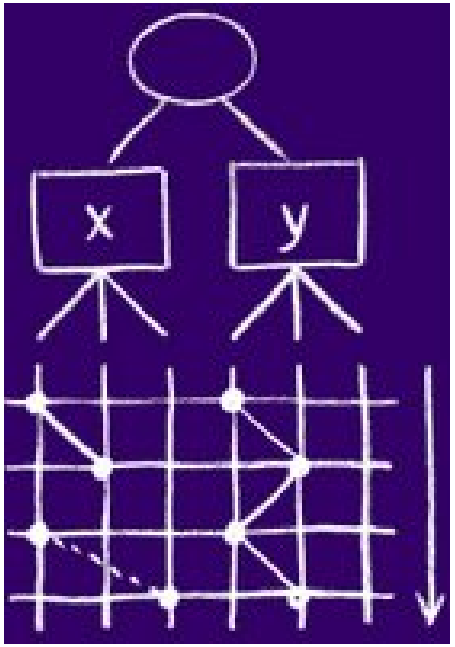
Model-Based Testing of Reactive Systems

Principles, Methods, and Tools

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



Lecture Plan

12:30 – 13:15	**Model-Based Testing: Principles, Methods and Tools**
13:15 – 13:25	break
13:25 – 14:10	**Modeling, Verification and Testing of Real-time Systems**
14:10 – 16:00	Tutored exercises

Agenda

- Overview
- Labelled Transition System (LTS)-based testing
- Finite State Machine (FSM)-based testing
- 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

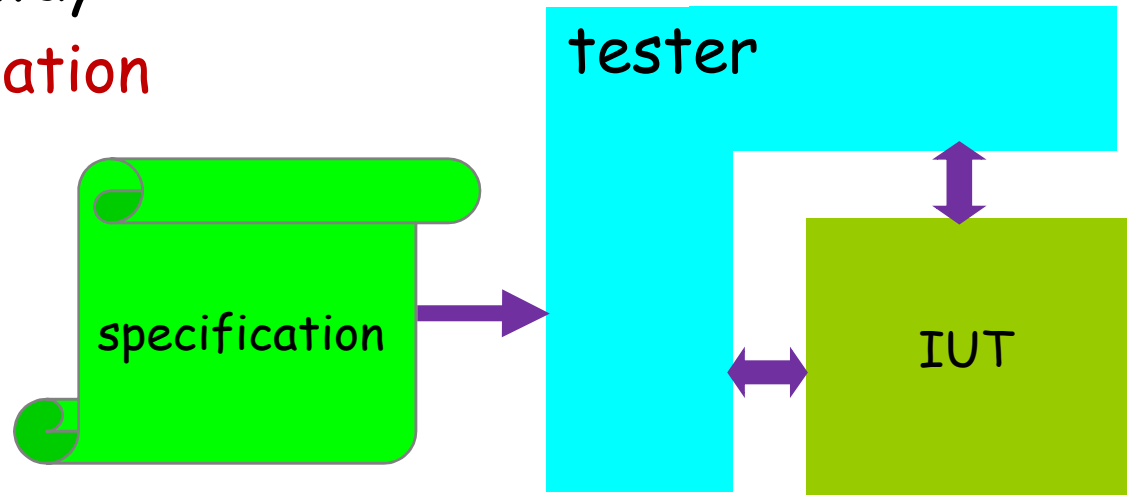
in a controlled way

w.r.t. a specification

rather than on models (which are formal verification or simulation)

rather than by reasoning

to decide whether it passes or fails



IUT: the Implementation Under Test

Model-Based Testing

- Driving forces for MBT:
  - testing effort grows exponentially with complexity
  - testing cannot keep pace with development
  - ever-changing software requirements
  - demands for high-quality software
  - reduced time-to-market

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

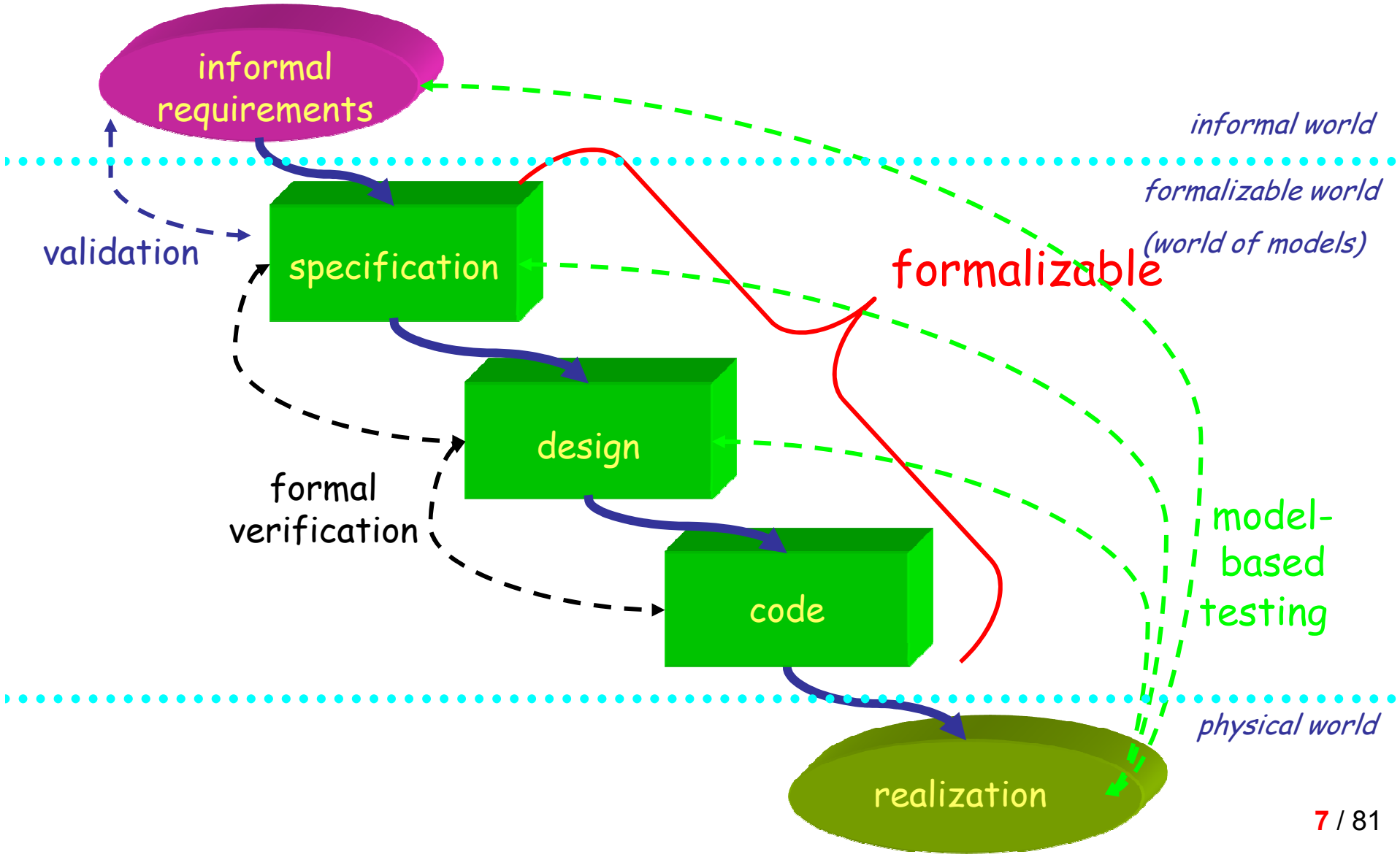
- Model-based testing has potential to combine
  - practice (testing) with
  - theory (formal methods)

Model-Based Testing (cont'd)

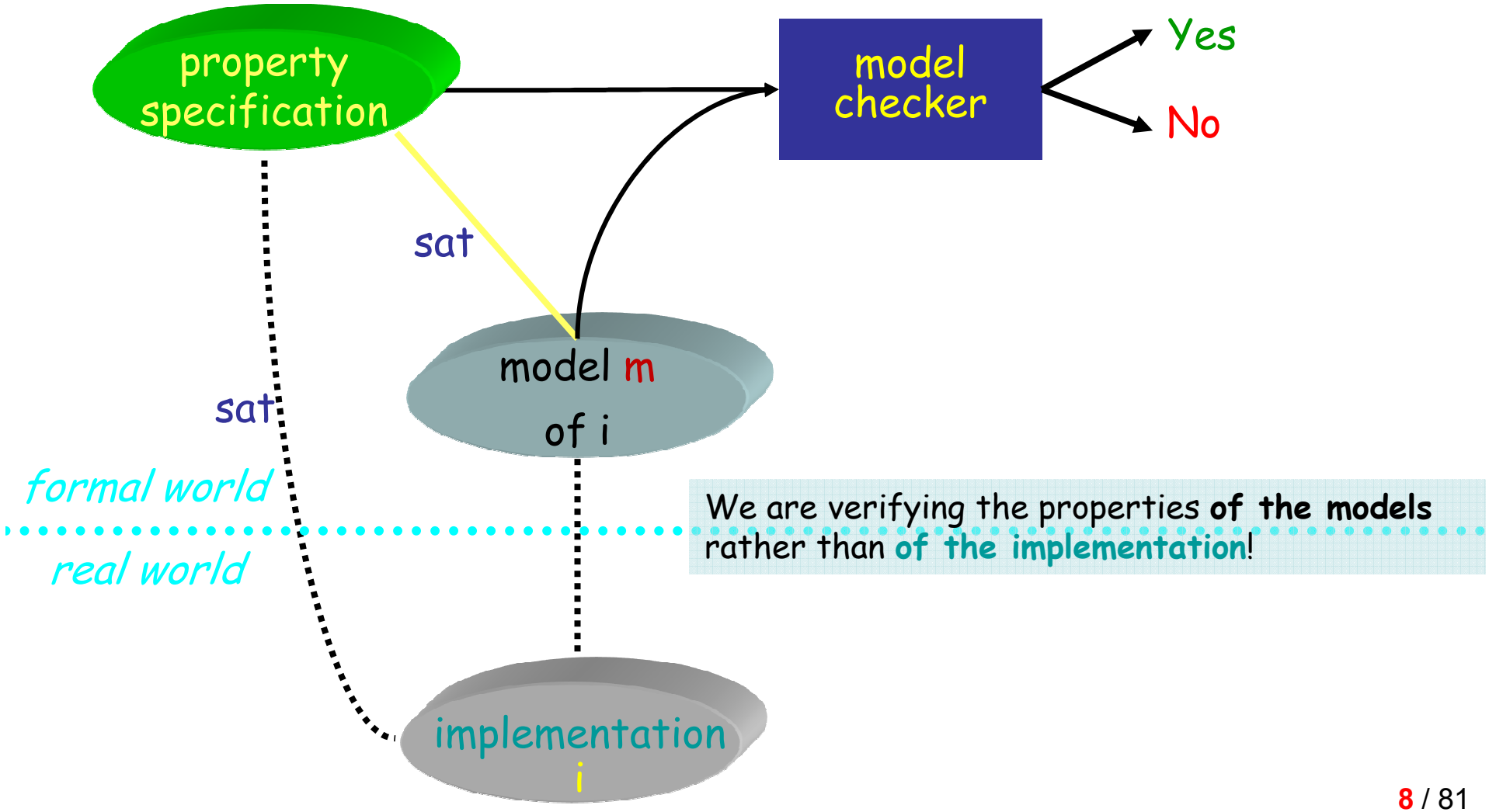
- Essence
  - generating tests from a (formal) model / specification
    - state model, pre/post, CSP, Promela, UML, Spec#, ...
  - testing with respect to a (formal) model / specification

- Benefits
  - promises better, faster, cheaper testing:
    - algorithmic generation of tests and test oracles : tools
    - formal and unambiguous basis for testing
    - measuring the completeness of tests
    - maintenance of tests through model modification

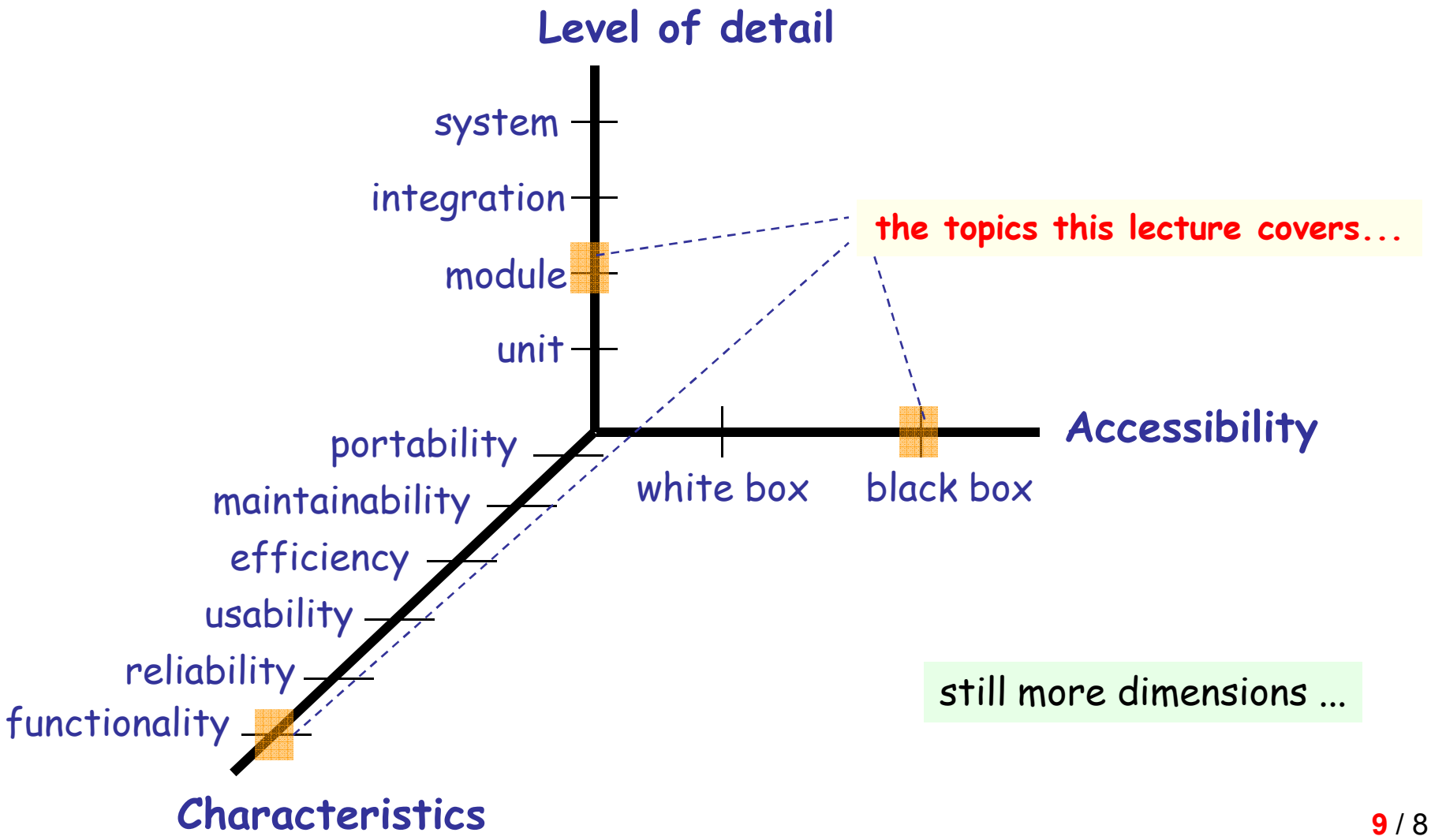
A Model-Based Development Process



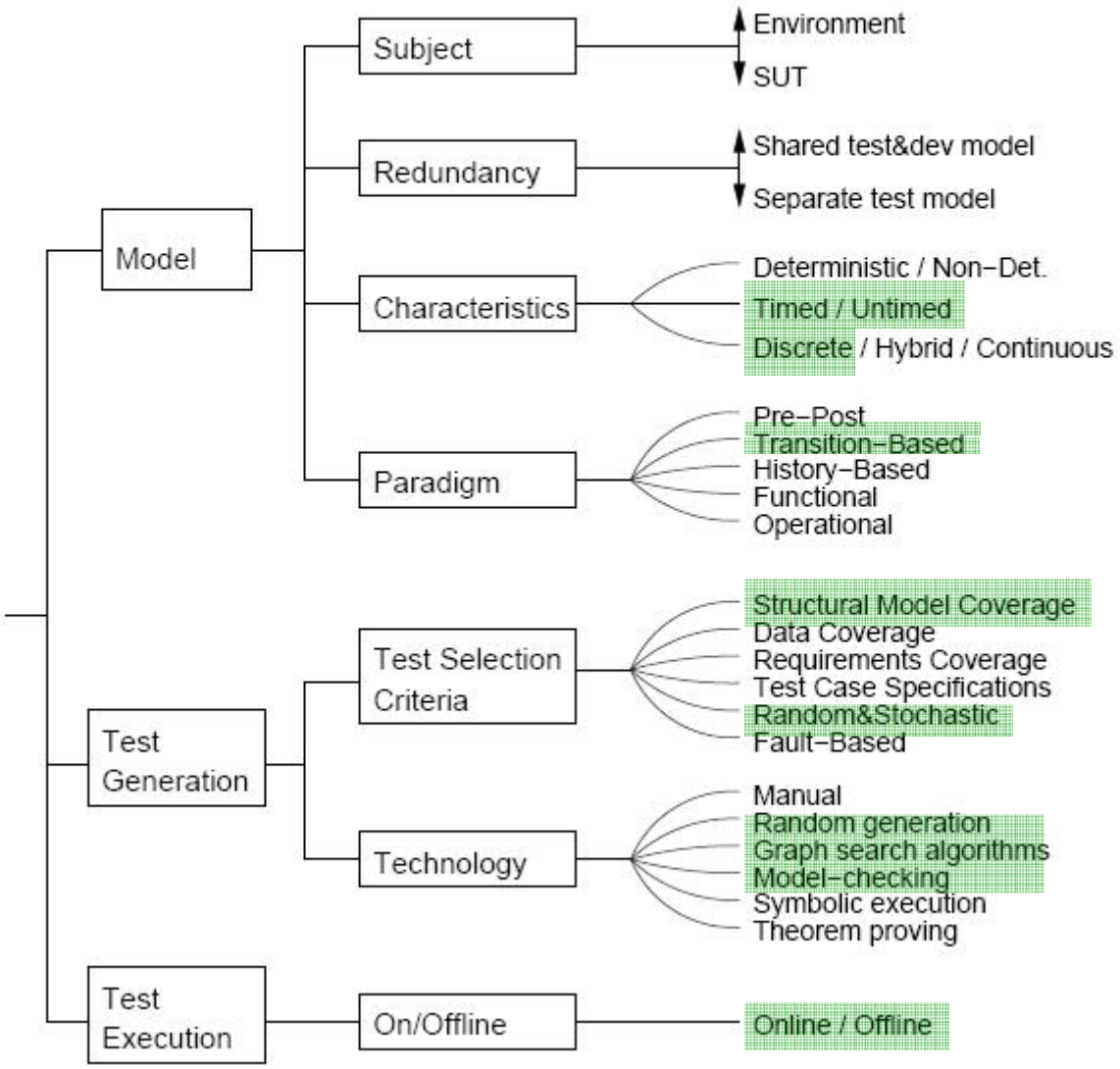
Formal Verification



Types of Testing

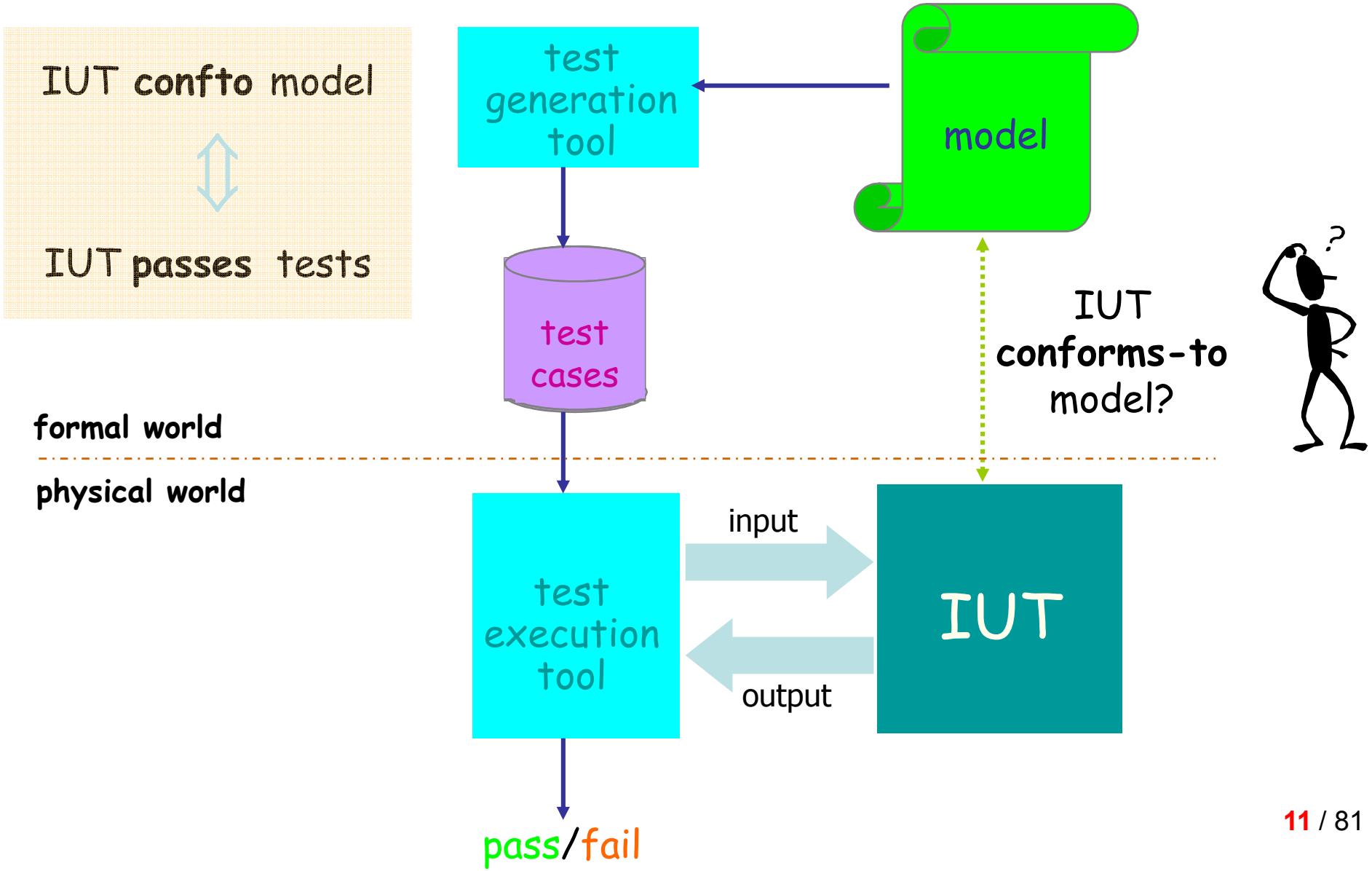


A Taxonomy of Model-Based Testing

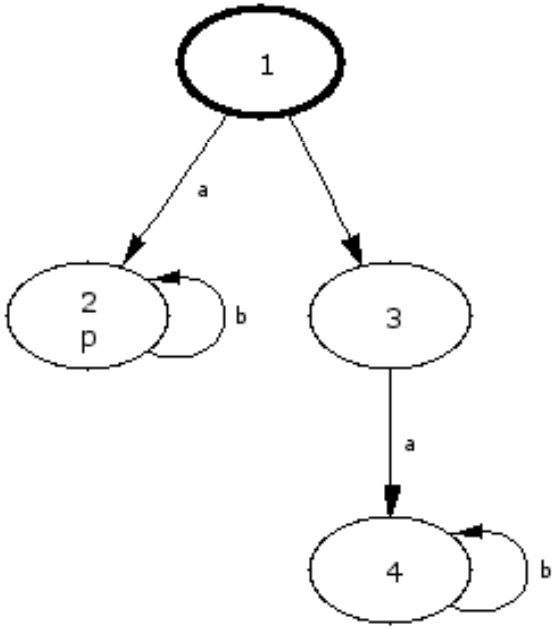


[Mark Utting 2006]

Automated Model-Based Testing



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

- Why LTS-based testing:
  - FSM is required to be "deterministic" and "complete" for testing
  - LTS is more fundamental, more naive, and simpler, thus has better supports for the descriptions of non-determinancy, concurrency and composition
    - FSM has always alternation between inputs and outputs

though sometimes they may be "-"

  - LTS can serve as underlying semantics model for many other formalisms (e.g., FSM, EFSM, and timed models)

Recall...

An example LTS

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

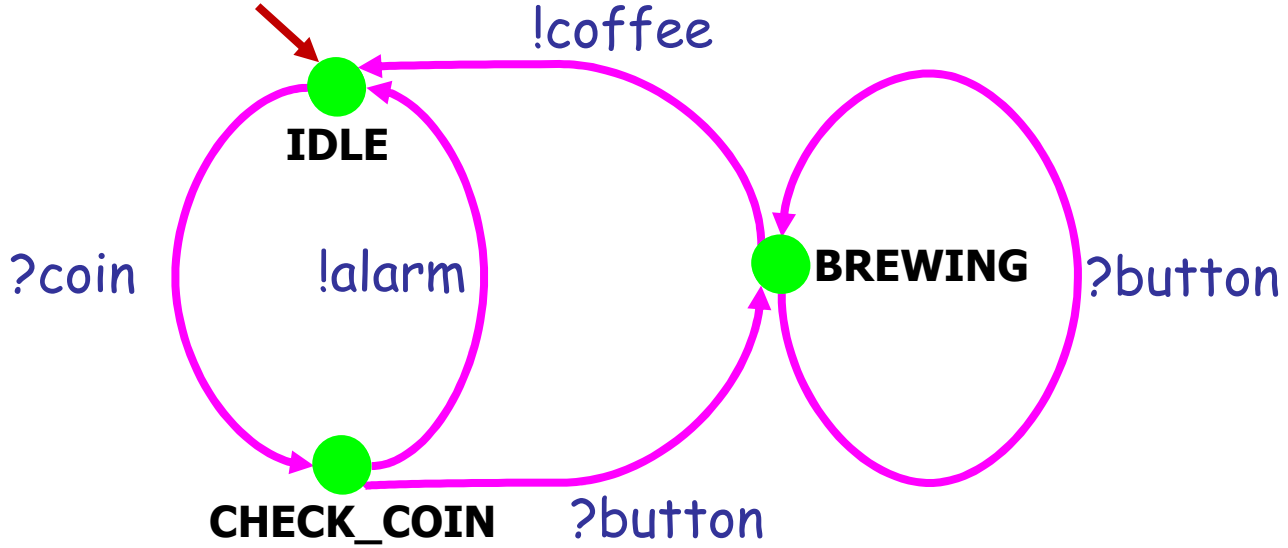
coffee vending machine

states

actions

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

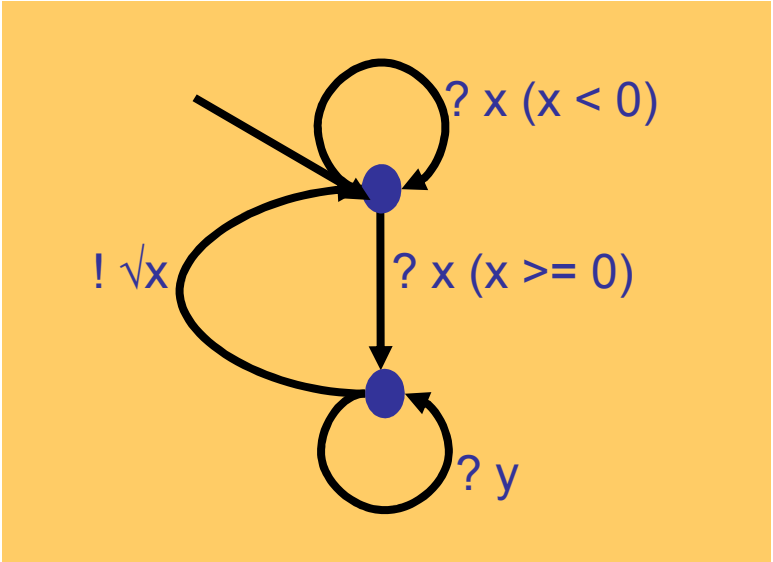
initial state
$s_0 \in S$

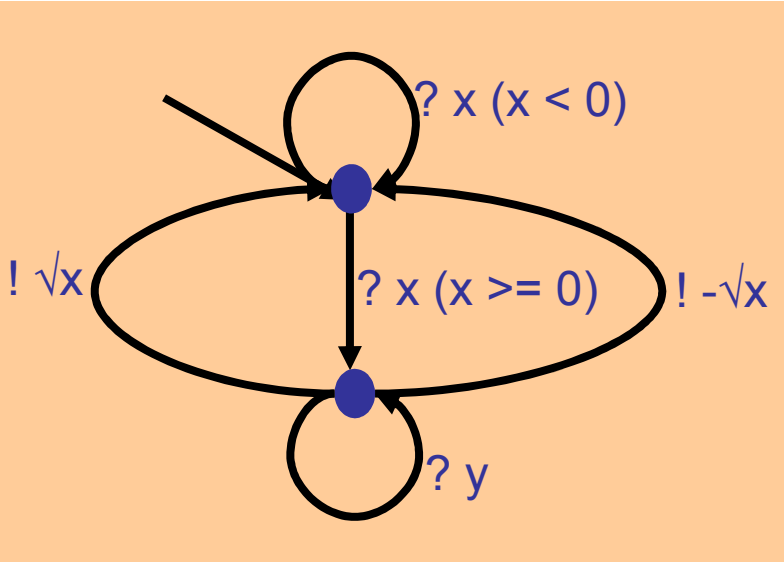


Recall...

Input-Output LTS (IOLTS)

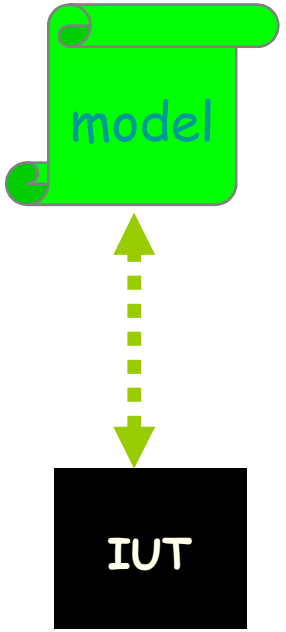
- Special kind of LTS:
  Input-Output Labelled Transition System - IOLTS
  - distinction between outputs (!) and always-enabled inputs (?)
  - implementations modelled as IOLTS
- IOLTS with variables - equation solver for $y^2 = x$:



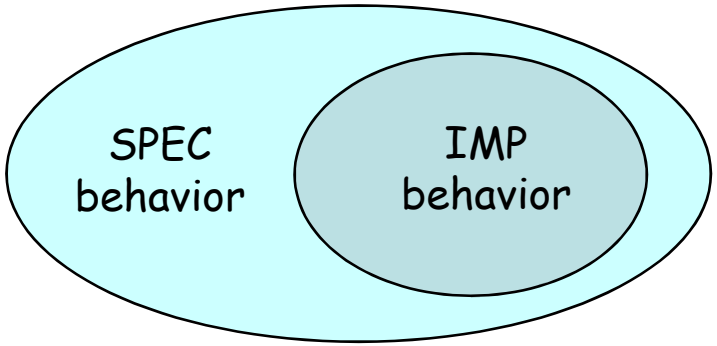


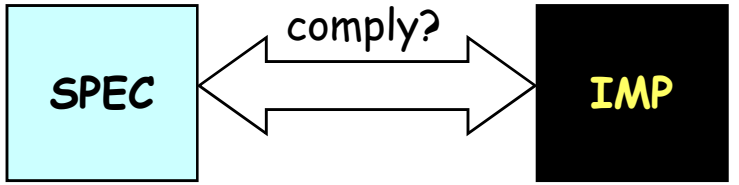
Conformance Relation

- Assume that the Implementation Under Test (IUT) is a black box
  - The internal states 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")
  - the IUT should:
    - **do** what are **required** to do, and
    - **never do** what are **forbidden** to do



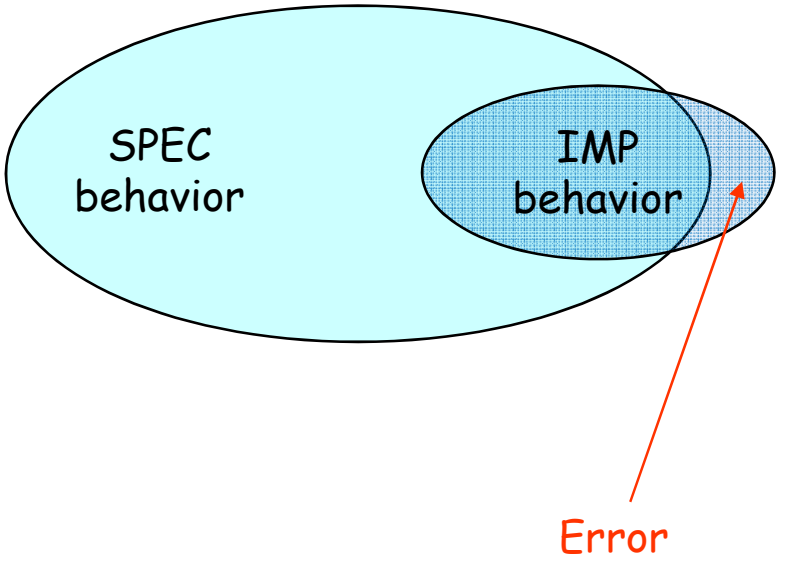
Notion of "Conformance"





what are "behavior"???

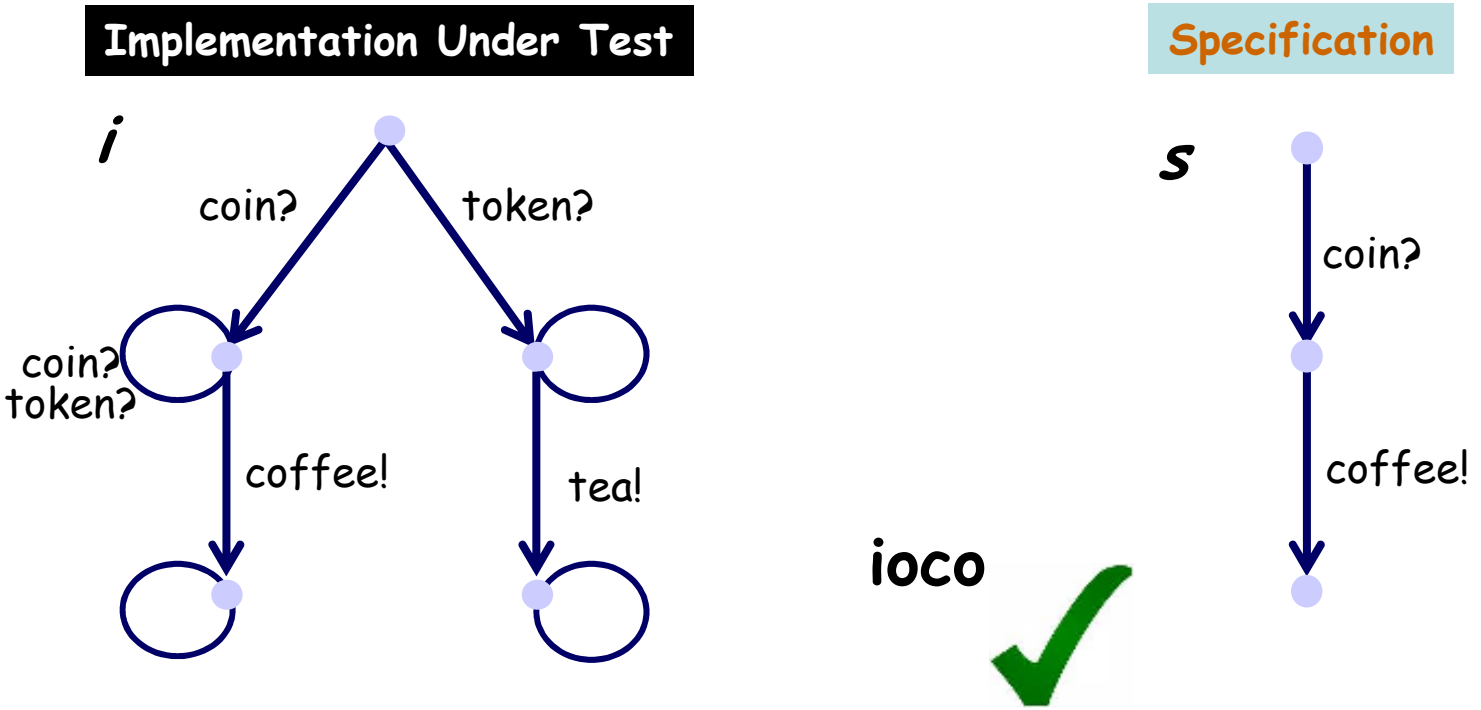
internally, ...
externally, ...



conformance relations

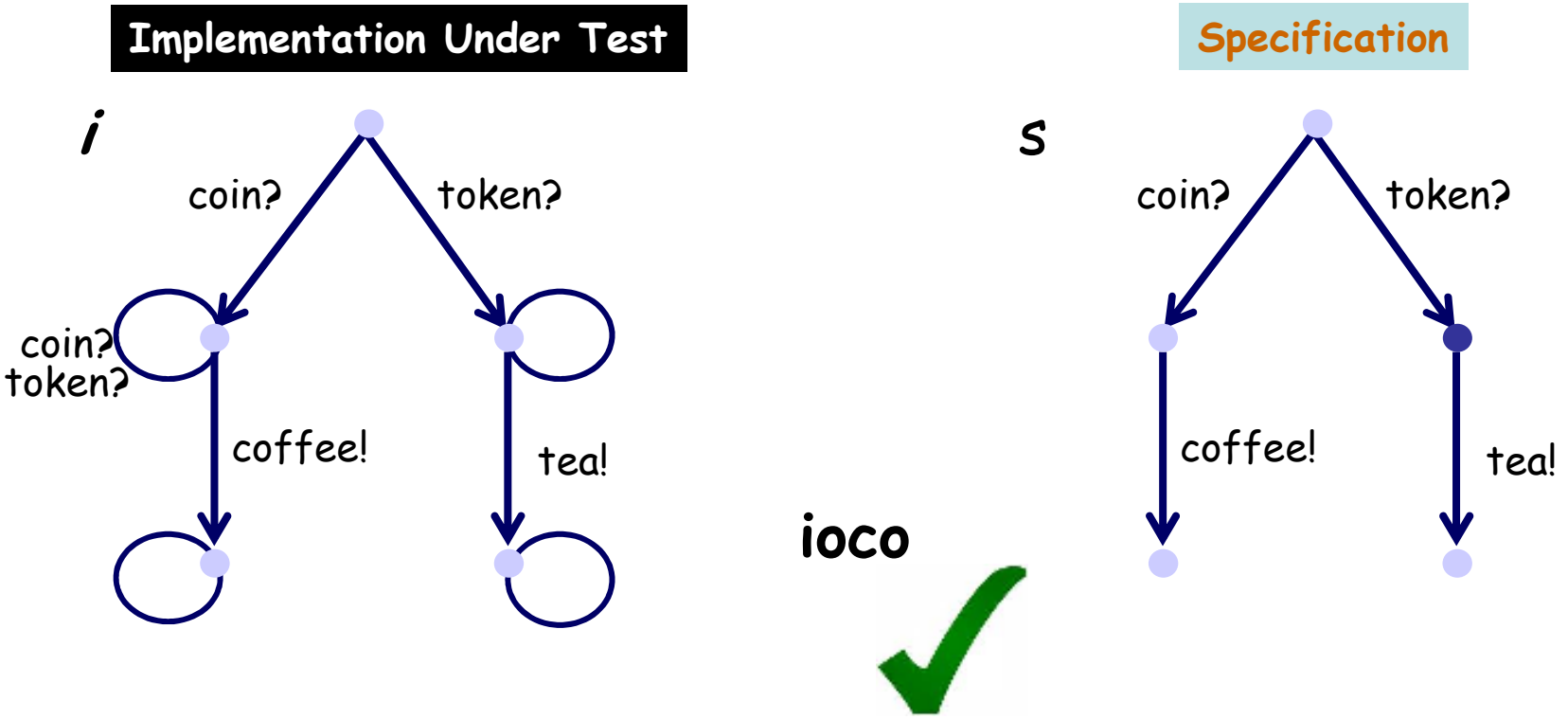
- Trace equivalence, bisimulation
- Trace inclusion
- Input/output sequence inclusion
- Observation sequence inclusion
- ...

i conforms-to s ?? (a)



[Jan Tretmans]

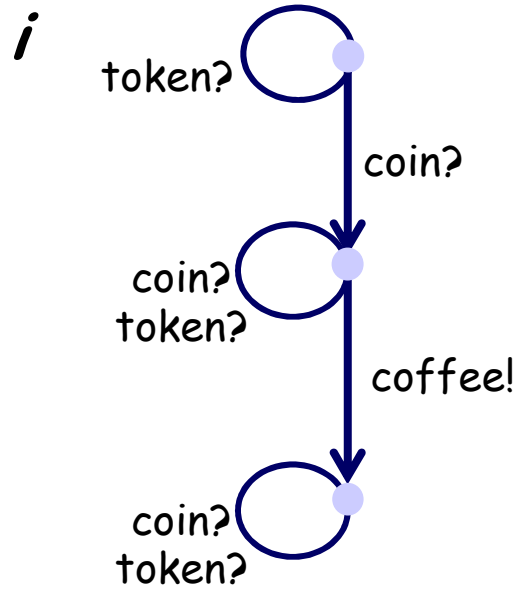
i conforms-to s ?? (b)



[Jan Tretmans]

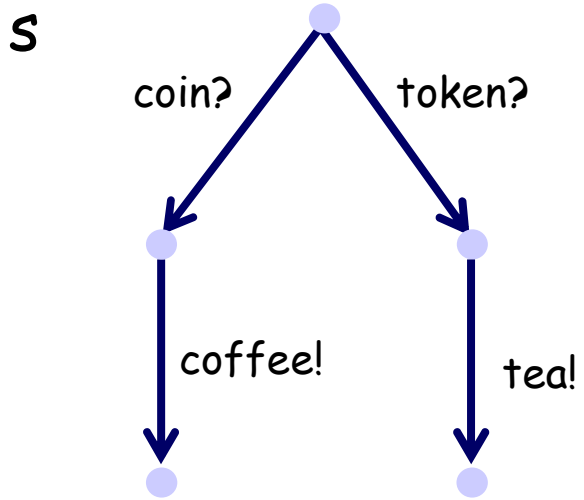
i conforms-to s ?? (c)

Implementation Under Test



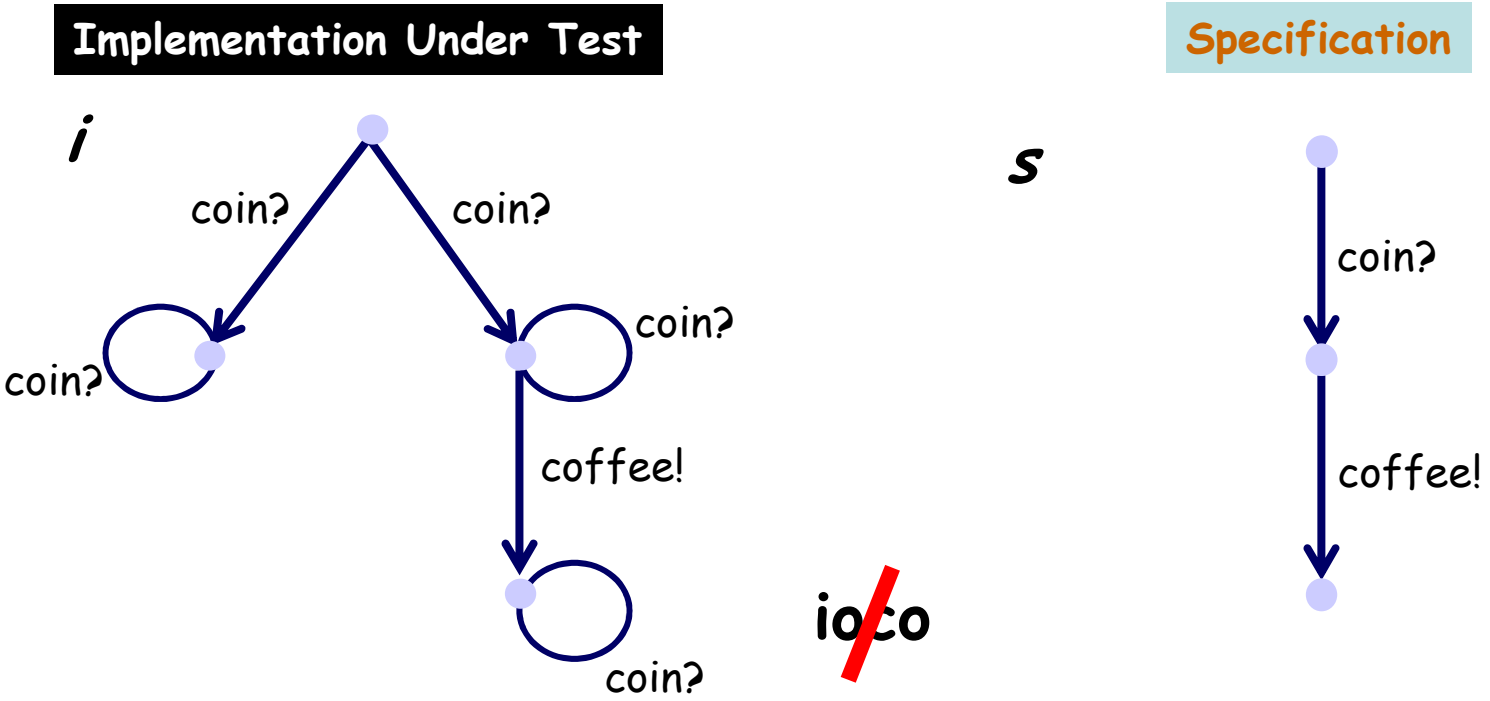
ioco

Specification



[Jan Tretmans].

i conforms-to s ?? (d)



[Jan Tretmans].

Tretman's ioco-coformance

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

"suspension trace"

$$\textbf{\textit{Straces}}(s) = \{\ \sigma \in (L \cup \{\delta\})^* \mid s \stackrel{\sigma}{\Longrightarrow} \}$$

"reachable states"

$$p \textbf{ after } \sigma = \{\ p' \mid p \stackrel{\sigma}{\Longrightarrow} p'\ \}$$

"quiescence"

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

L_u is the subset of output actions of L

"outputs"

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

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

[Jan Tretmans].

ioco: intuitively

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

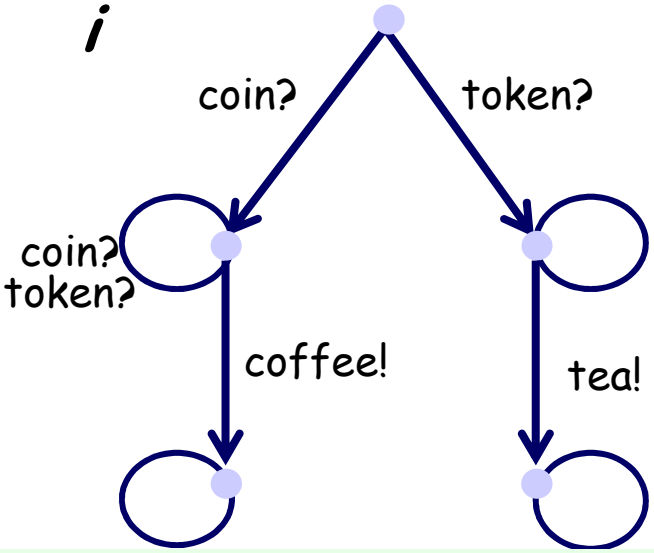
Intuitively:

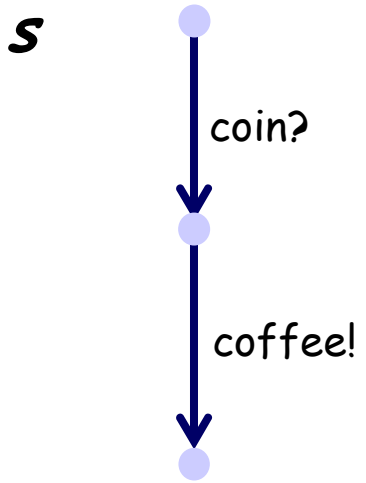
i **ioco**-conforms to s, iff

- if i produces output x after trace σ,
  then s should be able to produce x after σ
- if i cannot produce any output after trace σ, i.e.,
  i produces a quiescence δ after σ,
  then s should also be able to produce δ after σ, i.e.,
  s should not be able to produce any output after σ.

ioco-conformance (a)

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





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

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

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

out (*s* **after** token?) = ∅

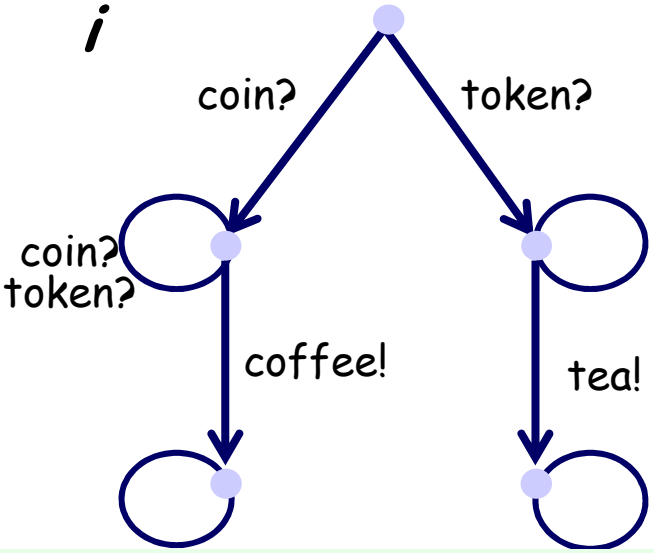
But token? $\notin$ *Straces* (*s*)

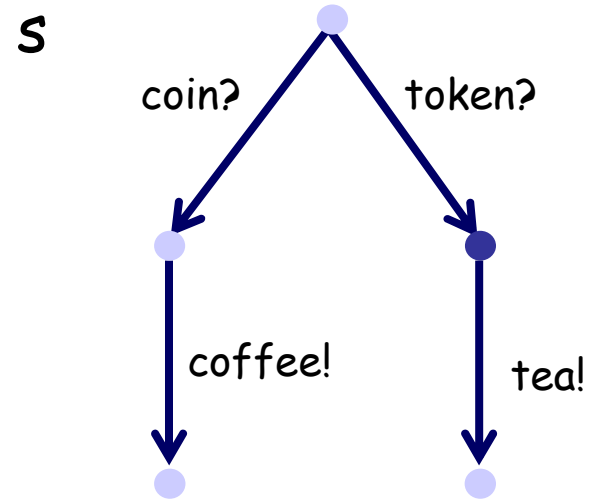
ioco ✓

[Jan Tretmans].

ioco-conformance (b)

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





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

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

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

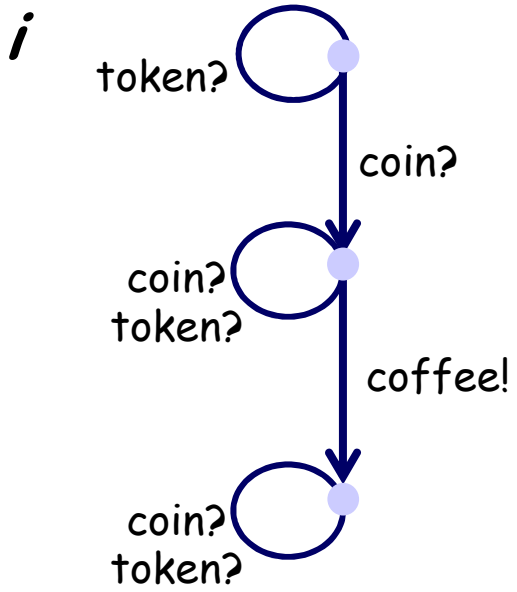
out (*s* **after** token?) = { tea! }

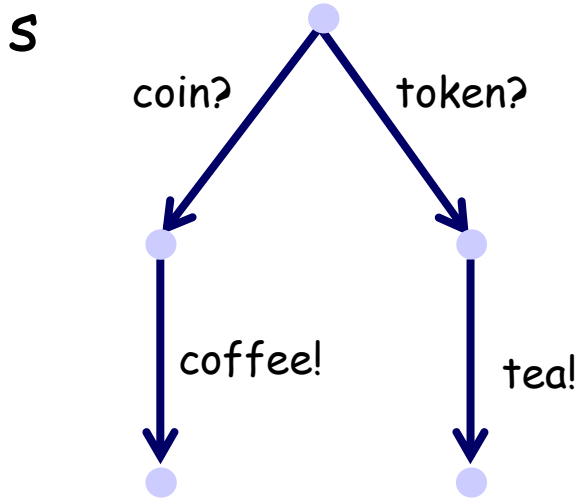
ioco ✓

[Jan Tretmans].

ioco-conformance (c)

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





$out(i \textbf{ after } token?) = \{ \delta \}$

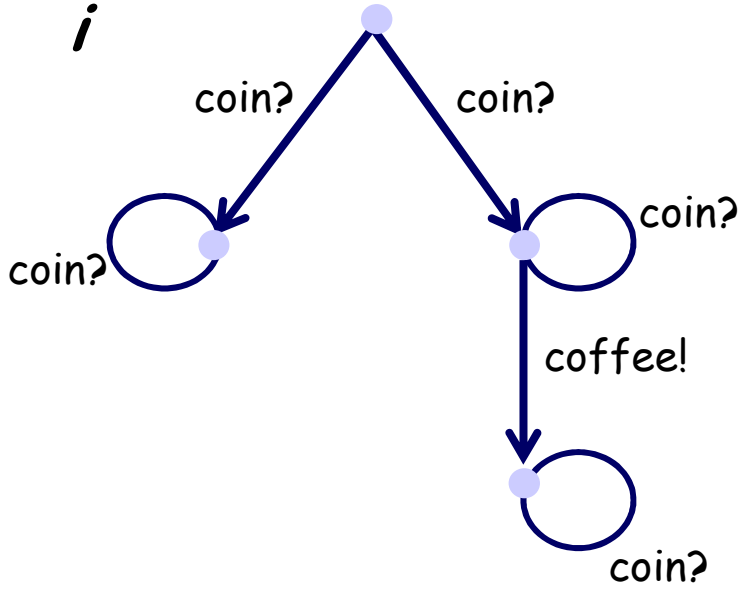
$out(s \textbf{ after } token?) = \{ tea! \}$

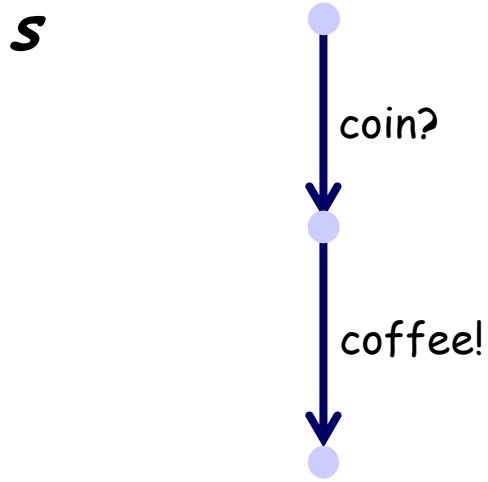
ioco

[Jan Tretmans].

ioco-conformance (d)

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





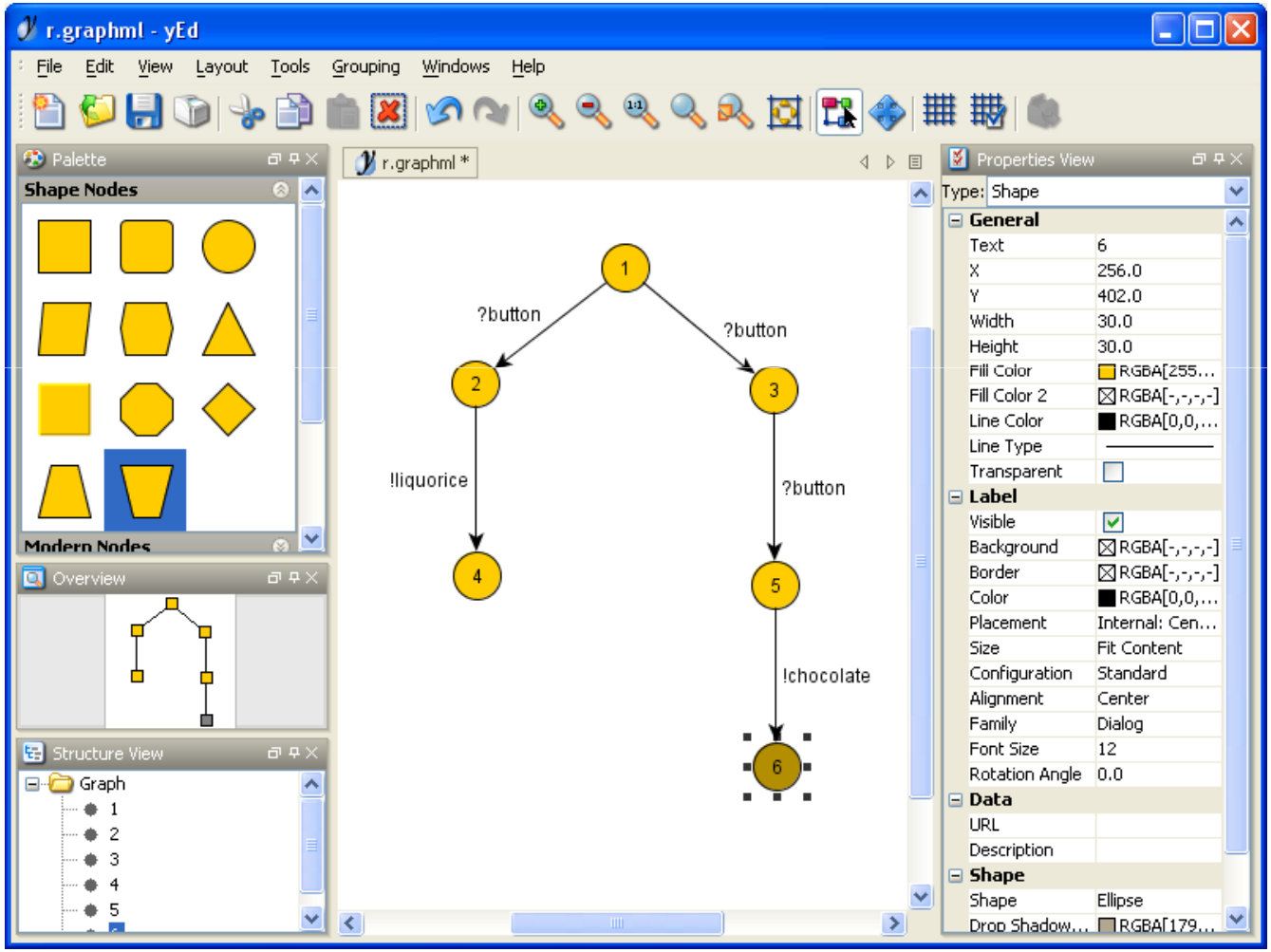
out (*i* **after** coin?) = { δ, coffee! }

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

ioco

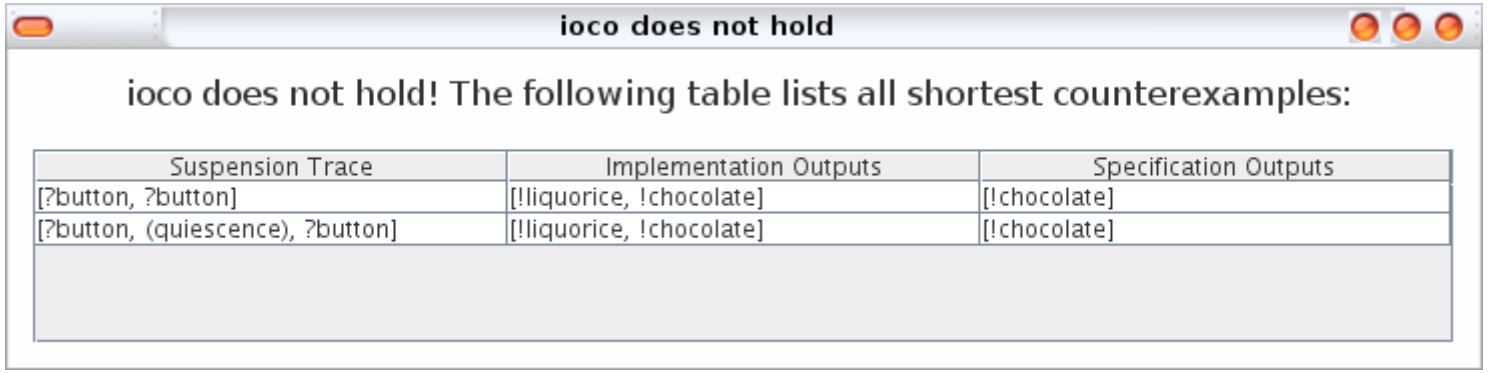
[Jan Tretmans].

LTS Modeling Tool: yEd Java Graph Editor



http://www.yworks.com/en/products_yed_about.html

Conformance Checking Tool: iocoChecker



http://www.cs.ru.nl/~lf/tools/iocochecker/

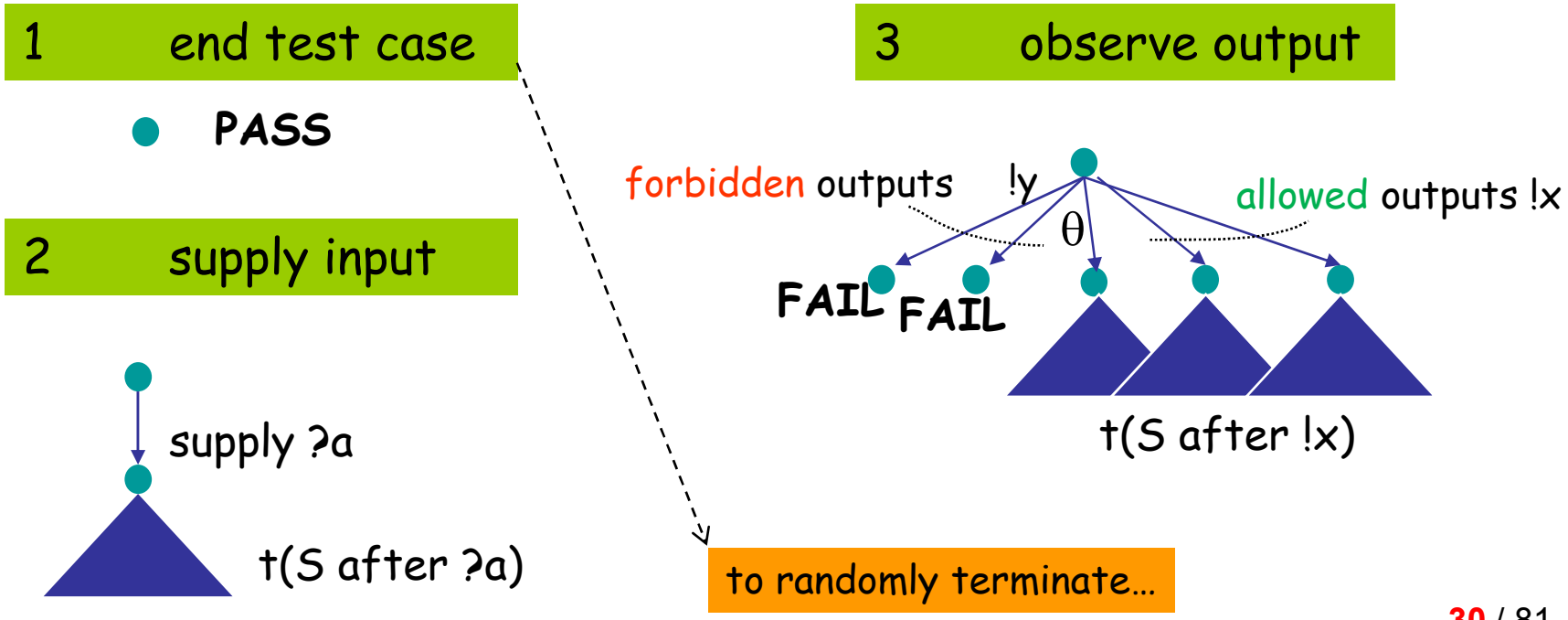
Test Generation Algorithm

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

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

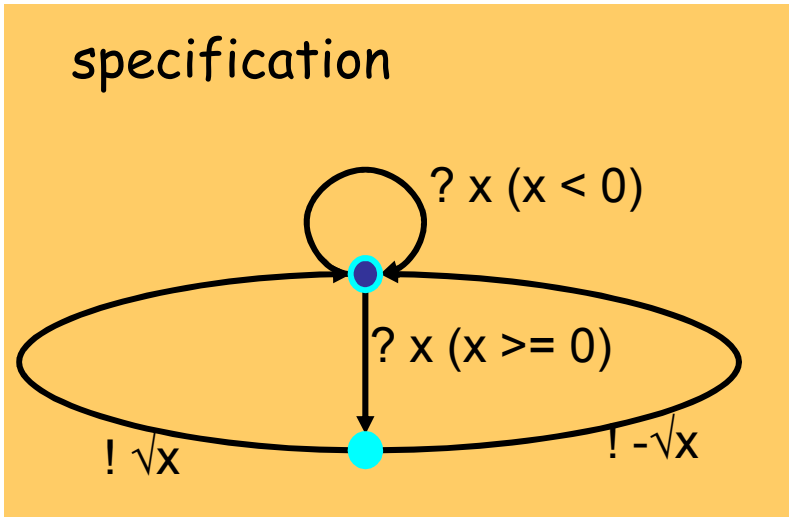
Algorithm:

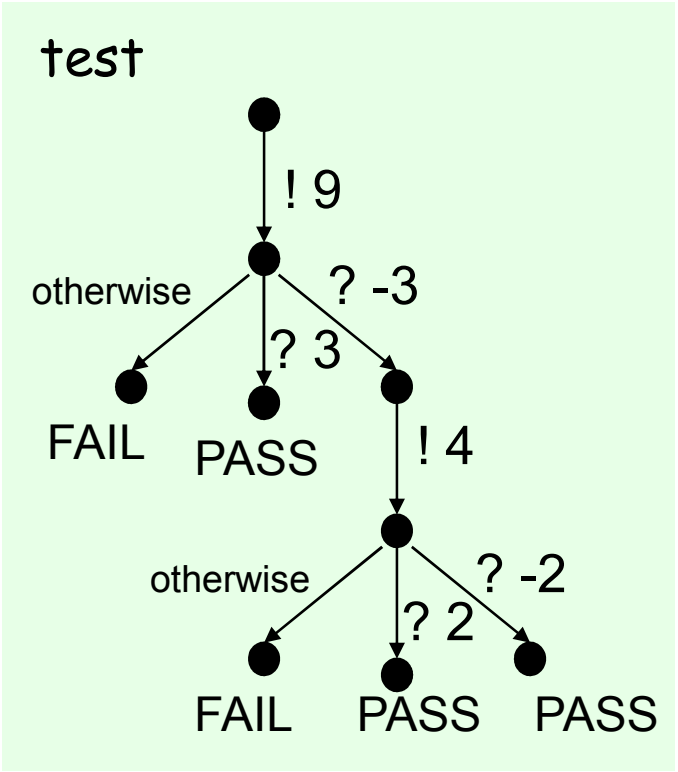
Apply the following steps recursively, non-deterministically



Test Generation Example

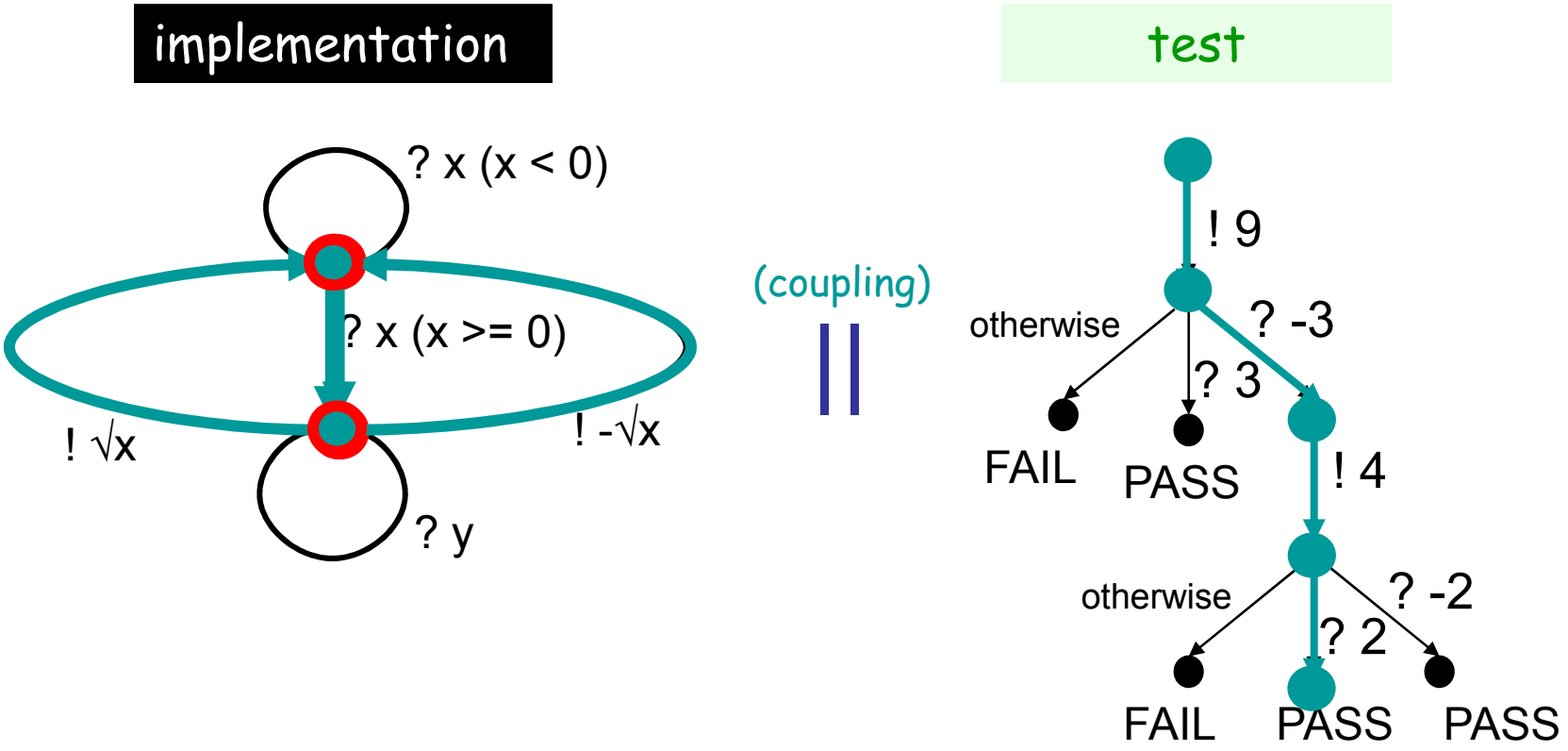
Equation solver for $y^2=x$





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

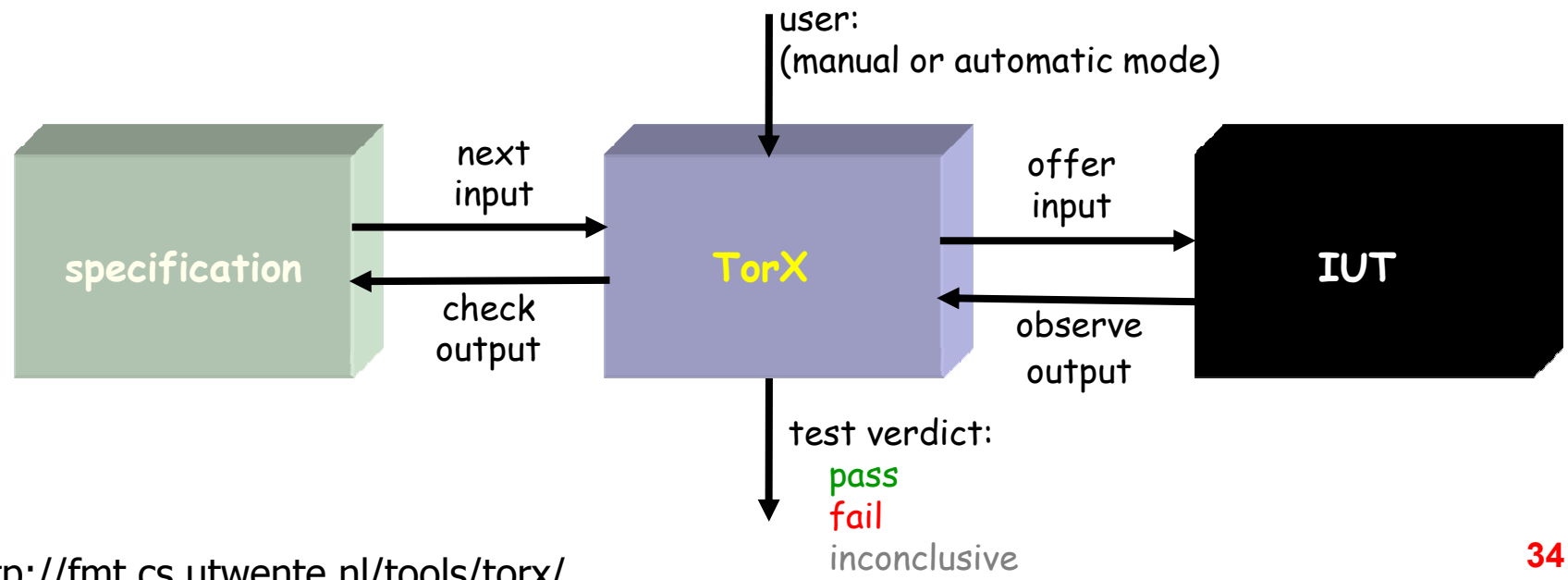
Exhaustiveness :

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

  i ~~ioco~~ s implies ∃t : i fails t

LTS-based Conformance Testing Tool: TorX

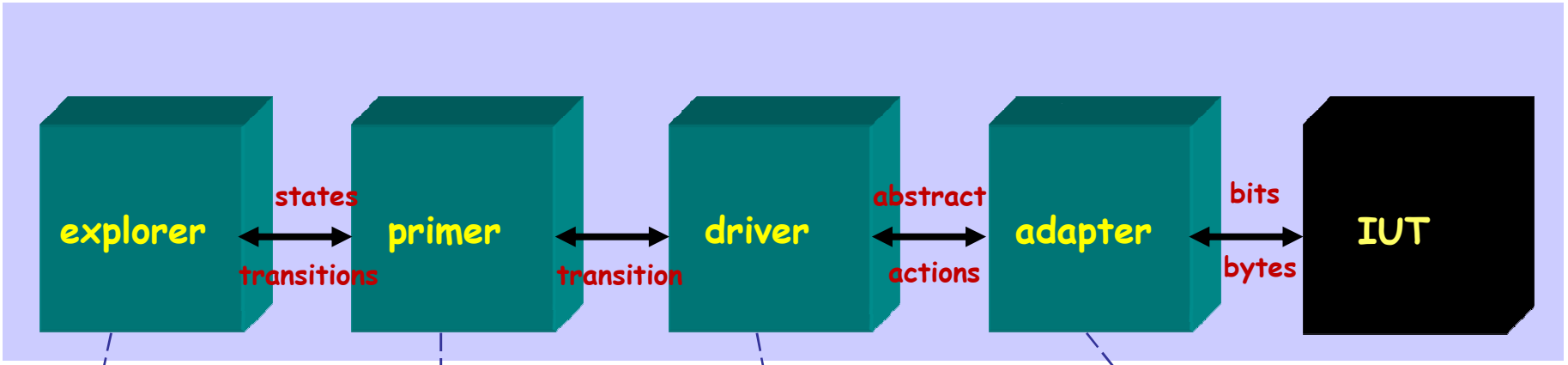
- On-the-fly test generation and test execution
- Implementation relation: **ioco**
- Specification languages: LOTOS and Promela



http://fmt.cs.utwente.nl/tools/torx/

TorX Tool Architecture

on-the-fly testing



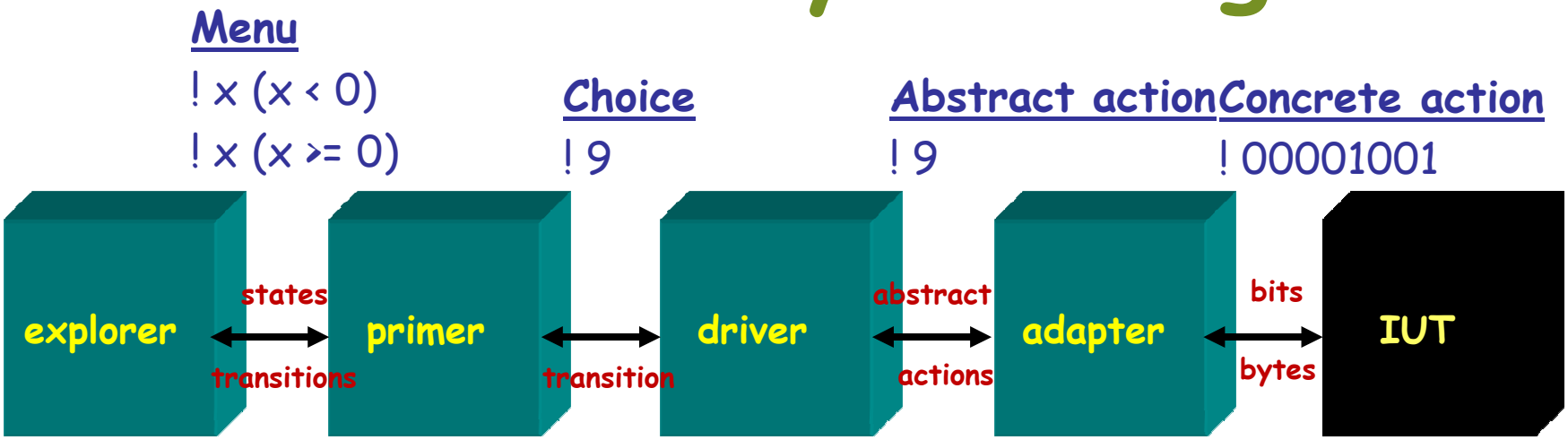
responsible for sending inputs to and receiving outputs from the IUT on request of the driver

to control the progress of the testing process

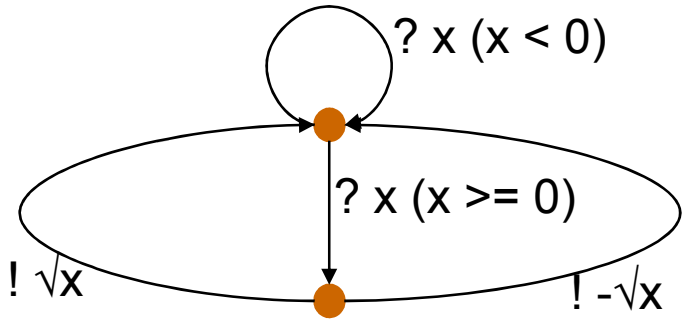
to implement the test derivation algorithm (to generate inputs for the implementation and to check outputs from the implementation)

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

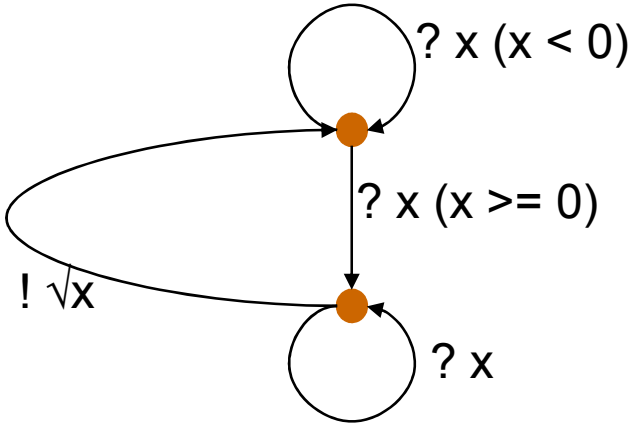
On-The-Fly Testing



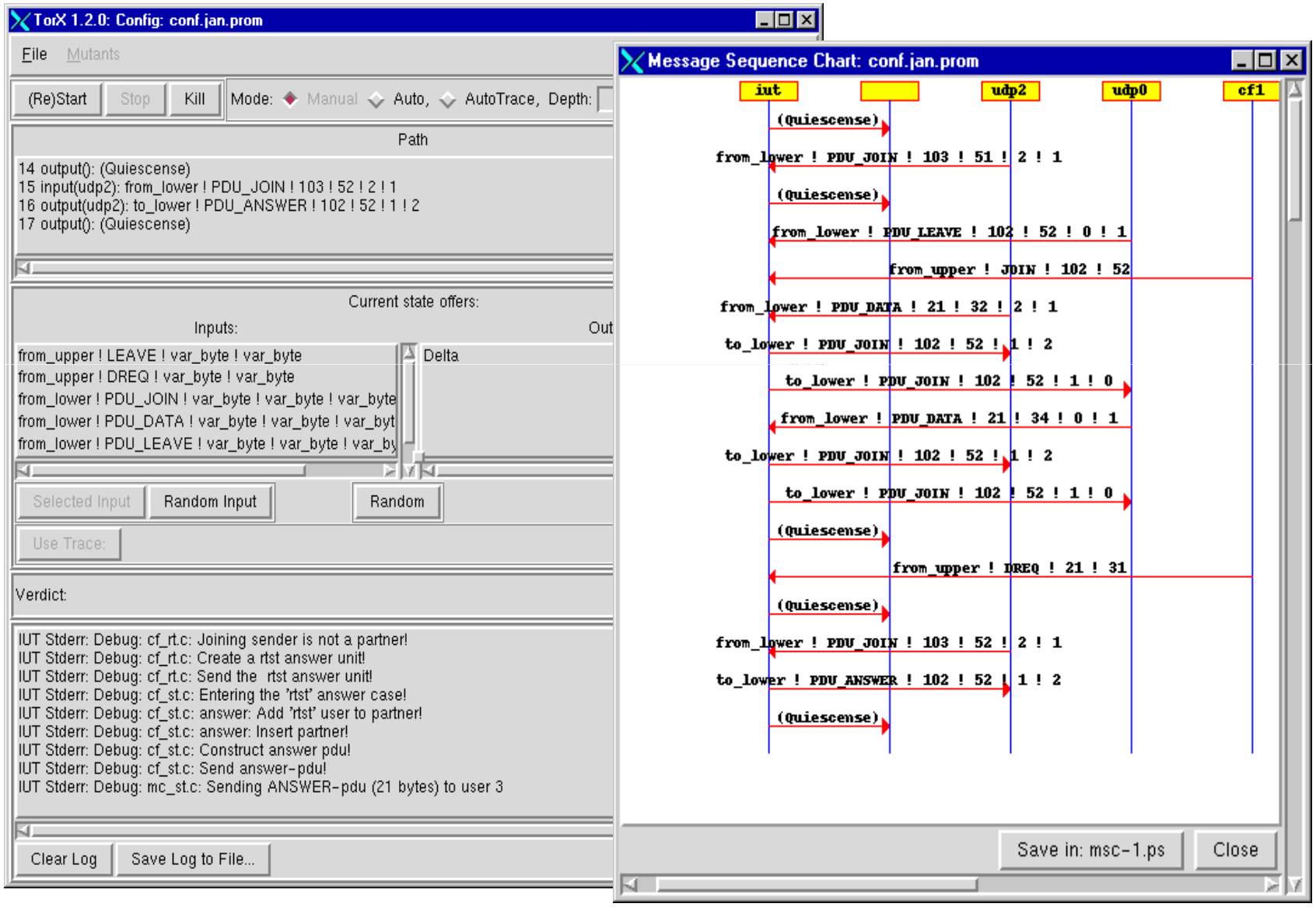
specification



implementation



TorX Screenshot



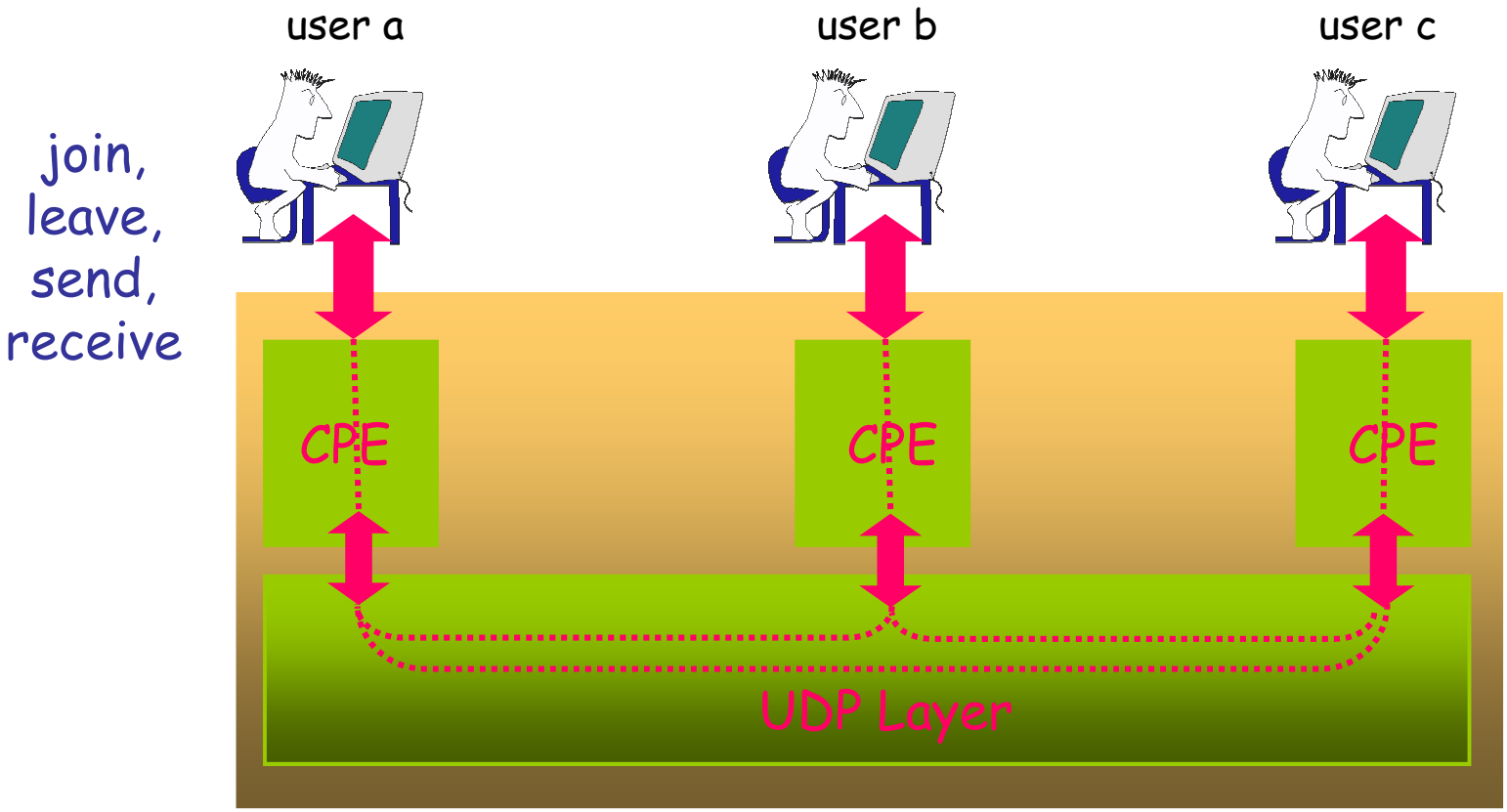
Case Study of LTS-based Testing
Conference Protocol Experiment

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

a single error is injected deliberately

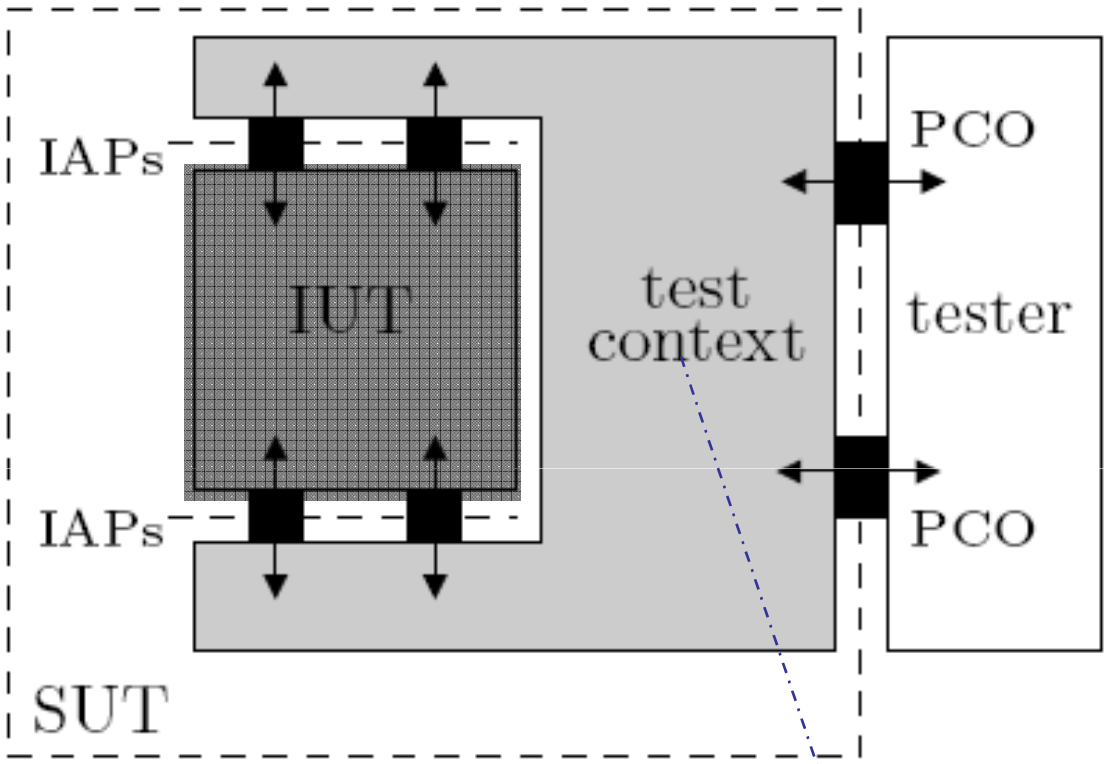
errors:
- no outputs
- no internal checks
- no internal updates

The Conference Protocol



CEP: Conference Protocol Entity
UDP: User Datagram Protocol

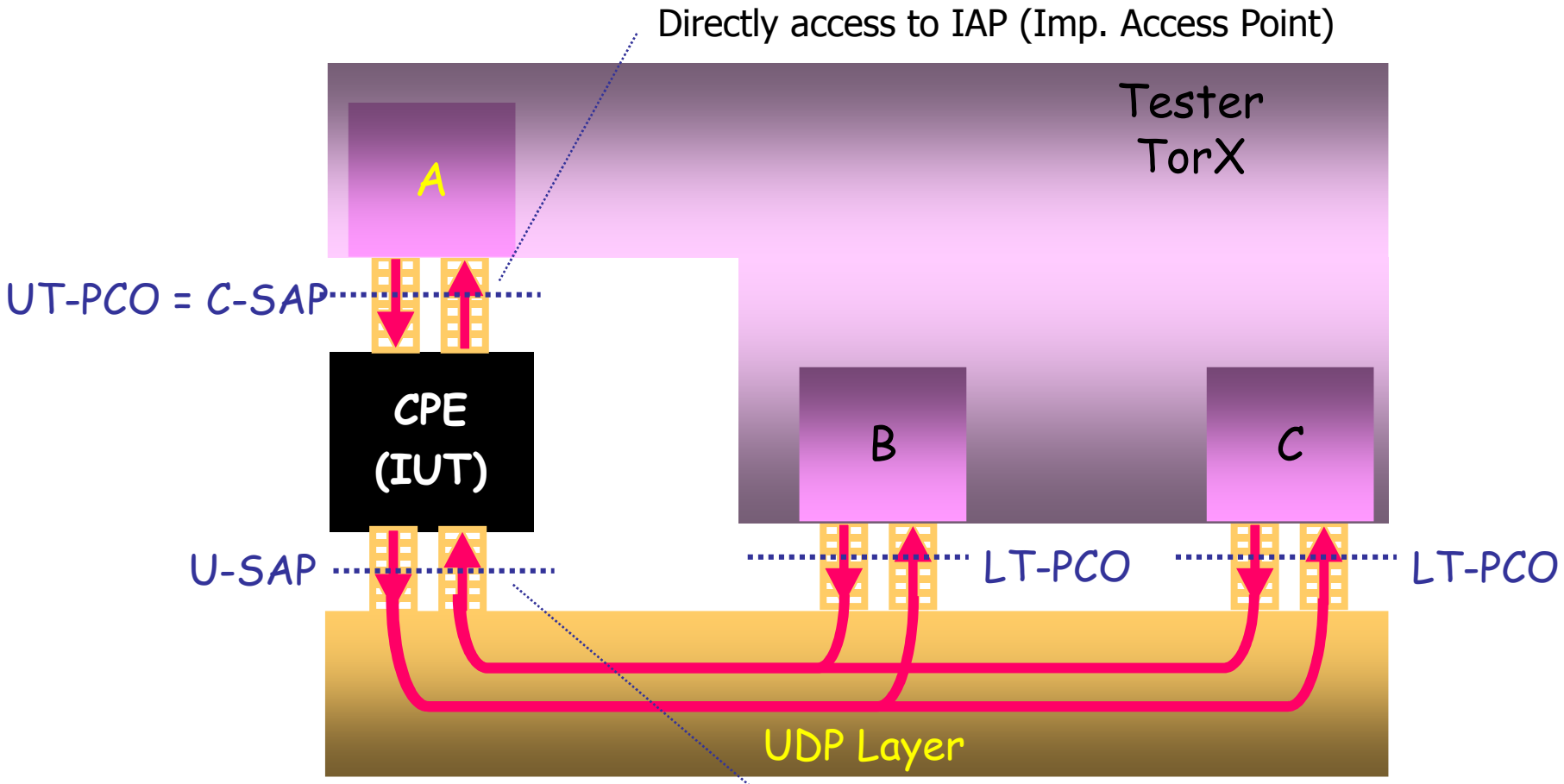
Abstract Test Architecture



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	steps max	*Promela* *verdict*	steps min	steps 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

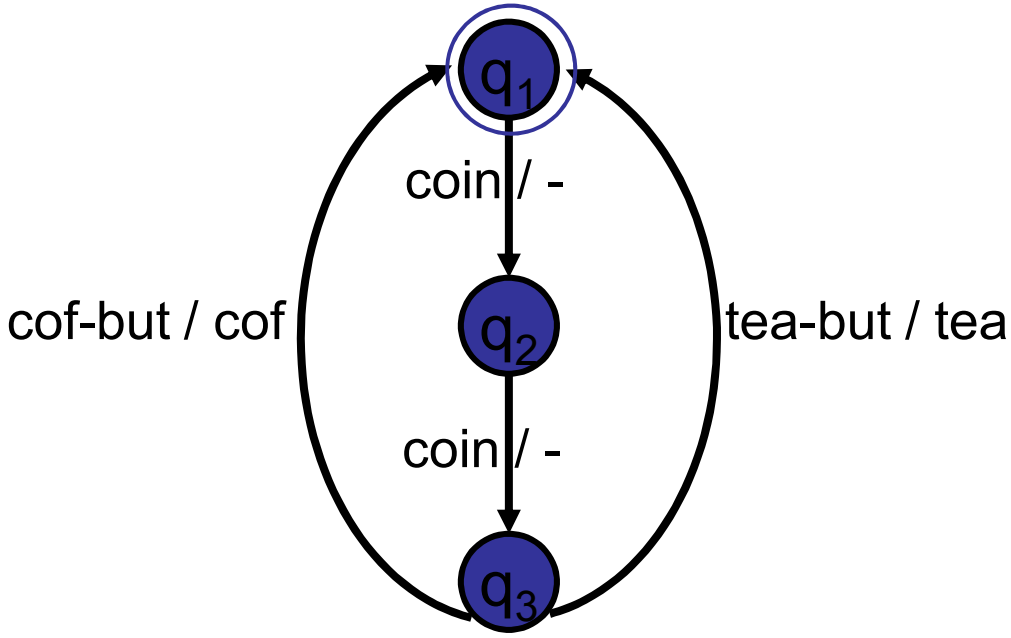
Reported experiments:

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

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

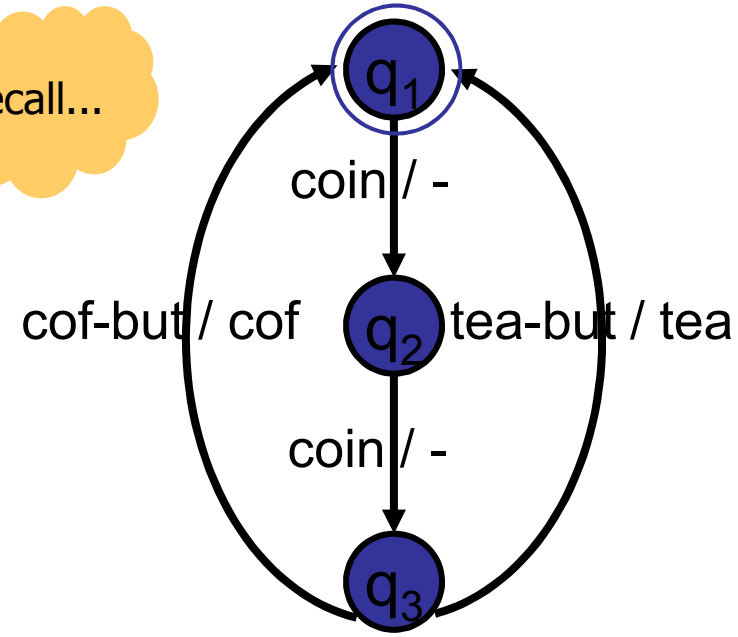
- TorX statistics (with LOTOS and Promela)
  - all errors found after 2 - 498 test events
  - maximum length of tests : > 500,000 test events
  - 2 mutants react to PDU's from non-existent partners:
    - no explicit reaction is specified for such PDU's,
      so ioco-correct, and TorX does not test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

Finite State Machine (FSM)-Based Testing



FSM example (Mealy machine)

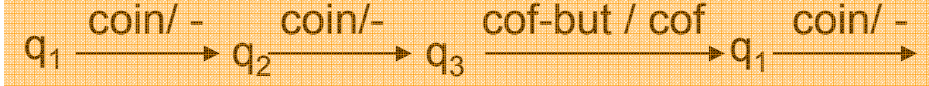
Recall...

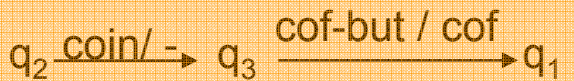


condition		effect	
current state	input	output	next state
q_1	coin	-	q_2
q_2	coin	-	q_3
q_3	cof-but	cof	q_1
q_3	tea-but	tea	q_1

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:





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 (transition) function
$\lambda : S \times I \rightarrow O$	output function

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

$\lambda : S \times I^* \rightarrow O^*$

Recall...

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 M_0 and M_1 are equivalent iff the two initial states of them are equivalent

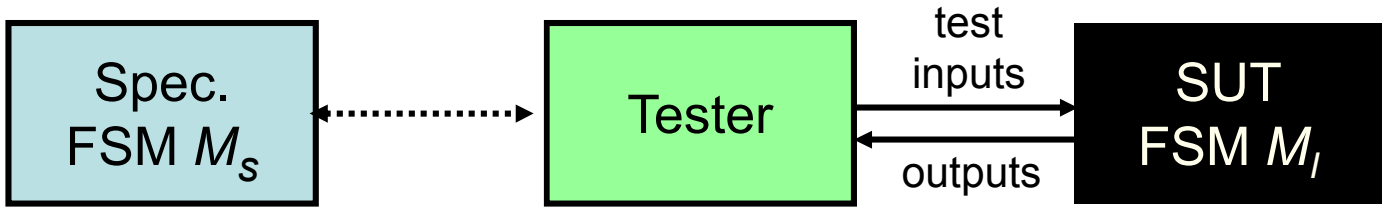
- A minimized (or reduced) M is one that has no equivalent states
  - for all states s, t : (s equivalent t) ==> (s = t)

Recall...

Fundamental Results

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

FSM Conformance Testing



Given: a specification FSM M_S , and
a (black-box) implementation FSM M_I

and we assume:

- Deterministic specifications
- M_I is an (unknown) deterministic FSM (the "testing hypothesis")

much stronger assumptions than LTS-based testing

Task: To determine whether M_I conforms to M_S,

i.e., whether M_I behaves in accordance with M_S , or

whether outputs of M_I are allowed by M_S , or

whether the reduced M_I is equivalent to M_S

Restrictions on FSM

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

- **deterministic**

  $\delta : S \times I \rightarrow S$ and $\lambda : S \times I \rightarrow O$ are *functions* (not "relations")

- **completely specified**

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

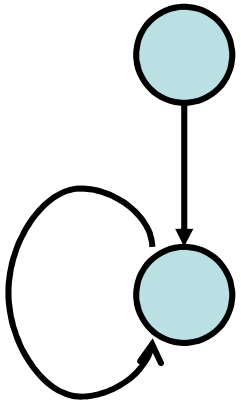
  (empty output is allowed; sometimes implicit completeness)

- **strongly connected**

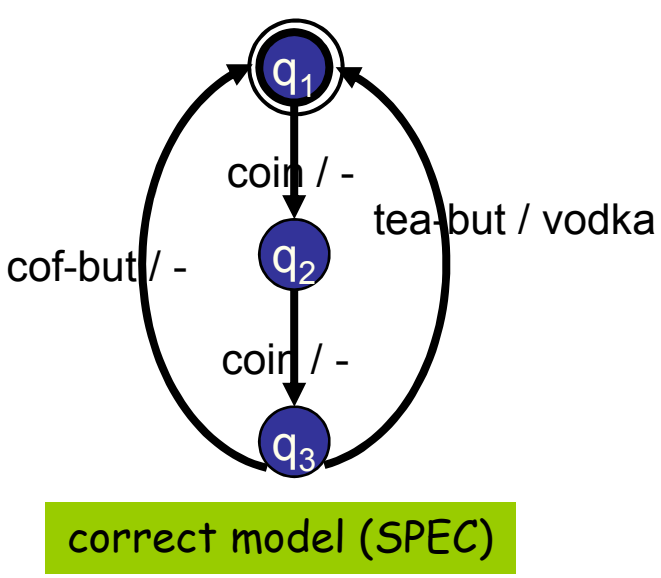
  from any state any other state can be reached

- **reduced**

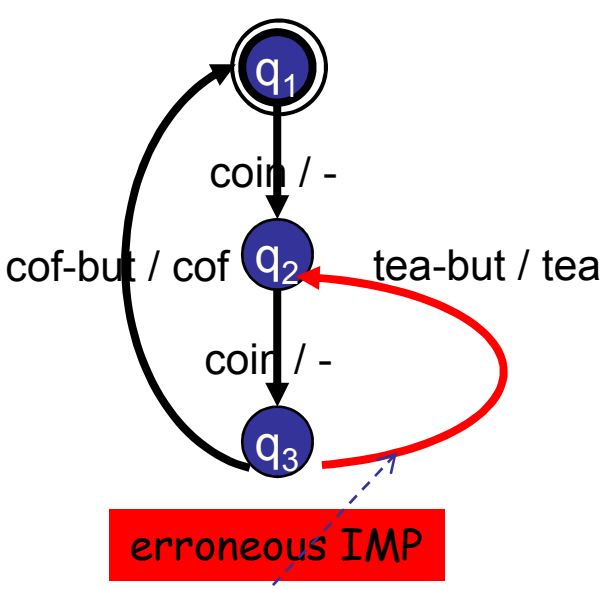
  there are no equivalent states



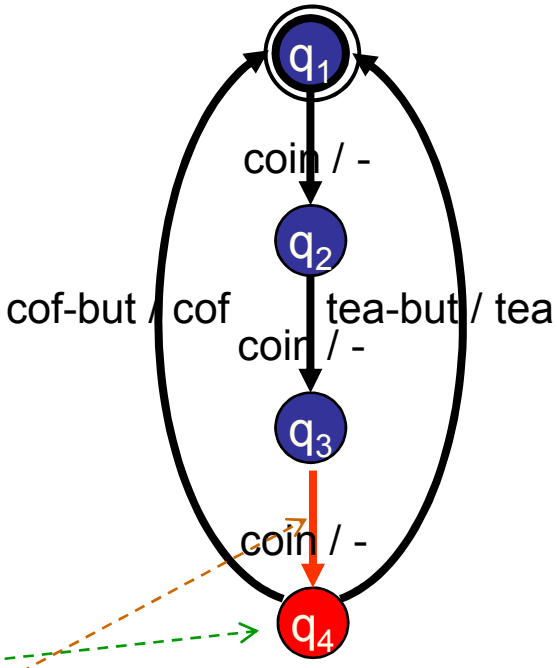
Type of Faults



correct model (SPEC)



erroneous IMP

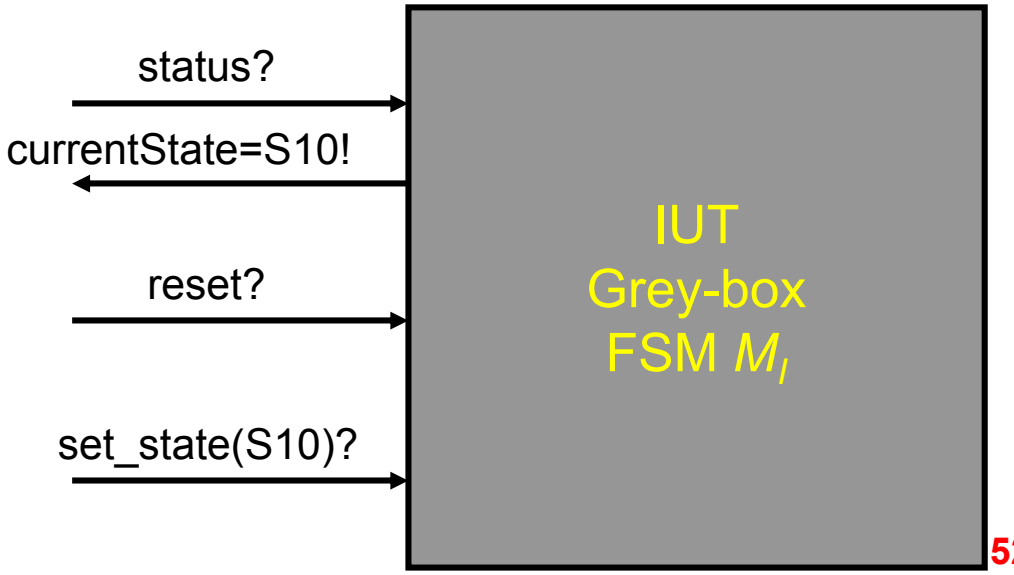


erroneous IMP

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

Desired Utilities for Tester

- Nice, but rarely realistic assumptions
  - "status" message: Assume that tester can enquire the implementation (IUT) for its current state (reliably!!!) without changing the IUT state
  - reset: to reliably bring IUT to the initial state
  - set_state(): to 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 program testing technique)

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

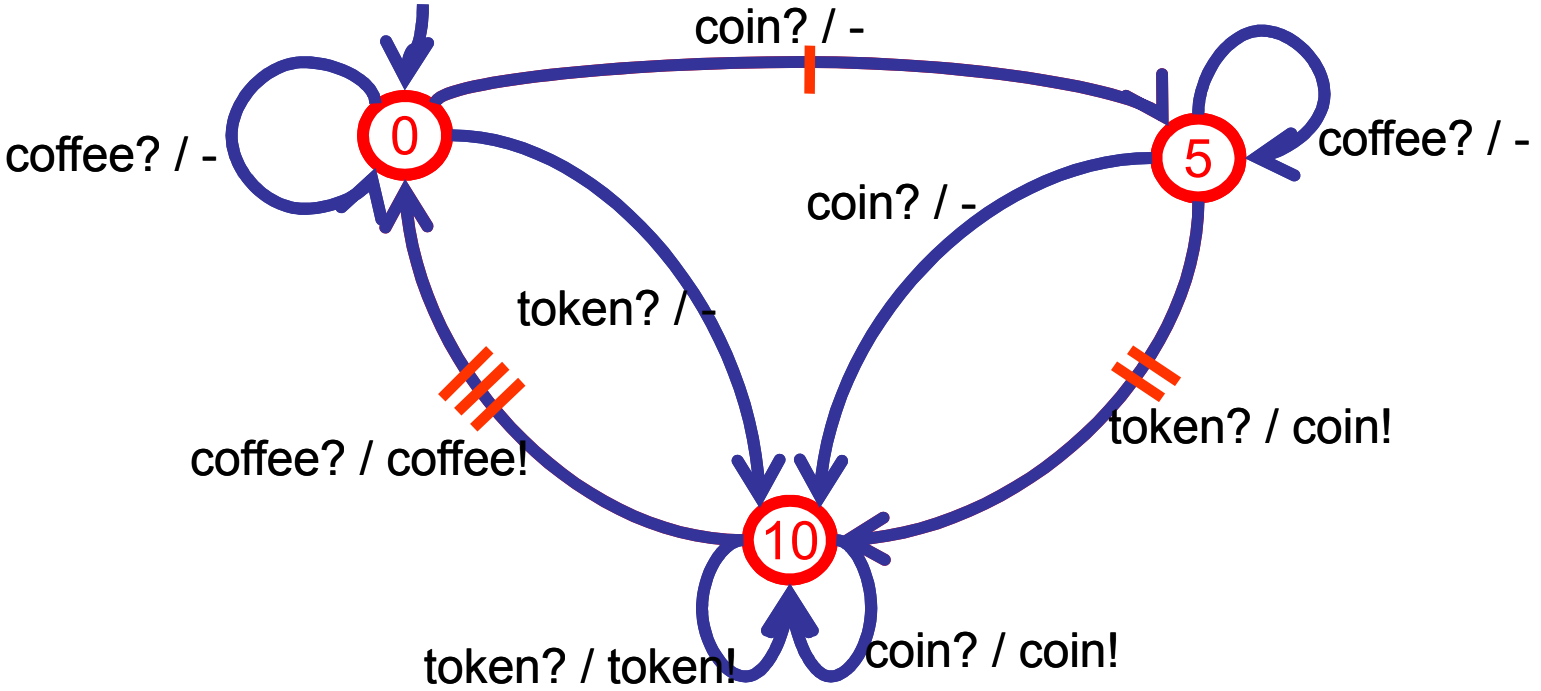
To find a path or a set of paths to cover all the states in the specification FSM

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

A Coffee Machine FSM (Mealy)

State Coverage

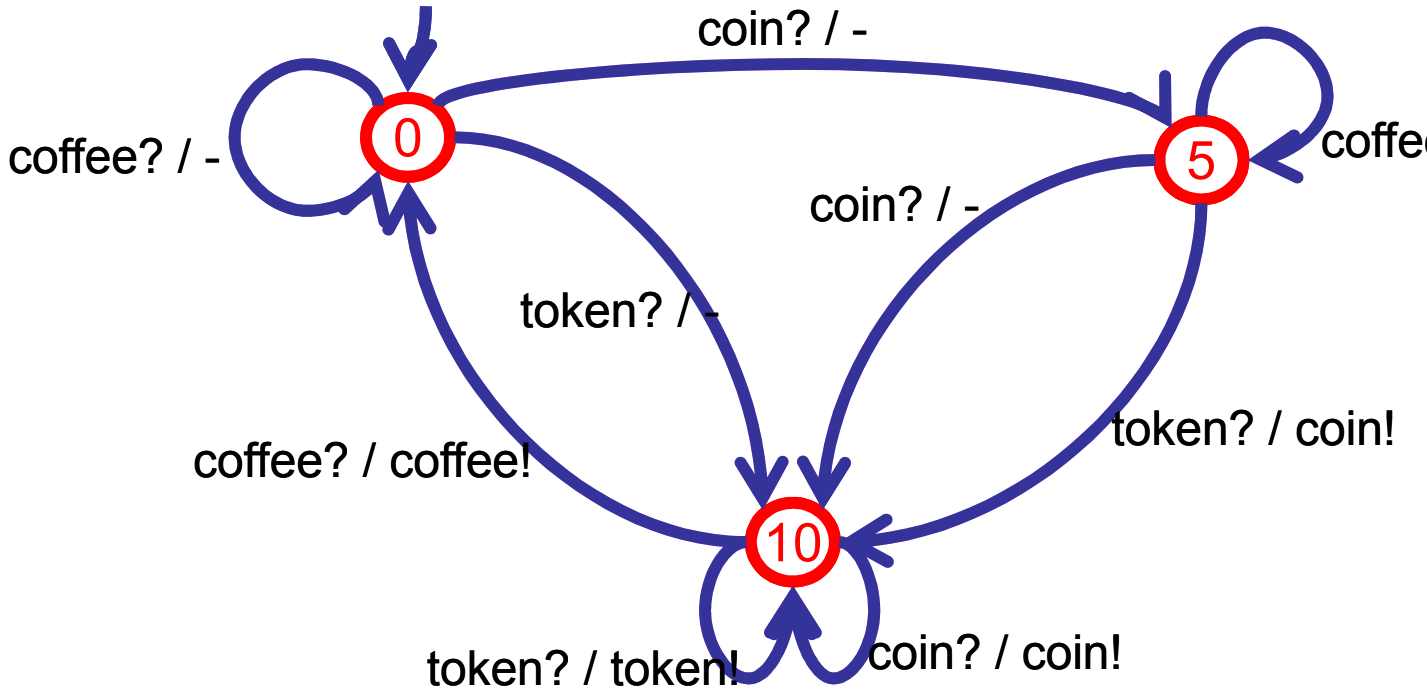
- Make *State Tour* that covers every state (in spec.)



Test sequence : coin? token? coffee?

Transition Coverage

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

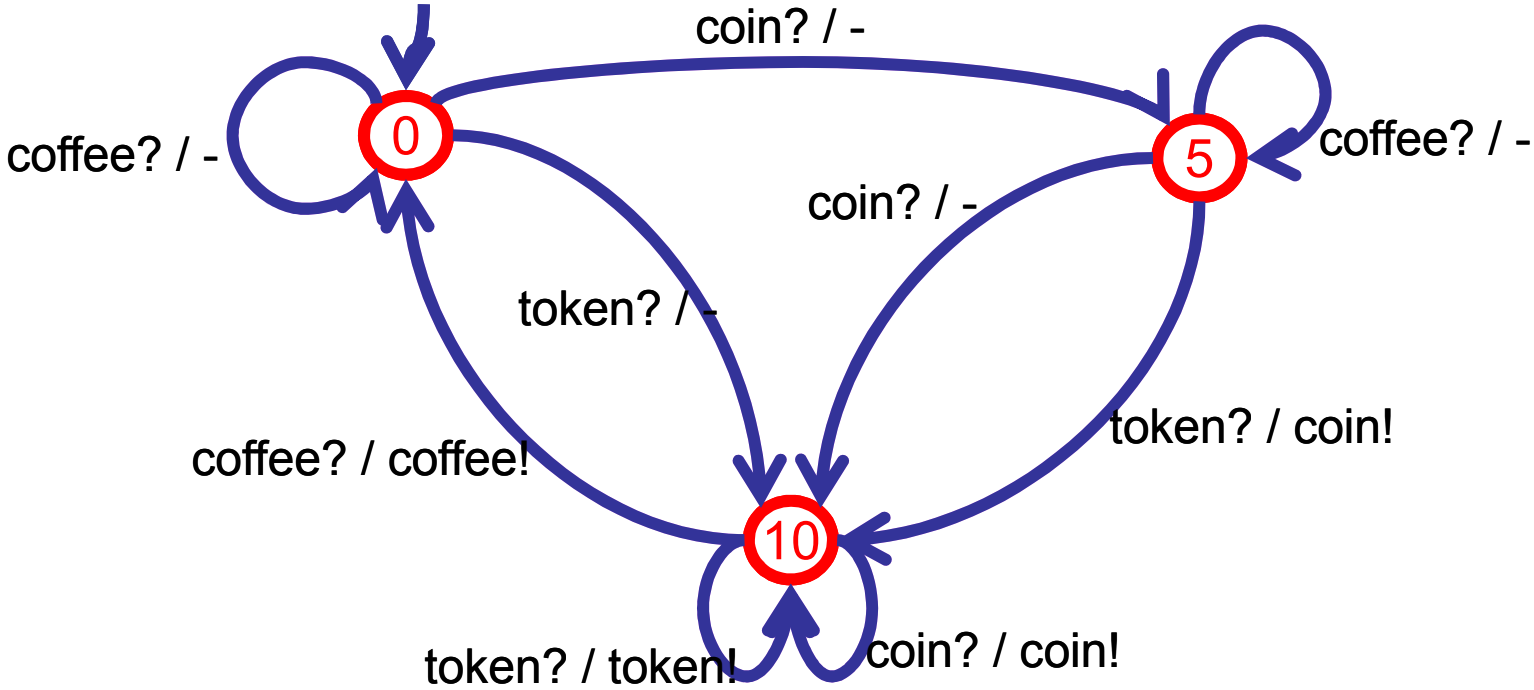


Test input sequence :

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)



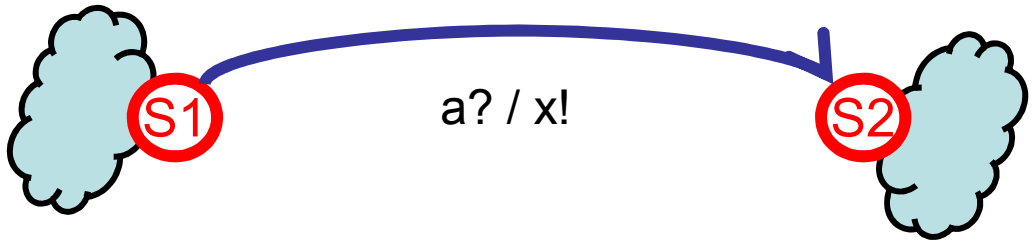
Test input sequence :

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

+ check expected output and target state using the "status" message

FSM Transition Testing

- Make a test case for each transition in SPEC separately:

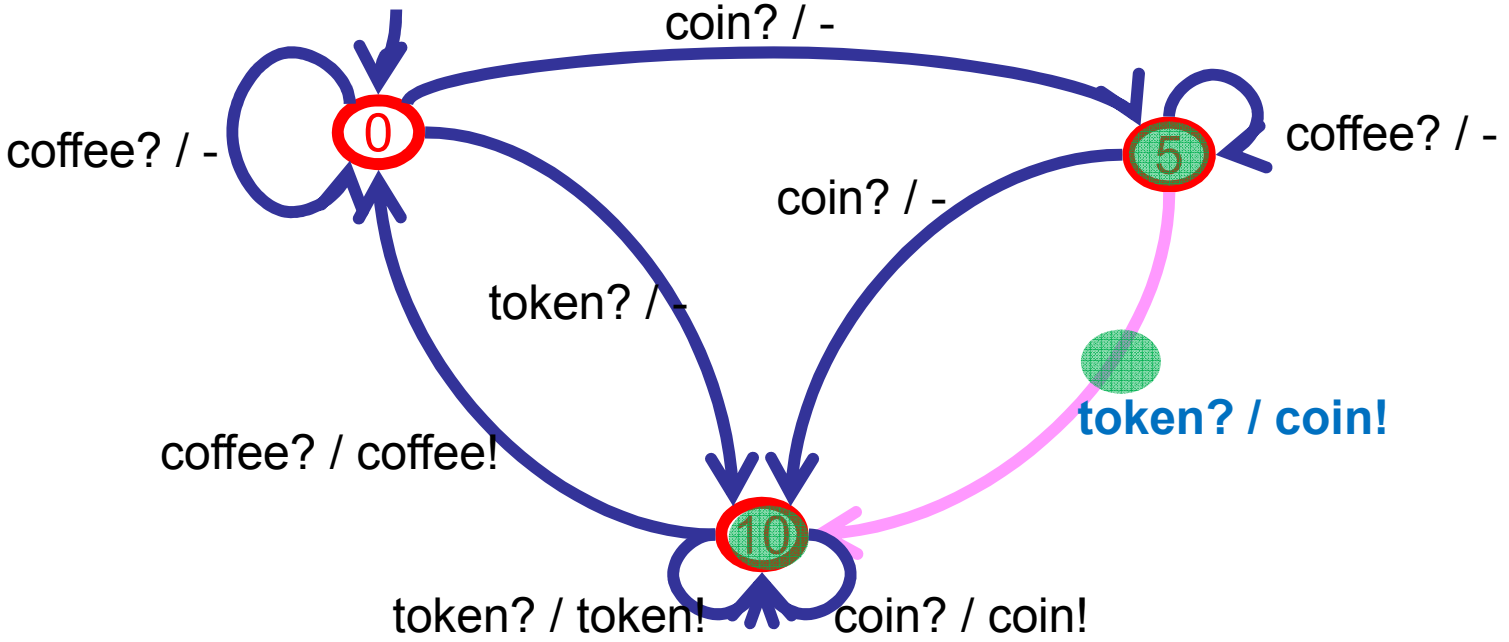


- Test purpose: "Test whether the system, when in state S1, produces output x! on input a? and goes to state S2"

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

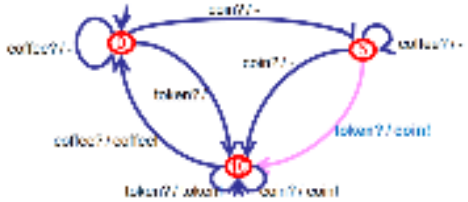
Transition Testing – issue #1

- To test **token? / coin!** :
  - go to source state: set_state(5)
    give input token? check output coin!
    verify destination state: status? // currentState == 10 ??



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

Transition Testing – issue #1

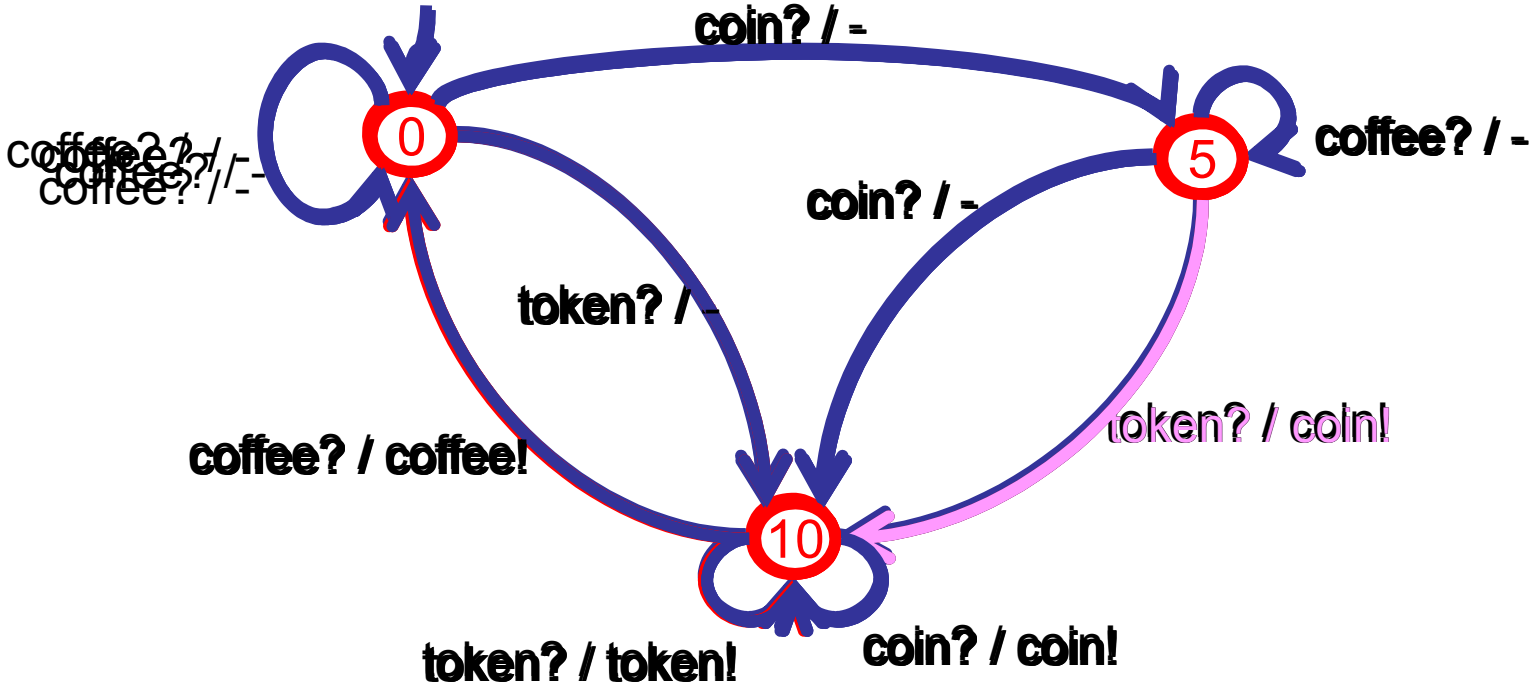


- "go to state S5" relies on the "set_state()" method
- What if "set_state()" method **not** available?
  - if the "reset" method is available, then use it instead
    - go from S0 to S5 (always possible because of determinism and completeness)
  - otherwise, use a **synchronizing sequence** to bring 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 – issue #1

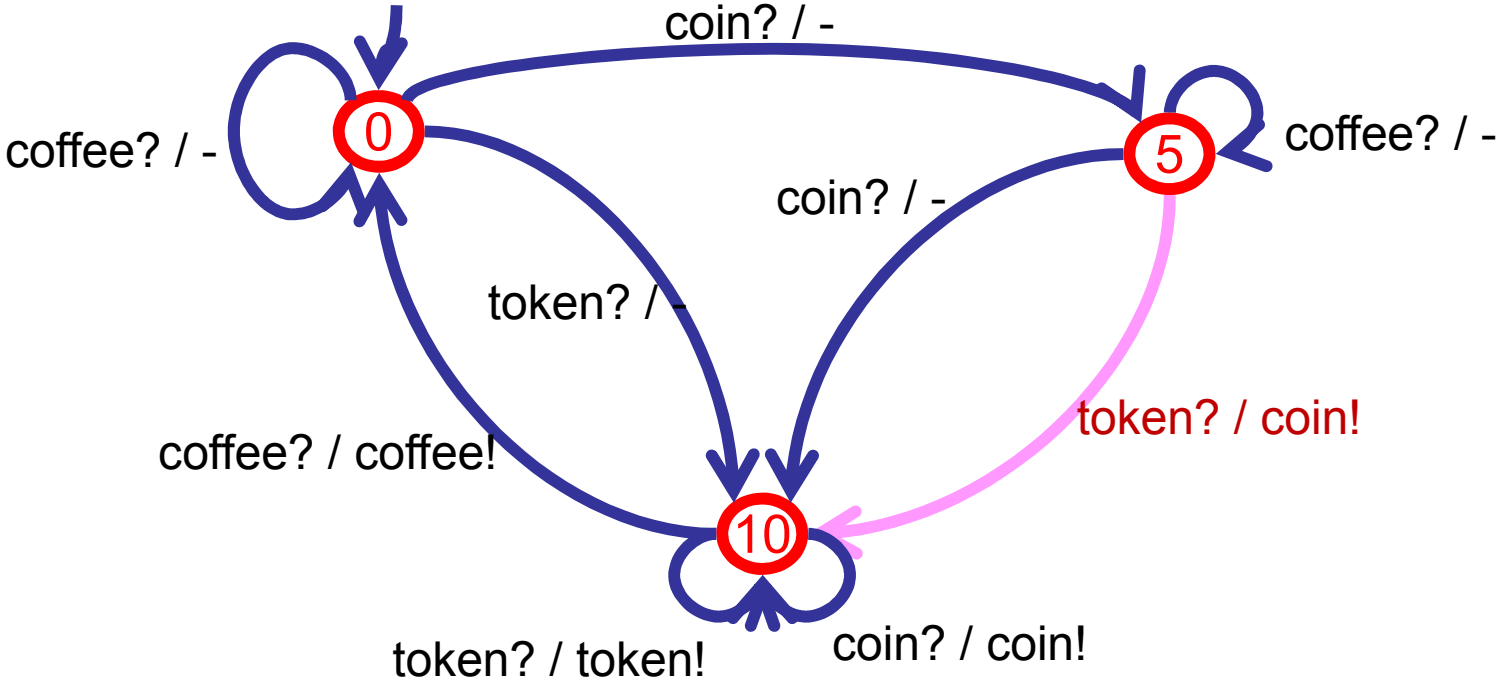
synchronizing sequence of state S0: token? coffee?



To test token? / coin! : go to state S5 by : token? coffee? coin?
(synchronizing sequence of S0)

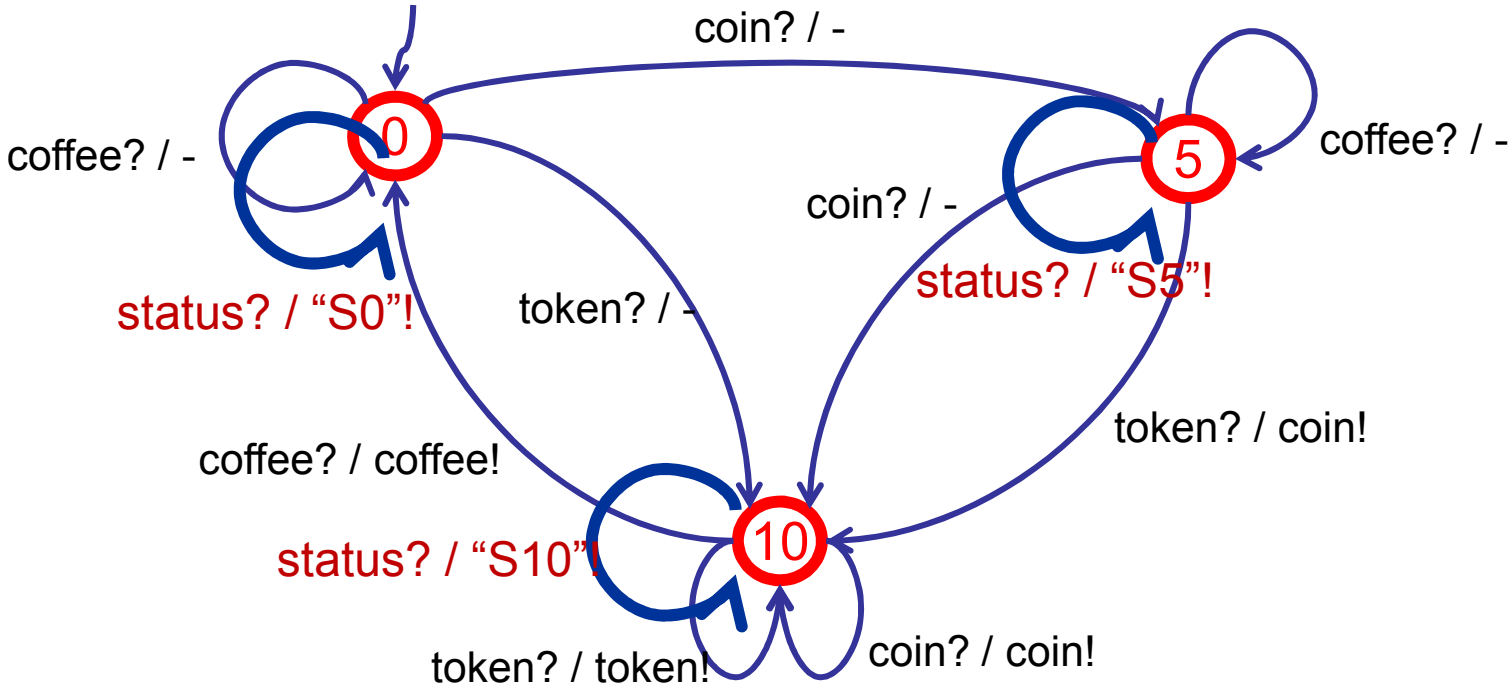
Transition Testing – issue #2

- To test token? / coin! :
  1. go to state S5 by : "token? coffee? coin?"
  2. give input token?
  3. check output coin!
  4. verify that machine is in state S10 by: "status? currentState==10"



Transition Testing – issue #2

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



Transition Testing – issue #2

- What if **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 the state where we started (**state identification**), or
    - verify that we were indeed in a particular start state (**state verification**)
  - Different kinds of sequences (dating back to 1960s)
    - UIO sequences (Unique Input Output sequence)
    - Distinguishing Sequence (DS)
    - W-set (characterizing set of sequences)
    - UIOv
    - SUIO
    - MUIO
    - Overlapping UIO

Transition Testing – issue #2

State check :

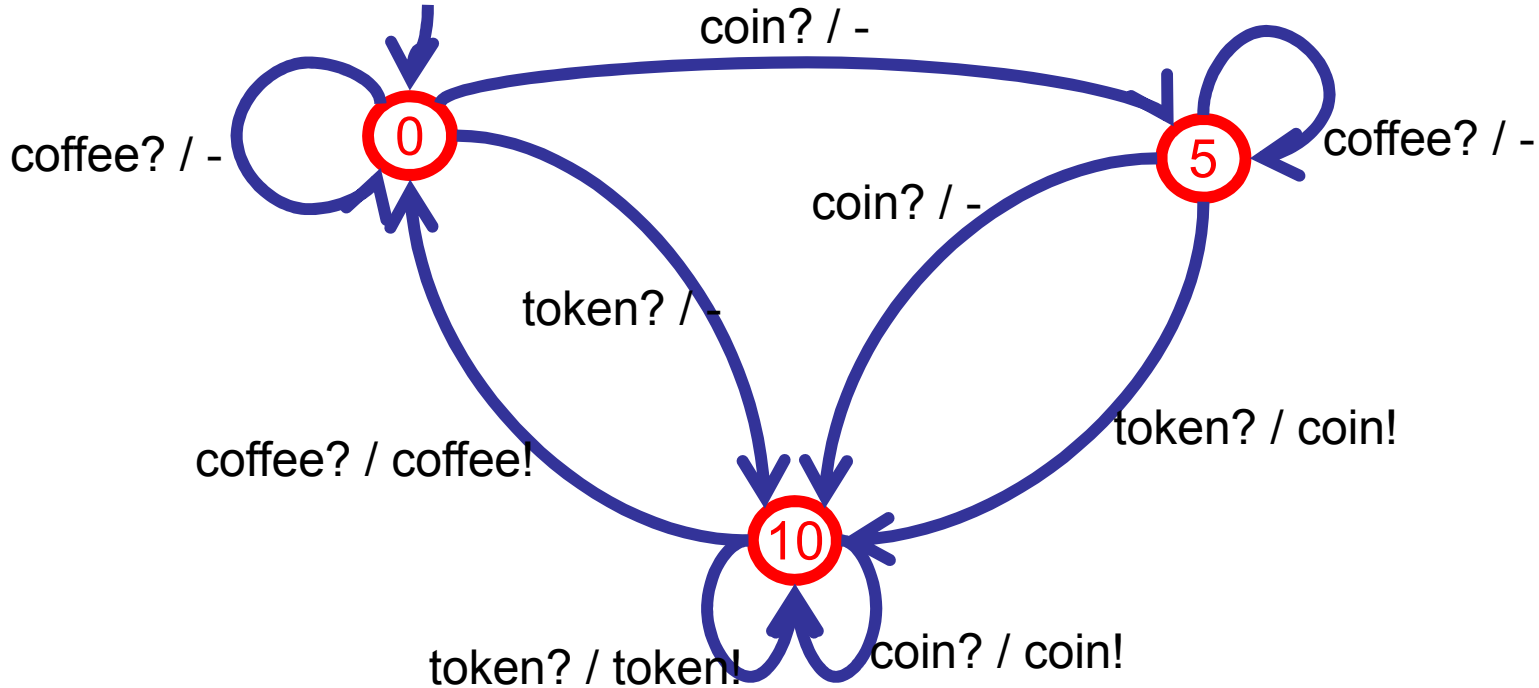
UIO: each state has **its own** input sequence that produces different outputs when applied in other states.

- **UIO sequences** (state verification)
  - sequence x_s that distinguishes state s from <u>all other</u> 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
- **Distinguishing Sequence** (state 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** (state 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

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

Transition Testing – issue #2: UIO

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

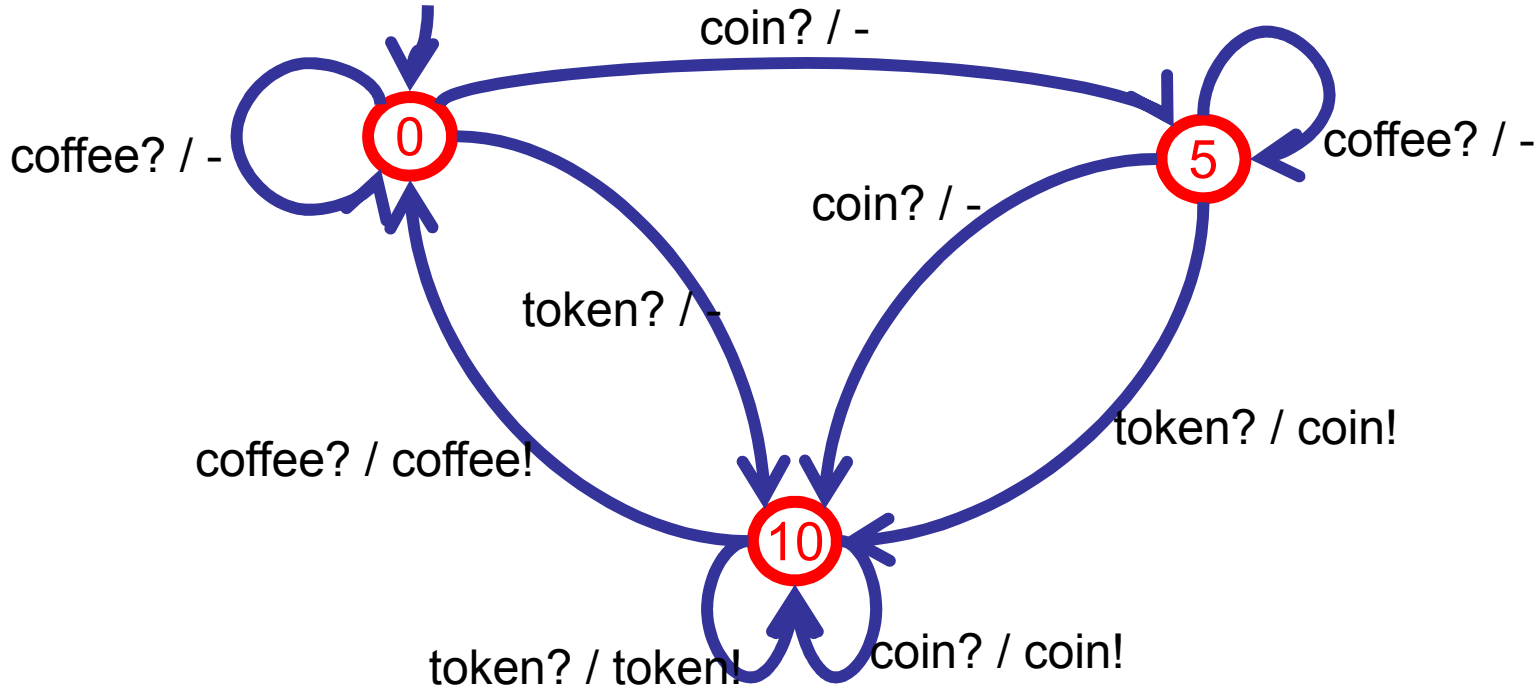


UIO sequences

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

Transition Testing – issue #2: DS

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



DS sequence: token?

output state 0 : -
output state 5 : coin!
output state 10 : token!

Transition Testing – issue #2: done

synchronizing sequence

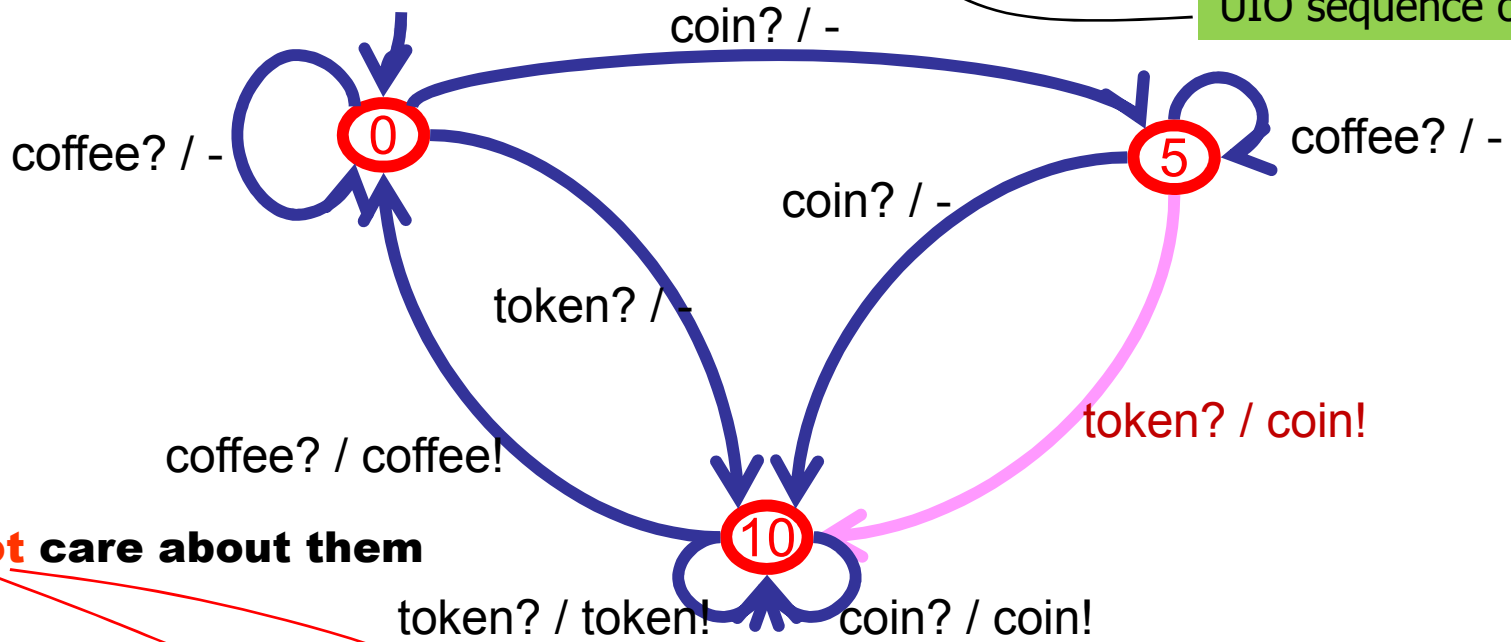
•To test token? / coin! :

1) go to state S5 : token? coffee? coin?

2) give input token? check output coin!

3) apply UIO of state S10 : coffee? / coffee!

UIO sequence of S10

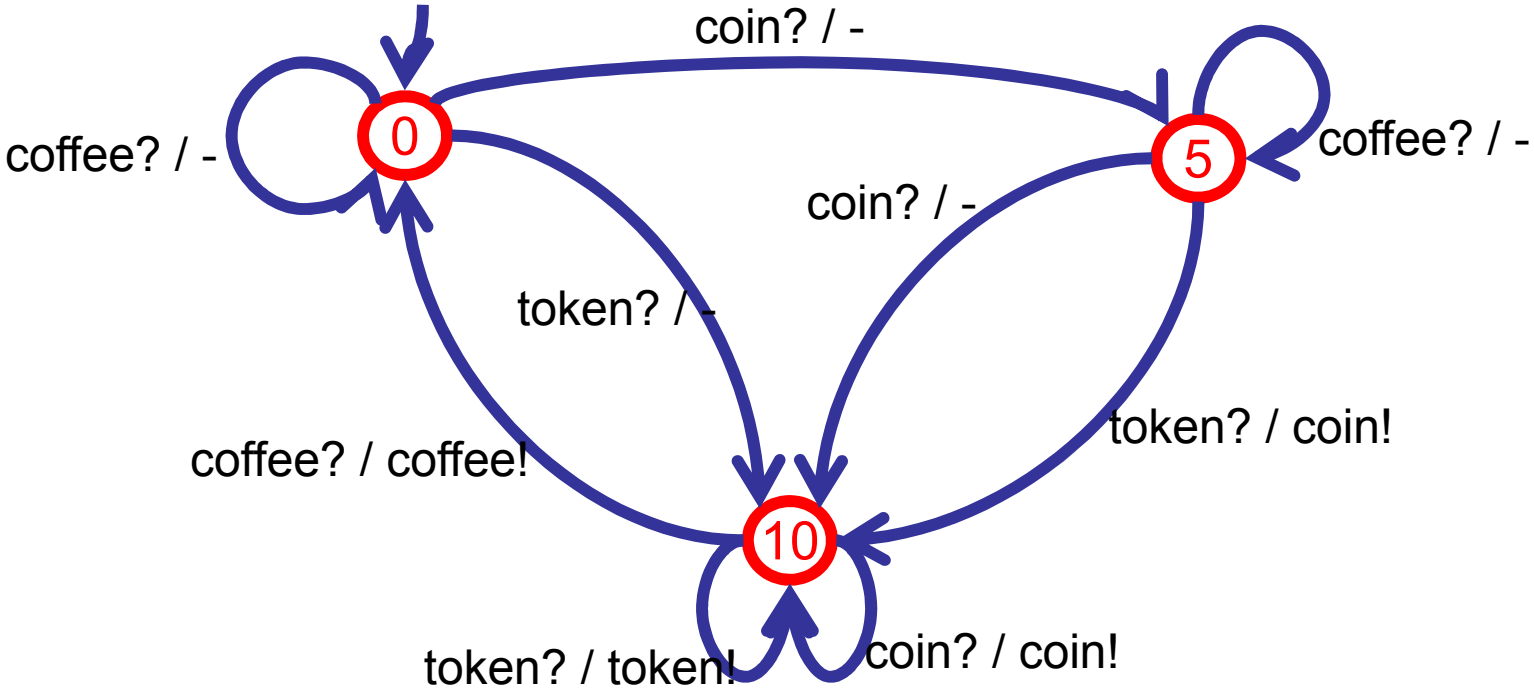


we do not care about them

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

(goto state S5) offer input expected output (check state S10)

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 various tools to support this

FSM Transition testing: further results

- Test transition "S1 --a?/x!--> S2":
  1. Go to state S1 // synchronizing sequence
  2. Apply input a?
  3. Check output x!
  4. Verify state S2 // UIO sequence of 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.

Tools for Model-Based Testing

Academic MBT Tools

Tool name	Tool provider	Modeling notation	Testing method	Short description
Lutess		Lustre		
Lurette		Lustre		an automated testing tool of reactive programs written in Lustre
GATeL		Lustre	CLP	a tool that automatically generates test sequences from SCADE/Lustre models, according to a user defined test objective
Autofocus		Autofocus	CLP	a graphical tool for developing and modeling distributed systems with integrated testing facilities
Conformance Kit		EFSM	FSM	
Phact		EFSM	FSM	
TVEDA		SDL, Estelle	FSM	
AsmL		AsmL	FSM?	To generate tests directly from an AsmL model

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	a tool for the generation of conformance test suites for protocols
TorX	Twente University	LTS (LOTOS, Promela, FSP)	LTS	a prototype testing tool for conformance testing of reactive software
STG	Irisa, France	NTIF	LTS	
AGEDIS		UML/AML	LTS	
Uppaal Tron	Aalborg University	TA	TLTS	a tool for on-line conformance of real-time systems based on Timed Automata models
Uppaal Cover	Uppsala University	TA	TLTS	a tool for off-line conformance of real-time systems based on Timed Automata models

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

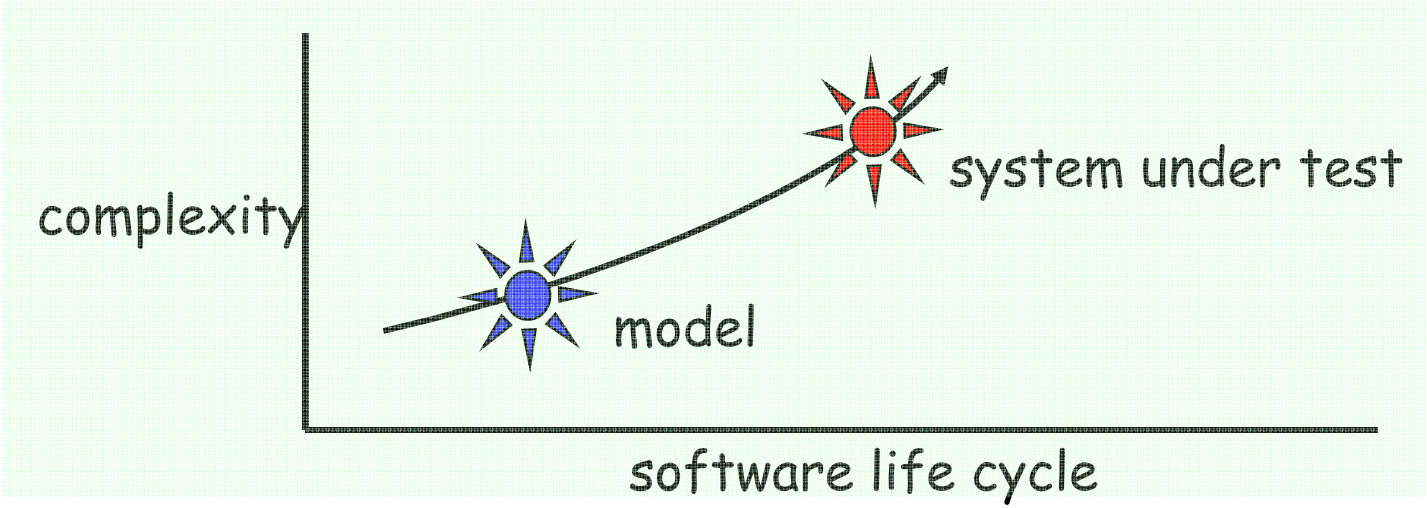
LTS Testing vs. FSM Testing

- 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 (to enable driving the test executions):
    - deterministic
    - completeness
  - FSM has always alternations between inputs and outputs
  - Difficult to specify interleaving in FSM
  - FSM is not compositional

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
  - Learning curve
- Skill sets
  - Need testers who can design
- Expectations
  - Models can be a significant upfront investment
  - But will never catch all bugs
- Metrics
  - Bad metrics: bug counts, number of test cases
  - Better metrics: spec coverage, code coverage

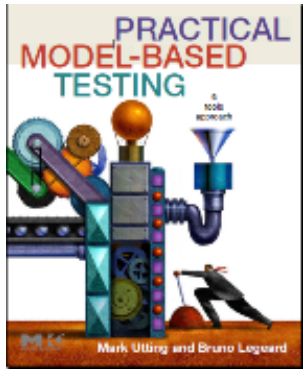
Main Readings

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

- Jan Tretmans. **Model Based Testing with Labelled Transition Systems**. In: Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers. Lecture Notes in Computer Science 4949 Springer 2008, pp.1-38. (http://www.springerlink.com/content/y390356226x154j0/)

Further Readings

- 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



- Papers:
  - David Lee, Mihalis Yannakakis. **Principles and methods of testing finite state machines - A survey**. In: Proceedings of the IEEE, 84(8): 1090-1126, 1996.

Other resources

- Model-based testing website:
  www.model-based-testing.org

- General conferences
  - **FORTE** (International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols)
  - **TACAS**, **FM/FME**, **ISSTA**, ...

- Specialized conferences/workshops:
  - **TestCom'00- TestCom'08 (**IFIP Int. Conference on Testing of Communicating Systems), and previously as
    - IWPTS'88 - IWPTS'96 (International Workshop for Protocol Test Systems)
    - IWTCS'97 - IWTCS'99 (International Workshop on Testing of Communicating Systems)
  - **FATES (**Int. Workshop on Formal Approaches to Testing of Software**)**
  - **MBT (**Int. Workshop on Model-Based Testing**)**
  - **A-MOST** (Workshop on Advances in Model Based Testing)
  - ...