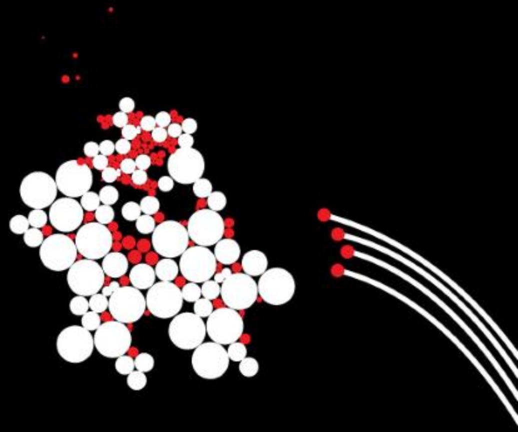


Model-based testing

NATO Summer School
Marktoberdorf, August, 2012

Ed Brinksma
University of Twente

CONTENTS

CONTENTS

PRACTICAL PROBLEMS OF TESTING

Testing is:

- important
- much practiced
- 30% - 50% of project effort
- expensive
- time critical
- not constructive
  (but sadistic?)

But also:

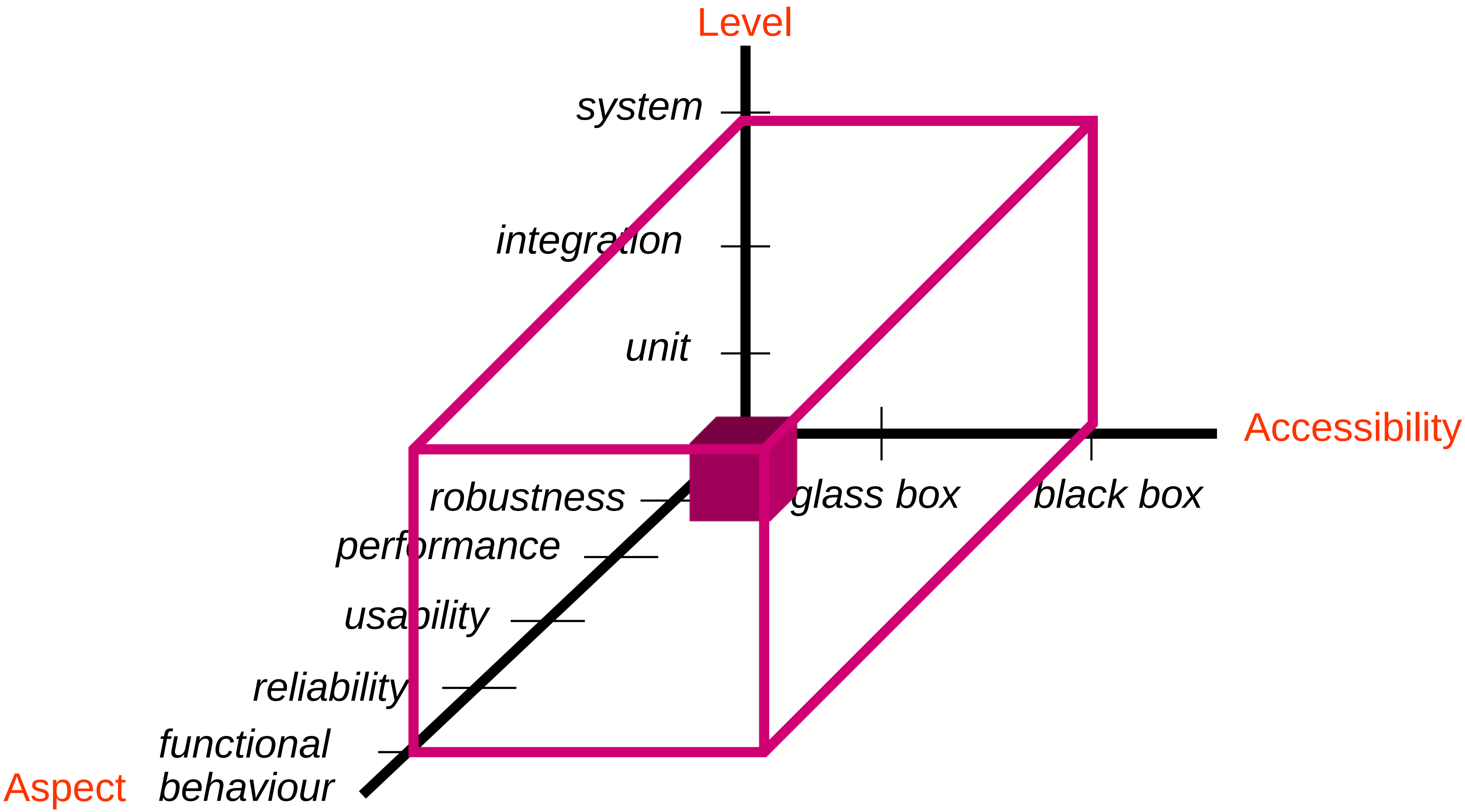
- ad-hoc, manual, error-prone
- limited theory / research
- little attention in curricula
- not *cool* :
  "if you're a bad programmer
  you might be a tester"

Attitude is changing:

- more awareness
- more professional

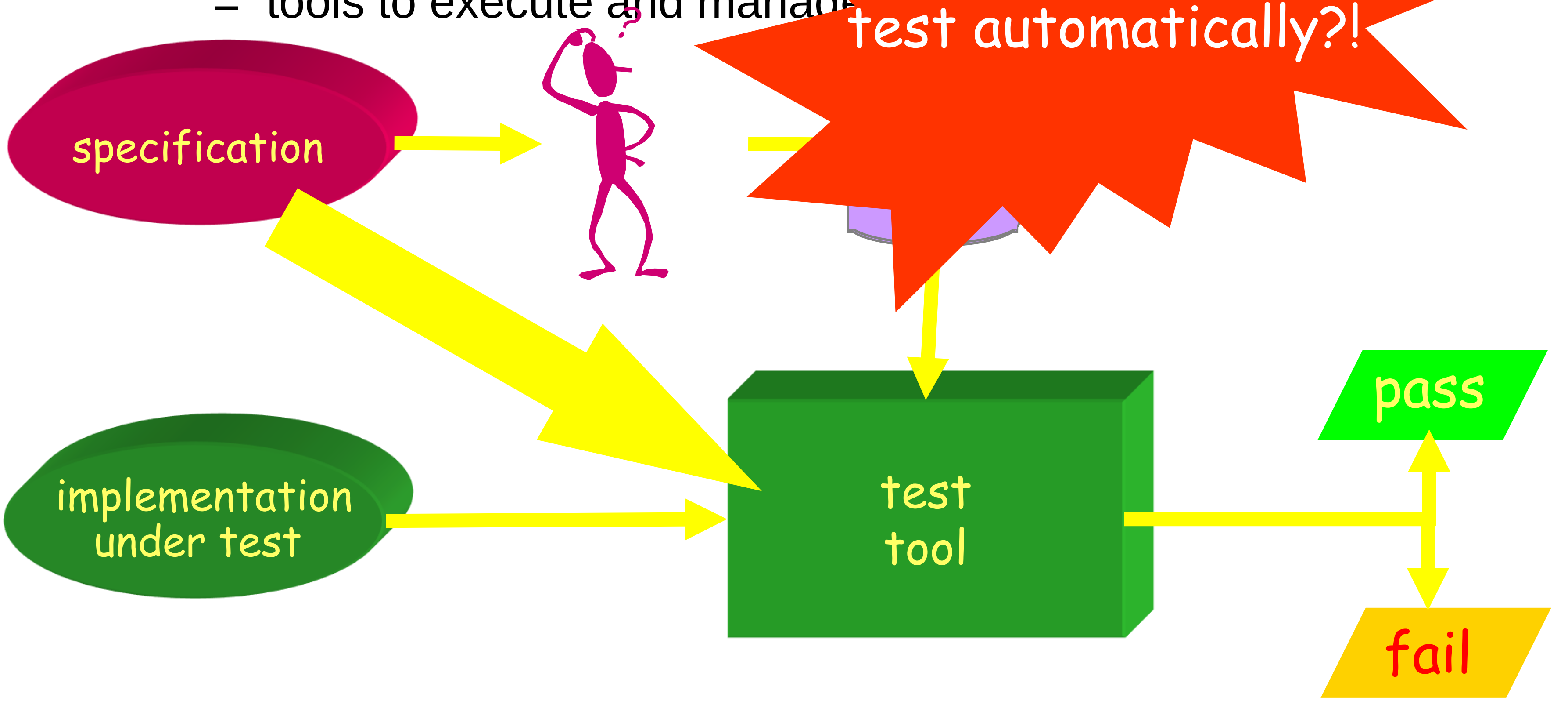
Improvements possible
with formal methods ! ?

TYPES OF TESTING



TEST AUTOMATION

Traditional test automation
= tools to execute and manag



Why not generate test automatically?!

VERIFICATION AND TESTING

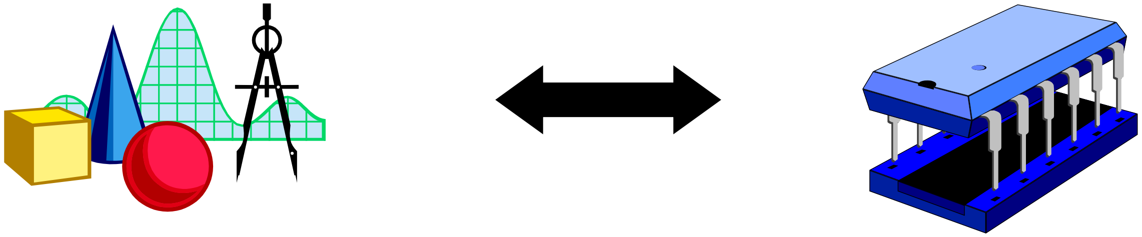
Verification :

- formal manipulation
- prove properties
- performed on model

Testing :

- experimentation
- show error
- concrete system

formal world



concrete world

Verification is only as good as the validity of the model on which it is based

Testing can only show the presence of errors, not their absence

TESTING WITH FORMAL METHODS

- Testing with respect to a formal specification
- Precise, formal definition of correctness :
  good and unambiguous basis for testing
- Formal validation of tests
- Algorithmic derivation of tests :
  tools for automatic test generation
- Allows to define measures expressing coverage
  and quality of testing

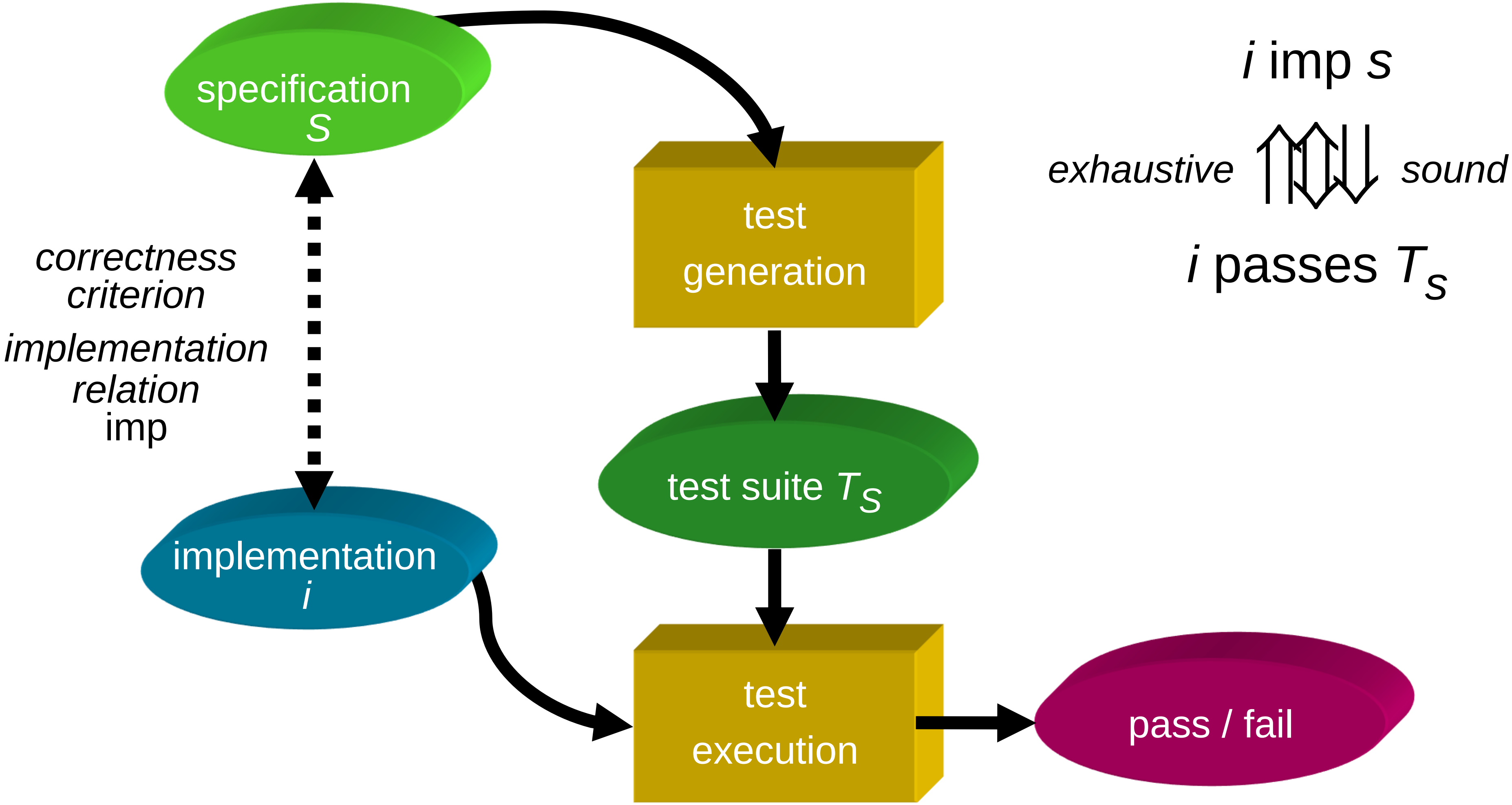
CHALLENGES OF TESTING THEORY

- Infinity of testing:
  - too many possible input combinations: infinite breadth
  - too many possible input sequences: infinite depth
  - too many invalid and unexpected inputs

- Exhaustive testing never possible:
  - when to stop testing ?
  - how to invent effective and efficient test cases with high probability of detecting errors ?

- Optimization problem of testing yield vs. effort
  - usually stop when time is over ......

FORMAL TESTING



specification *S*

correctness criterion

implementation relation imp

implementation *i*

test generation

test suite T_S

test execution

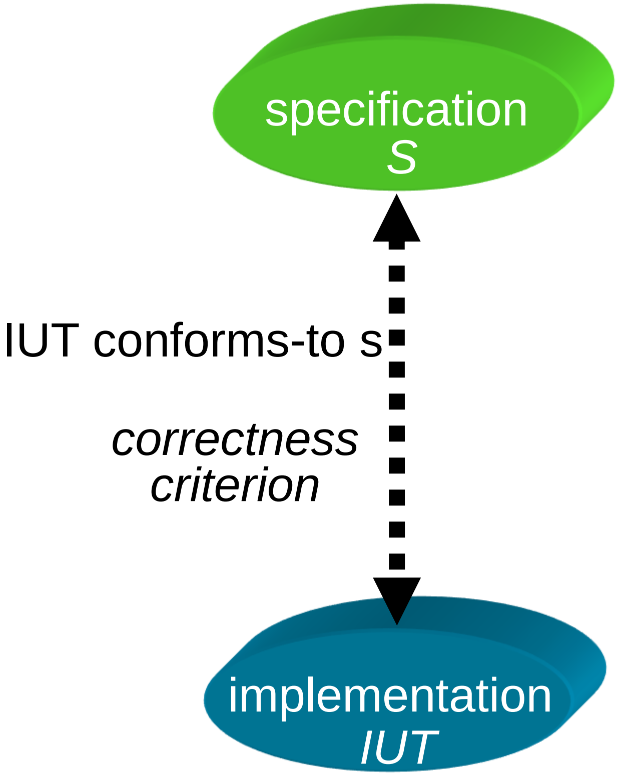
pass / fail

i imp s

exhaustive $\Uparrow\Downarrow$ *sound*

i passes T_s

FORMAL TESTING : CONFORMANCE



$s \in SPECS$	specification
IUT	implementation under test

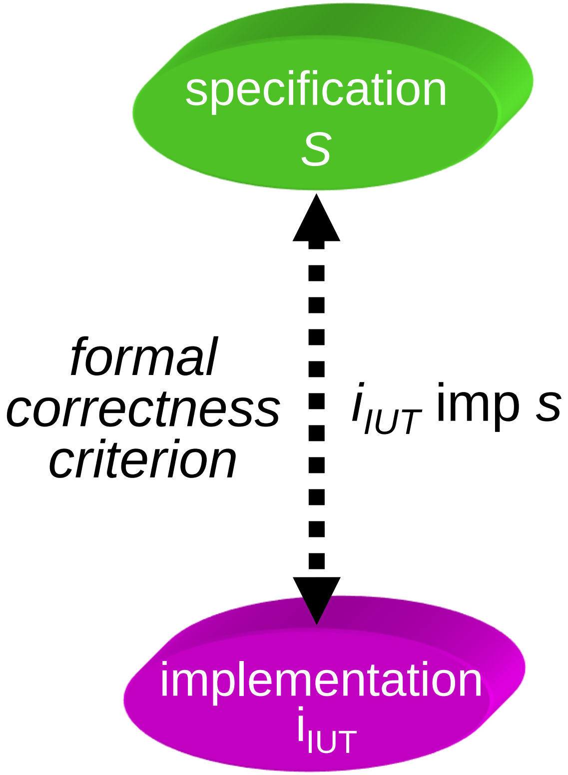
IUT is concrete, physical object

Model the physical world

But *IUT* is black box ! ?

Assume that model i_{IUT} exists

FORMAL TESTING : CONFORMANCE



specification

s

formal correctness criterion

i_{IUT} imp s

implementation

i_{IUT}

$s \in SPECS$ Specification

$i_{IUT} \in MODS$ model of *IUT*

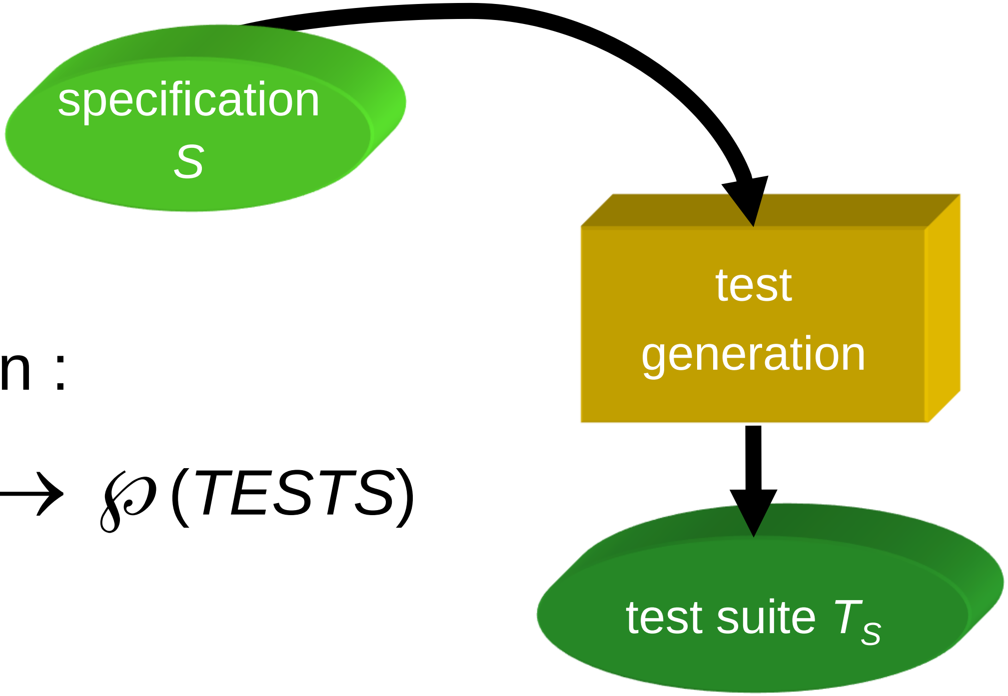
Test assumption :

each concrete *IUT* can be modelled by some $i_{IUT} \in MODS$

Conformance : i_{IUT} *imp* s

i_{IUT} is not known ;
testing to learn about i_{IUT}

FORMAL TESTING : TEST DERIVATION



Test generation :

$der : SPECS \rightarrow \wp(TESTS)$

Test suite - set of test cases : $T \subseteq TESTS$

Test case : $t \in TESTS$

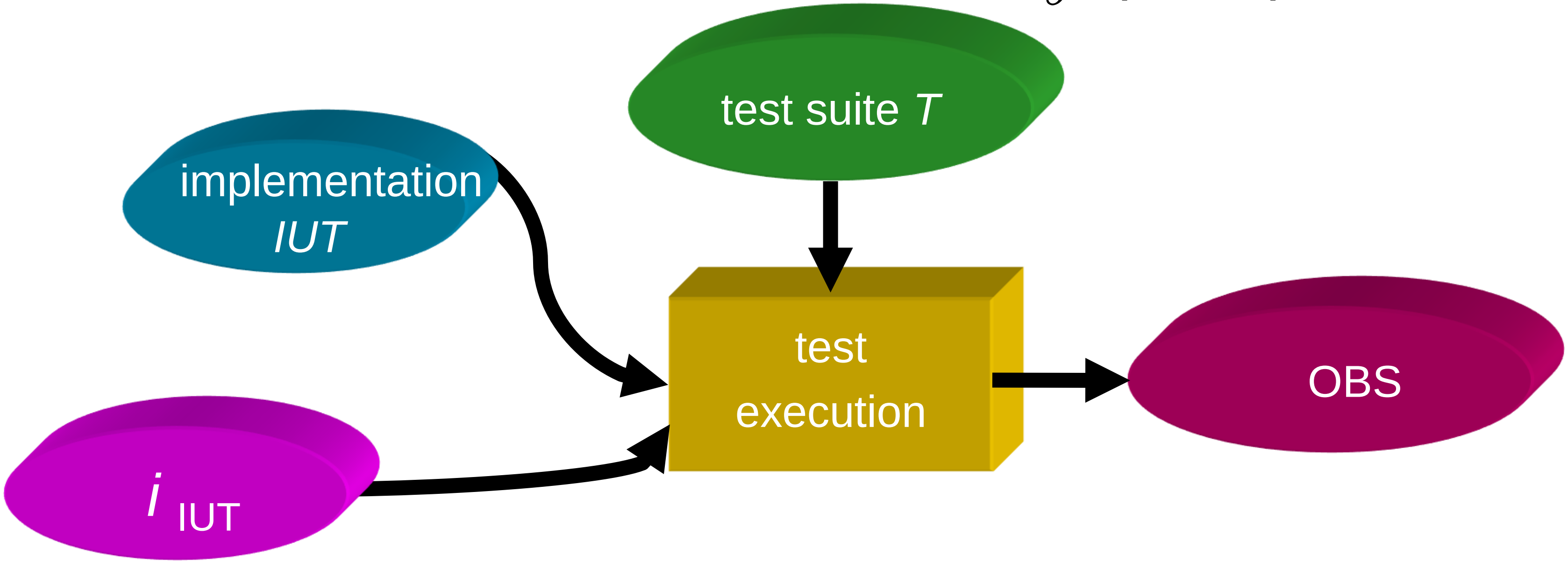
FORMAL TESTING : TEST EXECUTION

Test execution leads to a set of observations :

$$exec : \ TESTS \times IMPS \rightarrow \wp(OBS)$$

Model of test execution :

$$obs : \ TESTS \times MODS \rightarrow \wp(OBS)$$

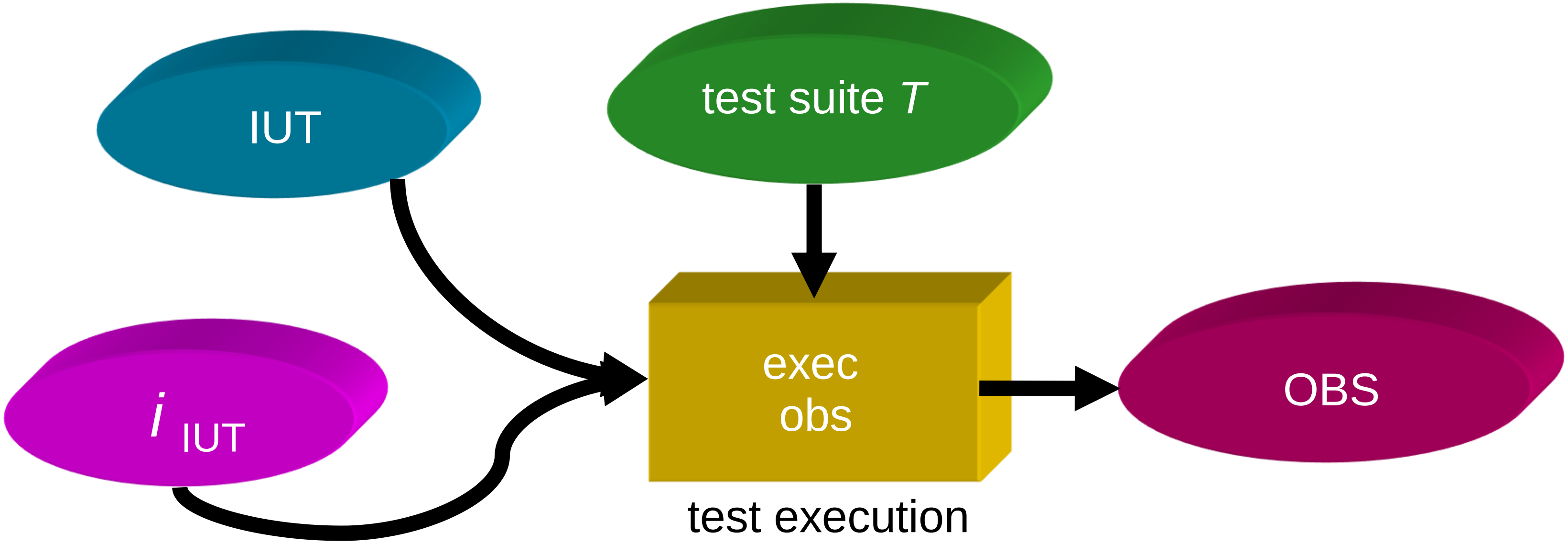


TEST HYPOTHESIS

Observational framework : *TESTS, OBS, exec, obs*

Test hypothesis : for all *IUT* in *IMPS* . $\exists\, i_{IUT} \in MODS$.

$$\forall t \in TESTS \,.\, exec\,(t, IUT) = obs\,(t, i_{IUT})$$

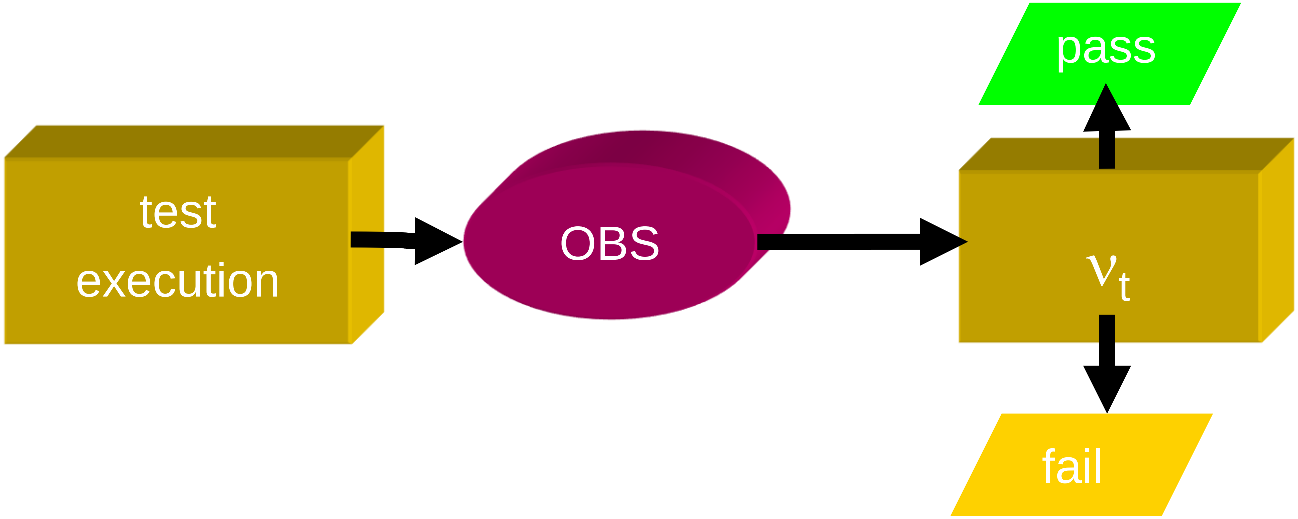


test execution

FORMAL TESTING : VERDICTS

Observations are interpreted :

$$\nu_t : \wp(OBS) \rightarrow \{ fail, pass \}$$



TESTING FOR CONFORMANCE

$IUT \ passes \ T_s \overset{?}{\Leftrightarrow} i \ conforms\text{-}to \ s$

$\boldsymbol{IUT \ passes \ T_s}$

$\Leftrightarrow \quad \boldsymbol{\forall t \in T_s . \ IUT \ passes \ t}$

$IUT \ passes \ T_s \Leftrightarrow_{def} \forall t \in T_s . \ IUT \ passes \ t$

$\Leftrightarrow \quad \boldsymbol{\forall t \in T_s . \ v_t (exec (t, IUT)) = pass}$

$IUT \ passes \ t \Leftrightarrow_{def} v_t (exec (t, IUT)) = pass$

$\Leftrightarrow \quad \boldsymbol{\forall t \in T_s . \ v_t (obs (t, i_{IUT})) = pass}$

Test hypothesis :

$\Leftrightarrow \quad \boldsymbol{i_{IUT} \ imp \ s}$

Proof obligation :

$\forall i \in MODS$

$(\forall t \in T_s . \ v_t (obs (t, i)) = pass) \Leftrightarrow i \ imp \ s$

$\Leftrightarrow \quad$ **IUT conforms-to s**

Definition : $i \ conforms\text{-}to \ s$

TESTING FOR CONFORMANCE

Proof obligation :

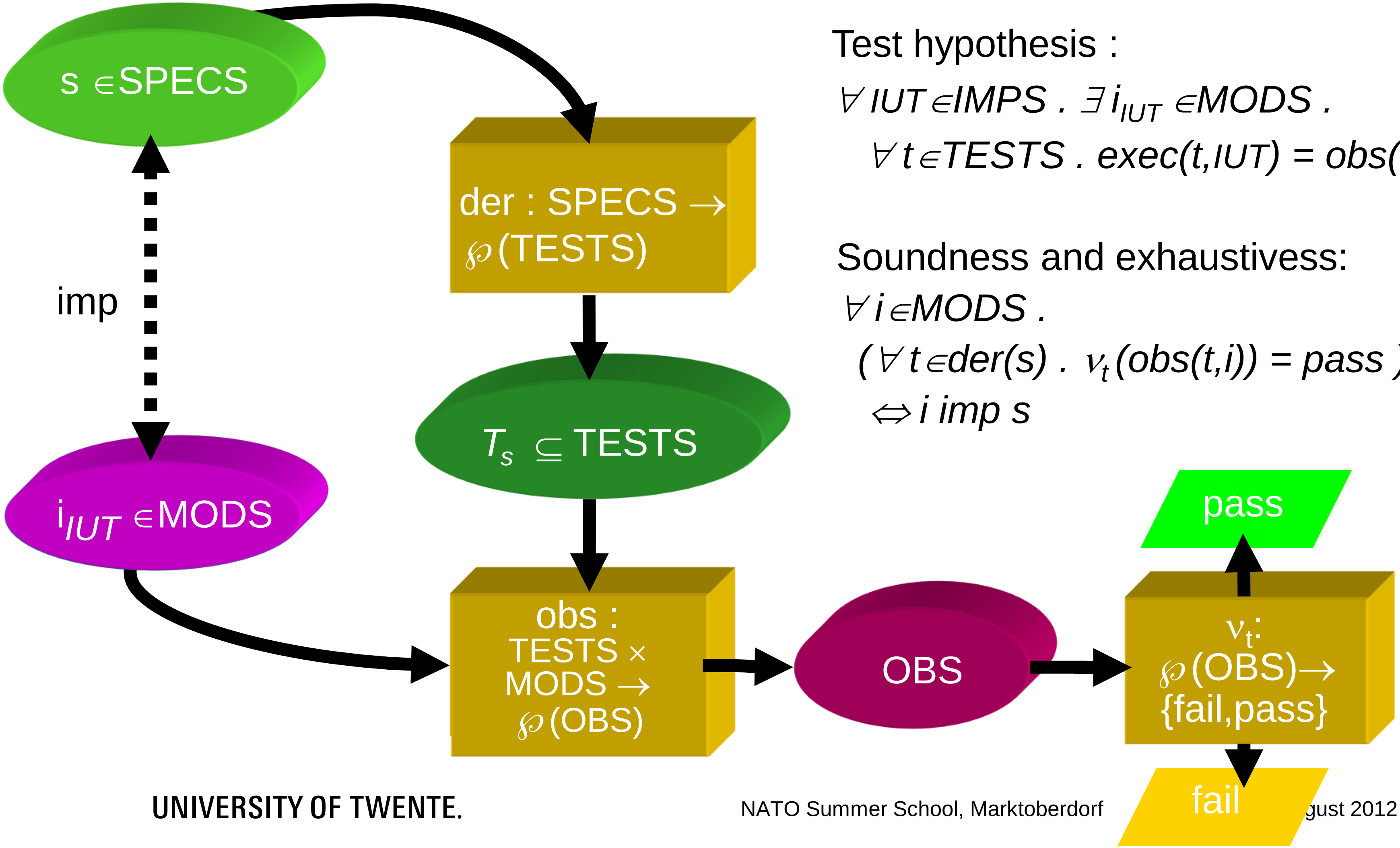
$$\forall\, i \in MODS\ .$$

$$(\ \forall\, t \in T_s\ .\ \ \nu_t\,(\,obs\,(\,t, i\,)\,)\ =\ pass\ )\ \ \Leftrightarrow\ \ i\ imp\ s$$

Proof of completeness on model leads to completeness for tested systems :

$$IUT\ \ passes\ \ T_s \quad \overset{exhaustive}{\Longrightarrow} \quad \underset{sound}{\Longleftarrow} \quad i\ conforms\text{-}to\ \ s$$

FORMAL TESTING



Test hypothesis :

$\forall\, IUT \in IMPS\ .\ \exists\, i_{IUT} \in MODS\ .$

$\quad \forall\, t \in TESTS\ .\ exec(t,IUT) = obs(t,i_{IUT})$

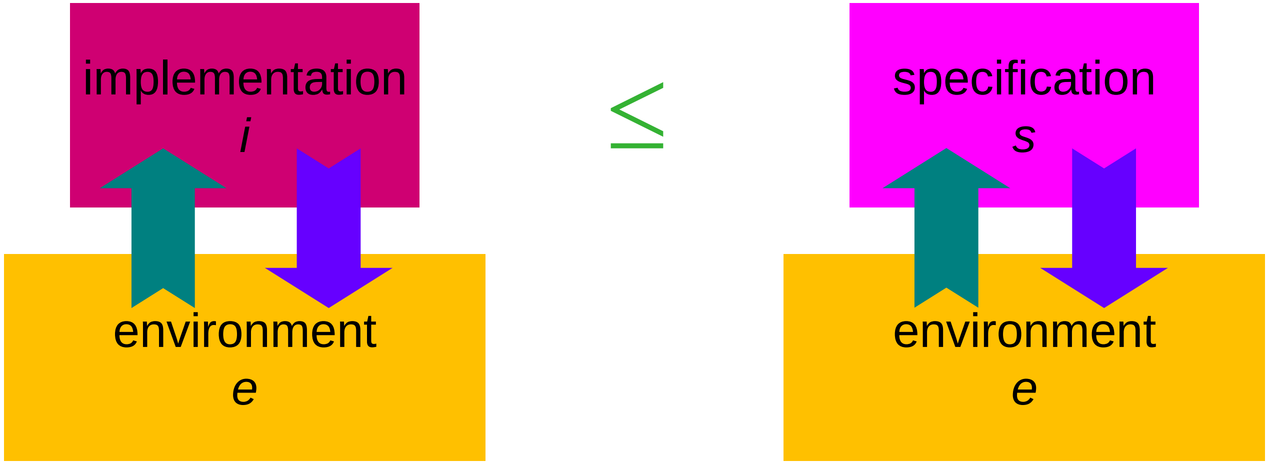
Soundness and exhaustivess:

$\forall\, i \in MODS\ .$

$\quad (\forall\, t \in der(s)\ .\ \nu_t(obs(t,i)) = pass\ )$

$\quad \Leftrightarrow i\ imp\ s$

TESTING PREORDERS



For all environments *e*

all observations of an implementation *i* in *e* should be explained by observations of the specification *s* in *e*.

$i \leq s \quad \Leftrightarrow \quad \forall\, e \in Env \,.\; obs\,(\,e, i\,) \;\subseteq\; obs\,(e, s\,)$

↓ ↓ ↓

? ? ?

LABELLED TRANSITION SYSTEMS

An LTS is a tuple $A = \left\langle S, S^0, L, \rightarrow \right\rangle$ with

- S a set of states
- $S^0 \subseteq S$ a nonempty set of initial states
- L a set of labels; $L_\tau = L \cup \{\tau\}$ with τ the invisible action
- $\rightarrow \subseteq S \times L_\tau \times S$ the transition relation;

We write:

- $s \xrightarrow{a} s'$ for $(s,a,s') \in \rightarrow$
- $s \xrightarrow{\sigma} s'$ for $\sigma = a_1 \ldots a_n$ and $s = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$
- $s \overset{\sigma}{\Longrightarrow} s'$ for $\sigma = a_1 \ldots a_n$ and $s = s_0 \xrightarrow{\tau^{k_1}} \xrightarrow{a_1} \xrightarrow{\tau^{m_1}} s_1 \xrightarrow{\tau^{k_2}} \xrightarrow{a_2} \xrightarrow{\tau^{m_2}} \ldots \xrightarrow{\tau^{k_n}} \xrightarrow{a_n} \xrightarrow{\tau^{m_n}} s_n$
- $CTraces = \{\, \sigma \mid \exists s_0 \in S_0 . s_0 \overset{\sigma}{\Longrightarrow} s \wedge s \overset{a}{\not\Rightarrow} \; for\ any\ a \in L \,\}$

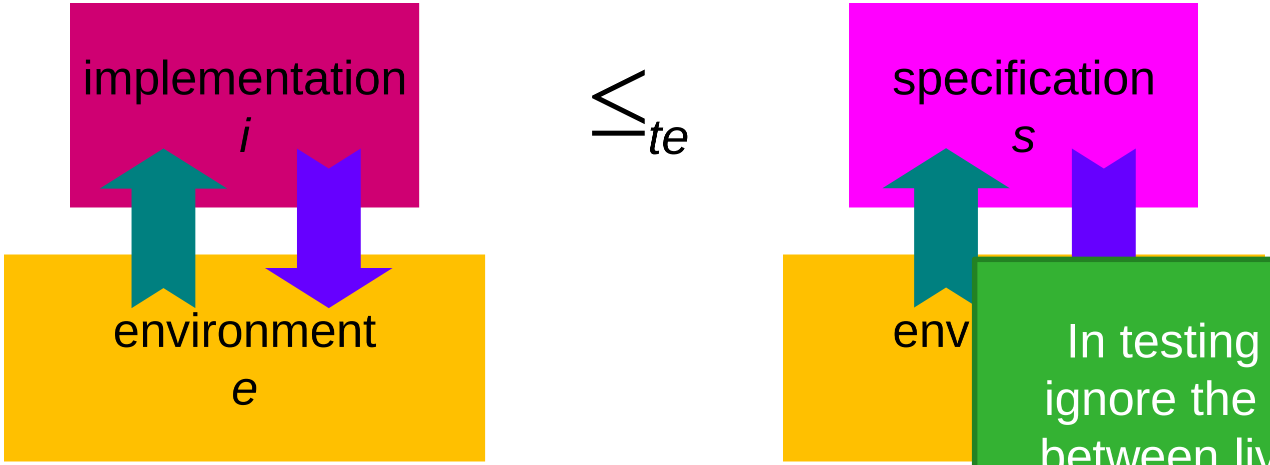
PARALLEL COMPOSITION

Let $A =$ and $B = \left\langle S_2 \ , S_2^0, L_2, \to_2 \right\rangle$ be two LTSs.

Then $A \| B = \left\langle S_1 \times S_2, \left(S_1^0, S_2^0\right), L_1 \cup L_2, \to \right\rangle$ with

$$\begin{aligned}
\to = \{ &\left(\left(s,t\right), a, \left(s',t'\right)\right) \mid \left(s,a,s'\right) \in \longrightarrow_1, \left(t,a,t'\right) \in \longrightarrow_2, a \neq \tau \} \\
\cup \{ &\left(\left(s,t\right), a, \left(s',t\right)\right) \mid \ \mid \left(s,a,s'\right) \in \longrightarrow_1, t \in S_2, a \in \left(L_1 \setminus L_2\right) \cup \{\tau\} \} \\
\cup \{ &\left(\left(s,t\right), a, \left(s,t'\right)\right) \mid \ \mid \left(t,a,t'\right) \in \longrightarrow_2, s \in S_1, a \in \left(L_2 \setminus L_1\right) \cup \{\tau\} \}
\end{aligned}$$

CLASSICAL TESTING PREORDER



implementation
i

$\leq_{te}$

specification
s

environment
e

env

In testing one may ignore the difference between livelock and deadlock

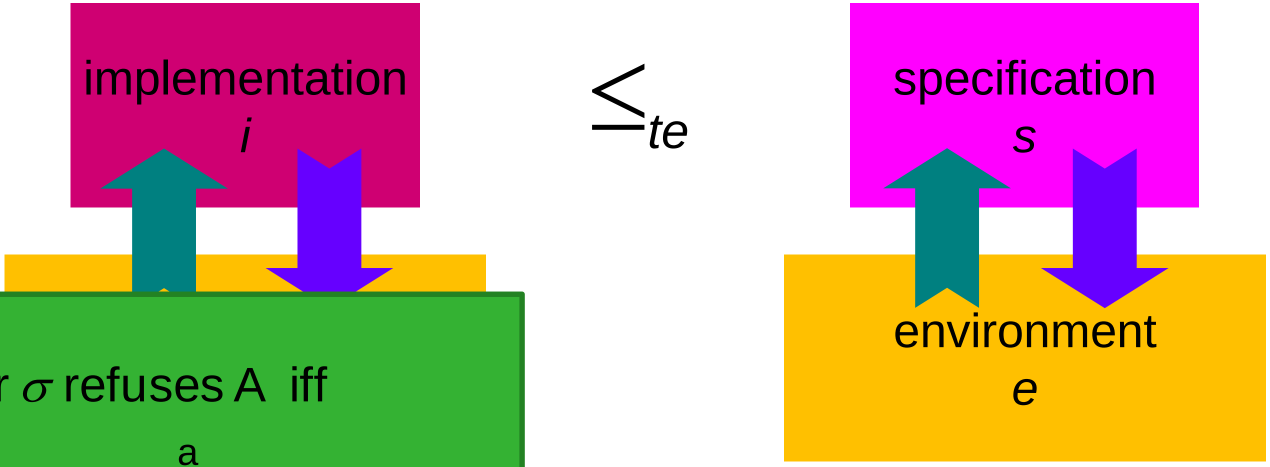
Philosophical question:
can we observe deadlocks?

$s\,(\,e, i\,) \;\subseteq\; obs \quad s\,)$

$\downarrow \qquad \downarrow$

$LTS(L) \quad CTraces(e||s)$

TESTING PREORDER



p after σ refuses A iff

$p \overset{\sigma}{\Rightarrow} p'$ and $p' \overset{a}{\not\Rightarrow}$ for all $a \in A$

$i \leq_{te}$ … $\sigma \in L^*$,

$\{ e||i$ deadlocks after $\sigma \} \subseteq \{ e||s$ deadlocks after $\sigma \}$

$\{ \sigma$ … $(e||i) \} \subseteq \{ \sigma$ Otraces$(e||s) \}$

$\Leftrightarrow \quad FP(i) \subseteq FP(s)$

$FP(p) = \{ \langle \sigma, A \rangle \mid p \text{ after } \sigma \text{ refuses } A \}$

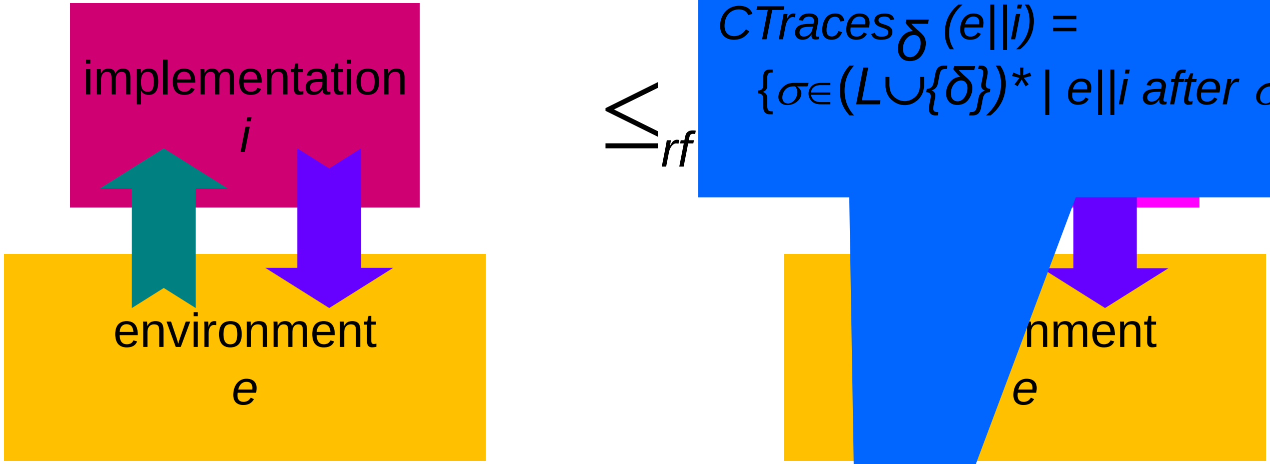
Can we distinguish between these machines?

QUIRKY COFFEE MACHINE [Langerak]

$\approx_{te}$

They are
testing equivalent!

REFUSAL PREORDER



$CTraces_{\delta}(e||i) =$
$\{\sigma \in (L \cup \{\delta\})^* \mid e||i \text{ after } \sigma \text{ refuses } L\}$

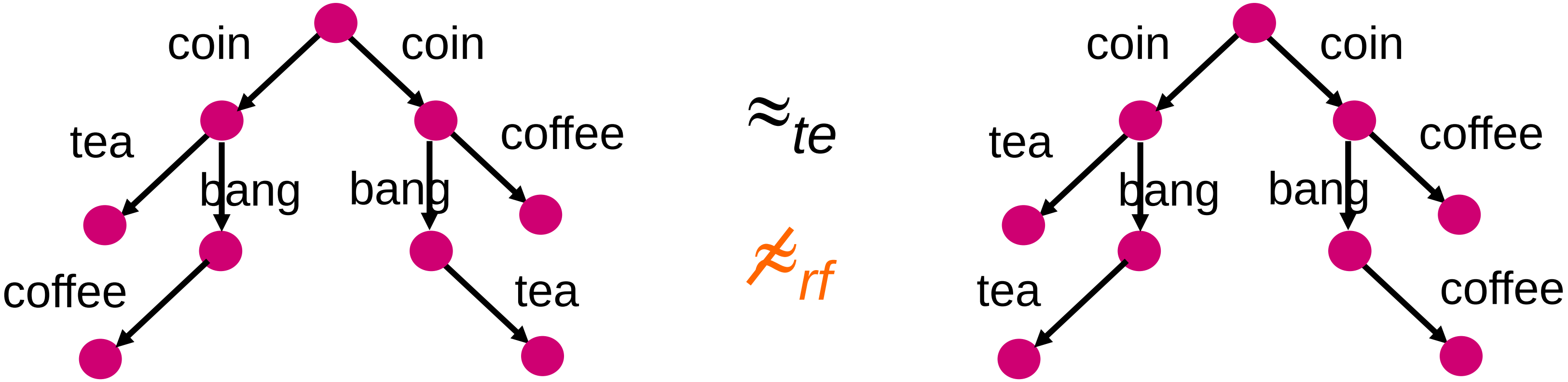
$i \leq_{rf} s \iff \forall e \in E\,.\; obs(e, i) \subseteq bs(e, s)$

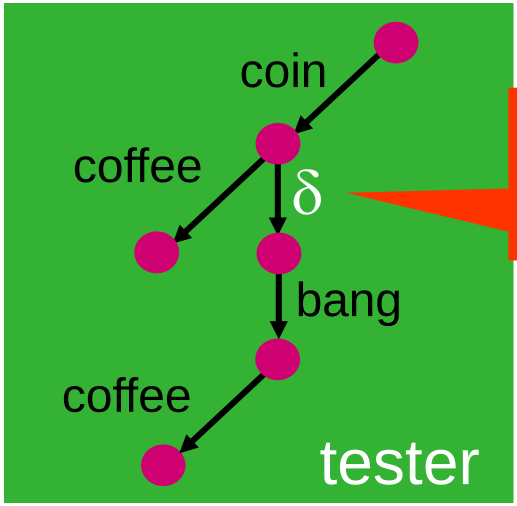
↓ ↓

$LTS(L \cup \{\delta\})$ $CTraces_{\delta}(e||i)$

e observes with δ deadlock on all alternative actions

QUIRKY COFFEE MACHINE REVISITED





δ only enabled
if coffee is not

CONTENTS

CONTENTS

1. Introduction control-oriented testing
2. Input-output conformance testing
3. Real-time conformance testing
4. Test coverage measures

I/O TRANSITION SYSTEMS

- testing actions are usually directed, i.e. there are inputs and outputs

  $L = L_{in} \cup L_{out}$ with $L_{in} \cap L_{out} = \emptyset$

- systems can always accept all inputs: *input enabledness*

  for all states s, for all $a \in L_{in}$ $s \overset{a}{\Rightarrow}$

- testers are I/O systems
  - output (stimulus) is input for the SUT
  - input (response) is output of the SUT

QUIESCENCE

- Because of input enabledness $S||T$ deadlocks iff T produces no stimuli and S no responses. This is known as *quiescence*
- Observing quiescence leads to two implementation relations for I/O systems I and S :

1. $I \leq_{iote} S$ iff for all I/O testers T :

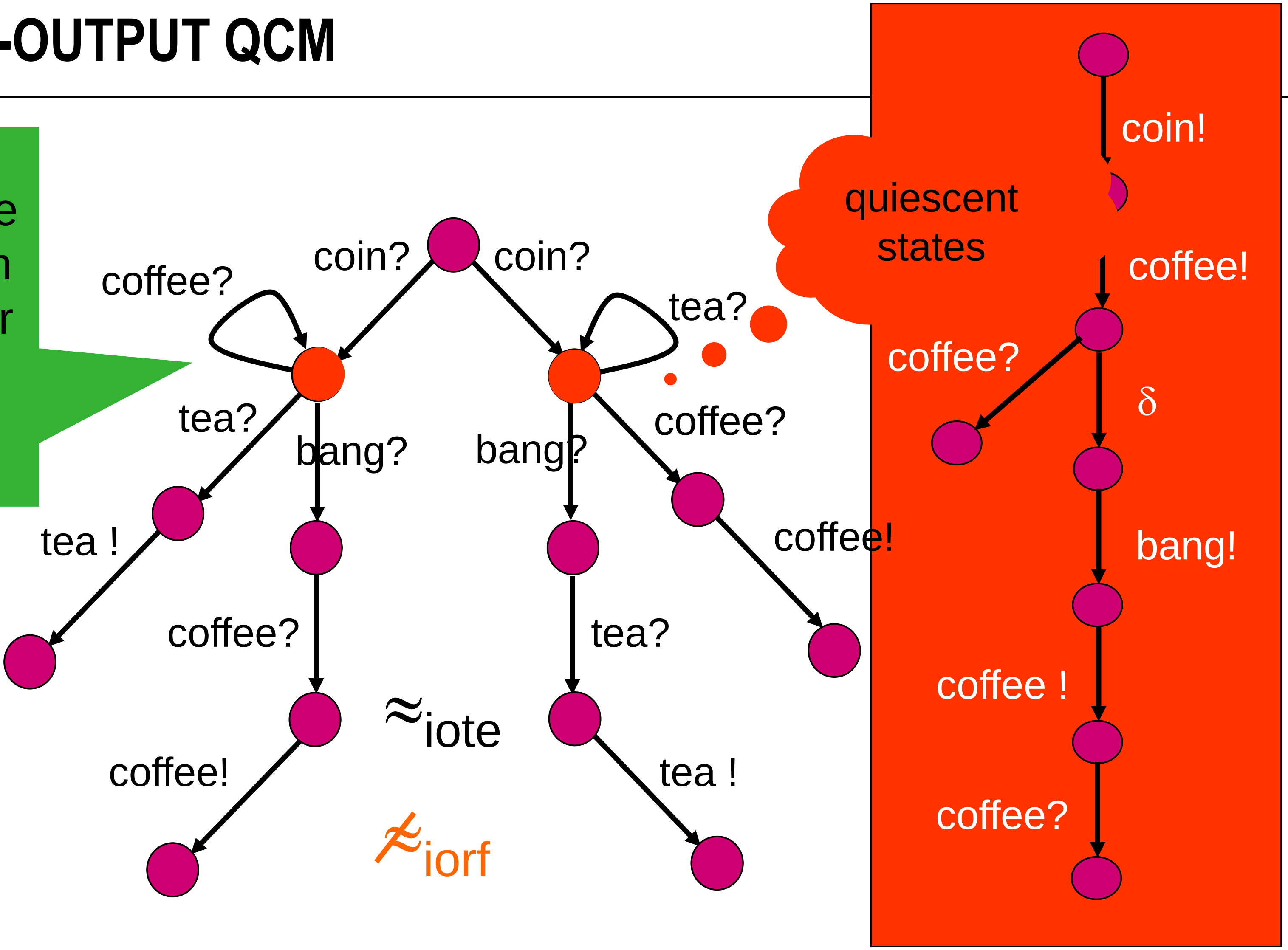
   $CTraces(I||T) \subseteq CTraces(S||T)$ *(quiescence)*

2. $I \leq_{iorf} S$ iff for all I/O testers T :

   $CTraces_{\delta}(I||T) \subseteq Ctraces_{\delta}(S||T)$ *(repetitive quiescence)*

INPUT-OUTPUT QCM

states must be saturated with input loops for input enabledness



quiescent states

QUIESCENT LABELLED TRANSITION SYSTEMS

A QLTS is an LTS $A = \langle S, S^0, L_1 \cup L_O^\delta, \rightarrow \rangle$

with special (output) label δ

such that if $s \xrightarrow{\delta} s'$ then $s' \xrightarrow{\delta} s'$ and s' is quiescent.

This definition is closed under determinisation.

Let $A = \langle S, S^0, L, \rightarrow \rangle$ be an LTS with $L = L_I \cup L_O$ and $\delta \notin L$,

then its underlying QLTS $\delta(A)$ is the QLTS

$\langle S, S^0, L \cup \{\delta\}, \rightarrow' \rangle$ with

$\rightarrow' = \rightarrow \cup \{(s, \delta, s) \mid s \in S, s$ is quiescent$\}$

Moreover, $traces_{\delta(A)} = traces_A \setminus \{\delta\}$

IMPLEMENTATION RELATION IOCO

Let i and s be be QTLSs

(possibly after applying $\delta(.)$)

over $L = L_I \cup L_O^{\delta}$, then we define

$$i \subseteq_{iorf} s \qquad \text{iff} \qquad \forall \sigma \in L^* \, out_i(\sigma) \subseteq out_s(\sigma)$$

For implementations we will require input-enabledness,

But not for specifications.

In this setting it makes sense to restrict testing

to the traces of the implementation:

$$i \subseteq_{ioco} s \qquad \text{iff} \qquad \boxed{\forall \sigma \in traces_s} \, out_i(\sigma) \subseteq out_s(\sigma)$$

INTUITION BEHIND *IOCO*

$i \subseteq_{ioco} s$ iff $\forall \sigma \in traces_s \; out_i(\sigma) \subseteq out_s(\sigma)$

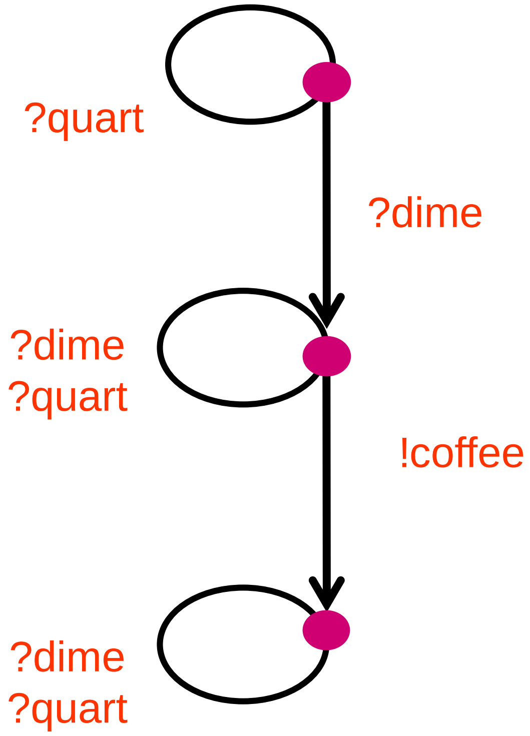
Intuition:

i ioco-conforms to *s*, iff

1. if *i* produces output *x* after a specified trace σ,
   then *s* can produce *x* after σ
2. if *i* cannot produce any output after a specified trace σ,
   then *s* cannot produce any output after σ (*quiescence* δ)

ADDING QUIESCENCE

i



?quart

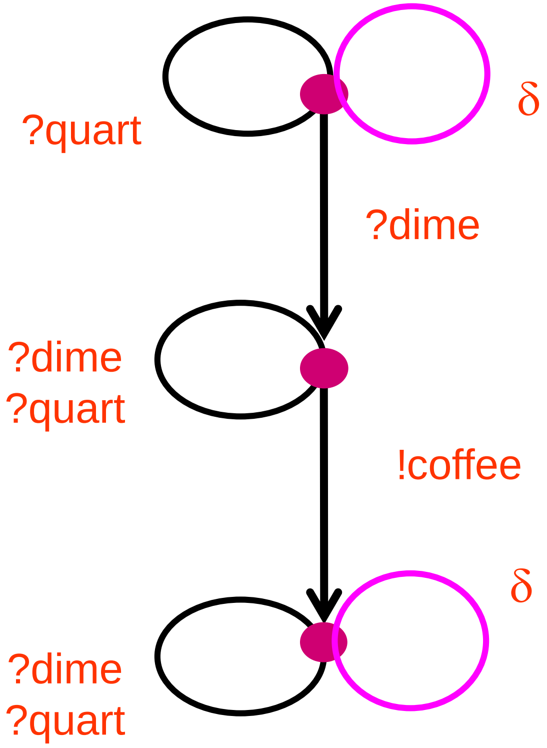
?dime

?dime
?quart

!coffee

?dime
?quart

δ(*i*)



?quart

δ

?dime

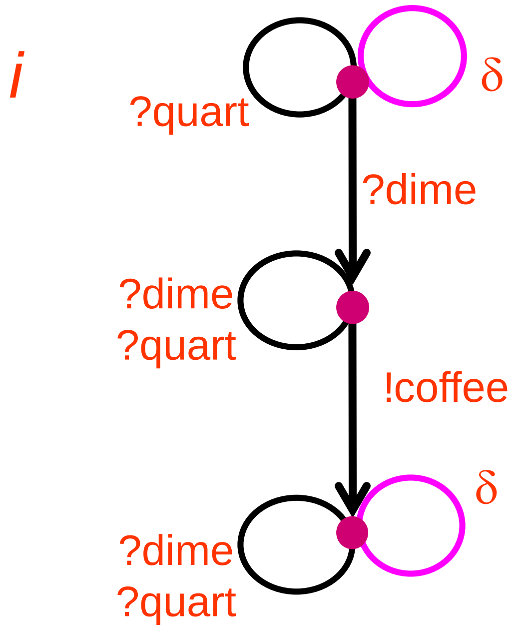
?dime
?quart

!coffee

δ

?dime
?quart

CALCULATING *OUT*



$out_i\,(\varepsilon) = \{\, \delta \,\}$

$out_i\,(\text{?dime}) = \{\, \text{!coffee} \,\}$

$out_i\,(\text{?dime.?dime}) = \{\, \text{!coffee} \,\}$

$out_i\,(\text{?dime.!coffee}) = \{\, \delta \,\}$

$out_i\,(\text{?quart}) = \{\, \delta \,\}$

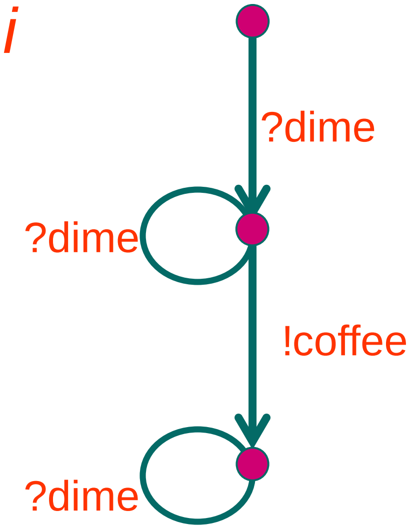
$out_i\,(\text{!coffee}) = \emptyset$

$out_i\,(\text{?dime.!tea}) = \emptyset$

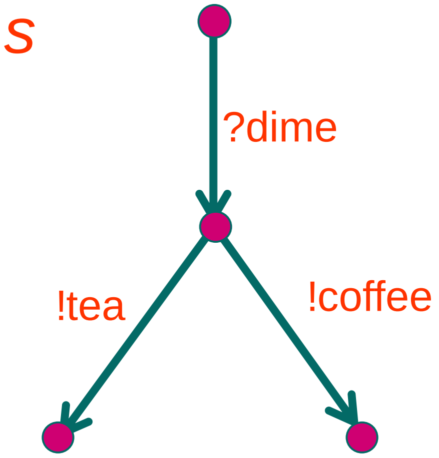
$out_i\,(\delta) = \{\delta \,\}$

IMPLEMENTATION RELATION *IOCO*

$i \subseteq_{ioco} s$ iff $\forall \sigma \in traces_s \; out_i(\sigma) \subseteq out_s(\sigma)$



$\subseteq_{ioco}$

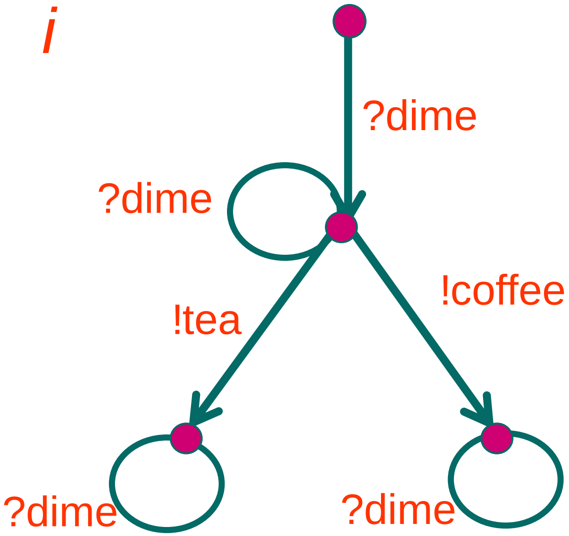


out_i(?dime) = { !coffee }

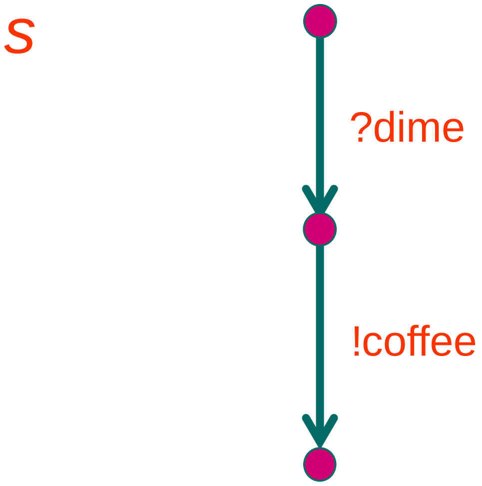
out_s(?dime) = { !coffee, !tea }

IMPLEMENTATION RELATION *IOCO*

$i \sqsubseteq_{ioco} s$ iff $\forall \sigma \in traces_s \; out_i(\sigma) \subseteq out_s(\sigma)$



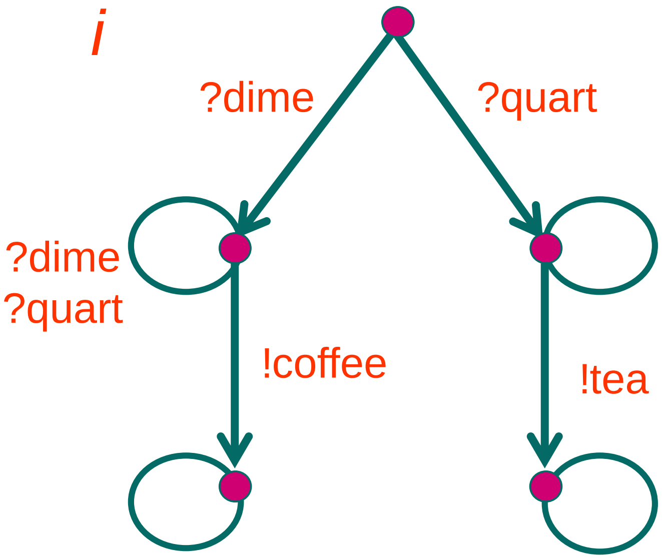
$\not\sqsubseteq_{ioco}$



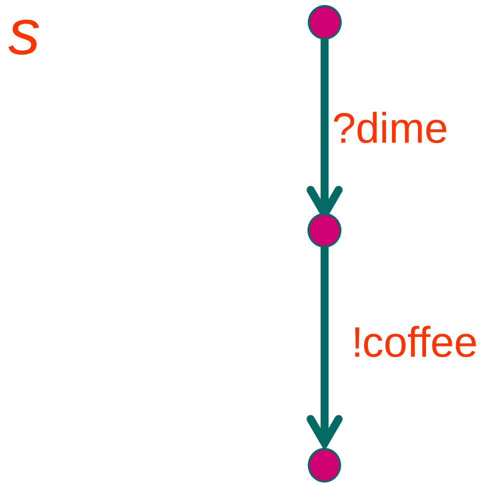
out_i(?dime) = { !coffee, !tea } $\not\subset$ out_s(?dime) = { !coffee}

IMPLEMENTATION RELATION *IOCO*

$i \sqsubseteq_{ioco} s$ iff $\forall \sigma \in traces_s \; out_i(\sigma) \subseteq out_s(\sigma)$



$\sqsubseteq_{ioco}$



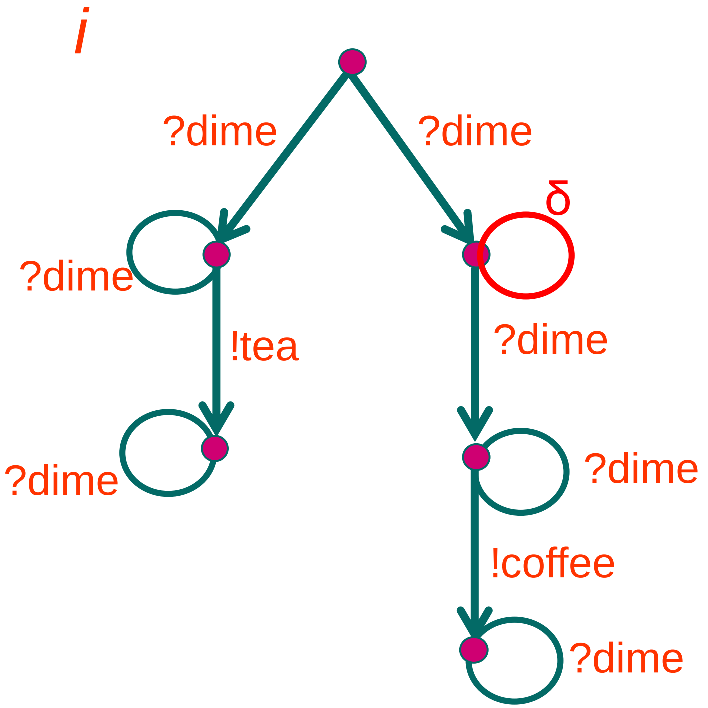
out_i(?dime) = { !coffee }
out_i(?quart) = { !tea }

out_s(?dime) = { !coffee }
out_s(?quart) = $\varnothing$

But ?quart $\notin$ $Traces_\delta$(s)

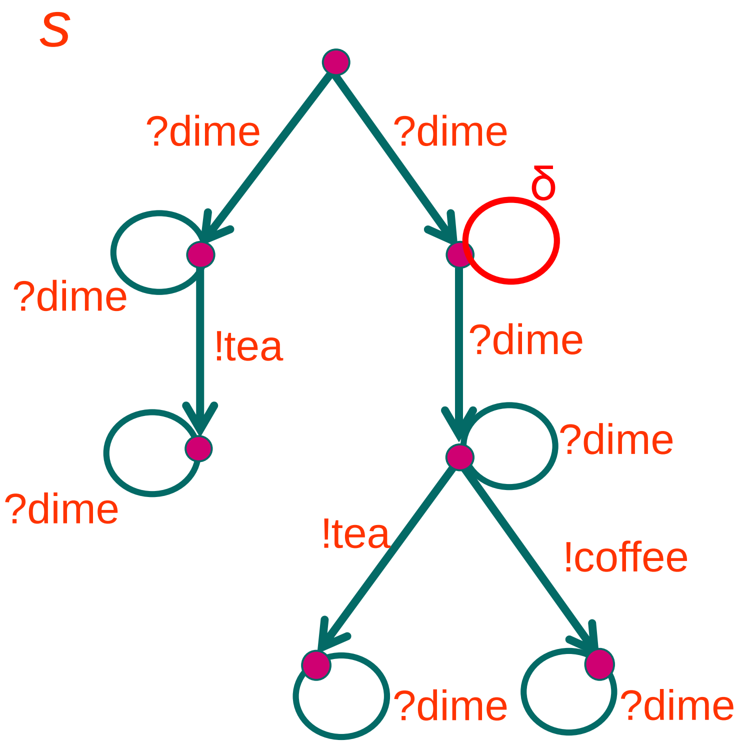
IMPLEMENTATION RELATION *IOCO*

$i \subseteq_{ioco} s$ iff $\forall \sigma \in traces_s \; out_i(\sigma) \subseteq out_s(\sigma)$



$i \subseteq_{ioco} s$

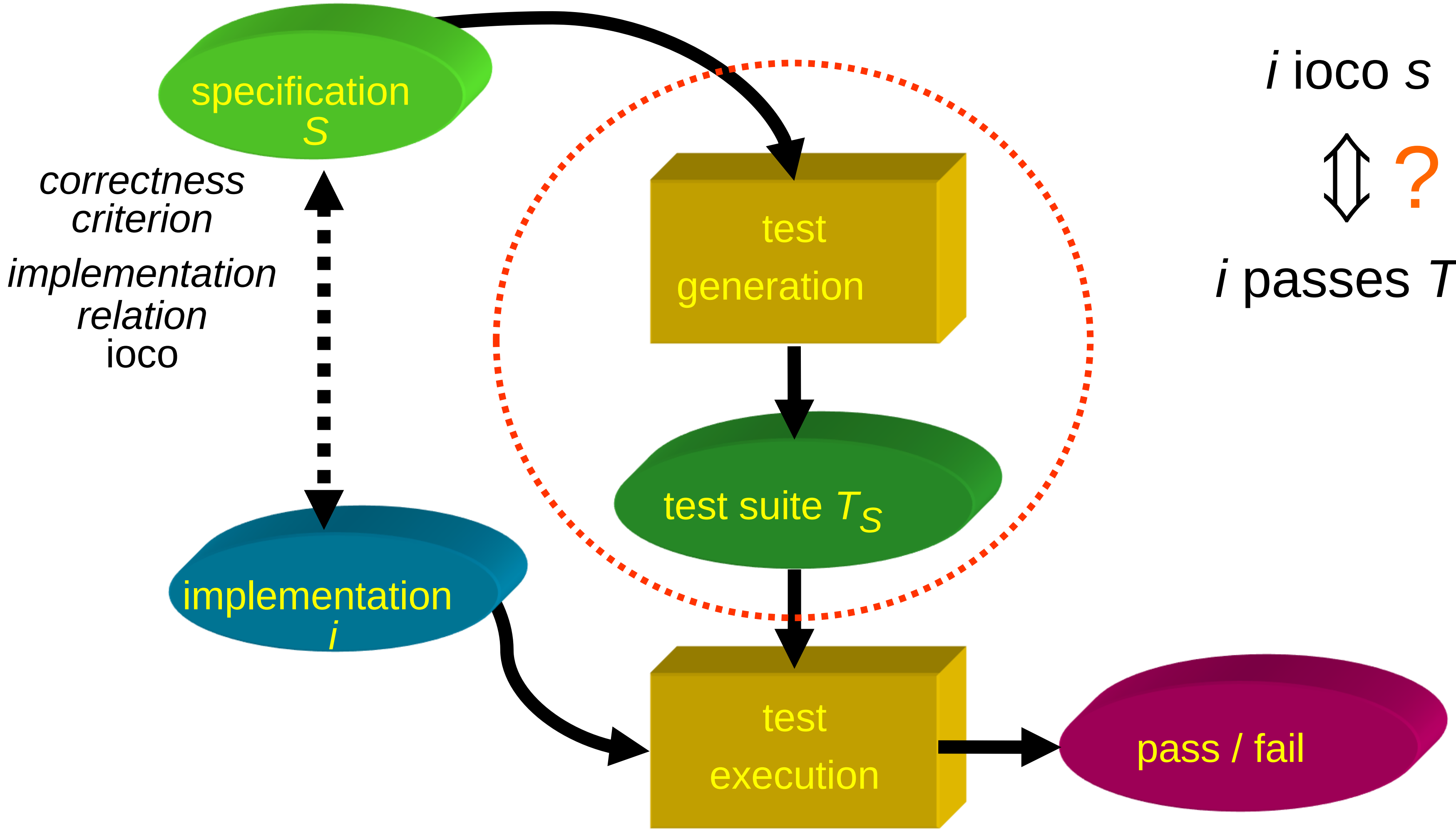
$s \not\subseteq_{ioco} i$



out_i (?dime.? dime) = out_s (? dime.? dime) = { !tea, !coffee }

out_i (? dime.δ.? dime) = { !coffee } $\neq$ out_s (? dime.δ.? dime) = { !tea, !coffee }

FORMAL TESTING



specification S

correctness criterion

implementation relation
ioco

test generation

test suite T_S

implementation i

test execution

pass / fail

i ioco s

$\Updownarrow$?

i passes T_s

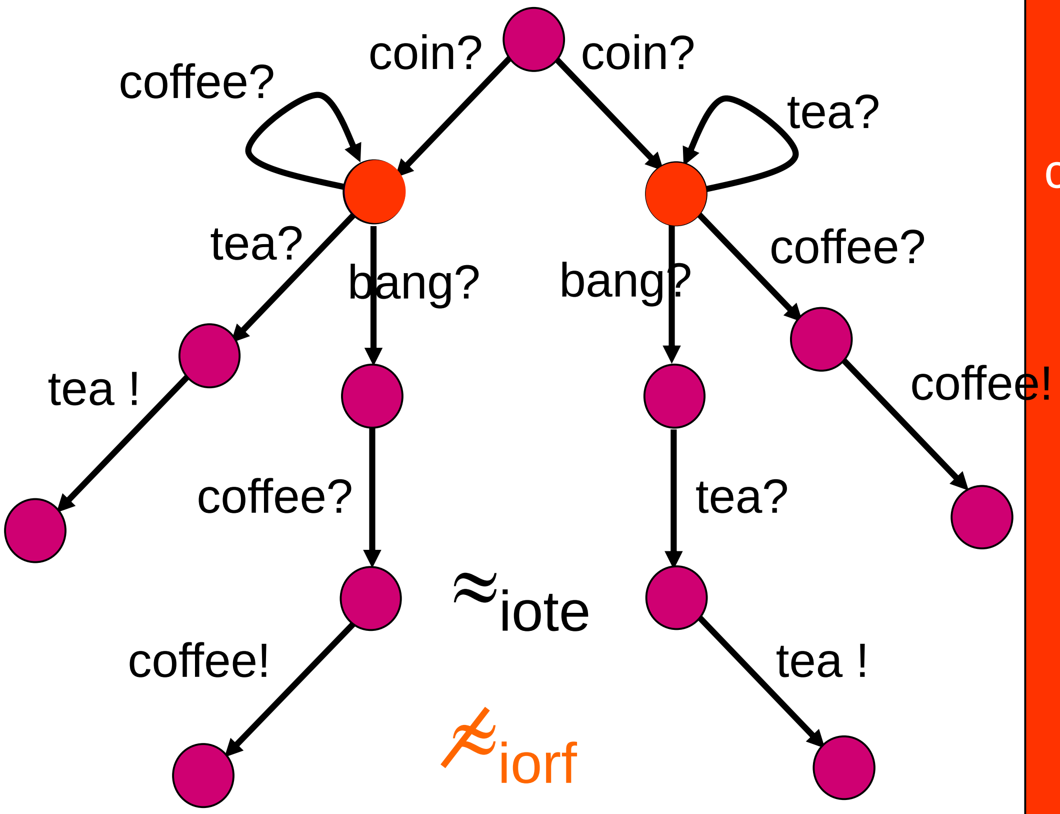
TEST CASES

A test (case) *t* over $L = L_I \cup L^{\delta}_O$ is an LTS with

- *t* is deterministic
- *t* does not contain an infinite
- *t* is acyclic and connected
- for all states *s* of *t* we have
    - *after*(*s*)= $\emptyset$, or (termination)
    - *after*(*s*)= L^{δ}_O , or (response observation)
    - *after*(*s*)= {a?} $\cup$ L_O (stimulus)

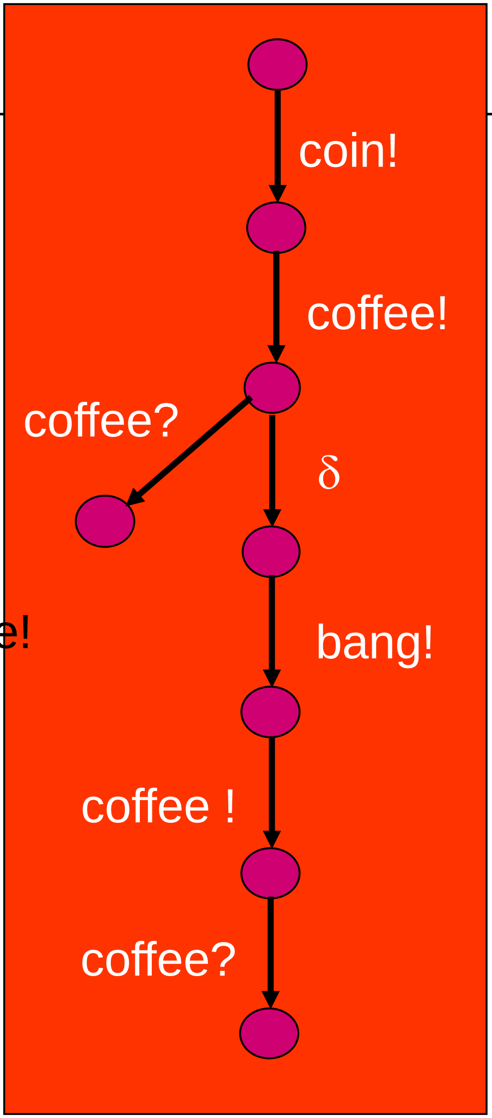
Alternatively, a test case can be characterised by the prefix-closed set of its traces

INPUT-OUTPUT QCM AGAIN



$\approx_{iote}$

$\not\approx_{iorf}$



TEST ANNOTATIONS

Let t be a test case:

- an annotation of t is a function

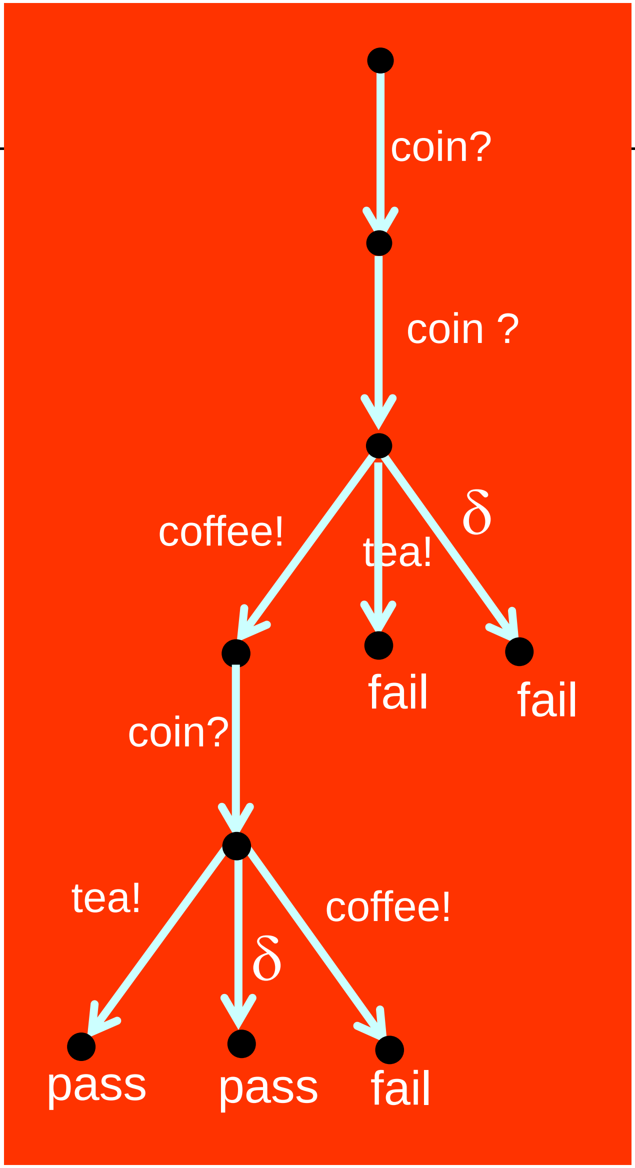
  a: $Ctraces_t \rightarrow \{ pass, fail \}$

- the pair $\dot{t} = (t,a)$ is an annotated test case

When a is clear from the context, or irrelevant,
we use t for both test case and its annotation.

ANNOTATED TEST CASES VS TTCN

test case t	
!coin	
!coin ; Start timer1	
?tea	fail
?timer1	fail
?coffee	
!coin ; Start timer2	
?tea	pass
?timer2	pass
?coffee	fail



EXECUTION AND EVALUATION

Let A be a QLTS over L and t a test over L.

The executions of t with A are defined as

- $exec_t(A) = Ctraces(t||A)$
- in fact, $exec_t(A) = Ctraces(t) \cap traces($

This can be lifted in the obvious way to sets of tests, i.e. test suites.

Let $\dot{t} = (t,a)$ be an annotated test case.

The verdict of $\dot{t}$ is the function

$$v_{\dot{t}} : \mathrm{QLTS}(L) \rightarrow \{pass, fail\} \text{ with}$$

$$v_{\dot{t}}(\mathrm{A}) = \begin{cases} pass & \text{if for all } \sigma \in exec_t(A) \;\; a(\sigma) = pass \\ fail & \text{otherwise} \end{cases}$$

A SOUND AND COMPLETE TEST SUITE

Given a specification *s* we define the annotation

$a_s^{ioco}(\sigma)$ = *fail* if $\exists \sigma_1 \in traces_s$,
and $\sigma_1 a \notin tra$
pass otherwise

Tests(*s*) contains all tests over the same label set *L* as *s*

Given *s* and any $t \in Tests(s)$, its annotated version (t, a_s^{ioco}) is *sound* w.r.t. *s* under $\subseteq_{ioco}$.

The test suite $T=\{(t, a_s^{ioco}) \mid t \in Tests(s)\}$ is *sound* and *complete* w.r.t. *s* under $\subseteq_{ioco}$.

DESIRABLE TEST CASE PROPERTIES

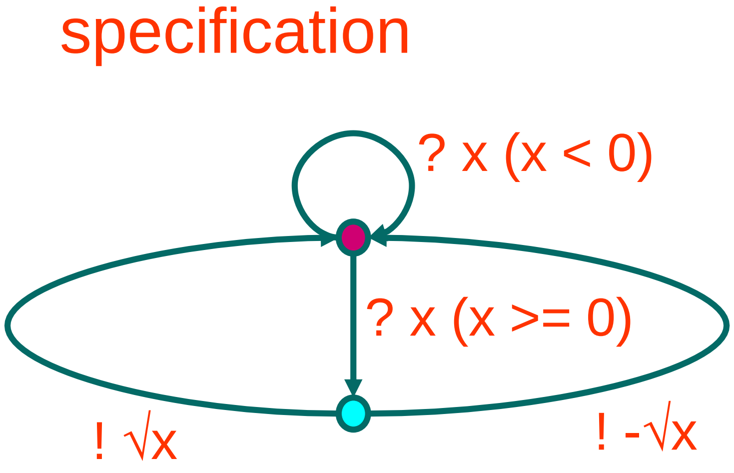
Let *s* be specification over a label set *L*, then

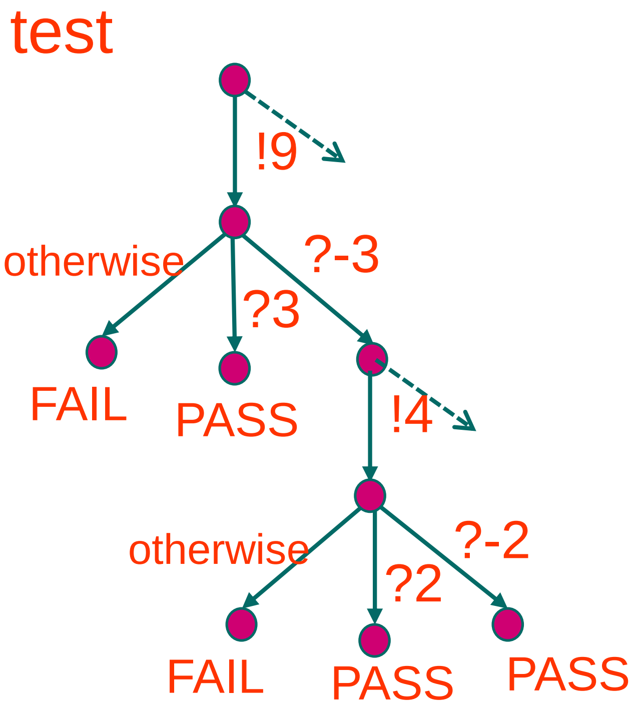
- a test *t* is *fail-fast* w.r.t. *s* if
  $\sigma \notin traces_s$ implies that $\forall a \in L \; \sigma a \notin t$

- a test *t* is *input-minimal* w.r.t. *s* if
  for all $\sigma a? \in t$ with $a? \in L_I$ it holds that
  $\sigma \in traces_s$ implies $\sigma a? \in traces_s$

TEST GENERATION EXAMPLE

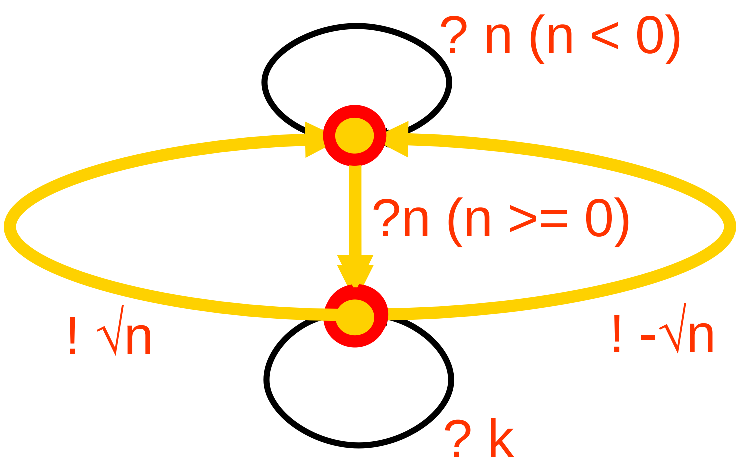
Equation solver for $y^2=x$

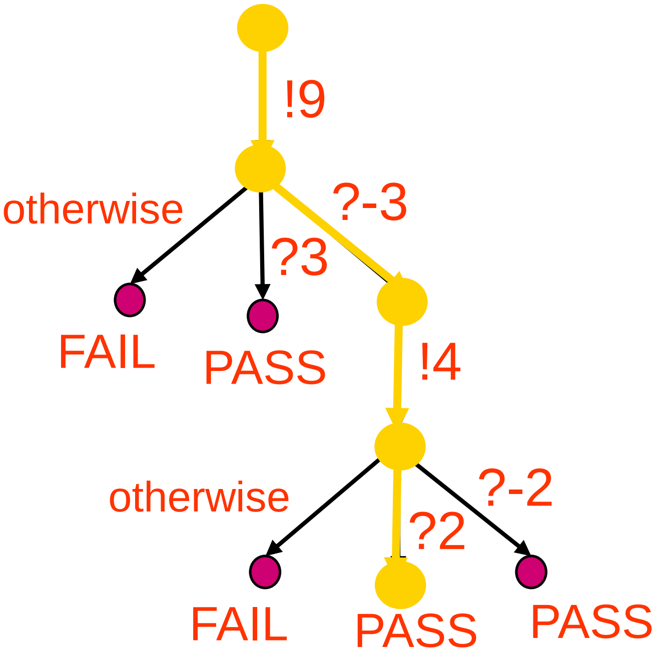




TEST EXECUTION EXAMPLE

implementation || test





ANNOTATED TEST GENERATION AL THM

To g rate a test case ystem

speci ith *S* s *S*)

Apply t

fail-fast, input-minimal,
ioco-sound &
(in the limit) ioco-complete

d outputs *o*!

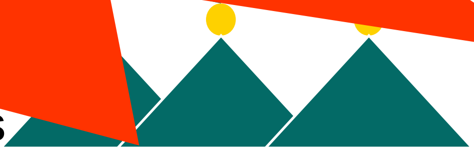
spec

input

$t \in S$

outputs

ve ou (no δ)



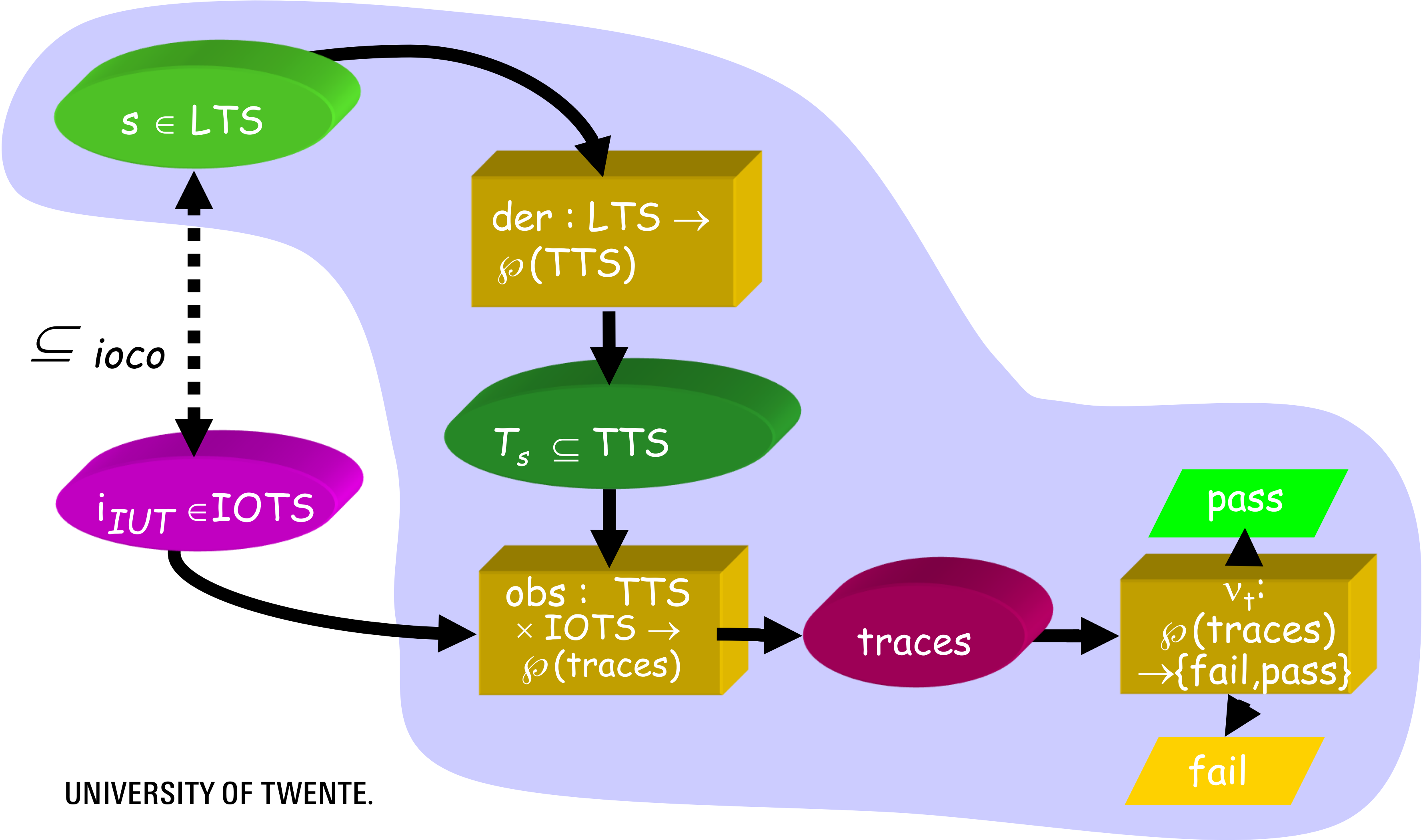
$t(S\ after\ o!)$

Allowed outputs:

$out_s(S) = \cup_{t \in S}\ out_s(t)$

$t(S\ after\ i?)$

FORMAL TESTING WITH TRANSITION SYSTEMS



$s \in LTS$

der : $LTS \rightarrow \wp(TTS)$

$T_s \subseteq TTS$

$\subseteq$ ioco

$i_{IUT} \in IOTS$

obs : $TTS \times IOTS \rightarrow \wp(traces)$

traces

ν_t: $\wp(traces) \rightarrow \{fail, pass\}$

pass

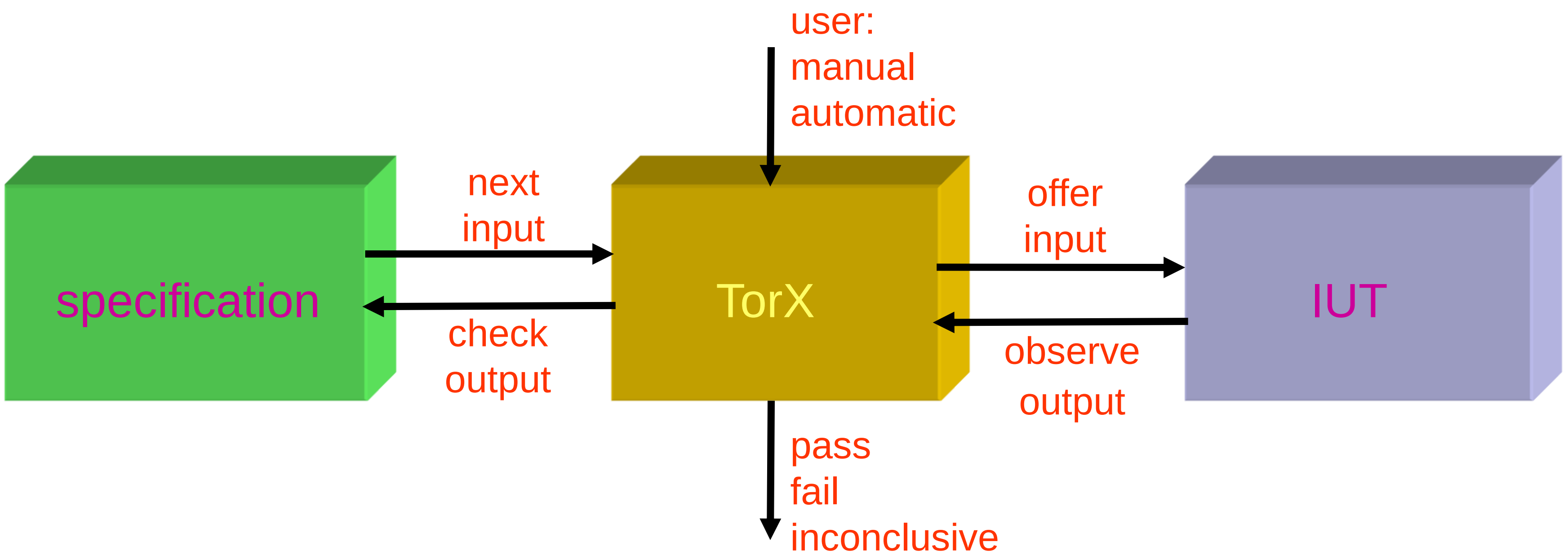
fail

SOME TEST GENERATION TOOLS FOR *IOCO*

- TVEDA (CNET - France Telecom)
  - derives TTCN tests from single process SDL specification
  - developed from practical experiences
  - implementation relation R1 ≈ ioco
- TGV (IRISA - Rennes)
  - derives tests in TTCN from LOTOS or SDL
  - uses test purposes to guide test derivation
  - implementation relation: unfair extension of ioco
- TestComposer (Verilog)
  - Combination of TVEDA and TGV in ObjectGeode
- TestGen (Stirling)
  - Test generation for hardware validation
- TorX (University of Twente, ESI)

A TEST TOOL : TORX

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



TORX TOOL ARCHITECTURE

specification text

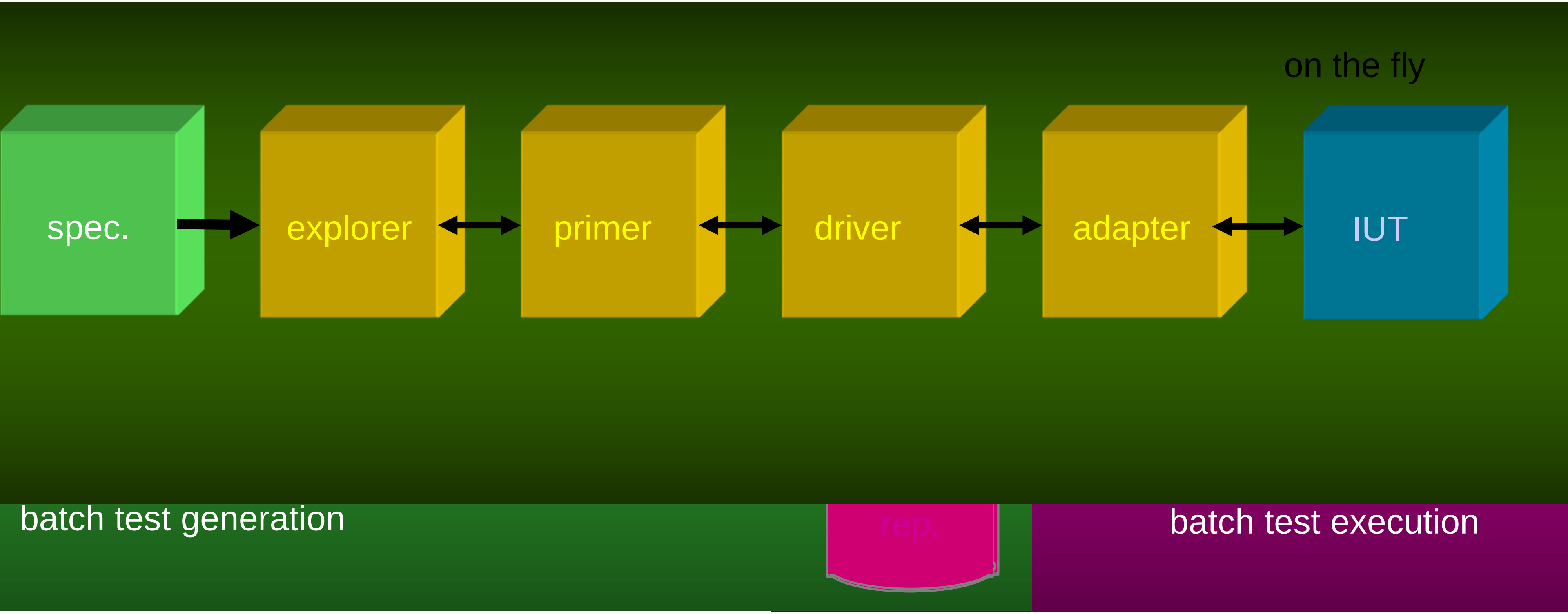
states transitions

abstract actions

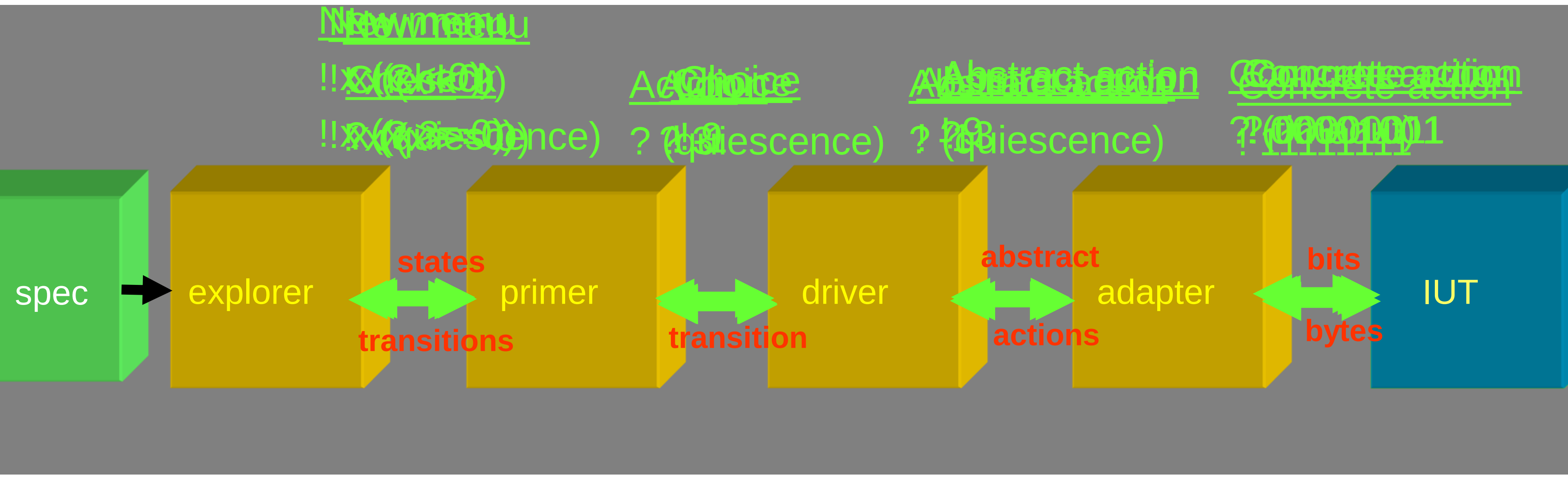
abstract actions

concrete actions

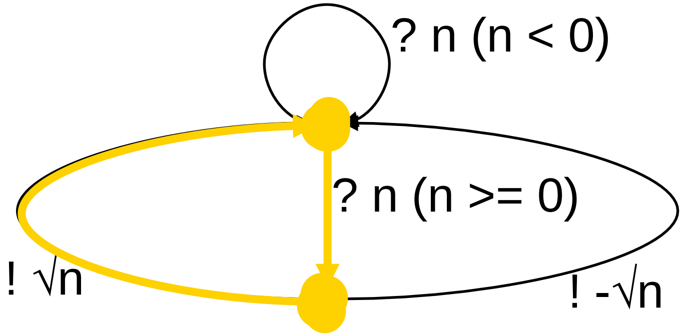
ON-THE-FLY ↔ BATCH TESTING



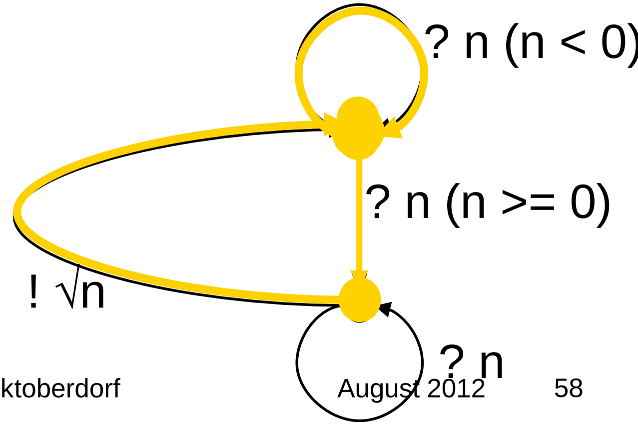
ON-THE-FLY TESTING



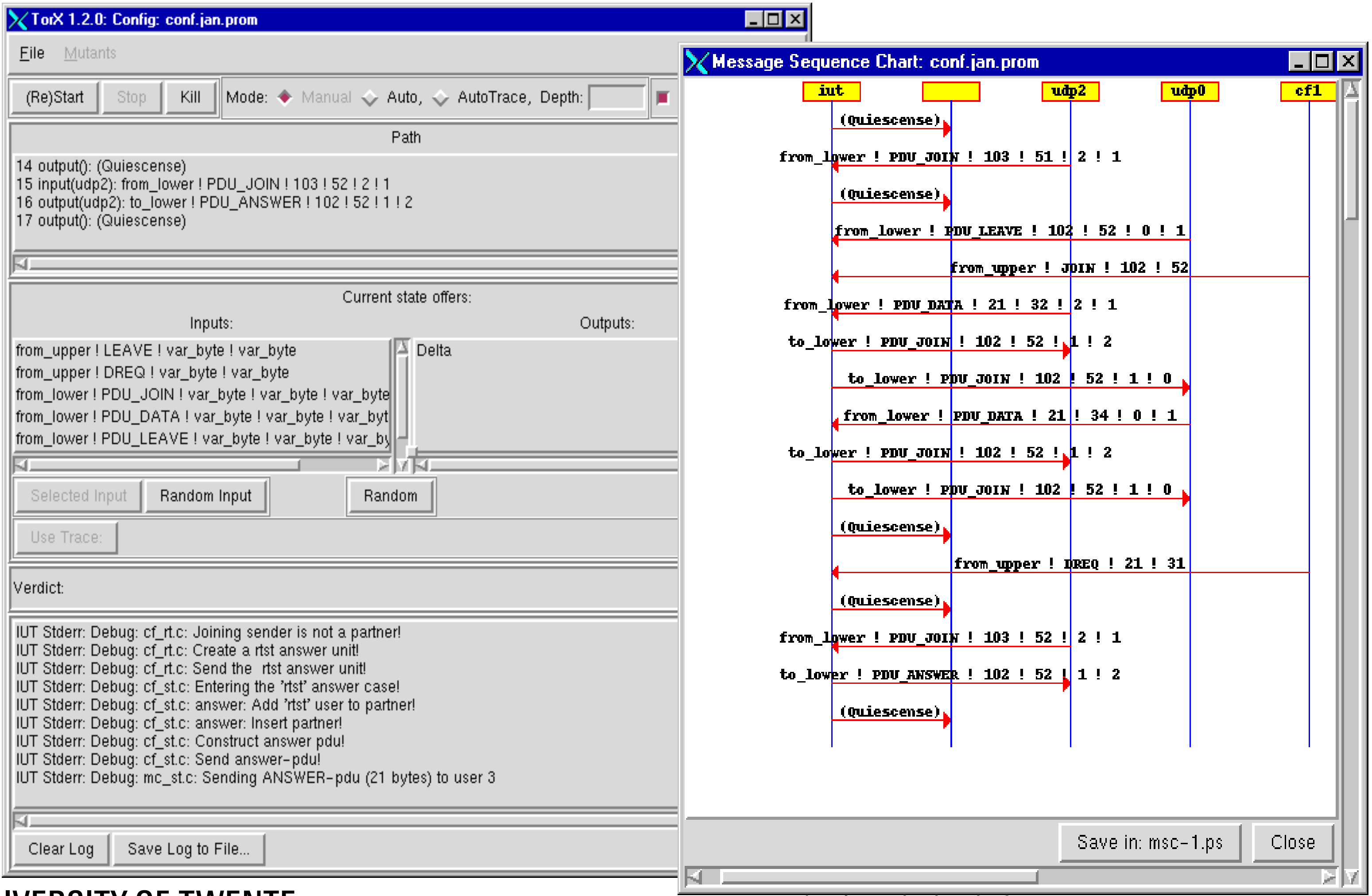
specification



implementation



TORX



```
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)
```

Inputs:

```
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
```

Outputs:

```
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)
```

SOME TORX CASE STUDIES

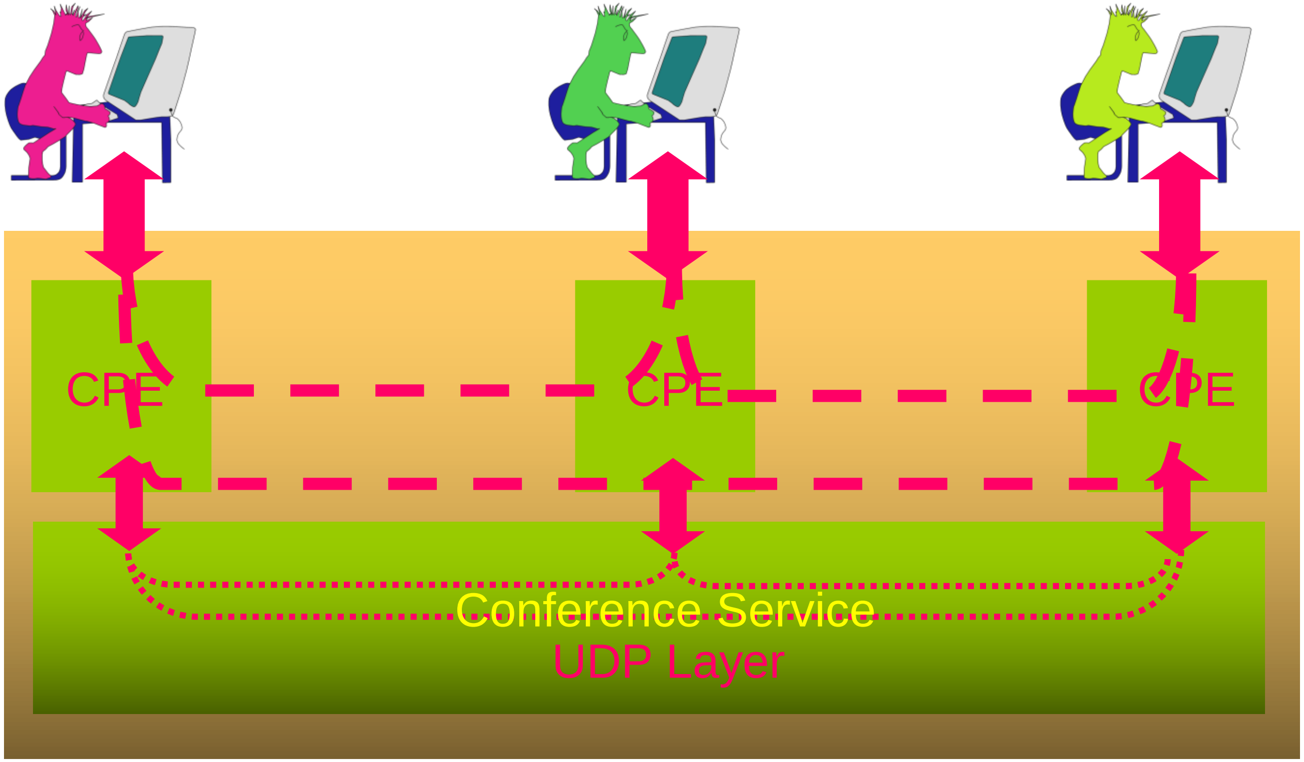
Conference Protocol	academic
EasyLink TV-VCR protocol	Philips
Cell Broadcast Centre component	CMG
Road Toll Payment Box protocol	Interpay
V5.1 Access Network protocol	Lucent
Easy Mail Notification	CMG
FTP Client	academic
“Oosterschelde” storm surge barrier-control	CMG
TANGRAM: testing VLSI lithography machine	ASML

THE CONFERENCE PROTOCOL EXPERIMENT

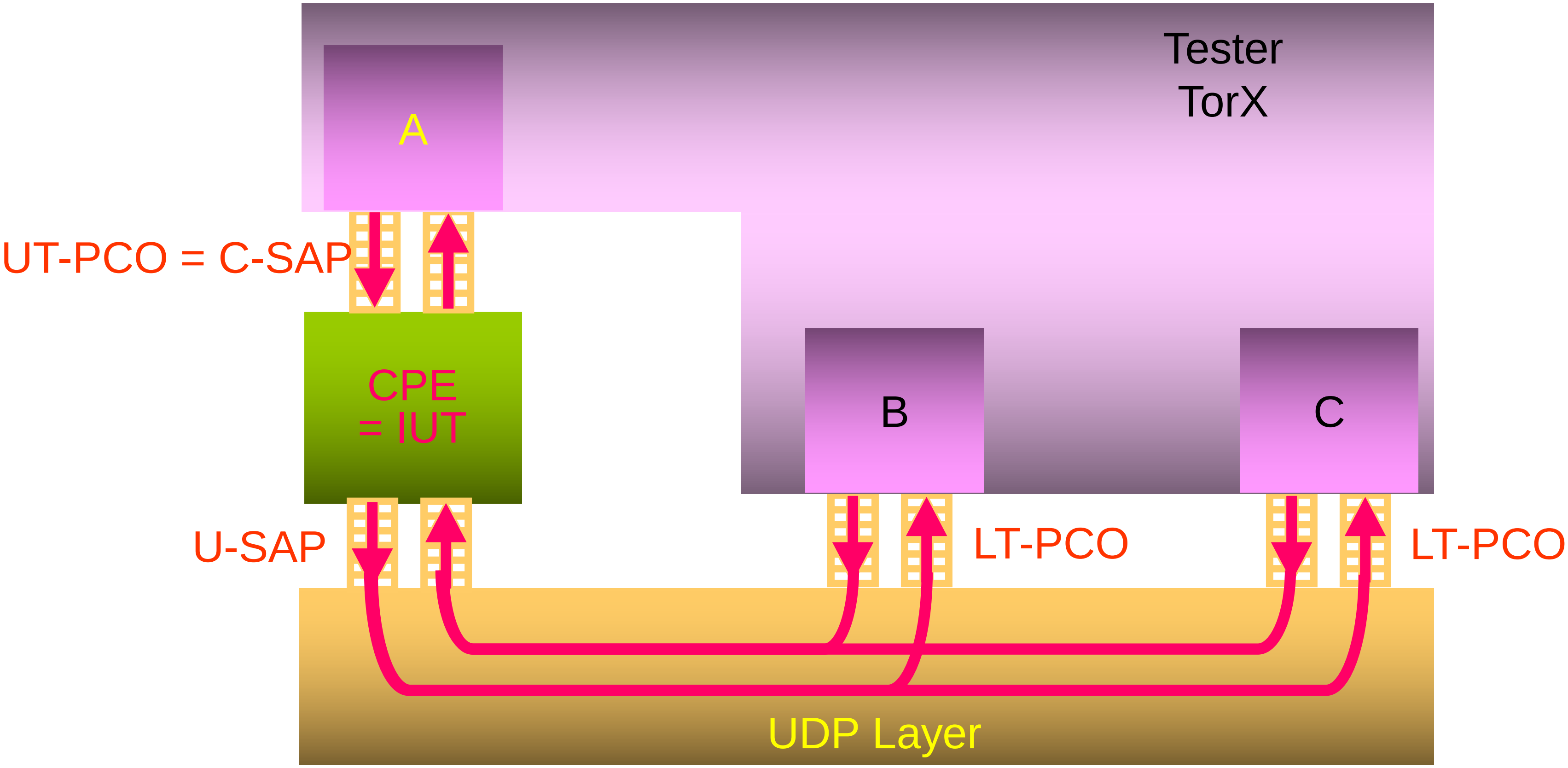
- Academic benchmarking experiment,
  initiated for test tool evaluation and comparison
- Based on really testing different implementations
- Simple, yet realistic protocol (chatbox service)
- Specifications in LOTOS, Promela, SDL, EFSM
- 28 different implementations in C
  - one of them (assumed-to-be) correct
  - others manually derived mutants
- http://fmt.cs.utwente.nl/ConfCase

THE CONFERENCE PROTOCOL

join
leave
send
receive



CONFERENCE PROTOCOL TEST ARCHITECTURE



THE CONFERENCE PROTOCOL EXPERIMENTS

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

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

- Tau Autolink - SDL : semi-automatic batch testing
- TGV - LOTOS : automatic batch testing with test purposes

  Lydie Du Bousquet et al.,
  Formal Test Automation: The Conference Protocol with TGV/TorX
  TestCom 2000, Ottawa.

- PHACT/Conformance KIT - EFSM : automatic batch testing

  Lex Heerink et al.,
  Formal Test Automation: The Conference Protocol with PHACT
  TestCom 2000, Ottawa.

CONFERENCE PROTOCOL RESULTS

Results:	TorX LOTOS	TorX Promela	PHACT EFSM	TGV LOTOS random	TGV LOTOS purposes
fail	25	25	21	25	24
pass	3	3	6	3	4
core dump	0	0	1	0	0
pass	000 444 666	000 444 666	000 444 666 289 293 398	000 444 666	000 444 666 332

CONFERENCE PROTOCOL ANALYSIS

- Mutants 444 and 666 react to PDU’s from non-existent partners:
  - no explicit reaction is specified for such PDU’s,

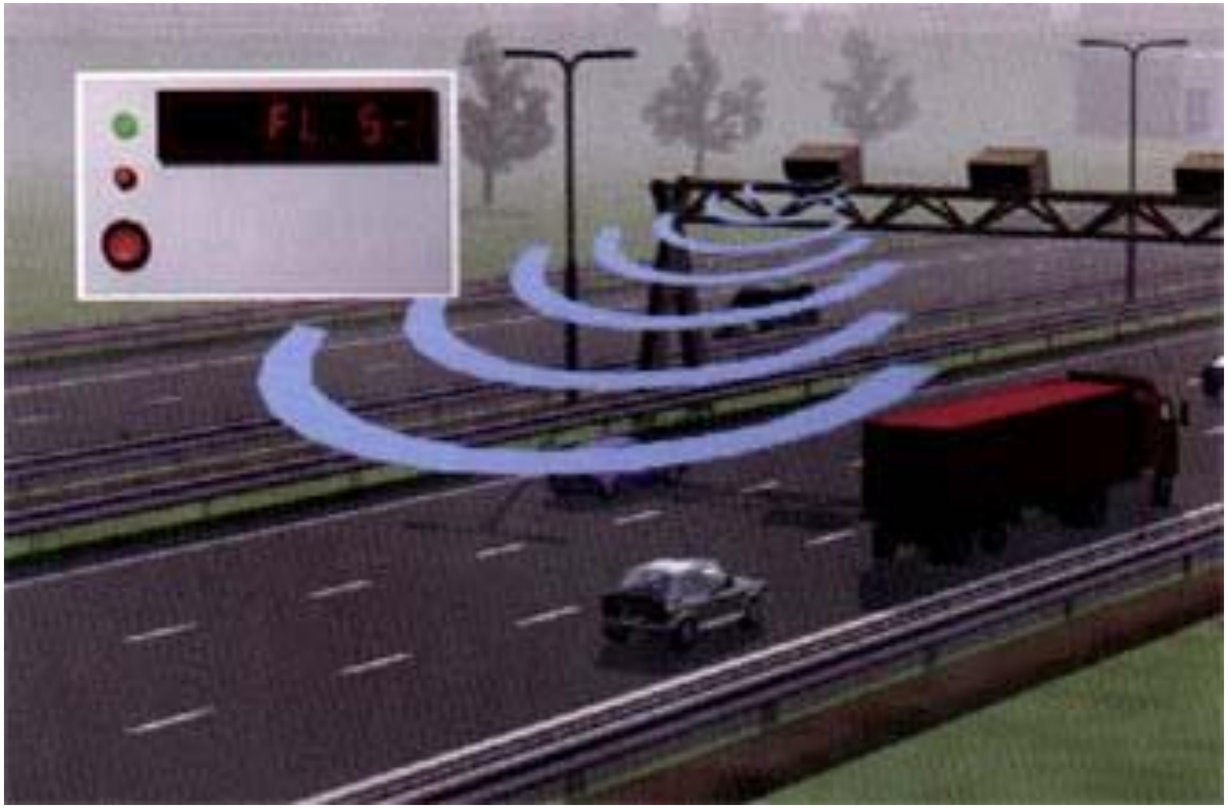
    so *ioco*-correct, and TorX does not test such behaviour
- So, for LOTOS/Promela with TGV/TorX:

  All *ioco*-erroneous implementations detected
- EFSM:
  - two “additional-state” errors not detected
  - one implicit-transition error not detected

CONFERENCE PROTOCOL ANALYSIS

- TorX statistics
  - all errors found after 2 - 498 test events
  - maximum length of tests : > 500,000 test events
- EFSM statistics
  - 82 test cases with “partitioned tour method” (= UIO)
  - length per test case : < 16 test events
- TGV with manual test purposes
  - ~ 20 test cases of various length
- TGV with random test purposes
  - ~ 200 test cases of 200 test events

INTERPAY HIGHWAY TOLLING SYSTEM

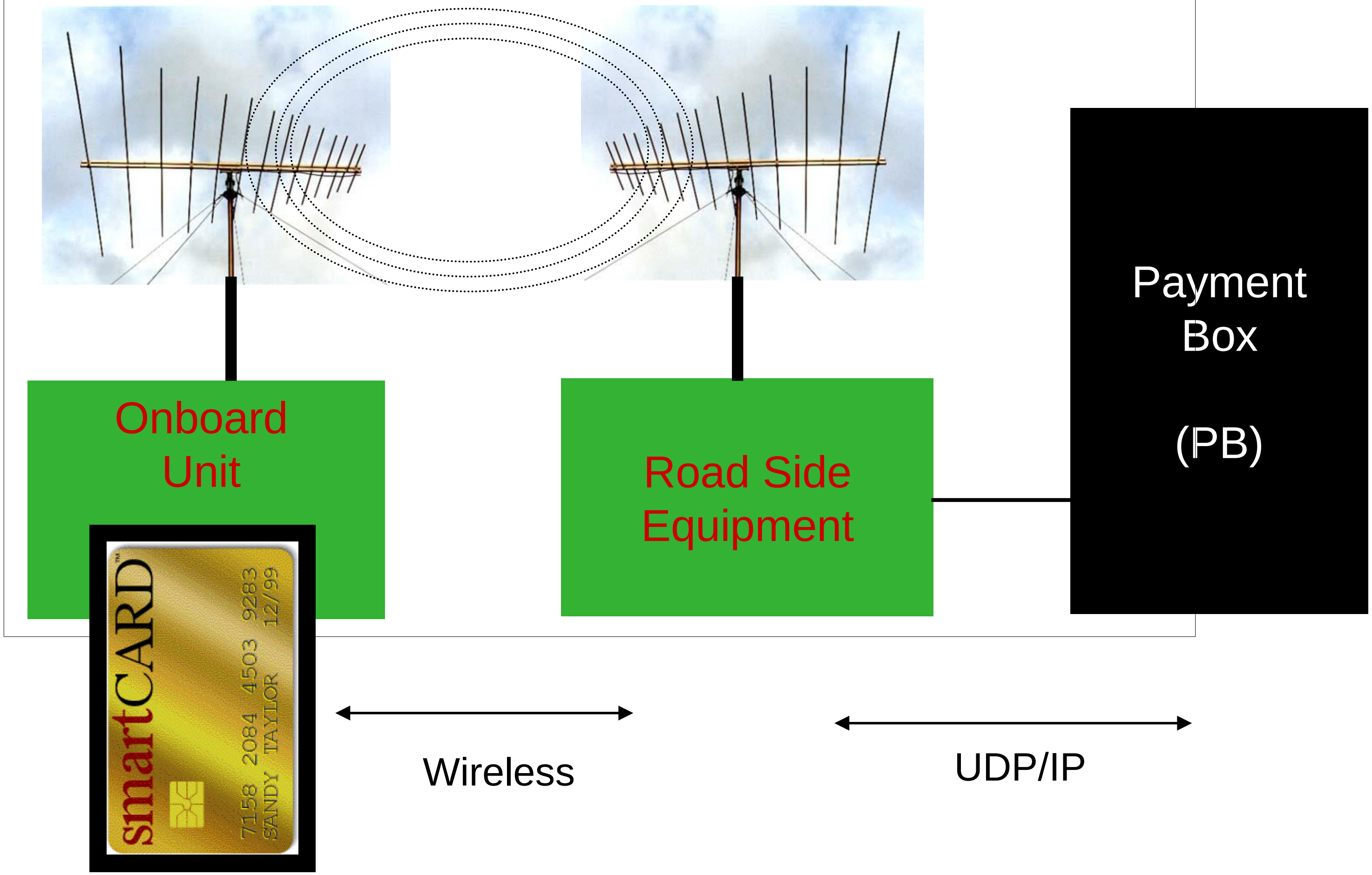


HIGHWAY TOLLING PROTOCOL

Characteristics :

- simple protocol
- parallelism: many cars at the same time
- encryption
- system passed traditional testing phase

HIGHWAY TOLLING SYSTEM



Onboard Unit

Road Side Equipment

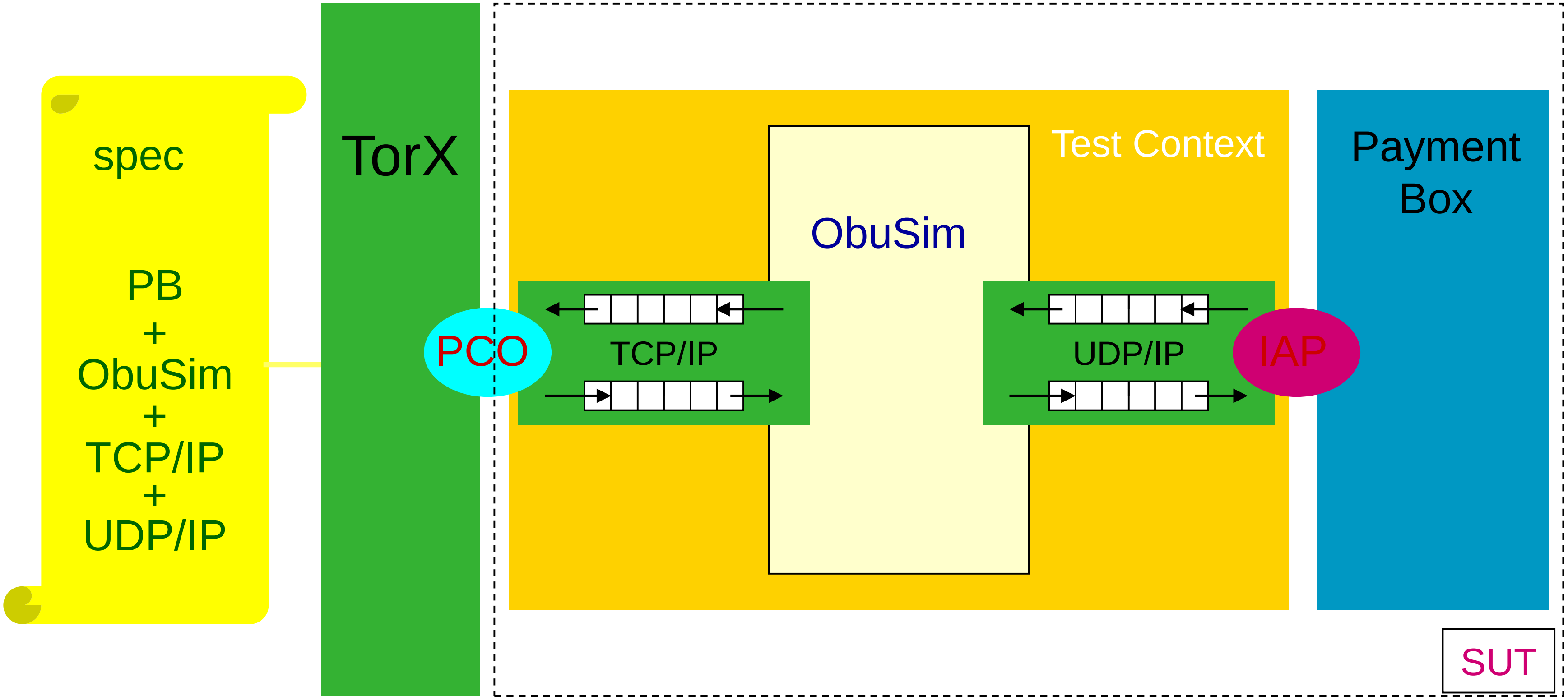
Payment Box

(PB)

Wireless

UDP/IP

HIGHWAY TOLLING: TEST ARCHITECTURE



HIGHWAY TOLLING: RESULTS

- Test results :
  - 1 serious error during validation   (design error)
  - 1 serious error during testing   (coding error)
- Automated testing :
  - beneficial: high volume and reliability
  - many and long tests executed   (> 50,000 test events)
  - very flexible: adaptation and many configurations
- Real-time :
  - interference computation time on-the-fly testing
  - interference quiescence and time-outs

STORM SURGE BARRIER CONTROL

Oosterschelde Stormvloedkering (OSVK)

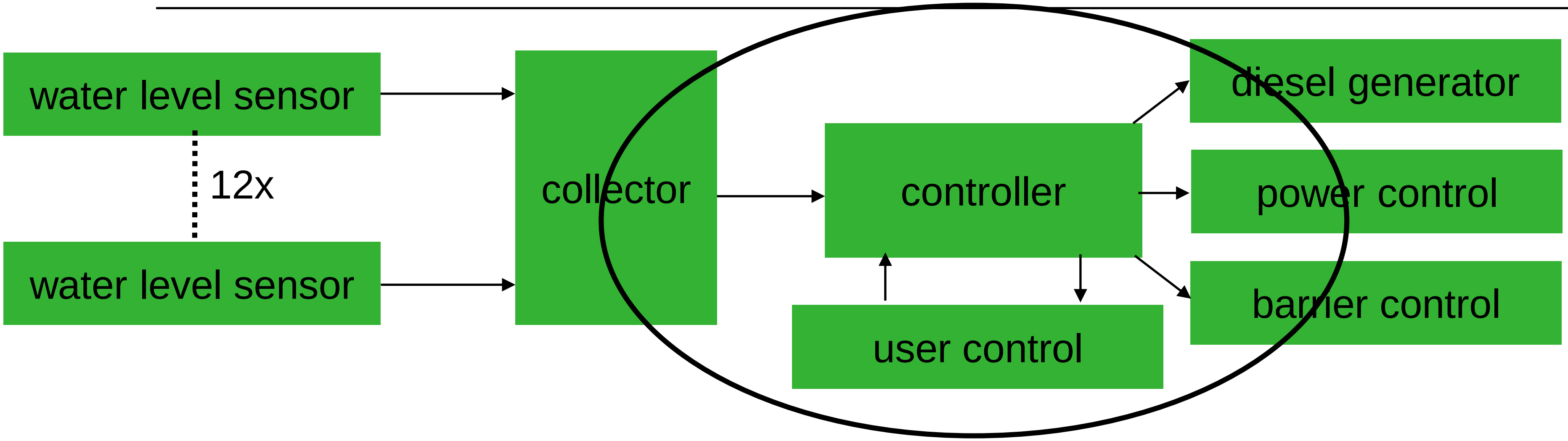
SVKO EMERGENCY CLOSING SYSTEM

- Collect water level sensor readings (12x, 10Hz)
- Calculate mean outer-water level and mean inner-water level
- Determine closing conditions

```
if (closing_condition)
{notify officials
 start diesel engines
 block manual control
 control local computers}
```

- Failure rate: 10^{-4}/closing event

TESTING SVKO



signal wire communication

- test controller (Unix port)
- many timed observations
    - shortest timed delay: 2 seconds
    - longest timed delay: 85 minutes

RESULTS

- real-time control systems can be tested with TorX-technology
  - addition of discrete real time
  - time stamped actions
- quiescence action is not used
  - time spectrum of 3 orders of magnitude
  - deterministic system
- adhoc implementation relation

RT TORX HACKS: APPROACH 1

Ignore RT functionality:

1. test pure functional behaviour
2. analyse timing requirements using TorX log files & assumed frequency of wire polling actions

RT TORX HACKS: APPROACH 2

Add timestamps to observations

1. adapter adds timestamps to observations when they are made and passed on to the driver
2. timestamps are used to analyse TorX log files

TIMING ERROR LOGGING

```
int time,newtime;
...
input?value,time               // input: stimulus
...                            // variable time is set here
output!param,&newtime          // output: observation
         // & is extension to promela:
if         // variable newtime is set here
:: (newtime==time+60)          // expected delay of 60 is
    print(“OK”,...)            // checked & logged
:: (newtime!=time+60)
    print(“NOK”,...)
fi;
```

RT TORX HACKS: APPROACH 3

Add timestamps to stimuli & observations

1. adapter add timestamps to observations when they are made and passed on to the driver
2. adapter adds timestamps to stimuli when they are applied and returned to the driver
3. analysis:
    a. timing error logging: observed errors are written to TorX log file
    b. timing error failure: observed errors cause fail verdict of test case

TIMING ERROR FAILURE

```
int time;
...          // input: stimulus
input?value,time      // variable time is set here
...
if              // output: observation
:: output!param,(time+59)    // wait 60 (-1)
:: output!param,(time+60)     // wait 60
:: output!param,(time+61)     // wait 60 (+1)
fi            // if observation is not made
                                   // after approx. 60 units,
               // quiescence will be
             // observed
```

CONTENTS

CONTENTS

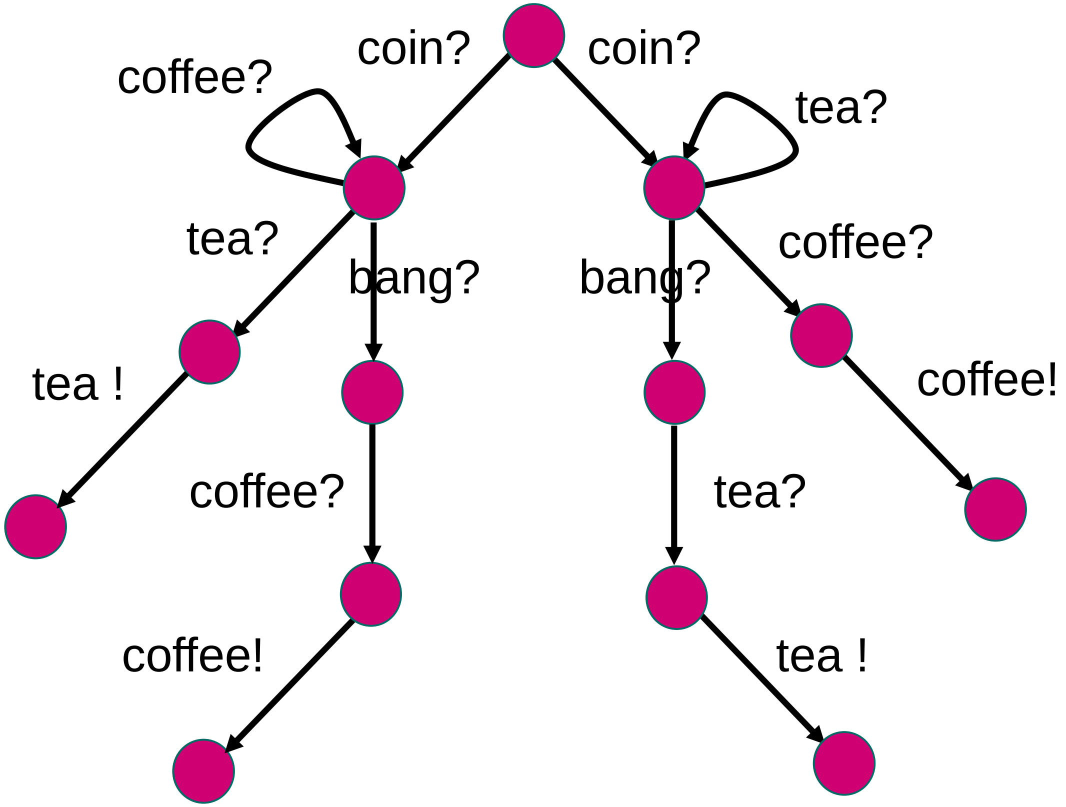
REAL-TIME TESTING AND I/O SYSTEMS

- can the notion of repetitive quiescence be combined with real-time testing?
- is there a well-defined and useful conformance relation that allows sound and (relative) complete test derivation?
- can the TorX test tool be adapted to support Real-timed conformance testing?

WITH REAL-TIME DO WE STILL NEED QUIESCENCE?

Yes!

the example processes should also be distinct in a real-time context



REAL-TIME AND QUIESCENCE

- s is *quiescent* iff:

  for no output action a and delay d: $s \overset{a(d)}{\not\Rightarrow}$

- special transitions:

  $s \xrightarrow{\delta} s$ for every quiescent system state s

- testers observing quiescence take time:

  $Test_M$: set of test processes having only $\delta_{(M)}$-actions to observe quiescence

- assume that implementations are M-quiescent:

  for all reachable states s and s':

  if $s \overset{\varepsilon(M)}{\Longrightarrow} s'$ then s' is quiescent

REAL-TIME AND QUIESCENCE

$$i \leq^{M}_{tiorf} s \;\Leftrightarrow\; \forall T \in Test_{M}:$$

$$CTraces_{\delta}(i||T) \subseteq CTraces_{\delta}(s||T)$$

$$\Leftrightarrow\; \forall \sigma \in (L \cup \{\delta(M)\})^{*}:$$

$$out_{i,M}(\sigma) \subseteq out_{s,M}(\sigma)$$

$$i \; tioco_{M} \; s \;\Leftrightarrow\; \forall \sigma \in Traces_{\delta(M)}(s):$$

$$out_{i,M}(\sigma) \subseteq out_{s,M}(\sigma)$$

PROPERTIES

1. for all $M_1 \leq M_2$:

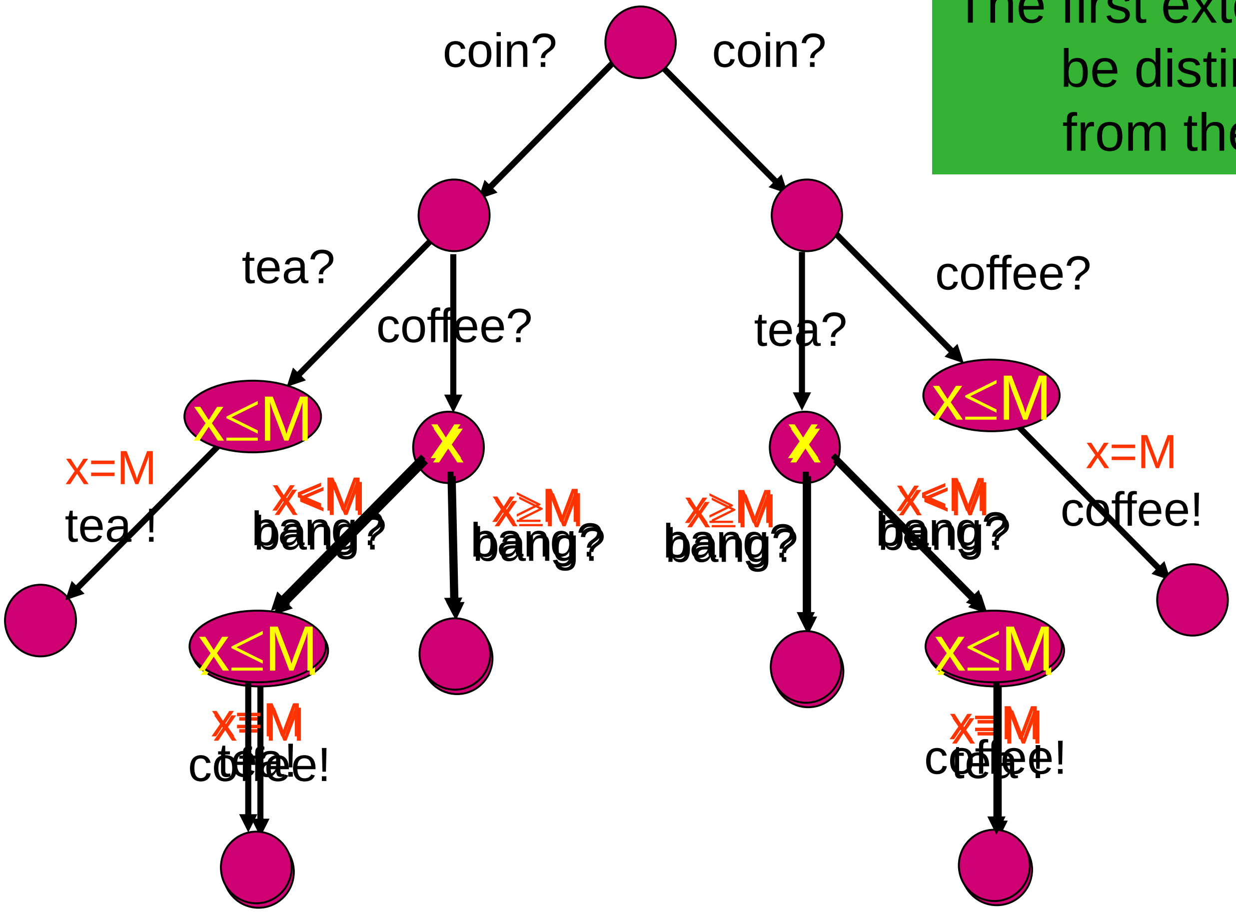
   $$i \leq_{tiorf}^{M_1} s \text{ implies } i \leq_{tiorf}^{M_2} s$$

2. for all time-independent i,s and $M_1,M_2>0$

   $$i \leq_{tiorf}^{M_1} s \quad \textit{iff} \quad i \leq_{tiorf} s \quad \textit{iff} \quad i \leq_{iorf}^{M_2} s$$

A LIMITATION

The first extention cannot be distinguished from the second

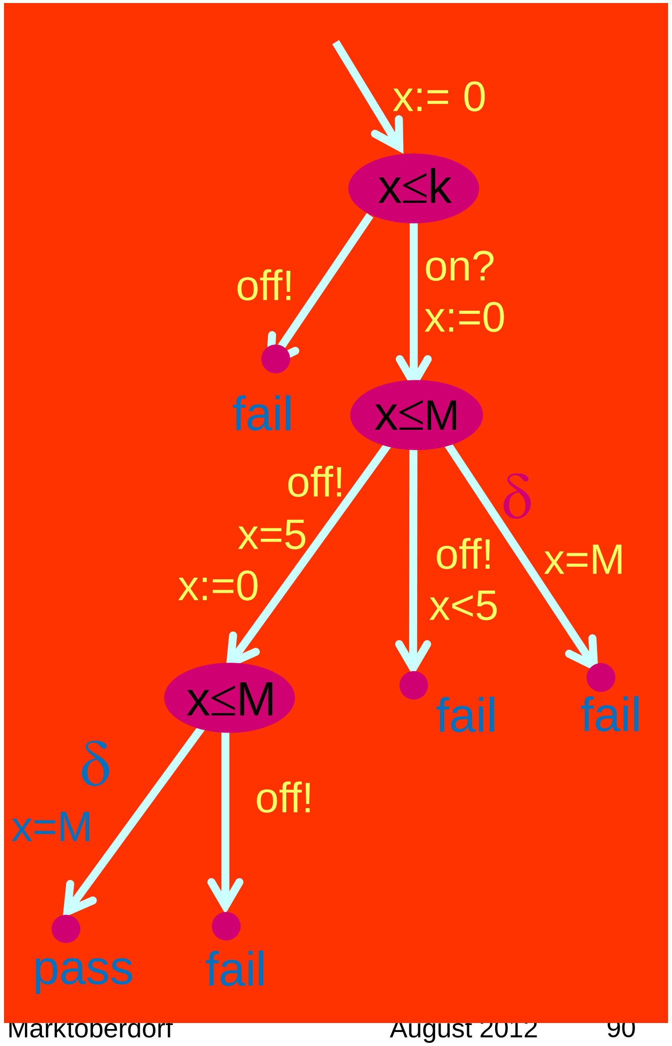


ANNOTATED REAL-TIME TEST CASES

Test case *t* ∈ *TTA*

TTA – Timed Test Automata :

- tree-structured
- finite, deterministic
- final states pass and fail
- from each state ≠ pass, fail
  - choose an input *i?* and a time *k* and wait for the time *k* accepting all outputs *o!* and after *k* time units provide input *i?*
  - or wait for time *M* accepting all outputs *o!* and δ



TIMED TEST GENERATION PROTO-ALGORITHM

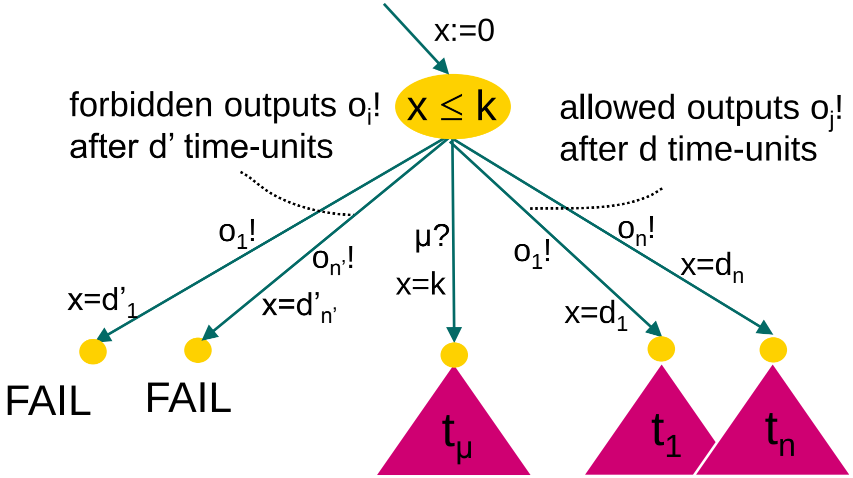
To generate a test case $t(S)$ from a timed transition system specification with S set of states (initially $S = \{s_0\}$)

Apply the following steps recursively, non-deterministically

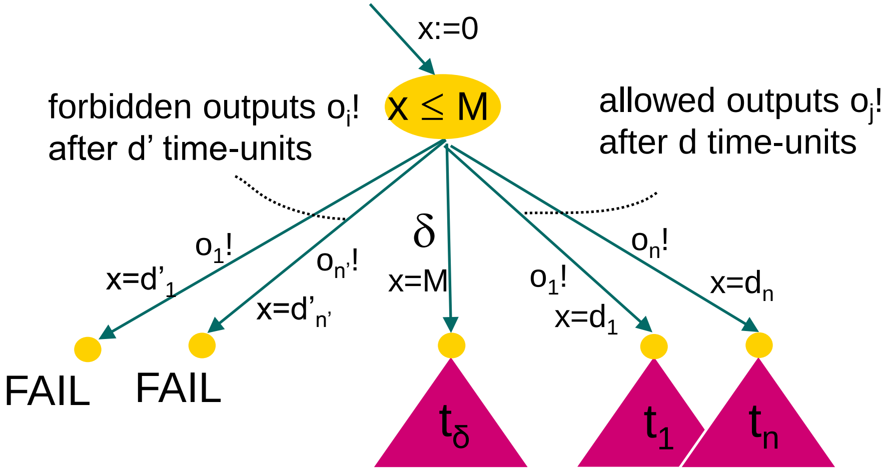
1. end test case

● PASS

2. choose $k \in (0, M)$ and input μ



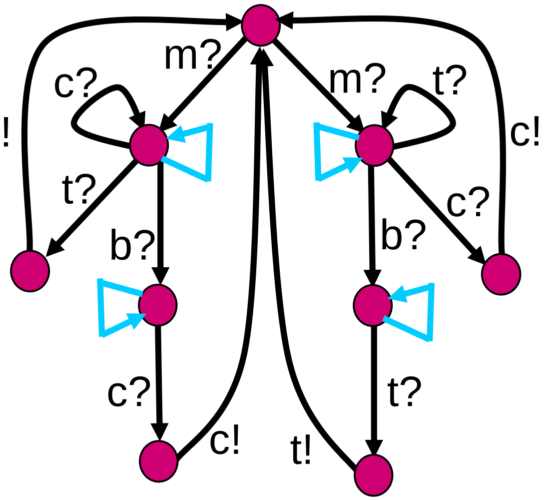
3. wait for observing possible output



Example

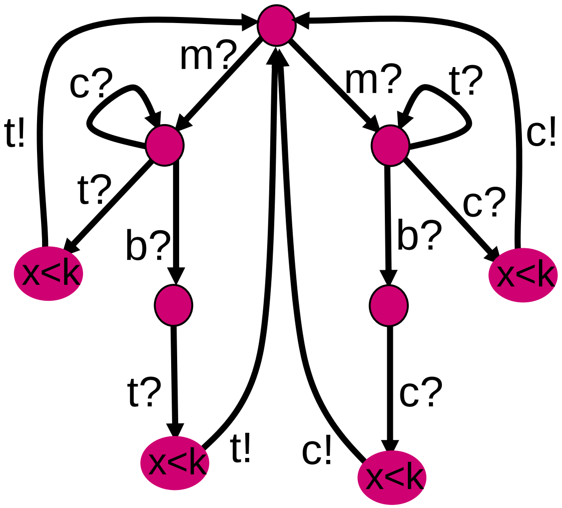
spec:

δ

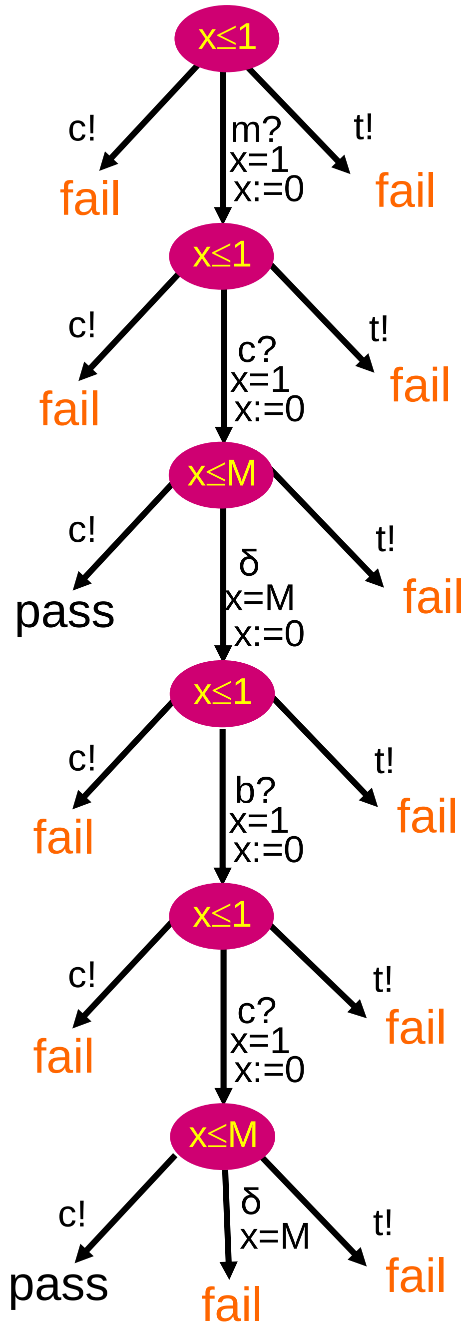


impl:

M=k



test:



SOUNDNESS & COMPLETENESS

- the non-timed generation algorithm can be s
  only sound real-time test cases
- test generation is complete

  for every erroneous trace it can generate a

  test that exposes it
- test generation is not limit complete

  because of continuous time there ar uncountably many timed error traces and only countably many te t are generated by repeated runs
- test generation is almost limit complete

  repeated test geration runs will eventually generate a test case that will expose one of the non-spurious errors of a non-conforming implementation

non-spurious errors
=
errors with a positive probability of occurring

CONTENTS

CONTENTS

1. Introduction control-oriented testing
2. Input-output conformance testing
3. Real-time conformance testing
4. Test coverage measures

COVERAGE: MOTIVATION

- Testing is inherently incomplete
  - Test selection is crucial

- Coverage metrics
  - Quantitative evaluation of test suite
  - Count how much of specification/implementation has been examined

- Examples:
  - Transparent box (implementation coverage):
    Statement, path, condition coverage
  - Black box (specification coverage)
    State, transition coverage

TRADITIONAL COVERAGE MEASURES

Traditional measures are:

- based on syntactic model features

  states, transitions, statements, tests

- uniform

  all system parts treated as bequally important

Disadvantages:

- replacing the spec by an equivalent one yields different coverage

  needs a semantic *approach*

- some bugs are more important than others;

  test crucial behaviour first and better

SEMANTIC APPROACH

- Considers black box coverage

  similar ideas could apply to white box coverage

- Semantically equivalent specs yield same coverage
- Risk-based

  more important bugs/system parts

  → *higher contribution to coverage*

- Allows for optimization

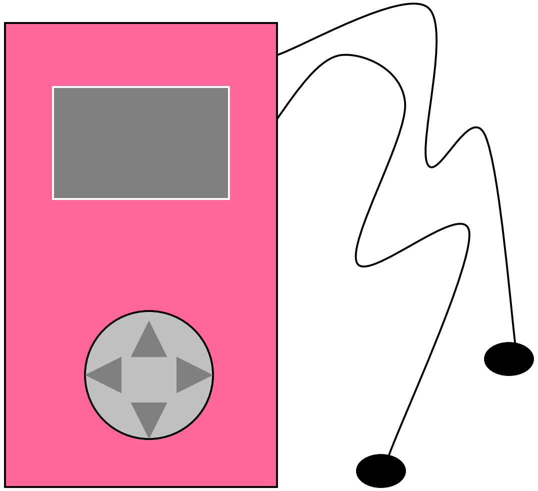
  cheapest test suite with 90% coverage; or

  maximal coverage within cost budget

FAULT MODELS

- f: *Observation* → $R^{\geq 0}$
  - $f(\sigma) = 0$: correct behaviour
  - $f(\sigma) > 0$: incorrect behaviour

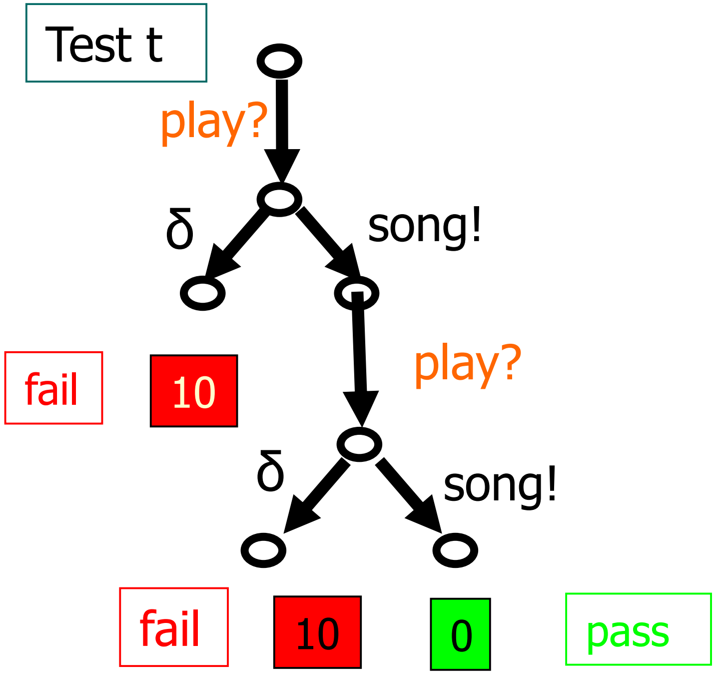
    : $f(\sigma)$ severity
  - $0 < \Sigma_\sigma f(\sigma) < \infty$
- Observations are traces
  - *Observations* = L^*
  - $L = (L_I, L_U)$
- How to obtain f?
  - E.g. via fault automaton



$f: L^* \rightarrow R^{\geq 0}$

f(play? song!)	= 0	correct
f(play? silence!)	= 10	incorrect
f(song!)	= 3	incorrect

EXAMPLE TEST CASE

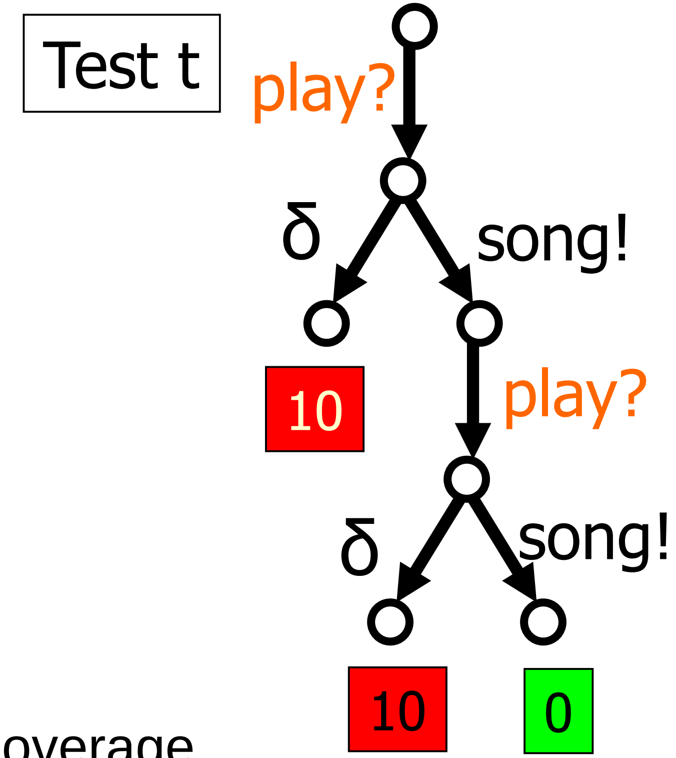


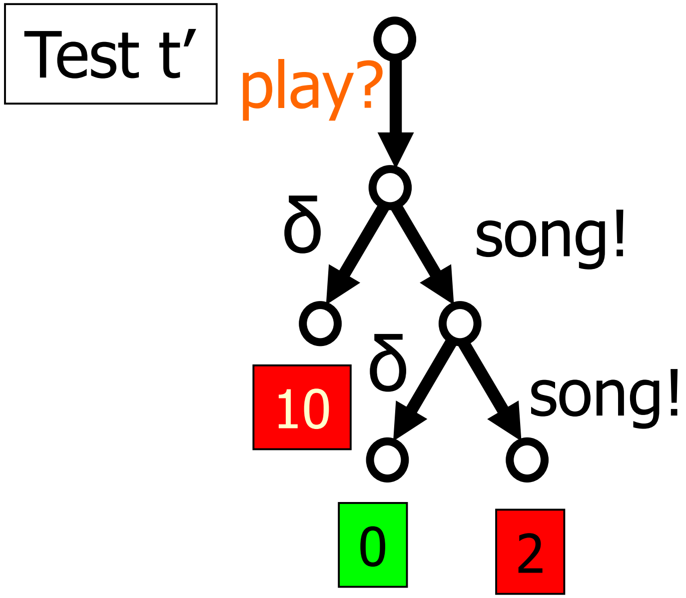
- $f: L^* \rightarrow R$
  - f(play? song!) = 0
  - f(play? δ) = 10
  - f(play? song! play? δ) = 10
  - f(song!) = 3
- $\sum_\sigma f(\sigma) = 100$ (assumption)
- Absolute Coverage abscov(f,t)
  - sum the error weights
  - 10 + 10 + 0 = 20
- Relative Coverage

$$\frac{abscov(f,t)}{totcov(f)} = \frac{20}{100}$$

should be ≠ 0, ≠∞

EXAMPLE TEST SUITE





Absolute Coverage

- count each trace once !
- 10 + 10 + 0 + 0 + 2 = 22

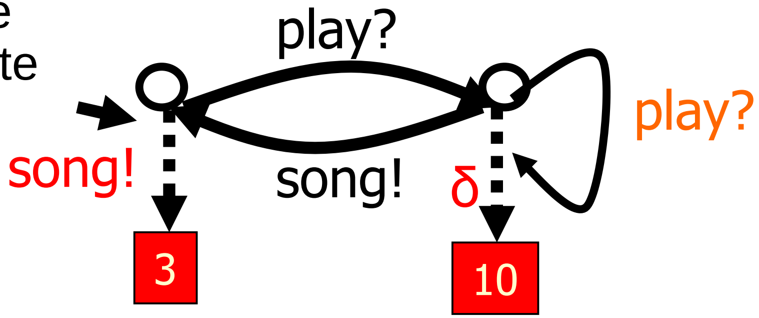
Relative Coverage

$$\frac{abscov(f,t)}{totcov(f)} = \frac{22}{100} = 22\%$$

FAULT SPECIFICATIONS

fault model

- $f(\sigma) = 0$ if σ trace of automaton
- $f(\sigma) = 3 \cdot \alpha^{|\sigma|-1}$ if σ ends in 3-state
- $f(\sigma) = 10 \cdot \alpha^{|\sigma|-1}$ if σ ends in 10-state



infinite total coverage !!

- $\sum_{\sigma} f(\sigma) = 3 + 10 + 3 + 10 + \ldots = \infty$

Solution 1: restrict to traces of lenght k

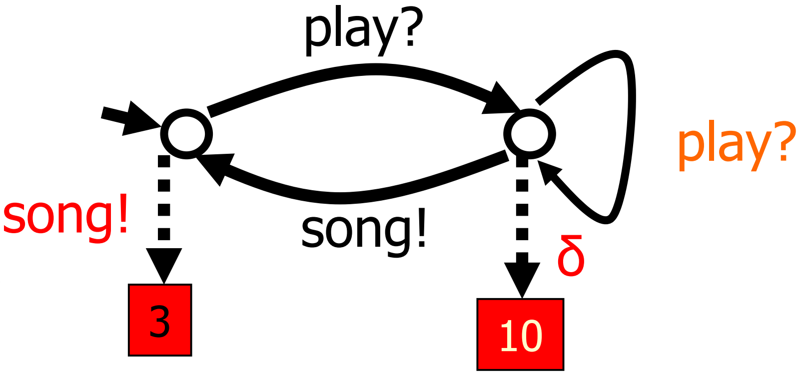
- Omit here, works as solution 2, less efficient, more boring

Solution 2: discounting

- errors in short traces are worse
- Lower the weight proportional to length

Use your favorite Formalism, e.g. UML state charts, LOTOS, etc

FAULT SPECIFICATIONS



fault model

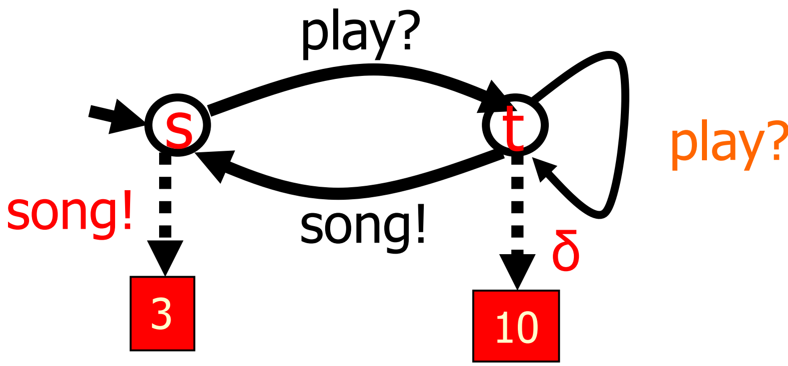
- $f(\sigma) = 0$ if σ automaton trace
- $f(\sigma) = 3 \cdot \alpha^{|\sigma|-1}$ if σ end in 3-state
- $f(\sigma) = 10 \cdot \alpha^{|\sigma|-1}$ if σ ends in 10-state

Example

- f(play?) = 0
- f(play? δ) = 10 $\cdot\ \alpha$
- f(play? song! song!) = 3 $\cdot\ \alpha^{2}$
- ….

- $\alpha < 1/\text{out(spec)} = 1/2$
- α can vary per transition
- tune α

FAULT SPECIFICATIONS



Total coverage becomes fcomputable:

$\mathrm{tc}(s) = 3 + \alpha\, \mathrm{tc}(t)$
$\mathrm{tc}(t) = 10 + \alpha\, \mathrm{tc}(t) + \alpha\, \mathrm{tc}(s)$

$\mathrm{tc}(x) = \mathrm{wgt}(x) + \alpha \sum_{y:\, \mathrm{succ}(x)} \mathrm{tc}(y)$

Solve linear equations

$\mathrm{tc} = \mathrm{wgt}\,(I - \alpha A)^{-1}$

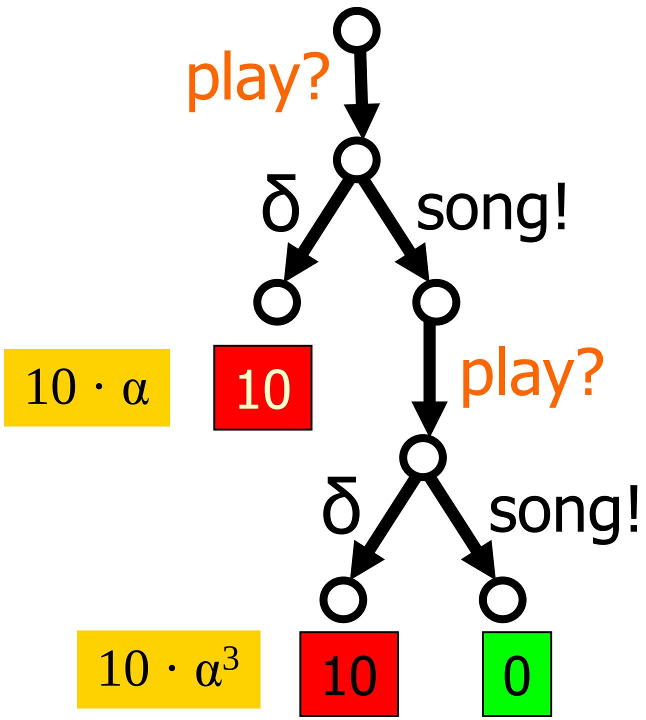
with A adjacency matrix

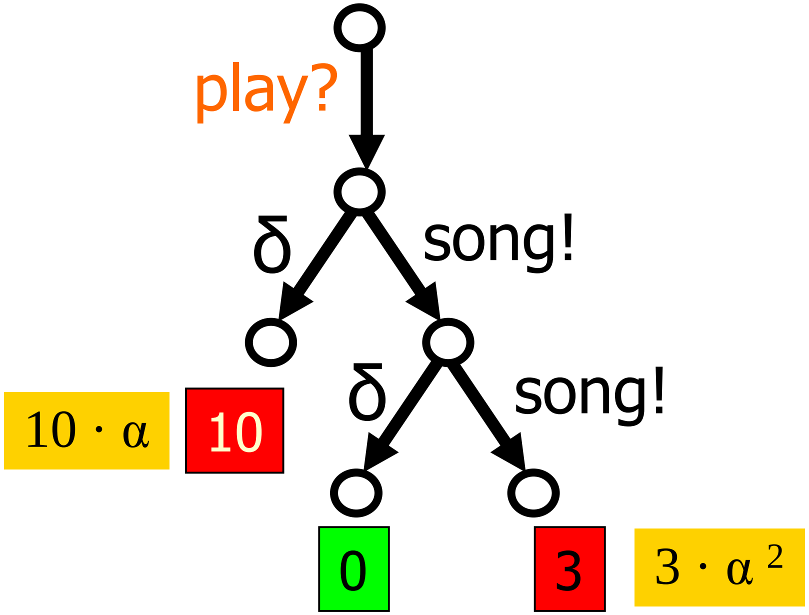
$$\mathrm{tc}(s) = \frac{10 + 7\alpha}{1 - \alpha - \alpha^2}$$

Relative Coverage

$$\frac{\mathrm{abscov}(f,t)}{\mathrm{totcov}(f)}$$

TEST SUITE COVERAGE





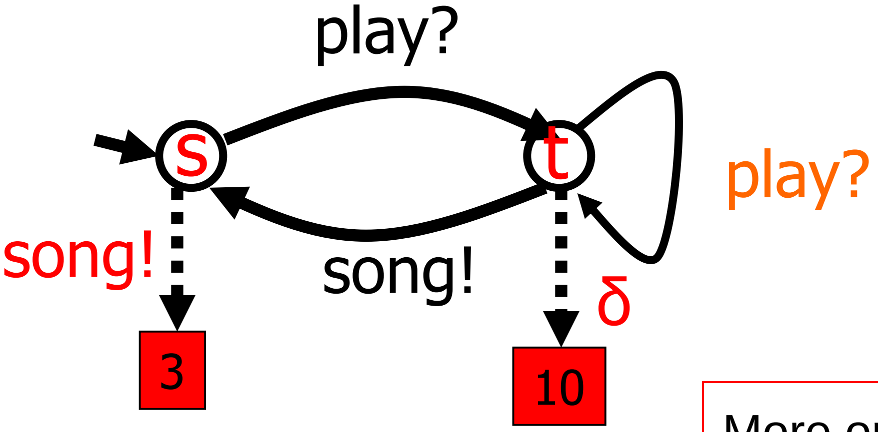
Absolute test suite coverage

- count each trace once!
- merge test cases first

Relative test suite coverage

$$\frac{\text{abscov}(f,t)}{\text{totcov}(f)} = \frac{10\alpha + 3\alpha^2 + 10\alpha^3}{\dfrac{10 + 7\alpha}{1 - \alpha - \alpha^2}}$$

OPTIMIZATION



Find best test case of lenght *n*

$v_1(s) = 3$
$v_1(t) = 10$
$v_{k+1}(s) = \max(3, \alpha\, v_k(t))$
$v_{k+1}(t) = \max(10 + \alpha\, v_k(s), \alpha\, v_k(t))$

Complexity: O(*n* #transitions in spec)

More optimizations:

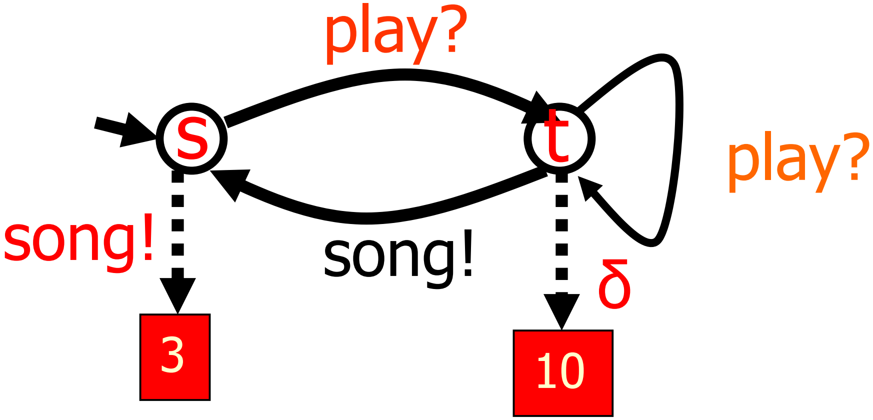
- Test suite of k tests & lenght n;
- Best test case in budget;
- Add costs
- ....

PROPERTIES

Framework for black box coverage

- robustness
  - small changes in weight yield small changes in coverage
  - relcov(s) continuous
- tunable (calibration)
  - change α : get as much total coverage as desired

CALIBRATION



α small → present is important, future unimportant

→ few small test cases with high (>0.999) coverage

tune α → make tests with length >*k* important, i.e
make cov(T_k, f) as small as desired.

→ $\alpha(s) = 1/n(s) - \varepsilon$ n(s) =outinf(s)

→ $\lim_{\varepsilon \to 0} \mathrm{cov}(T_k,f_\alpha) = 0$ for all k

CONCLUSIONS

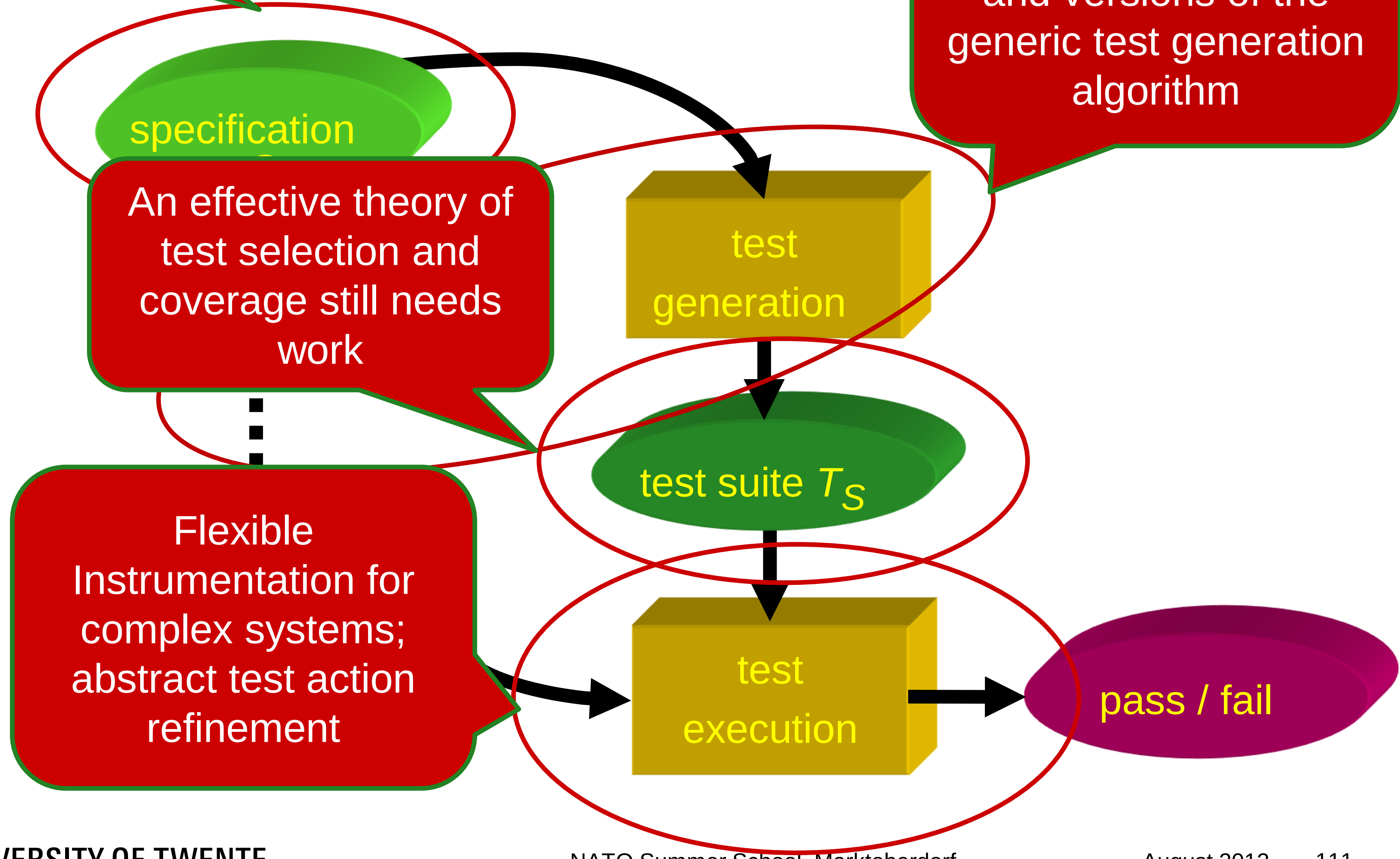
CONCLUSIONS

- model-based testing offers theory and tools for (real-time) conformance testing, in particular:
  - test generation, execution & evaluation
  - coverage analysis
- ioco-theory, TorX and related tools have been evaluated against many industrial cases
  - on-the-fly application very productive
  - good coverage with random test execution
- current theory is mostly control-oriented
  - OK for classical embedded applications
  - Is being extended to cope with data-intensive systems

URE REVISITED

A major problem getting good models

A fairly robust understanding of conformance relations and versions of the generic test generation algorithm



An effective theory of test selection and coverage still needs work

Flexible Instrumentation for complex systems; abstract test action refinement

CURRENT DEVELOPMENTS

- Methods for model inference
  - Incremental model learning
  - Process mining
  - Test-based modelling
- Stochastic approaches
  - Importance of stochastic features
  - Approach to coverage metrics
  - Source of (highly) abstract models

CURRENT DEVELOPMENTS

- Rigorous integration of control and data-oriented testing
  - Symbolic approaches
  - Deeper integration of methods for test data selection
  - Leaving transparent data transfer paradigm (ubiquitous/wireless/smart dust networking)
- Engineering tool kit for instrumentation
  - Mapping abstract action level to machine (inter)action level
  - Use of compilation techniques
  - Use of solutions for reconfigurable HW/SW platforms

RESOURCES

- http://fmt.cs.utwente.nl/tools/torx/introduction.html
- http://www.testingworld.org/
- http://www.laquso.com/knowledge/toolstable.php
- http://www.irisa.fr/vertecs/
- http://www.uppaal.com/