



Model Based Testing

An Attempt to Combine
Provable Soundness
& Effective Automation
& Industrial Applicability

Jan Tretmans

Radboud University
Nijmegen

Embedded Systems Institute
Eindhoven

jan.tretmans@esi.nl

1

Model Based Testing Contents

- ☞ Introduction
 - ◆ Testing
 - ◆ Model based testing
- ☞ Theory: formal model based testing
 - ◆ Testing as a decision procedure
 - ◆ Test assumption, implementation relation, tests
- ☞ Theory: testing with transition systems
 - ◆ Implementation relation **ioco**
 - ◆ Test generation for **ioco**
- ☞ A Tool
 - ◆ TorX
- ☞ Applications
 - ◆ Lessons
- ☞ Conclusions, Perspectives

Model Based Testing

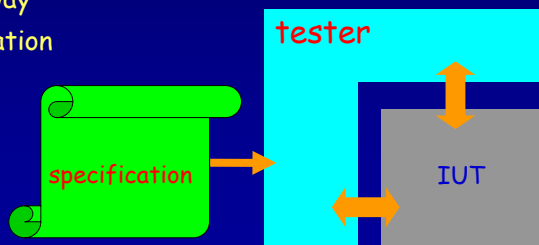
Contents

- ☞ Introduction
 - ◆ Testing
 - ◆ Model based testing
- ☞ Theory: formal model based testing
 - ◆ Testing as a decision procedure
 - ◆ Test assumption, implementation relation, tests
- ☞ Theory: testing with transition systems
 - ◆ Implementation relation *ioco*
 - ◆ Test generation for *ioco*
- ☞ A Tool
 - ◆ TorX
- ☞ Applications
 - ◆ Lessons
- ☞ Conclusions, Perspectives

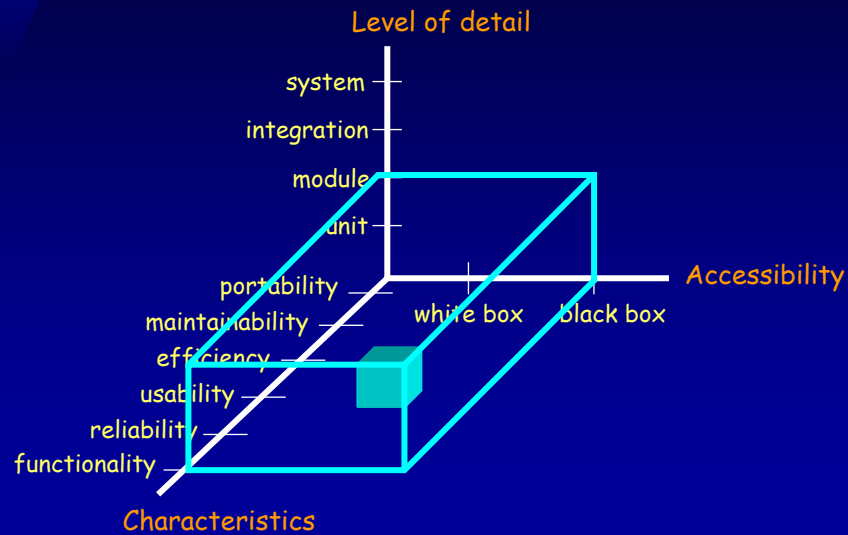
Testing

Testing:

checking or measuring some quality characteristics of an executing object by performing experiments in a controlled way w.r.t. a specification



Types of Testing



© Jan Tretmans

5

Towards Model Based Testing

- ☞ **Increase in complexity**, and quest for higher quality software
 - ◆ testing effort grows exponentially with complexity
 - ◆ testing cannot keep pace with development
- ☞ **More abstraction**
 - ◆ less detail
 - ◆ model based development
- ☞ **Checking quality**
 - ◆ practice: testing - ad hoc, too late, expensive, lot of time
 - ◆ research: formal verification - proofs, model checking,
with disappointing practical impact

© Jan Tretmans

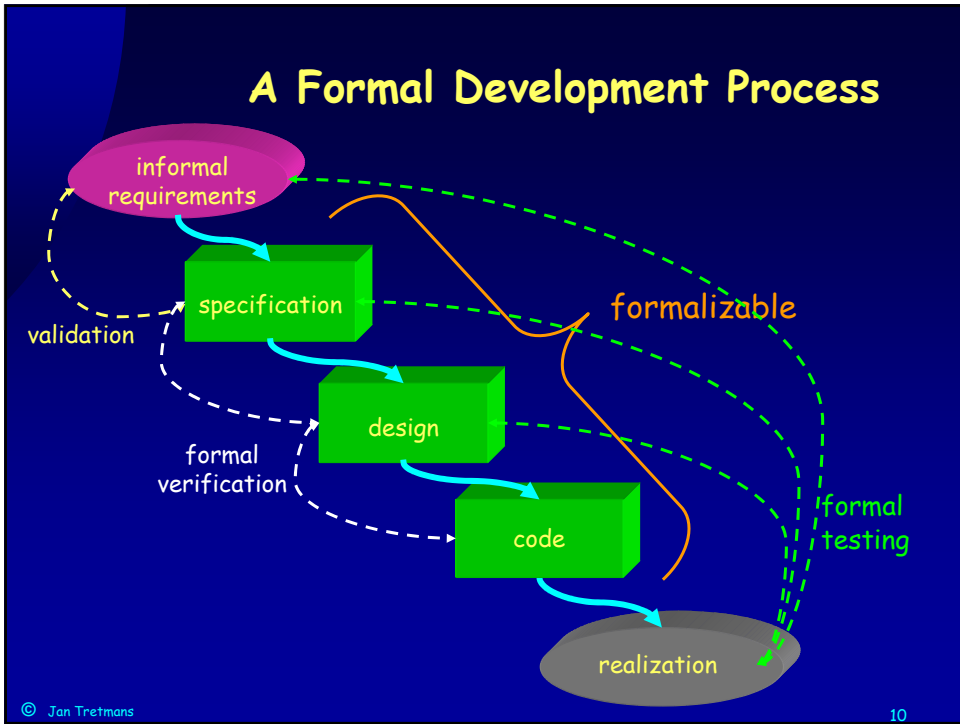
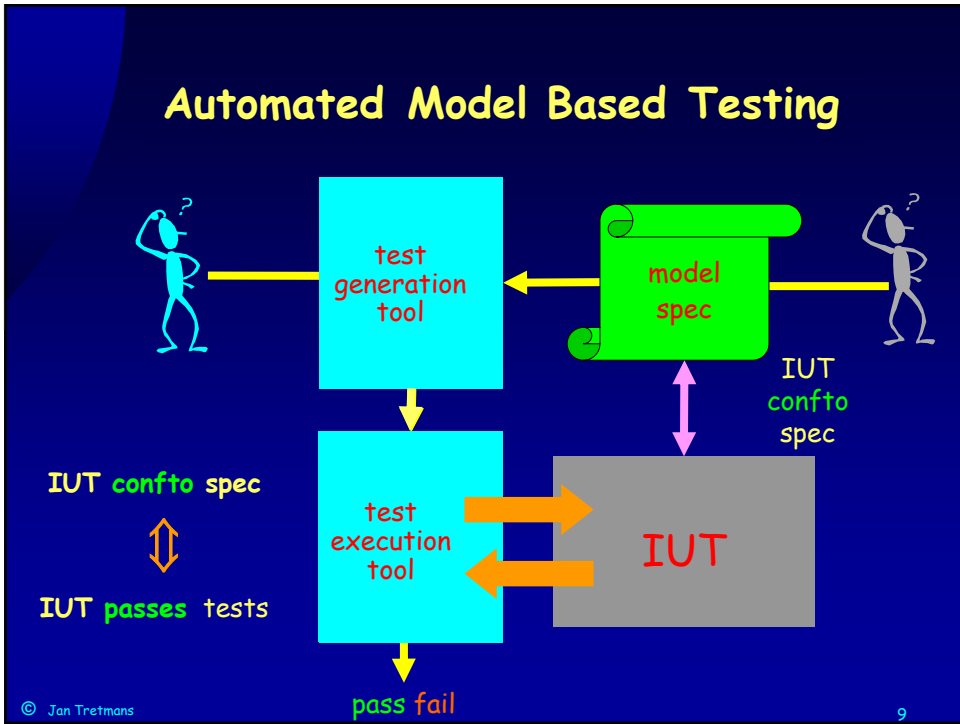
6

Towards Model Based Testing

- ☞ **Model based testing** has potential to combine
 - ◆ practice - testing
 - ◆ theory - formal methods
- ☞ **Model Based Testing :**
 - ◆ testing with respect to a (formal) model / specification state model, pre/post, CSP, Promela, UML, Spec#,
 - ◆ 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

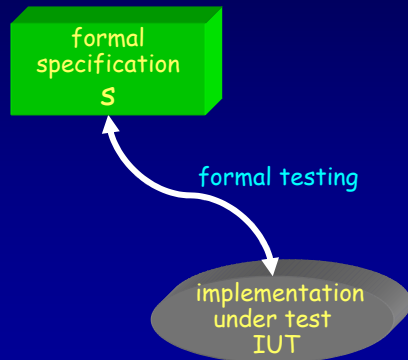
Model Based Testing Contents

- ☞ Introduction
 - ◆ Testing
 - ◆ Model based testing
- ☞ Theory: formal model based testing
 - ◆ Testing as a decision procedure
 - ◆ Test assumption, implementation relation, tests
- ☞ Theory: testing with transition systems
 - ◆ Implementation relation ioco
 - ◆ Test generation for ioco
- ☞ A Tool
 - ◆ TorX
- ☞ Applications
 - ◆ Lessons
- ☞ Conclusions, Perspectives



Specification Based Formal Functional Testing

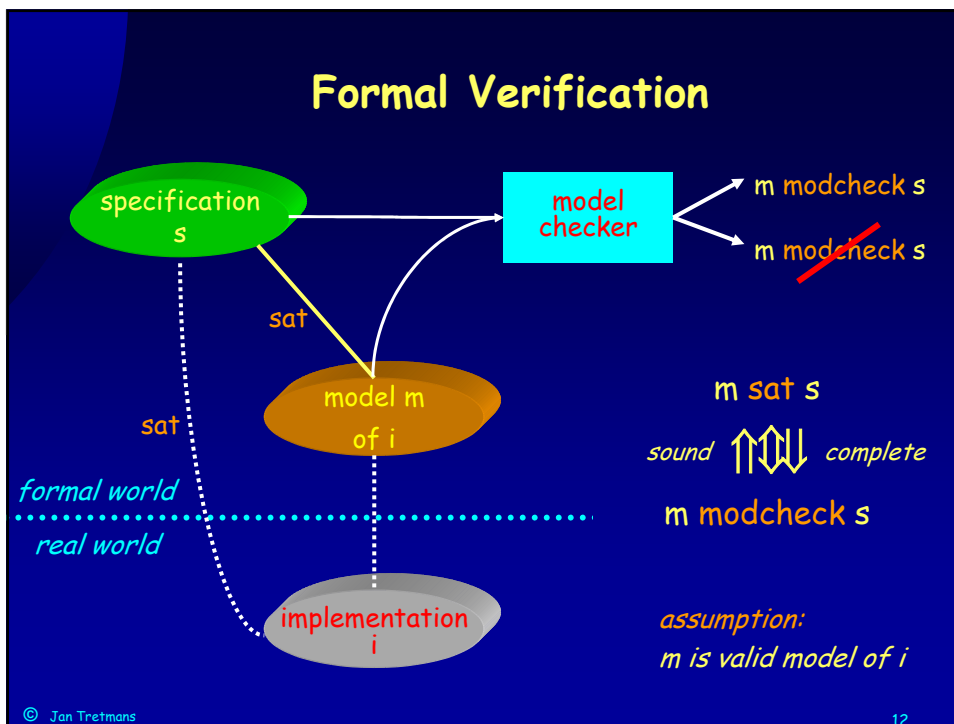
Model Based Testing



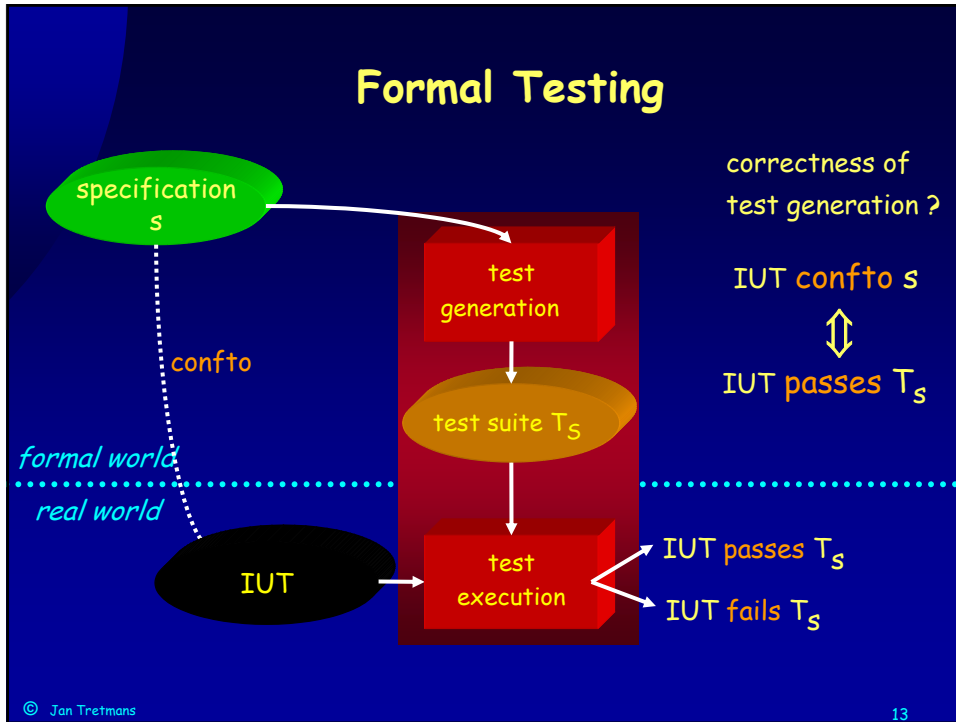
Testing functional behaviour of black-box implementation with respect to specification in a formal language based on formal definition of conformance

Specification/model assumed to be correct

Formal Verification



Formal Testing



Correctness of Formal Testing

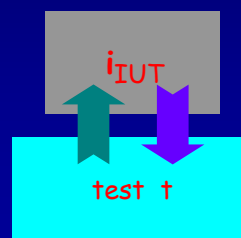
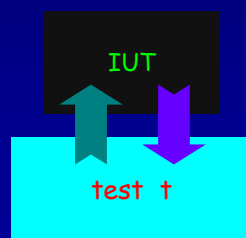


Formal Testing : Test Assumption

Test assumption :

$$\forall \text{IUT} . \exists i_{\text{IUT}} \in \text{MOD}.$$

$$\forall t \in \text{TEST} . \text{IUT passes } t \Leftrightarrow i_{\text{IUT}} \text{ passes } t$$



Testing Addition



Test a function adding numbers of two dice:

`int dadd (int x, y) for x, y ∈ [1..6]`

Is the following a complete test suite?

(1,1) (1,2) (1,6)
(2,1) (2,2) (2,6)
...
...
...
(6,1) (6,2) (6,6)

Testing Addition



Test a function adding numbers of two dice:
`int dadd (int x, y) for $x, y \in [1..6]$`

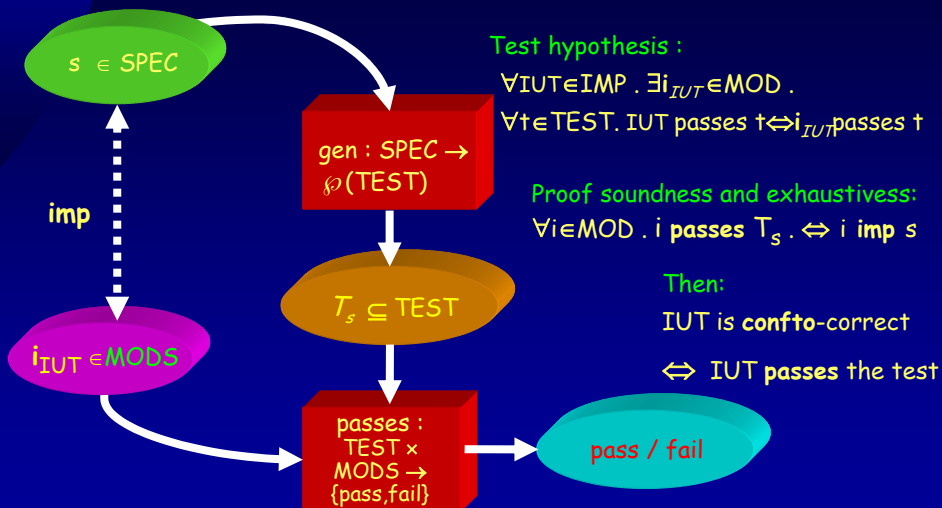
The test suite

(1,1)	(1,2)	(1,6)
(2,1)	(2,2)	(2,6)
...			
...			
(6,1)	(6,2)	(6,6)

is sound & exhaustive if

- ☞ `imp` is such that it requires correct addition for $x, y \in [1..6]$ only;
- ☞ the test assumption is that implementations can be modelled as functions : $MOD = int^{int \times int}$

Formal Testing



Model Based Testing

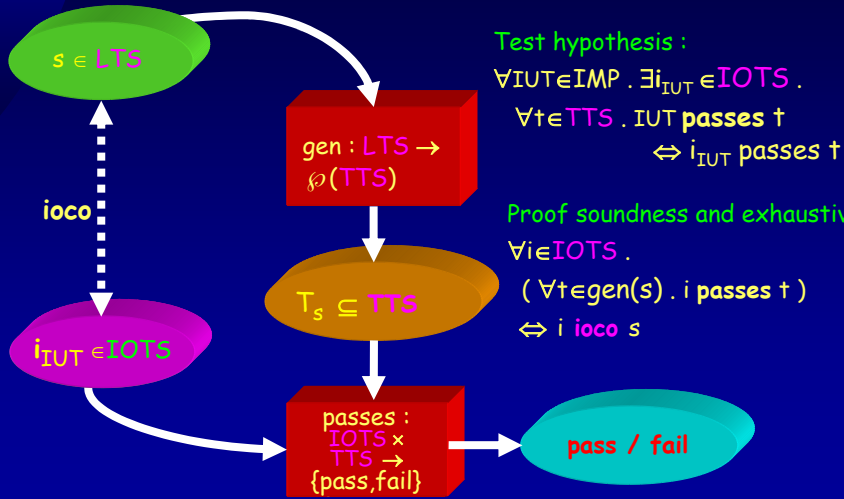
Contents

- ☞ Introduction
 - ◆ Testing
 - ◆ Model based testing
- ☞ Theory: formal model based testing
 - ◆ Testing as a decision procedure
 - ◆ Test assumption, implementation relation, tests
- ☞ Theory: testing with transition systems
 - ◆ Implementation relation **ioco**
 - ◆ Test generation for **ioco**
- ☞ A Tool
 - ◆ TorX
- ☞ Applications
 - ◆ Lessons
- ☞ Conclusions, Perspectives

Approaches to Formal Testing

- ☞ Finite State Machine
- ☞ Pre/post-conditions
- ☞ **Labelled Transition Systems**
- ☞ Programs as functions
- ☞ Abstract Data Type testing
- ☞

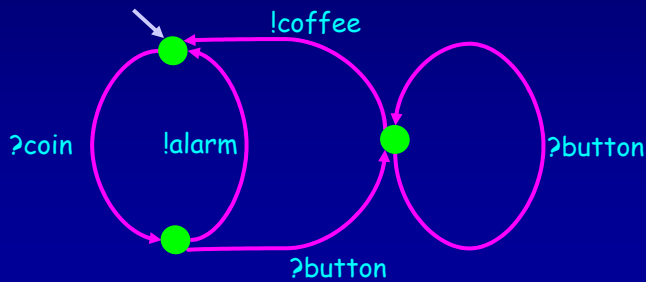
Formal Testing with Transition Systems



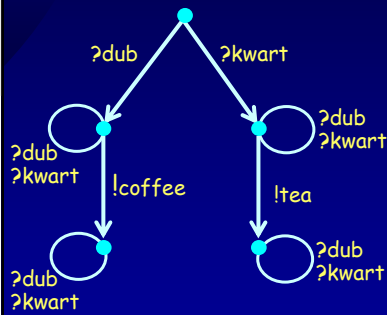
Specifications: Labelled Transition Systems

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

states
actions
transitions $T \subseteq S \times L \times S$
initial state $s_0 \in S$



Models of Implementations: Input-Output Transition Systems



$$L_I = \{ ?dub, ?kward \}$$

$$L_U = \{ !coffee, !tea \}$$

Input-Output Transition Systems

$$IOTS(L_I, L_U) \subseteq LTS(L_I \cup L_U)$$

IOTS is LTS with Input-Output
and **always enabled inputs**:

for all states s ,

$$\text{for all inputs } ?a \in L_I: s \xrightarrow{?a}$$

Implementation Relation **ioco**

Correctness expressed by **ioco** \subseteq IOTS \times LTS :

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

$$p \xrightarrow{\delta} p = \forall !x \in L_U \cup \{\tau\}. p \not\xrightarrow{!x}$$

$$\text{Straces}(s) = \{ \sigma \in (L \cup \{\delta\})^* \mid s \xrightarrow{\sigma} \}$$

$$p \text{ after } \sigma = \{ p' \mid p \xrightarrow{\sigma} p' \}$$

$$\text{out}(P) = \{ !x \in L_U \mid p \xrightarrow{!x}, p \in P \} \cup \{ \delta \mid p \xrightarrow{\delta}, p \in P \}$$

Implementation Relation ioco

Correctness expressed by $ioco \subseteq IOTS \times LTS$:

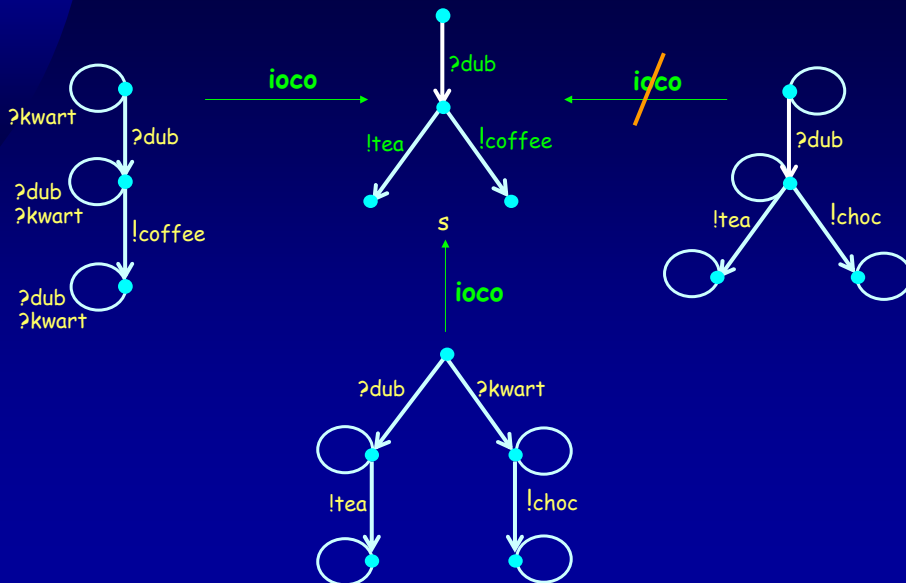
$$i \text{ ioco } s \stackrel{\text{def}}{=} \forall \sigma \in \text{Straces}(s) : \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma)$$

Intuition:

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

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

Implementation Relation ioco

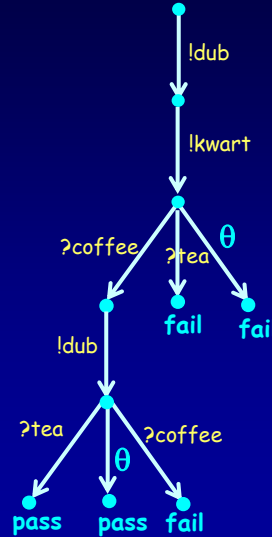


Test Cases

Test case $t \in TTS$

TTS - Test Transition System :

- ◆ labels in $L \cup \{\theta\}$
- ◆ tree-structured
- ◆ finite, deterministic
- ◆ final states **pass** and **fail**
- ◆ from each state \neq **pass**, **fail** :
 - either one input $!a$
 - or all outputs $?x$ and θ



ioco Test Generation Algorithm

Algorithm

To generate a test case from transition system specification s_0 , compute $T(S)$, with S a set of states, and initially $S = s_0$ after ε ;

For $T(S)$, apply the following recursively, non-deterministically:

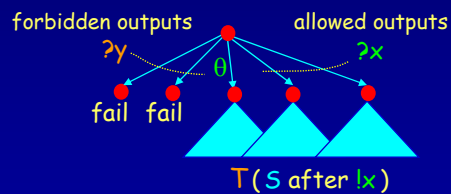
1 end test case

- pass

2 supply input

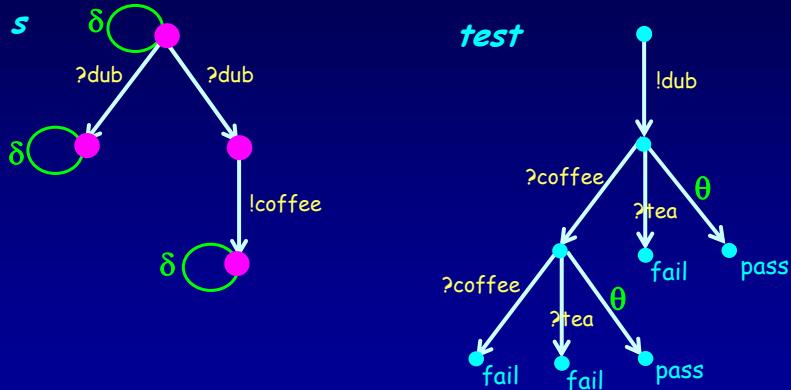


3 observe output

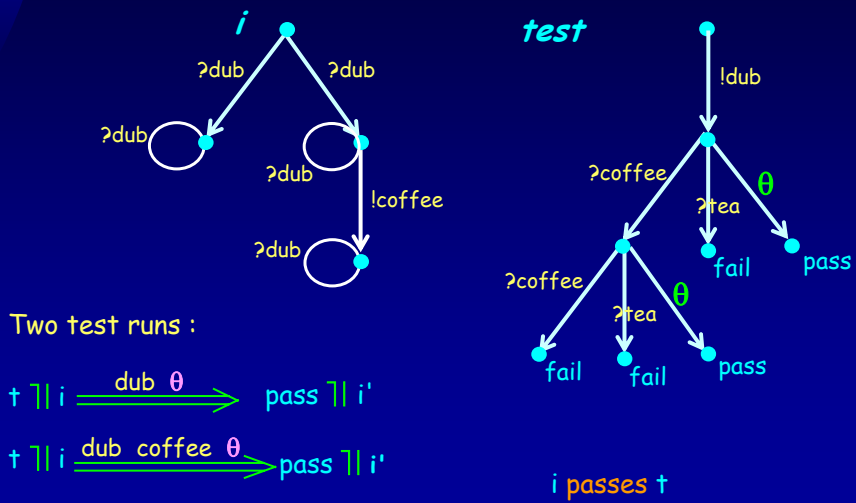


allowed outputs or δ : $!x \in out(S)$
 forbidden outputs or δ : $!y \notin out(S)$

Test Generation Example



Test Execution Example



Completeness of ioco Test Generation

For every test t generated with algorithm we have:

☞ Soundness :

t will never fail with correct implementation

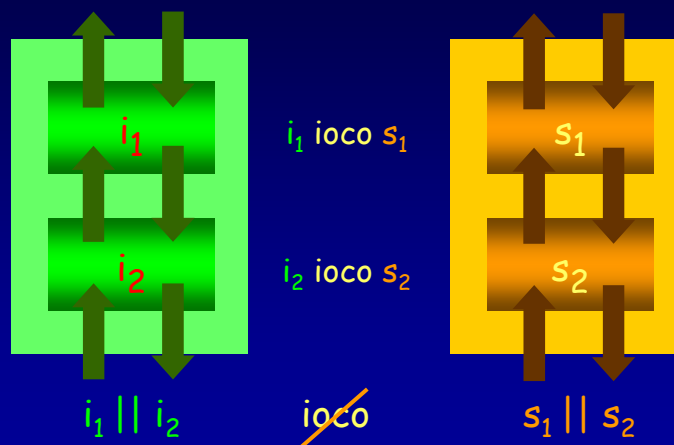
$i \text{ ioco } s$ implies $i \text{ passes } t$

☞ Exhaustiveness :

each incorrect implementation can be detected with a generated test t

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

Compositional Testing Component Based Testing



If s_1, s_2 input enabled - $s_1, s_2 \in \text{IOTS}$ - then ioco is preserved !

Model Based Testing

Contents

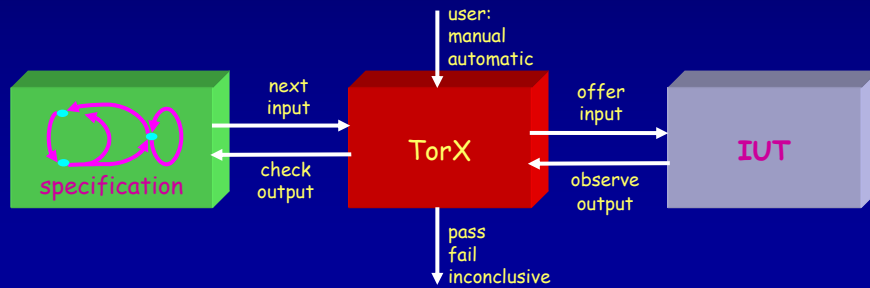
- ☞ Introduction
 - ◆ Testing
 - ◆ Model based testing
- ☞ Theory: formal model based testing
 - ◆ Testing as a decision procedure
 - ◆ Test assumption, implementation relation, tests
- ☞ Theory: testing with transition systems
 - ◆ Implementation relation *ioco*
 - ◆ Test generation for *ioco*
- ☞ A Tool
 - ◆ TorX
- ☞ Applications
 - ◆ Lessons
- ☞ Conclusions, Perspectives

Some Model Based Testing Approaches and Tools

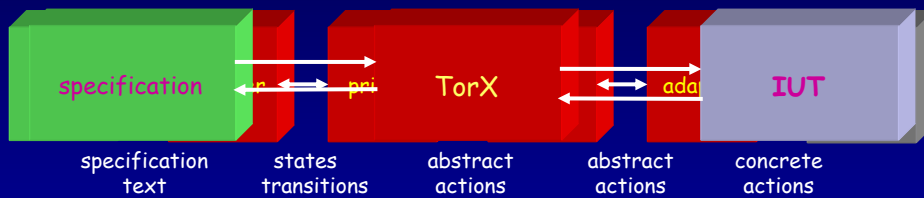
- | | |
|----------------------|------------------------|
| ☞ Agatha | ☞ Statemate MAGNUM ATG |
| ☞ Agedis | ☞ STG |
| ☞ Autolink | ☞ TestGen (Stirling) |
| ☞ Casper | ☞ TestGen (INT) |
| ☞ Gvst | ☞ TestComposer |
| ☞ Gotcha | ☞ TGV |
| ☞ Leirios | ☞ TorX |
| ☞ Phact/The Kit | ☞ T-Uppaal |
| ☞ RT-Tester | ☞ Tveda |
| ☞ SaMsTaG | ☞ |
| ☞ Spec#-SpecExplorer | |

A Tool for Transition Systems Testing: TorX

- ☞ On-the-fly test generation and test execution
- ☞ Implementation relation: **ioco**
- ☞ Mainly applicable to **reactive systems** / state based systems:
 - ◆ specification languages: **LOTOS, Promela, FSP, Automata**



TorX Tool Architecture



Model Based Testing

Contents

- ☞ Introduction
 - ◆ Testing
 - ◆ Model based testing
- ☞ Theory: formal model based testing
 - ◆ Testing as a decision procedure
 - ◆ Test assumption, implementation relation, tests
- ☞ Theory: testing with transition systems
 - ◆ Implementation relation *ioco*
 - ◆ Test generation for *ioco*
- ☞ A Tool
 - ◆ TorX
- ☞ Applications
 - ◆ Lessons
- ☞ Conclusions, Perspectives

TorX Case Studies

- | | |
|-----------------------------------------------|--------------|
| ☞ Conference Protocol | academic |
| ☞ EasyLink TV-VCR protocol | Philips |
| ☞ Cell Broadcast Centre component | LogicaCMG |
| ☞ "Rekeningrijden" Payment Box protocol | Interpay |
| ☞ V5.1 Access Network protocol | Lucent |
| ☞ Easy Mail Melder | LogicaCMG |
| ☞ FTP Client | academic |
| ☞ "Oosterschelde" storm surge barrier-control | LogicaCMG |
| ☞ DO/DG dose control | ASML/Tangram |
| ☞ Laser interface | ASML/Tangram |

TorX Case Study Lessons

- ☞ model construction
 - ◆ difficult: missing or bad specs
 - ◆ leads to detection of design errors
 - ◆ not supported by integrated environment yet
 - ◆ research on **test based modelling**
- ☞ adapter/test environment
 - ◆ development is cumbersome
 - ◆ specific for each system
- ☞ longer and more flexible tests
 - ◆ full automation : test generation + execution + analysis
- ☞ no notion of test selection or specification coverage yet
 - ◆ only random coverage or user guided test purposes

Concluding

- ☞ Testing can be formal, too (M.-C. Gaudel, TACAS'95)
 - ◆ Testing shall be formal, too
- ☞ A test generation algorithm is not just another algorithm :
 - ◆ Proof of **soundness** and **exhaustiveness**
 - ◆ Definition of **test assumption** and **implementation relation**
- ☞ For labelled transition systems :
 - ◆ **ioco** for expressing conformance between **imp** and **spec**
 - ◆ a sound and exhaustive **test generation algorithm**
 - ◆ **tools** generating and executing tests:
TGV, TestGen, Agedis, TorX,

Perspectives

Model based formal testing can improve the testing process :

- ☺ model is precise and unambiguous basis for testing
 - ◆ design errors found during validation of model
- ☺ longer, cheaper, more flexible, and provably correct tests
 - ◆ easier test maintenance and regression testing
- ☺ **Automatic test generation and execution**
 - ◆ full automation : test generation + execution + analysis
- ☺ Extra effort of modelling compensated by better tests

Thank You