Model-Based Testing

Ed Brinksma

University of Twente
Dept. of Computer Science
Formal Methods & Tools group
Enschede
The Netherlands

ARTIST2 Summer School
Nässlingen

Contents

- introduction & background
- testing pre-orders
- input/output & quiescence
- ioco implementation relation
- test generation
- TorX
- test case study
- real-time testing

Contents

- introduction & background
- testing pre-orders
- input/output & quiescence
- ioco implementation relation
- test generation
- TorX
- test case study
- real-time testing

Practical problems of testing

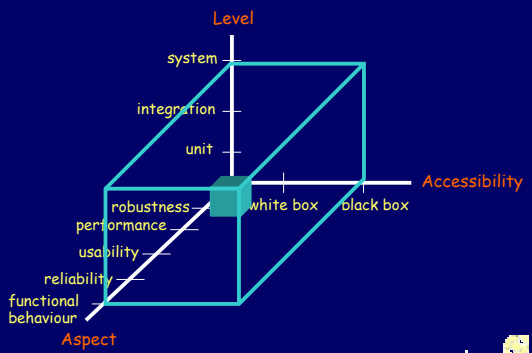
Testing is:
- important
- much practiced
- 30% - 50% of project effort
- expensive
- time critical
- not constructive (but sadistic?)

But also:
- ad-hoc, manual, error-prone
- hardly theory / research
- no attention in curricula
- not *cool*: "if you're a bad programmer you might be a tester"

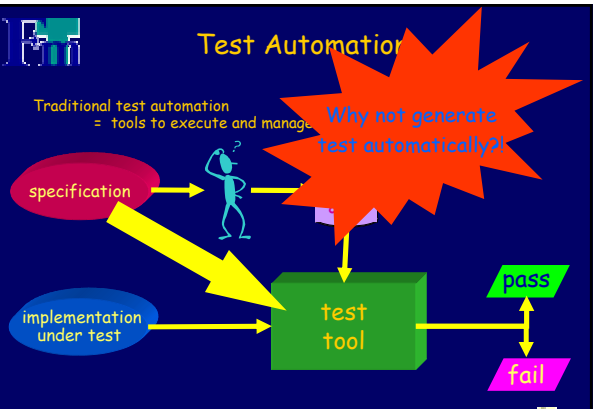
Improvements possible with formal methods!?

Attitude is changing:
- more awareness
- more professional

Types of Testing



Test Automation



Verification and Testing

Verification :
- formal manipulation
- prove properties
- performed on model

Testing :
- experimentation
- show error
- concrete system

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

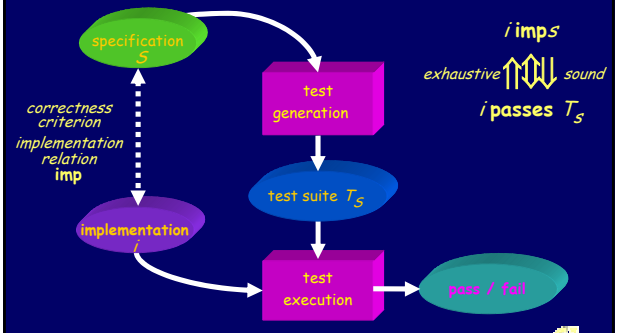
Testing with Formal Methods

- 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

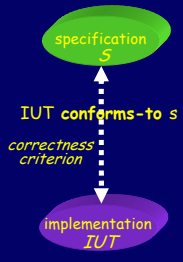
Challenges of Testing Theory

- 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 and invested effort
  - usually stop when time is over .....

Formal Testing



Formal Testing : Conformance



s ∈ SPECS Specification
IUT Implementation under Test

IUT is concrete, physical object

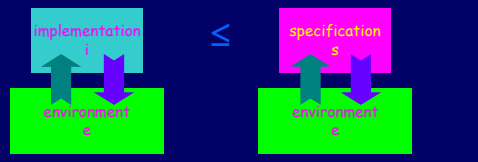
Model the physical world

But IUT is black box ! ?

Assume that model i_{IUT} exists

Contents

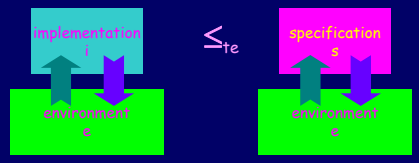
Testing Preorders on Transition Systems



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

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

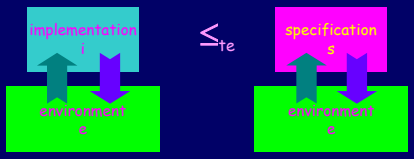
Classical Testing Preorder



$i \leq_{te} s \Leftrightarrow \forall e \in E . obs(e,i) \subseteq obs(e,s)$

LTS(L) Deadlocks(e||s)

Classical Testing Preorder



$i \leq_{te} s \Leftrightarrow \forall e \in LTS(L) . \forall \sigma \in L^* .$

e||i deadlocks after σ ⇒ e||s deadlocks after σ

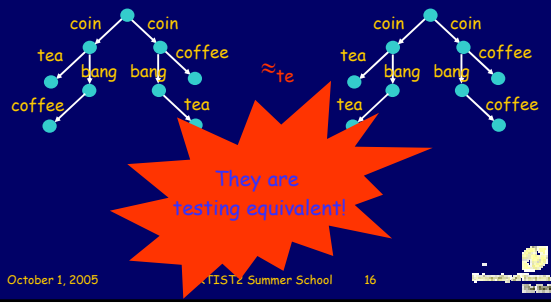
$\{ \sigma \mid e||i \text{ deadlocks after } \sigma \} \subseteq \{ \sigma \mid e||s \text{ deadlocks after } \sigma \}$

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

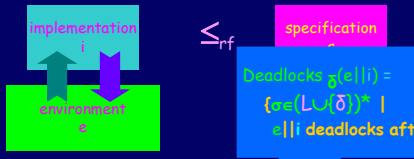
$FP(p) = \{ \langle \sigma, A \rangle \mid p \text{ afer } \sigma \text{ refuses } A \}$

Quirky Coffee Machine [Langerak]

Can we distinguish between these machines?



Refusal Preorder



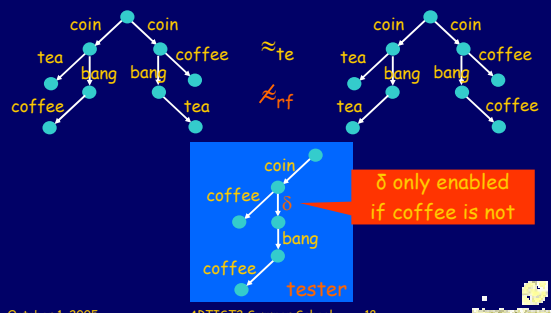
$Deadlocks_\delta(e||i) = \{ \sigma \in (L \cup \{\delta\})^* \mid e||i \text{ deadlocks after } \sigma \}$

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

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

e observes with δ deadlock on all alternative actions

Quirky Coffee Machine Revisited



Contents

- introduction & background
- testing pre-orders
- input/output & quiescence
- ioco implementation relation
- test generation
- TorX
- test case study
- real-time testing

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)

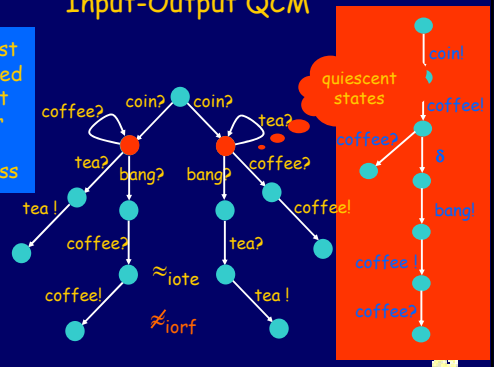
  for all states s, for all $a \in L_{in}$ $s \stackrel{a}{\Rightarrow}$
- testers are I/O systems
  - output (stimulus) is input for the SUT
  - input (response) is output of the SUT

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:
     - Deadlocks(I||T) ⊆ Deadlocks(S||T) (quiescence)
  2. $I \leq_{iorf} S$ iff for all I/O testers T:
     - $Deadlocks_{\delta}(I||T) \subseteq Deadlocks_{\delta}(S||T)$ (repetitive quiescence)

Input-Output QCM

states must be saturated with input loops for input enabledness



Contents

- introduction & background
- testing pre-orders
- input/output & quiescence
- ioco implementation relation
- test generation
- TorX
- test case study
- real-time testing

Implementation Relation ioco

By adding a transition $p \stackrel{\delta}{\rightarrow} p$ to every quiescent state of a system we treat quiescence as an observable (synchronizable) action:

$i \leq_{iorf} s \Leftrightarrow \forall$ I/O tests T: $Deadlocks_{\delta}(i||T) \subseteq Deadlocks_{\delta}(s||T)$

$\Leftrightarrow \forall \sigma \in (L \cup \{\delta\})^*$: $out(i \text{ after } \sigma) \subseteq out(s \text{ after } \sigma)$

To allow under-specification we restrict the set of traces:

$i \text{ ioco } s \Leftrightarrow \forall \sigma \in Traces_{\delta}(s) : out(i \text{ after } \sigma) \subseteq out(s \text{ after } \sigma)$

Implementation Relation ioco

Correctness expressed by implementation relation **ioco**:

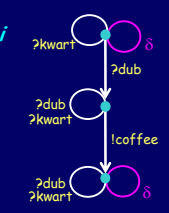
$i \text{ ioco } s =_{def} \forall \sigma \in Traces_{\delta}(s) : out(i \text{ after } \sigma) \subseteq out(s \text{ after } \sigma)$

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

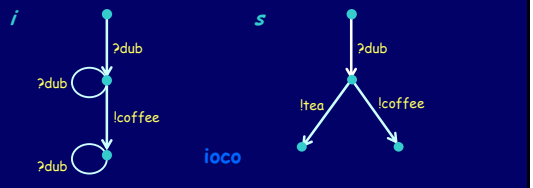
$i \text{ ioco } s =_{def} \forall \sigma \in Straces(s) : out(i \text{ after } \sigma) \subseteq out(s \text{ after } \sigma)$



$out(i \text{ after } \varepsilon) = \{\delta\}$
$out(i \text{ after } ?dub) = \{!coffee\}$
$out(i \text{ after } ?dub.?dub) = \{!coffee\}$
$out(i \text{ after } ?dub.!coffee) = \{\delta\}$
$out(i \text{ after } ?kwart) = \{\delta\}$
$out(i \text{ after } !coffee) = \emptyset$
$out(i \text{ after } ?dub.!tea) = \emptyset$
$out(i \text{ after } \delta) = \{\delta\}$

Implementation Relation ioco

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



$out(i \text{ after } ?dub) = \{!coffee\}$ $out(s \text{ after } ?dub) = \{!coffee, !tea\}$

Implementation Relation ioco

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

$out(i \text{ after } ?dub) = \{!coffee, !tea\} \not\subseteq out(s \text{ after } ?dub) = \{!coffee\}$

Implementation Relation ioco

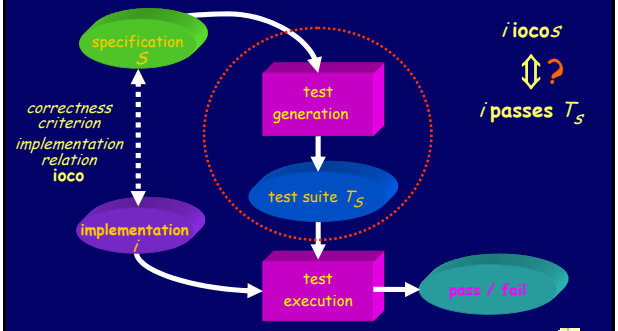
$i \text{ ioco } s =_{def} \forall \sigma \in Straces(s) : out(i \text{ after } \sigma) \subseteq out(s \text{ after } \sigma)$

$out(i \text{ after } ?dub) = \{!coffee\}$ $out(s \text{ after } ?dub) = \{!coffee\}$
$out(i \text{ after } ?kwart) = \{!tea\}$ $out(s \text{ after } ?kwart) = \emptyset$

But $?kwart \notin Traces_{\delta}(s)$

Contents

Formal Testing

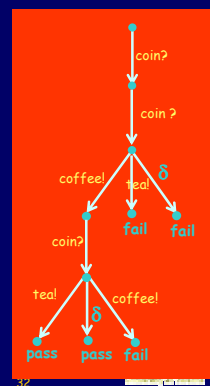


Test Cases

Test case t ∈ TTS

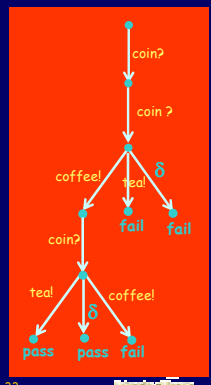
TTS - Test Transition System :

- labels in $L \cup \{\delta\}$
- tree-structured
- finite, deterministic
- final states pass and fail
- from each state ≠ pass, fail
  - either one input i?
  - or all outputs o! and δ



Test Cases

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



Test Generation Algorithm

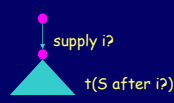
Algorithm

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

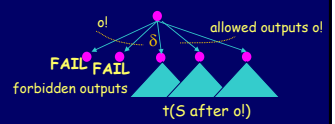
Apply the following steps recursively, non-deterministically

1 end test case
- PASS

2 supply input



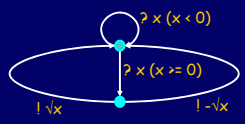
3 observe output



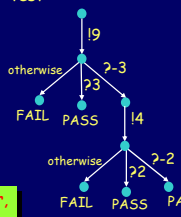
Test Generation Example

Equation solver for $y^2=x$

specification



test



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

Validity of Test Generation

For every test t generated with algorithm :

- Soundness :
  t will never fail with correct implementation

  i **ioco** s implies i passes t

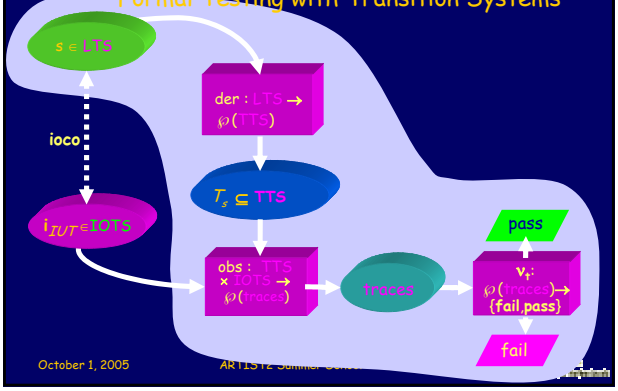
- Exhaustiveness :
  each incorrect implementation can be detected with a generated test t

  i **ioco** s implies ∃ t : i fails t

Contents

- introduction & background
- testing pre-orders
- input/output & quiescence
- ioco implementation relation
- test generation
- TorX
- test case study
- real-time testing

Formal Testing with Transition Systems

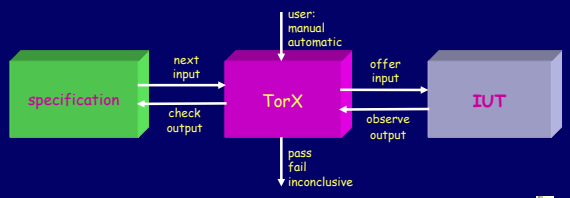


Test Generation Tools for ioco

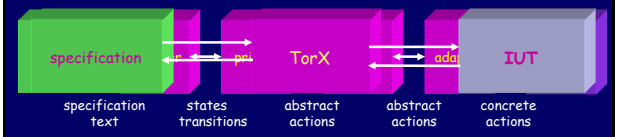
- TVEDA (CNET - France Telecom)
  - derives TTCN tests from single process SDL specification
  - developed from practical experiences
  - implementation relation R1 ≈ **ioco**
- TGV (IRISA - Rennes)
  - derives tests in TTCN from LOTOS or SDL
  - uses test purposes to guide test derivation
  - implementation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

A Test Tool : TorX

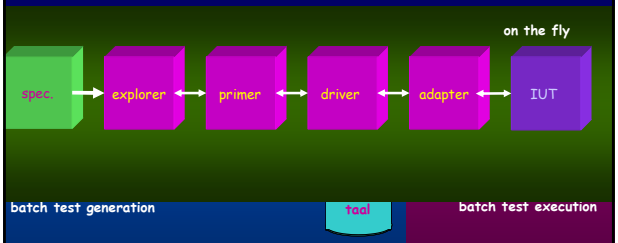
- On-the-fly test generation and test execution
- Implementation relation: **ioco**
- Specification languages: LOTOS, Promela, FSP, Automata



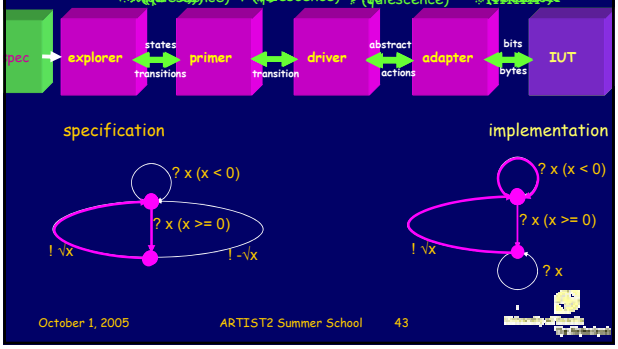
TorX Tool Architecture



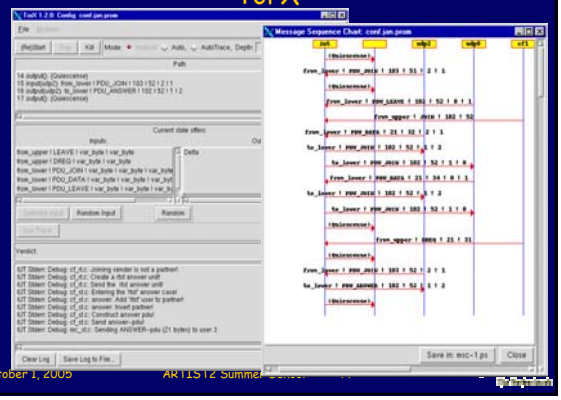
On-the-Fly ↔ Batch Testing



On-the-Fly Testing



TorX



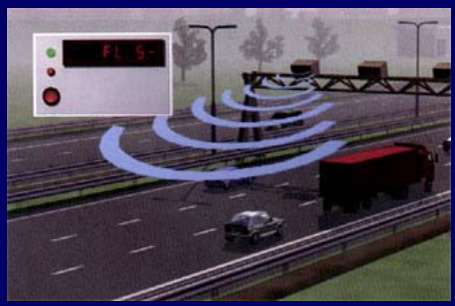
Contents

- introduction & background
- testing pre-orders
- input/output & quiescence
- ioco implementation relation
- test generation
- TorX
- test case study
- real-time testing

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 Melder	CMG
FTP Client	academic
"Oosterschelde" storm surge barrier-control	CMG
TANGRAM: testing VLSI lithography machine	ASML

Interpay
Highway Tolling System

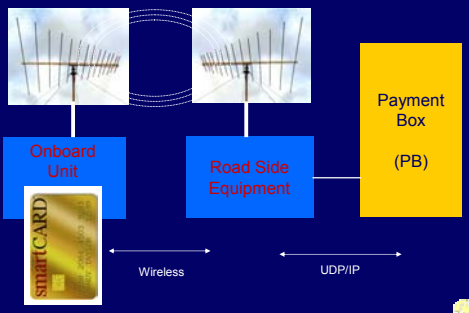


Highway Tolling Protocol

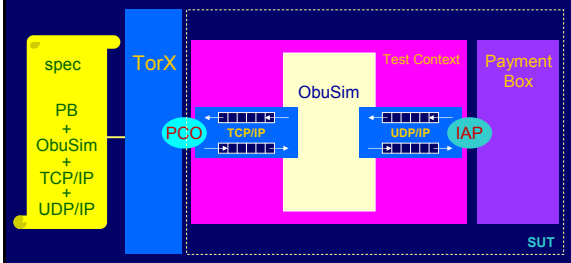
Characteristics :

- Simple protocol
- Parallellism :
  many cars at the same time
- Encryption
- System passed traditional testing phase

Highway Tolling System



Highway Tolling: Test Architecture



Highway Tolling: Results

- Test results :
  - 1 error during validation (design error)
  - 1 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
- **Step ahead in formal testing of realistic systems**

Contents

- introduction & background
- testing pre-orders
- input/output & quiescence
- ioco implementation relation
- test generation
- TorX
- test case study
- real-time testing

RT TorX Hacking Approaches

1. **Ignore RT functionality:**
   - test pure functional behaviour
   - analyse timing requirements using TorX log files & assumed timing constraints
2. **Add timestamps to observations**
   - adapter adds timestamps to observations when they are made and passed on to the driver
   - timestamps are used to analyse TorX log files
3. Add timestamps to stimuli & observations
   - adapter add timestamps to observations when they are made and passed on to the driver
   - adapter adds timestamps to stimuli when they are applied and returned to the driver
   - 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

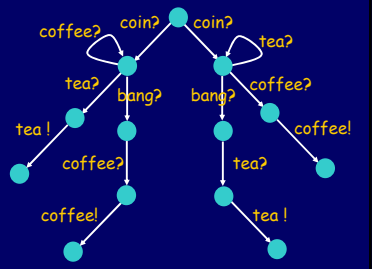
Real-time Testing and I/O Systems

- 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?

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: $s \stackrel{a(d)}{\Rightarrow}$
- special transitions:
  $s \stackrel{\delta}{\rightarrow} s$ for every quiescent system state s
- testers observing quiescence take time:
  $Test_M$: set of test processes having only δ(M)-actions to observe quiescence
- assume that implementations are M-quiescent:
  for all reachable states s and s':
  if $s \stackrel{\varepsilon(M)}{\Rightarrow} s'$ then s' is quiescent

Real-Time and Quiescence

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

$Deadlocks_\delta(i||T) \subseteq Deadlocks_\delta(s||T)$

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

$out_M(i \text{ after } \sigma) \subseteq out_M(s \text{ after } \sigma)$

$i \text{ tioco}_M s \Leftrightarrow \forall \sigma \in Traces_{\delta(M)}(s)$:

$out_M(i \text{ after } \sigma) \subseteq out_M(s \text{ after } \sigma)$

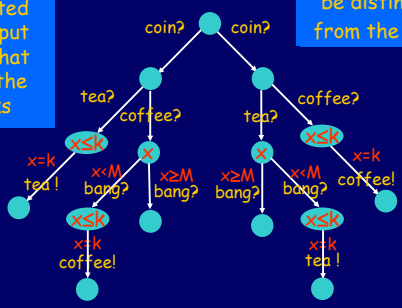
Properties

1. for all $M_1 \leq M_2$:
   $i \leq^{M_1}_{tiorf} s$ implies $i \leq^{M_2}_{tiorf} s$
2. for all time-independent i,s and $M_1, M_2 \geq 0$
   $i \leq^{M_1}_{tiorf} s$ iff $i \leq^{M_2}_{tiorf} s$ iff $i \leq_{iorf} s$

A limitation

states are saturated with input loops that reset the clocks

this process cannot be distinguished from the previous

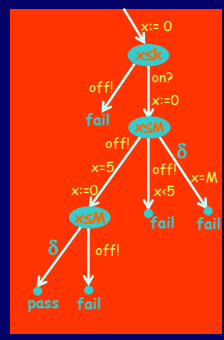


Test Cases

Test case $t \in TTA$

TTA – Test Timed Automata :

- labels in $L \cup \{\delta\}$, G(d)
- 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 unit provide input i?
  - or wait for time M accepting all outputs o! and δ



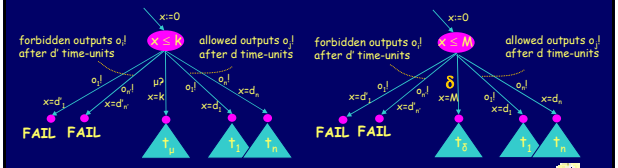
Timed test Gen

To generate a test case t(S) from specification with S set of states

apply the following steps recursive

can be calculated effectively only for subclasses of timed transition systems!

1. end test case
   - PASS
2. choose k ∈ (0, M) and input μ
3. wait for observing possible output

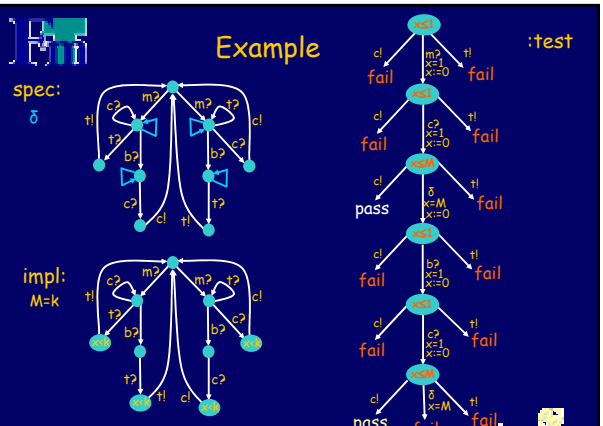


Example

spec: δ

impl: M=k

:test



Soundness & Completeness

- the non-timed generation algorithm can be shown [illegible] 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 are uncount[illegible] traces and only countably many test are gen[illegible] 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

Current Work

- Extension of the framework
  - M as a function of the specification state/output channel
  - integration with symbolic data generation
  - test action refinement
  - robustness & tolerance in real-time testing
- Extending TorX environment using CORBA IDL
  - generate abstract TorX actions
  - generate TTCN-3 signatures
  - generate adapter code
- Practical application
  - TANGRAM project: testing control software for VLSI lithography machines (ASML)
  - smooth transition between timed & untimed testing

Future Work

- stochastic systems
- quality of service
- hybrid systems
- coverage measures
- integration white/black box spectrum
- ...

For more information

fmt.cs.utwente.nl/research/testing