sigma_\delta^*$. An input-enabled IOLTS $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{P}} \rangle$ is* $\mathbf{ioco}_{\mathcal{F}}$*-conform to $\mathcal{S}$, denoted by $\mathcal{P}$* $\mathbf{ioco}_{\mathcal{F}}$ *$\mathcal{S}$, iff*

$$\forall \sigma \in \mathcal{F} : out(\, p_1 \ \mathbf{after}\ \sigma \,) \subseteq out(\, s_1 \ \mathbf{after}\ \sigma \,)$$



Definition 6. *Let $\mathcal{S} = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{S}} \rangle$ be an IOLTS, and let $\mathcal{F} \subseteq \Sigma_\delta^*$. An input-enabled IOLTS $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{P}} \rangle$ is* $\mathbf{ioco}_{\mathcal{F}}$*-conform to $\mathcal{S}$, denoted by $\mathcal{P}$* $\mathbf{ioco}_{\mathcal{F}}$ *$\mathcal{S}$, iff*

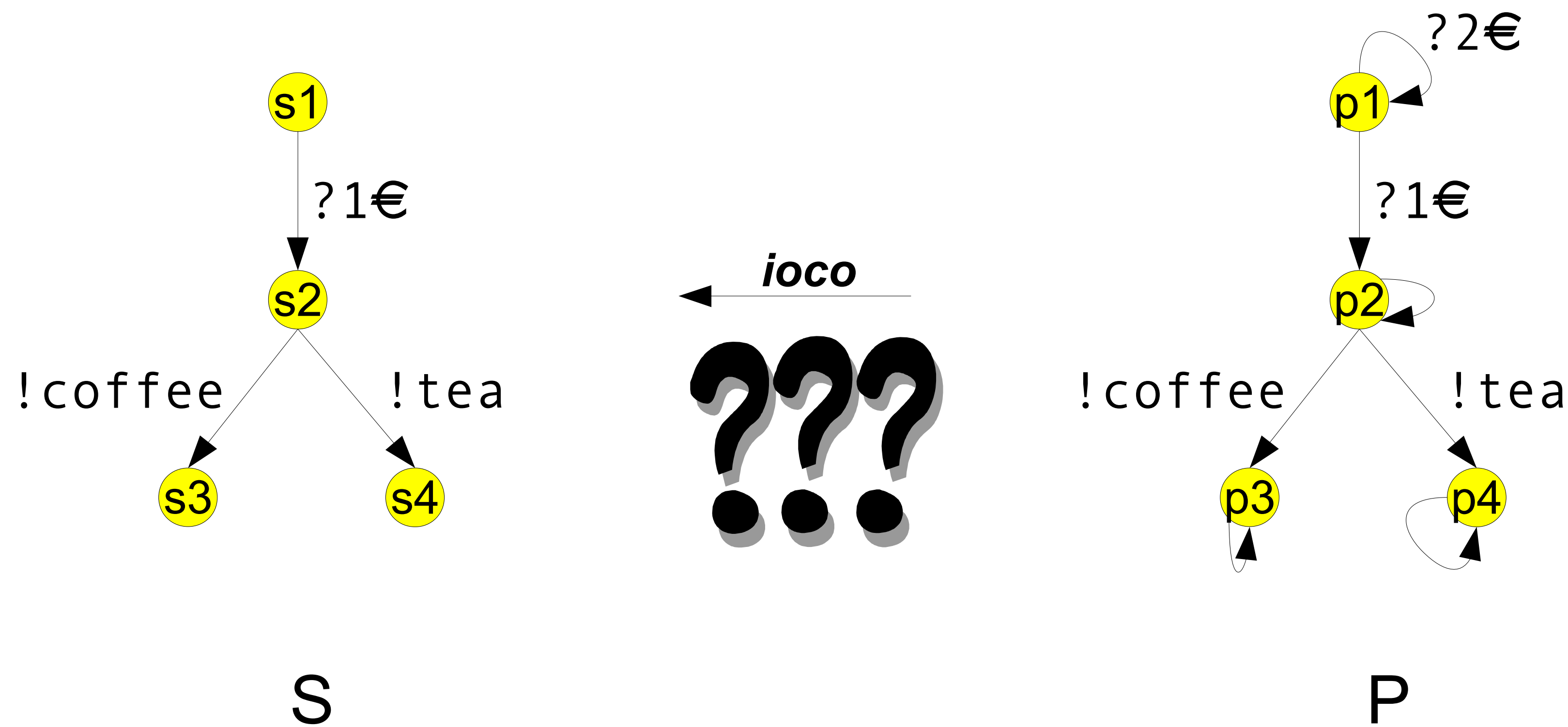
$$\forall \sigma \in \mathcal{F} : out(\, p_1 \ \mathbf{after}\ \sigma\,) \subseteq out(\, s_1 \ \mathbf{after}\ \sigma\,)$$



ioco

Definition 6. *Let* $\mathcal{S} = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{S}} \rangle$ *be an IOLTS, and let* $\mathcal{F} \subseteq \Sigma_\delta^*$*. An input-enabled IOLTS* $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{P}} \rangle$ *is* $\mathbf{ioco}_{\mathcal{F}}$*-conform to* $\mathcal{S}$*, denoted by* $\mathcal{P}\ \mathbf{ioco}_{\mathcal{F}}\ \mathcal{S}$*, iff*

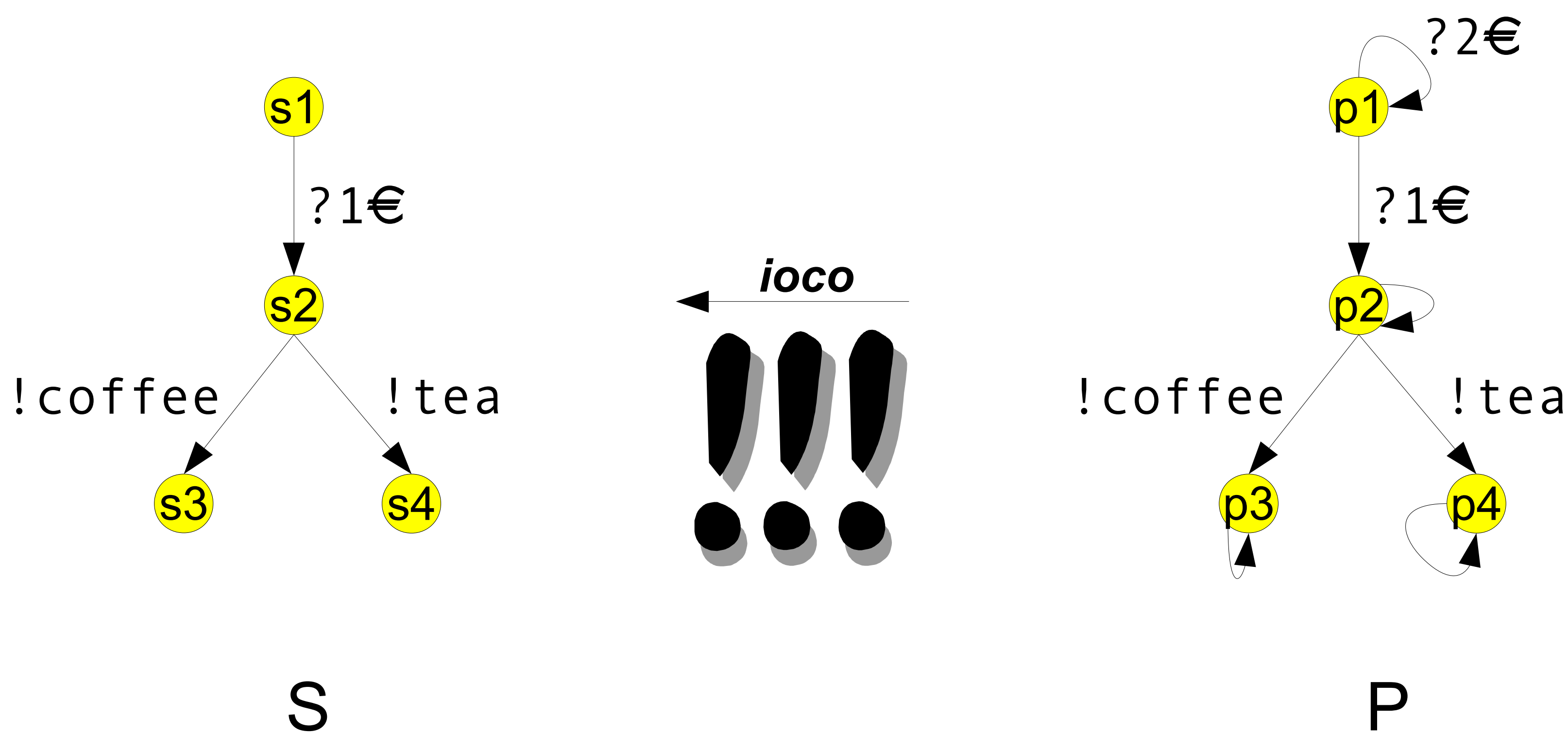
$$\forall \sigma \in \mathcal{F} : out(\, p_1 \ \mathbf{after}\ \sigma\,) \subseteq out(\, s_1 \ \mathbf{after}\ \sigma\,)$$



ioco

Definition 6. *Let* $\mathcal{S} = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{S}} \rangle$ *be an IOLTS, and let* $\mathcal{F} \subseteq \Sigma_\delta^*$. *An input-enabled IOLTS* $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{P}} \rangle$ *is* $\mathbf{ioco}_{\mathcal{F}}$*-conform to* $\mathcal{S}$, *denoted by* $\mathcal{P}\ \mathbf{ioco}_{\mathcal{F}}\ \mathcal{S}$, *iff*

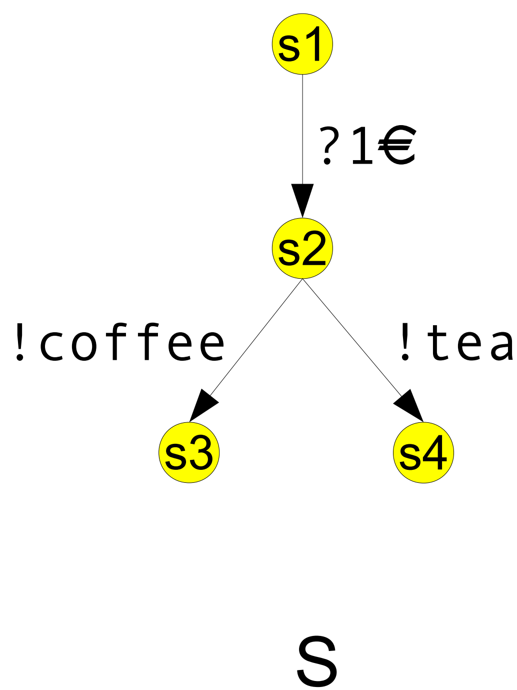
$$\forall \sigma \in \mathcal{F} : out(\, p_1 \ \mathbf{after}\ \sigma\,) \subseteq out(\, s_1 \ \mathbf{after}\ \sigma\,)$$

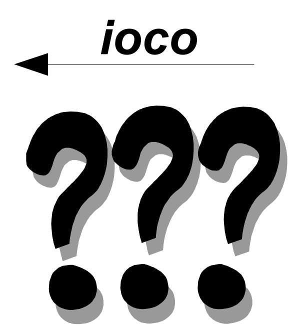


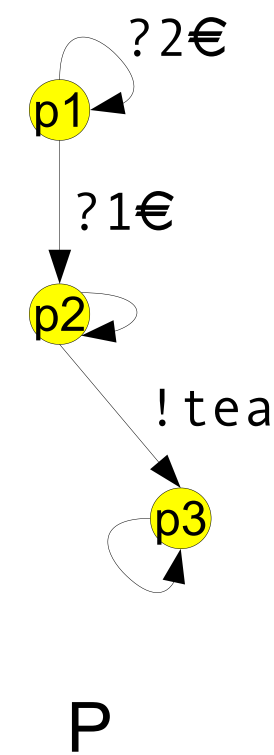
ioco

Definition 6. *Let $\mathcal{S} = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{S}} \rangle$ be an IOLTS, and let $\mathcal{F} \subseteq \Sigma_\delta^*$. An input-enabled IOLTS $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{P}} \rangle$ is* $\mathbf{ioco}_{\mathcal{F}}$*-conform to $\mathcal{S}$, denoted by $\mathcal{P}$* $\mathbf{ioco}_{\mathcal{F}}$ *$\mathcal{S}$, iff*

$$\forall \sigma \in \mathcal{F} : out(\, p_1 \ \mathbf{after}\ \sigma\,) \subseteq out(\, s_1 \ \mathbf{after}\ \sigma\,)$$



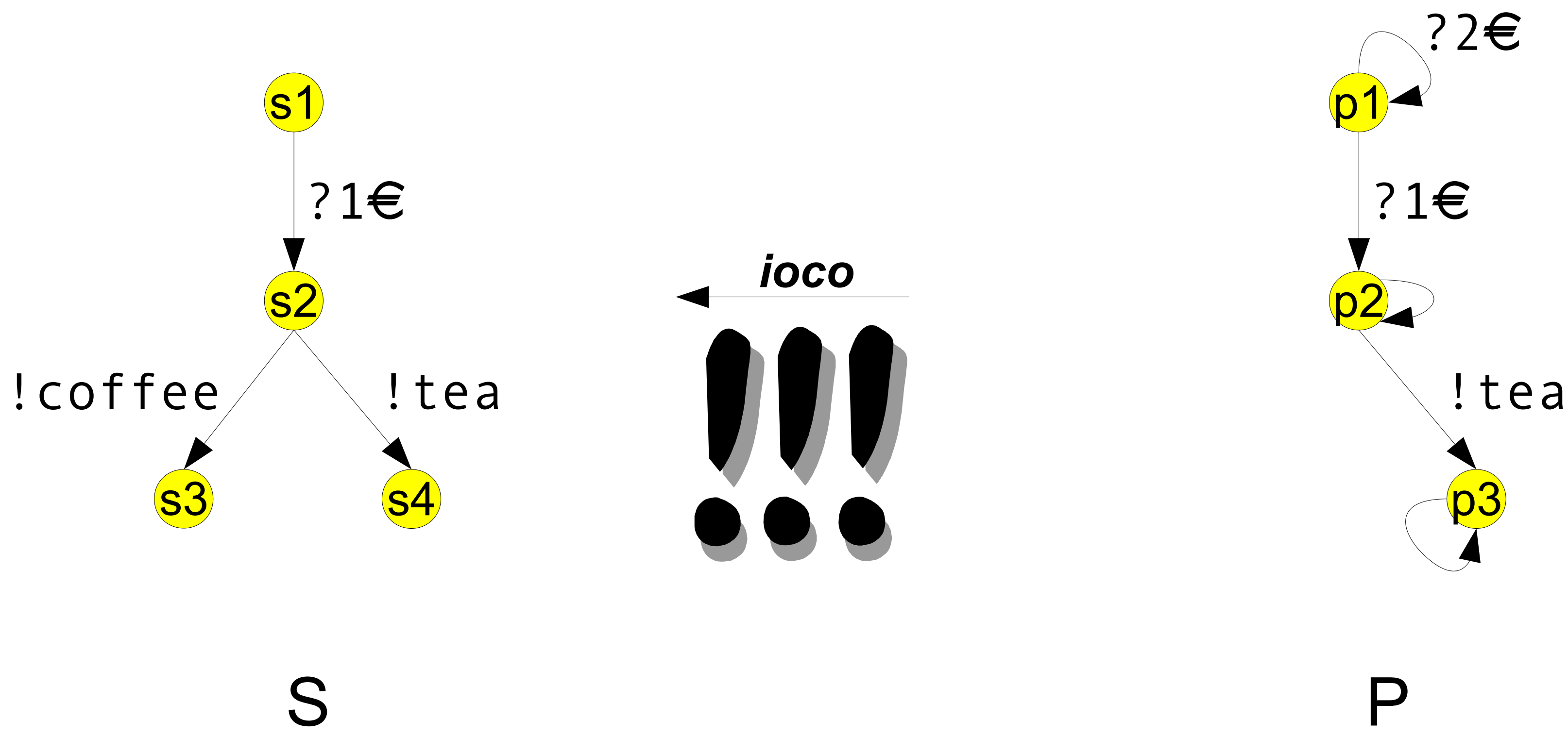




ioco

Definition 6. *Let* $\mathcal{S} = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{S}} \rangle$ *be an IOLTS, and let* $\mathcal{F} \subseteq \Sigma_\delta^*$. *An input-enabled IOLTS* $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{P}} \rangle$ *is* $\mathbf{ioco}_{\mathcal{F}}$*-conform to* $\mathcal{S}$, *denoted by* $\mathcal{P}\ \mathbf{ioco}_{\mathcal{F}}\ \mathcal{S}$, *iff*

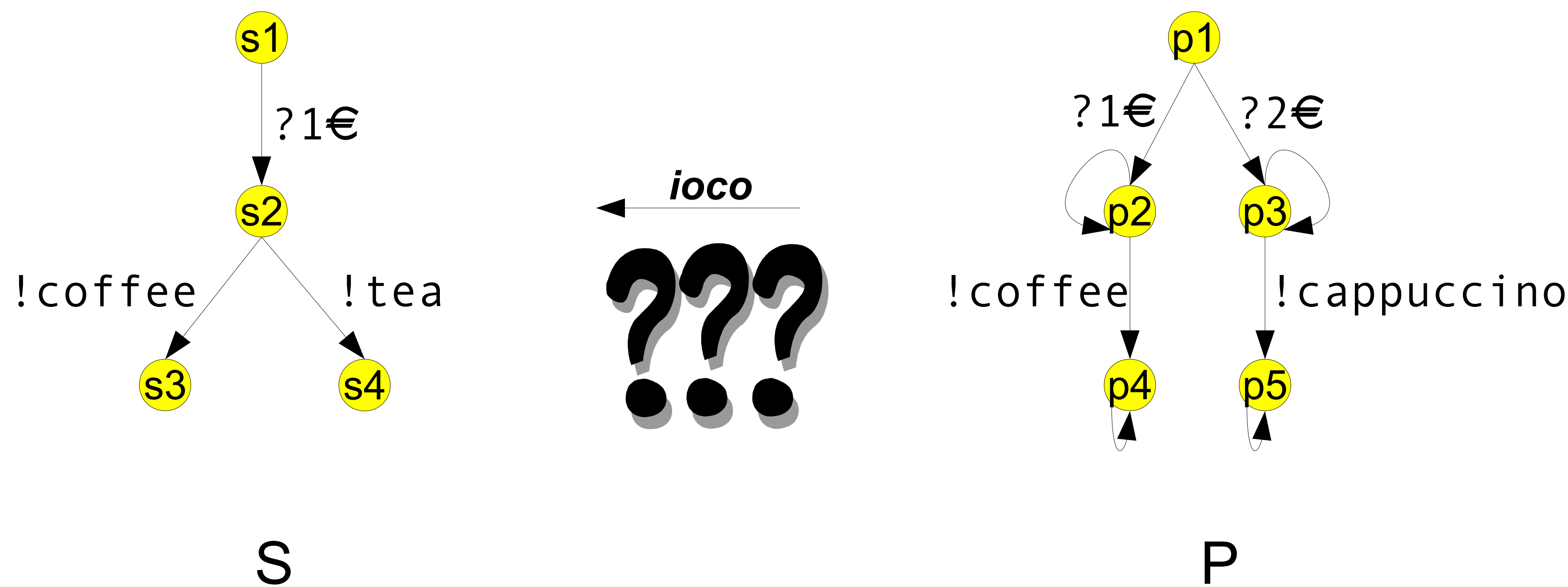
$$\forall \sigma \in \mathcal{F} : out(\, p_1 \textbf{ after } \sigma \,) \subseteq out(\, s_1 \textbf{ after } \sigma \,)$$



ioco

Definition 6. *Let* $\mathcal{S} = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{S}} \rangle$ *be an IOLTS, and let* $\mathcal{F} \subseteq \Sigma_\delta^*$. *An input-enabled IOLTS* $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{P}} \rangle$ *is* $\textbf{ioco}_{\mathcal{F}}$*-conform to* $\mathcal{S}$*, denoted by* $\mathcal{P}\ \textbf{ioco}_{\mathcal{F}}\ \mathcal{S}$*, iff*

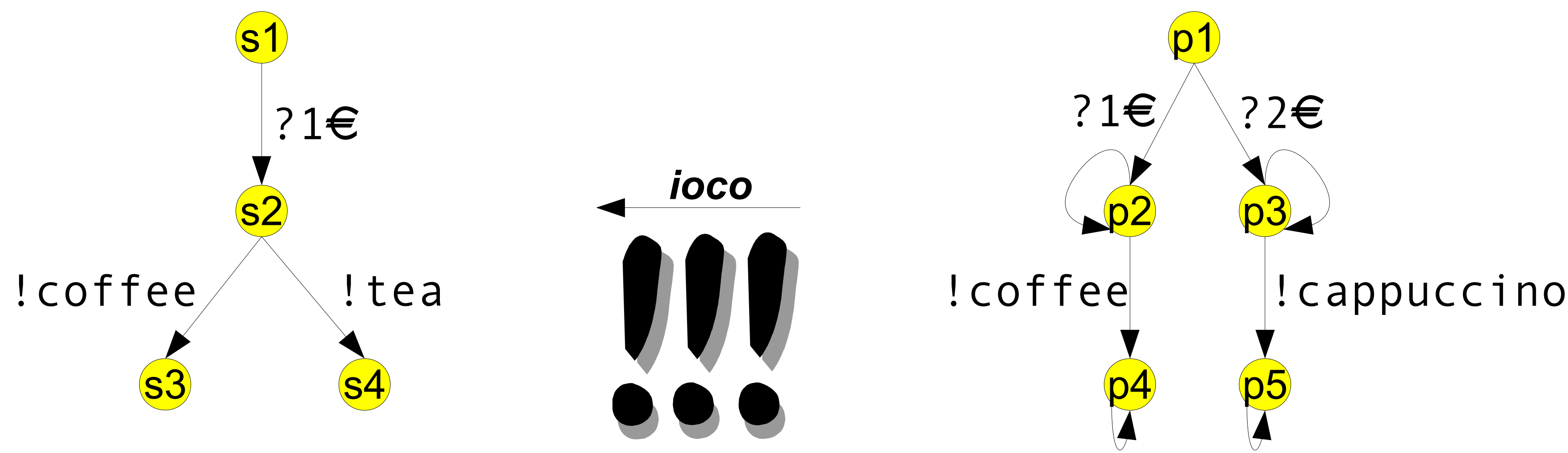
$$\forall \sigma \in \mathcal{F} : out(\, p_1\ \textbf{after}\ \sigma\,) \subseteq out(\, s_1\ \textbf{after}\ \sigma\,)$$



ioco

Definition 6. *Let* $\mathcal{S} = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{S}} \rangle$ *be an IOLTS, and let* $\mathcal{F} \subseteq \Sigma_\delta^*$*. An input-enabled IOLTS* $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{P}} \rangle$ *is* $\mathbf{ioco}_{\mathcal{F}}$*-conform to* $\mathcal{S}$*, denoted by* $\mathcal{P}\ \mathbf{ioco}_{\mathcal{F}}\ \mathcal{S}$*, iff*

$$\forall \sigma \in \mathcal{F} : out(\, p_1 \textbf{ after } \sigma \,) \subseteq out(\, s_1 \textbf{ after } \sigma \,)$$



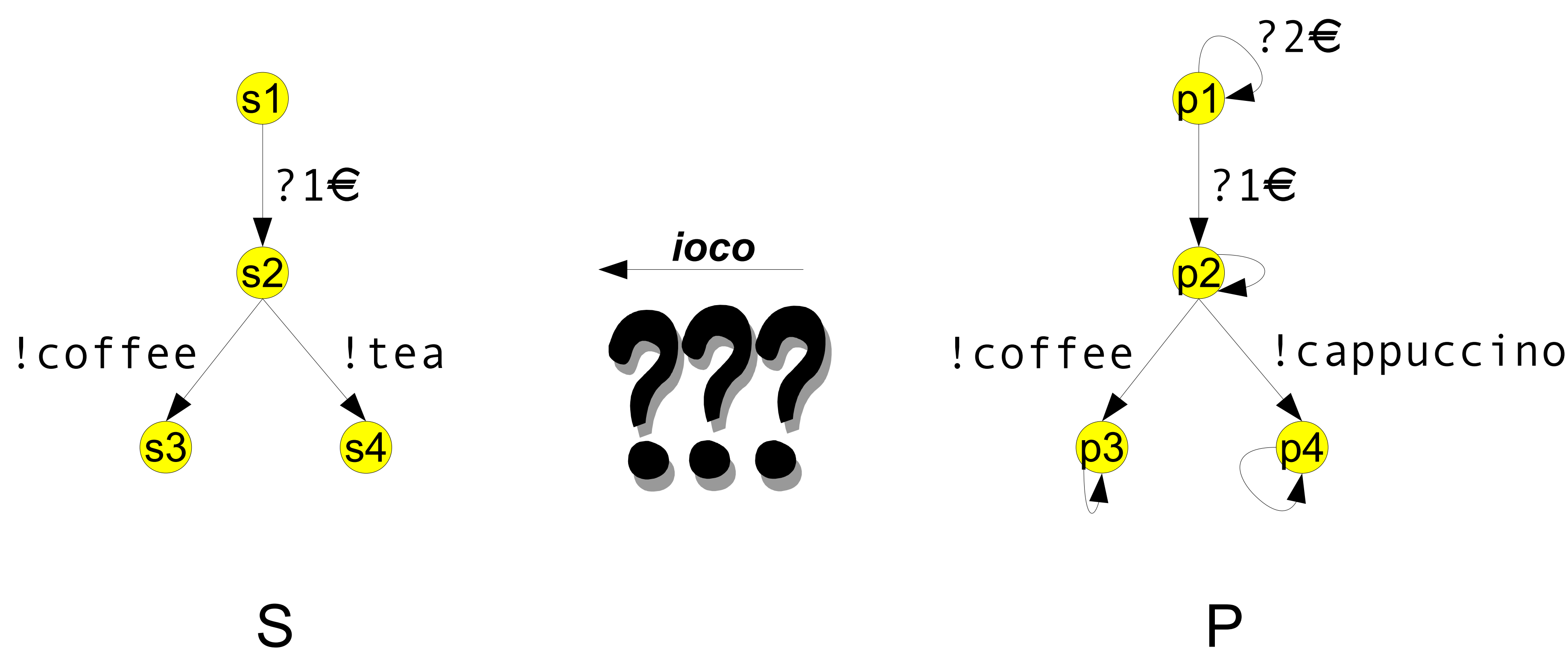
S

P

ioco

Definition 6. *Let* $\mathcal{S} = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{S}} \rangle$ *be an IOLTS, and let* $\mathcal{F} \subseteq \Sigma_\delta^*$. *An input-enabled IOLTS* $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{P}} \rangle$ *is* $\mathbf{ioco}_{\mathcal{F}}$*-conform to* $\mathcal{S}$*, denoted by* $\mathcal{P}\ \mathbf{ioco}_{\mathcal{F}}\ \mathcal{S}$*, iff*

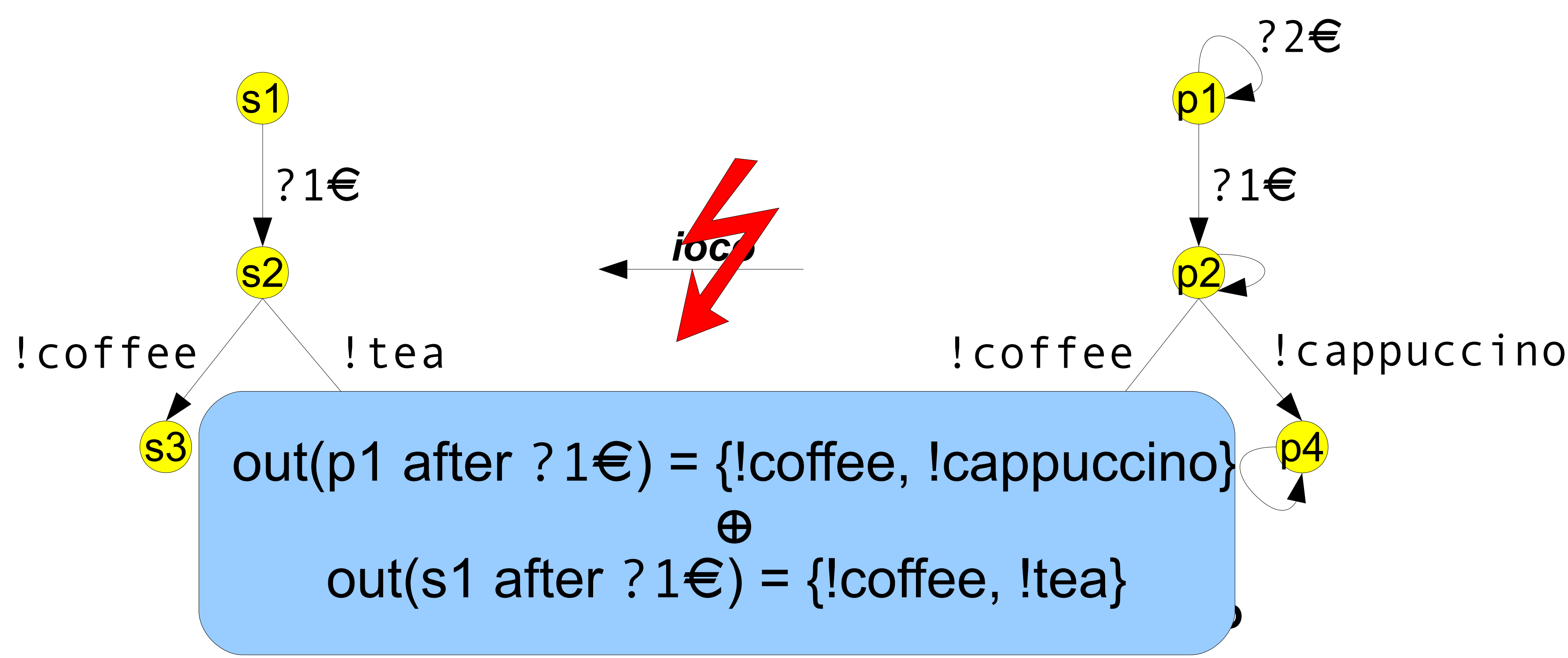
$$\forall \sigma \in \mathcal{F} : out(\, p_1 \ \mathbf{after}\ \sigma\,) \subseteq out(\, s_1 \ \mathbf{after}\ \sigma\,)$$



ioco

Definition 6. *Let* $\mathcal{S} = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{S}} \rangle$ *be an IOLTS, and let* $\mathcal{F} \subseteq \Sigma_\delta^*$. *An input-enabled IOLTS* $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{P}} \rangle$ *is* $\mathbf{ioco}_{\mathcal{F}}$*-conform to* $\mathcal{S}$, *denoted by* $\mathcal{P}\ \mathbf{ioco}_{\mathcal{F}}\ \mathcal{S}$, *iff*

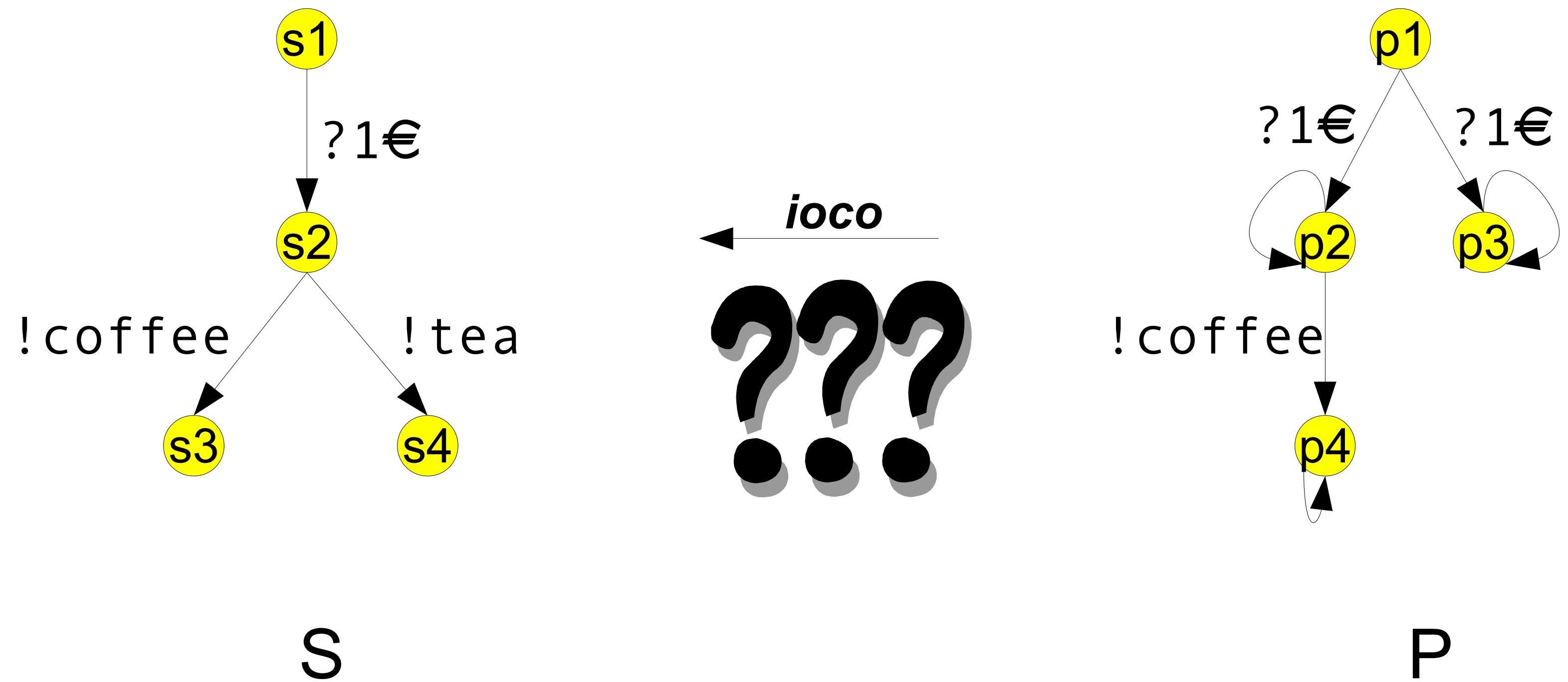
$$\forall \sigma \in \mathcal{F} : out(\, p_1 \ \mathbf{after}\ \sigma \,) \subseteq out(\, s_1 \ \mathbf{after}\ \sigma \,)$$



ioco

Definition 6. *Let* $\mathcal{S} = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_\mathcal{S} \rangle$ *be an IOLTS, and let* $\mathcal{F} \subseteq \Sigma_\delta^*$. *An input-enabled IOLTS* $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_\mathcal{P} \rangle$ *is* $\mathbf{ioco}_\mathcal{F}$*-conform to* $\mathcal{S}$, *denoted by* $\mathcal{P}\ \mathbf{ioco}_\mathcal{F}\ \mathcal{S}$, *iff*

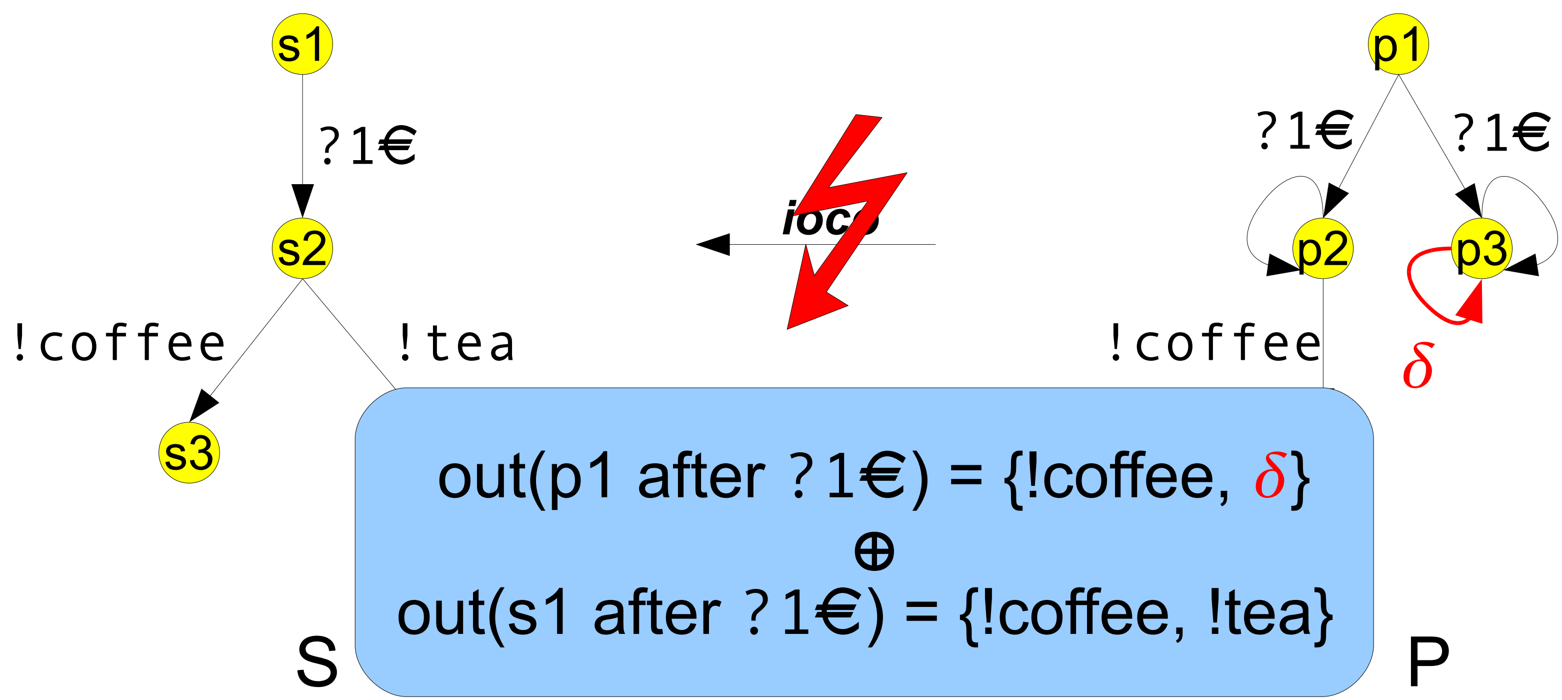
$$\forall \sigma \in \mathcal{F} : out(\, p_1 \ \mathbf{after}\ \sigma\,) \subseteq out(\, s_1 \ \mathbf{after}\ \sigma\,)$$



ioco

Definition 6. *Let $\mathcal{S} = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_\mathcal{S} \rangle$ be an IOLTS, and let $\mathcal{F} \subseteq \Sigma_\delta^*$. An input-enabled IOLTS $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_\mathcal{P} \rangle$ is* $\mathbf{ioco}_\mathcal{F}$*-conform to* $\mathcal{S}$*, denoted by* $\mathcal{P}\ \mathbf{ioco}_\mathcal{F}\ \mathcal{S}$*, iff*

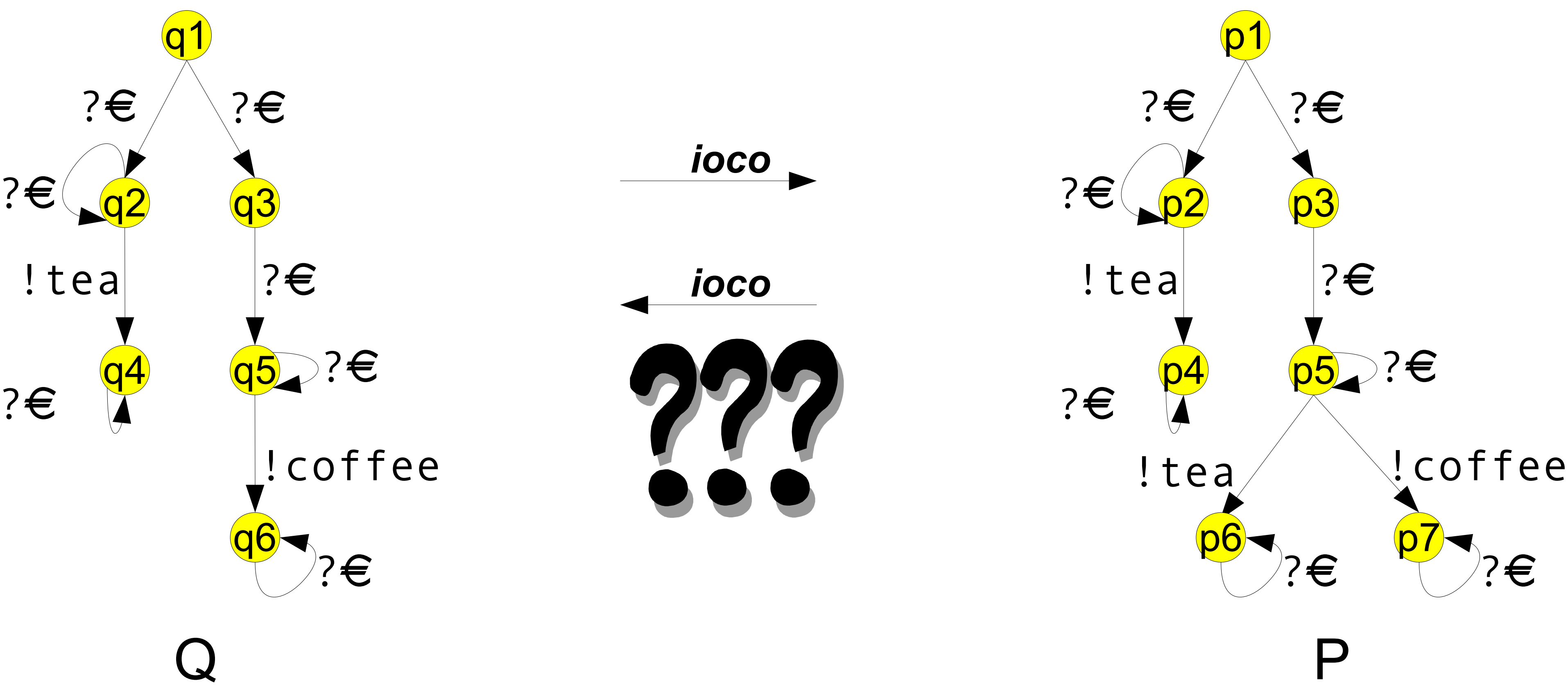
$$\forall \sigma \in \mathcal{F} : out(\, p_1 \ \mathbf{after}\ \sigma\,) \subseteq out(\, s_1 \ \mathbf{after}\ \sigma\,)$$



ioco

Definition 6. *Let* $\mathcal{S} = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{S}} \rangle$ *be an IOLTS, and let* $\mathcal{F} \subseteq \Sigma_\delta^*$. *An input-enabled IOLTS* $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{P}} \rangle$ *is* $\mathbf{ioco}_{\mathcal{F}}$*-conform to* $\mathcal{S}$*, denoted by* $\mathcal{P}\ \mathbf{ioco}_{\mathcal{F}}\ \mathcal{S}$*, iff*

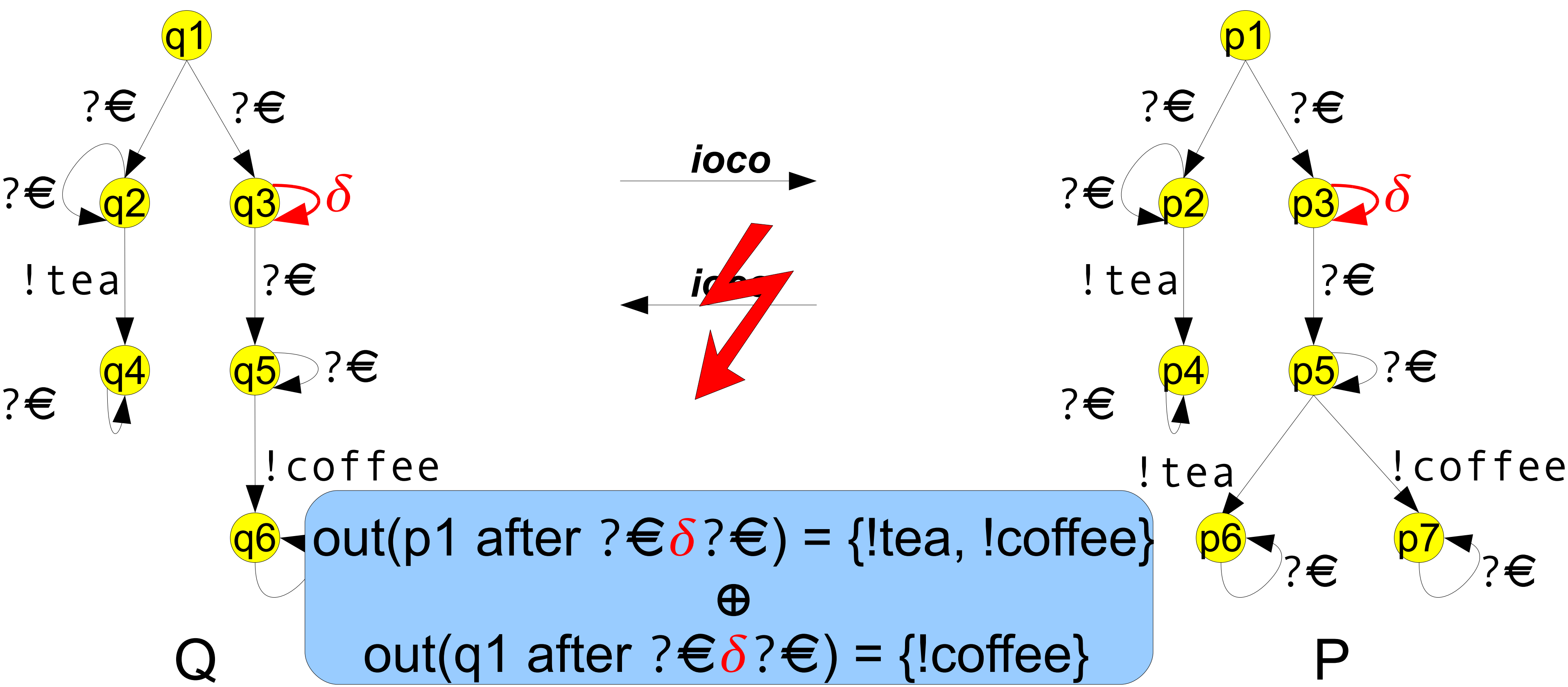
$$\forall \sigma \in \mathcal{F} : out(\, p_1 \ \mathbf{after}\ \sigma\,) \subseteq out(\, s_1 \ \mathbf{after}\ \sigma\,)$$



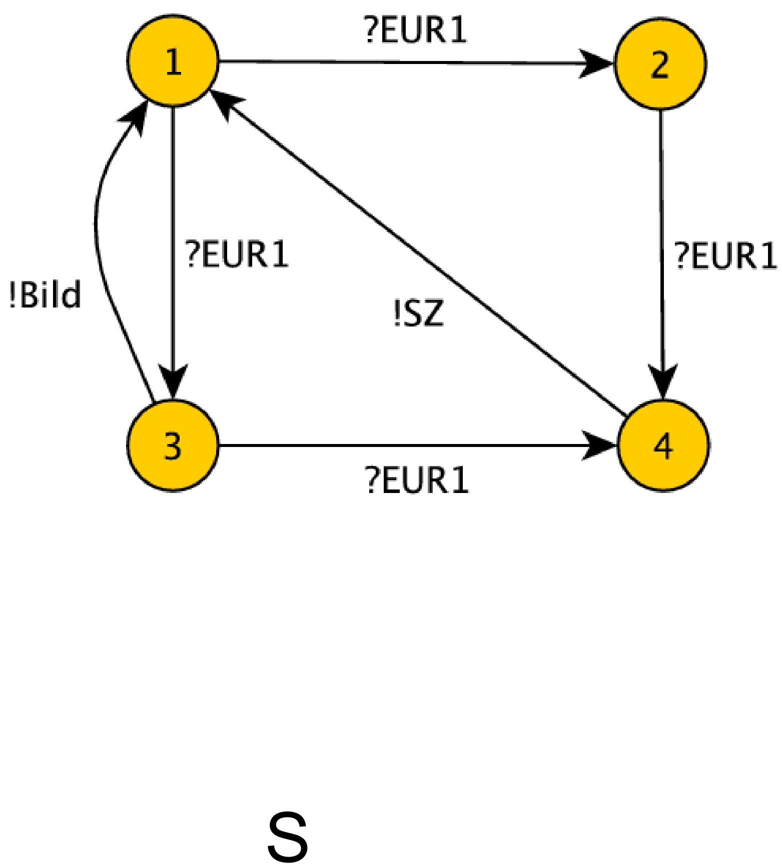
ioco

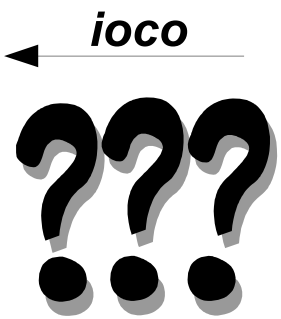
Definition 6. *Let* $\mathcal{S} = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{S}} \rangle$ *be an IOLTS, and let* $\mathcal{F} \subseteq \Sigma_\delta^*$. *An input-enabled IOLTS* $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{P}} \rangle$ *is* $\mathbf{ioco}_{\mathcal{F}}$*-conform to* $\mathcal{S}$*, denoted by* $\mathcal{P}\ \mathbf{ioco}_{\mathcal{F}}\ \mathcal{S}$*, iff*

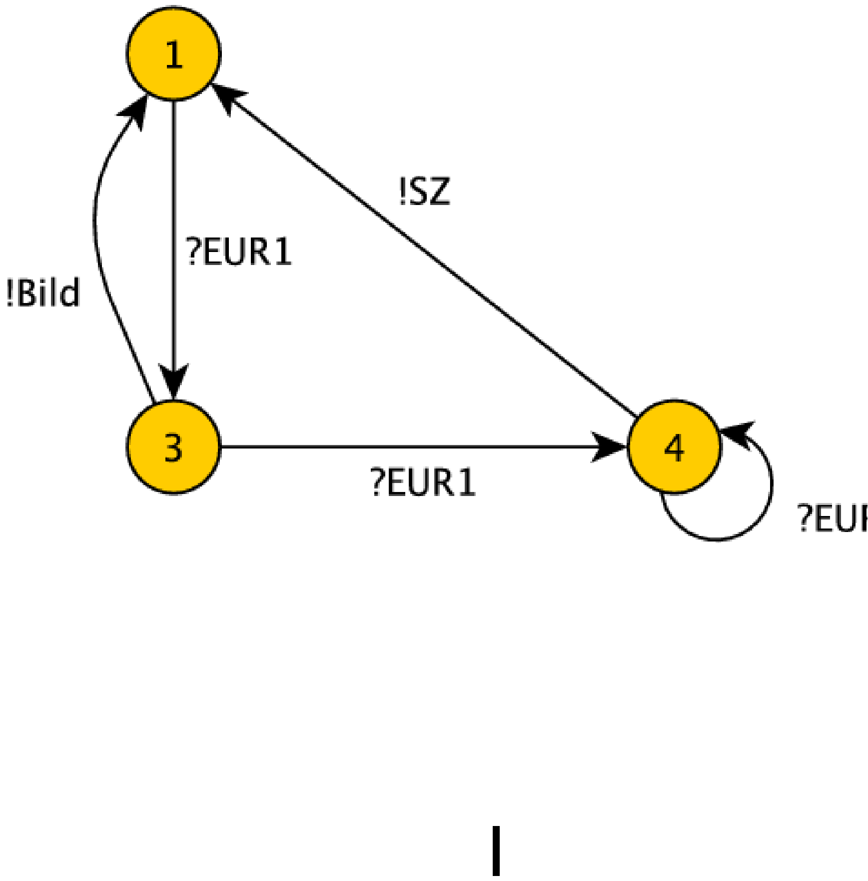
$$\forall \sigma \in \mathcal{F} : out(\, p_1 \ \mathbf{after}\ \sigma \,) \subseteq out(\, s_1 \ \mathbf{after}\ \sigma \,)$$



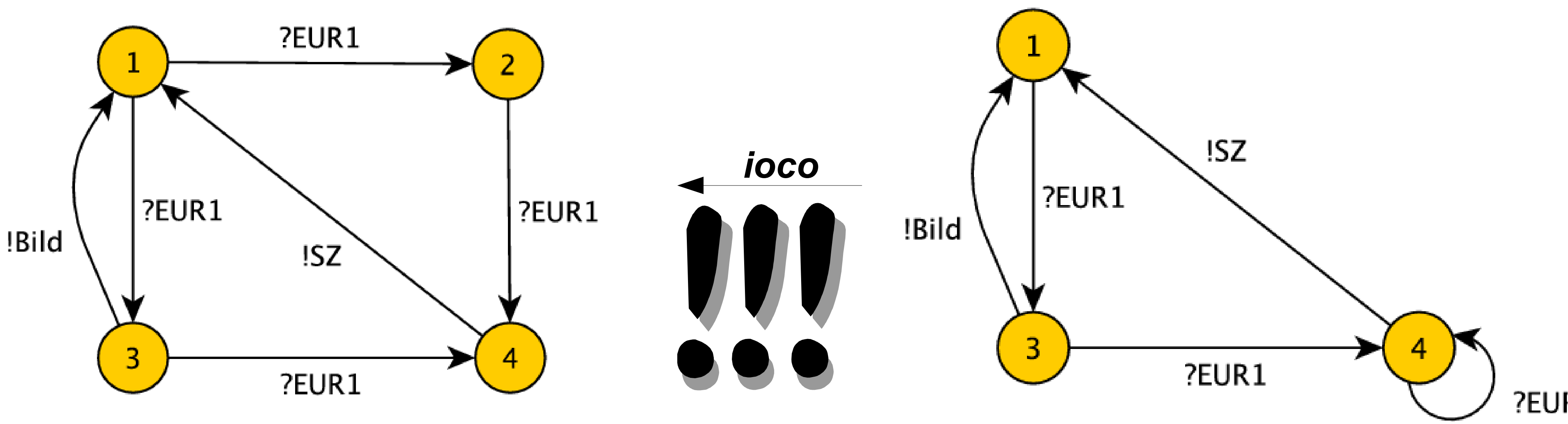
ioco





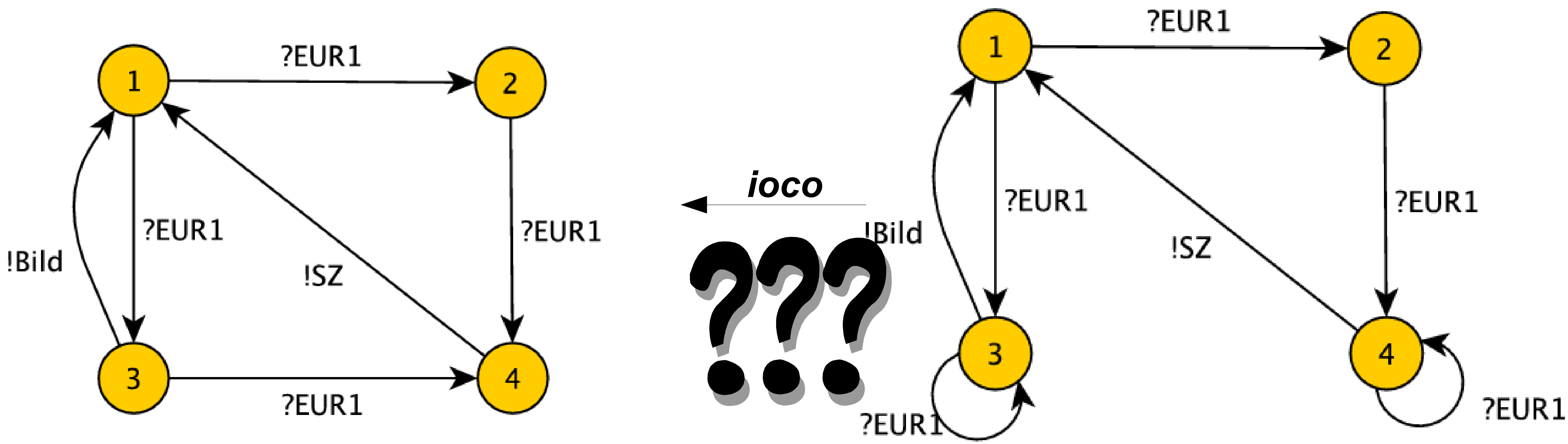


ioco



S I

ioco

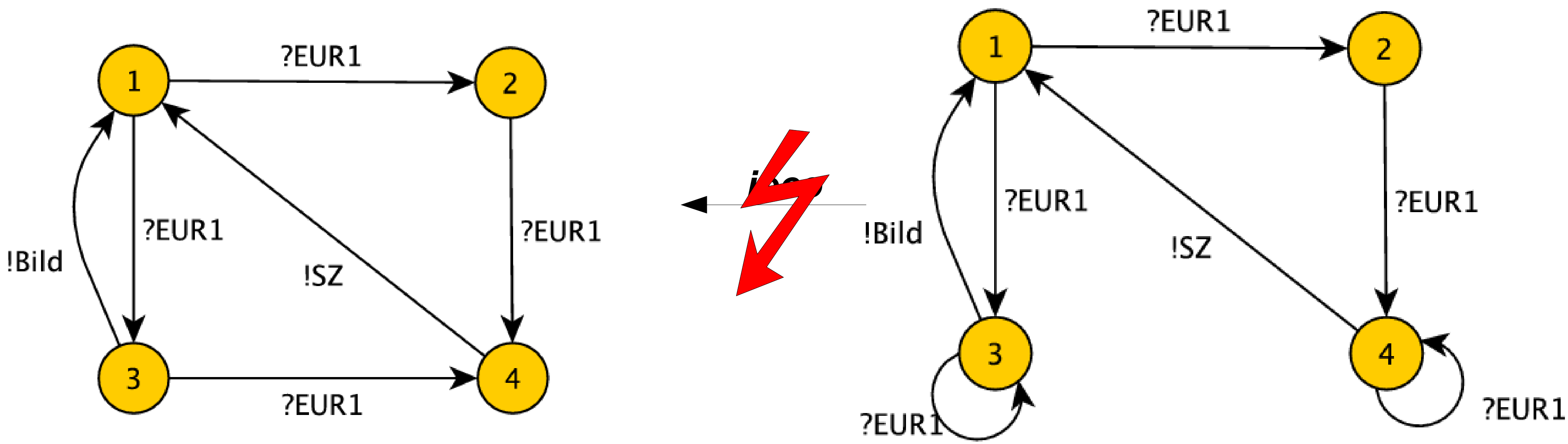


1
2
3
4
?EUR1
?EUR1
?EUR1
?EUR1
!Bild
!SZ
ioco
???
1
2
3
4
?EUR1
?EUR1
?EUR1
?EUR1
?EUR1
!Bild
!SZ

S

I

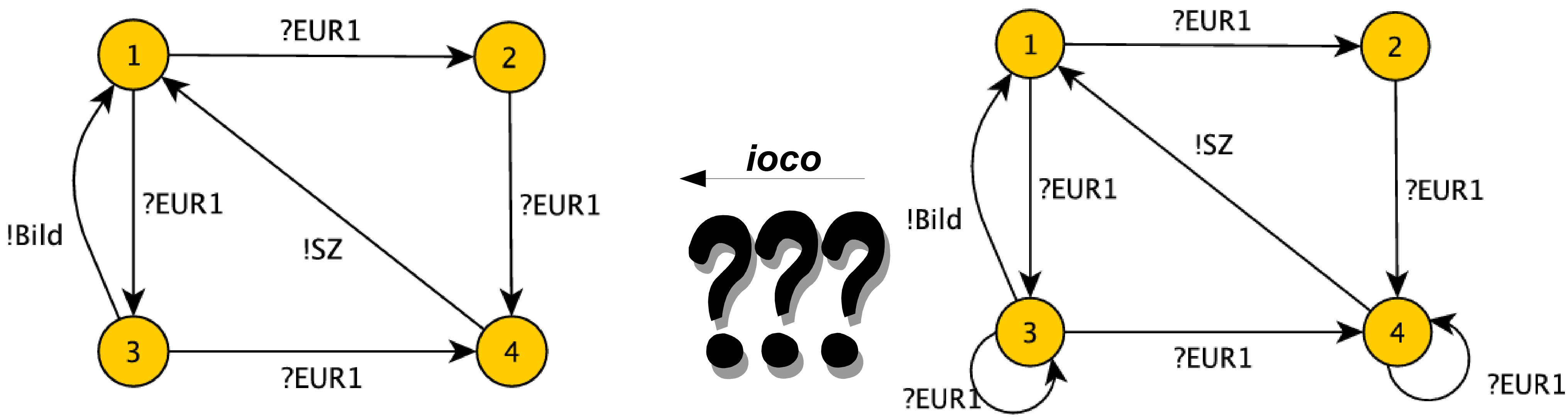
ioco



S

I

ioco

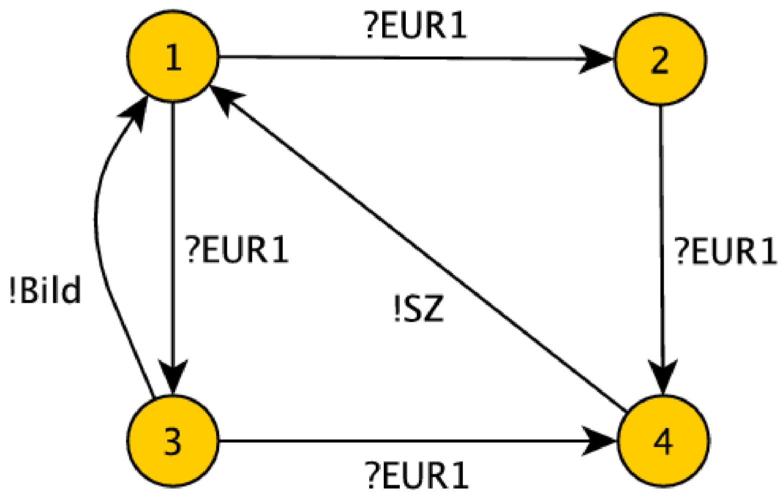


1
2
3
4
?EUR1
?EUR1
?EUR1
?EUR1
!Bild
!SZ
ioco
???
1
2
3
4
?EUR1
?EUR1
?EUR1
?EUR1
?EUR1
?EUR1
!Bild
!SZ

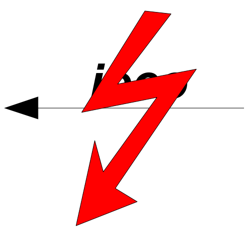
S

I

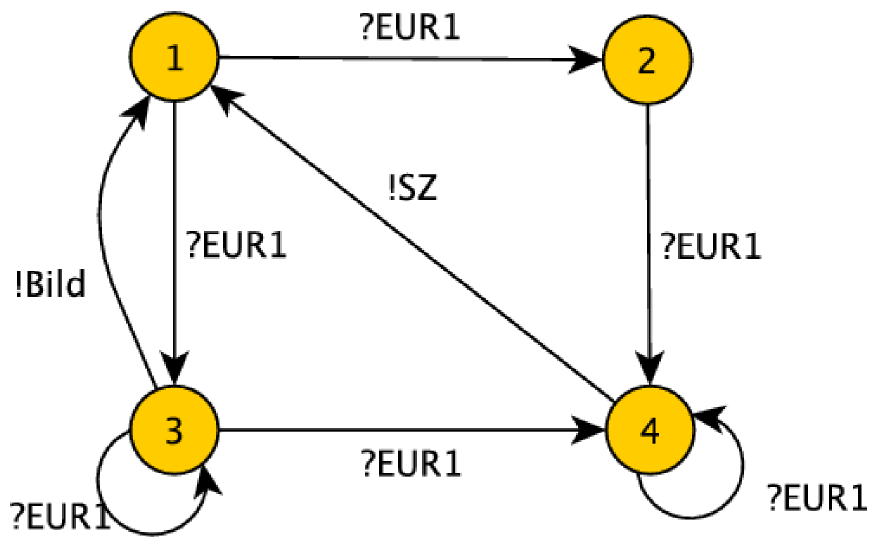
ioco



?EUR1
1
2
3
4
?EUR1
?EUR1
!Bild
!SZ
?EUR1



ioco



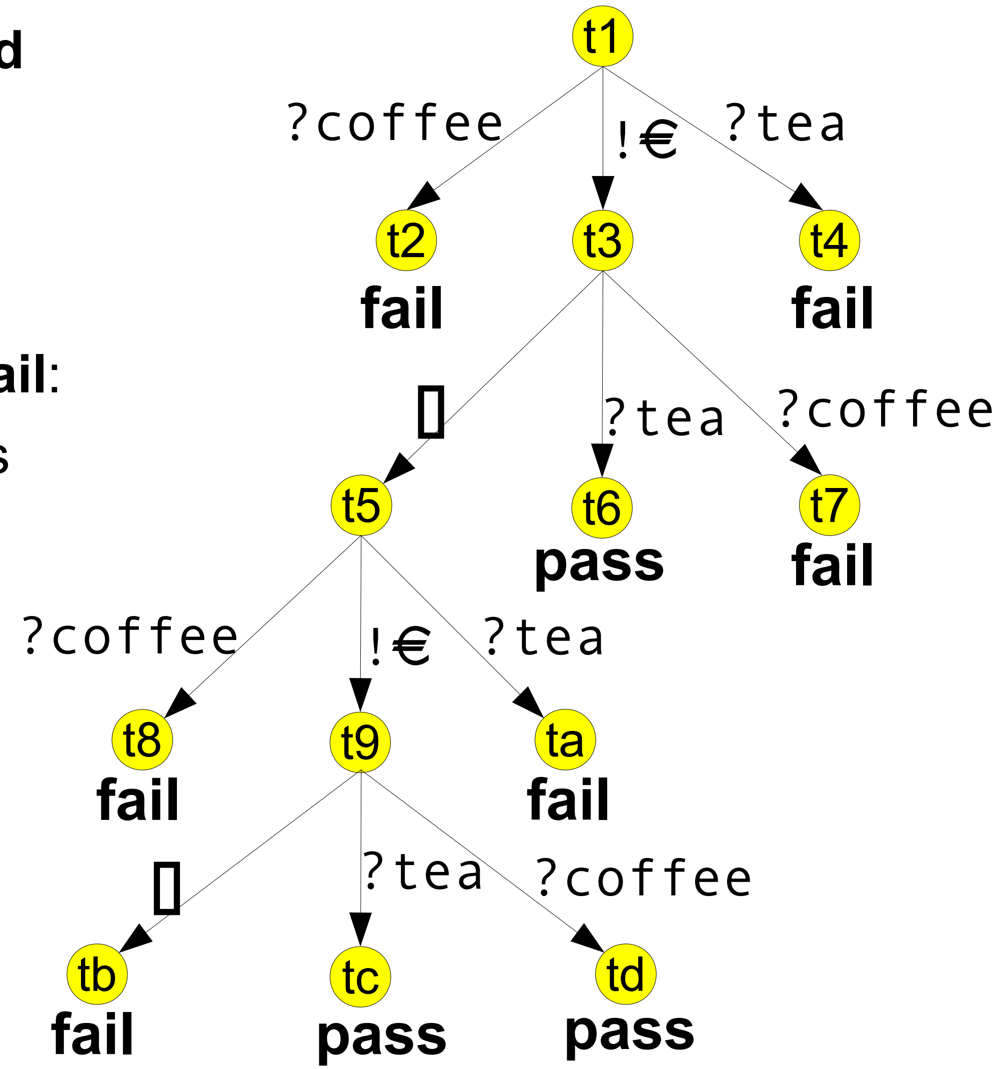
?EUR1
1
2
3
4
?EUR1
?EUR1
!Bild
!SZ
?EUR1
?EUR1
?EUR1

S

I

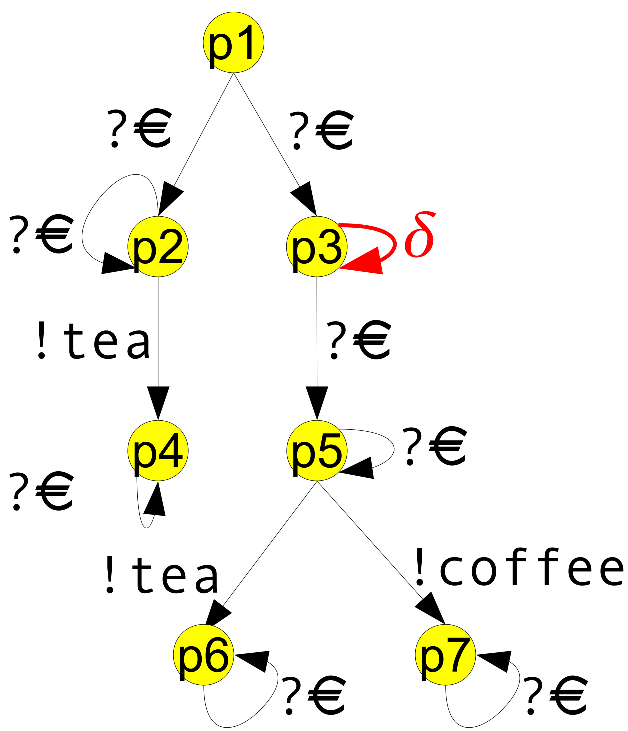
Test Cases

- A **test case** is an IOLTS
  - having a **quiescence label** ▯ (modeling the observation of quiescence)
  - having **inputs and outputs swapped**
  - being **tree-structured**
  - being **finite** and **deterministic**
  - having final states **pass** and **fail**
  - where from each state $\neq$ **pass** and **fail**:
    - either a single output and all inputs
    - or all inputs and θ



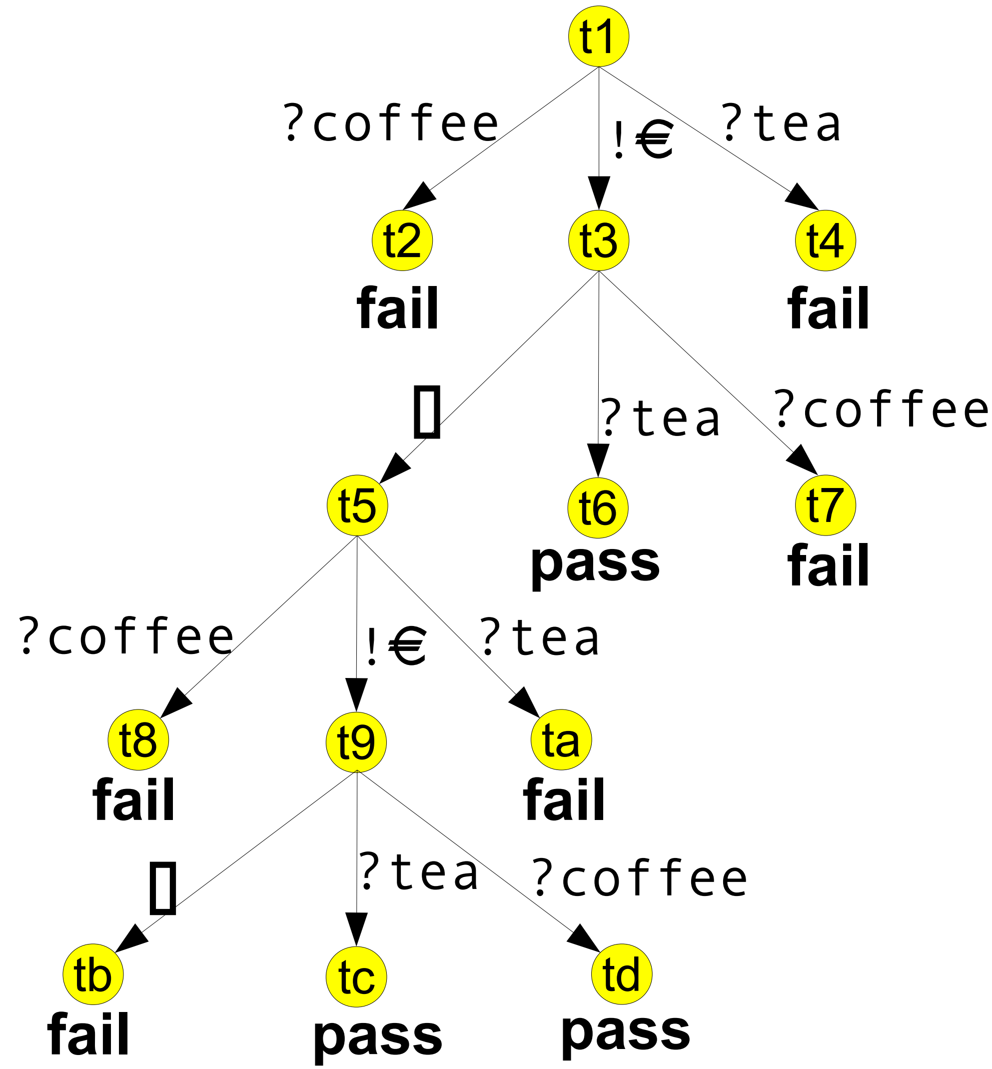
Formal Test Execution

- **Formally executing** a test case means putting it in parallel with the implementation model, leading to a **verdict**.



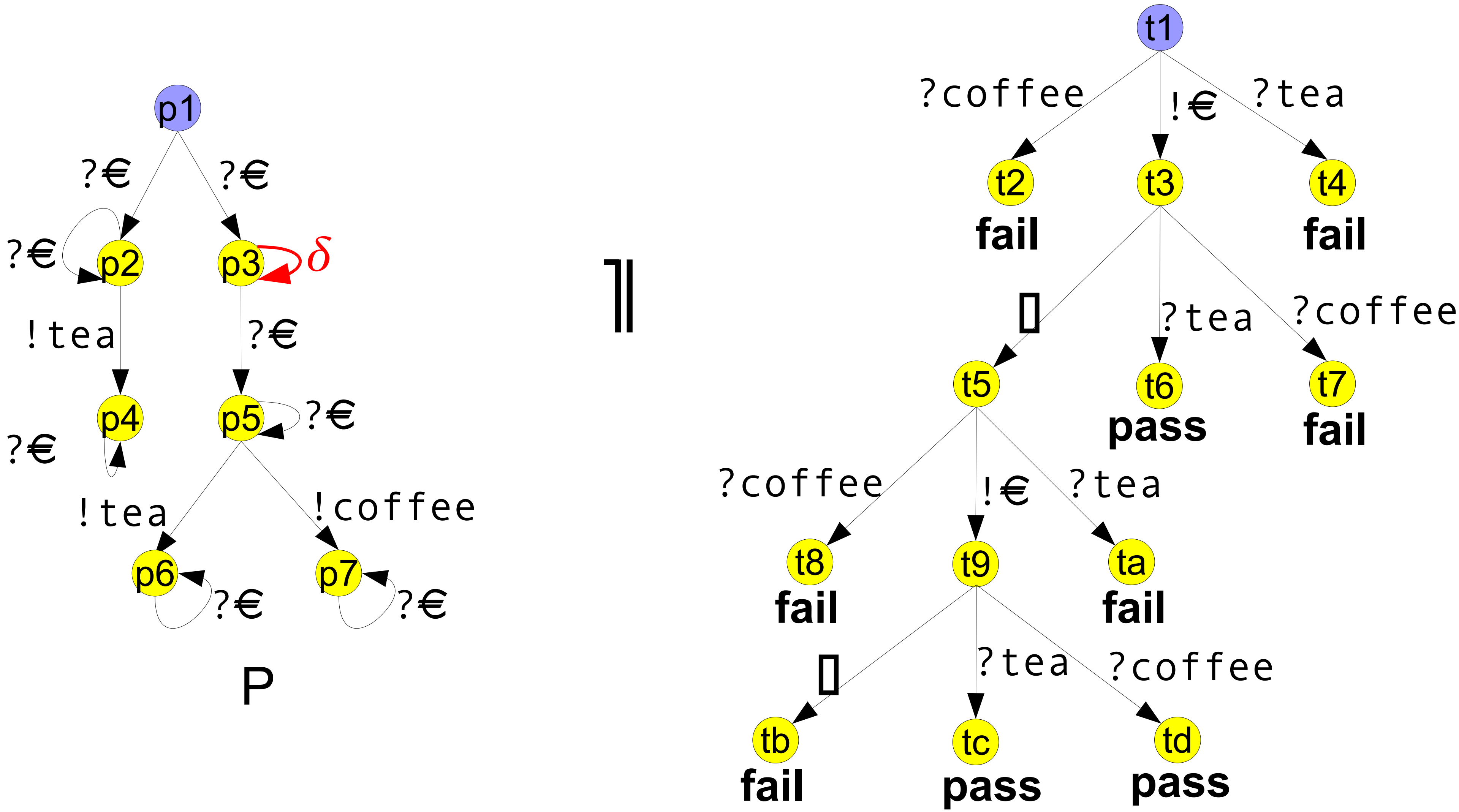
P

‖



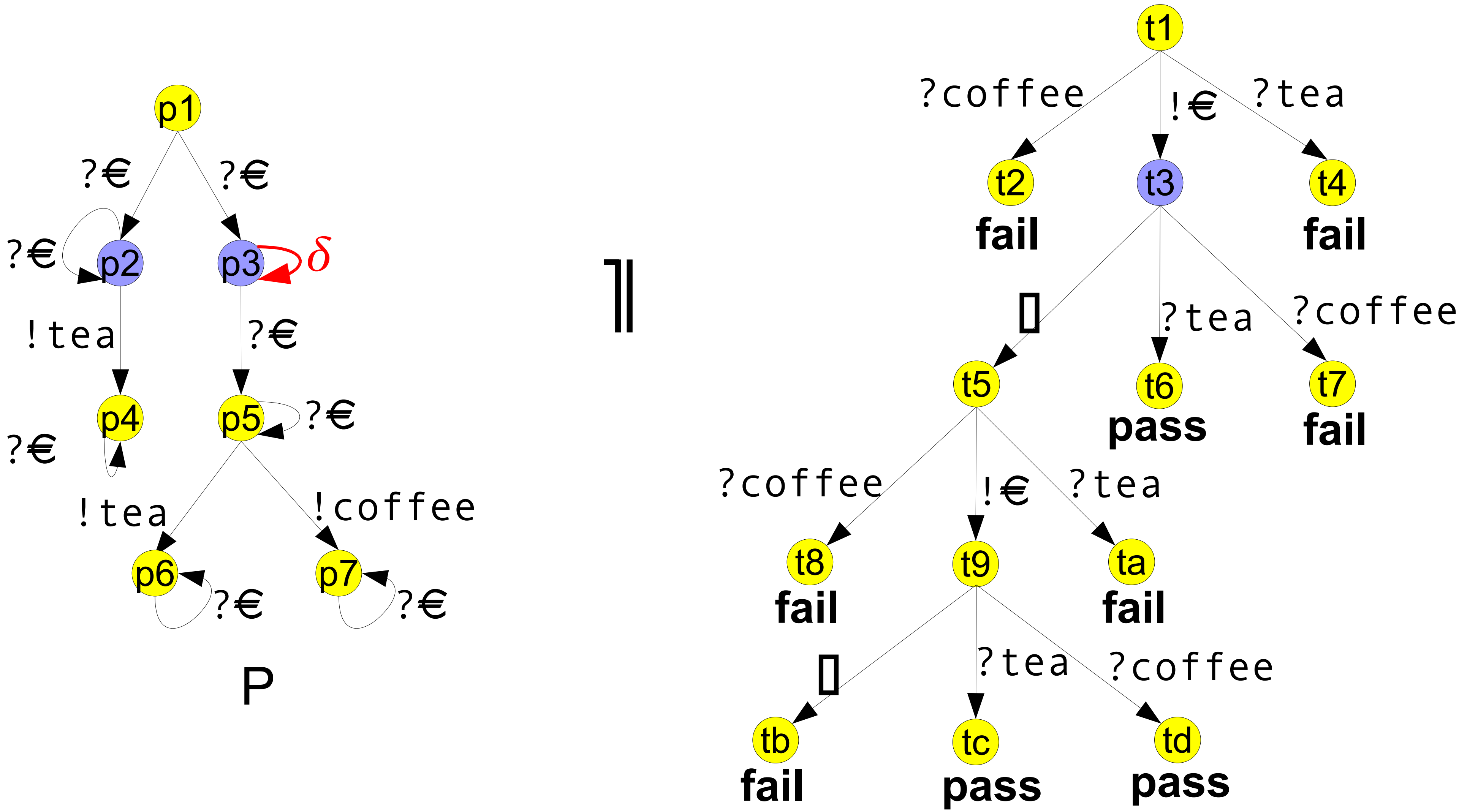
Formal Test Execution

- **Formally executing** a test case means putting it in parallel with the implementation model, leading to a **verdict**.



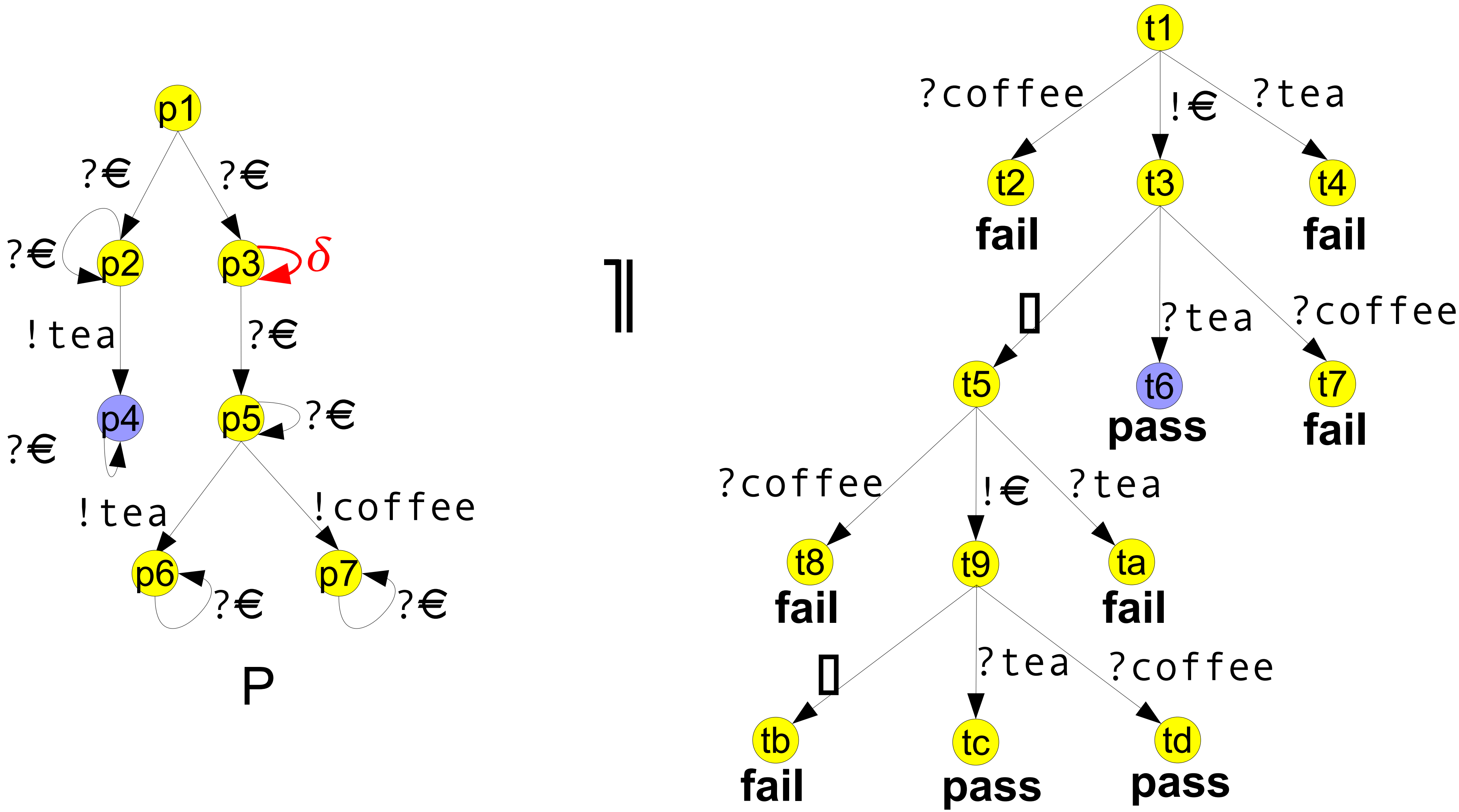
Formal Test Execution

- **Formally executing** a test case means putting it in parallel with the implementation model, leading to a **verdict**.



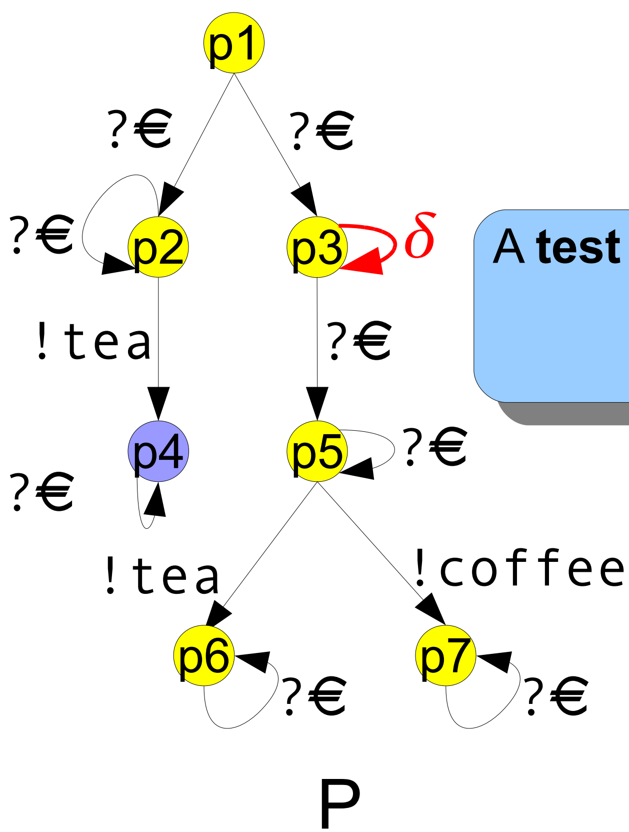
Formal Test Execution

- **Formally executing** a test case means putting it in parallel with the implementation model, leading to a **verdict**.

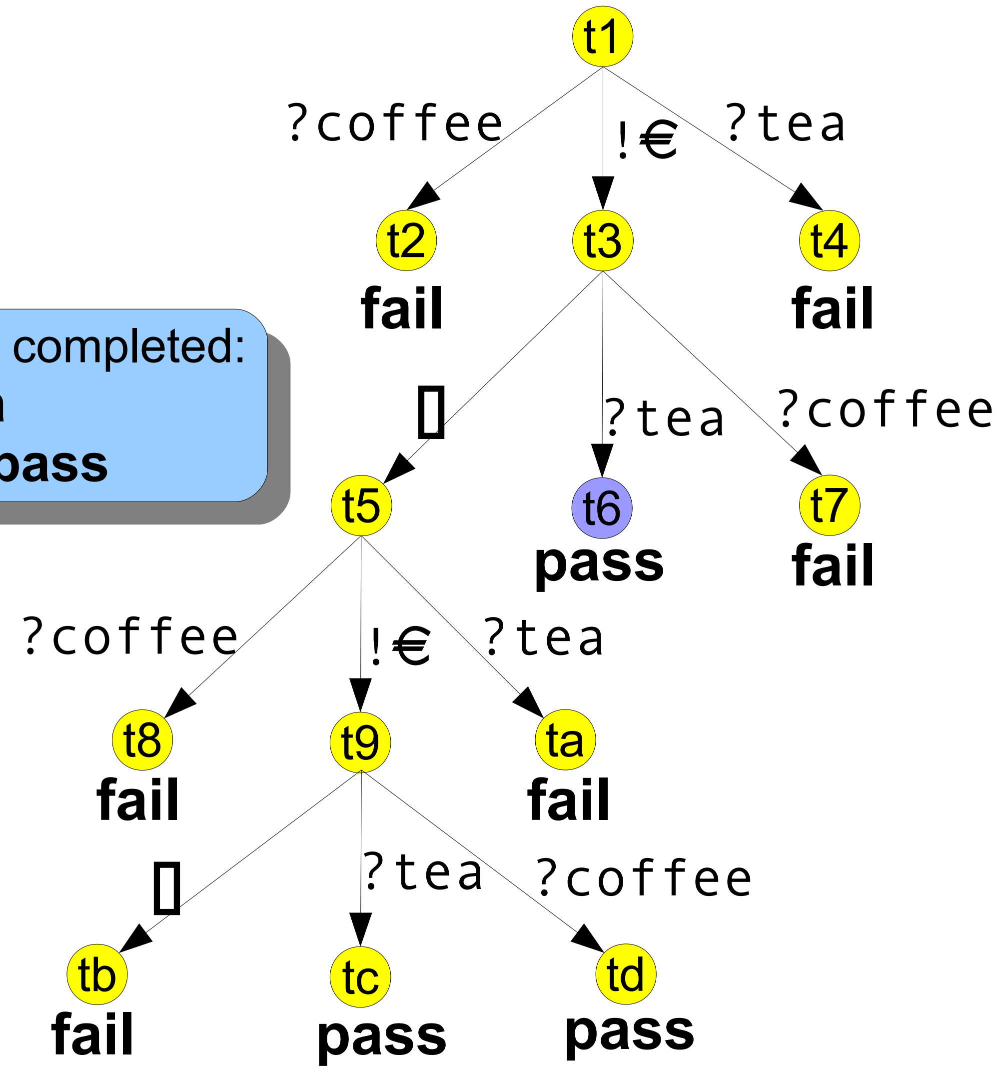


Formal Test Execution

- **Formally executing** a test case means putting it in parallel with the implementation model, leading to a **verdict**.

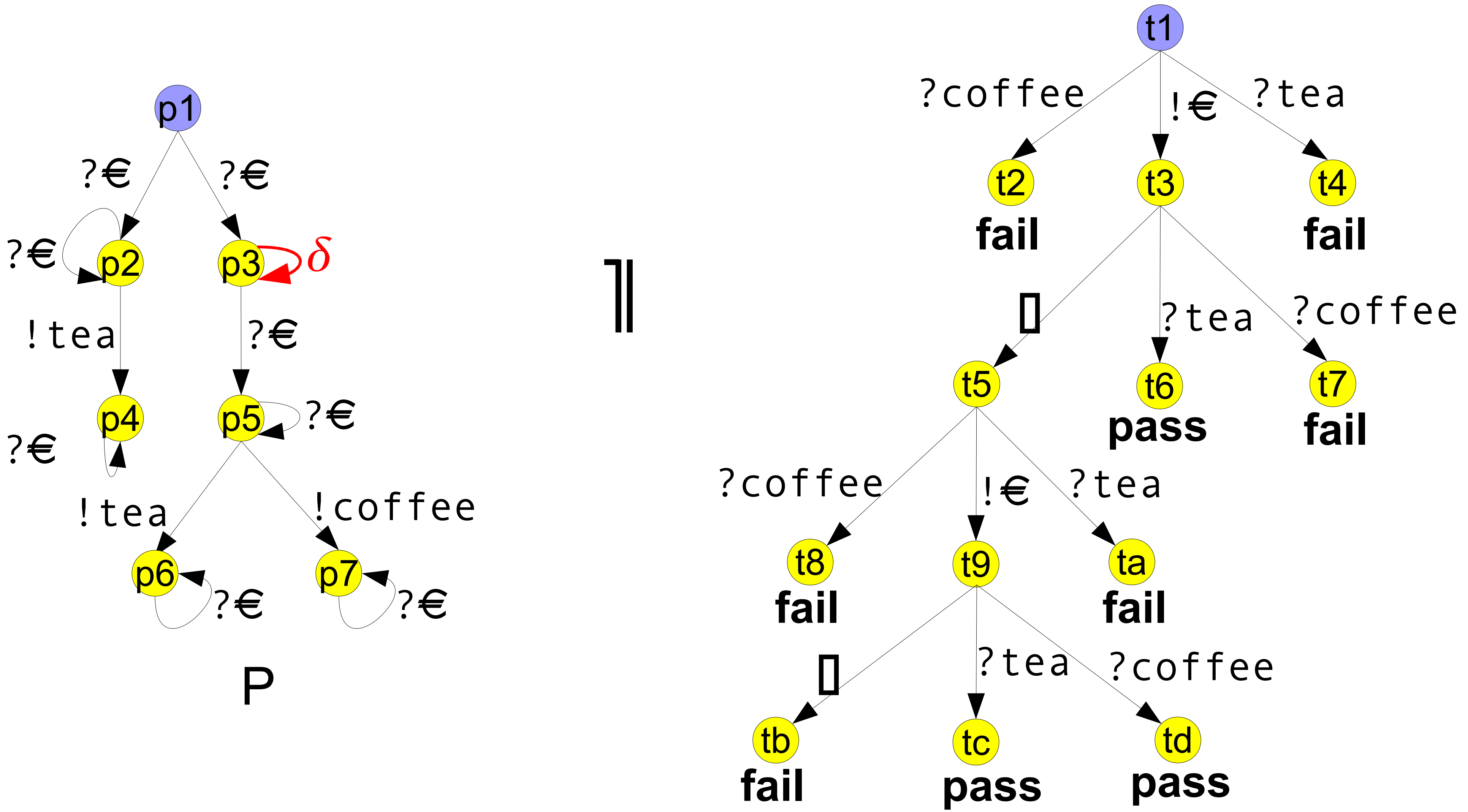


A **test run** has been completed:
!€?tea
with verdict **pass**



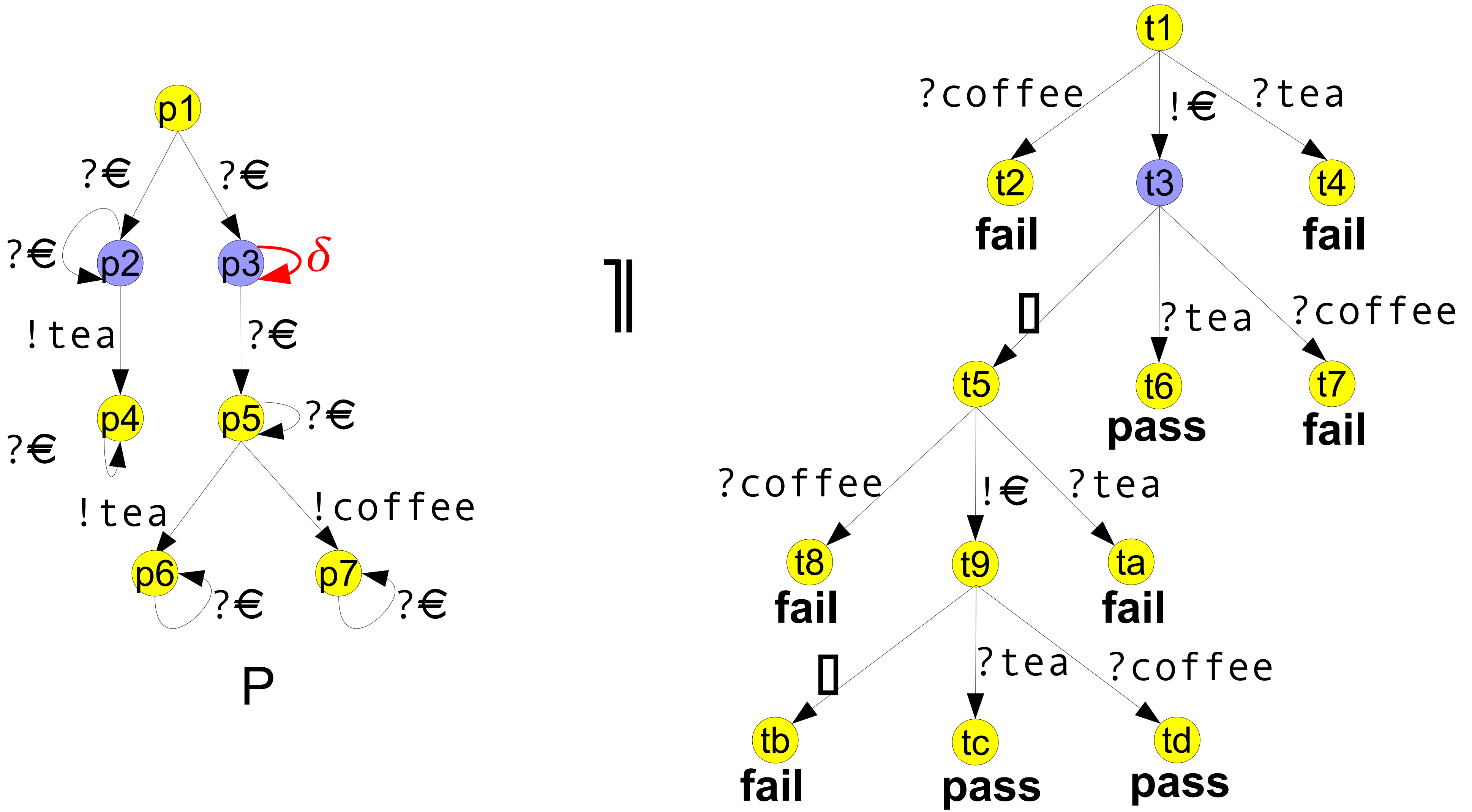
Formal Test Execution

- **Formally executing** a test case means putting it in parallel with the implementation model, leading to a **verdict**.



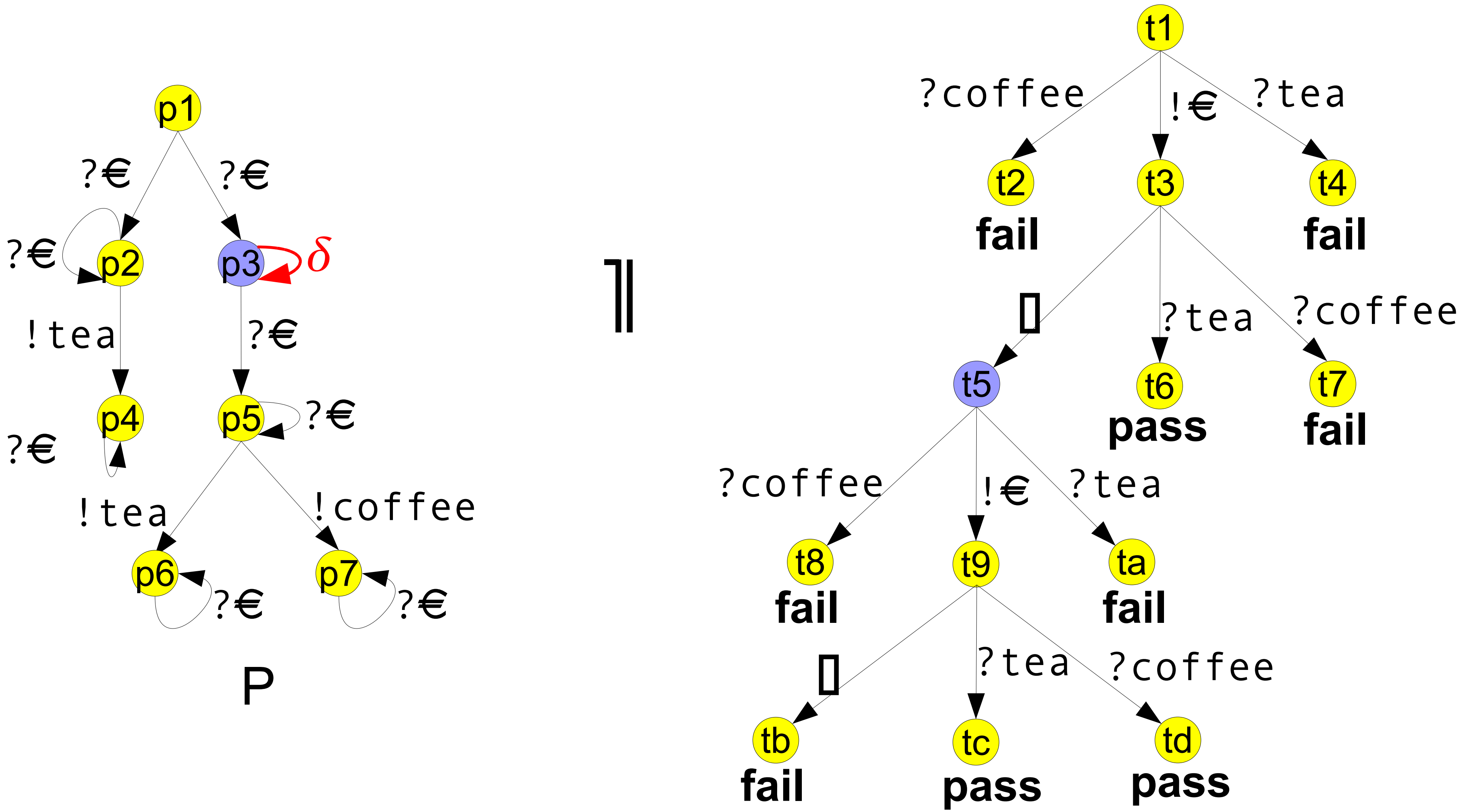
Formal Test Execution

- **Formally executing** a test case means putting it in parallel with the implementation model, leading to a **verdict**.



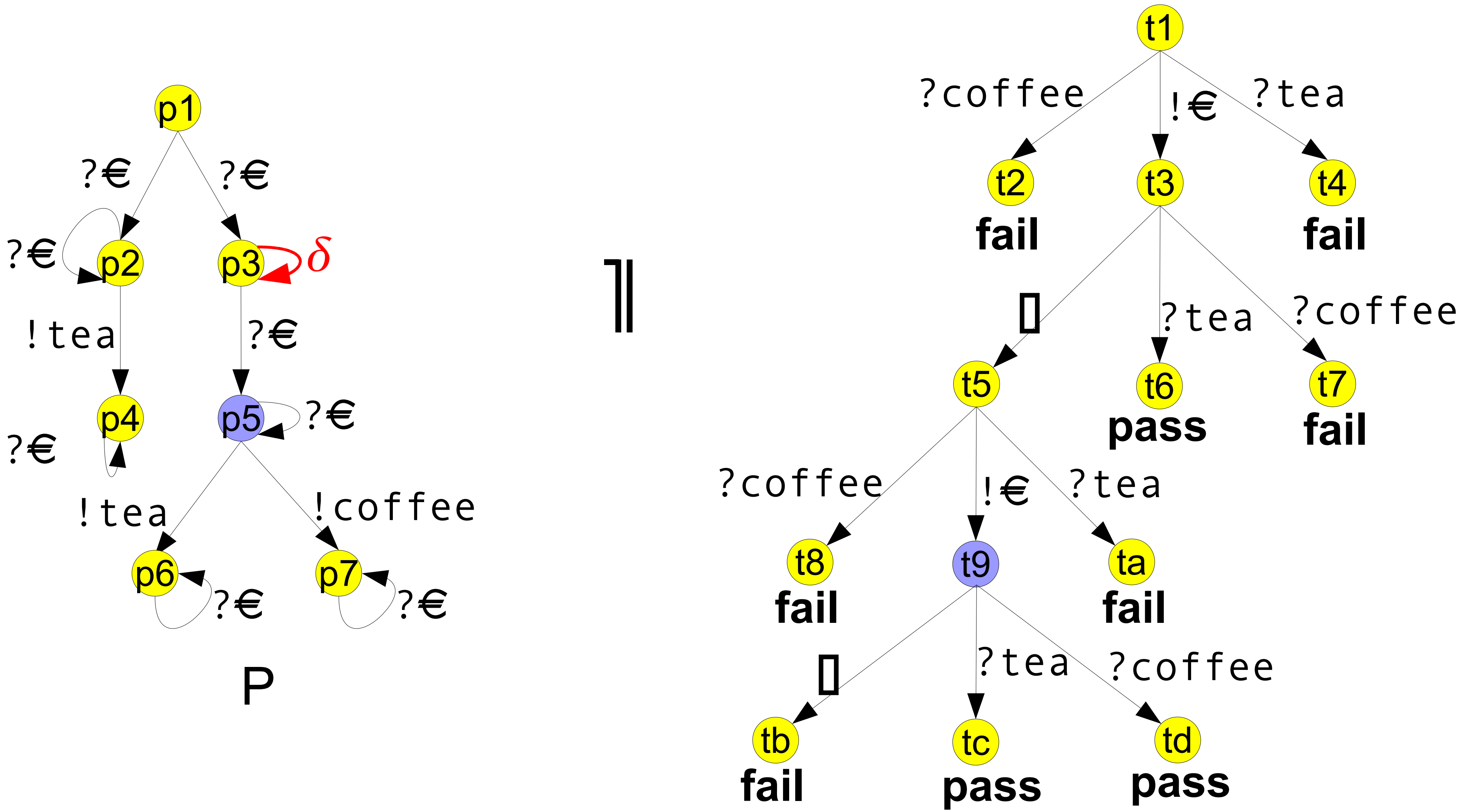
Formal Test Execution

- **Formally executing** a test case means putting it in parallel with the implementation model, leading to a **verdict**.



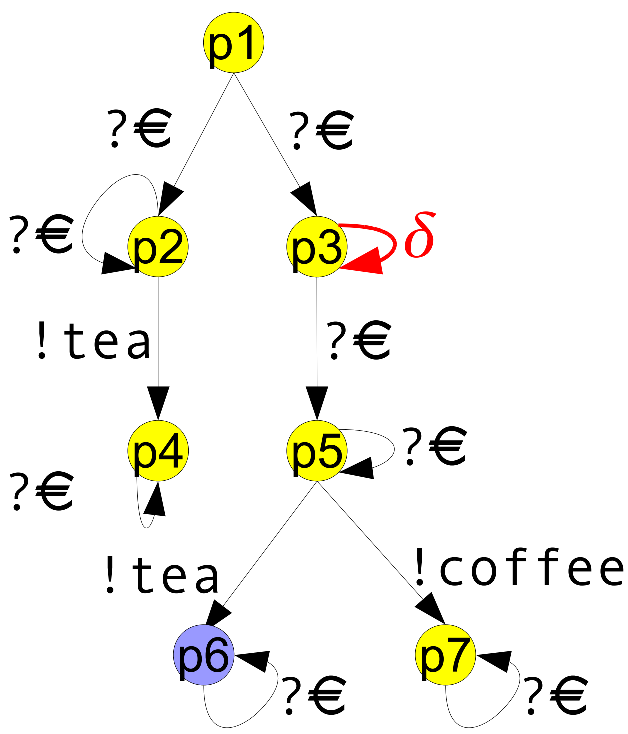
Formal Test Execution

- **Formally executing** a test case means putting it in parallel with the implementation model, leading to a **verdict**.



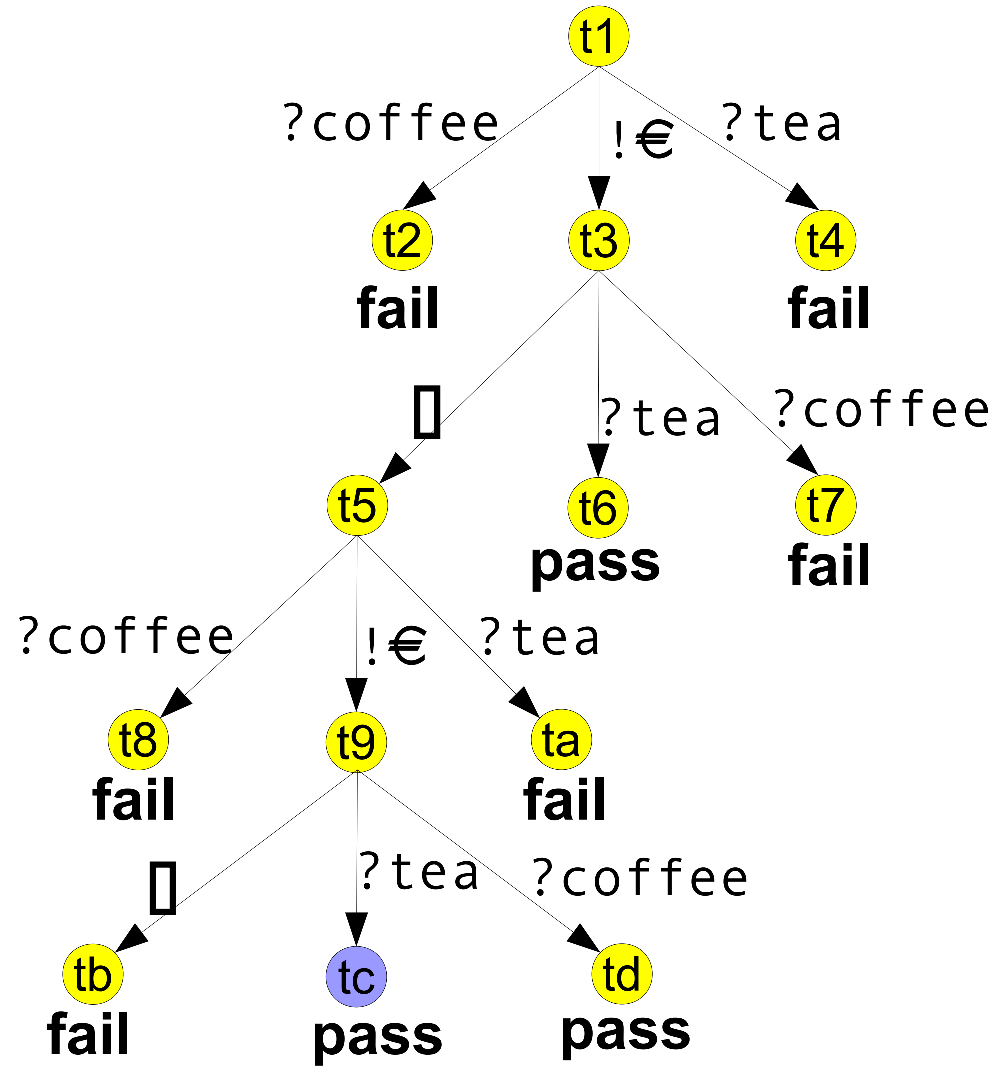
Formal Test Execution

- **Formally executing** a test case means putting it in parallel with the implementation model, leading to a **verdict**.



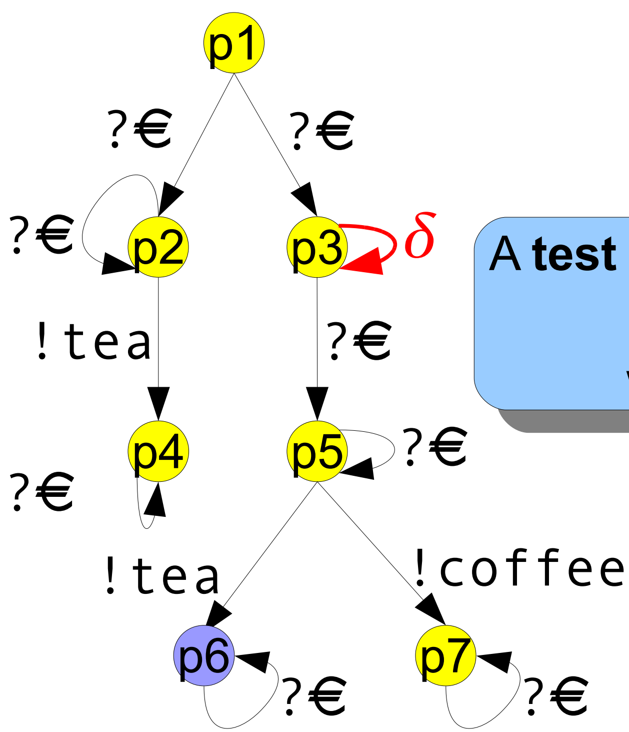
P

||



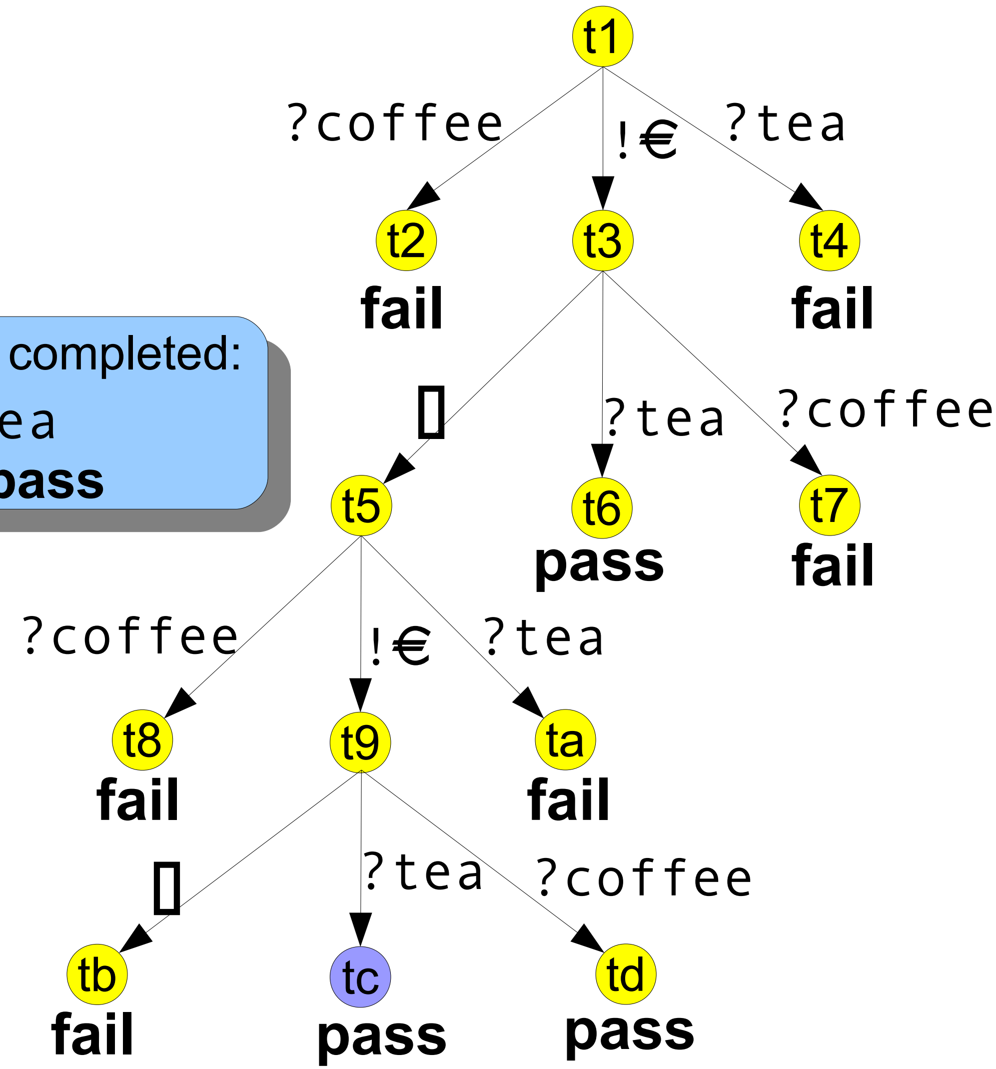
Formal Test Execution

- **Formally executing** a test case means putting it in parallel with the implementation model, leading to a **verdict**.



P

A **test run** has been completed:
!€▯!€?tea
with verdict **pass**



Observations

- The test runs represent the **observations**.
- In the previous example, two observations have been made:
  - `!€?tea`
  - `!€□!€?tea`

- Note that the set of all test runs for a given test case comprises **all possible observations** for all nondeterministic cases.
- One more observation could have been made in the previous example:
  - `!€□!€?coffee` (with verdict **pass**)

Dijkstra Revisited

- **When** should we stop testing?
- **Which** test cases shall we select?

  ⇒ How to deal with the practical incompleteness of testing?

1) Accept it, and focus on **heuristics** like code coverage, model coverage, timing constraints, randomness, test purposes, etc.

2) Try to find further **assumptions**, which makes testing complete in practice, i.e., leading to a **finite** sound and complete test suite.

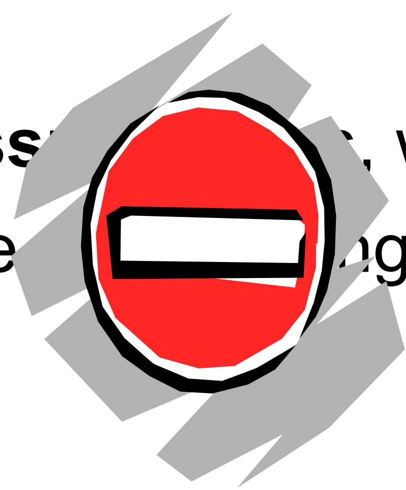
Dijkstra Revisited

- **When** should we stop testing?
- **Which** test cases shall we select?

  ⇒ How to deal with the practical incompleteness of testing?

The possibly infinite state space, and the nondeterministic character, make computing a **finite** sound and complete test suite an infeasible task!

2) Try to find further **ass** , which makes testing complete in practice g to a **finite** sound and complete test suite.



Dijkstra Revisited

- **When** should we stop testing?
- **Which** test cases shall we select?

  ⇒ How to deal with the practical incompleteness of testing?

1) Accept it, and focus on **heuristics** like code coverage, model coverage, timing constraints, randomness, test purposes, etc.

2) Try to find further **assumptions**, which makes testing complete in practice, i.e., leading to a **finite** sound and complete test suite.

Dijkstra Revisited

- **When** should we stop testing?
- **Which** test cases shall we select?

  ⇒ How to deal with the practical incompleteness of testing?

1) Accept it, and focus on **heuristics** like code coverage, model coverage, timing constraints, randomness, test purposes, etc.

2) Try to find further **assumptions**, which makes testing complete in practice, i.e., leading to a **finite** sound and complete test suite.

A Sound and Complete Test Generation Algorithm

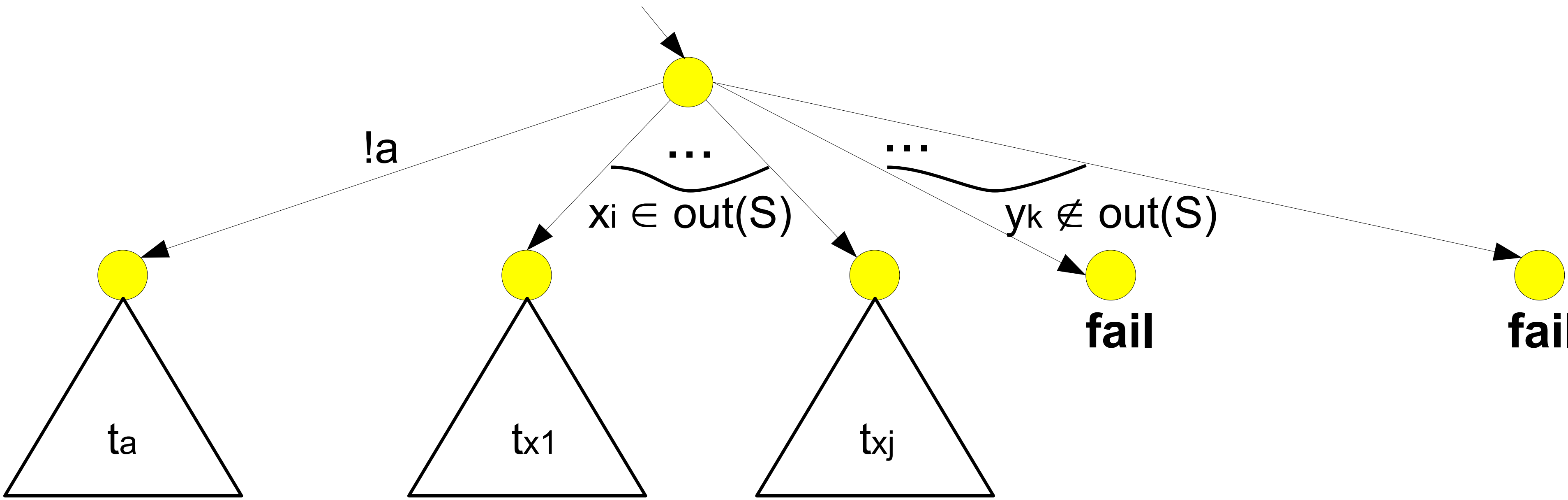
- Given a specification LTS with initial state s0
- Initially compute the set of states **K = s0 after ▯**
- Do a finite number of recursive applications of the following three nondeterministic choices:

a) Stop the test case with the verdict **pass**

pass

A Sound and Complete Test Generation Algorithm

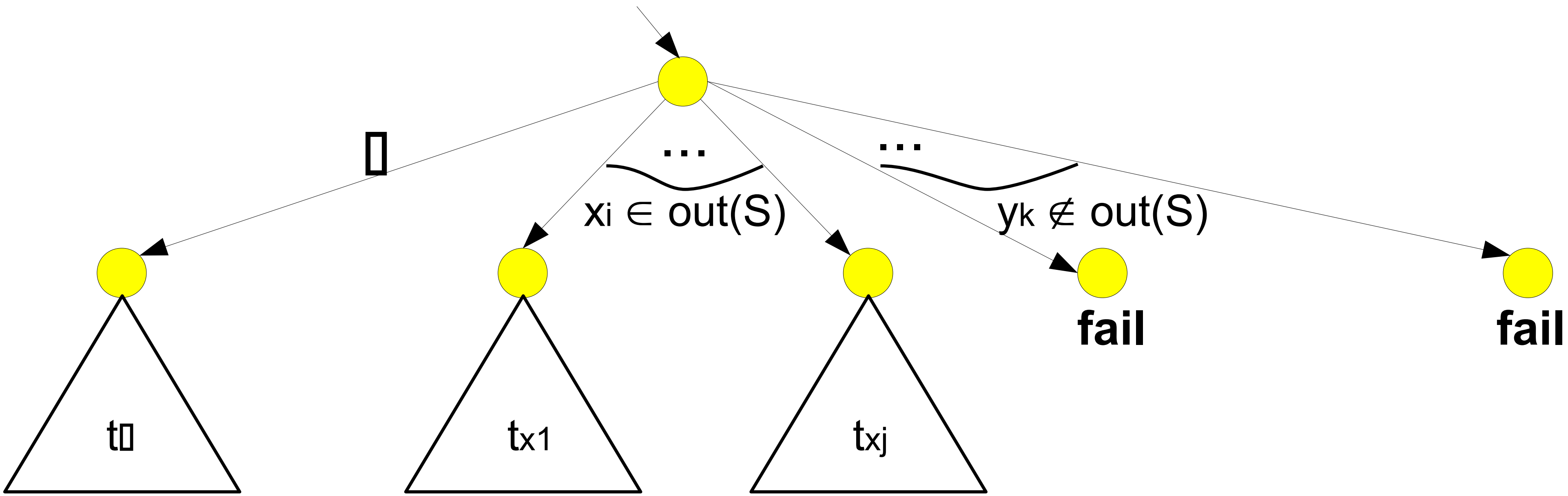
b) Let the test case produce an **output !a** with **K after a**$\leq\nabla$
Also accept all inputs at the same time.



t_a is obtained by applying the algorithm with **K = K after !a**
t_{xi} are obtained by applying the algorithm with **K = K after** $\mathbf{x_i}$

A Sound and Complete Test Generation Algorithm

c) Let the test case accept all inputs – and quiescence.

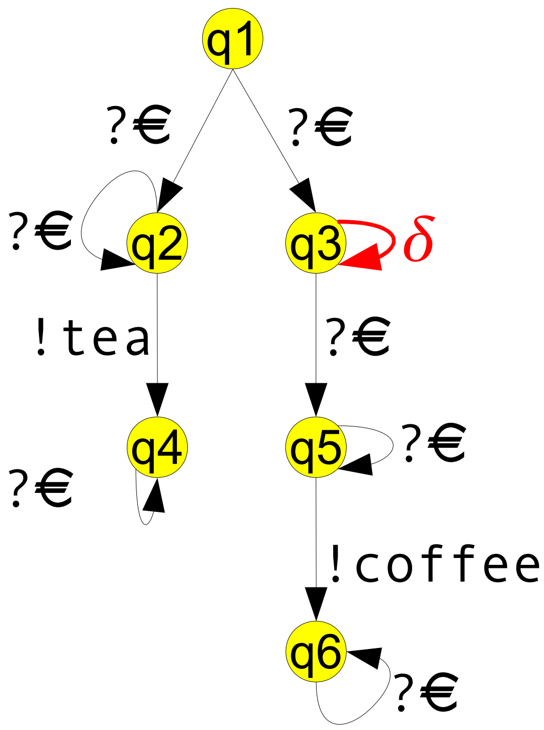


t is obtained by applying the algorithm with **K = K after**

t_{xi} are obtained by applying the algorithm with **K = K after x_i**

A Sound and Complete Test Generation Algorithm

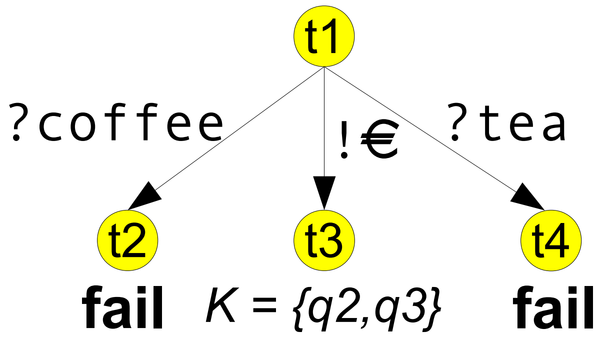
- We generate a test case out of Q.
- Initially, **K = {q1}**

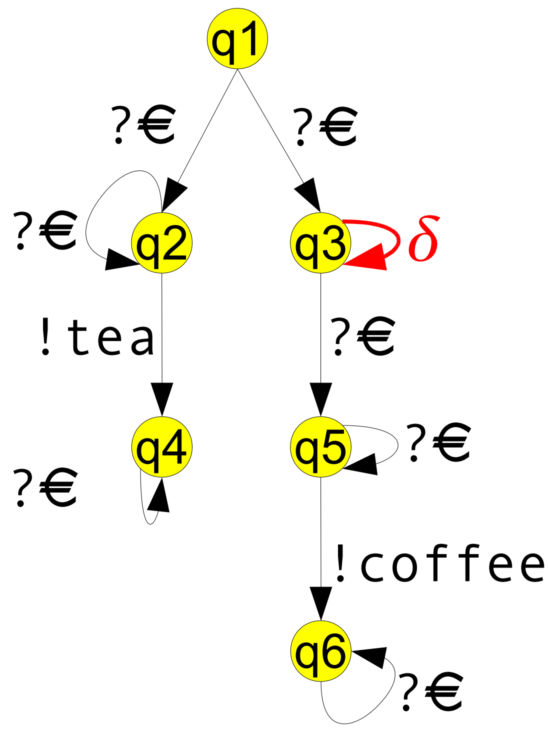


Q

A Sound and Complete Test Generation Algorithm

b) Let the test case produce an **output !a** with **K after a≤∇**
Also accept all inputs at the same time.

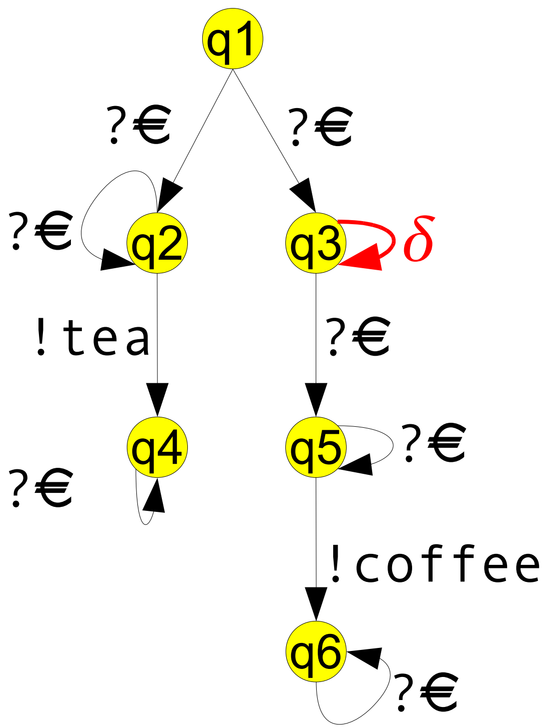




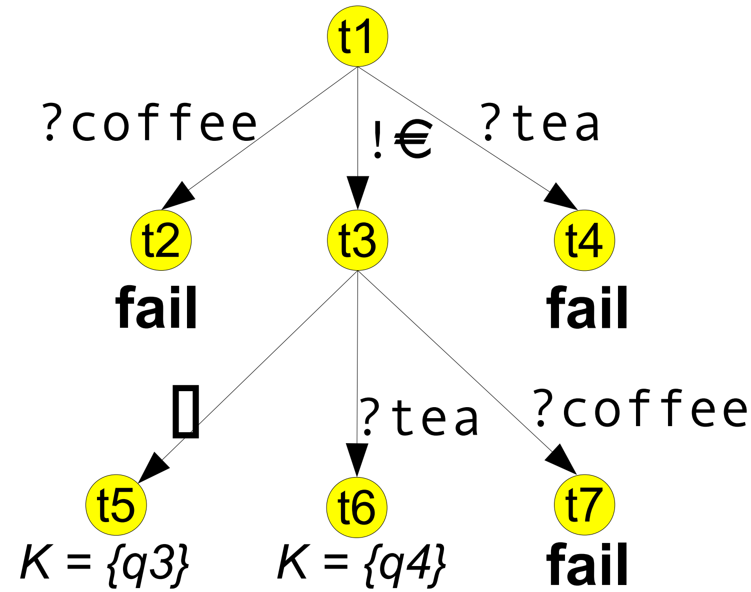
Q

A Sound and Complete Test Generation Algorithm

c) Let the test case accept all inputs – and quiescence.

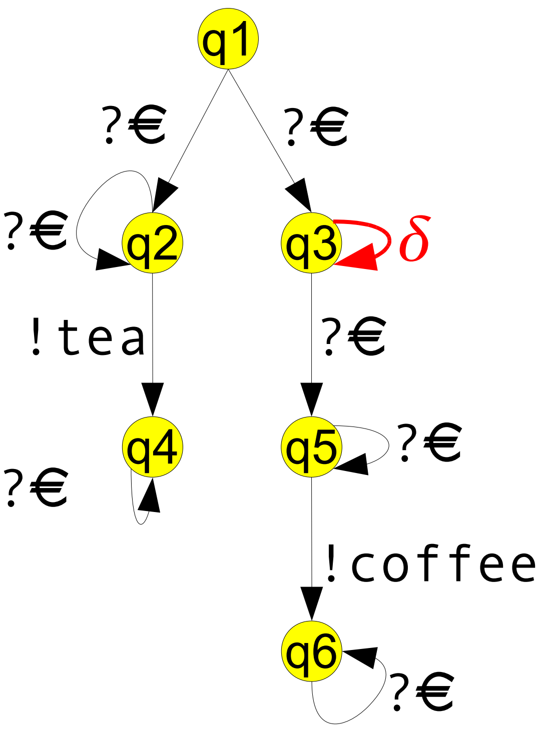


Q

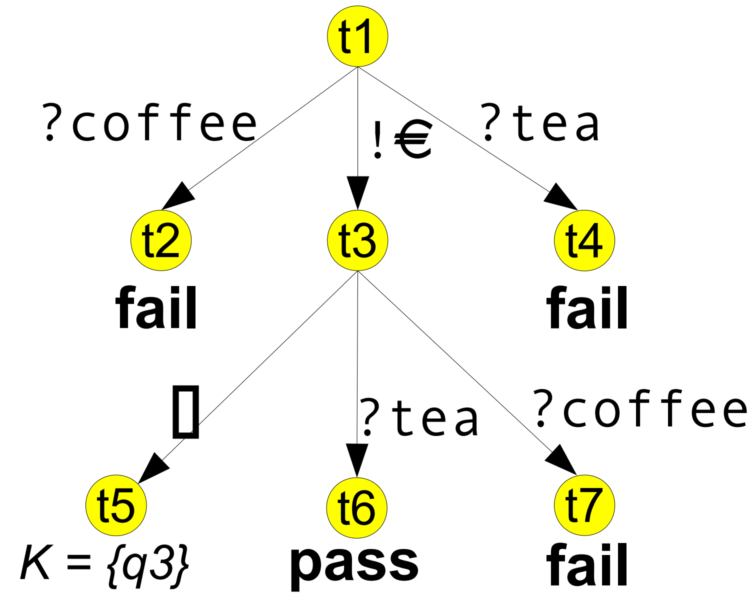


A Sound and Complete Test Generation Algorithm

a) Stop the test case with the verdict **pass**

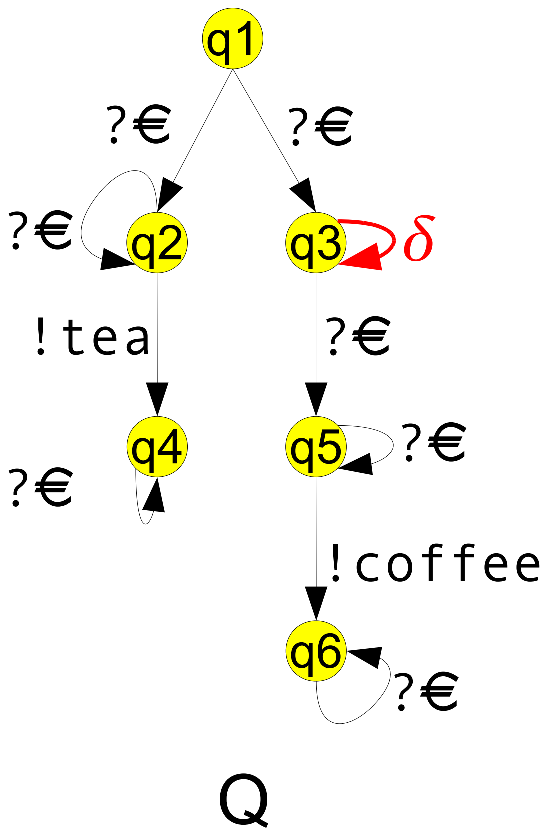


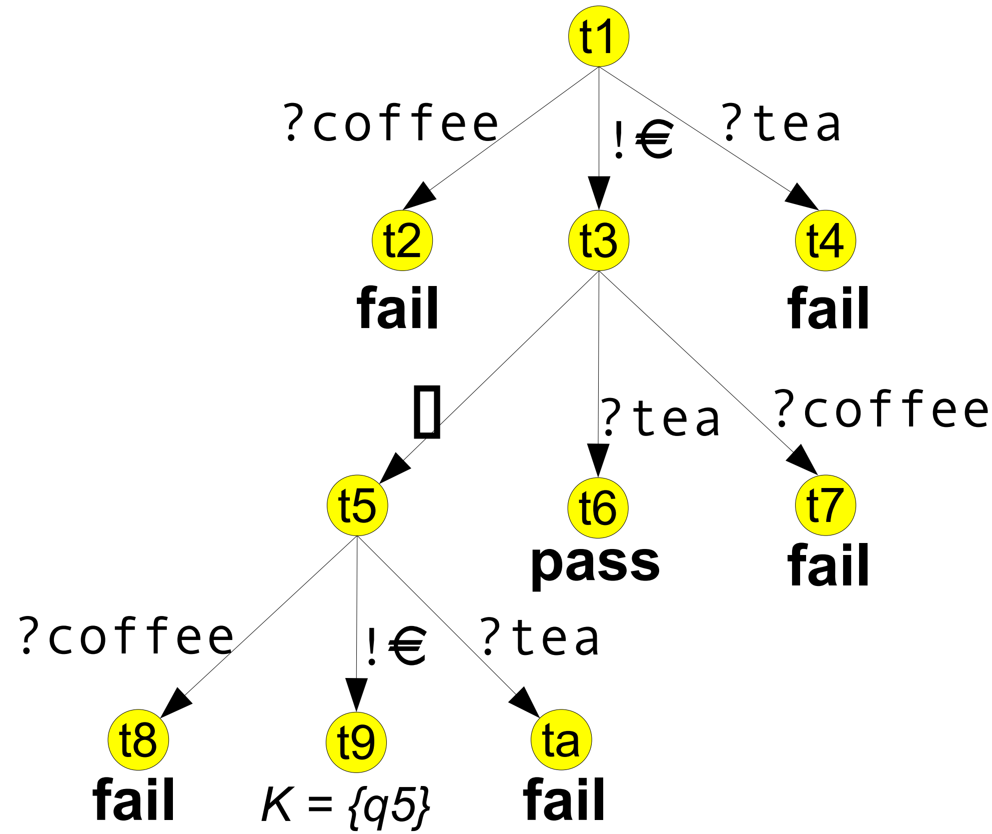
Q



A Sound and Complete Test Generation Algorithm

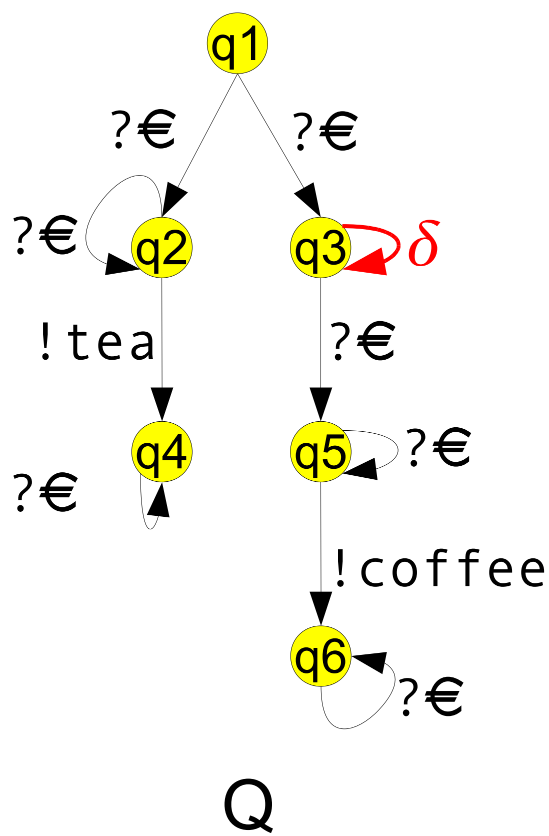
b) Let the test case produce an **output !a** with **K after a≤∇**
Also accept all inputs at the same time.

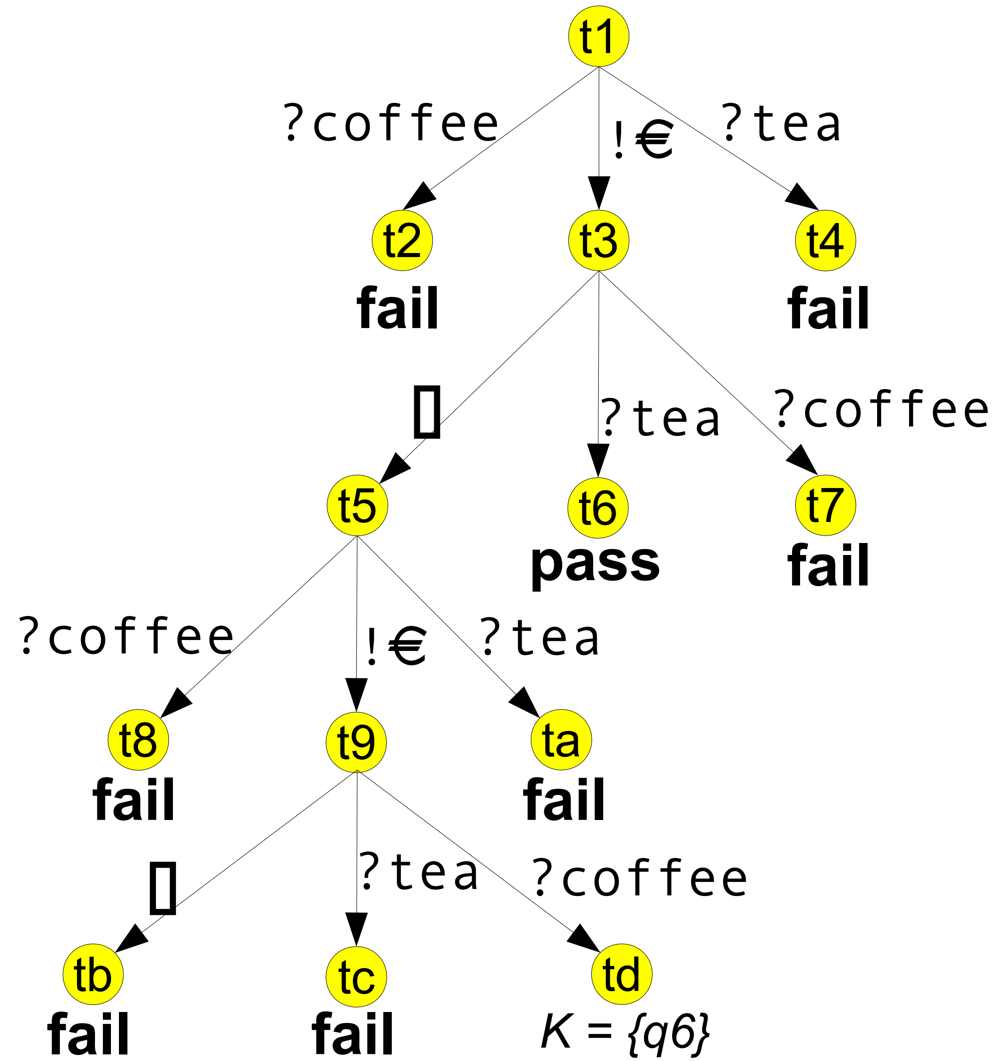




A Sound and Complete Test Generation Algorithm

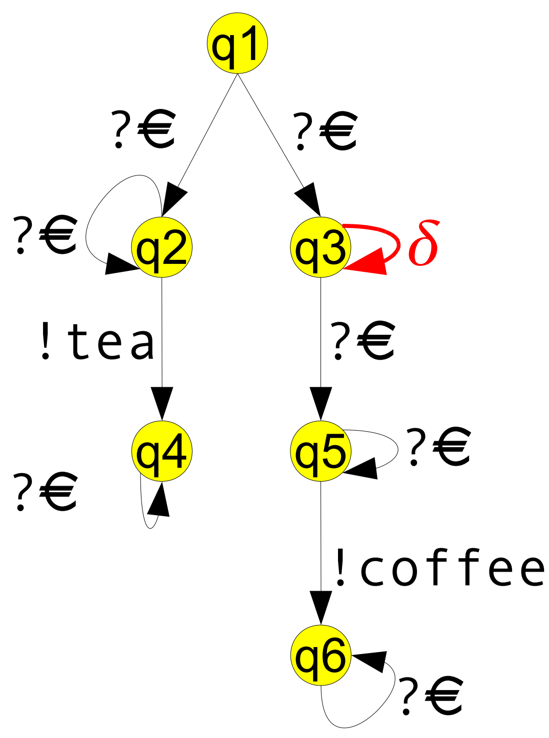
c) Let the test case accept all inputs – and quiescence.



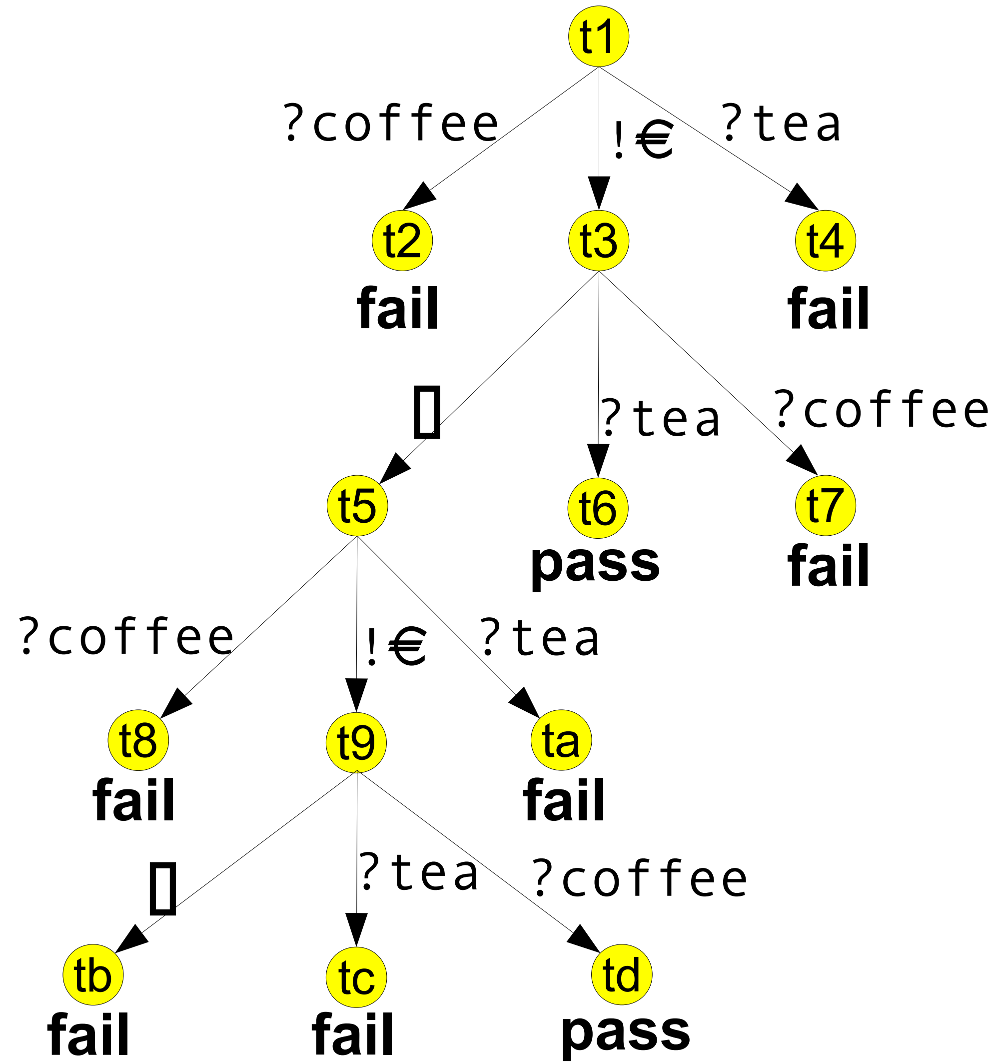


A Sound and Complete Test Generation Algorithm

a) Stop the test case with the verdict **pass**



Q

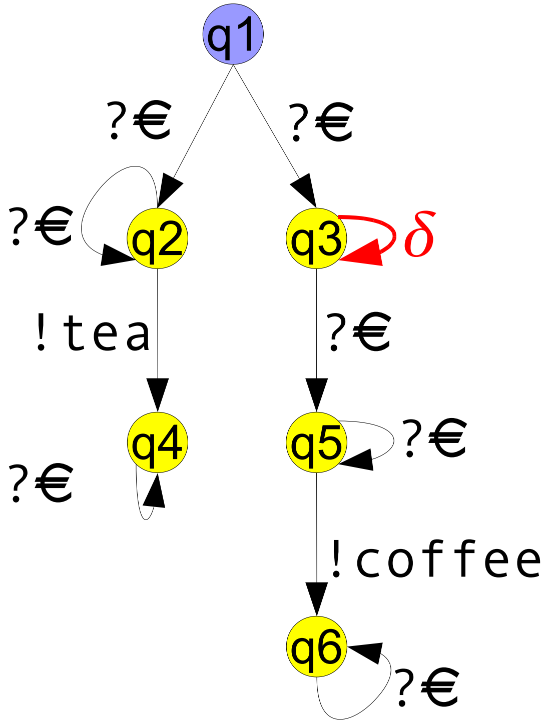


On-The-Fly Testing

- In every state a test case has to be defined **for all possible inputs**, i.e., outputs from the system.
- This can easily let the **state space explode**.
- Some tools do not firstly generate a test suite, and then apply it on the system.
- They **combine the test case generation and execution** process.
- By so doing, **outputs observed from the system guide the "test case" generation**.
- So doing avoids this state space explosion problem.
- This kind of testing is called **on-the-fly testing**.

On-The-Fly Testing

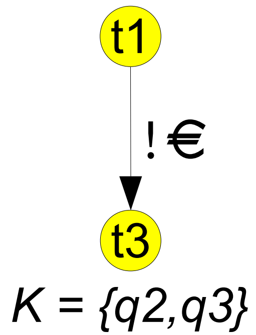
- We test on-the-fly with Q.
- Initially, **K = {q1}**

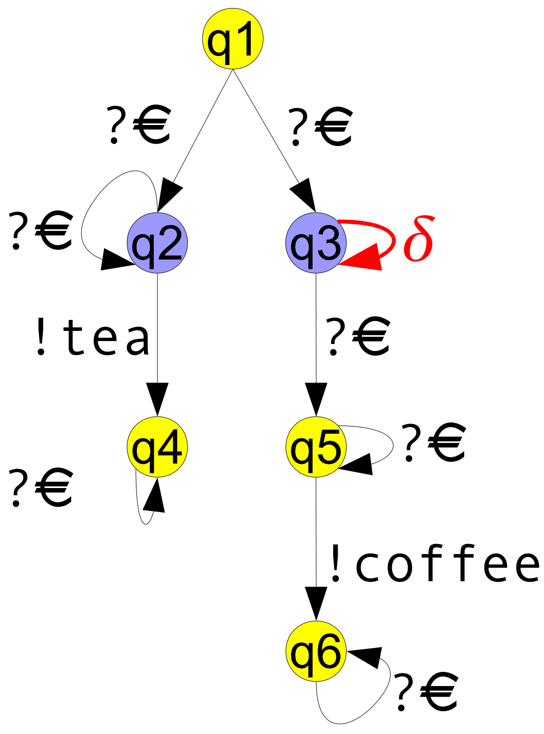


Q

b) We choose some **output !a** with **K after a≤∇**
We also accept all inputs at the same time.

We choose to give ! € to the system.



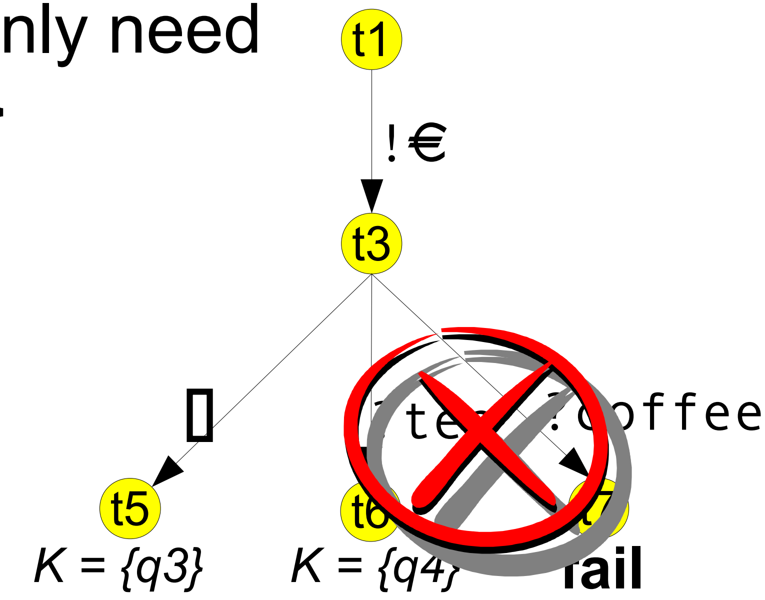


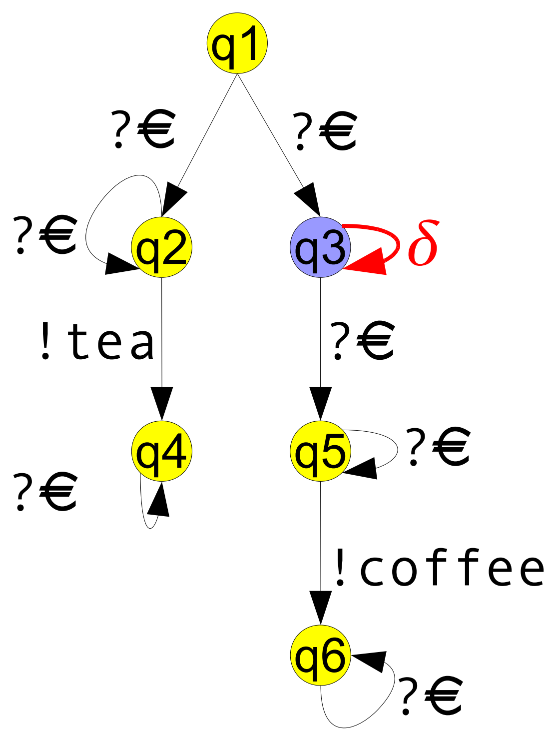
Q

On-The-Fly Testing

c) We accept all inputs – and quiescence.

We observe quiescence and hence only need to continue with state t5 and **K = {q3}**





Q

Summary

- LTS are a common formalism to model **reactive systems**.
- LTS are the underlying semantics of several other formalisms like like statecharts or process algebras.
- Relating two LTS can be done in a **variety of manners**.
- Not all relations are suited for testing purposes.
- Partitioning the action labels into *inputs* and *outputs* leads to an **IOLTS**.
- A common implementation relation for IOLTS is ***ioco***.
- *ioco* assumes implementation models to be **input enabled**.
- *ioco* allows specifications to be not input enabled – allowing for **partial specifications**.

Summary

- A test case is a tree-structured IOLTS with **pass** and **fail** leaves.
- Test cases must be **output-complete** for all possible outputs of the system.
- To avoid a state space explosion in test cases, the generation and execution of test cases can be combined – called **on-the-fly testing**.
- A simple sound and complete test case generation for *ioco* exists.
- This algorithm is implemented in an on-the-fly manner in the **TorX tool**.
- The **TGV tool** combines ioco testing with **test purposes**.