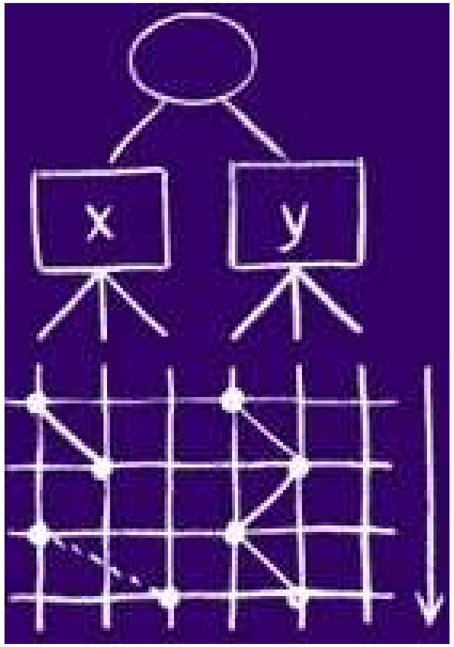
Model-Based Testing

--- Principles, Methods, and Tools

(based on the slides of Brian Nielsen and Jan Tretmans)



Agenda

- Overview
- Finite State Machine (FSM)-based testing
- Labelled Transition System (LTS)-based testing
- Model-Based Real-time System Testing --- The Uppaal Approach
- Tools for Model-Based Testing
- Summary

The Nature of Testing

Testing: the activity of

checking or measuring some quality characteristics

of an executing object (i.e., IUT)

by performing experiments

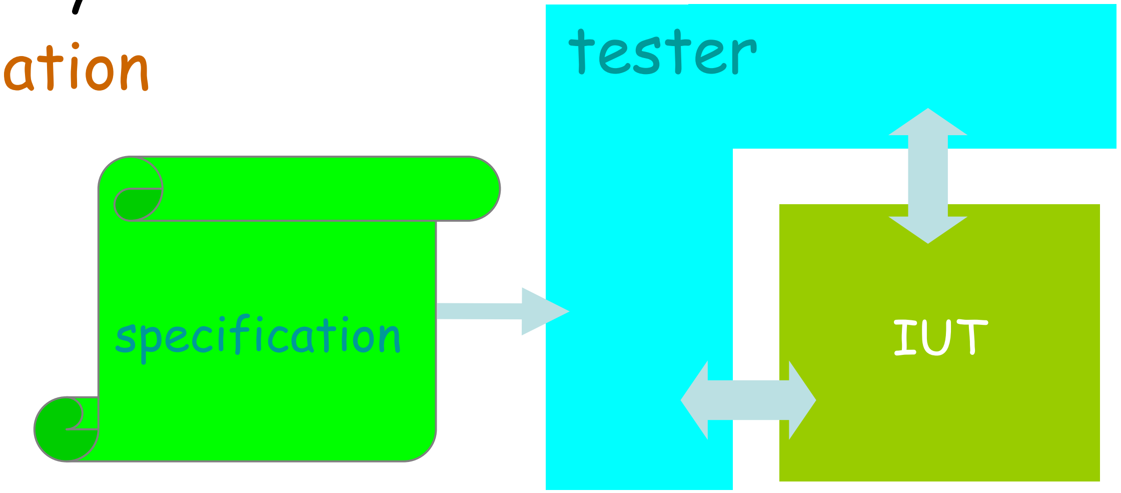
in a controlled way

w.r.t. a specification

not just on models (that's formal verification or simulation)

not just by reasoning

to decide whether it passes or fails



IUT: the Implementation Under Test

Towards Model-Based Testing

- Increase in complexity, and quest for higher quality software
  - testing effort grows exponentially with complexity
  - testing cannot keep pace with development

- Ever-changing requirements
  - model-based development

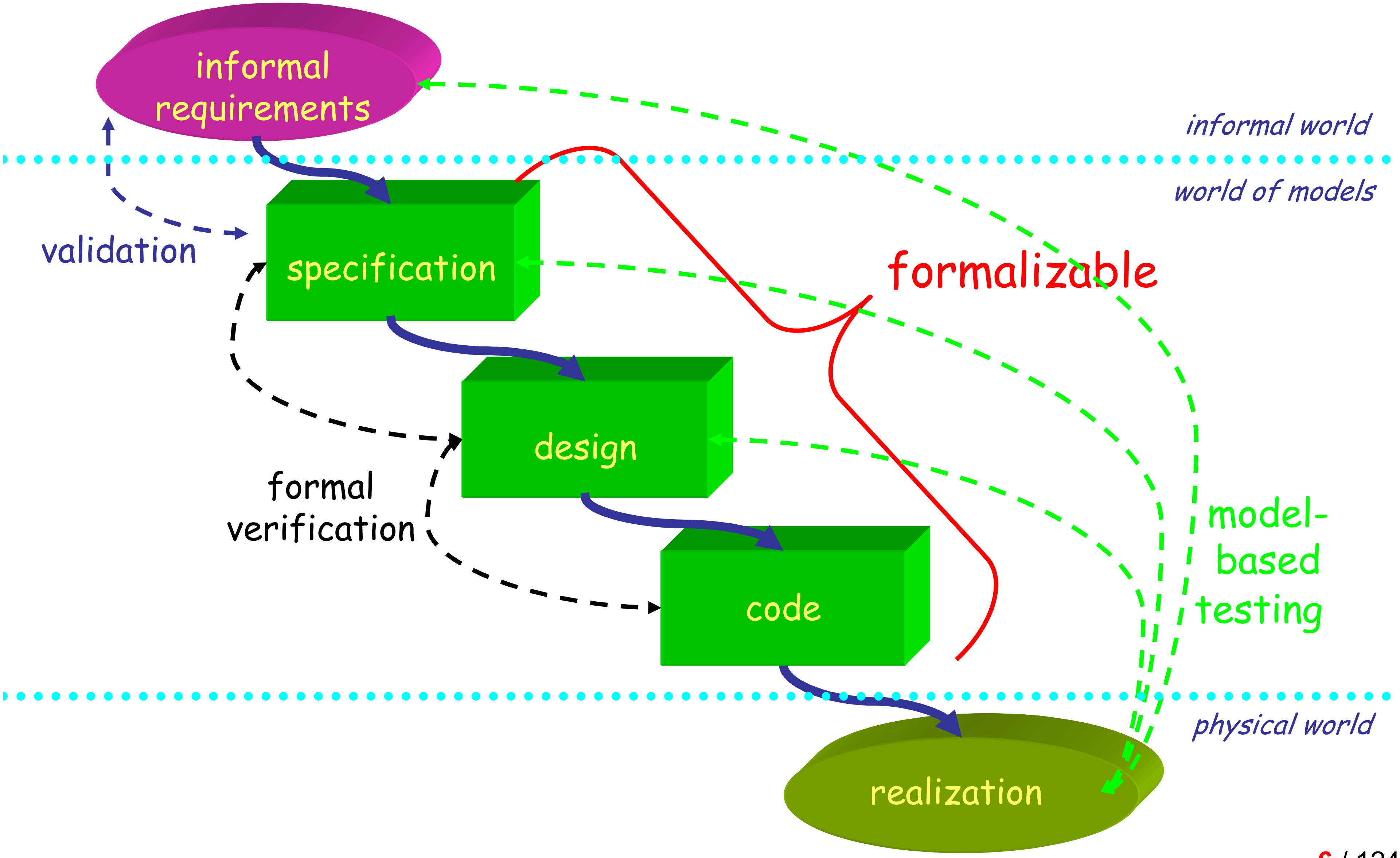
- Checking software quality
  - practice: testing - ad hoc, too late, expensive, lot of time
  - research: formal verification - proofs, model checking, . . . , with disappointing practical impact

Towards Model-Based Testing (cont'd)

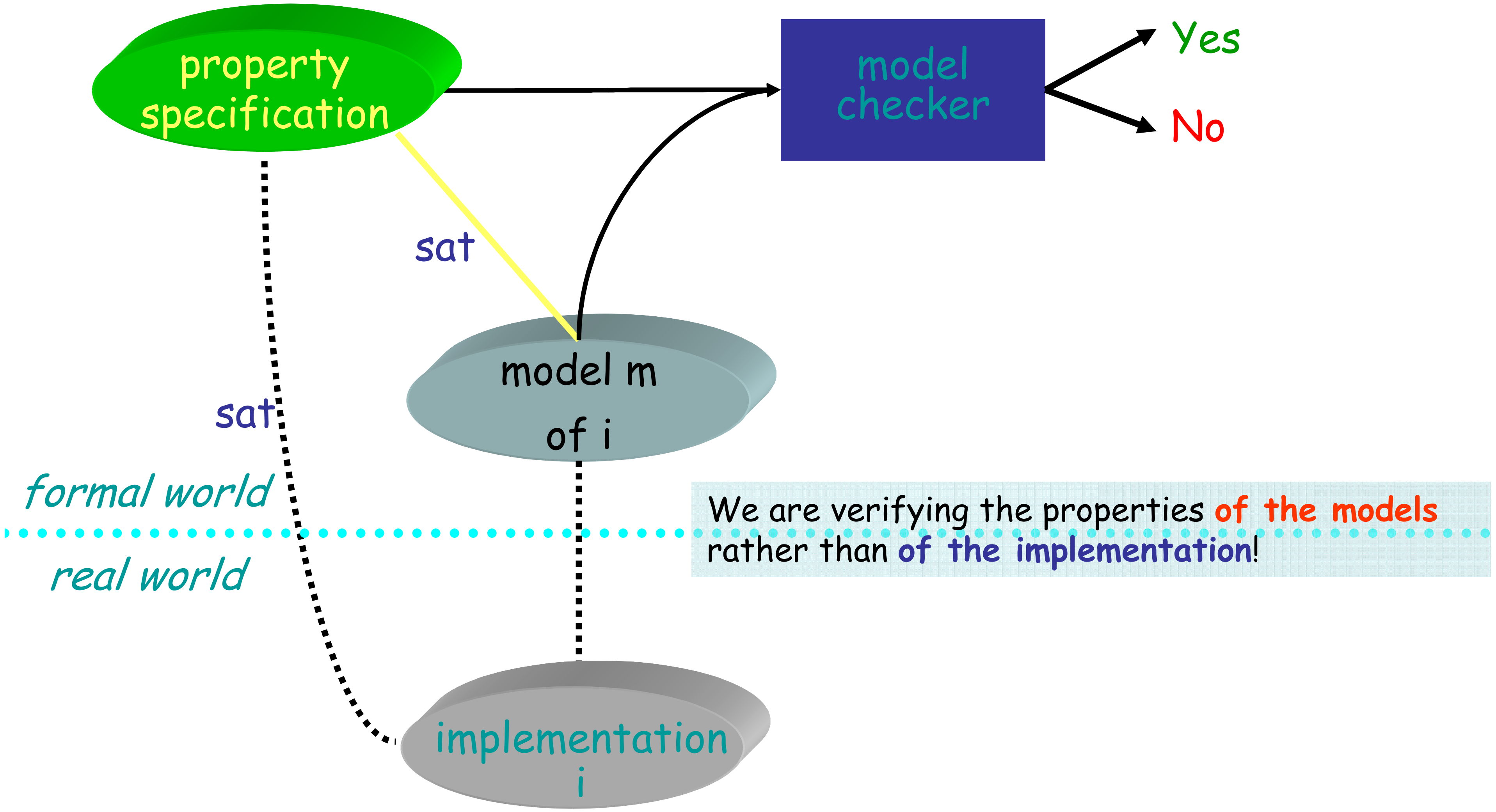
- Model-based testing has potential to combine
  - practice - testing
  - theory - formal methods

- Model-Based Testing:
  - generating tests from a (formal) model / specification
    - state model, pre/post, CSP, Promela, UML, Spec#, ...
  - testing with respect to a (formal) model / specification
  - promises better, faster, cheaper testing:
    - algorithmic generation of tests and test oracles, with tool support
    - formal and unambiguous basis for testing
    - measuring the completeness of tests
    - maintenance of tests through model modification

A Model-Based Development Process

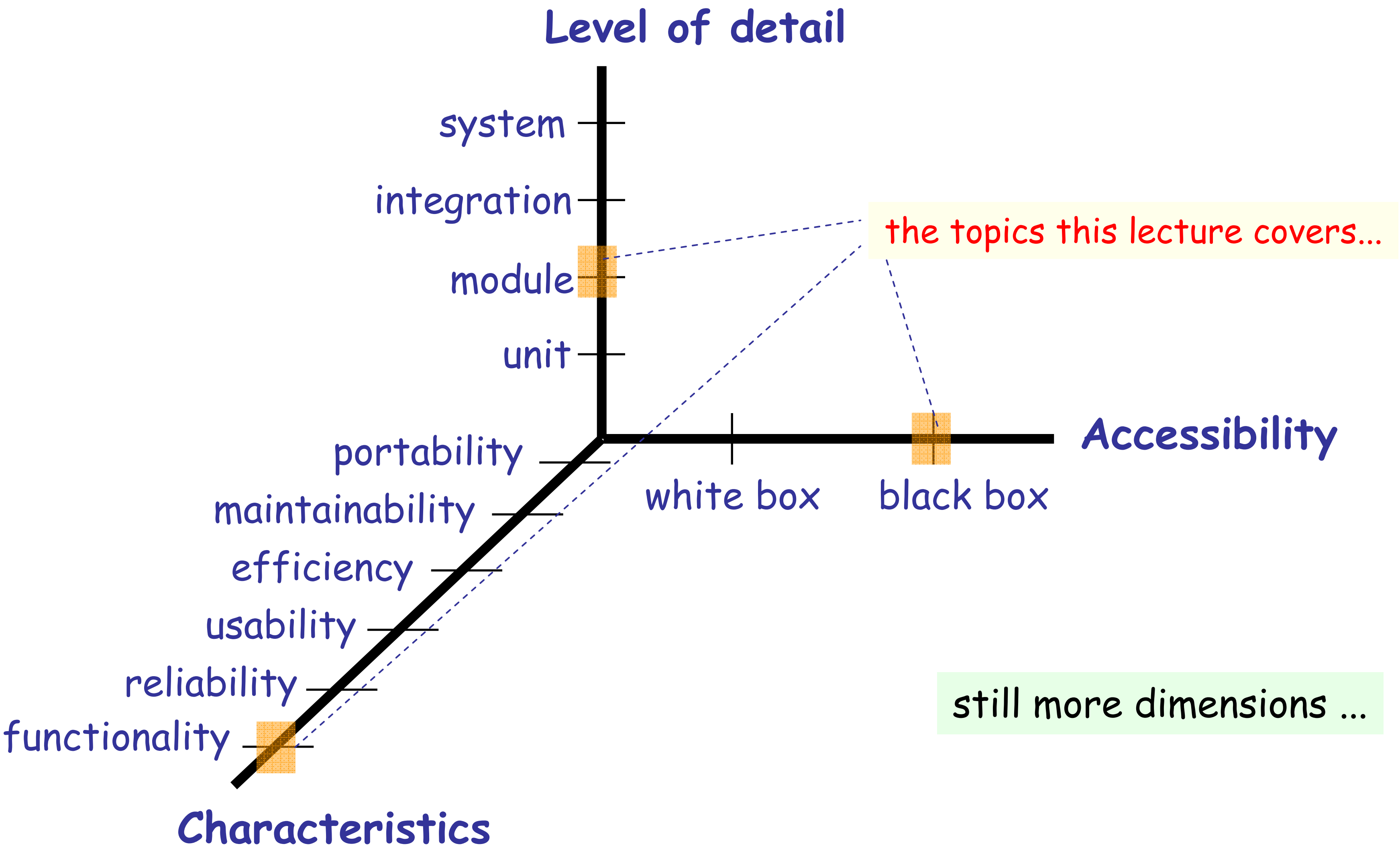


Formal Verification



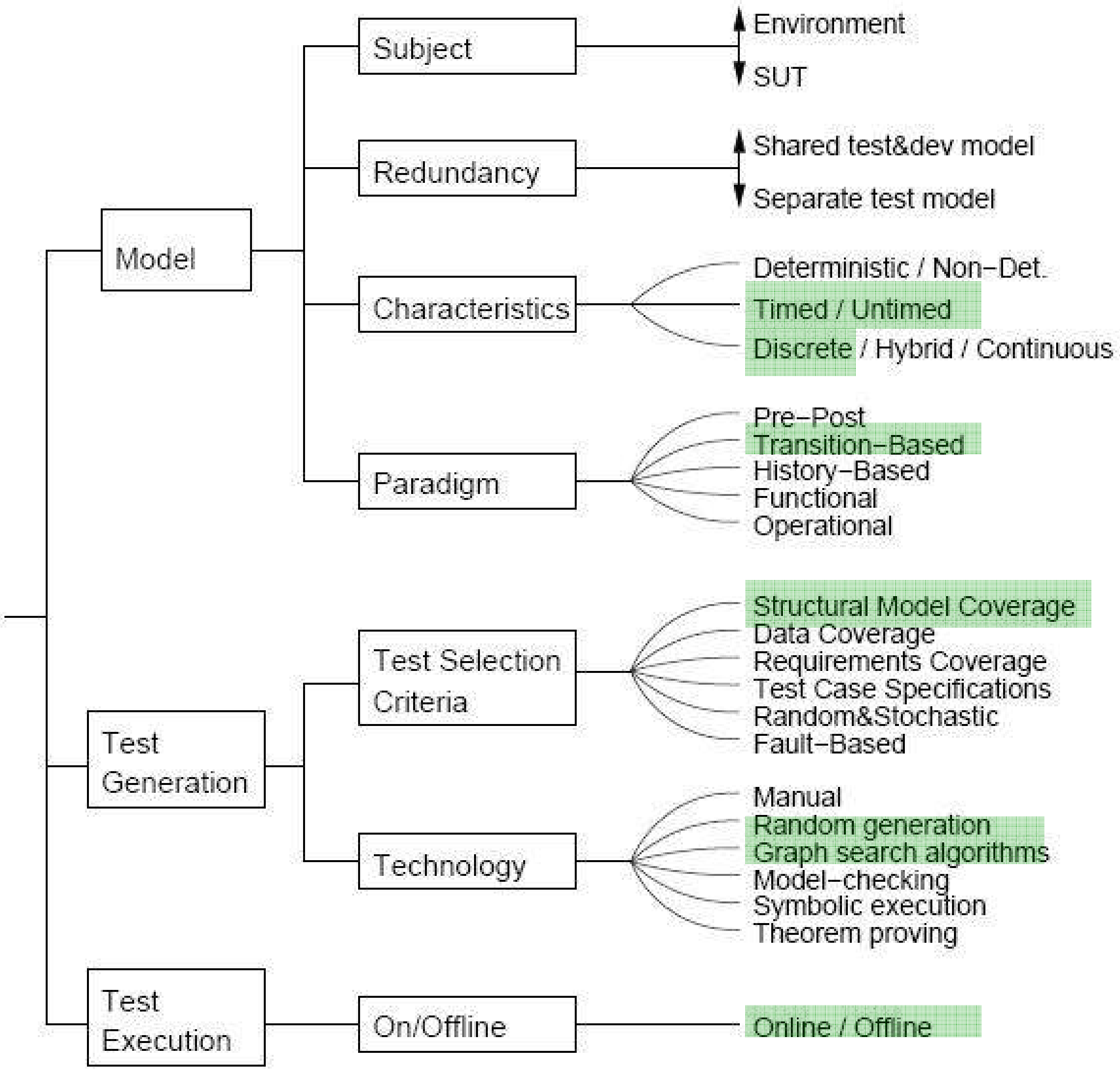
We are verifying the properties **of the models** rather than **of the implementation**!

Types of Testing



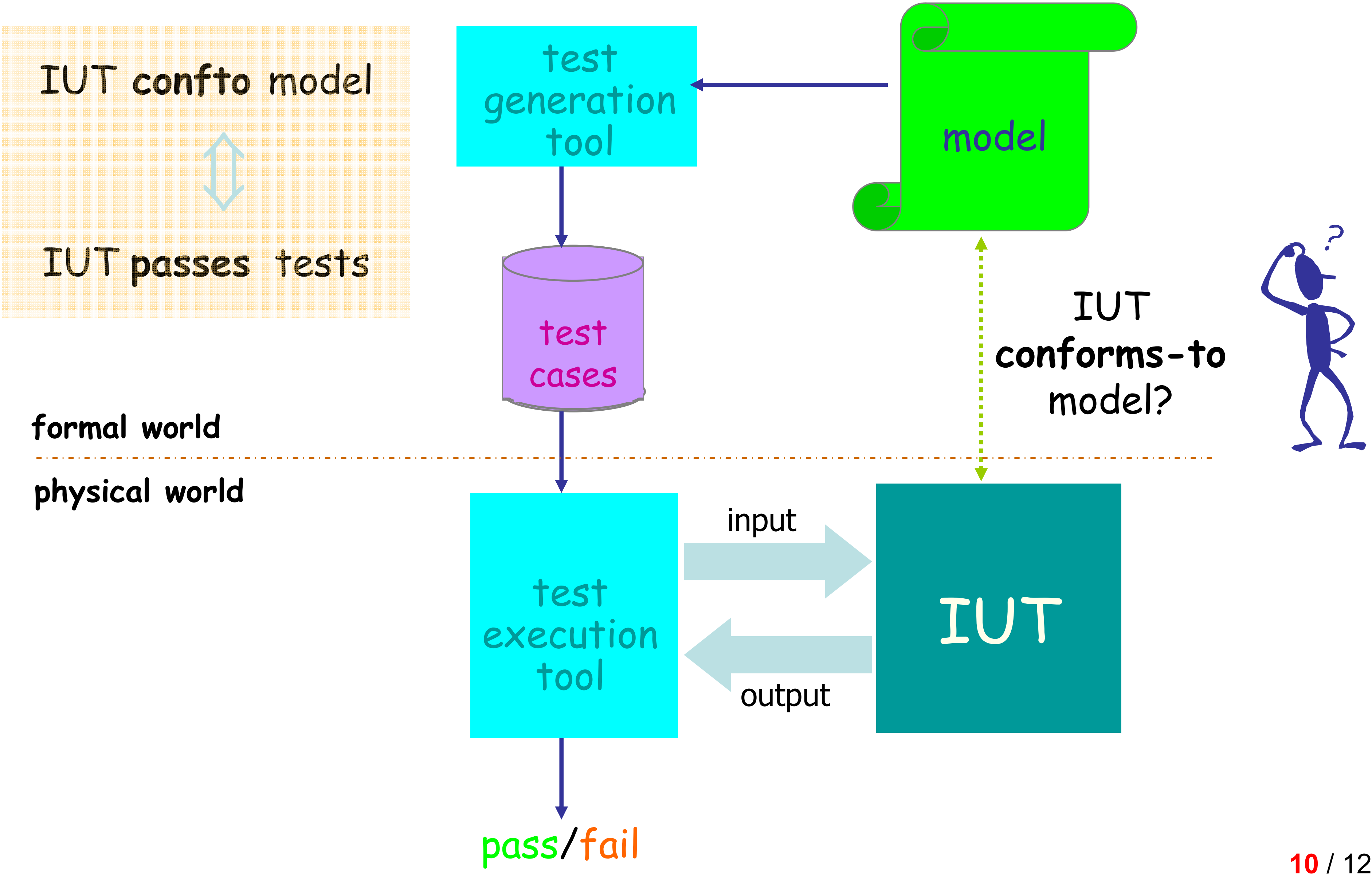
still more dimensions ...

A Taxonomy of Model-Based Testing



[Mark Utting 2006]

Automated Model-Based Testing



IUT **confto** model

⇕

IUT **passes** tests

test generation tool

model

test cases

IUT **conforms-to** model?

formal world

physical world

test execution tool

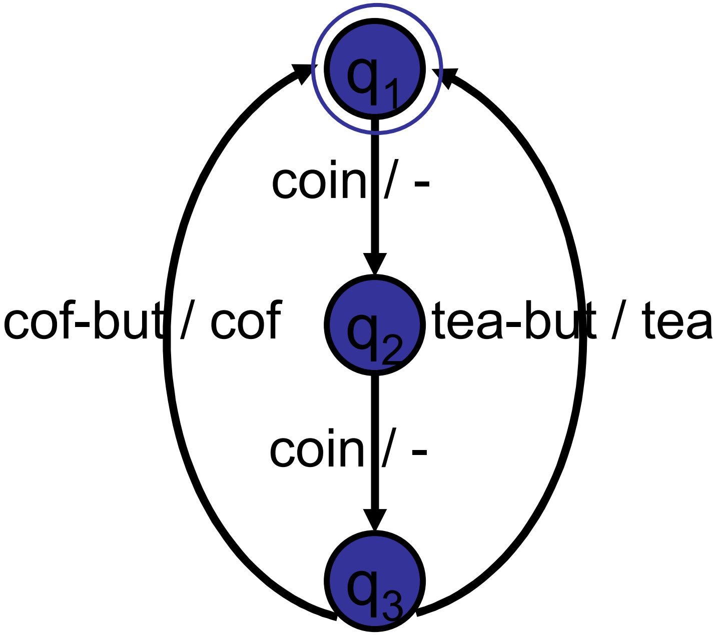
input

output

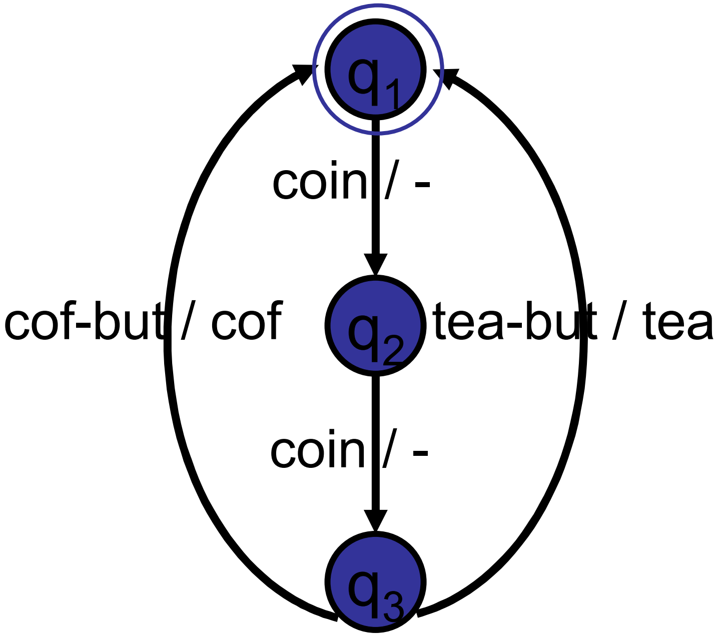
IUT

pass/fail

Finite State Machine (FSM)-Based Testing



FSM example (Mealy machine)



condition		effect	
current state	input	output	next state
q_1	coin	-	q_2
q_2	coin	-	q_3
q_3	cof-but	cof	q_1
q_3	tea-but	tea	q_1

Inputs = {cof-but, tea-but, coin}
Outputs = {cof,tea}
States: $\{q_1,q_2,q_3\}$
Initial state = q_1
Transitions= {
$(q_1$, coin, -, $q_2)$,
$(q_2$, coin, -, $q_3)$,
$(q_3$, cof-but, cof, $q_1)$,
$(q_3$, tea-but, tea, $q_1)$
}

Sample run:

$q_1 \xrightarrow{\text{coin/ -}} q_2 \xrightarrow{\text{coin/-}} q_3 \xrightarrow{\text{cof-but / cof}} q_1 \xrightarrow{\text{coin/ -}}$

$q_2 \xrightarrow{\text{coin/ -}} q_3 \xrightarrow{\text{cof-but / cof}} q_1$

A Formal Definition

The Mealy Machine is 5-tuple

$$M = (S, I, O, \delta, \lambda)$$

S	finite set of states
I	finite set of inputs
O	finite set of outputs
$\delta : S \times I \rightarrow S$	transfer function
$\lambda : S \times I \rightarrow O$	output function

Natural extension to sequences : $\delta : S \times I^* \rightarrow S$
$\lambda : S \times I^* \rightarrow O^*$

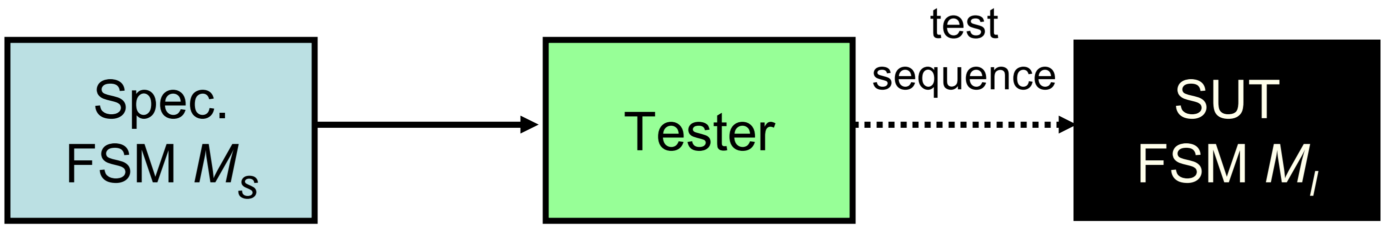
Basic Concepts

- Two states s and t of FSM are (language) equivalent iff
  - s and t accept same language
  - have same traces: tr(s) = tr(t)
- Two Machines M0 and M1 are equivalent iff the two initial states of them are equivalent
- A minimized (or reduced) M is one that has no equivalent states
  - for no two states s,t, s!=t, s equivalent t

Fundamental Results

- Every FSM may be determinized accepting the same language.
- For each FSM there exists a language-equivalent minimal deterministic FSM.
- FSM's are closed under $\cap$ and $\cup$
- FSM's may be described as regular expressions (and vice versa)

Conformance Testing



Given: a specification FSM M_S

a (black-box) implementation FSM M_I

Task: To determine whether M_I conforms to M_S,

i.e., M_I behaves in accordance with M_S

i.e., whether outputs of M_I are the same as of M_S

i.e., whether the reduced M_I is equivalent to M_S

Today we assume:

- Deterministic Specifications
- SUT is an (unknown) deterministic FSM (the testing hypothesis)

Some Restrictions

FSM restrictions:

$M = (S, I, O, \delta, \lambda)$

- deterministic

  $\delta : S \times I \rightarrow S$ and $\lambda : S \times I \rightarrow O$ are *functions*

  (rather than ordinary "relations“)

- completely specified

  $\delta : S \times I \rightarrow S$ and $\lambda : S \times I \rightarrow O$ are *complete* functions

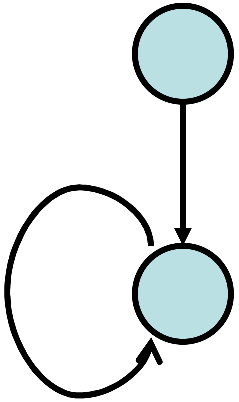
  (empty output is allowed; sometimes implicit completeness)

- strongly connected

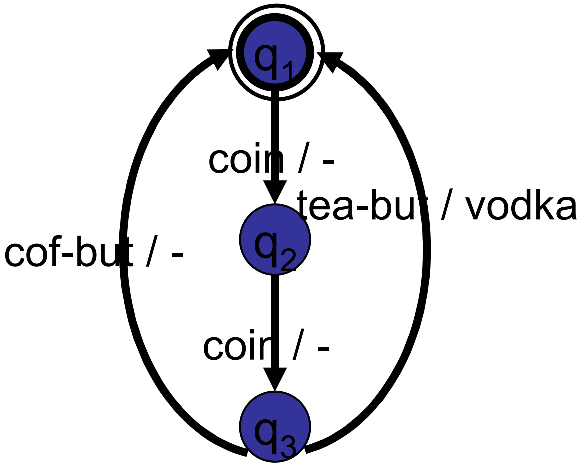
  from any state any other state can be reached

- reduced

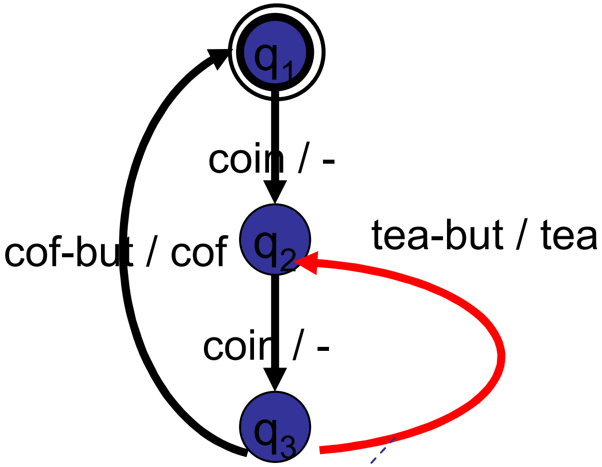
  there are no equivalent states



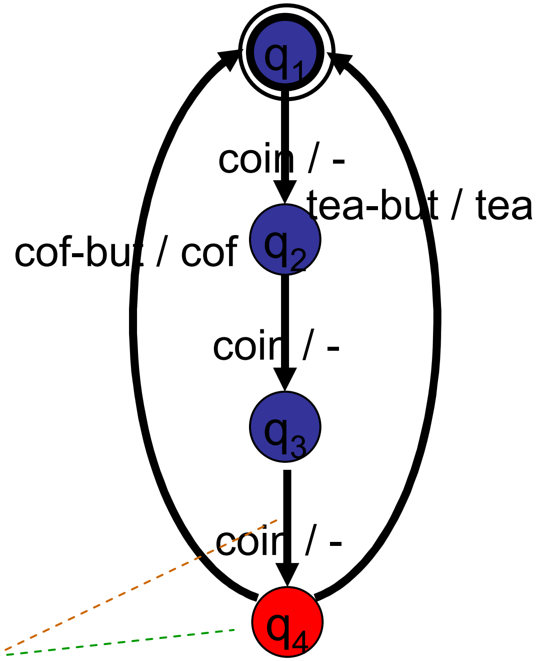
Type of Faults



correct model



erroneous model

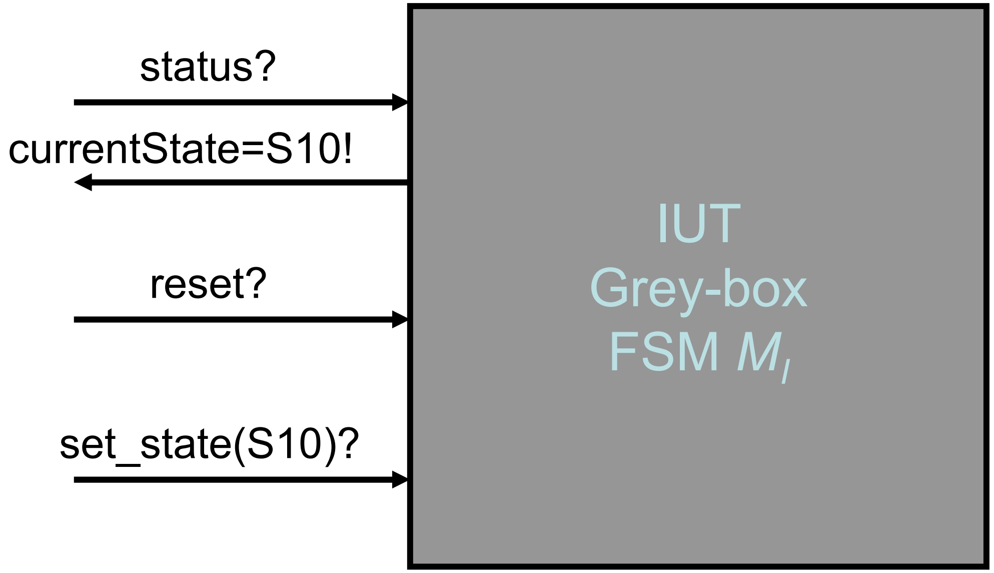


erroneous model

- output fault (wrong outputs or missing outputs)
- extra or missing states
- transition fault
  - to other state
  - to new state

Desired Properties

- Nice, but rare / problematic
  - "status" message: Assume that tester can enquire implementation for its current state (reliably!!) without changing state
  - reset: reliably bring IUT to the initial state
  - set_state(): reliably bring IUT to a specified state



FSM Testing

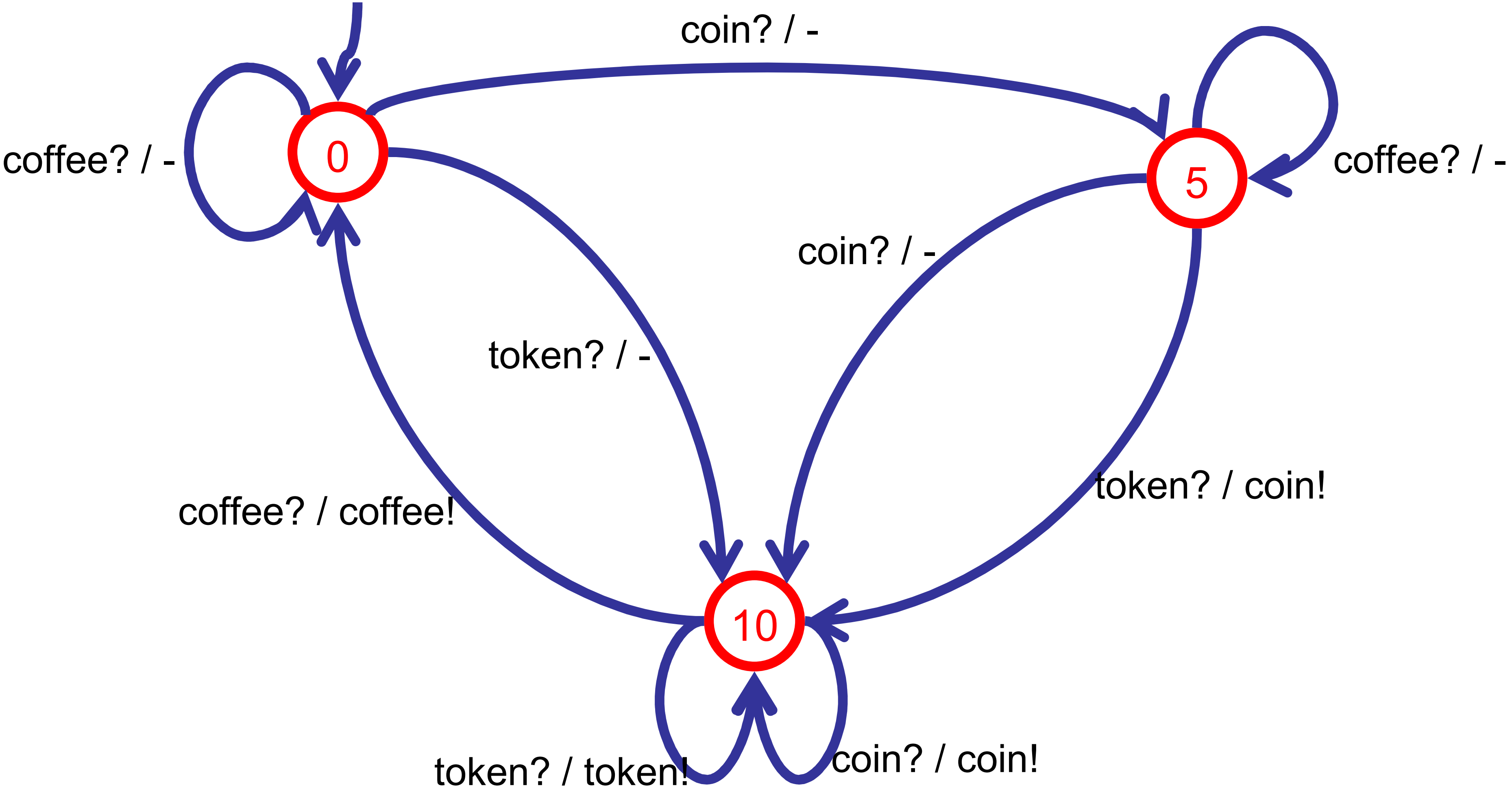
- Test with paths of the (specification) FSM
  - A path is a sequence of inputs with expected outputs
  - (cf. path testing as white-box technique)

- Infinitely many paths : how to select ?
- Different strategies :
  - test every state : state coverage (of specification !)
  - test every transition : transition coverage
    - test output of every transition
    - test output + resulting state of every transition
  - …

To find a path or a set of paths to cover all the states in the FSM

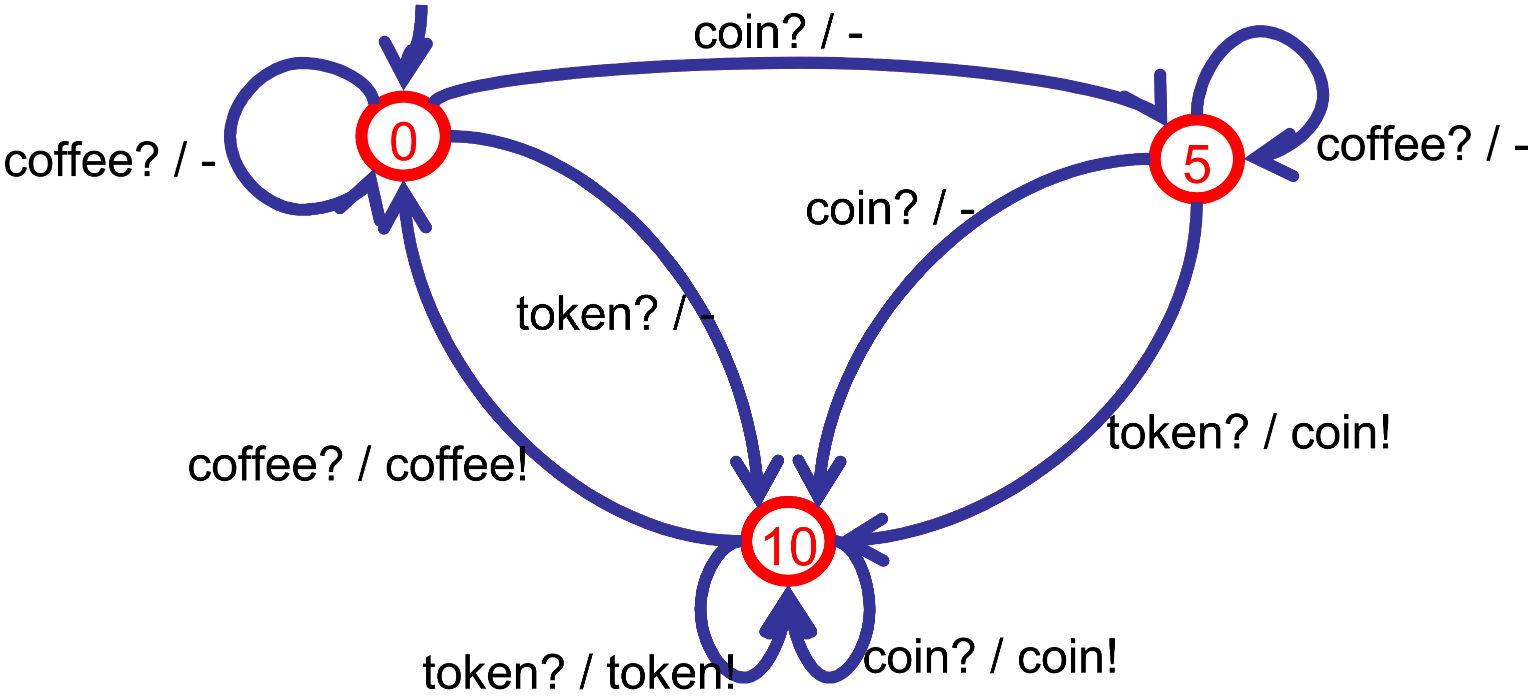
To find a path or a set of paths to cover all the transitions in the FSM

A Coffee Machine FSM (Mealy)



State Coverage

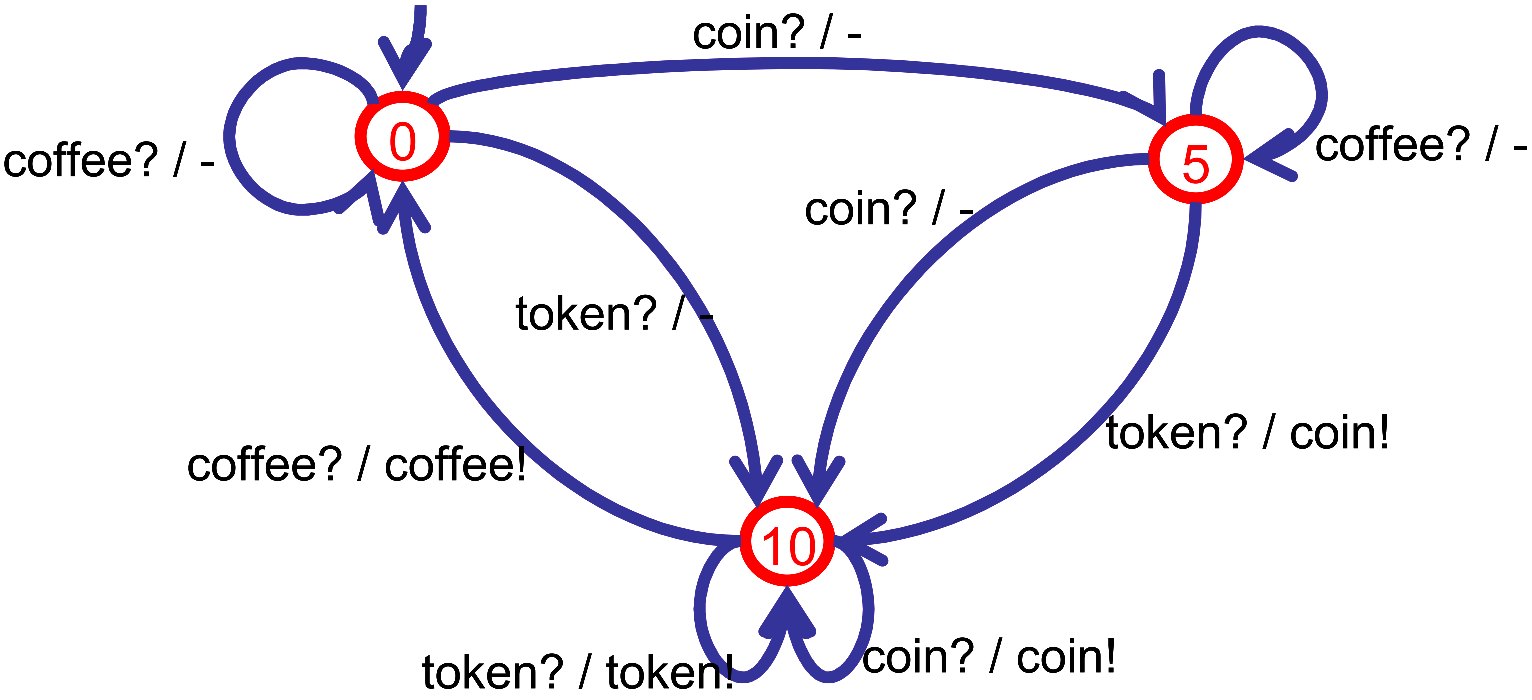
- Make *State Tour* that covers every state (in spec)



Test sequence : coin? token? coffee?

Transition Coverage

- Make *Transition Tour* that covers every transition (in spec)

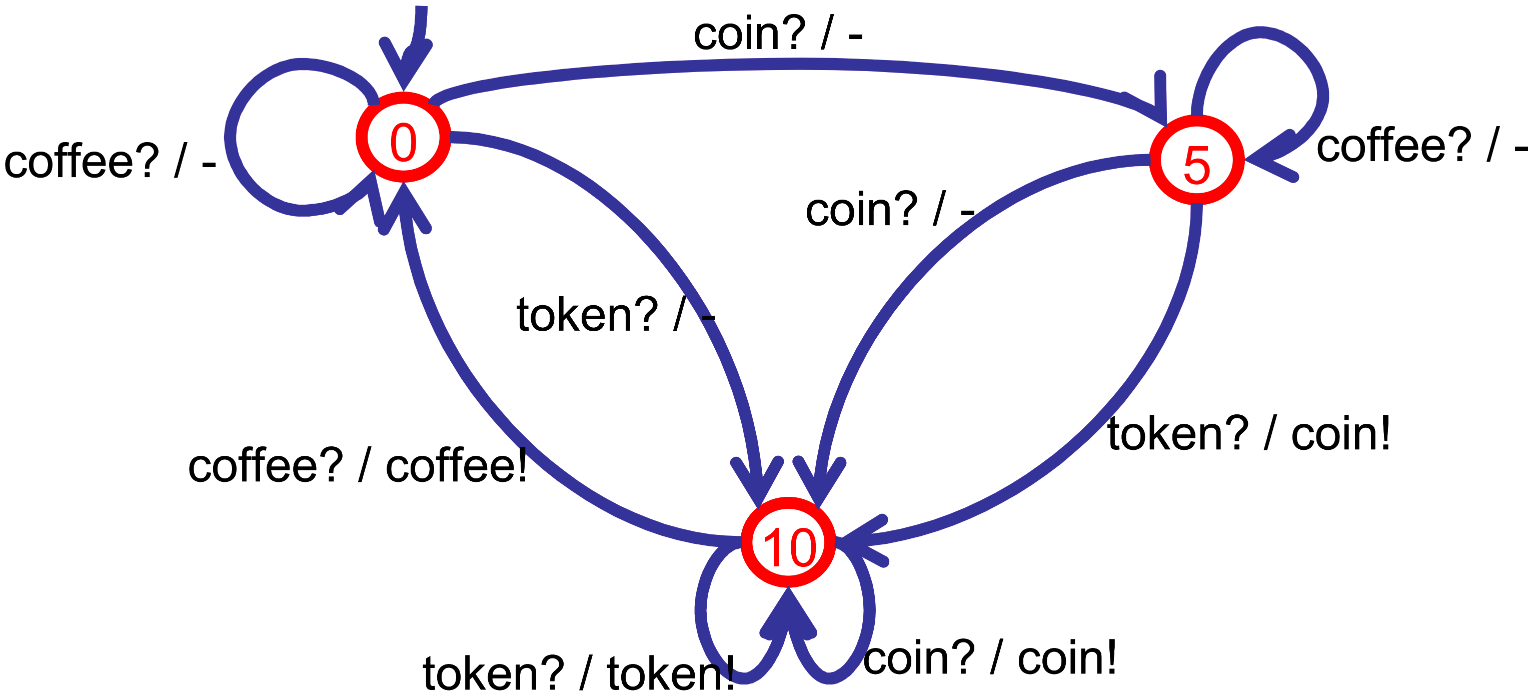


Test input sequence :

reset? coffee? coin? coffee? coin? coin? token? coffee? token? coffee? coin? token? coffee?

FSM Transition Tour

- Make Transition Tour that covers every transition (in spec)



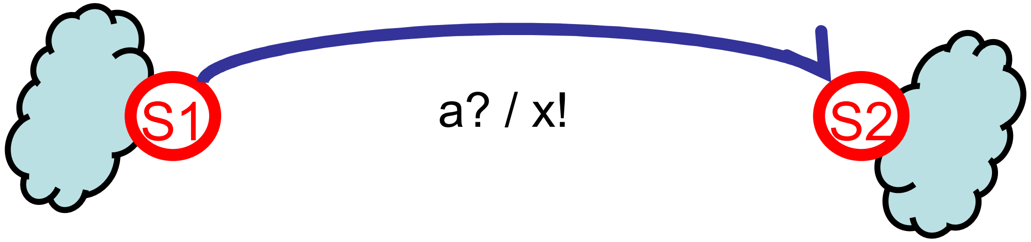
Test input sequence :

reset? coffee? coin? coffee? coin? coin? token? coffee? token? coffee? coin? token? coffee?

+ check expected output and target state by “status” message

FSM Transition Testing

- Make test case for every transition in SPEC separately:



- Test transition "S1 --a?/x!--> S2":
  1. Go to state S1
  2. Apply input a?
  3. Check output x!
  4. Verify state S2 (optionally)

- Test purpose: "Test whether the system, when in state S1, produces output x! on input a? and goes to state S2"

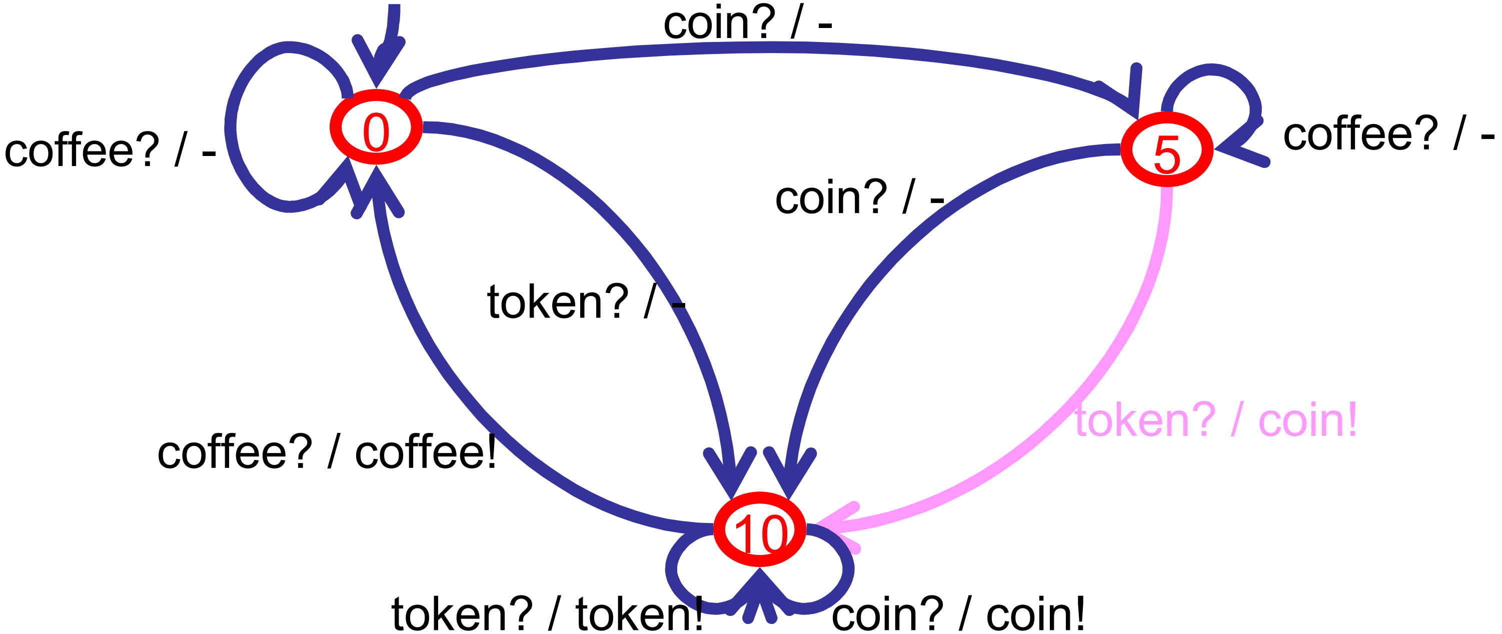
Transition Testing - 1

- To test **token? / coin!** :

  go to state 5 : set_state(5)

  give input token? check output coin!

  verify state: status? currentState=10



Test case : set_state(5)/ * - token? / coin! - status? / 10!

Transition Testing - 1

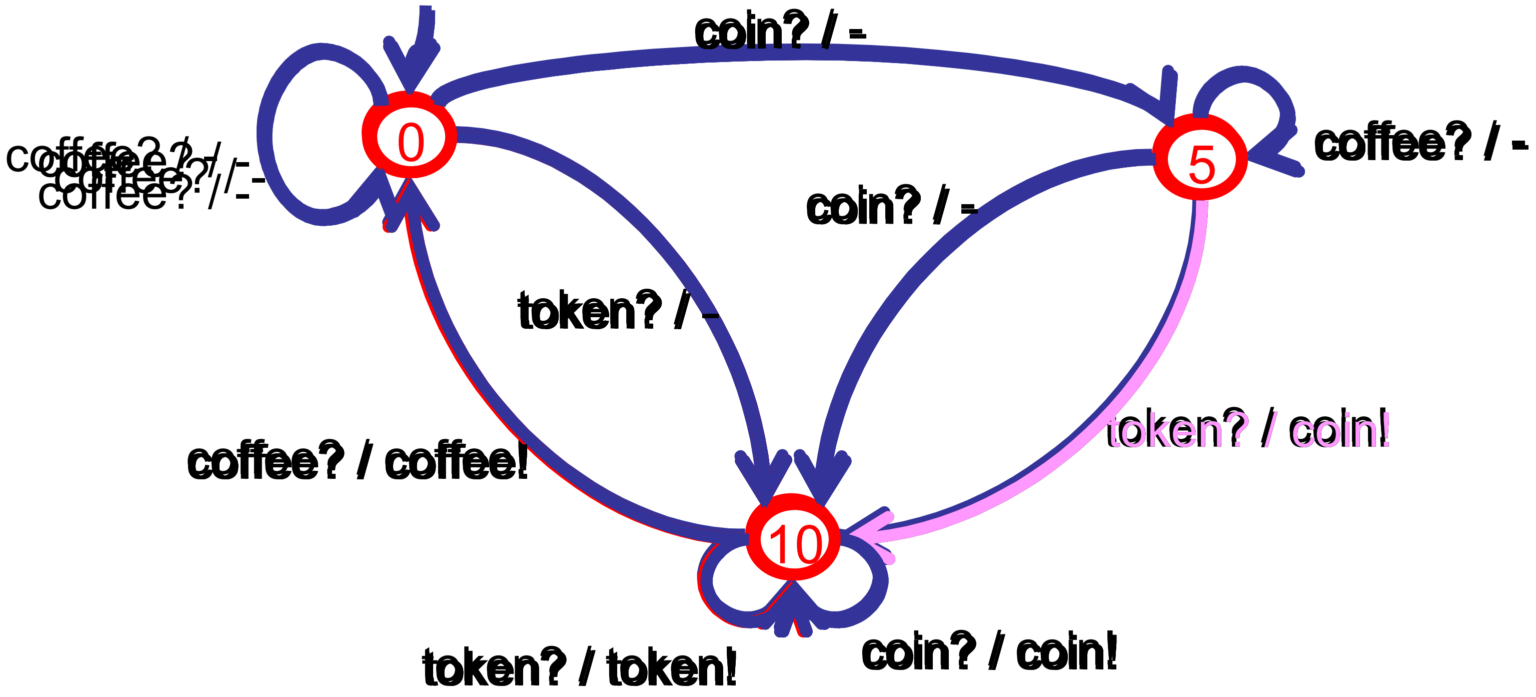
- "go to state S5" depends on the "set_state()" method

- What if no "set_state()" method available?
  - use the "reset" method if available
    - go from S0 to S5 (always possible because of determinism and completeness)
  - or, use synchronizing sequence to bring machine to a particular known state, say S0, from any state
    - (but synchronizing sequence may not exist 😠)

A **synchronizing sequence** of state s brings the FSM from **any** state to state s.

Transition Testing - 1

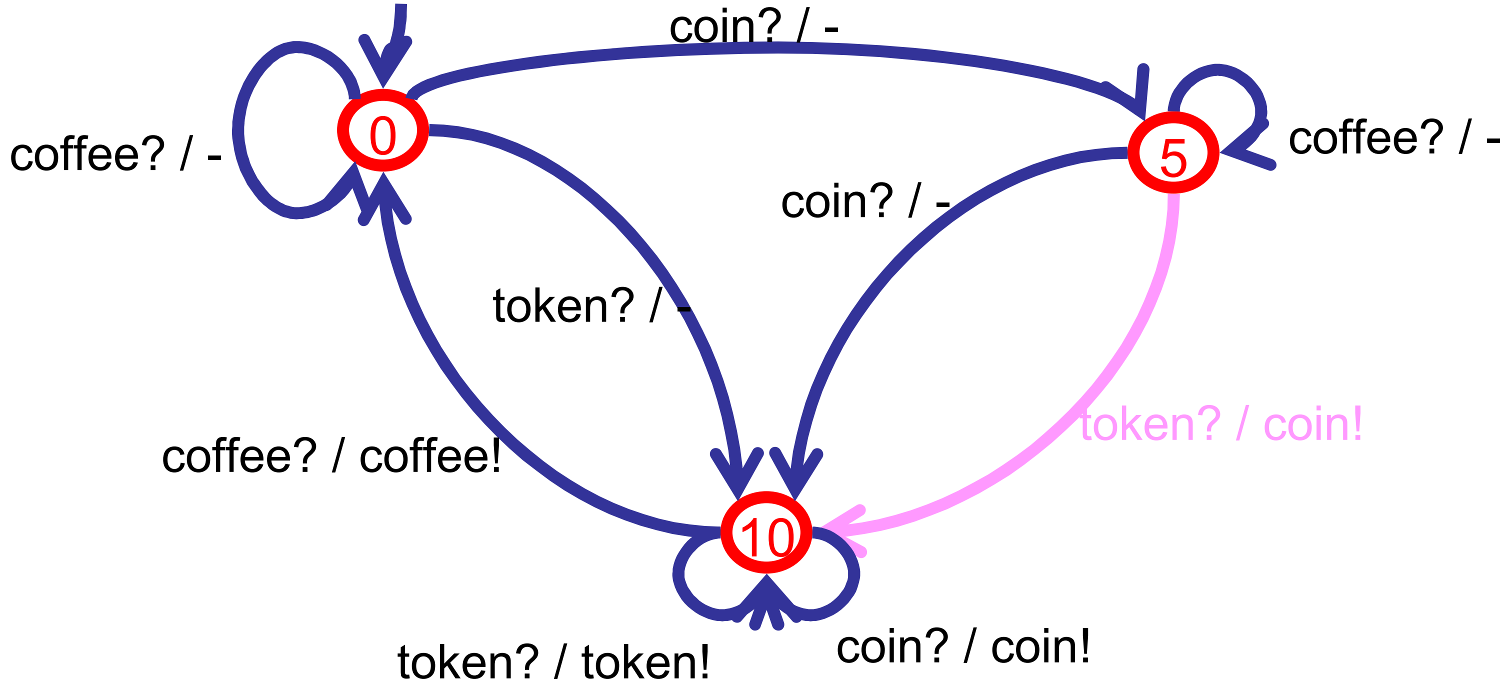
synchronizing sequence : token? coffee?



To test token? / coin! : go to state 5 by : token? coffee? coin?

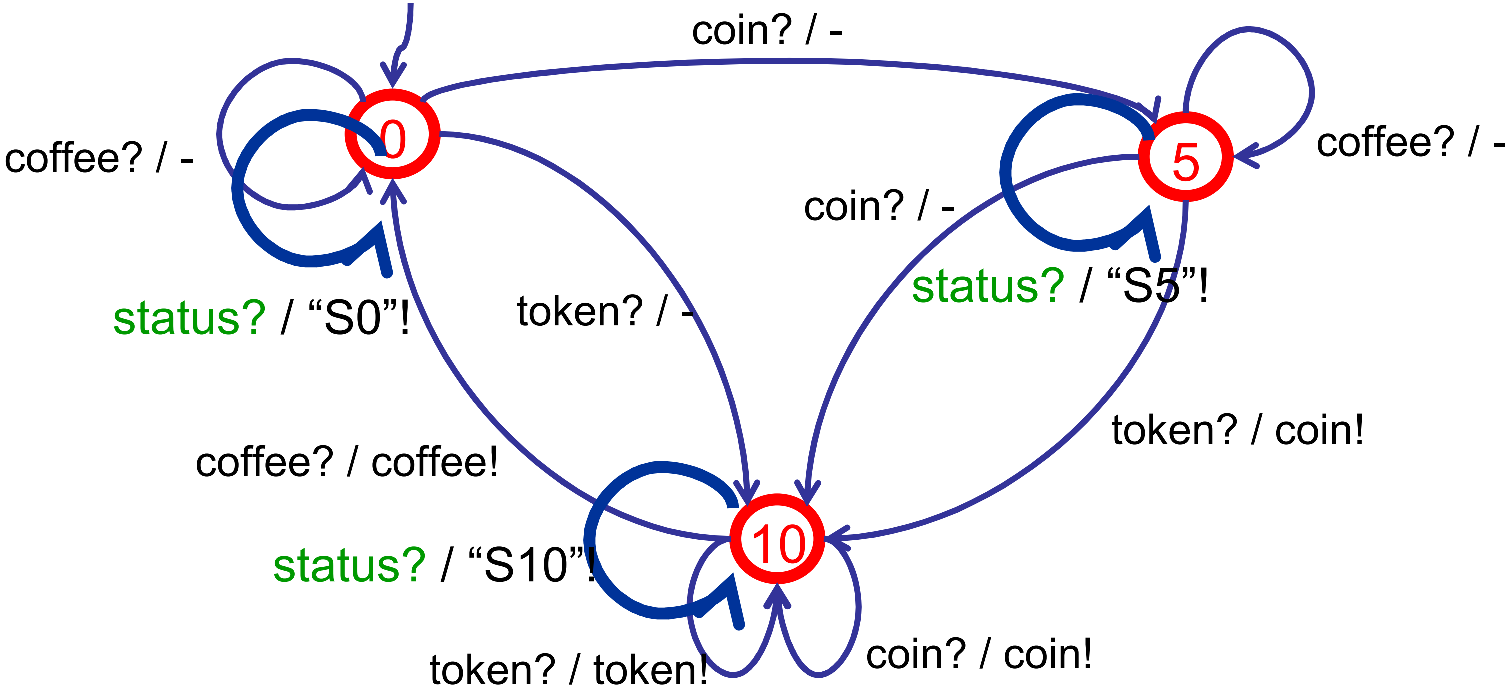
Transition Testing - 2

- To test token? / coin! :
  1. go to state 5 by : "token? coffee? coin?"
  2. give input token?
  3. check output coin!
  4. verify that machine is in state 10 by: "status? currentState=10!"



Transition Testing - 2

"status" message: Assume that tester can ask implementation for its current state (reliably!!)



Transition Testing - 2

- No “status” message??
  - State identification: **What state am I in?**
  - **State verification**: **Am I in state s?**
  - Apply sequence of inputs in the current state of the FSM such that from the outputs we can
    - identify that state where we started (state identification), or
    - verify that we were in a particular start state (**state verification**)
  - Different kinds of sequences
    - UIO sequences (Unique Input Output sequence)
    - Distinguishing sequence (DS)
    - W-set (characterizing set of sequences)
    - UIOv
    - SUIO
    - MUIO
    - Overlapping UIO

Transition Testing - 2

UIO: each state has its own input sequence that produces different outputs when applied in other states.

State check :

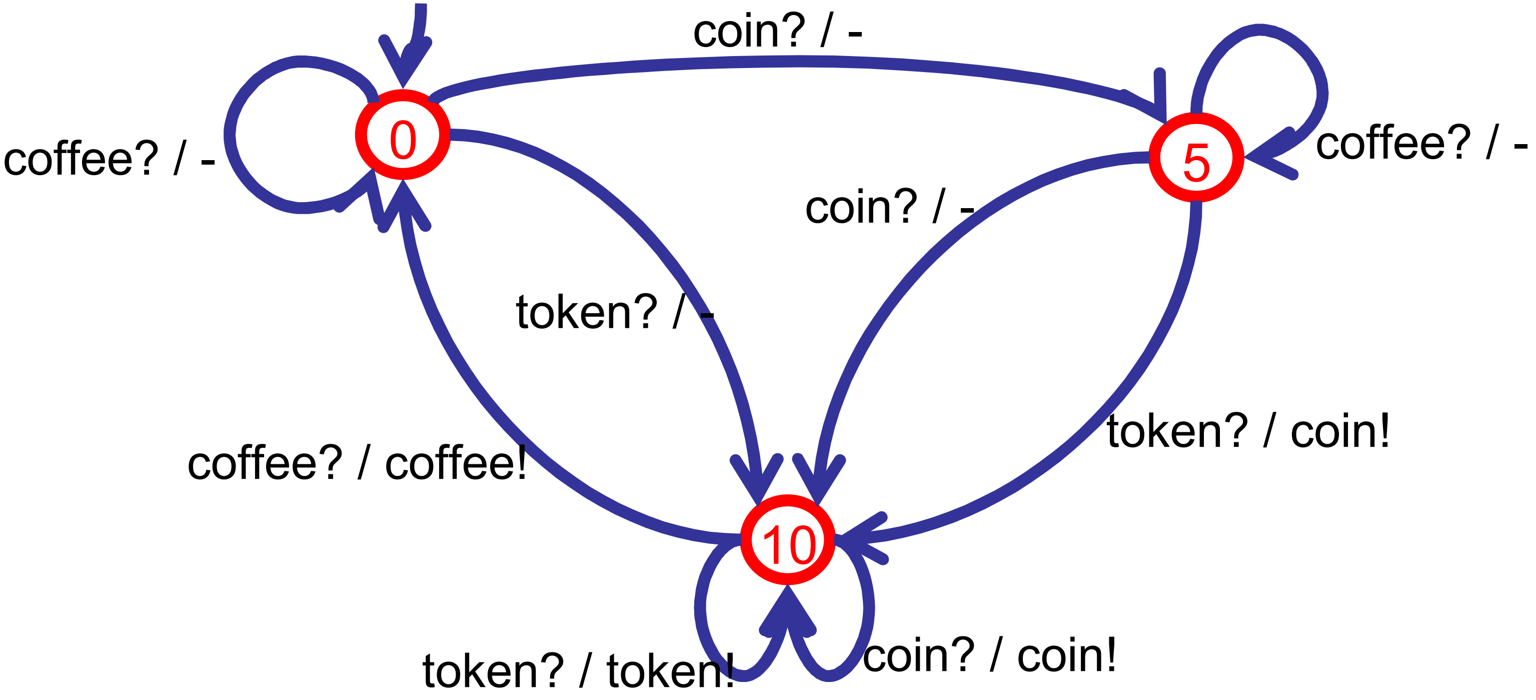
- UIO sequences (verification)
  - sequence x_s that distinguishes state s from all other states : for all $t \neq s$: $\lambda(s, x_s) \neq \lambda(t, x_s)$
  - each state has its own UIO sequence
  - UIO sequences may not exist

DS: special UIO such that it is a UIO for all states!!

- Distinguishing Sequence (identification)
  - sequence x that produces different output for <u>every</u> state : for all pairs t, s with $t \neq s$: $\lambda(s, x) \neq \lambda(t, x)$
  - a distinguishing sequence may not exist
- W - set of sequences (identification)
  - set of sequences W which can distinguish any pair of states : for all pairs $t \neq s$ there is $x \in W$: $\lambda(s, x) \neq \lambda(t, x)$
  - W - set always exists for reduced FSM

Transition Testing- 2: UIO

UIO: each state has its own input sequence that produces different outputs when applied in other states.

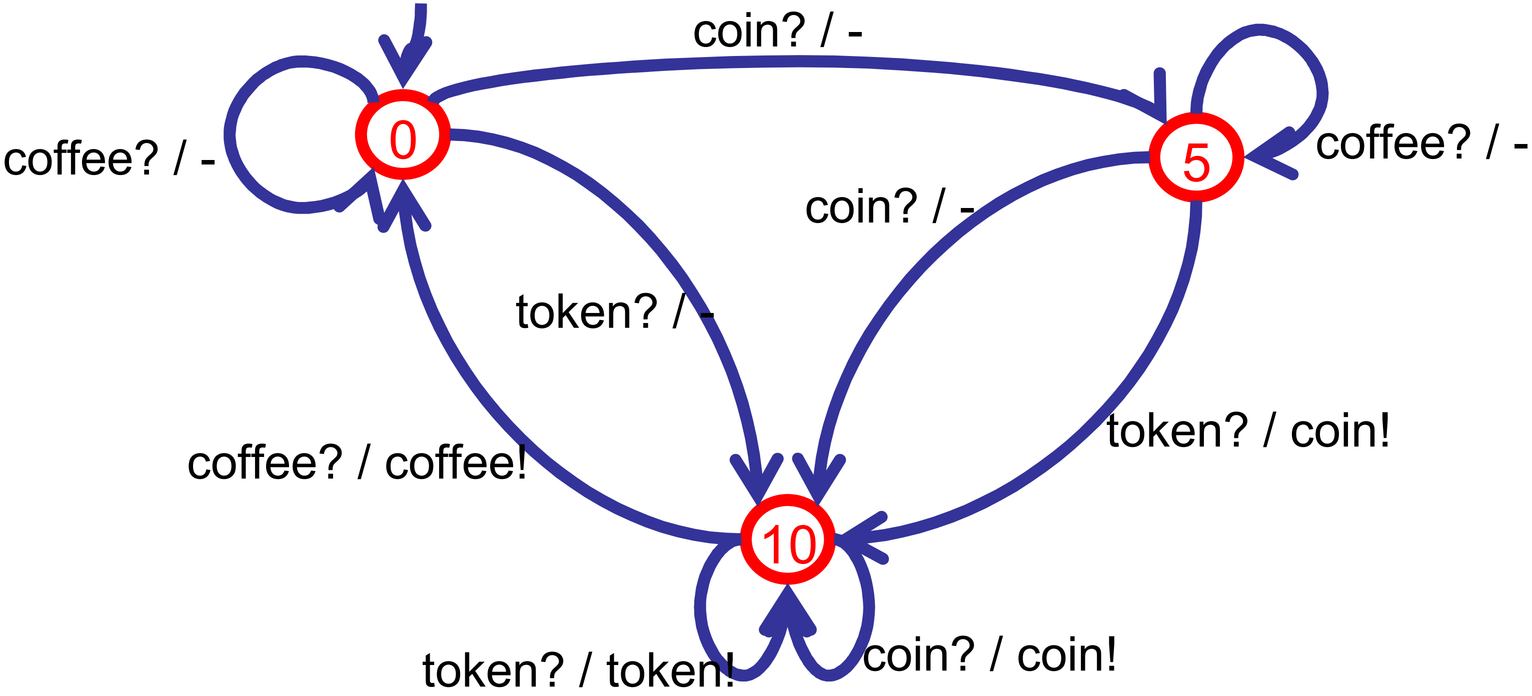


UIO sequences

state 0 :	coin? / - coffee? / -
state 5 :	token? / coin!
state 10 :	coffee? / coffee!

Transition Testing- 2: DS

DS: special UIO such that it is a UIO for all states!!16



DS sequence: token?	
	output state 0 : -
	output state 5 : coin!
	output state 10 : token!

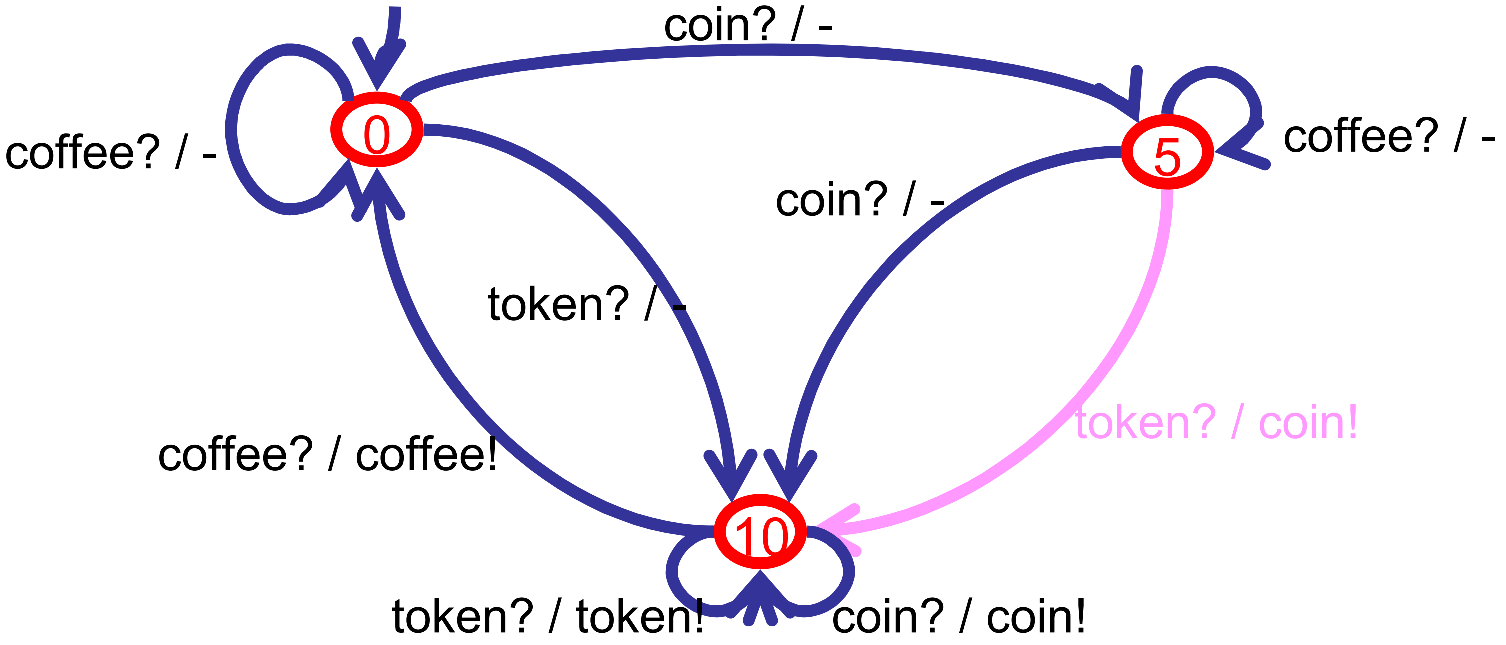
Transition Testing - 2: done

- To test token? / coin! :

  go to state 5 : token? coffee? coin?

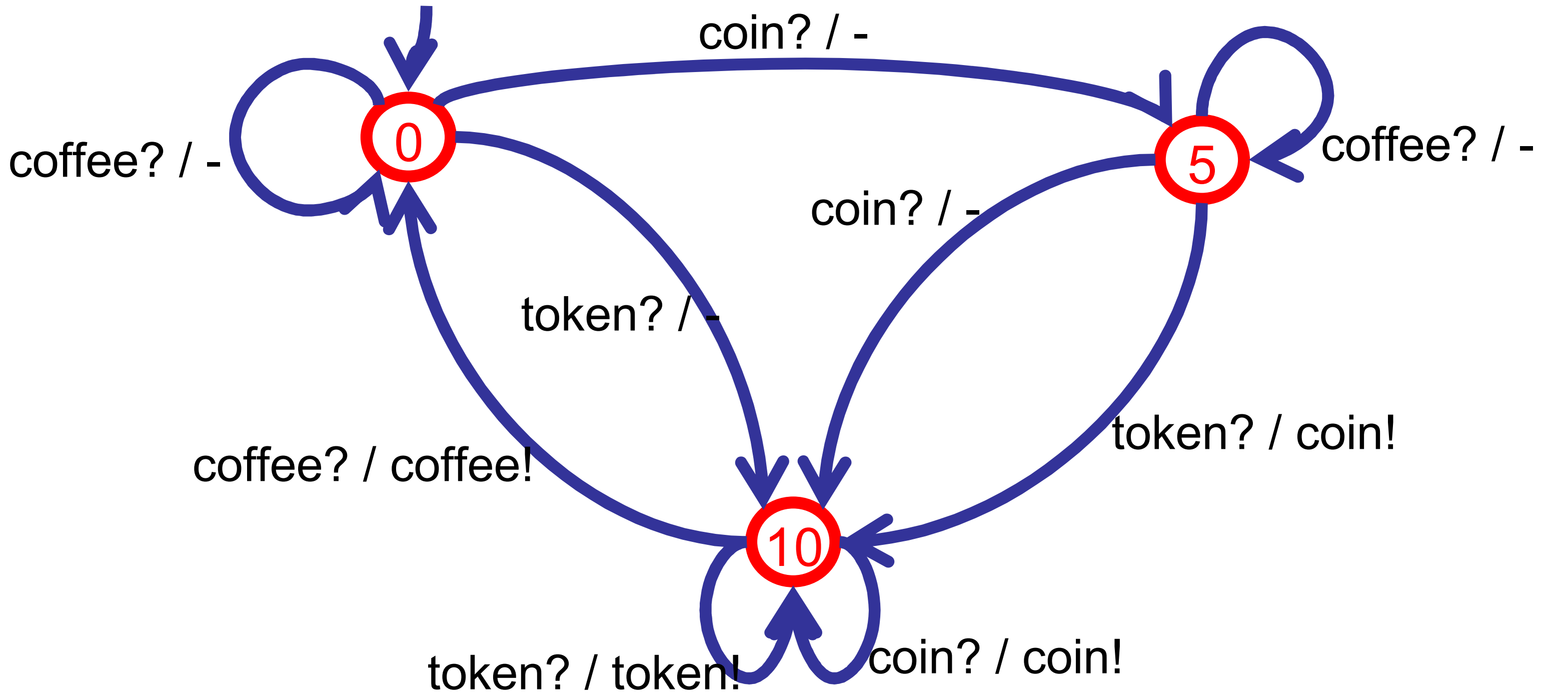
  give input token? check output coin!

  apply UIO of state 10 : coffee? / coffee!



Test case : token? / * coffee? / * coin? / - token? / coin! coffee? / coffee!

Transition Testing - done



- 9 transitions / test cases for coffee machine
- if end-state of one test case corresponds with start-state of next test case then concatenate
- different ways to optimize and remove overlapping / redundant parts
- there are (academic) tools to support this

FSM Transition testing: further results

- Test transition “S1 –-a?/x!--> S2”:
  1. Go to state S1
  2. Apply input a?
  3. Check output x!
  4. Verify state S2
- Checks every output fault and transfer fault (to existing state)
- **If** we assume that

  the number of states of the implementation machine M_I

  is less than or equal to

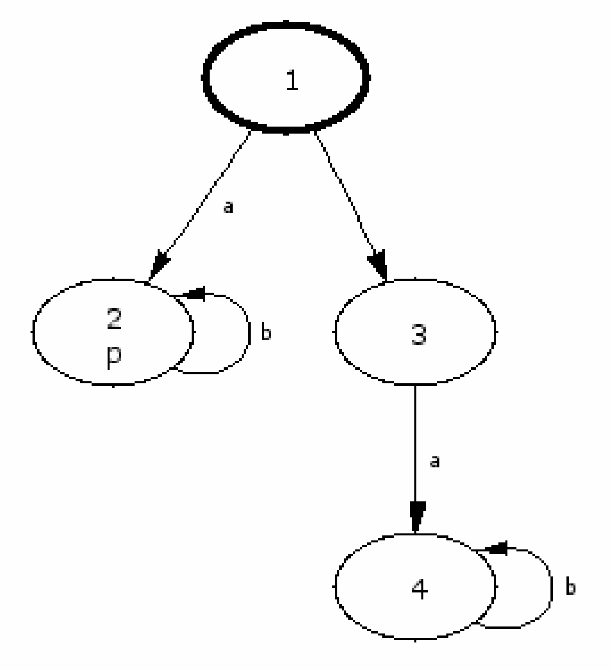
  the number of states of the specification machine M_S,

  then testing all transitions in this way

  leads to equivalence of reduced machines,

  i.e., **complete conformance**
- If not: exponential growth in test length in number of extra states in M_I.

Labelled Transition System (LTS)-Based Tetsing



Labelled Transition Systems

- Labelled Transition System (LTS)
  - Transition system labelled with (input, output, or internal) actions
  - A very basic model for describing system behavior

- Different from FSM
  - FSM is required to be "deterministic" and "complete"
  - FSM has always alternation between inputs and outputs

    though sometimes they may be "-"

  - LTS is more fundamental, more naive and simpler
  - LTS better supports the descriptions of non-determinancy, concurrency and composition
  - LTS serves as underlying semantics model for many other formalisms (including timed models)

An example LTS

Labelled Transition System $\langle S, L, T, s_0 \rangle$

states

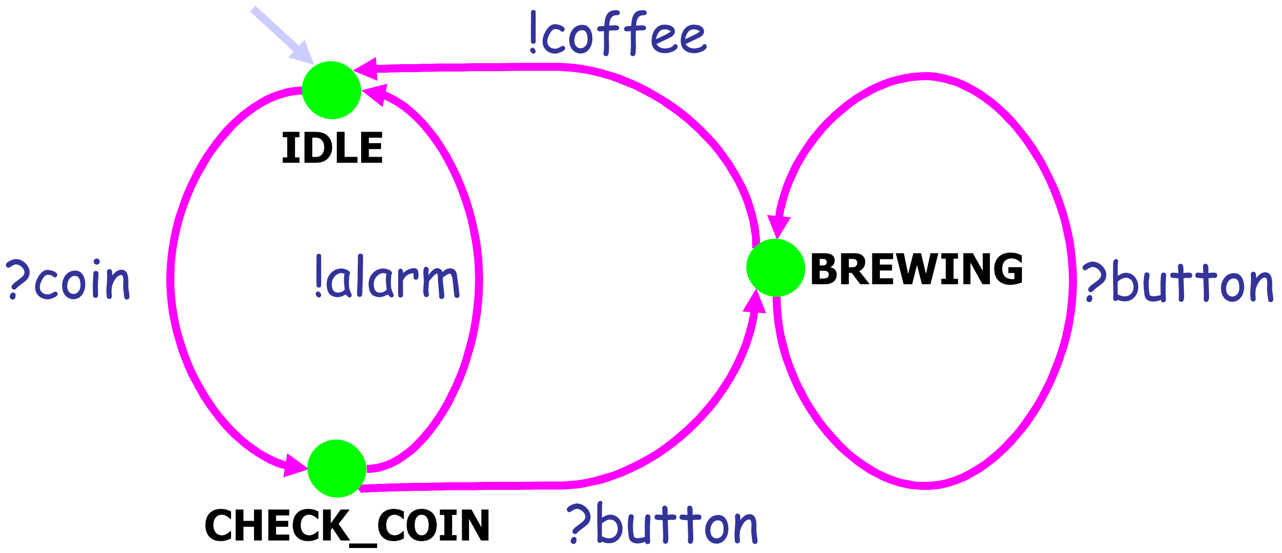
actions

transitions

$T \subseteq S \times (L \cup \{\tau\}) \times S$

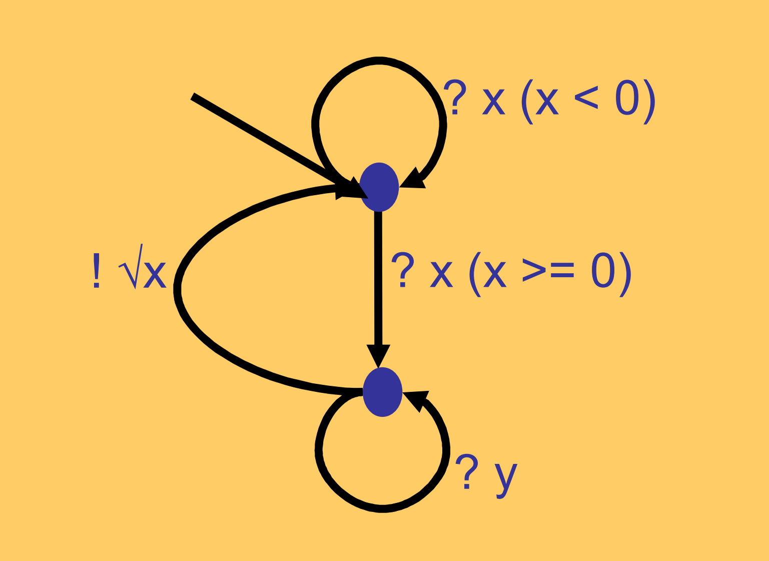
initial state

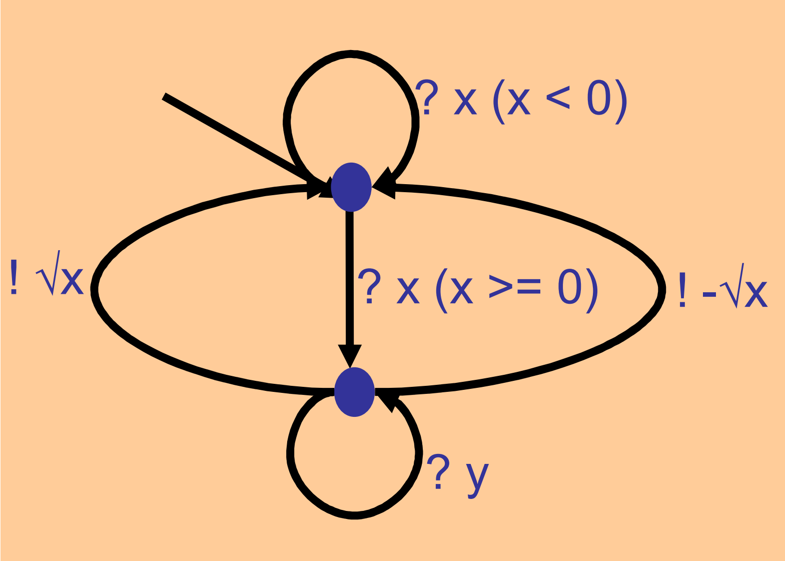
$s_0 \in S$



Input-Output LTS (IOLTS)

- Special kind of LTS:
  Input-Output Labelled Transition System - IOLTS
  - distinction between outputs (!) and always-enabled inputs (?)
  - implementations modelled as IOLTS
- IOLTS with variables - equation solver for $y^2 = x$:



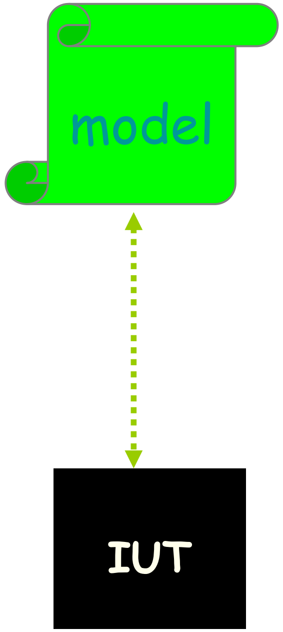


Conformance Relation

- Assume that the Implementation Under Test (IUT) is a black box
  - The internal state and internal actions of IUT are unobservable
  - We can observe the external actions of IUT from its interface

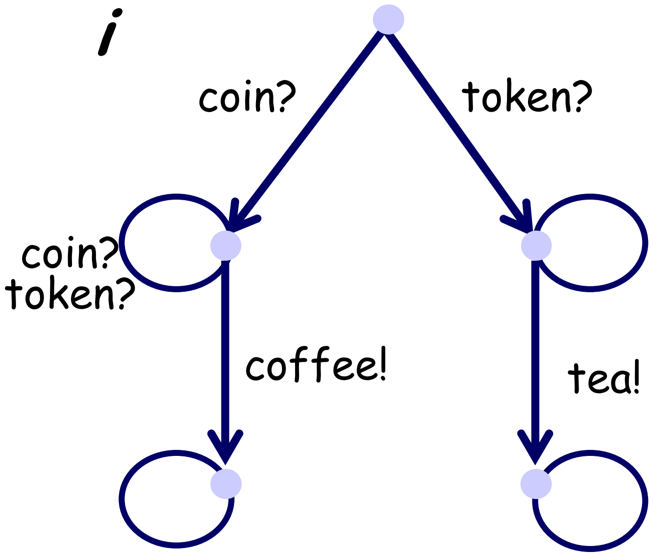
- Whether the behavior of IUT conforms to those specified by the specification model?

- input/output conformance ("ioco")
  - for the IUT:
    - do what are **required** to do, and
    - never do what are **forbidden** to do

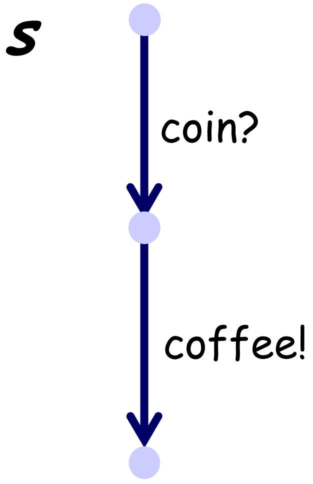


i conforms-to s ?? (a)

Implementation Under Test



Specification

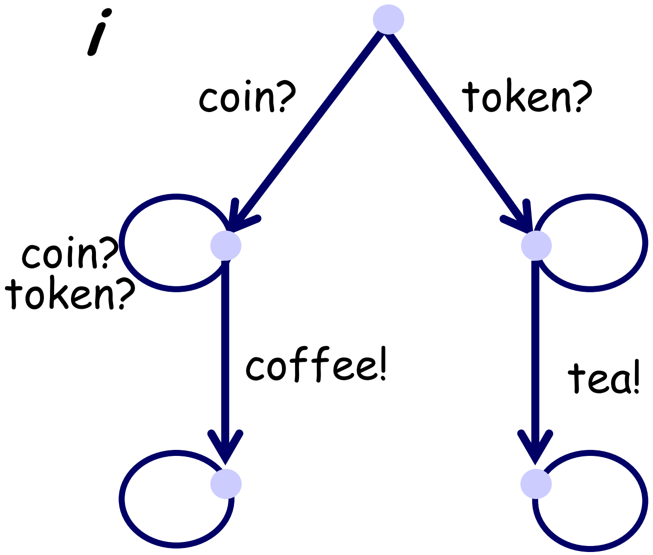


ioco ✓

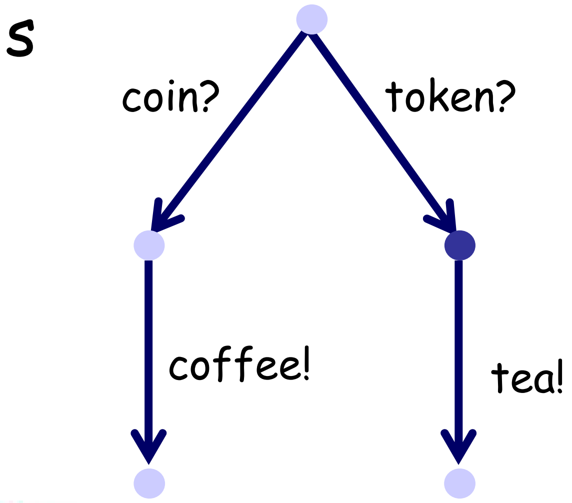
[Jan Tretmans]

i conforms-to s ?? (b)

Implementation Under Test



Specification

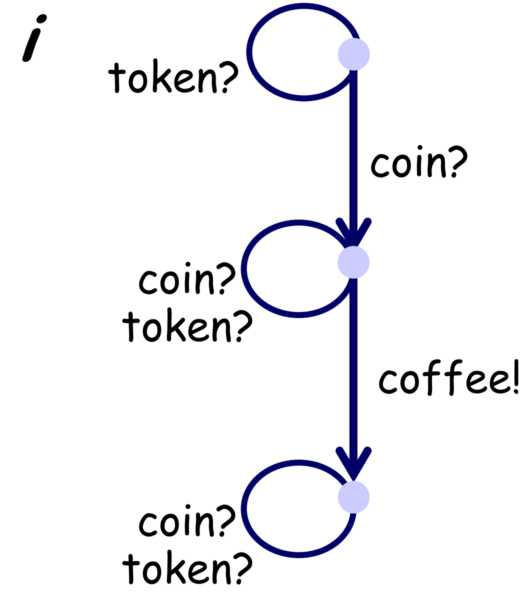


ioco ✓

[Jan Tretmans]

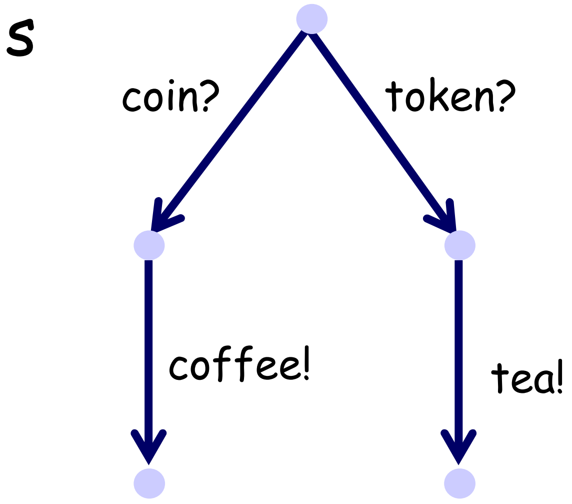
i conforms-to s ?? (c)

Implementation Under Test



~~ioco~~

Specification

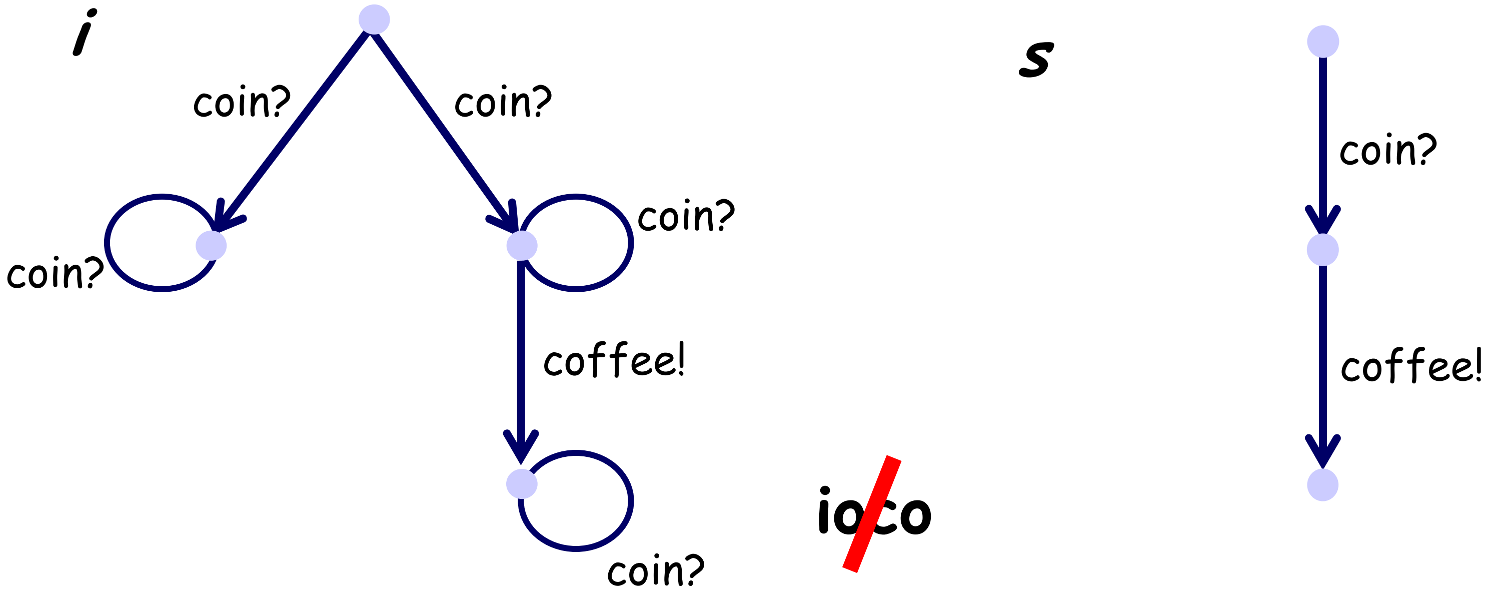


[Jan Tretmans].

i conforms-to s ?? (d)

Implementation Under Test

Specification



[Jan Tretmans].

Tretman's ioco-coformance

The conformance relation widely used for black-box LTS-based testing of (untimed) reactive systems

$$\mathbf{i}\ \mathbf{ioco}\ \mathbf{s} \ =_{\mathrm{def}}\ \forall \sigma \in \textbf{\textit{Straces}}\,(\mathbf{s}) :\ \textbf{\textit{out}}\,(\mathbf{i\ after}\ \sigma) \subseteq \textbf{\textit{out}}\,(\mathbf{s\ after}\ \sigma)$$

$$\textbf{\textit{Straces}}\,(s) \quad = \quad \{\ \sigma \in (L \cup \{\delta\})^{*} \mid s \xRightarrow{\sigma}\ \}$$

$$p\ \mathbf{after}\ \sigma \quad = \quad \{\ p' \mid p \xRightarrow{\sigma} p'\ \}$$

$$p \xrightarrow{\delta} p \quad \text{iff} \quad \forall\ o! \in L_U \cup \{\tau\} :\ p \not\xrightarrow{o!}$$

L_u is the subset of output actions of L

$$\textbf{\textit{out}}\,(\ P\ ) \quad = \quad \{\ o! \in L_U \mid p \xrightarrow{o!},\ p \in P\ \}$$
$$\cup\ \{\ \delta \mid p \xrightarrow{\delta} p,\ \ p \in P\ \}$$

[Jan Tretmans].

ioco: intuitively

$i \ \mathbf{ioco} \ s \ =_{def} \ \forall \sigma \in \mathit{Straces}(s) : \ \mathit{out}(i \ \mathbf{after} \ \sigma) \subseteq \mathit{out}(s \ \mathbf{after} \ \sigma)$

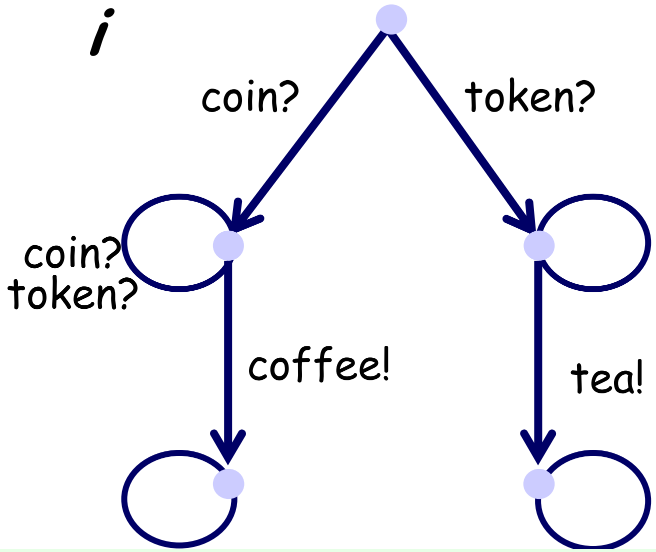
Intuition:

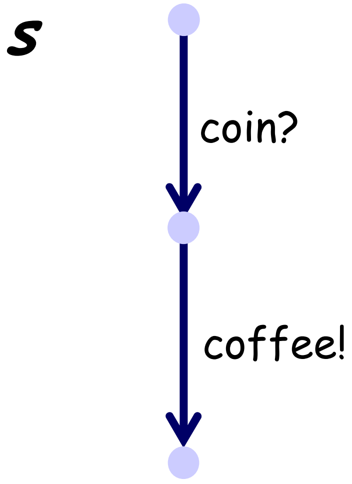
i **ioco**-conforms to s, iff

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

ioco-conformance (a)

$i \; \textbf{ioco} \; s \;\; =_{def} \;\; \forall \sigma \in \mathit{Straces}(s) : \;\; \mathit{out}(i \; \textbf{after} \; \sigma) \; \subseteq \; \mathit{out}(s \; \textbf{after} \; \sigma)$





out (*i* **after** coin?) = { coffee! }

out (*i* **after** token?) = { tea! }

out (*s* **after** coin?) = { coffee! }

out (*s* **after** token?) = ∅

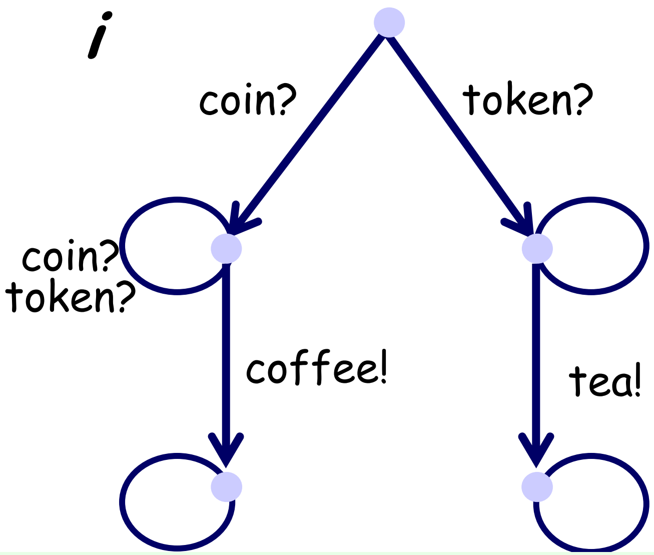
But token? ∉ *Straces* (*s*)

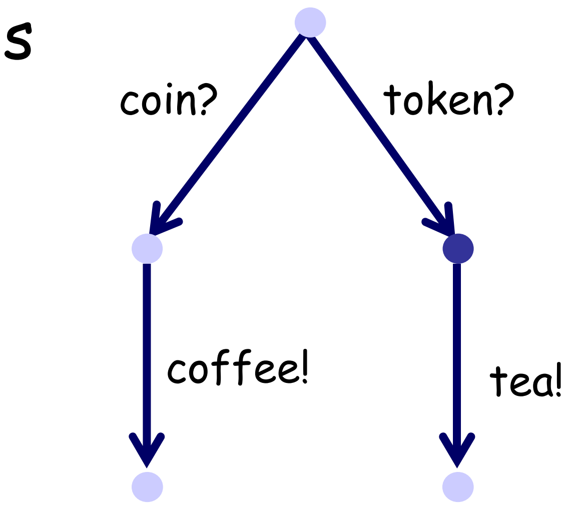
ioco ✓

[Jan Tretmans]

ioco-conformance (b)

i **ioco** s $=_{def}$ $\forall\sigma \in$ *Straces* (s) : *out* (i **after** σ) $\subseteq$ *out* (s **after** σ)





out (*i* **after** coin?) = { coffee! }

out (*i* **after** token?) = { tea! }

out (*s* **after** coin?) = { coffee! }

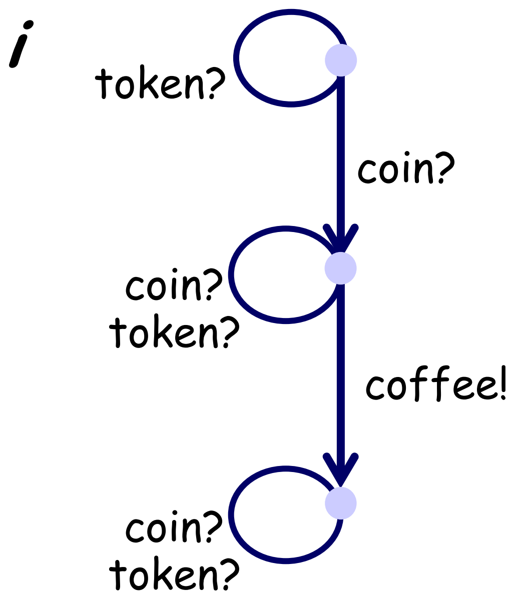
out (*s* **after** token?) = { tea! }

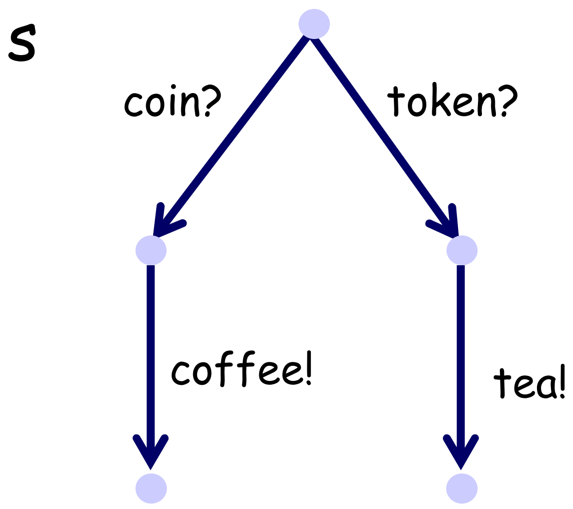
ioco ✓

[Jan Tretmans].

ioco-conformance (c)

i **ioco** s $=_{def}$ $\forall\sigma \in$ *Straces* (s) : *out* (i **after** σ) $\subseteq$ *out* (s **after** σ)





out (*i* **after** token?) = { δ }

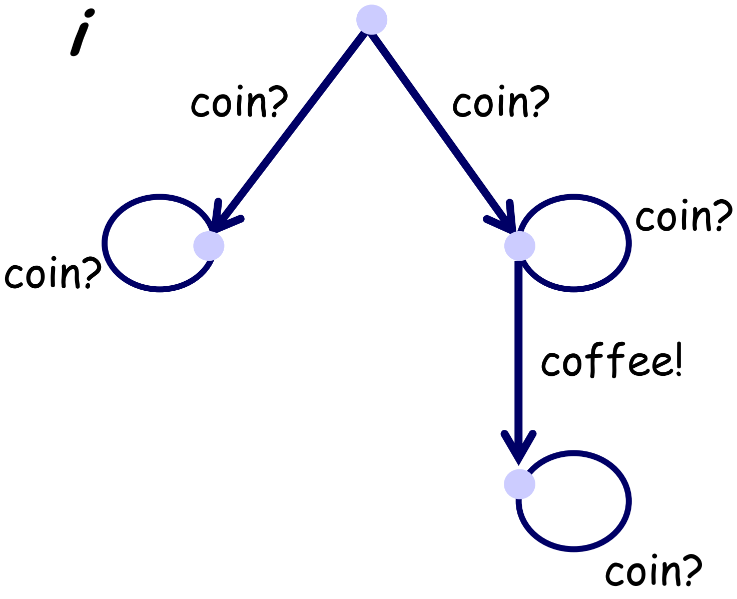
out (*s* **after** token?) = { tea! }

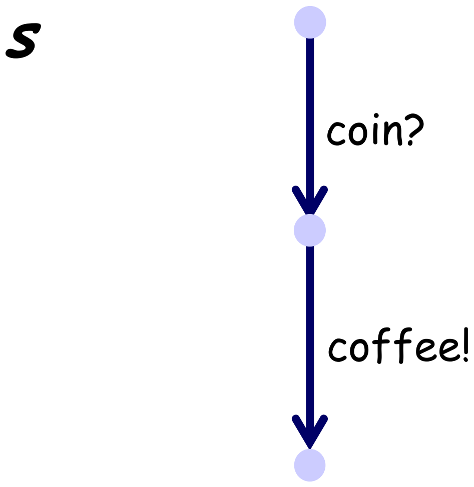
$\cancel{\textbf{ioco}}$

[Jan Tretmans].

ioco-conformance (d)

i **ioco** s $=_{def}$ $\forall \sigma \in$ *Straces* (s) : *out* $(i$ **after** $\sigma) \subseteq$ *out* $(s$ **after** $\sigma)$





out (*i* **after** coin?) = { δ, coffee! }

out (*s* **after** coin?) = { coffee! }

$\not\text{ioco}$

[Jan Tretmans].

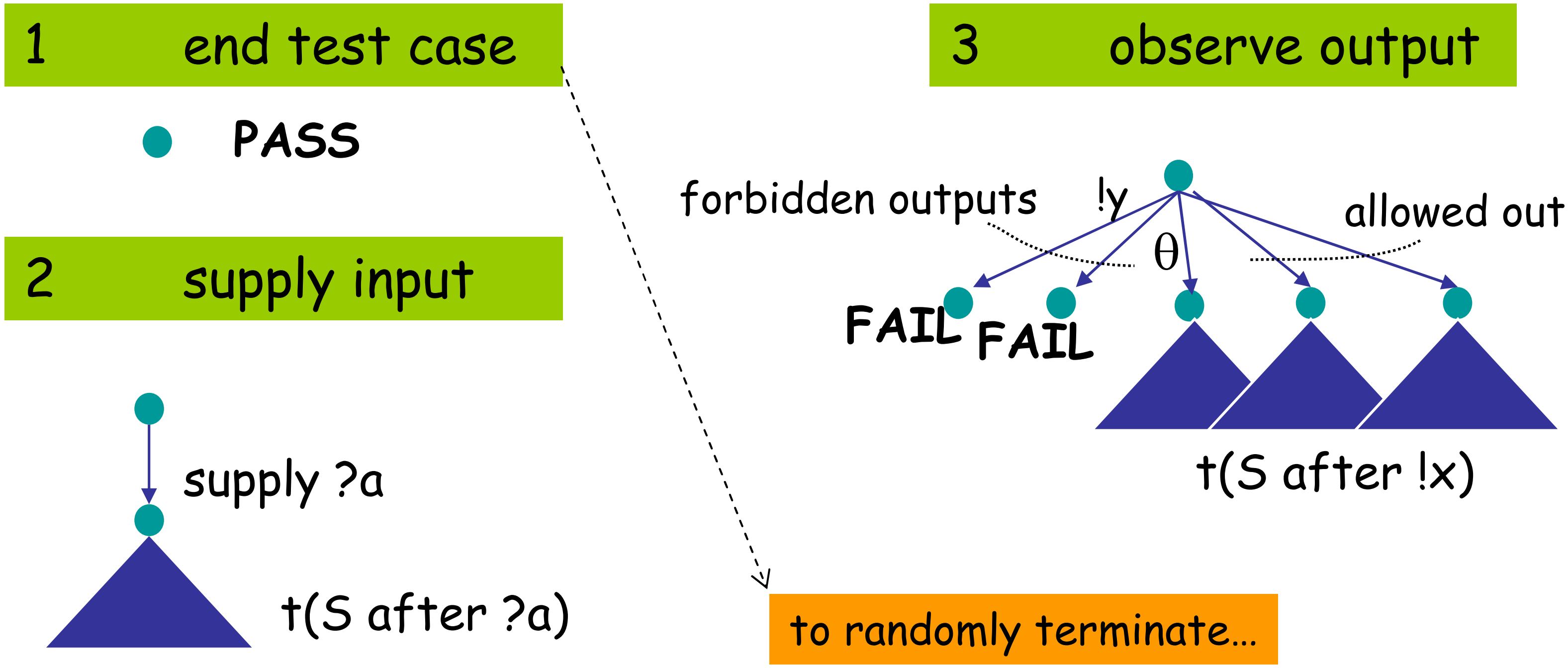
Test Generation Algorithm

Objective: To generate a test case t(S) from a transition system specification.

// Here S is a set of states (initially S = $\{s_0\}$)

Algorithm:

Apply the following steps recursively, non-deterministically



1 end test case

PASS

2 supply input

supply ?a

t(S after ?a)

3 observe output

forbidden outputs !y

allowed outputs !x

θ

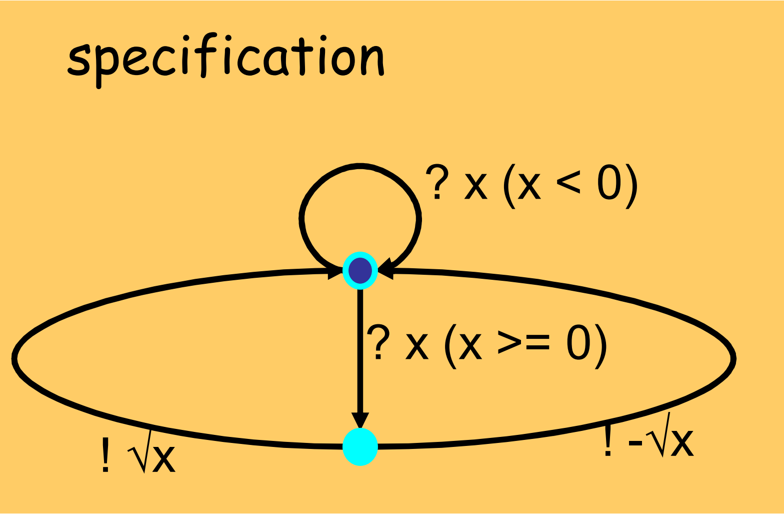
FAIL FAIL

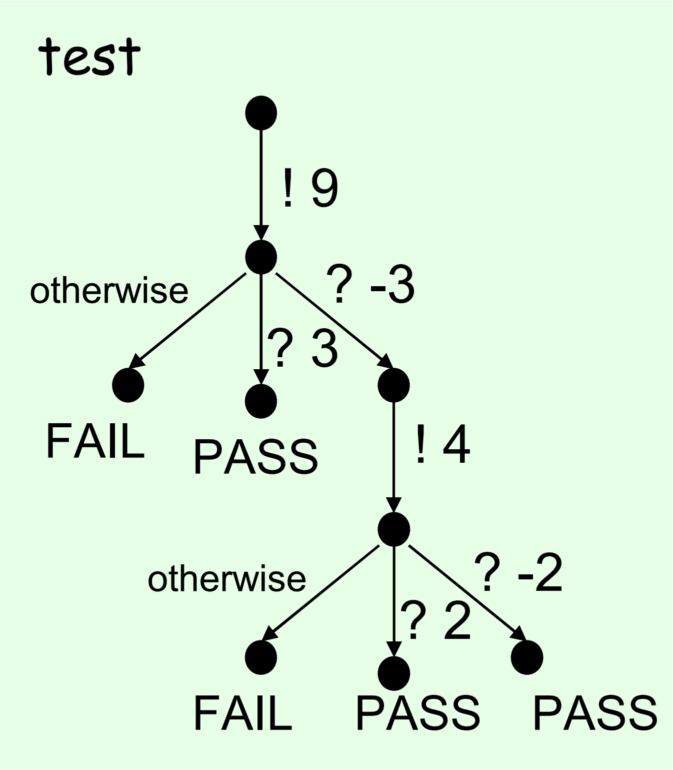
t(S after !x)

to randomly terminate…

Test Generation Example

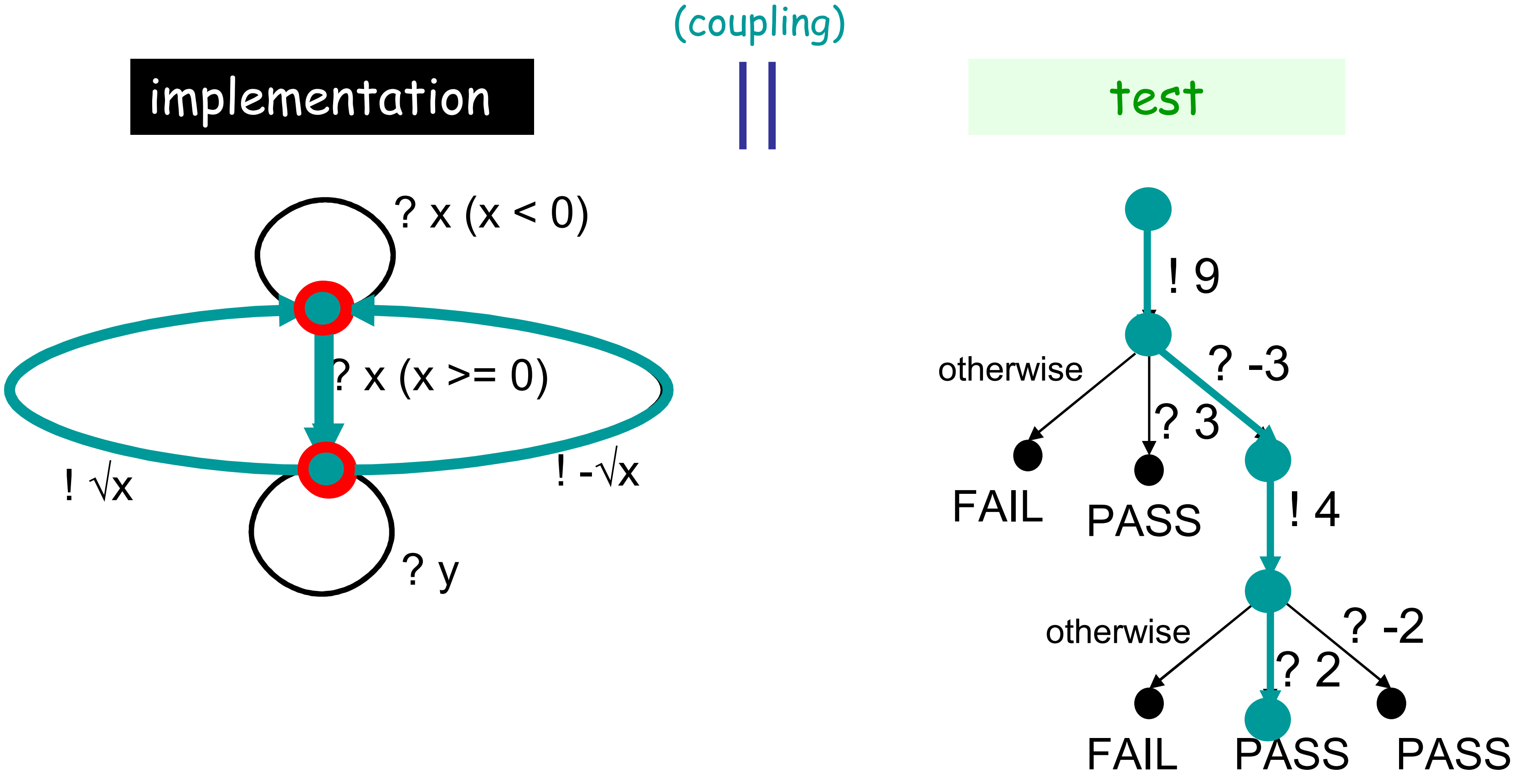
Equation solver for $y^2=x$





To cope with non-deterministic behaviour, tests are not linear traces, but trees

Test Execution Examples



Validity of Test Generation

For every test t generated with the algorithm:

Soundness :

- t will never fail with correct implementation

i ioco s implies i passes t

or: i fails t implies i not(ioco) s

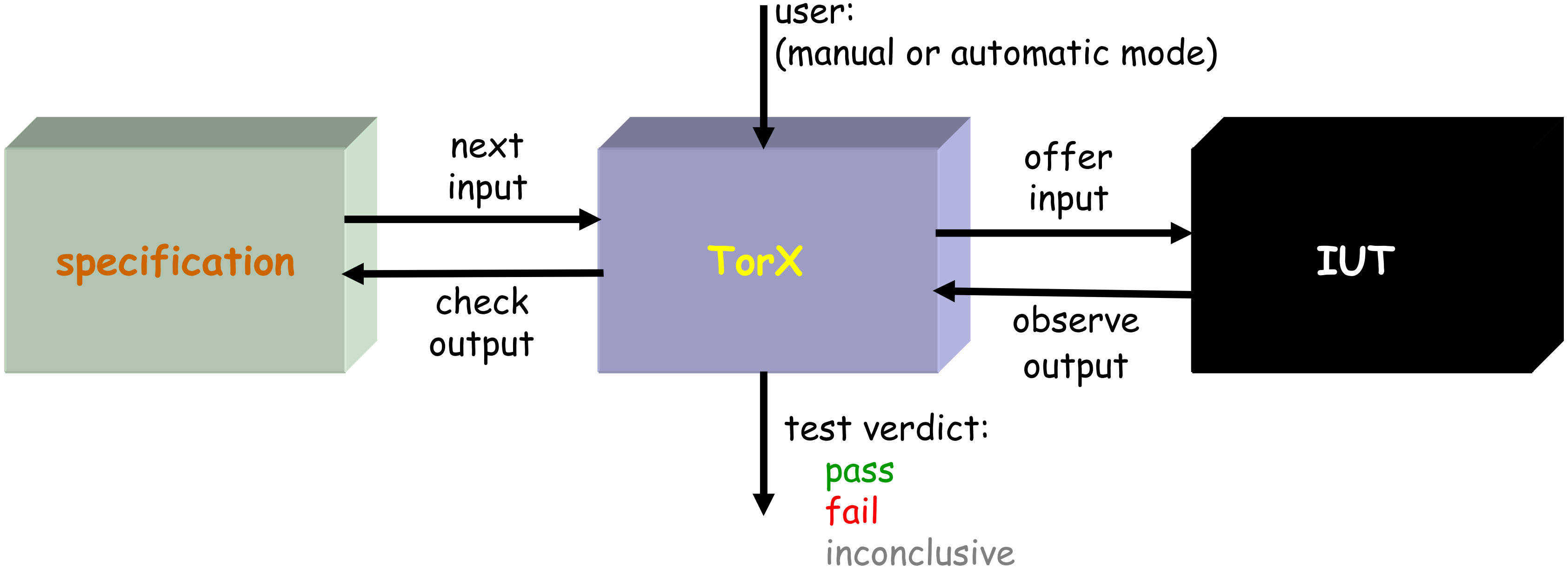
Exhaustiveness :

- each incorrect implementation can be detected with a generated test t

i $\cancel{\text{ioco}}$ s implies $\exists t$: i fails t

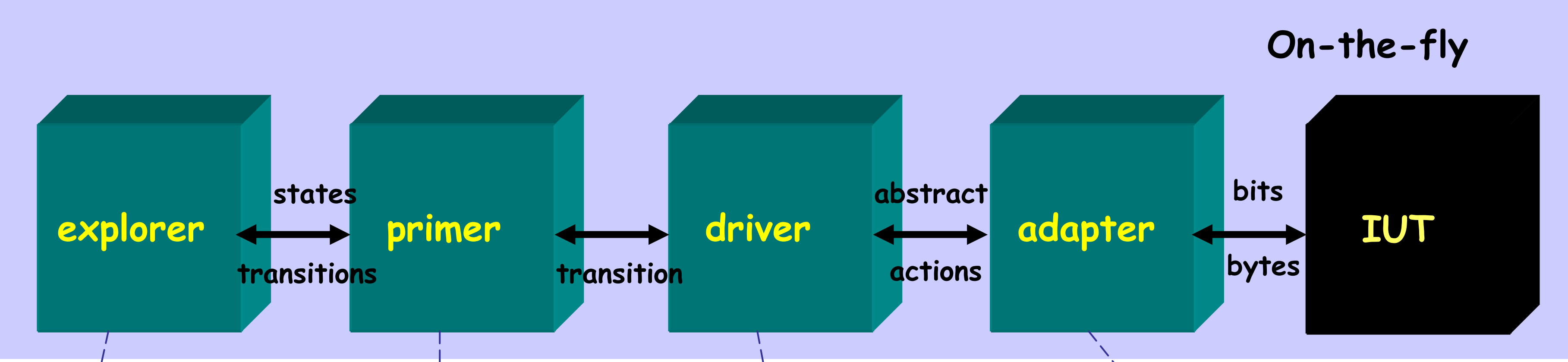
LTS Testing: The TorX Tool

- On-the-fly test generation and test execution
- Implementation relation: **ioco**
- Specification languages: LOTOS and Promela



TorX Tool Architecture

Concentrate on on-the-fly testing



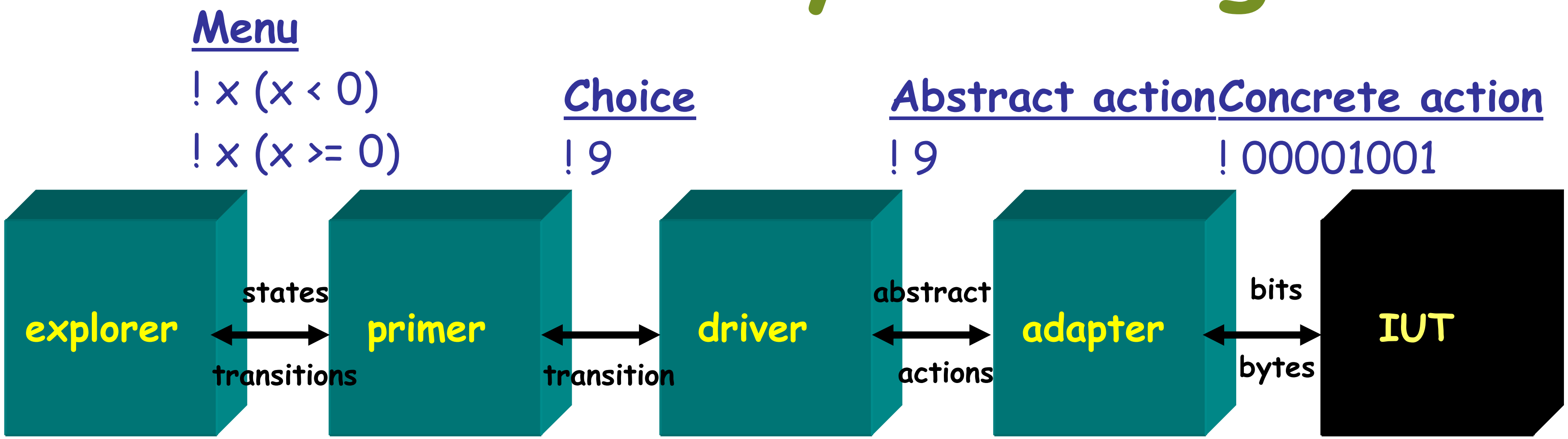
responsible for sending inputs to and receiving outputs from the IUT on request of the driver

to control the progress of the testing process

to implement the test derivation algorithm (to generate inputs for the implementation and to check outputs from the implementation)

to explore the transition-graph of the specification and to provide, for a given state, the set of transitions that are enabled in this state

On-The-Fly Testing



Menu

! x (x < 0)

! x (x >= 0)

Choice

! 9

Abstract action

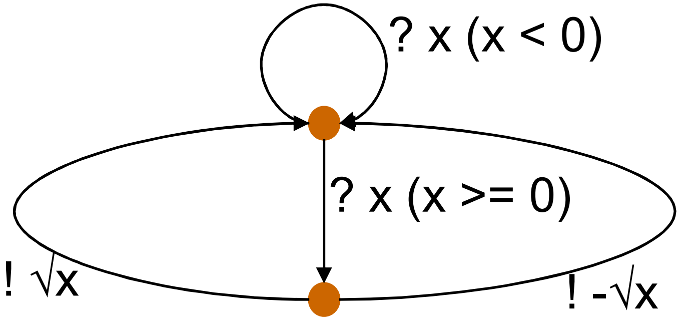
! 9

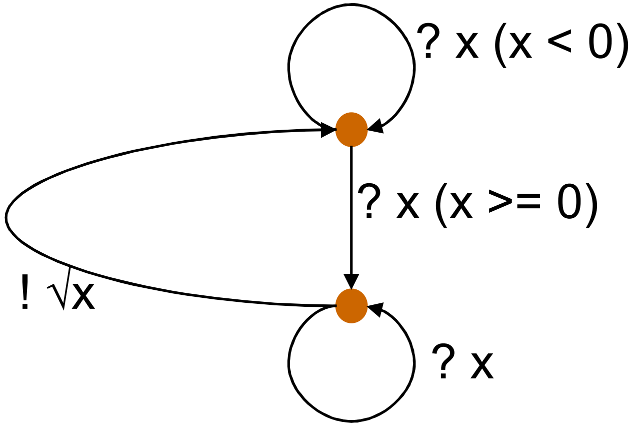
Concrete action

! 00001001

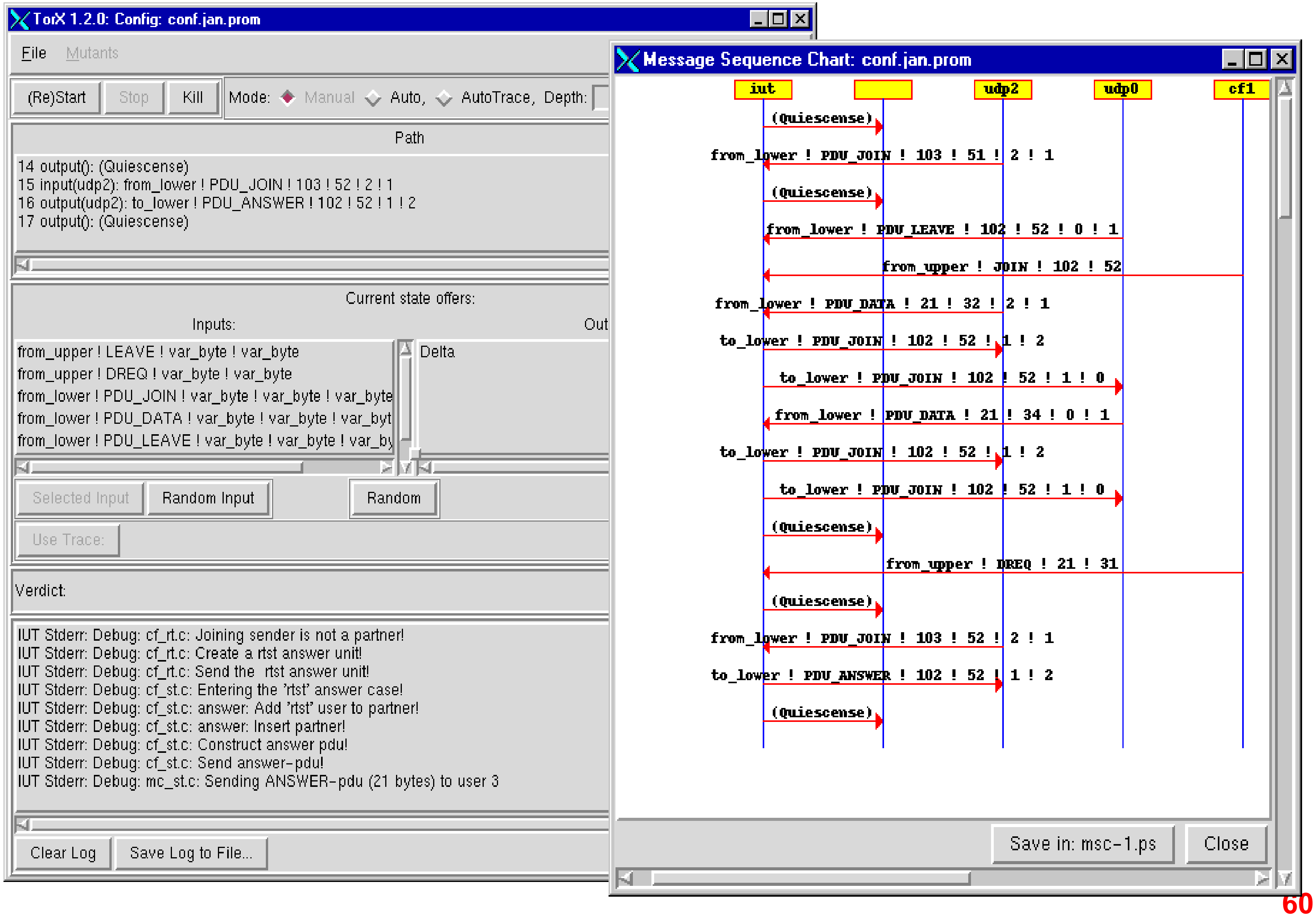
specification

implementation





TorX Screenshot



```
14 output(): (Quiescense)
15 input(udp2): from_lower ! PDU_JOIN ! 103 ! 52 ! 2 ! 1
16 output(udp2): to_lower ! PDU_ANSWER ! 102 ! 52 ! 1 ! 2
17 output(): (Quiescense)
```

```
from_upper ! LEAVE ! var_byte ! var_byte
from_upper ! DREQ ! var_byte ! var_byte
from_lower ! PDU_JOIN ! var_byte ! var_byte ! var_byte
from_lower ! PDU_DATA ! var_byte ! var_byte ! var_byt
from_lower ! PDU_LEAVE ! var_byte ! var_byte ! var_by
```

```
Delta
```

```
IUT Stderr: Debug: cf_rt.c: Joining sender is not a partner!
IUT Stderr: Debug: cf_rt.c: Create a rtst answer unit!
IUT Stderr: Debug: cf_rt.c: Send the  rtst answer unit!
IUT Stderr: Debug: cf_st.c: Entering the 'rtst' answer case!
IUT Stderr: Debug: cf_st.c: answer: Add 'rtst' user to partner!
IUT Stderr: Debug: cf_st.c: answer: Insert partner!
IUT Stderr: Debug: cf_st.c: Construct answer pdu!
IUT Stderr: Debug: cf_st.c: Send answer-pdu!
IUT Stderr: Debug: mc_st.c: Sending ANSWER-pdu (21 bytes) to user 3
```

```
(Quiescense)
from_lower ! PDU_JOIN ! 103 ! 51 ! 2 ! 1
(Quiescense)
from_lower ! PDU_LEAVE ! 102 ! 52 ! 0 ! 1
from_upper ! JOIN ! 102 ! 52
from_lower ! PDU_DATA ! 21 ! 32 ! 2 ! 1
to_lower ! PDU_JOIN ! 102 ! 52 ! 1 ! 2
to_lower ! PDU_JOIN ! 102 ! 52 ! 1 ! 0
from_lower ! PDU_DATA ! 21 ! 34 ! 0 ! 1
to_lower ! PDU_JOIN ! 102 ! 52 ! 1 ! 2
to_lower ! PDU_JOIN ! 102 ! 52 ! 1 ! 0
(Quiescense)
from_upper ! DREQ ! 21 ! 31
(Quiescense)
from_lower ! PDU_JOIN ! 103 ! 52 ! 2 ! 1
to_lower ! PDU_ANSWER ! 102 ! 52 ! 1 ! 2
(Quiescense)
```

Case Study

The Conference Protocol Experiment

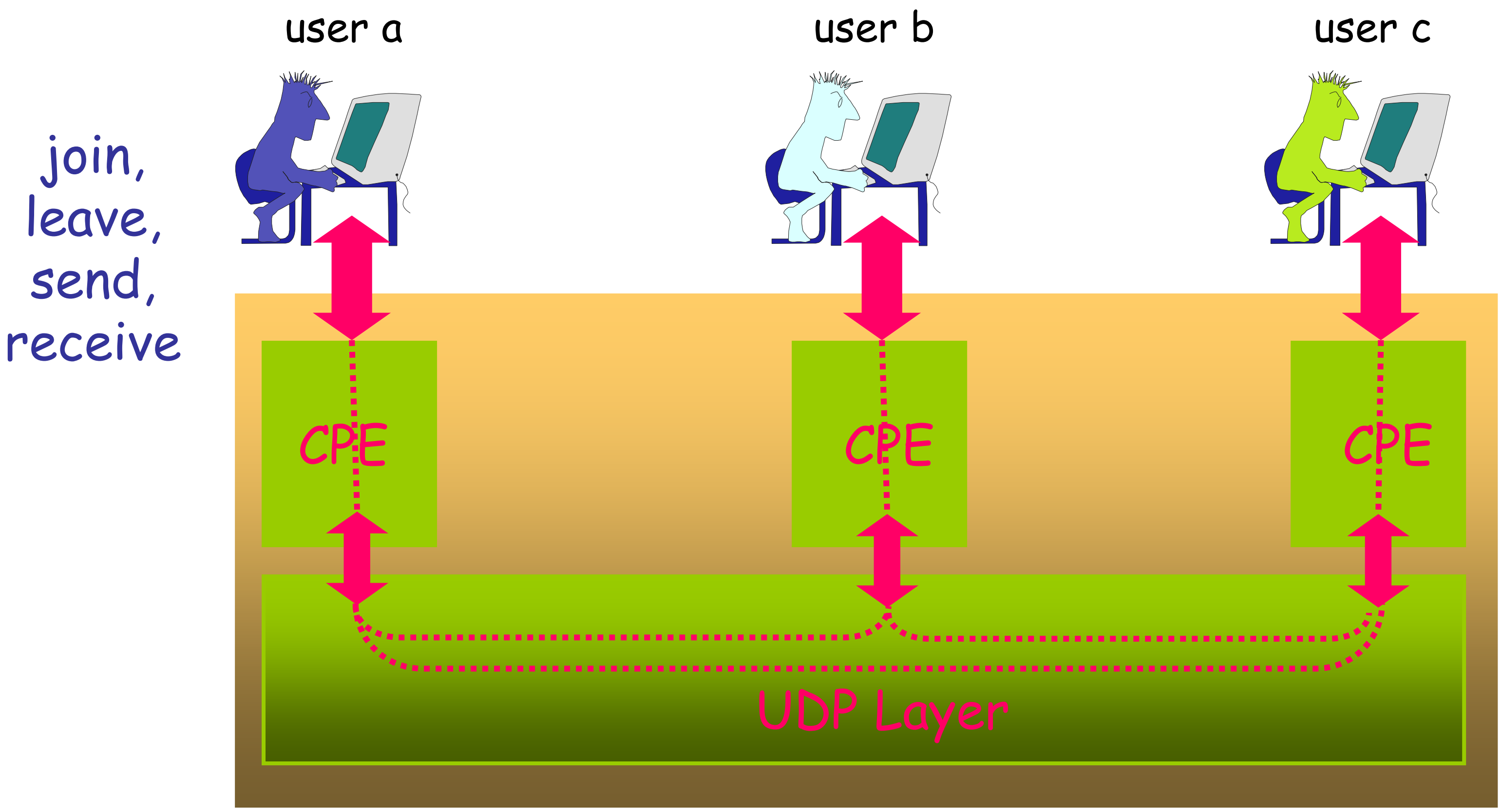
- Initiated for test tool evaluation and comparison
- Based on really testing different implementations
- Simple, yet realistic protocol
- Specifications in LOTOS, Promela, SDL, EFSM, …
- 28 different implementations in C
  - one of them (assumed-to-be) correct
  - others manually derived mutants

a single error is injected deliberately

- http://fmt.cs.utwente.nl/ConfCase

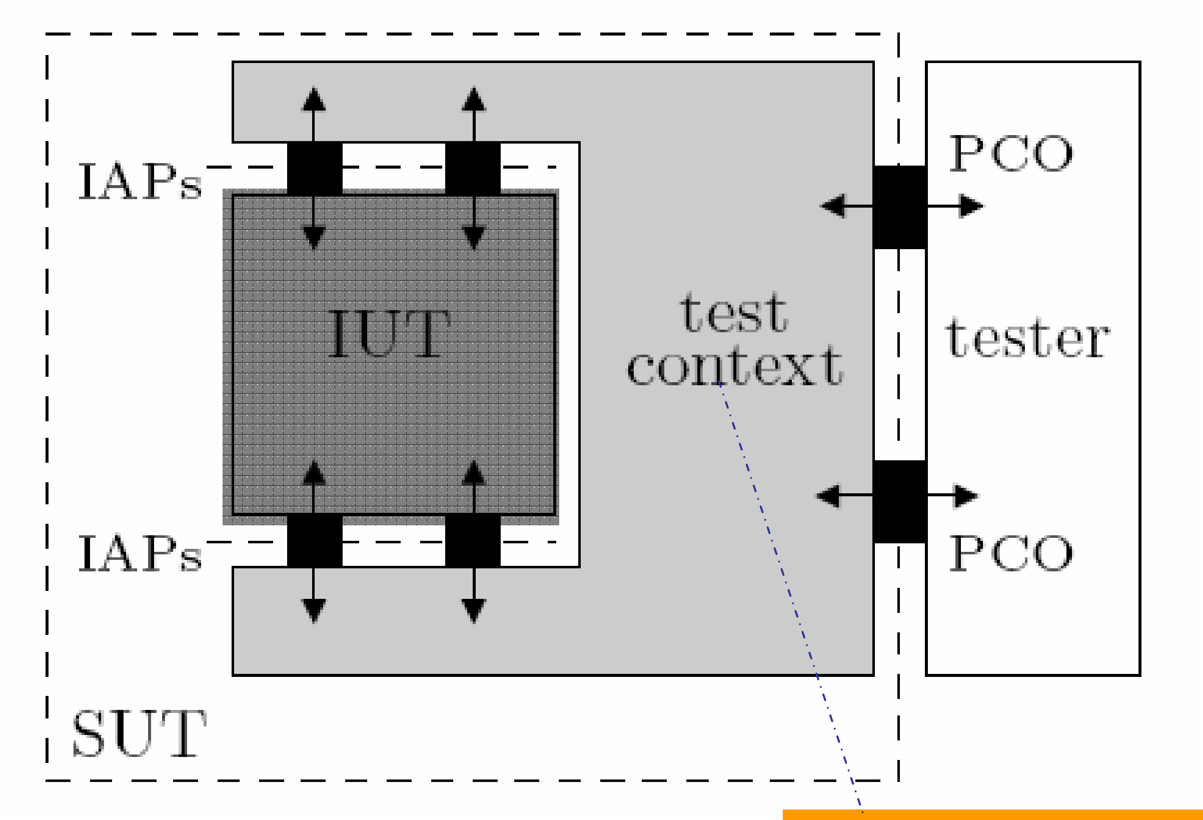
errors:
- no outputs
- no internal checks
- no internal updates

The Conference Protocol



CEP: Conference Protocol Entity
UDP: User Datagram Protocol

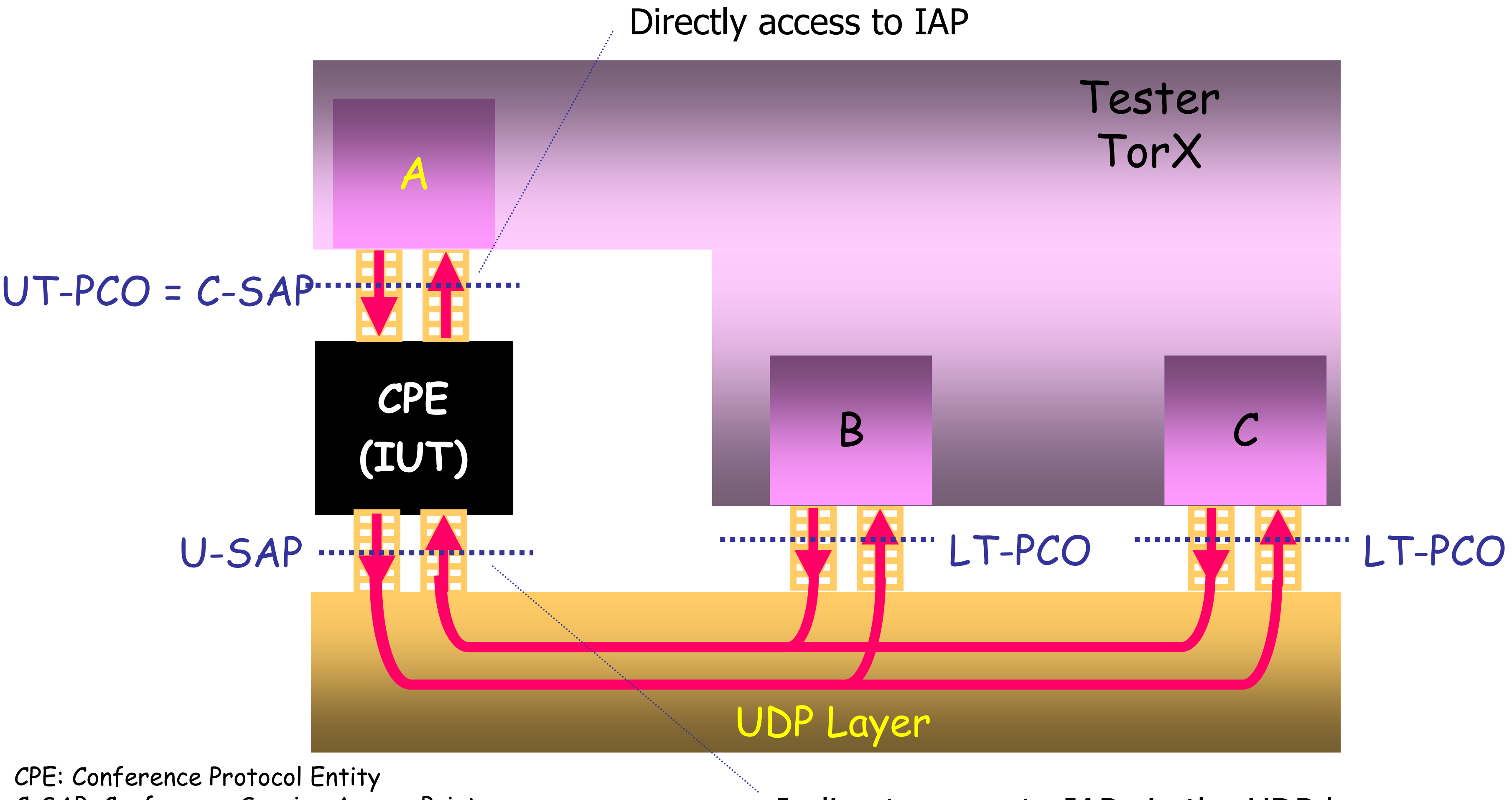
Abstract Test Architecture



The **test context** is the environment in which the IUT is embedded and that is present during testing, but it is not the aim of conformance testing.

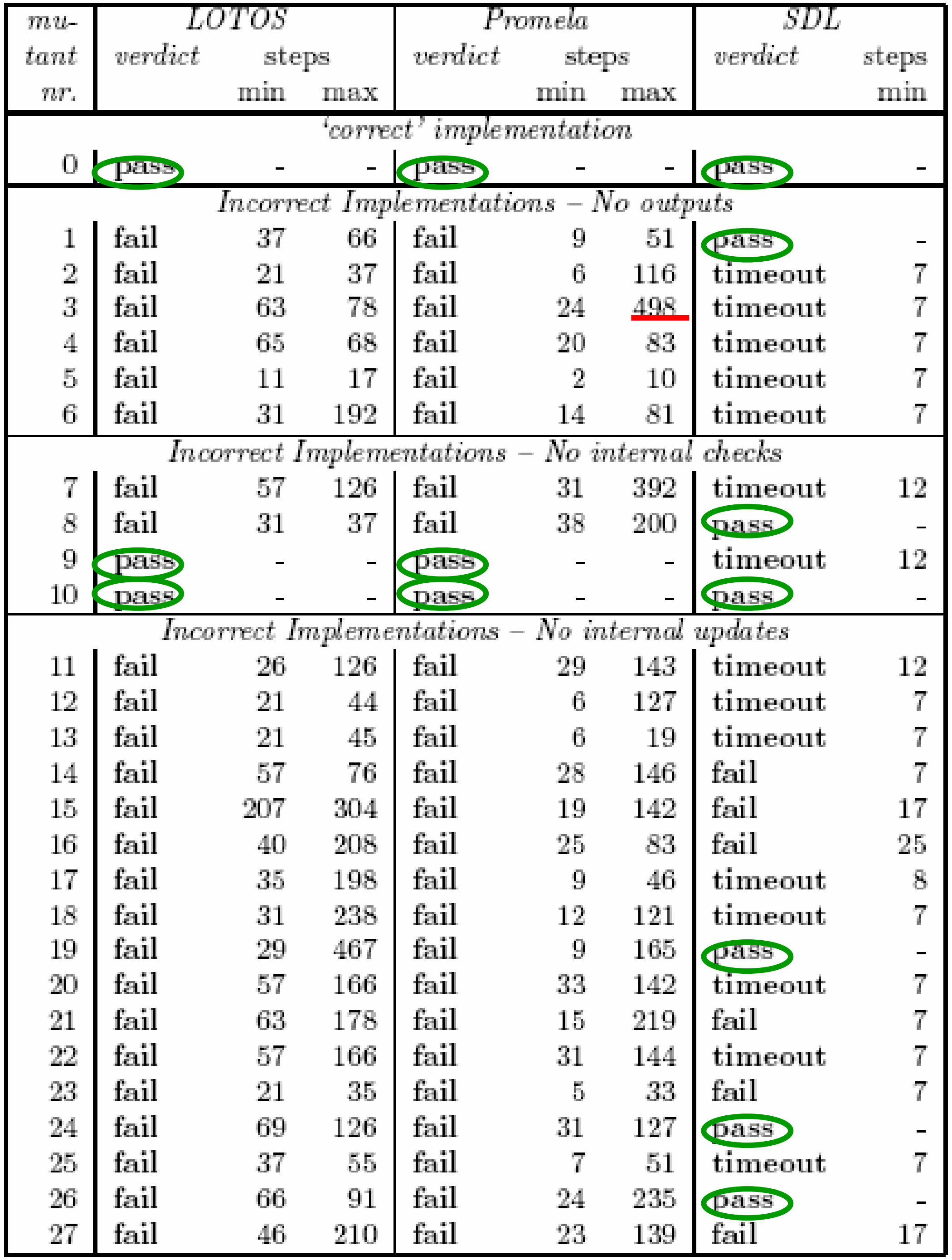
PCO: Point of Control and Observation
IAP: Implementation Access Point
IUT: Implementation Under Test
SUT: System Under Test (i.e., SUT = IUT + test context)

Conference Protocol: Concrete Test Architecture



CPE: Conference Protocol Entity
C-SAP: Conference Service Access Point
U-SAP: UDP Service Access Point
UT-PCO: Upper Tester Point of Control and Observation
LT-PCO: Lower Tester Point of Control and Observation

Test Results



mu-tant nr.	*LOTOS* *verdict*	steps min	max	*Promela* *verdict*	steps min	max	*SDL* *verdict*	steps min
	'correct' implementation							
0	pass	-	-	pass	-	-	pass	-
	Incorrect Implementations – No outputs							
1	fail	37	66	fail	9	51	pass	-
2	fail	21	37	fail	6	116	timeout	7
3	fail	63	78	fail	24	498	timeout	7
4	fail	65	68	fail	20	83	timeout	7
5	fail	11	17	fail	2	10	timeout	7
6	fail	31	192	fail	14	81	timeout	7
	Incorrect Implementations – No internal checks							
7	fail	57	126	fail	31	392	timeout	12
8	fail	31	37	fail	38	200	pass	-
9	pass	-	-	pass	-	-	timeout	12
10	pass	-	-	pass	-	-	pass	-
	Incorrect Implementations – No internal updates							
11	fail	26	126	fail	29	143	timeout	12
12	fail	21	44	fail	6	127	timeout	7
13	fail	21	45	fail	6	19	timeout	7
14	fail	57	76	fail	28	146	fail	7
15	fail	207	304	fail	19	142	fail	17
16	fail	40	208	fail	25	83	fail	25
17	fail	35	198	fail	9	46	timeout	8
18	fail	31	238	fail	12	121	timeout	7
19	fail	29	467	fail	9	165	pass	-
20	fail	57	166	fail	33	142	timeout	7
21	fail	63	178	fail	15	219	fail	7
22	fail	57	166	fail	31	144	timeout	7
23	fail	21	35	fail	5	33	fail	7
24	fail	69	126	fail	31	127	pass	-
25	fail	37	55	fail	7	51	timeout	7
26	fail	66	91	fail	24	235	pass	-
27	fail	46	210	fail	23	139	fail	17

The Conference Protocol Experiments

Reported experiments:

- TorX - LOTOS, Promela : on-the-fly ioco testing

  Axel Belinfante et al.,
  Formal Test Automation: A Simple Experiment
  IWTCS 12, Budapest, 1999.

- TorX statistics (with LOTOS and Promela)
  - all errors found after 2 - 498 test events
  - maximum length of tests : > 500,000 test events
  - 2 mutants react to PDU's from non-existent partners:
    - no explicit reaction is specified for such PDU's,
      so ioco-correct, and TorX does not test such behaviour

LTS Testing vs. FSM Testing

- FSM good at:
  - FSM has "more intuitive" theory
  - FSM test suite is complete
    -- but only w.r.t. assumption on number of states
  - FSM test theory has been around for a number (>40) of years

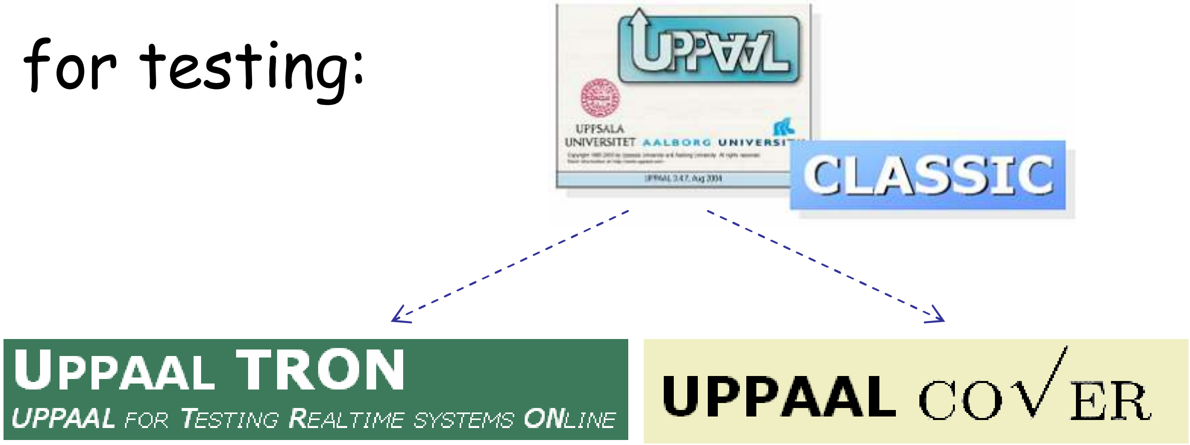
- FSM bad at:
  - Restrictions on FSM:
    - deterministic
    - completeness
  - FSM has always alternation between input and output
  - Difficult to specify interleaving in FSM
  - FSM is not compositional

Model-Based Real-time System Testing:
--- The Uppaal Approach

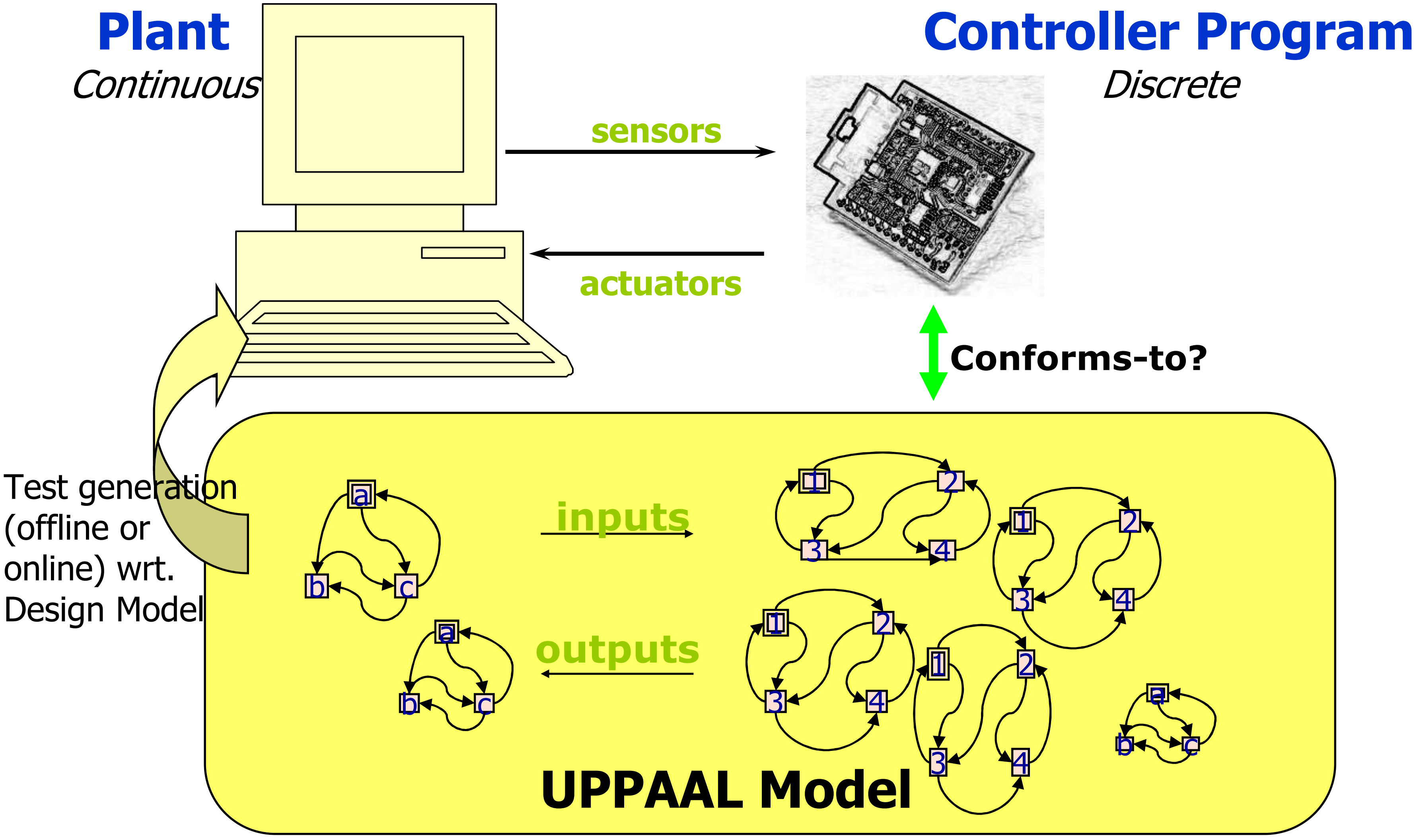
Uppaal Tool and it's Branches for Testing

- **Uppaal** is an integrated tool environment for modeling, simulation and verification of real-time systems modeled as networks of timed automata, extended with data types.

- Uppaal's branches for testing:
  - Uppaal-TRON
  - Uppaal-Cover



Real-time Model-Based Testing

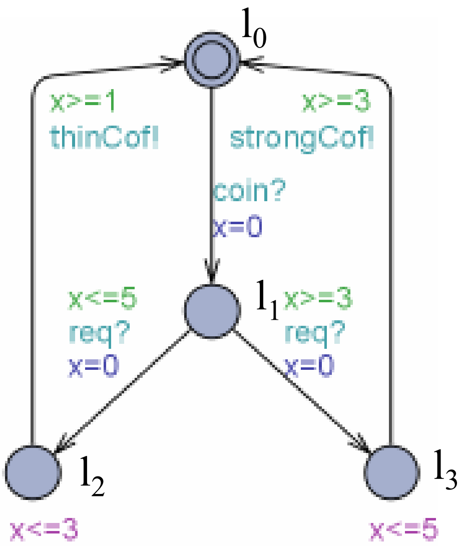


Timed System Testing

- Model:
  - Timed Input-Output Labelled Transition System (Timed IOLTS)
- Conformance relation:
  - Timed Input-Output Conformance (Timed ioco)

Timed IOLTS by Example

- Given a timed automaton:
  - location: $\{l_0, l_1, l_2, l_3\}$
  - actions:
    - {coin?, req?} --- input actions
    - {thinCof!, strongCof!} --- output actions
  - clock: {x}

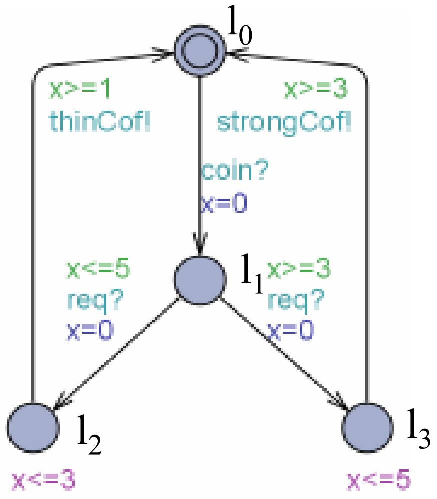


- Semantic state:
  - e.g.: $(l_0, x=0)$, $(l_0, x=2)$, $(l_1, x=4)$
- Semantic transition:
  - e.g.: $(l_0, x=0)$ --delay(2)--> $(l_0, x=2)$,
    $(l_0, x=2)$ --coin?--> $(l_1, x=0)$,

Such a transition system is a timed IOLTS
- as semantic interpretation of TA
- yypically infinite transition systems (because clocks are real variables)

Timed Conformance: tioco

- Derived from Tretman's ioco
- Let I, S be two timed IOLTS's, P a set of states
  - TTr(P): the set of timed traces from a state in P
    - eg.: σ = coin?.5.req?.2.thinCoffee!.9.coin?
  - Out(P after σ) = possible outputs and delays after σ
    - eg. out ({l_2,x=1}): {thinCoffee, 0...2}

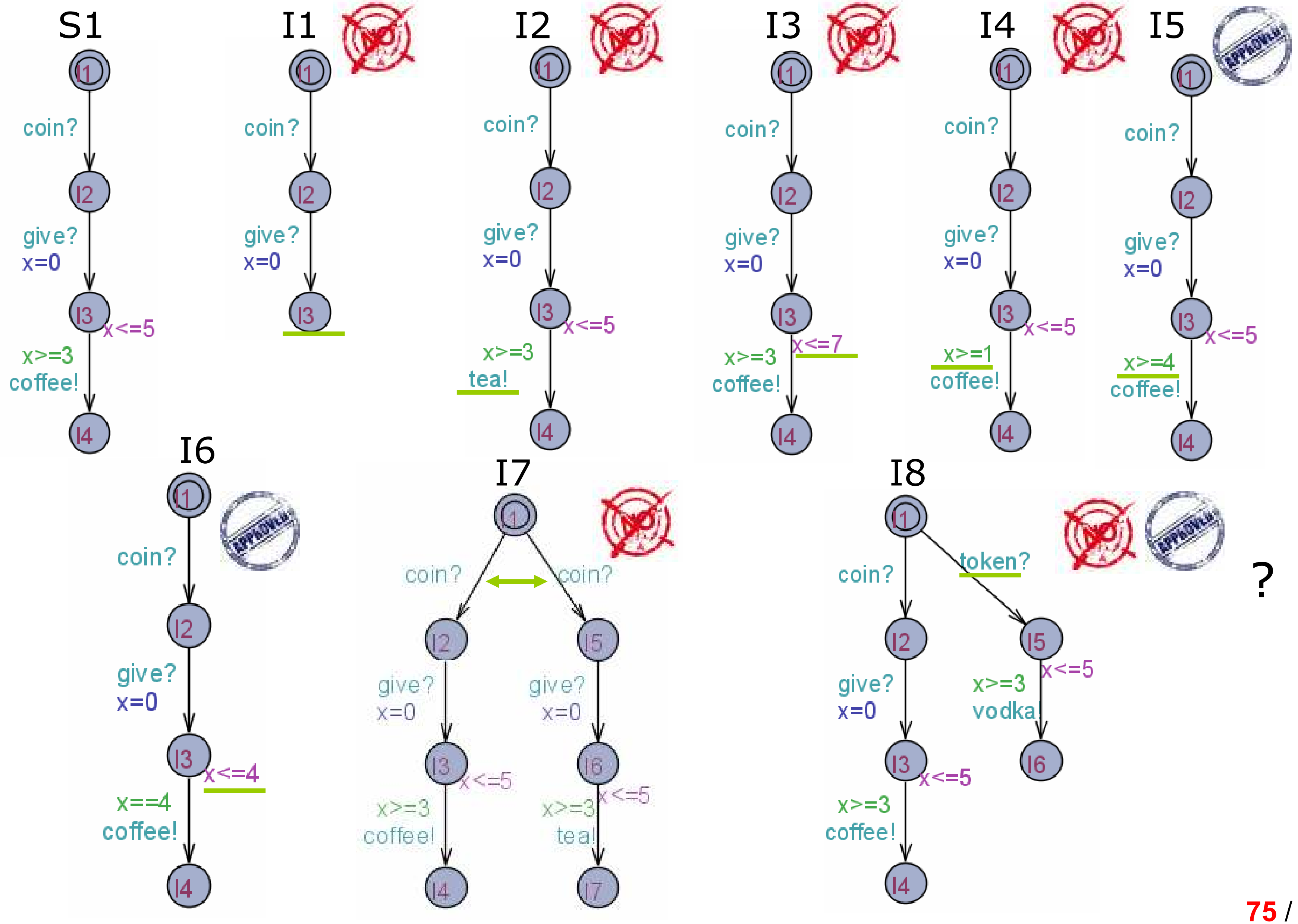


- I tioco S $=_{def}$
  - $\forall\sigma \in$ TTr(S): Out(I after σ) $\subseteq$ Out(S after σ), or
  - TTr(i_0) $\subseteq$ TTr(s_0), where i_0 and s_0 are the initial states of I and S respectively

- Intuition
  - IUT can accept all inputs for SPEC (and perhaps some other inputs)
  - if IUT ever produces an output as required by SPEC, it should be produced in time
  - but IUT is not allowed to produce any illegal output (w.r.t. SPEC)

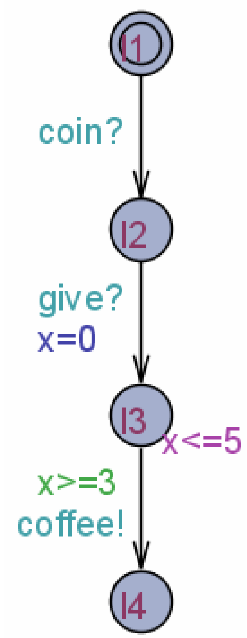
See also [Krichen&Tripakis, Khoumsi]

Does I_n Conform-to S_1?

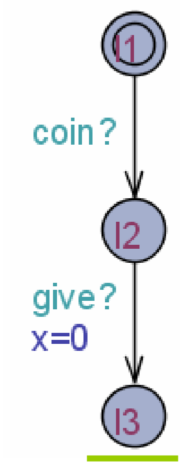


Does I_n Conform-to S_1?

S1



I1

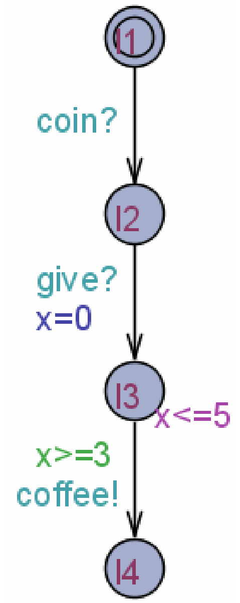


σ=coin.give.**10**
$\sigma \in$**TTr**(I1), $\sigma \notin$**TTr**(S1)

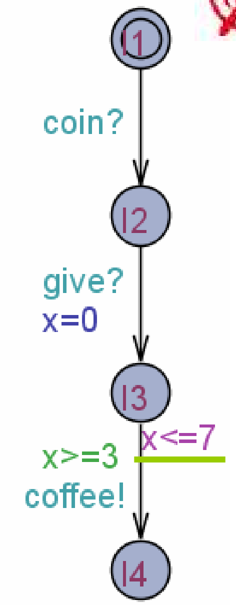
out(I1 **after** coin.give.3)={**0…** ∞}
$\not\subset$
out(S1 **after** coin.give.3)={coffee,0…2}

Does I_n Conform-to S_1?

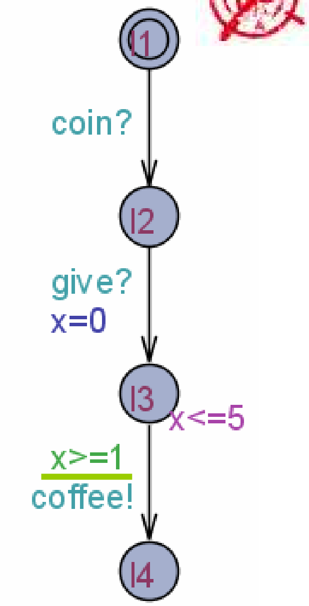
S1



I3



I4



σ=coin.give.**7**.coffee
σ∈**TTr**(I3), σ ∉TTr(S1)

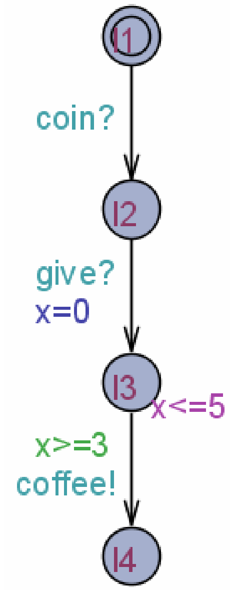
σ=coin.give.**1**.coffee
σ∈**TTr**(I4), σ ∉TTr(S1)

out(I3 after coin.give.7)={coffee,0}
⊄
out(S1 after coin.give.7)={}

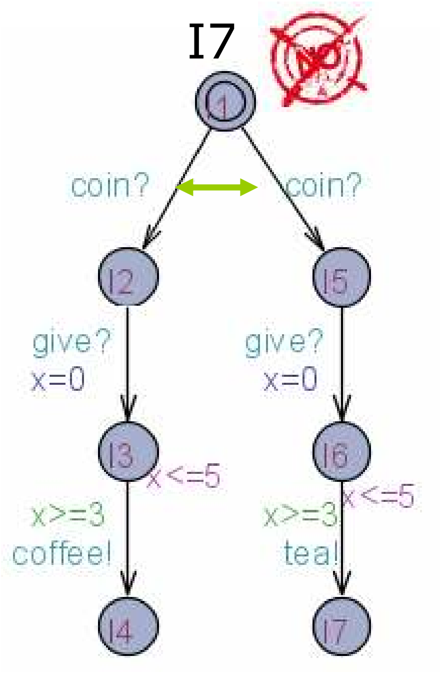
out(I4 after coin.give.1)={coffee,0...4}
⊄
out(S1 after coin.give.1)={0...4}

Does I_n Conform-to S_1?

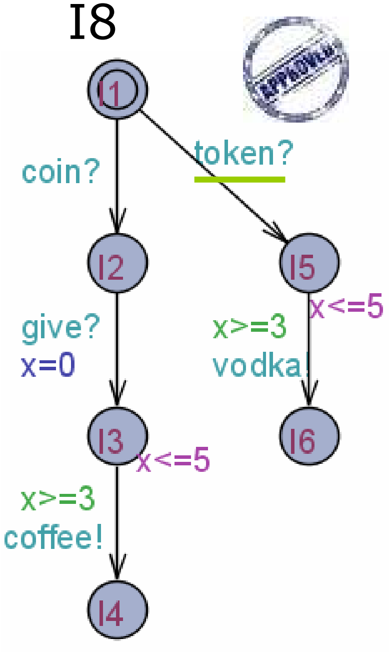
S1



I7



I8



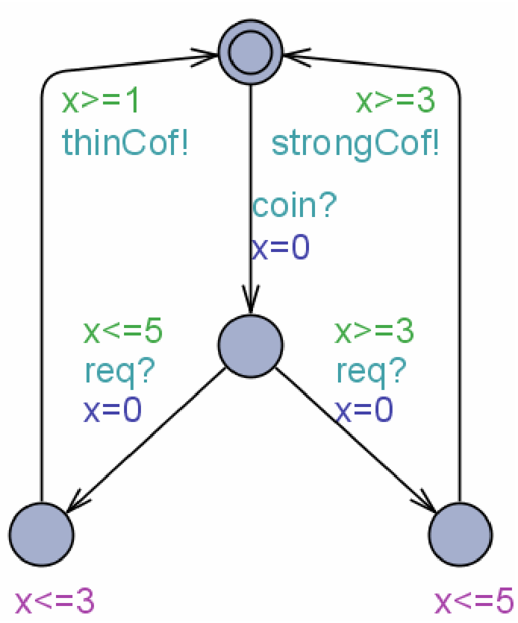
σ=coin.give.5.**tea**
σ∈**TTr**(I7), σ ∉**TTr**(S1)

σ=token.5.vodka
σ∈**TTr**(I8), σ ∉**TTr**(S1)
But σ was not specified in S1

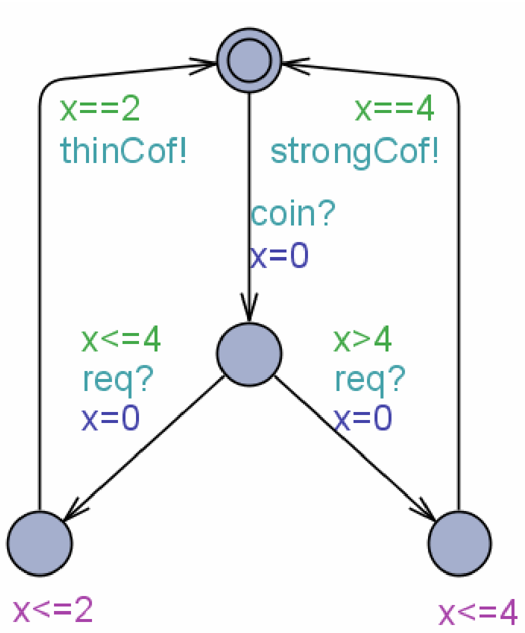
out(I7 **after** coin.give.5)={tea, coffee,0}
⊄
out(S1 **after** coin.give.5)={coffee,0}

Now, Back to Timed Coffee Machine

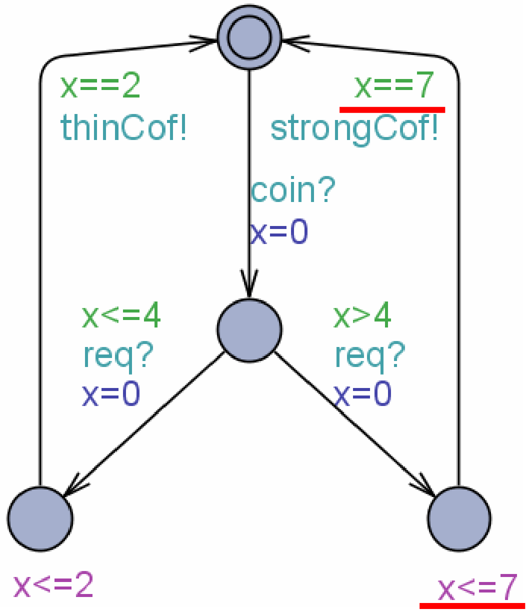
Specification



Implementation 1



Implementation 2



Example Traces

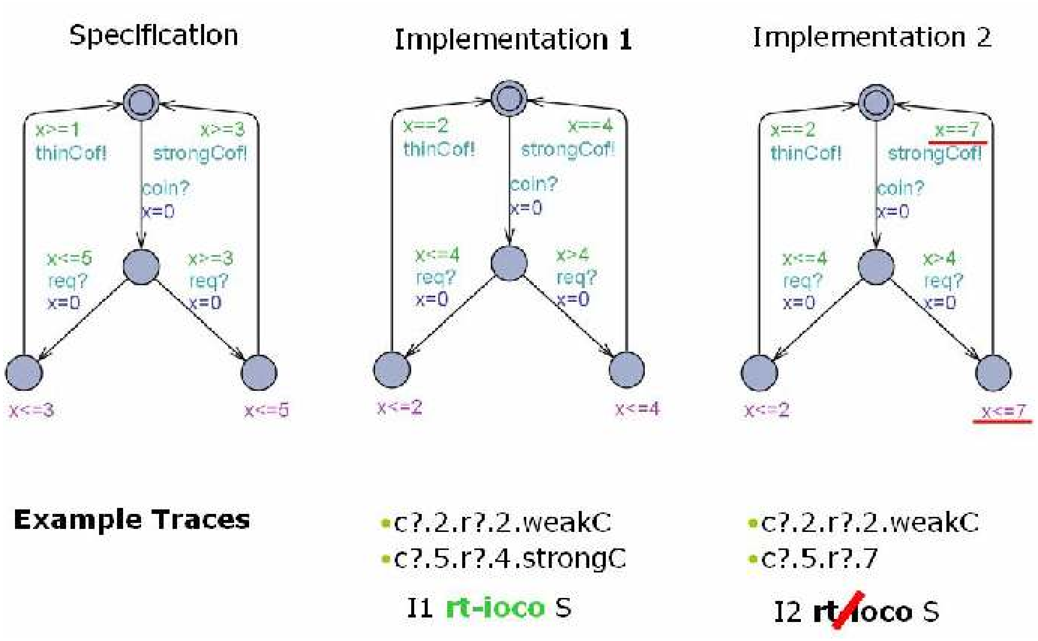
- c?.2.r?.2.weakC
- c?.5.r?.4.strongC

I1 **rt-ioco** S

- c?.2.r?.2.weakC
- c?.5.r?.7

I2 **rt~~-~~ioco** S

Essence of "Timed ioco"?



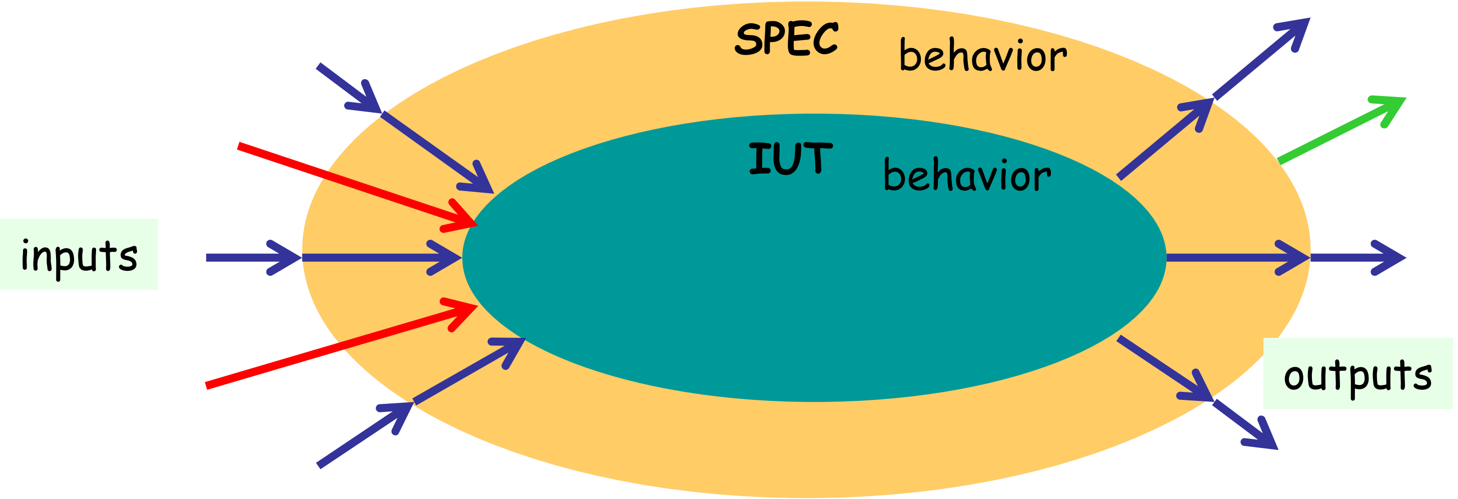
Example Traces

- c?.2.r?.2.weakC
- c?.5.r?.4.strongC

I1 **rt-ioco** S

- c?.2.r?.2.weakC
- c?.5.r?.7

I2 **rt$\not$-ioco** S



Explicit Environment Modelling

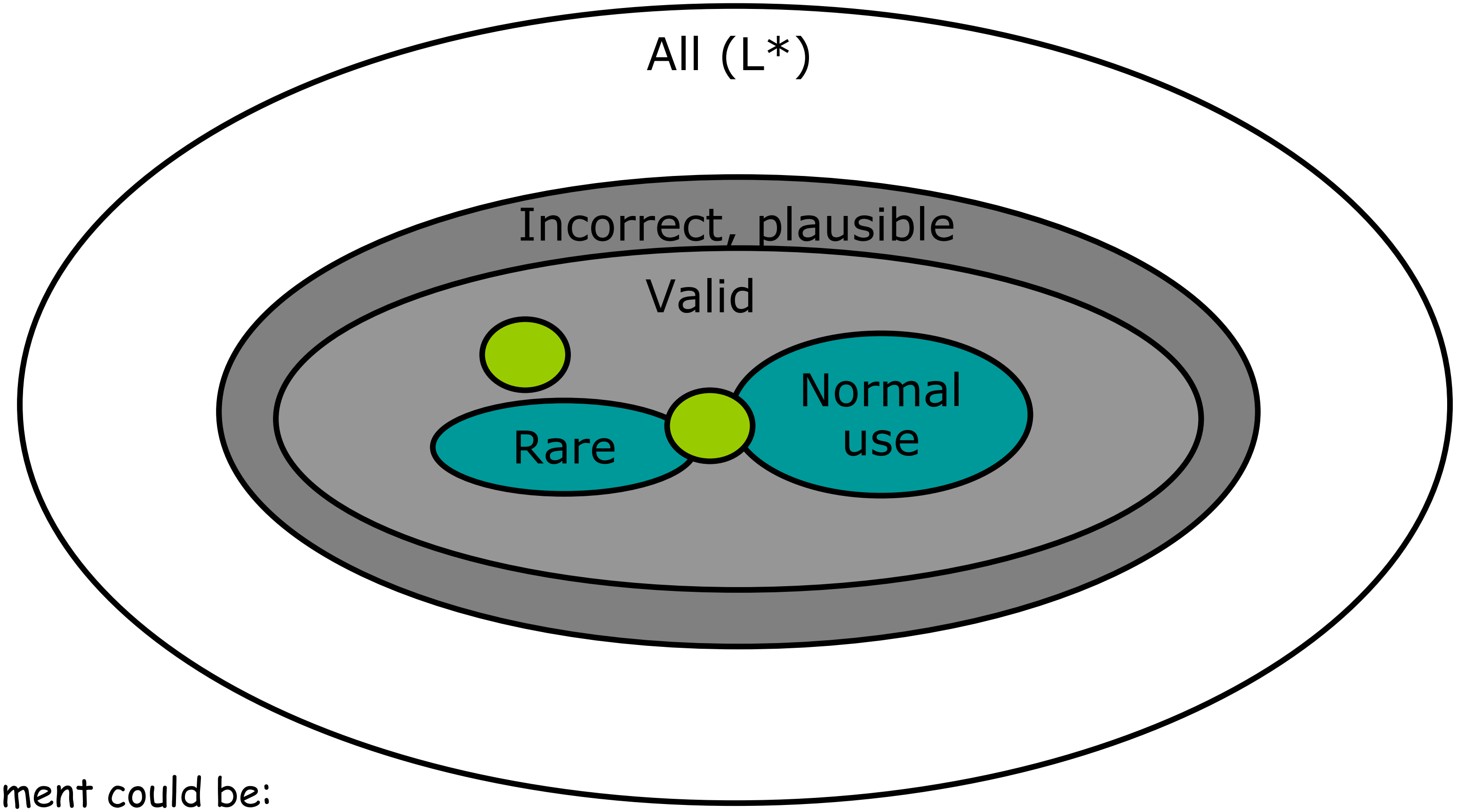
Recall that in "ioco" conformance…

- I tioco S $=_{def}$
  - $\forall\sigma \in$ TTr(S): Out(I after σ) $\subseteq$ Out(S after σ), or
  - TTr(i_0) $\subseteq$ TTr(s_0), where i_0 and s_0 are the initial states of I and S respectively

- Note that:
  - TTr(S) is a very big (infinite) set
  - We are usually interested in only a small portion of the behavior

- A solution:
  - To explicitly model the environment that the IUT will be operated in

The Environment "Universe"



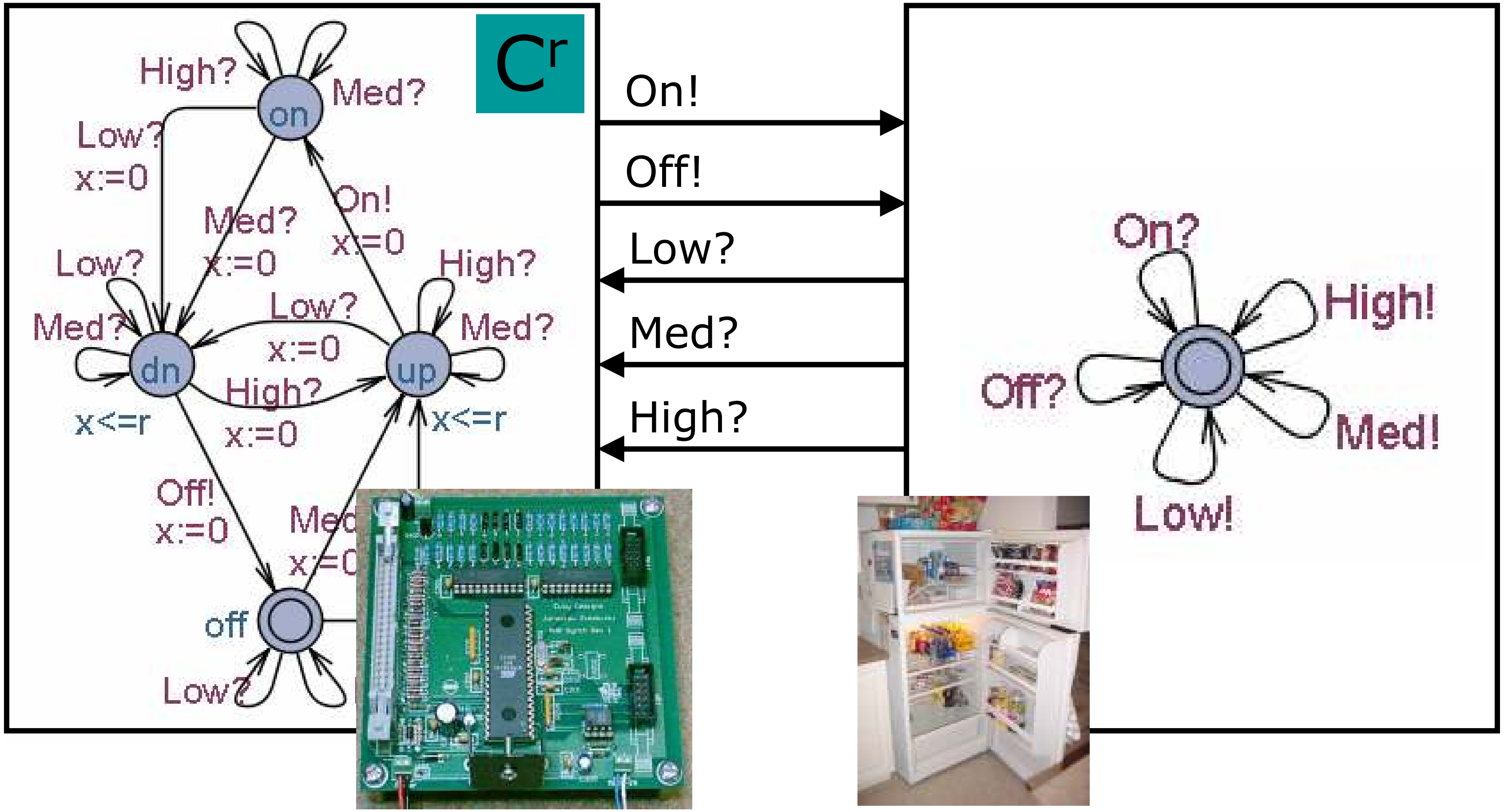
environment could be:

- Other external systems (Dedicate / open protocols)
- Other internal systems (eg powersupply, radio)
- Human Users
- Physical Plant via sensors / actuators

Sample Cooling Controller

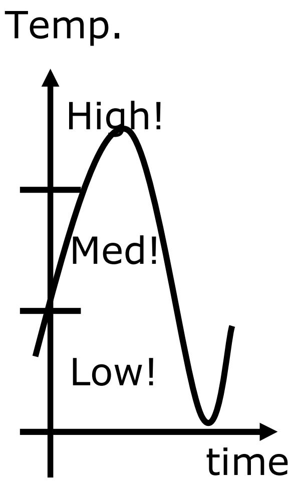
IUT-model

Env-model

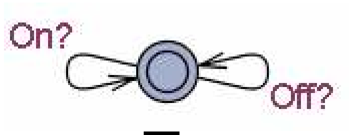


- When T is high (low) switch on (off) cooling within r secs.
- When T is medium cooling may be either on or off (impl. freedom)

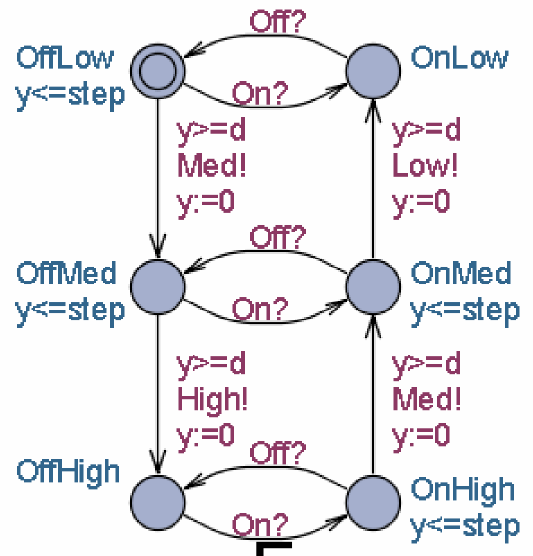
Environment Modelling



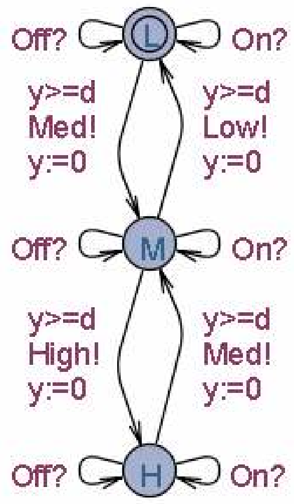
- E_M Any action possible at any time
- E_1 Only realistic temperature variations
- E_2 Temperature never increases when cooling
- E_L No inputs (completely passive)



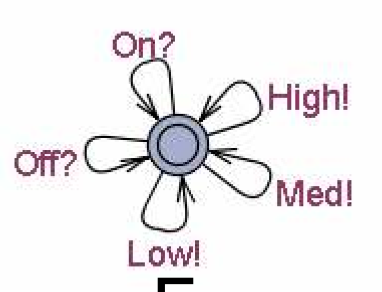
E_L



E_2



E_1

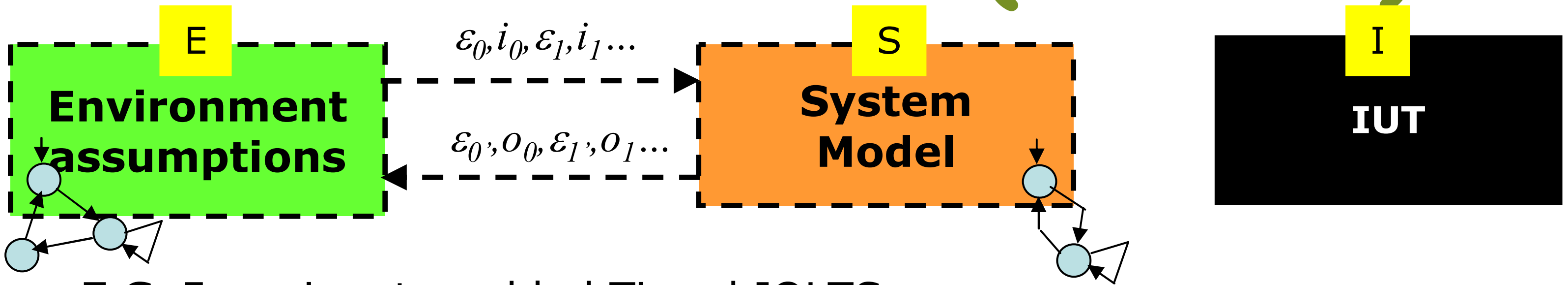


E_M

(strict) ⟷ (loose)

$$E_L \sqsubseteq E_2 \sqsubseteq E_1 \sqsubseteq E_M$$

Relativized Timed Input-Output Conformance (rt-ioco)



E

Environment assumptions

$\varepsilon_0, i_0, \varepsilon_1, i_1 \ldots$

$\varepsilon_0', o_0, \varepsilon_1', o_1 \ldots$

S

System Model

I

IUT

- **E,S, I** are input-enabled Timed IOLTS
- **Let** *P* **be a set of states**
- **TTr**(*P*): the set of *timed traces* from states in *P*
- *P* **after** σ = the set of states reachable after timed trace σ
- **Out**(*P*) = possible outputs and delays from states in *P*

- I rt-ioco$_E$ S $=_{def}$

  $\forall \sigma \in$ TTr(E): Out((E,I) after σ) $\subseteq$ Out((E,S) after σ)

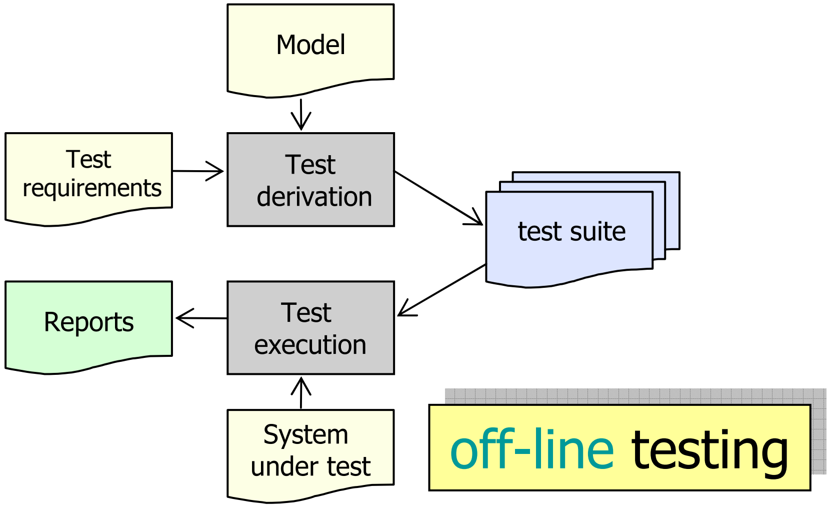
or

- I rt-ioco$_E$ S iff TTr(I) $\cap$ TTr(E) $\subseteq$ TTr(S) $\cap$ TTr(E) *// input enabled*

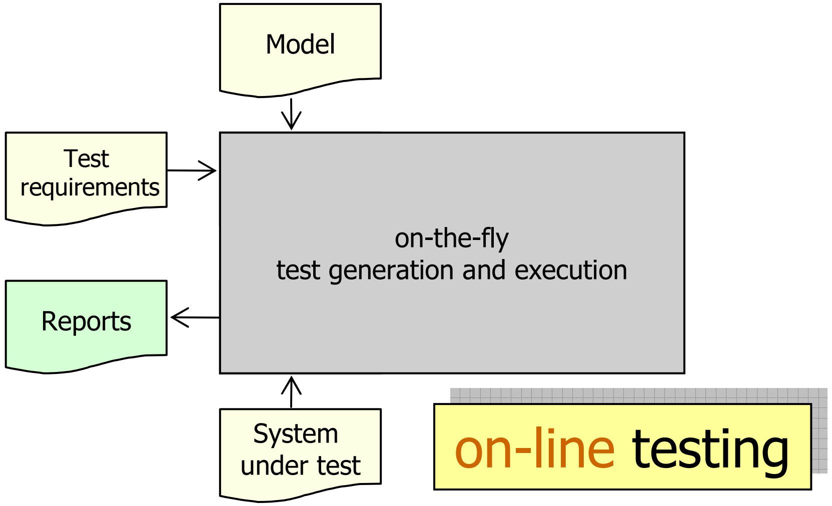
- ***Intuition: for all assumed environment behaviors, the IUT***
  - ***never produces illegal output, and***
  - ***if ever produces required output, then produces it in time***

See also [Larsen 04 FATES]

Off-line and On-line Testing



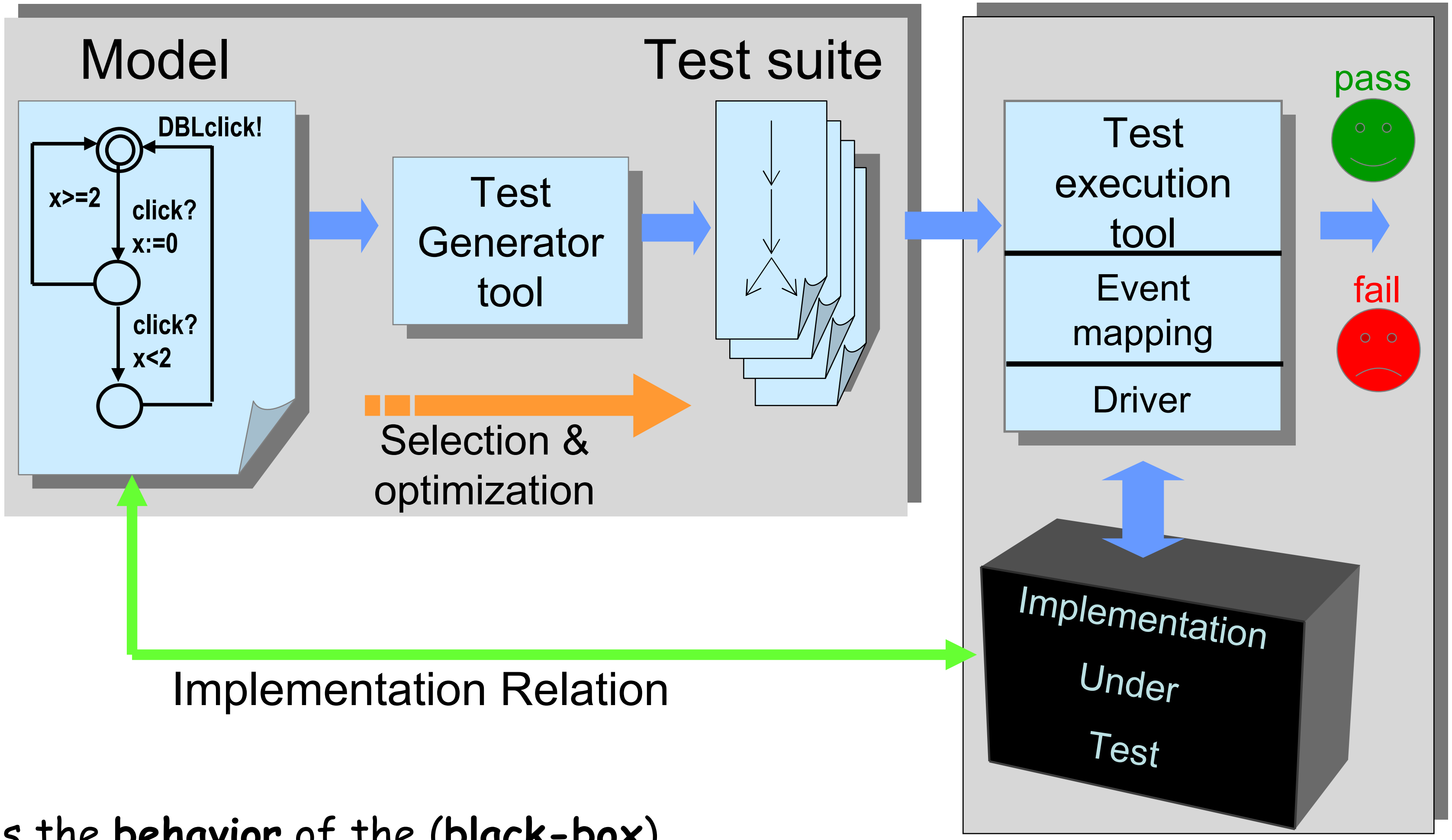
off-line testing



on-line testing

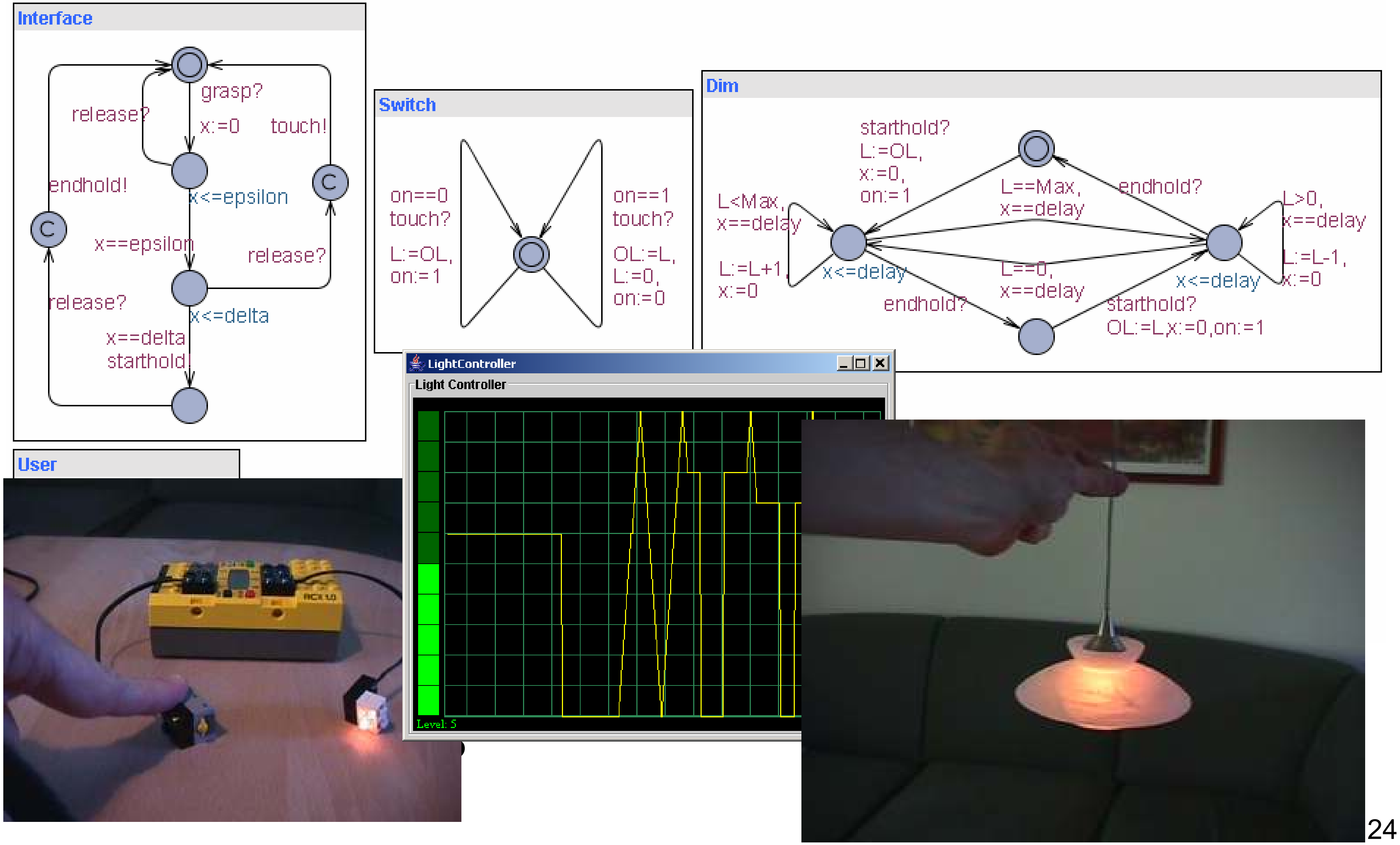
Model-Based Off-line Testing of Timed Systems

Automated Model-Based Off-line Conformance testing

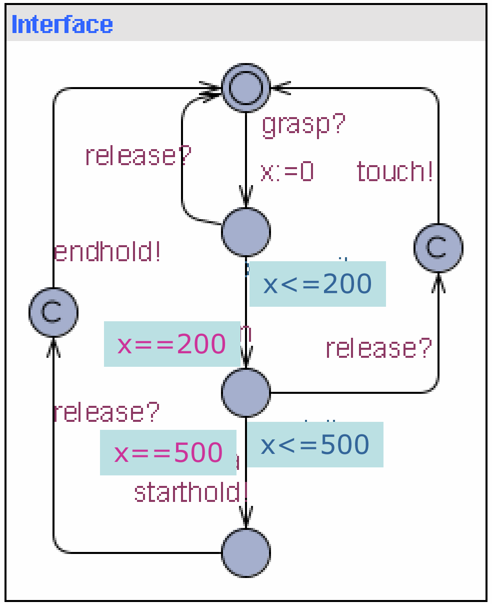


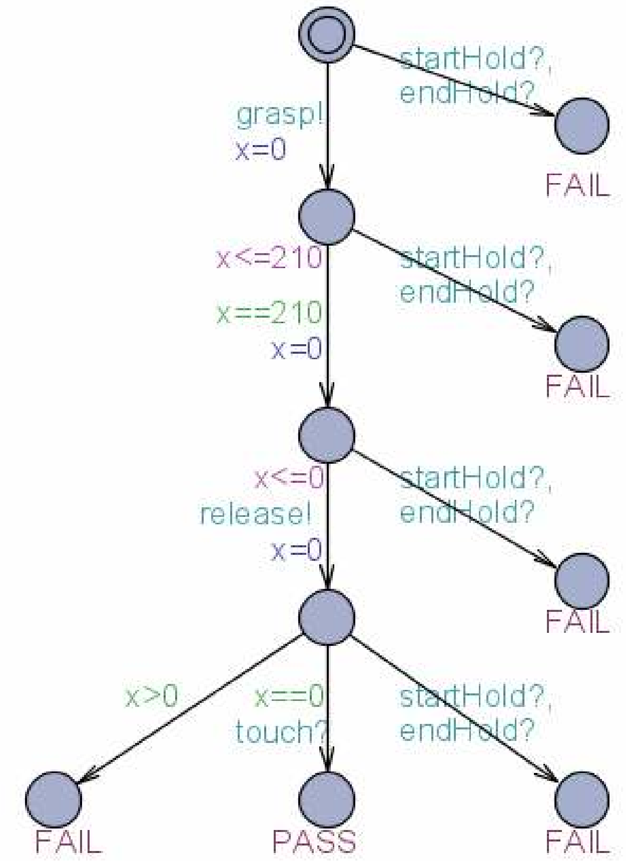
Does the **behavior** of the (**black-box**) implementation ***comply*** to that of the specification?

Touch-sensitive Light Controller



Timed Tests





EXAMPLE test cases for **Interface**

`0·grasp!·210·release!·touch?.PASS`

`0·grasp!·317·release!·touch?·2½·grasp!·220·release!·touch?·PASS`

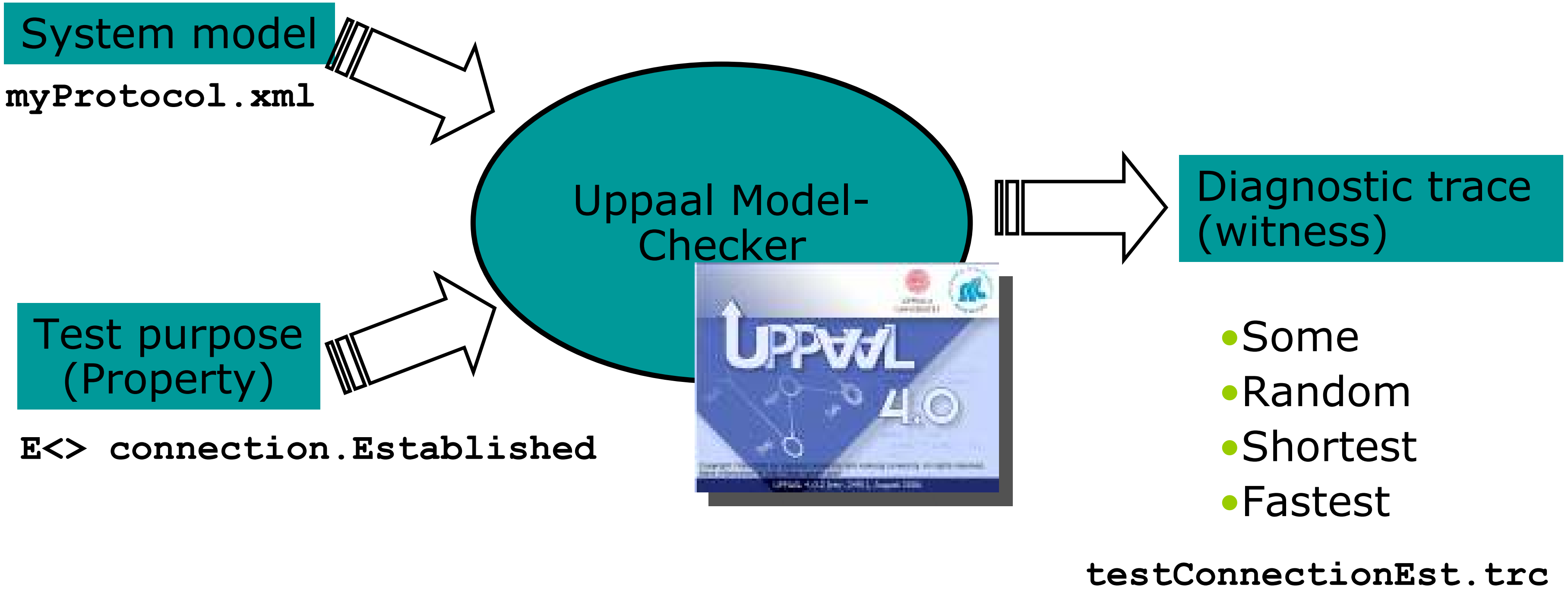
`1000·grasp!·517·starthold?·100·release!·endhold?·PASS`

Infinitely many sequences!!!!!!

Test Selection?

- Infinitely many sequences…
- But testing practice should definitely be finite
- To select finitely many out from an infinitely large pool
  - Test coverage criteria
  - Test purposes

Test Generation by Model-Checking



System model

`myProtocol.xml`

Test purpose (Property)

`E<> connection.Established`

Uppaal Model-Checker

Diagnostic trace (witness)

- Some
- Random
- Shortest
- Fastest

`testConnectionEst.trc`

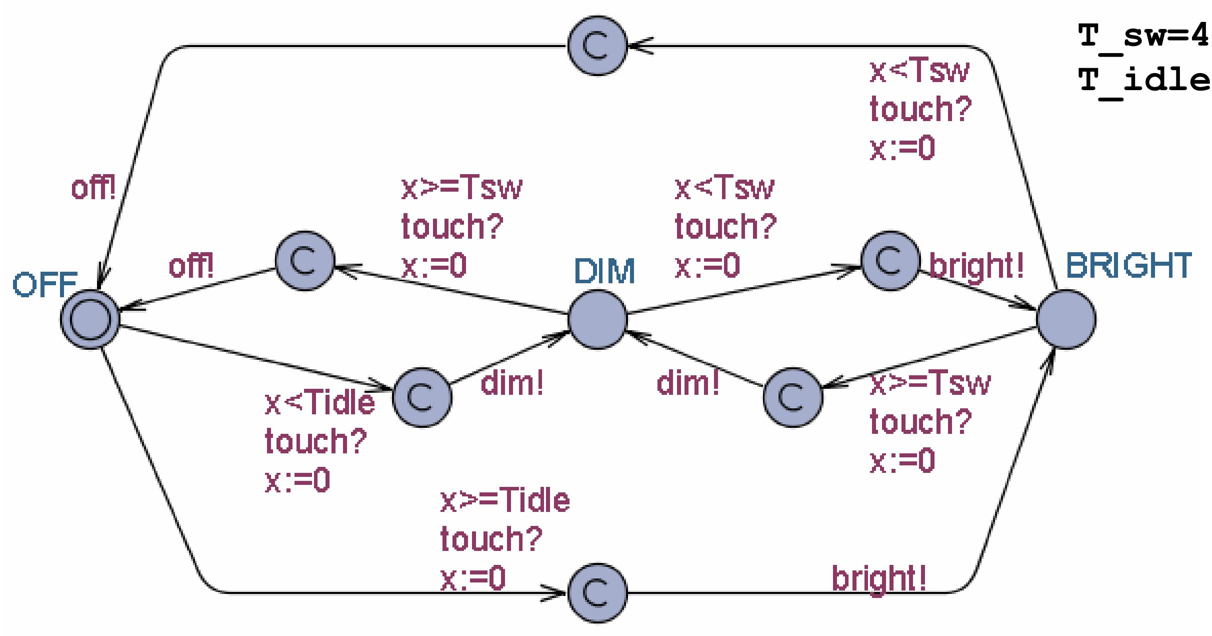
- Use diagnostic trace as test case??!!

Controllable Timed Automata

- **“DOUTA”-Model**
  - **Deterministic**: two transitions with same input/output leads to the same state
  - **Output-Urgent**: enabled outputs will occur immediately
  - **Isolated Outputs**: if an output is enabled, no other output is enabled
  - **Input-Enabled**: all inputs can always be accepted

A DOUTA Timed Automaton

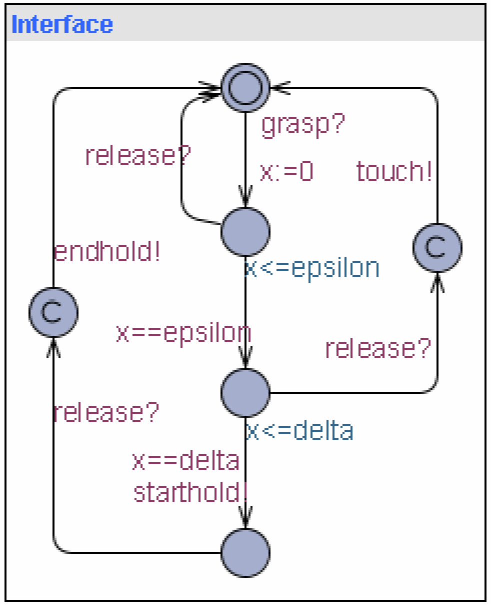
Deterministic,
Output-Urgent,
Isolated Outputs,
Input-Enabled

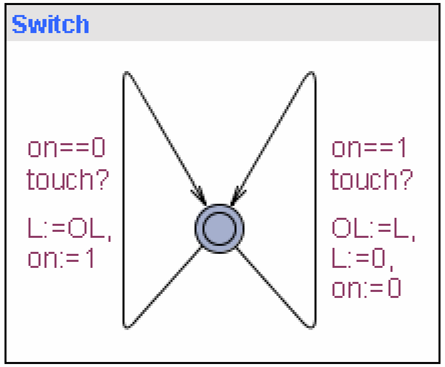


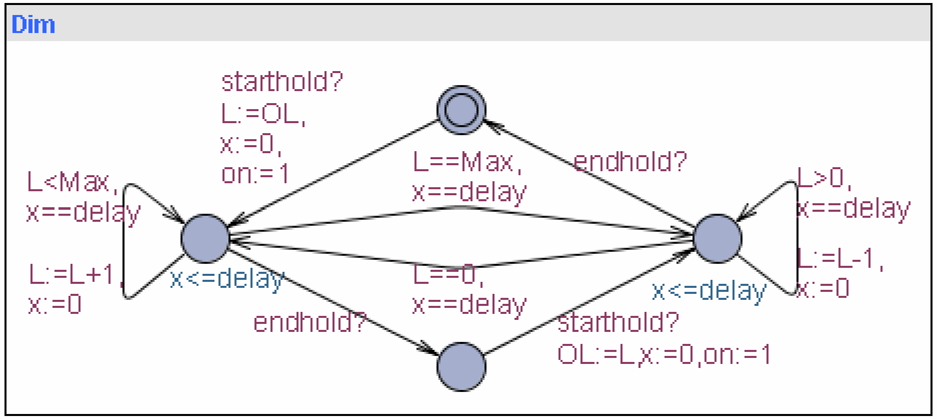
```
T_sw=4
T_idle=20
```

WANT: if touch is issued twice quickly then the light will get brighter; otherwise the light is turned off.

Without Test Purpose







- Epsilon=200ms
- Delta=500ms

EXAMPLE test cases for **Interface**

```
0·grasp!·210·release!·touch?.PASS
0·grasp!·317·release!·touch?·2½·grasp!·220·release!·touch?·PASS
1000·grasp!·517·starthold?·100·release!·endhold?·PASS
```

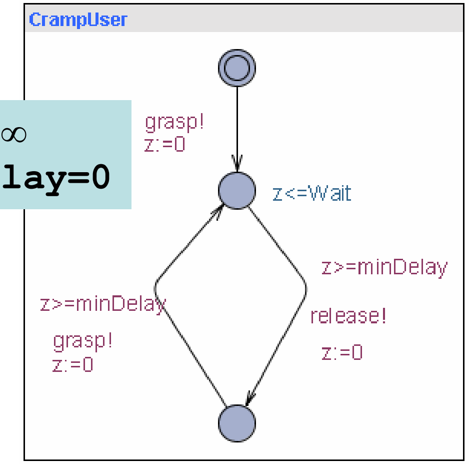
Infinitely many sequences!!!!!!

Test Purpose #1

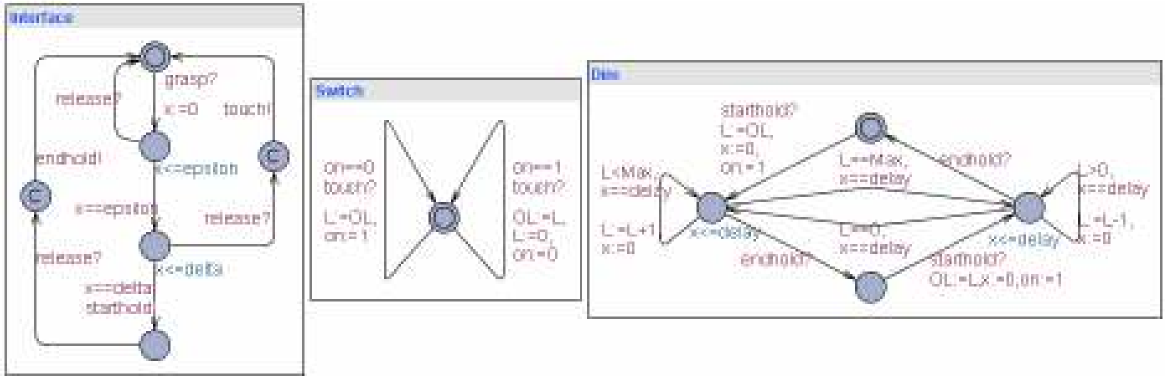
Test Purpose: A specific test objective (or observation) the tester wants to make on SUT

Environment model

Wait=∞
minDelay=0



System model



TP1: Check that the light can become bright:

E<> L==10

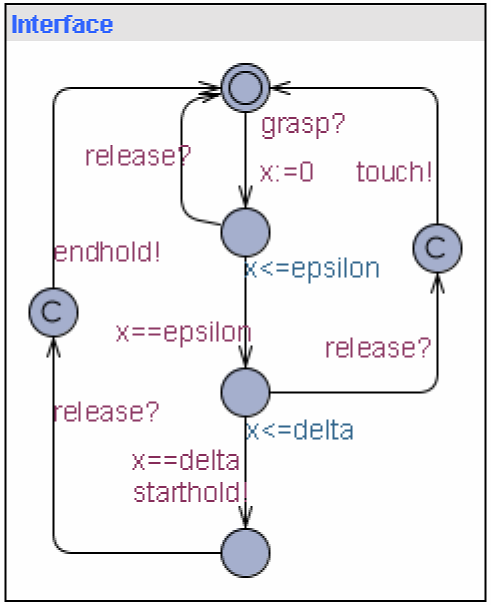
- *Shortest* (and *fastest*) Test:

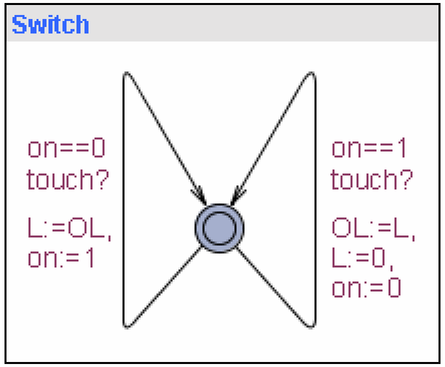
out(IGrasp);silence(500);in(OSetLevel,0);silence(1000);
in(OSetLevel,1);silence(1000);in(OSetLevel,2); silence(1000);
in(OSetLevel,3);silence(1000);in(OSetLevel,4);silence(1000);
in(OSetLevel,5);silence(1000);in(OSetLevel,6);silence(1000);
in(OSetLevel,7);silence(1000);in(OSetLevel,8);silence(1000);
in(OSetLevel,9);silence(1000);in(OSetLevel,10);
out(IRelease);

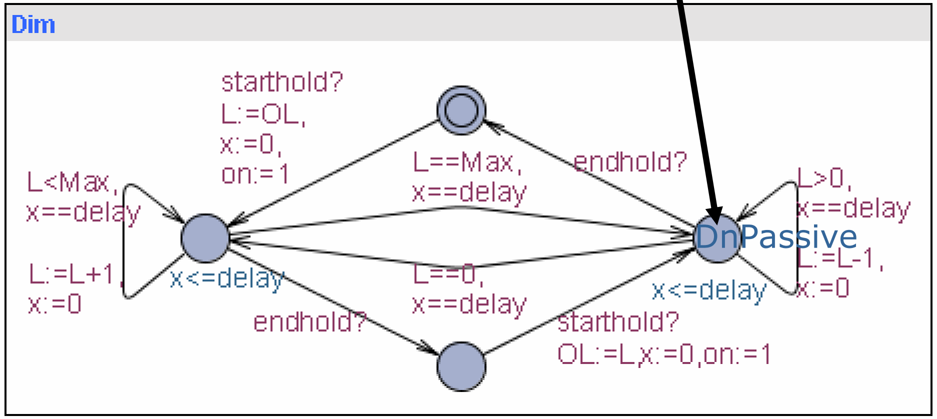
Test Purpose #2

TP2: Check that controller can enter location 'DnPassive':

`E<> Dim.DnPassive`

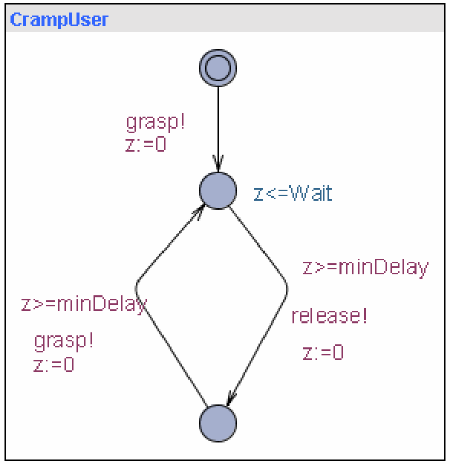






- If delay=1000
- *Shortest* (and *fastest*) Test:

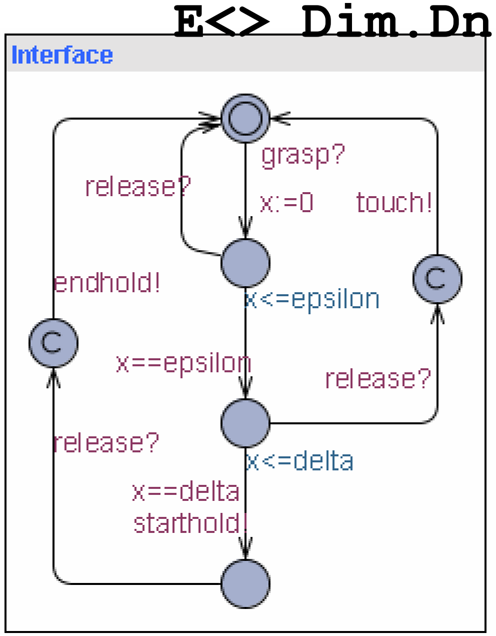
```
out(IGrasp);
silence(500);
in(OSetLevel,0);
out(IRelease);
out(IGrasp);
silence(500);
```

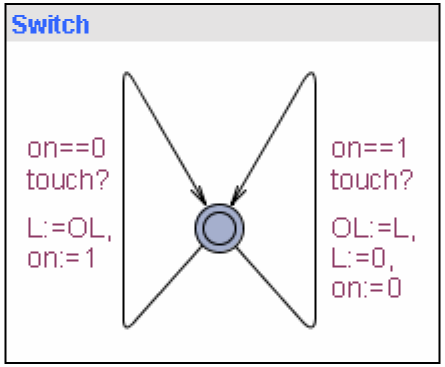


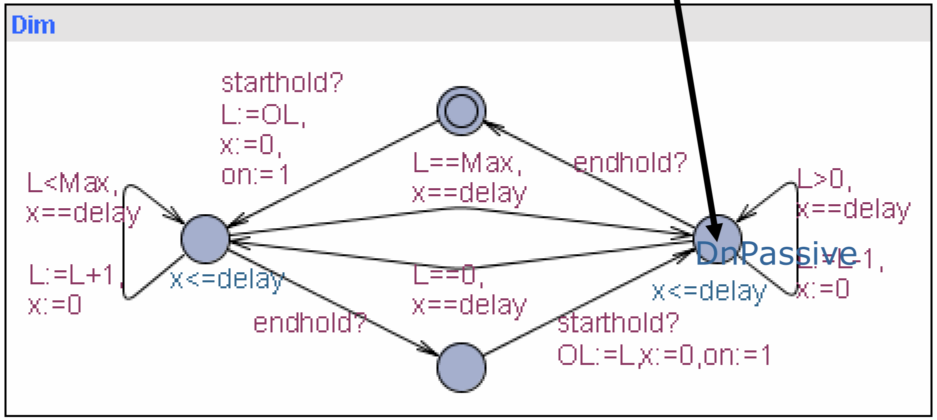
Test Purpose #2

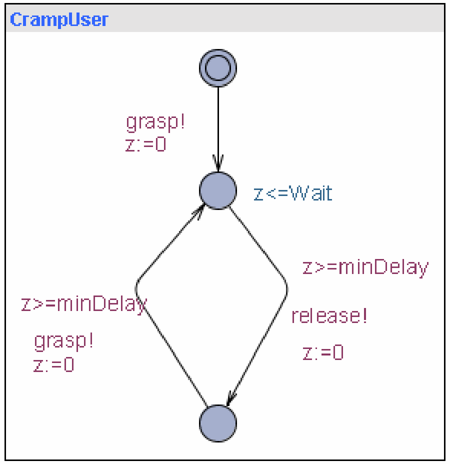
TP2: Check that controller can enter location 'DnPassive':

`E<> Dim.DnPassive`









- If delay=40?
- *Shortest* Test:

```
out(IGrasp);
silence(500);
in(OSetLevel,0);
out(IRelease);
out(IGrasp);
silence(500);
```

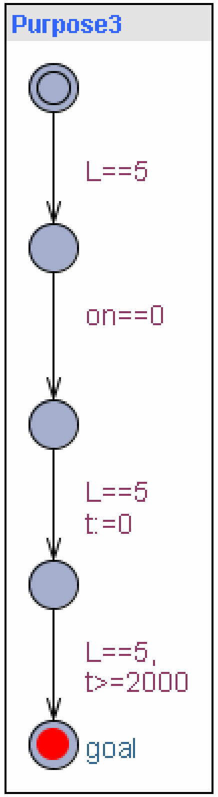
- *Fastest* Test:

```
out(IGrasp);silence(500);in(OSetLevel,0);silence(40);
in(OSetLevel,1);silence(40);in(OSetLevel,2); silence(40);
in(OSetLevel,3);silence(40);in(OSetLevel,4); silence(40);
in(OSetLevel,5);silence(40);in(OSetLevel,6); silence(40);
in(OSetLevel,7);silence(40);in(OSetLevel,8); silence(40);
in(OSetLevel,9);silence(40);in(OSetLevel,10);silence(40);
```

Test Purpose #3

TP3: Check that controller resets light level to previous value after switch-on.

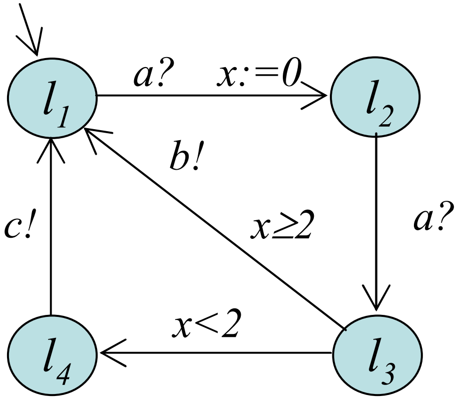
E<> Purpose3.goal



```
out(IGrasp);    //set level to 5
silence(500);
in(OSetLevel,0);
silence(1000);
in(OSetLevel,1);
silence(1000);
in(OSetLevel,2);
silence(1000);
in(OSetLevel,3);
silence(1000);
in(OSetLevel,4);
silence(1000);
in(OSetLevel,5);
out(IRelease);

out(IGrasp);    //touch To Off
silence(200);
out(IRelease);
in(OSetLevel,0);

out(IGrasp);    //touch To On
silence(200);
out(IRelease);
in(OSetLevel,5);

silence(2000);
```

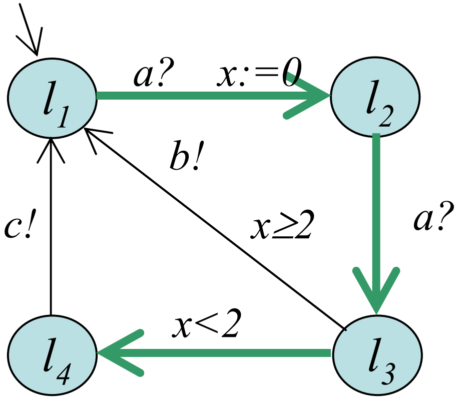
Coverage-Based Test Generation

- Multi purpose testing
- Cover measurement
- Examples:
  - Location coverage,
  - Edge coverage,
  - Definition/use pair coverage



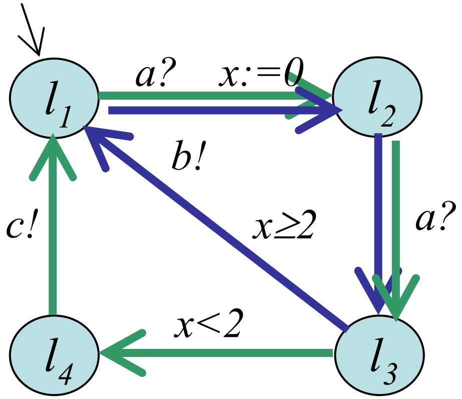
Location Coverage

- Multi purpose testing
- Cover measurement
- Examples:
  - Location coverage,
  - Edge coverage,
  - Definition/use pair coverage



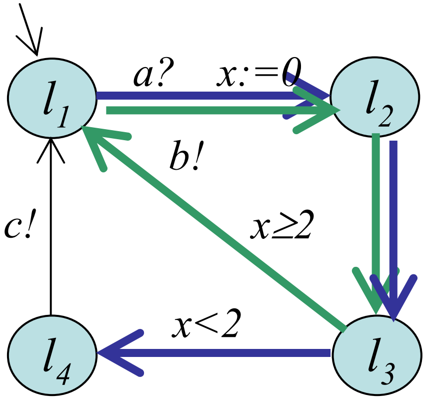
Edge Coverage

- Multi purpose testing
- Cover measurement
- Examples:
  - Location coverage,
  - **Edge coverage**,
  - Definition/use pair coverage



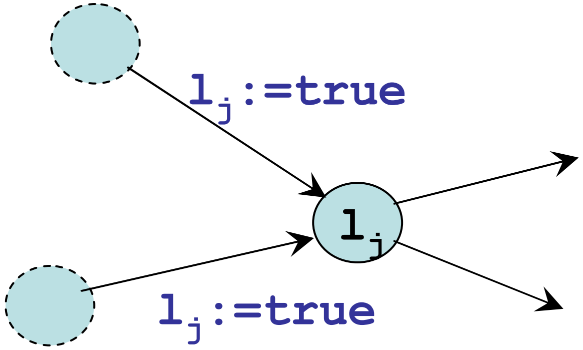
Definition/Use Pair Coverage

- Multi purpose testing
- Cover measurement
- Examples:
  - Location Coverage,
  - Edge Coverage,
  - Definition/Use Pair Coverage



Implementing Location Coverage

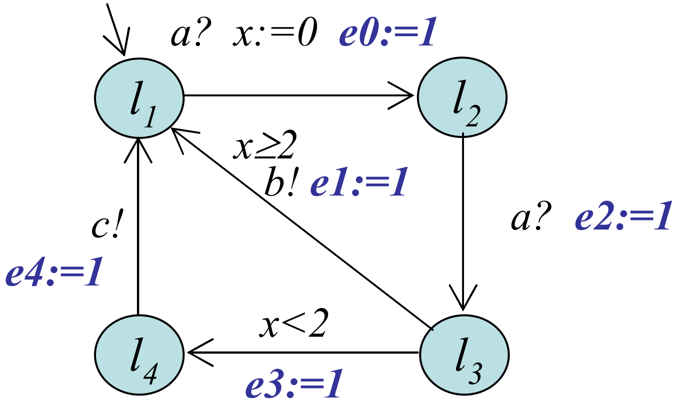
- Test sequence traversing all locations
- Encoding:
  - Enumerate locations $l_0,\dots,l_n$
  - Add an auxiliary variable l_i for each location
  - Label each ingoing edge to location i with l_i:=true
  - Mark initial visited l_0:=true
- Check: E<>(l_0=true $\wedge$ … $\wedge$ l_n=true)



UPPAAL CO√ER

Implementing Edge Coverage

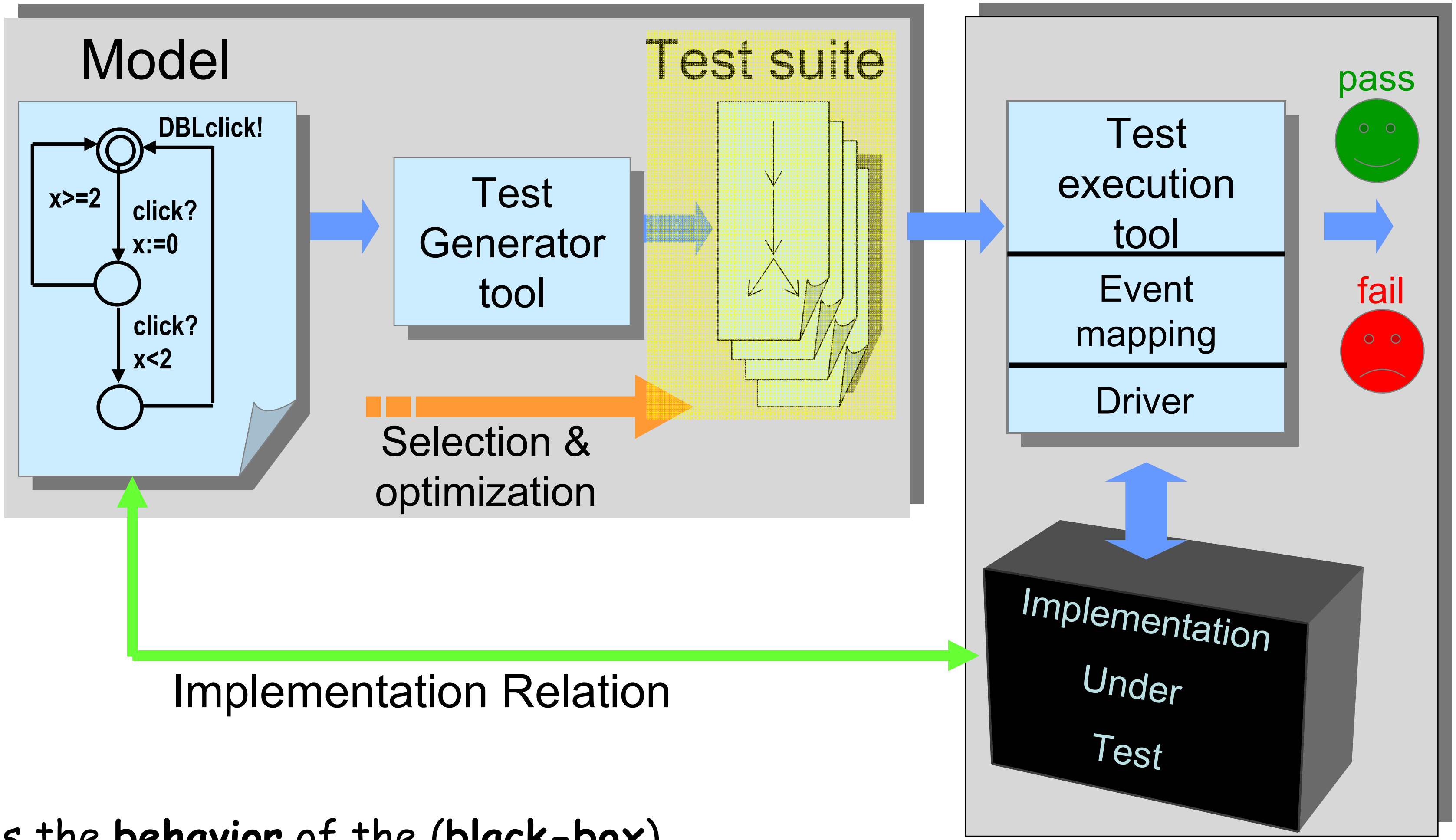
- Test sequence traversing all edges
- Encoding:
  - Enumerate edges $\mathtt{e_0,\ldots,e_n}$
  - Add auxiliary variable $\mathtt{e_i}$ for each edge
  - Label each edge $\mathtt{e_i}$`:=true`
- Check: `E<>(` $\mathtt{e_0}$`=true` ∧ … ∧ $\mathtt{e_n}$`=true )`



Model-Based On-line Testing of Timed Systems

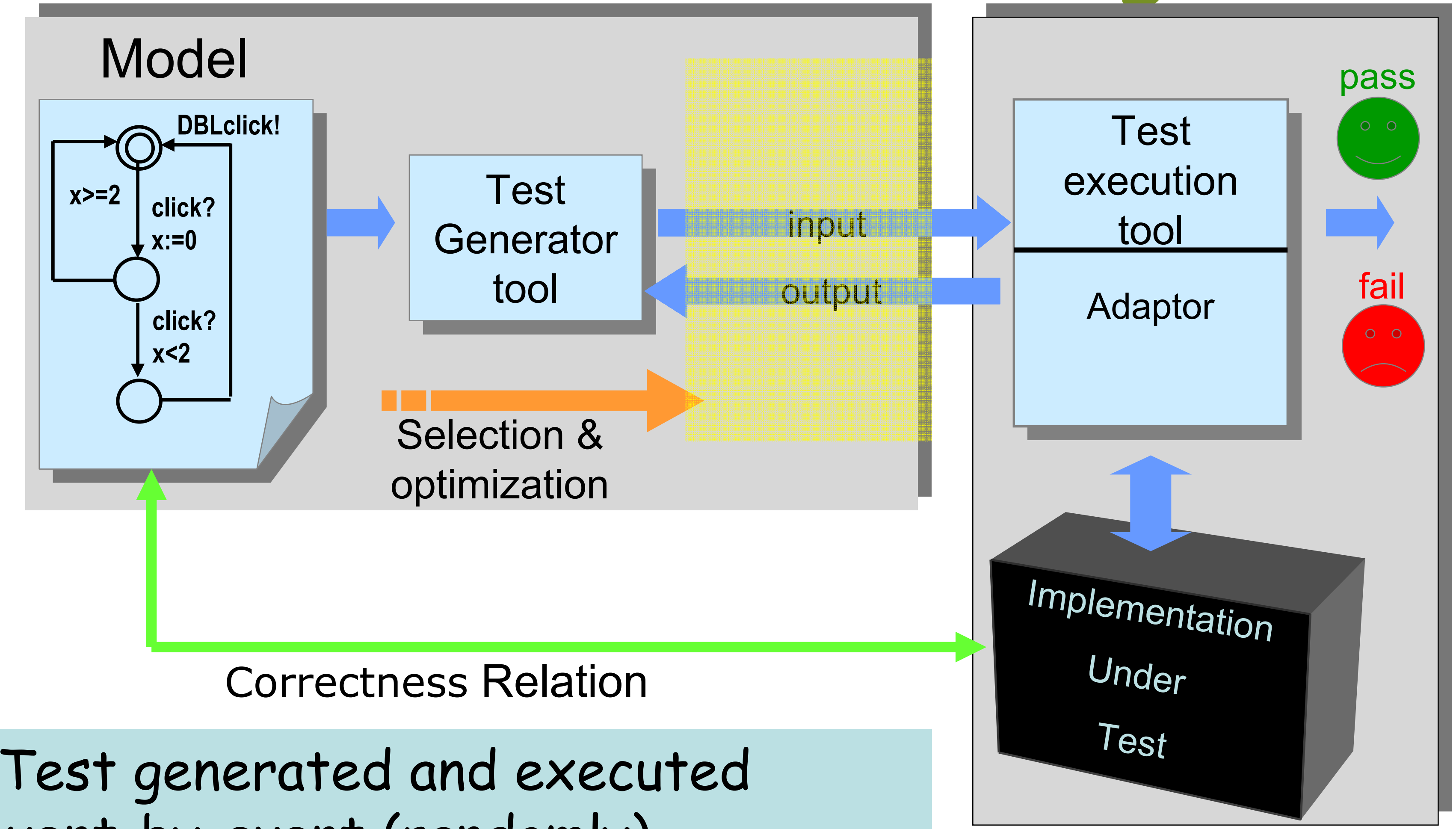
Autom ed Model-Based Off-line Conformance testing

Recall...



Does the **behavior** of the (**black-box**) implementation ***comply*** to that of the specification?

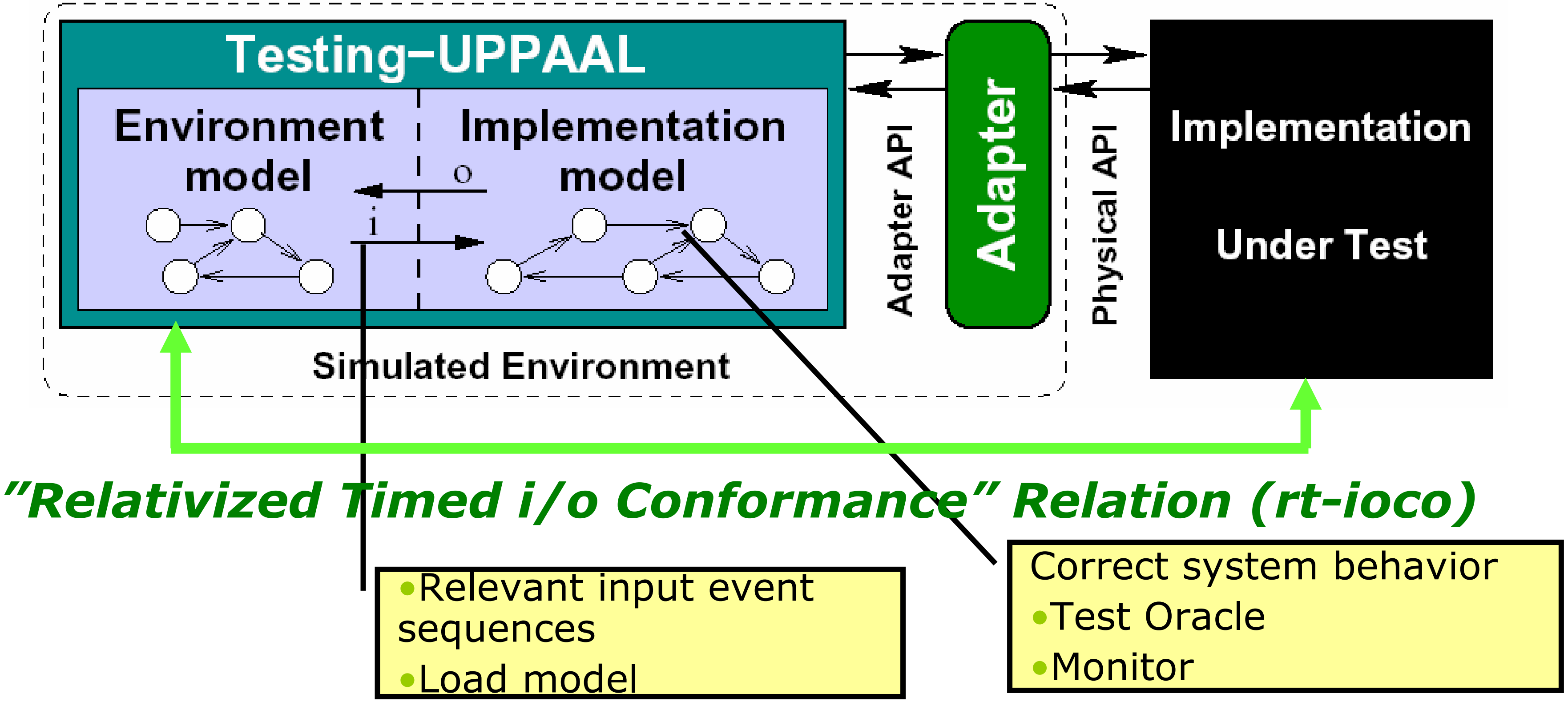
Automated Model-Based On-line Conformance testing



- Test generated and executed event-by-event (randomly)
- A.K.A. on-the-fly testing

The Framework of Uppaal-TRON

- ***UppAal Timed Automata*** *Network: Env || IUT*



"Relativized Timed i/o Conformance" Relation (rt-ioco)

- Relevant input event sequences
- Load model

Correct system behavior
- Test Oracle
- Monitor

- *Complete and sound algorithm*
- Efficient symbolic reachability algorithms
- **Uppaal-TRON:** Testing Real-time Systems ONline
- Release 1.4 http://www.cs.aau.dk/~marius/tron/

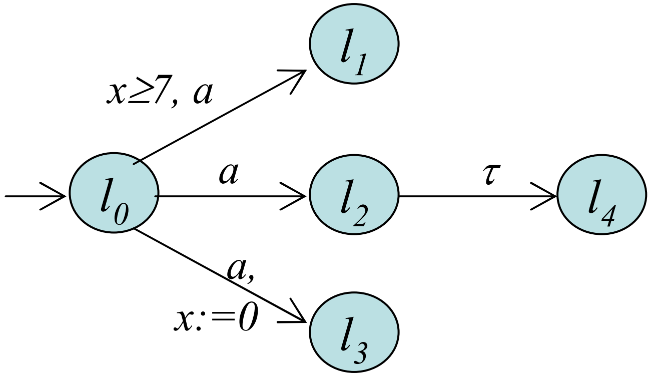
On-line Testing

- Characteristica
  - very imaginative, “ingenious” tests sequences
  - long test sequences
  - stressful load
  - effective fault detection
- Tools exists but mostly NON-real-time
  - **So-far systematic and explicit handling of real-time constraints missing**

State-set Computation

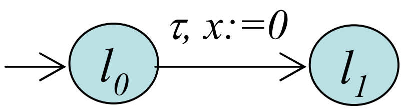
- Compute all potential states the model can occupy after the timed trace $\varepsilon_0, i_0, \varepsilon_1, o_1, \varepsilon_2, i_2, o_2, \ldots$
- *Let Z be a set of states*

Z* after *a: *possible states after a (and τ^*)*



{ ⟨l_0,x=3⟩ } **after** *a* =
{ ⟨l_2,x=3⟩, ⟨l_4, x=3⟩, ⟨l_3, x=0⟩ }

***Z* after ε** :*possible states after τ^* and ε_i , totaling a delay of ε*



{ ⟨l_0,x=0⟩} **after** 4 =
{ ⟨l_0,x=4⟩, ⟨l_1, 0 ≤ x ≤ 4⟩ }

$\langle l_0, x=0 \rangle \xrightarrow{1} \langle l_0, x=1 \rangle \xrightarrow{\tau} \langle l_1, x=0 \rangle \xrightarrow{3} \langle l_1, x=3 \rangle$

Algorithm Idea:

State-set tracking

- Dynamically compute all potential states that the model M can reach after the timed trace $\varepsilon_0, i_0, \varepsilon_1, o_1, \varepsilon_2, i_2, o_2, \ldots$

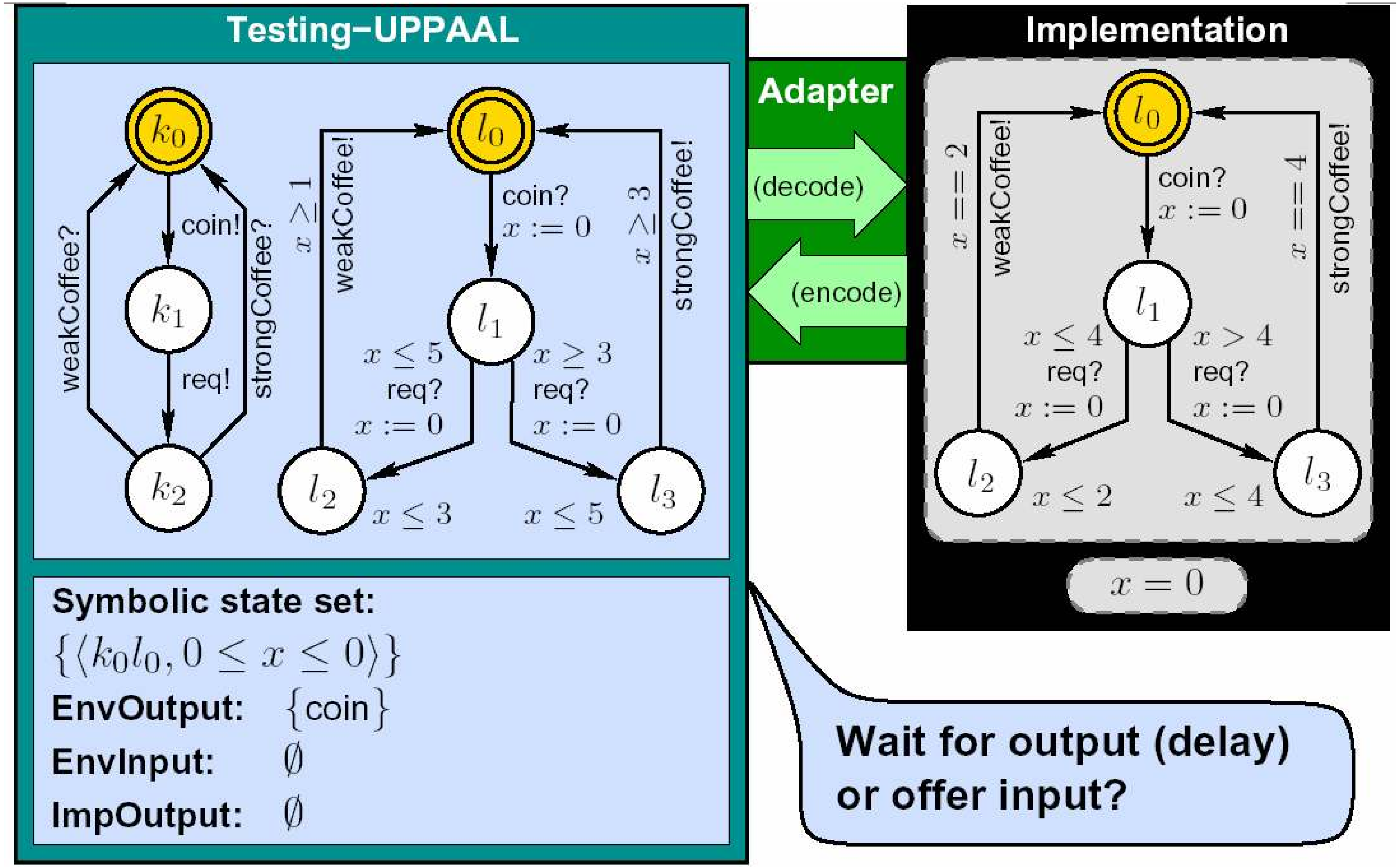
[Tripakis] Failure Diagnosis

- *Z = M* ***after*** $(\varepsilon_0, i_0, \varepsilon_1, o_1, \varepsilon_2, i_2, o_2)$
- *If* $Z=\emptyset$ *then IUT has made a computation not in model:* ***FAIL***
- *i is a relevant input in Env iff* $i \in$ *EnvOutput(Z)*

Uppaal-TRON On-line Testing Algorithm (skeleton)

```
Algorithm TestGenExe (S, E, IUT, T ) returns {pass, fail}
Z := {(s0, e0)}.
while $Z \neq \emptyset \wedge$ #iterations $\leq$ T do either randomly:
    1.  // offer an input
        if EnvOutput(Z) $\neq \emptyset$
            randomly choose $i \in$ EnvOutput(Z)
            send i to IUT
            Z := Z After i
    2.  // wait d for an output
        randomly choose d $\in$ Delays(Z)
        wait (for d time units or output o at $d' \leq d$)
        if o occurred then
                Z := Z After d′
                Z := Z After o // may become $\emptyset$ ($\Rightarrow$fail)
        else
                Z := Z After d  // no output within d delay
    3.  restart:
        Z := {(s0, e0)}, reset IUT //reset and restart
if Z = $\emptyset$ then return fail else return pass
```

On-line Testing Example



Symbolic state set:

$\{\langle k_0 l_0, 0 \leq x \leq 0\rangle\}$

EnvOutput: $\{$coin$\}$

EnvInput: $\emptyset$

ImpOutput: $\emptyset$

Wait for output (delay) or offer input?

Tools for Model-Based Testing

Academic MBT Tools

Tool name	Tool provider	Modeling notation	Testing method	Short description
Lutess		Lustre		
Lurette		Lustre		
GATeL		Lustre	CLP	
Autofocus		Autofocus	CLP	
Conformance Kit		EFSM	FSM	
Phact		EFSM	FSM	
TVEDA		SDL, Estelle	FSM	
AsmL		AsmL	FSM?	

Academic MBT Tools (cont'd)

Tool name	Tool provider	Modeling notation	Testing method	Short description
Cooper		LTS (Basic LOTOS)	LTS	
TGV	Irisa and Verimag, France	LTS-API (LOTOS, SDL, UML)	LTS	
TorX	Twente University	LTS (LOTOS, Promela, FSP)	LTS	
STG	Irisa, France	NTIF	LTS	
AGEDIS		UML/AML	LTS	
Uppaal Tron	Aalborg University	TA	TLTS	
Uppaal Cover	Uppsala University	TA	TLTS	

Commercial MBT Tools

Tool name	Tool type	Manufacturer	Web link	Modeling notation	Short description
AETG	1	Telcordia Technologies	aetgweb.argreenhouse.com	Model of input data domain	The AETG Web Service generates pairwise test cases.
Case Maker	1	Diaz & Hilterscheid Unternehmensberatung GmbH	www.casemakerinternational.com	Model of input data domain	CaseMaker uses the Pairwise method to compute test cases from input parameter domain specifications and constraints specified by business rules.
Conformiq Test Generator	3	Conformiq	www.conformiq.com	UML Statecharts	In Conformiq Test Generator, UML statecharts constitute a high-level graphical test script. Conformiq Test Generator is capable of selecting from the statechart models a large amount of test case variants and of executing them against tested systems.
CTesK, JTesK	3	UniTESK	www.unitesk.com	Pre-Post extensions of programming languages	UniTESK technology is a technology of software testing based on formal specifications. Specifications are written using specialized extensions of traditional programming languages. CTesK and JTesK can use a formal representation of requirements as a source of test development.
LEIRIOS Test Generator - LTG/B	3	LEIRIOS Technologies	www.leirios.com	B notation	LTG/B generates test cases and executable test scripts from a B model. It supports requirements traceability.
LEIRIOS Test Generator - LTG/UML	3	LEIRIOS Technologies	www.leirios.com	UML 2.0	LTG/UML generates test cases and executable test scripts from a UML 2.0 model. It supports requirements raceability.
MaTeLo	2	All4Tec	www.all4tec.net	Model usage editor using Markov chain	MaTeLo is based on Statistical Usage Testing and generates test caes from a usage model of the system under test.
Qtronic	3	Conformiq	www.conformiq.com		Qtronic derives tests from a design model of the system under test. This tool supports multi-threaded and concurrent models, timing constraints, and testing of nondeterministic systems.

Legend for Tool Type Column:
Category 1: Generation of Test Input Data from a Domain Model
Category 2: Generation of Test Cases from a Model of the Environment
Category 3: Generation of Test Cases with Oracles from a Behavioral Model
Category 4: Generation of Test Scripts from Abstract Tests

Commercial MBT Tools (cont'd)

Tool name	Tool type	Manufacturer	Web link	Modeling notation	Short description
Rave	3	T-VEC	www.t-vec.com	Tabular notation	Rave generates test cases from a tabular model. The test cases are then transformed into test drivers.
Reactis	3	Reactive Systems	www.reactive-systems.com	Mathlab, Simulink, Stateflow	Reactis generates tests from Simulink and Stateflow models. This tool targets embedded control software.
SmartTest	1	Smartware Technologies	www.smartwaretechnologies.com/smarttestprod.htm	Model of input data domain	The SmartTest test case generation engine uses pairwise techniques.
Statemate Automatic Test Generator / Rhapsody Automatic Test Generator (ATG)	3	i-Logix	www.Ilogix.com	Statemate Statcharts and UML State Machine	ATG is a module of Telelogic(I-Logix) Statemate and Rhapsody products. It allows test case generation from a statechart model of the System.
TAU Tester	4	Telelogic	www.telelogic.com/products/tau/tester/index.cfm	TTCN-3	An integrated test development and execution environment for TTCN-3 tests
Test Cover	1	Testcover.com	www.testcover.com	Model of input data domain	The Testcover.com Web Service generates test cases from a model of domain requirements. It uses pairwise techniques.
T-Vec Tester for Simulink - T-Vec Tester for MATRIXx	3	T-Vec	www.t-vec.com	Simulink and MATRIXx	Generates test vectors and test sequences, verifying them in autogenerated code and in the modeling tool simulator.
ZigmaTEST Tools	3	ATS	www.atssoft.com/products/testingtool.htm	Finite State Machine	ZigmaTEST uses an FSM-based test engine that can generate a test sequence to cover state machine transitions.

Legend for Tool Type Column:
Category 1: Generation of Test Input Data from a Domain Model
Category 2: Generation of Test Cases from a Model of the Environment
Category 3: Generation of Test Cases with Oracles from a Behavioral Model
Category 4: Generation of Test Scripts from Abstract Tests

Summary

Benefits of Model-Based Testing

- Automated testing

full automation : test generation + execution + analysis

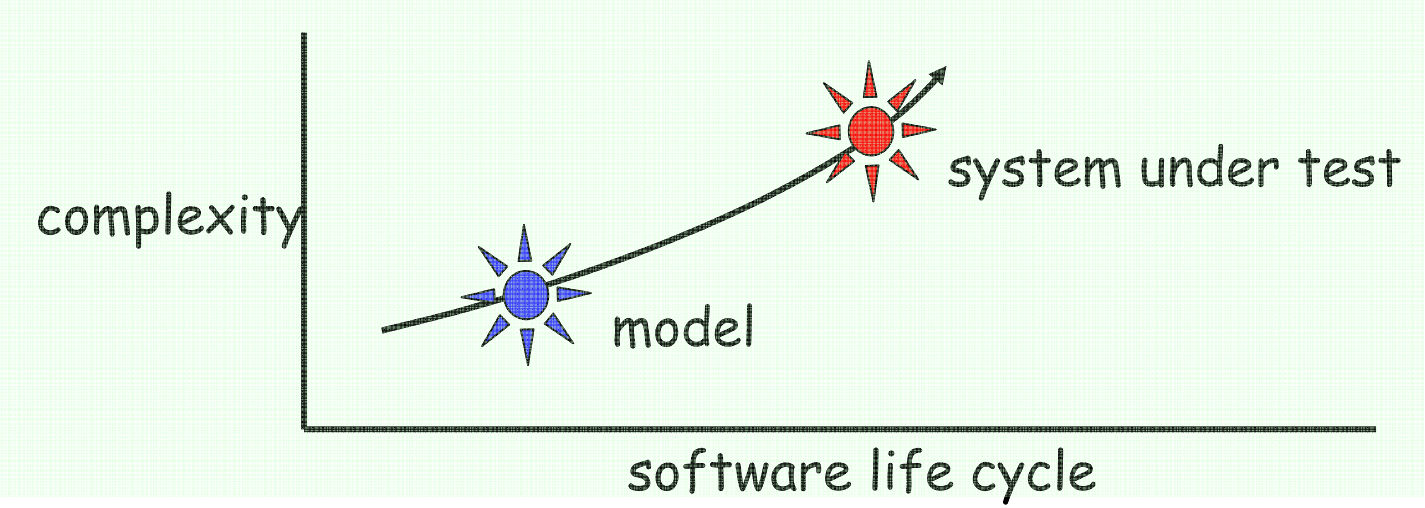
- Early testing

design errors found during validation of model

- Systematic and rigorous testing

model is precise and unambiguous basis for testing

longer, cheaper, more flexible, and provably correct tests



Obstacles to Model-Based Testing

- Comfort factor
  - This is not your parents' test automation
- Skill sets
  - Need testers who can <u>design</u>
- Expectations
  - Models can be a significant upfront investment
  - Will never catch all the bugs
- Metrics
  - Bad metrics: bug counts, number of test cases
  - Better metrics: spec coverage, code coverage

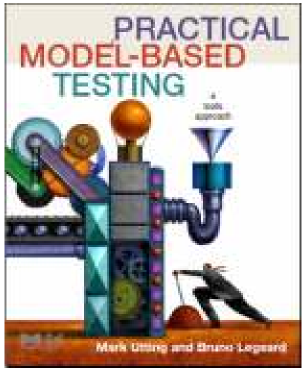
Main Readings

- Gerard J. Holzmann. Design and Validation of Computer Protocols, Chapter 9 "**Conformance Testing**"

- Jan Tretmans. **Testing Concurrent System – a Formal Approach**. In Proc. 10th Int'l Conf. on Concurrency Theory (CONCUR'99), Eindhoven, The Netherlands, August 1999, LNCS 1664. (http://www.springerlink.com/content/jf8b4tewecjlwrrq/)

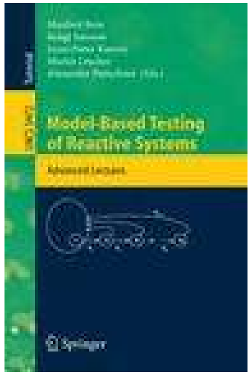
- Anders Hessel, Kim Guldstrand Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Formal Methods and Testing, chapter "**Automated Model-Based Conformance Testing of Real-Time Systems**". Springer-Verlag, 2006.

Further Readings



- Model-based testing website:

  www.model-based-testing.org



- Books:

  "Practical Model-Based Testing: A Tools Approach" by Mark Utting and Bruno Legeard, Morgan-Kaufmann, 2007.

  "Model-Based Testing of Reactive Systems", Advanced Lectures edited by M. Broy et al., LNCS 3472, Springer, 2005.

  "Black-Box Testing : Techniques for Functional Testing of Software and Systems" by Boris Beizer

  "Testing Object-Oriented Systems: Models, Patterns, and Tools" by Robert Binder

  "Software Testing: A Craftsman's Approach" by Paul Jorgensen