Radboud Universiteit

ESI

Principles of

Model-Based Testing

Jan Tretmans

TNO-ESI – Embedded Systems Innovation by TNO
Radboud University Nijmegen
jan.tretmans@tno.nl

TNO innovation for life

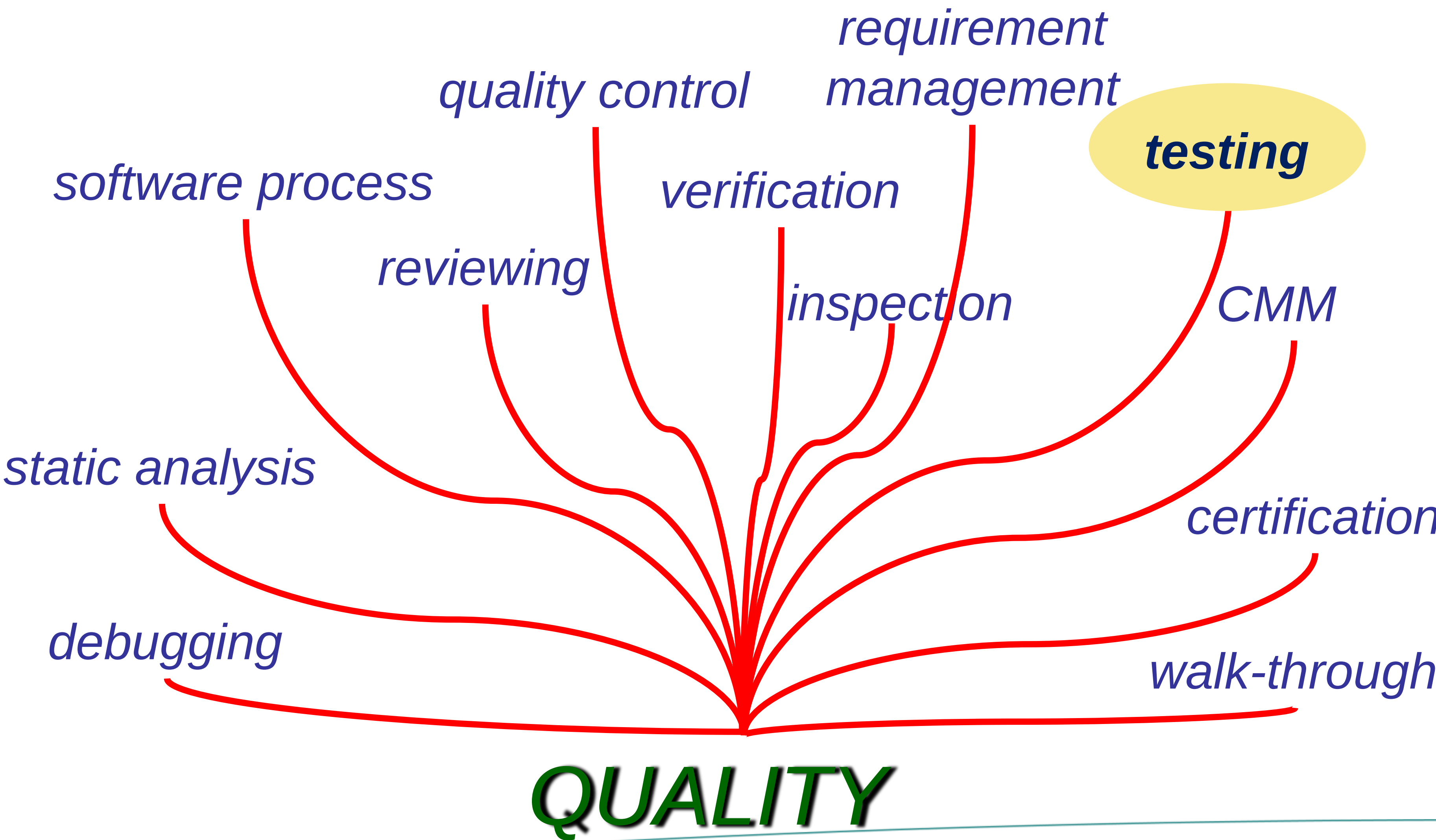
Software Testing

Testing: A Definition

Software testing is:

- a technical process,
- performed by executing / experimenting with a product,
- in a controlled environment, following a specified procedure,
- with the intent of measuring one or more characteristics / quality of the software product
- by demonstrating the deviation of the actual status of the product from the required status / specification.

Quality : There is more than Testing

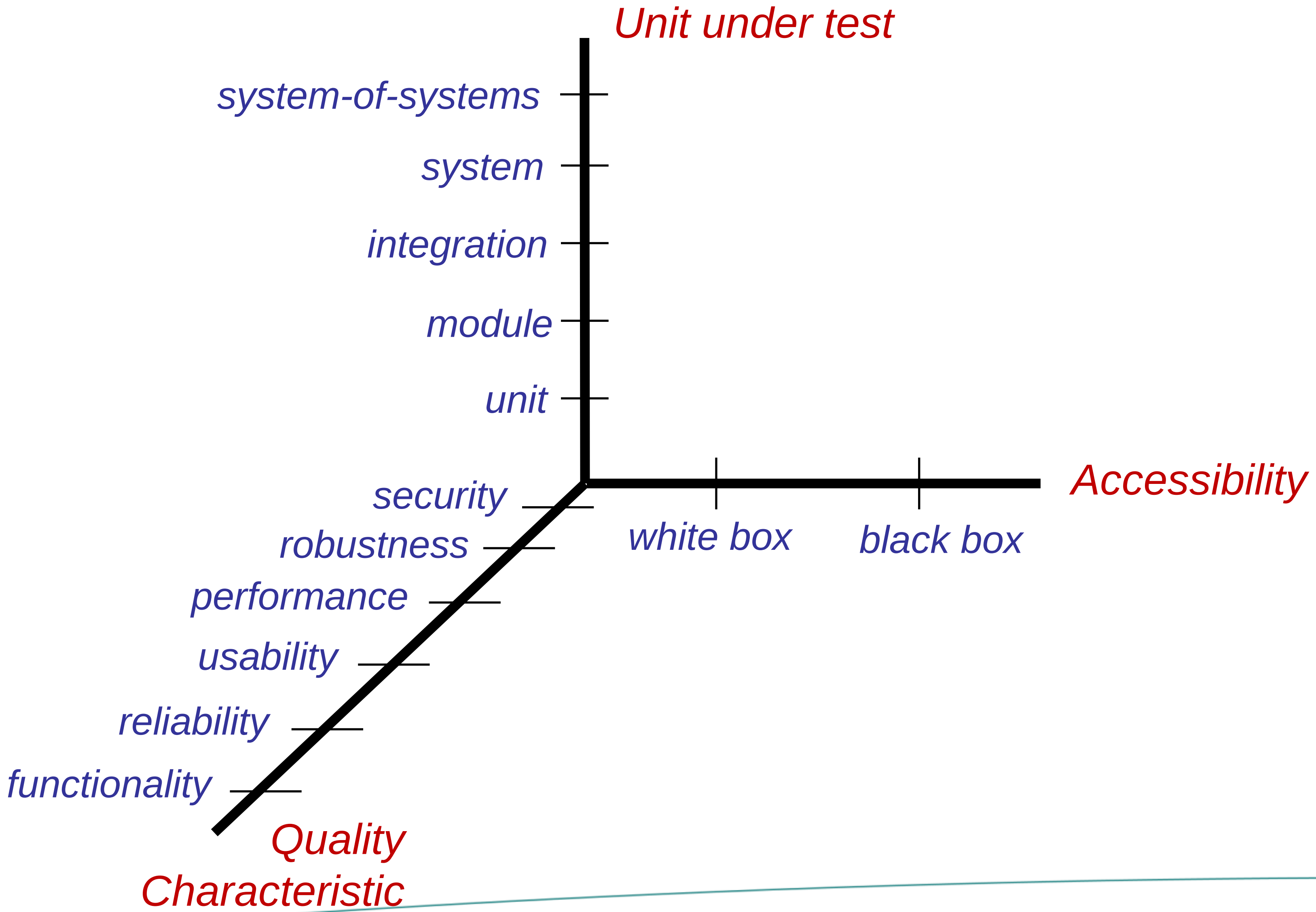


Sorts of Testing

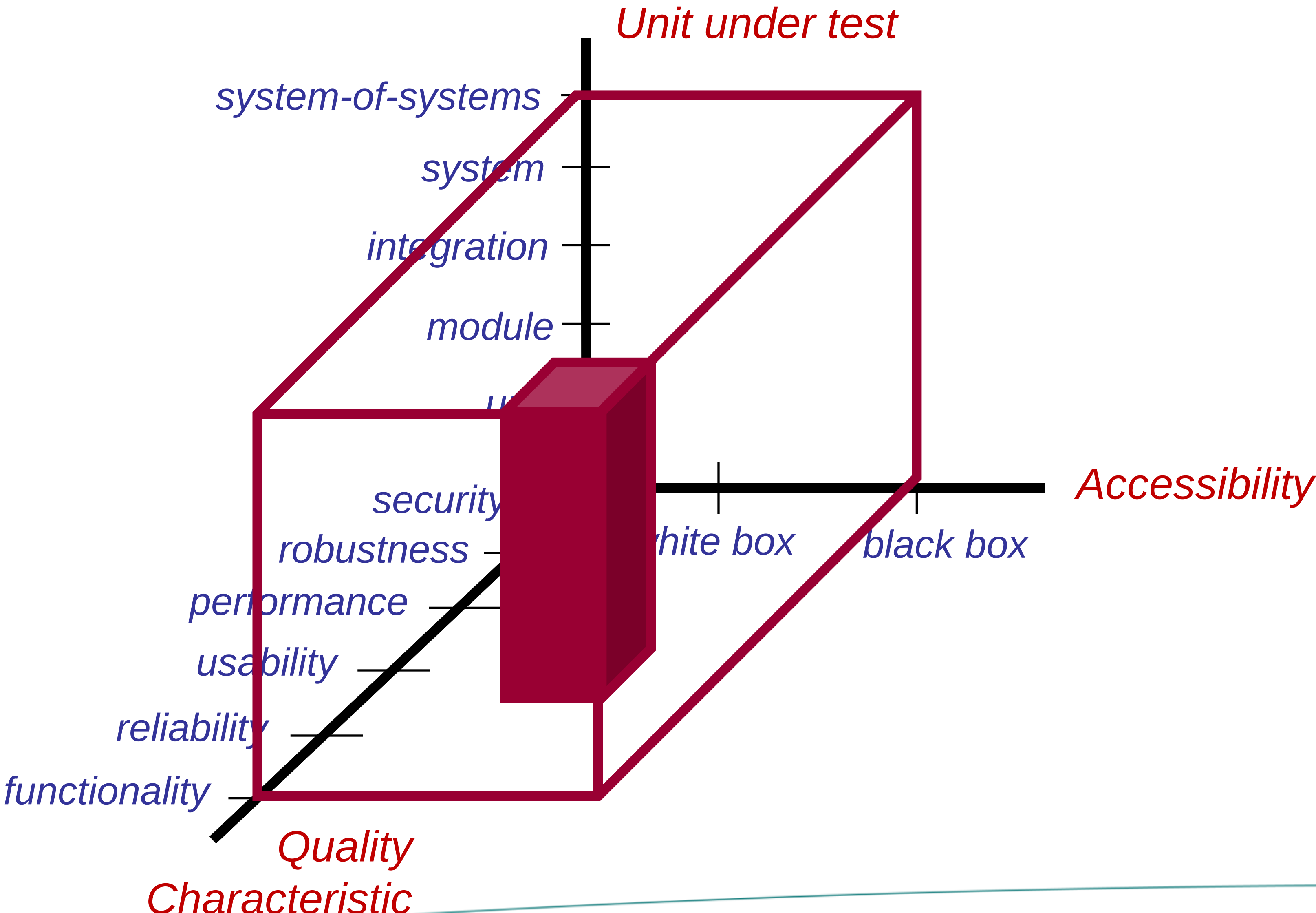
Sorts of Testing : Classification

- Quality characteristics
  - functional, security, compliance, interoperability
  - reliability, robustness, usability, learnability
  - performance, resource, stress, portability, conformance
- Who
  - developer, tester, user, QA, third party, certifier
  - alpha testing, beta testing, system admin, . . .
- Phase
  - programming, integration, acceptance, regression, . . .
- Unit under test
  - unit, module, component, subsystem, system, system-of-systems
  - documentation, system-in-context
- Goal of testing
  - bug finding, confidence, certification, . . .
- Testing techniques
  - Black / white-box, . . .
- . . . . .

Sorts of Testing



Sorts of Testing : Model-Based Testing



Software Testing

Measuring some quality characteristic
of an executing software object
by performing experiments
in a controlled environment
while comparing actual behaviour
with required behaviour

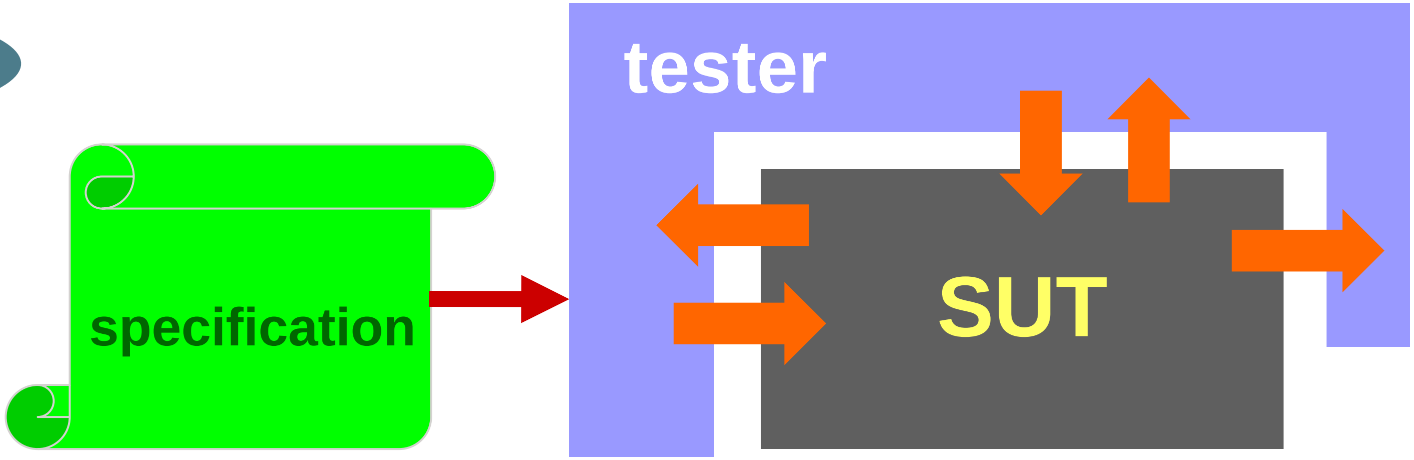
functionality

SUT

tester

specification

specification-based,
active,
black-box
testing
of functionality

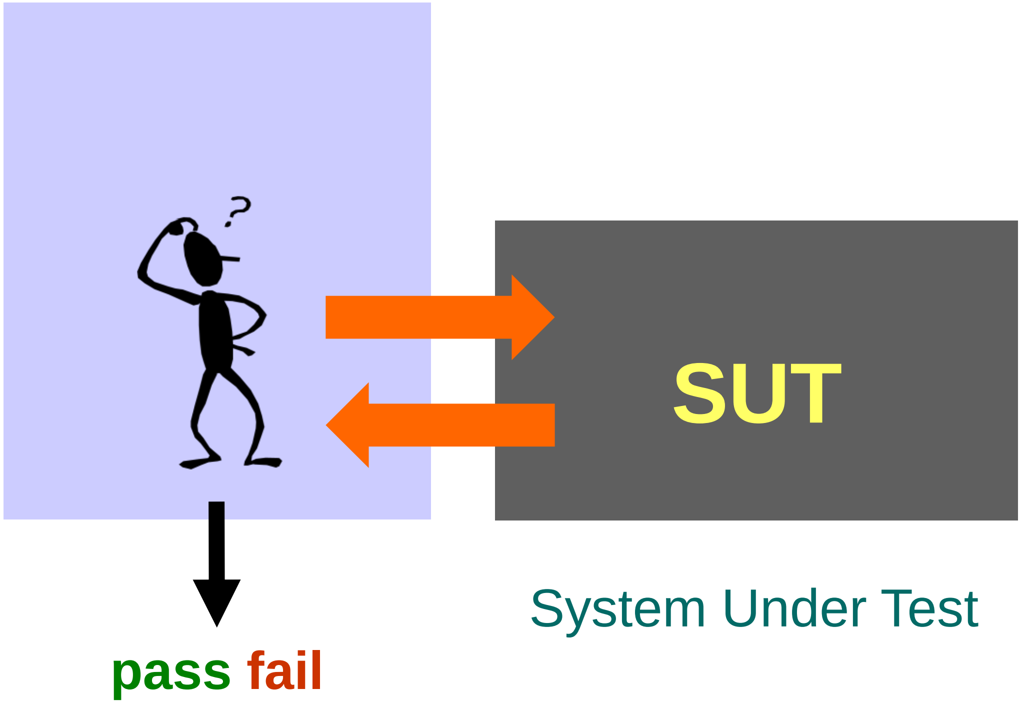


Model-Based Testing

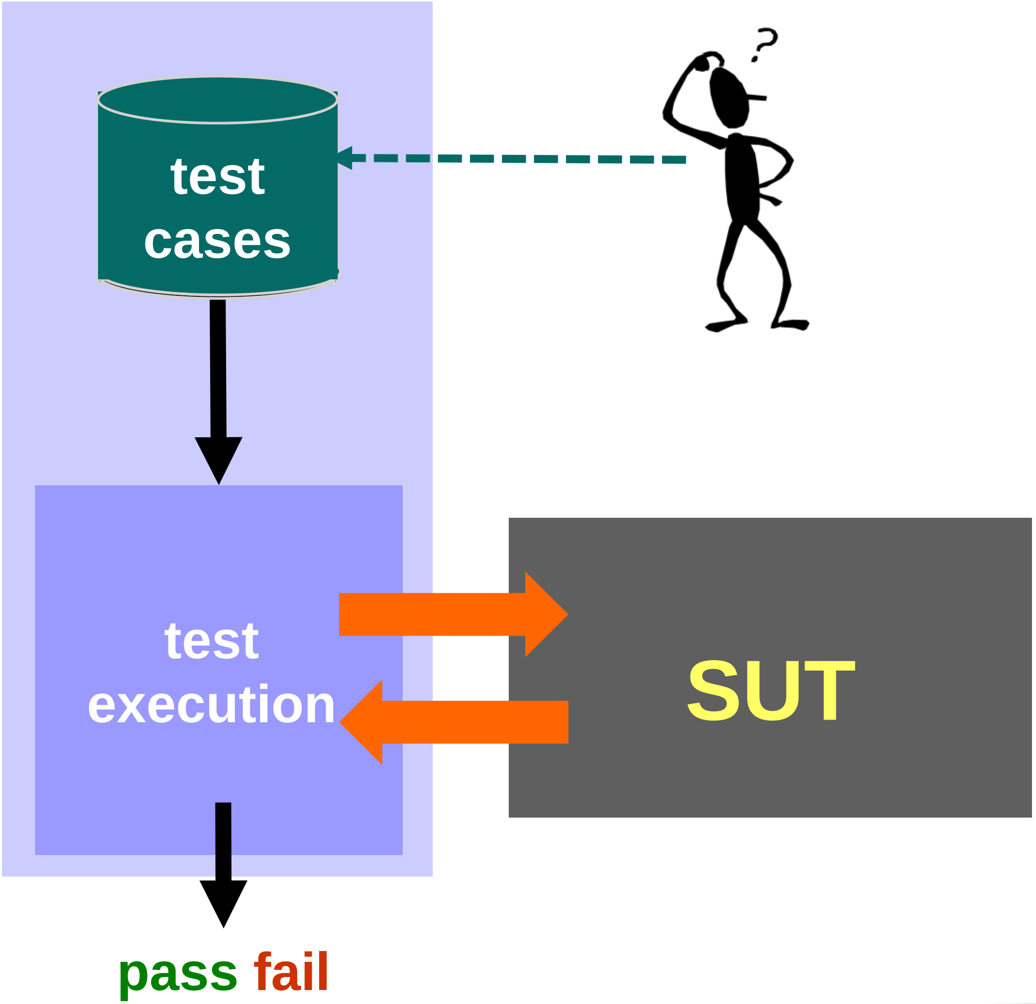
Basics

1 : Manual Testing

1. **Manual testing**



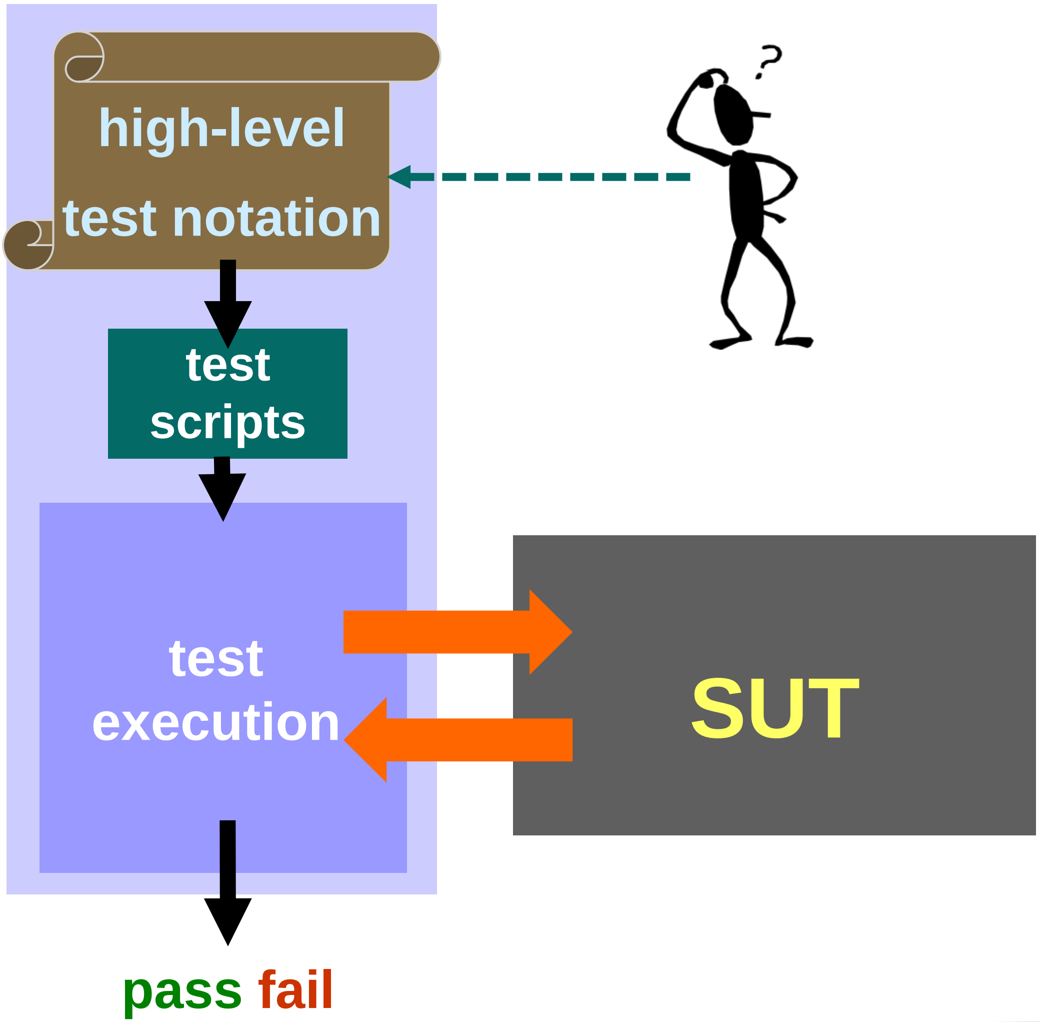
2 : Scripted Testing



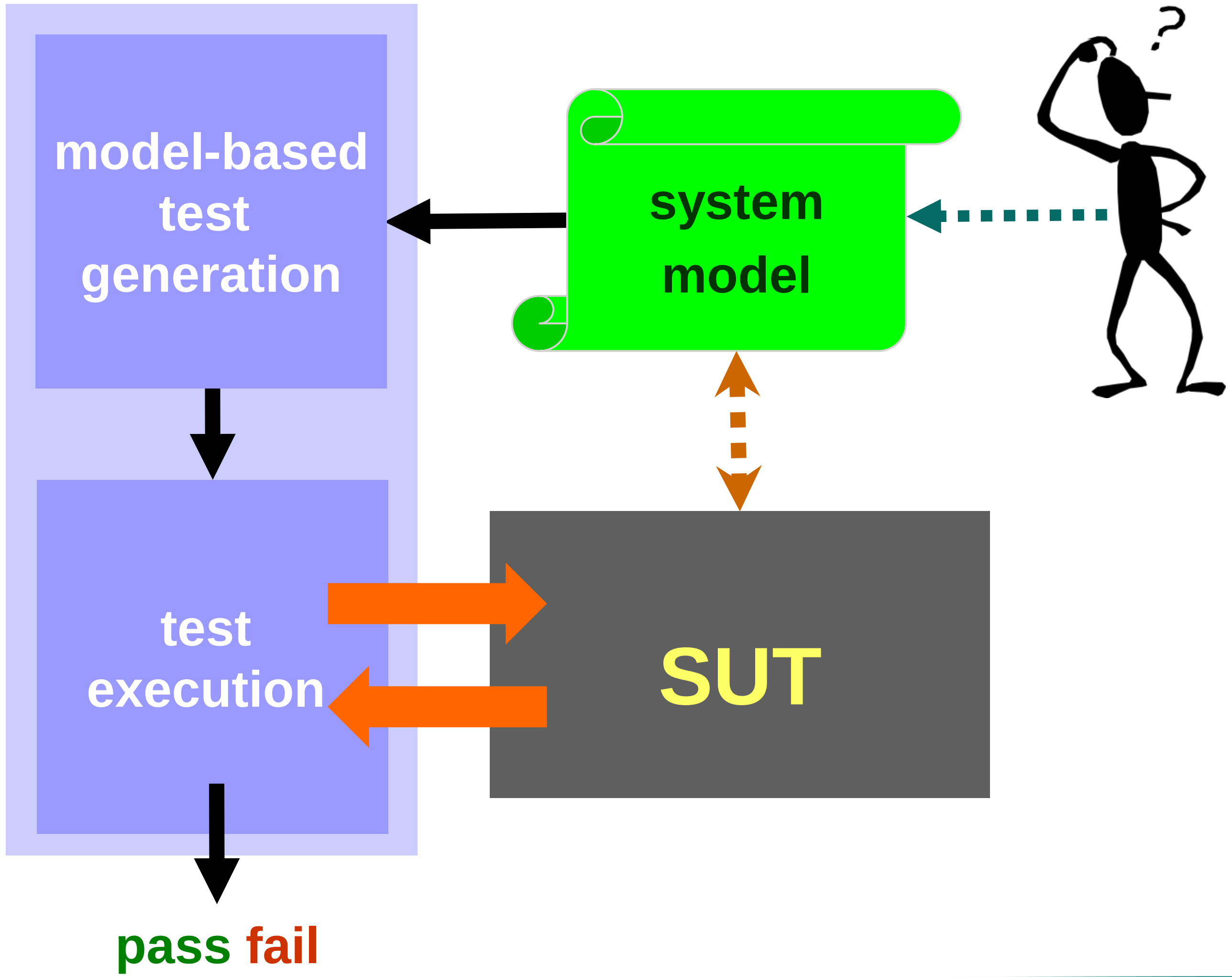
1. Manual testing
2. Scripted testing

3 : Keyword-Driven Testing

1. Manual testing
2. Scripted testing
3. **Keyword-driven testing**



4 : Model-Based Testing



1. Manual testing
2. Scripted testing
3. Keyword-driven testing
4. **Model-based testing**

Model-Based Testing

Measuring some quality characteristic
of an executing software object
by performing experiments
in a controlled environment
while comparing actual behaviour
with required behaviour

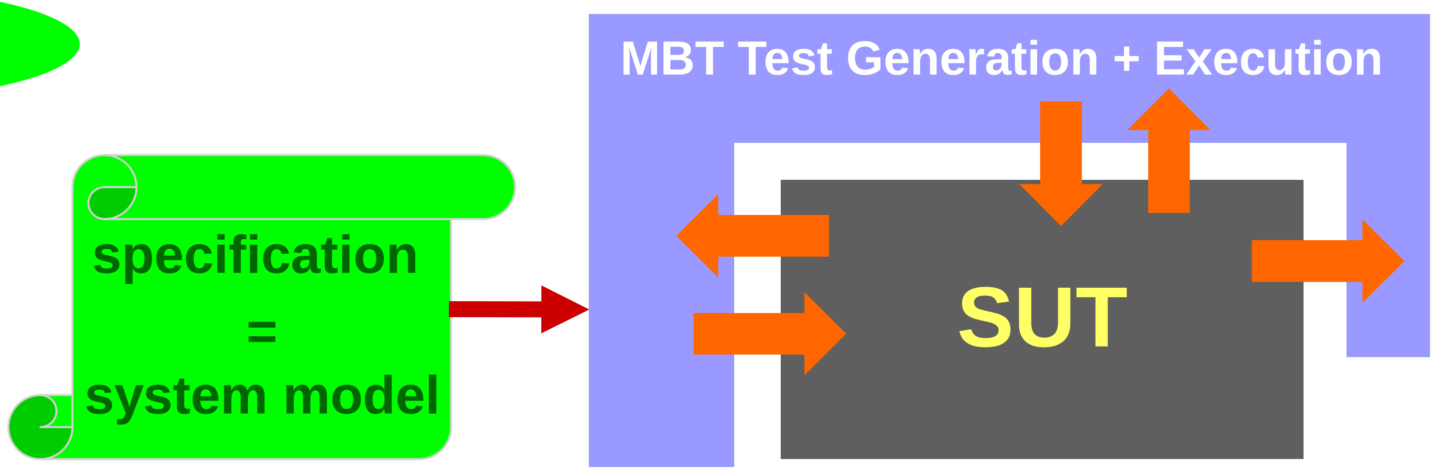
functionality

SUT

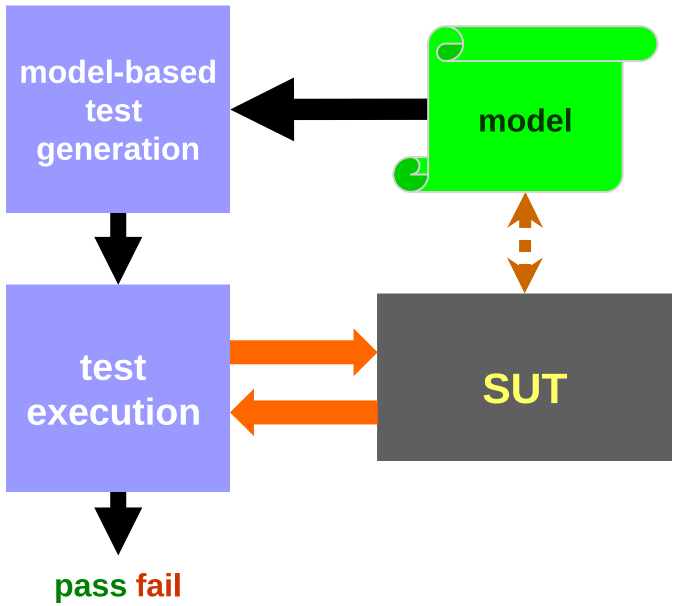
MBT Tester

system model

specification-based, active, black-box testing of functionality



MBT : Benefits



MBT: next step in test automation

- **Automatic test generation**

  \+ test execution + result analysis

- **More, longer, and diversified test cases**

  more variation in test flow and in test data

- **Model is precise and consistent test basis**

  unambiguous analysis of test results

- **Test maintenance by maintaining models**

  improved regression testing

- **Expressing test coverage**

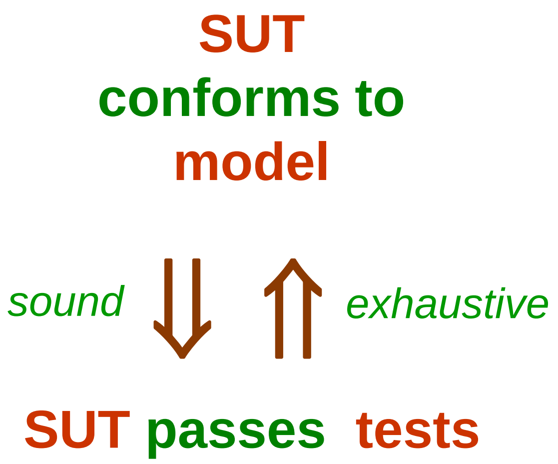
  model coverage

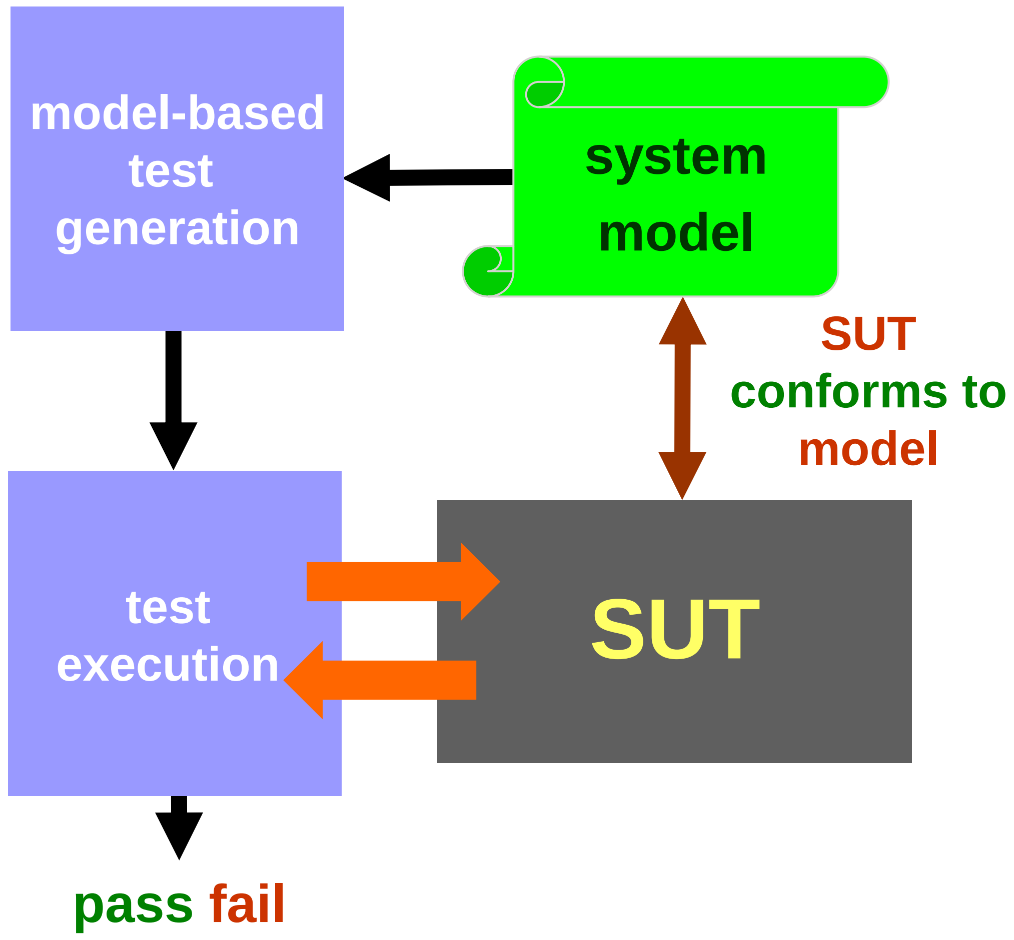
  customer profile coverage

detecting more bugs

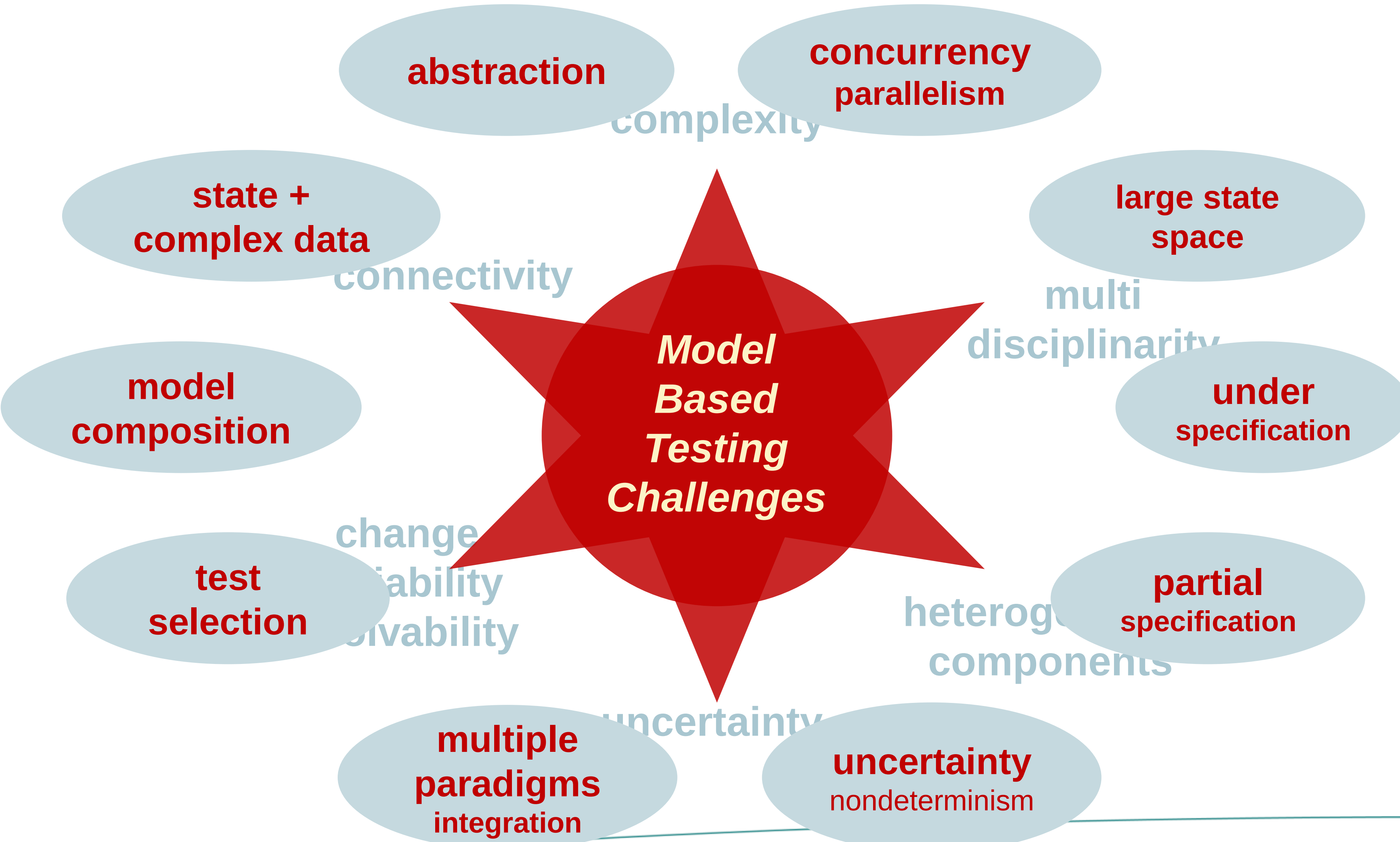
faster and cheaper

Model Based Testing





Model-Based Testing : Challenges



Models

of Systems

Modelling Methods and Formalisms

- Labelled Transition Systems
- Automata
- Formal Languages
- Petri Nets
- Finite-State Machines
  (Mealy - , Moore Machines)
- (First Order) Properties
- Abstract Data Types
- Streams
- Data Flow Models
- . . . . .

- Functions over Time
- Linear Differential Equations
- PDE
- Simulink Models
- Bayesian Networks
- Queueing Networks
- Fault Trees
- Programming Language Models
- Drawings
- Clay Models
- Paper model
- . . . .

Modelling Formalism $\neq$ Modelling Languages

Modelling Systems

- **Traditional systems theory**: (piecewise-) continuous functions of time; analysis and control with ordinary and partial differential equations
- Nowadays, **Digital systems:** discrete, event-driven

models

|

STATIC <----> DYNAMIC

|

TIME-VARYING <---> TIME-INVARIANT

|

LINEAR <---> NONLINEAR

|

CONTINUOUS-STATE <---> DISCRETE-STATE

|

TIME-DRIVEN <---> EVENT-DRIVEN

|

DETERMINISTIC <---> STOCHASTIC

|

DISCRETE-TIME <---> CONTINUOUS-TIME

Cassandras, Lafortune:
Introduction to Discrete Event Systems

Modelling Systems

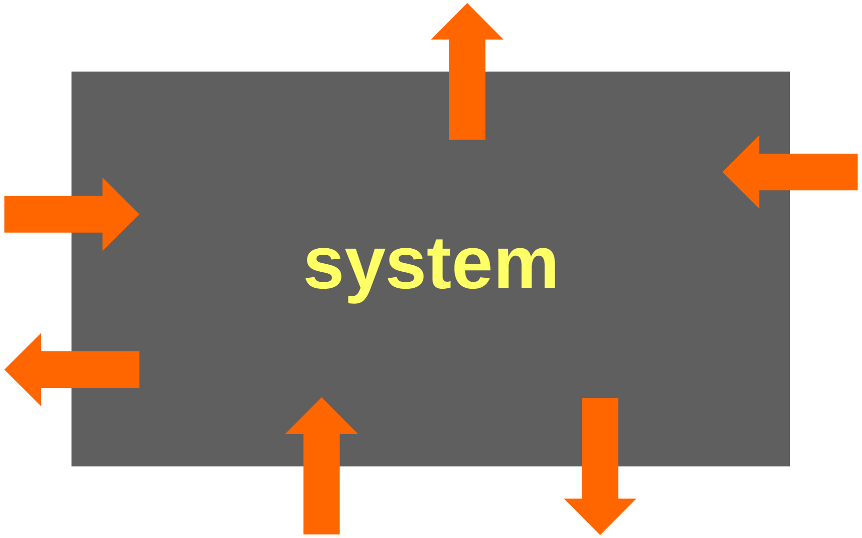
Model ≠ System

- **System** : "something real" (*realization*)
- **Model** : an "abstraction"
- Model : "*any representation of a system not being the system itself*" (Edward Lee)
- By choosing a model, or a class of models, or a modeling formalism, you determine a *view* on the system (and restrict the properties under consideration)
- *A system is not continuous or discrete, a model is*
- One system has many models, for different purposes:
  - quality characteristics
  - abstraction levels
  - prescriptive ←→ descriptive
  - black-box / functional ←→ white-box / structural
  - . . . . .

System Modelling for MBT

Our systems are digital :

- *discrete, event-driven,*
- *reactive, dynamic,*
- *data-intensive,*
- *black-box*



Typical modelling formalisms:

- automata
- formal languages, grammar
- symbolic transition systems
- (extended) finite-state machine
- petri nets
- *labelled transition system*
- . . . . .

View on a system:

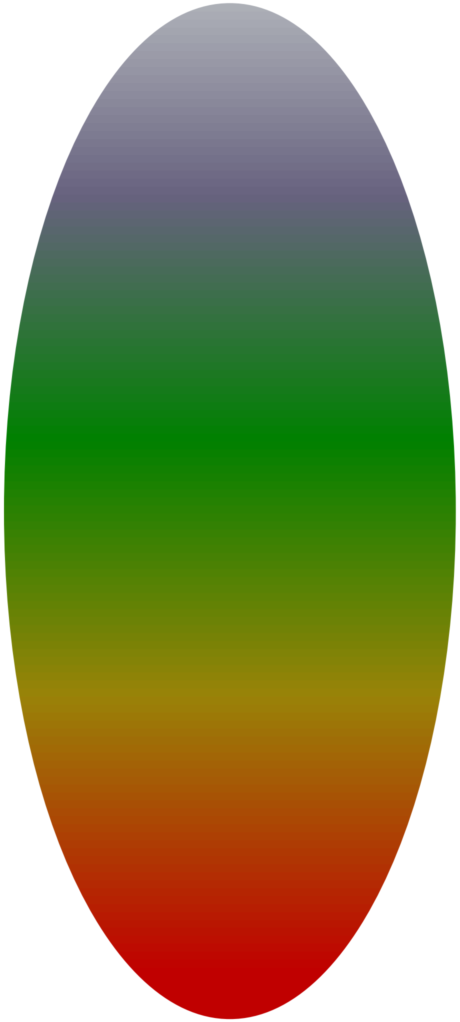
black box, with discrete,

atomic events on interfaces:

- inputs, initiated by environment
- outputs, initiated by the system

Spectrum of Models

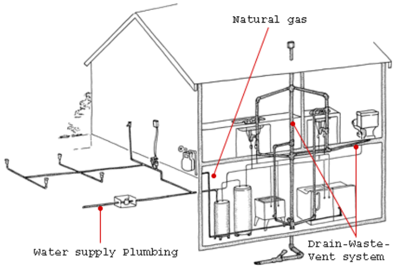
abstract
(test)
models

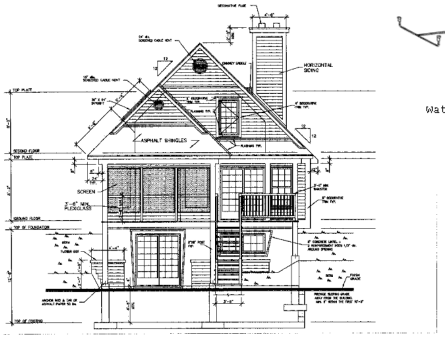


design
models

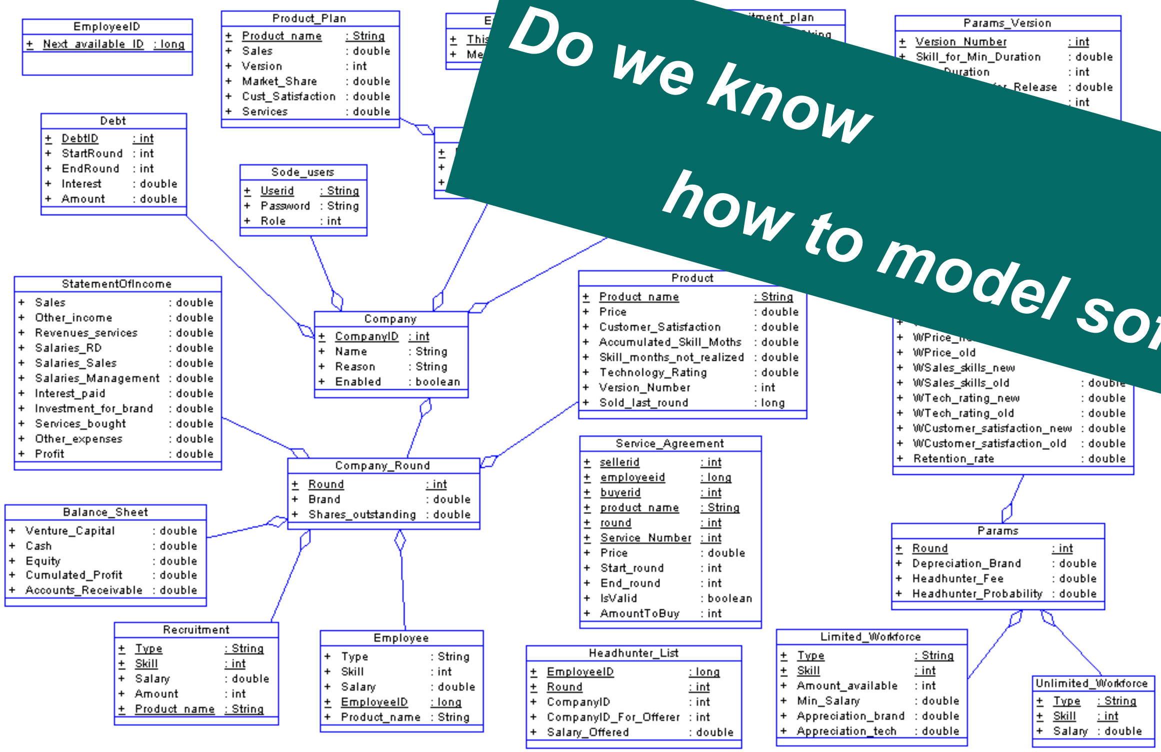
virtualization

realization

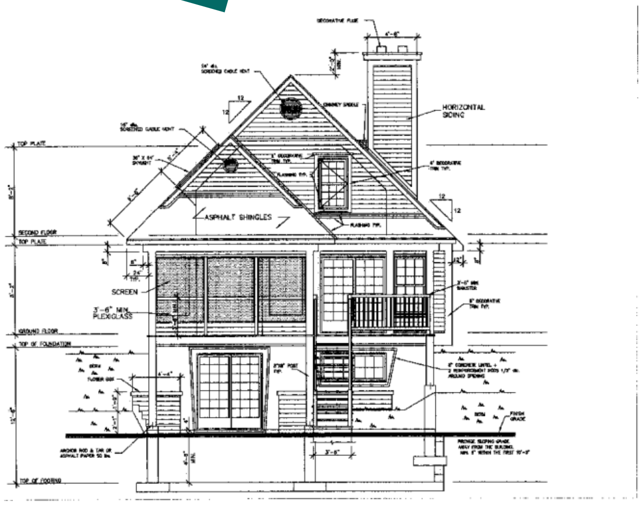




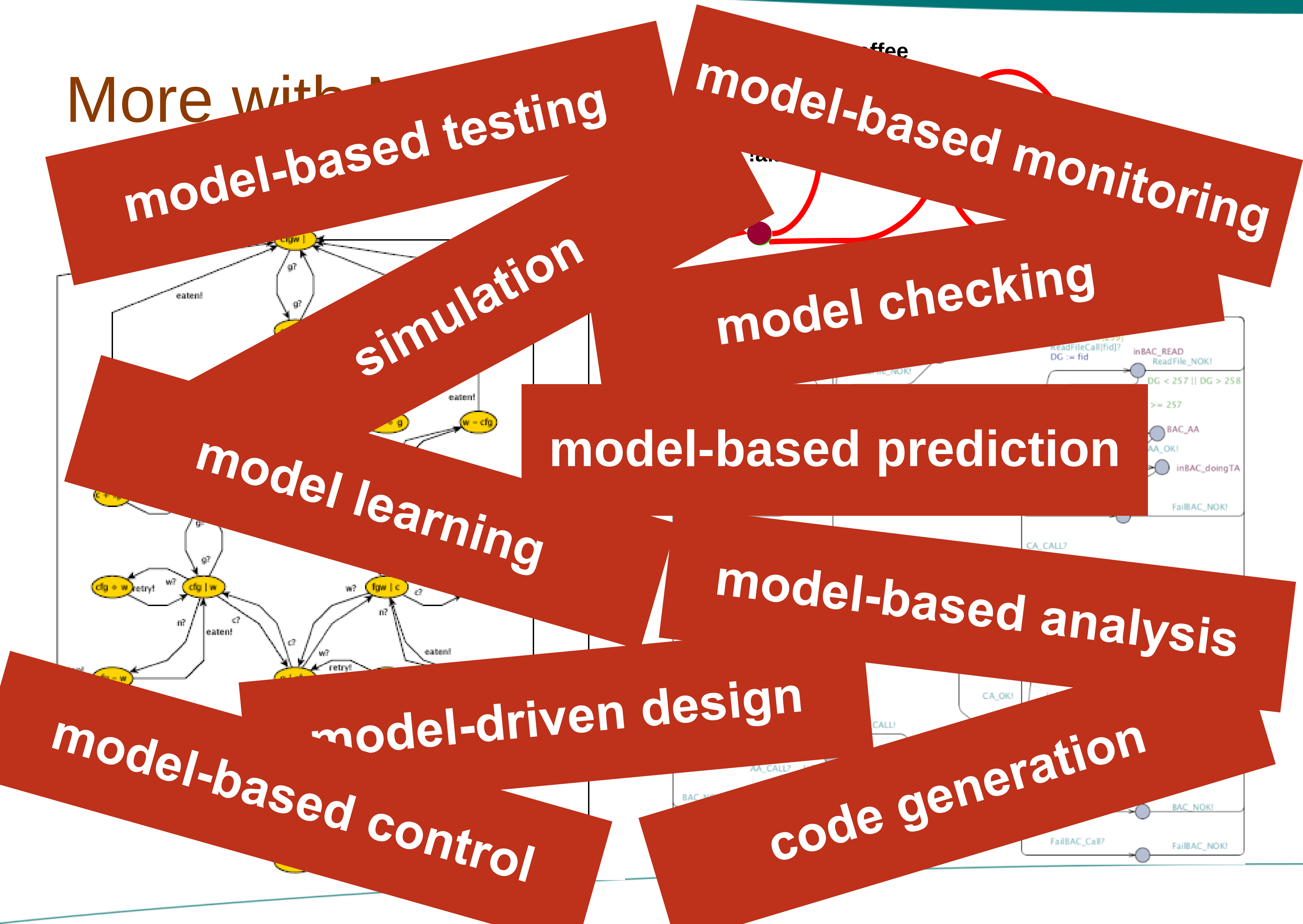
A Software Model



Do we know how to model software ?



More with


model-based testing
model-based monitoring
simulation
model checking
model learning
model-based prediction
model-based analysis
model-driven design
model-based control
code generation

Code Generation from a Model

men van selectie, configuratie dus; de modelgedreven gemeenschap denkt typisch in termen van creatie, customization dus. Er zou echter geen verschil moeten zijn. Wat ik wil duidelijk maken, is dat beide werelden

'Modellering zonder codegeneratie is zinloos'

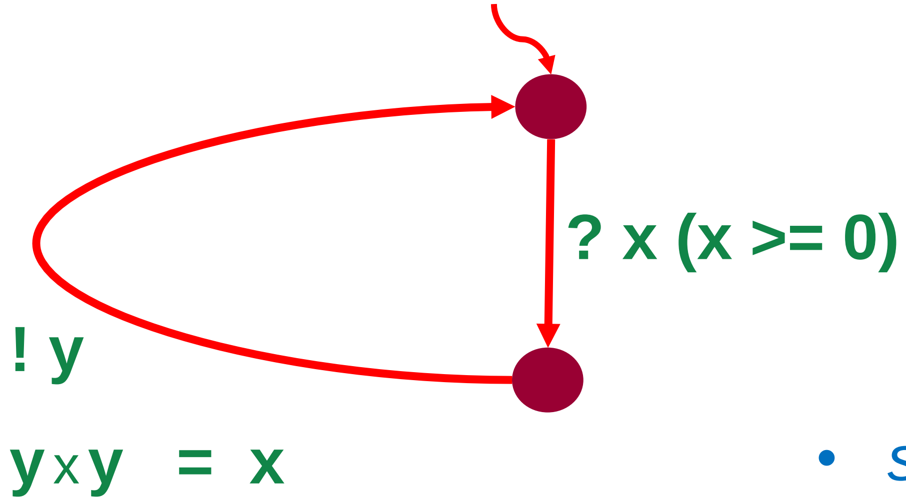
heel goed zijn te combineren. Sommige variabiliteit is te vatten in configuratie, andere variabiliteit in customization. Sommige dingen zijn het best uit te drukken met featuremodellering, andere zijn het best te representeren met domeinspecifieke talen.'

De combinatie is nog verder door te voeren. 'Configuratie is niet alleen te gebruiken om parameters in te stellen, maar ook om modellen te veranderen', licht Völter toe. 'Voor elke feature die je niet selecteert, vervalt er een aantal toestanden in je toestandsdiagram. Zo komen configuratie en customization samen, wat alles een stuk simpeler maakt. Een domeinspecifieke taal is beknopt, exact en high-level.'

A model is more (*less*) than code generation:

- views
- abstraction
- testing of aspects
- verification and validation of aspects

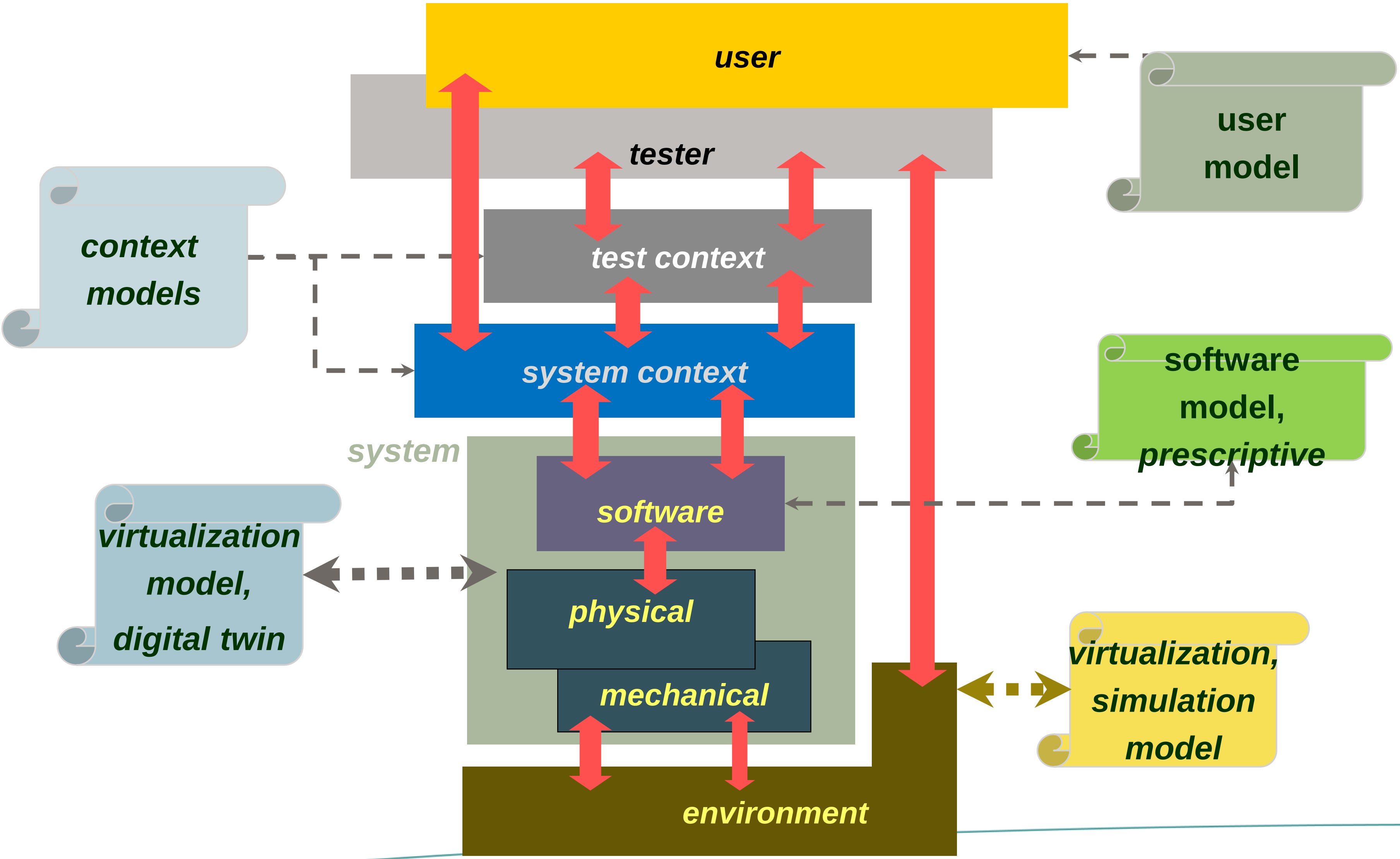
Code Generation from a Model Not Always Possible



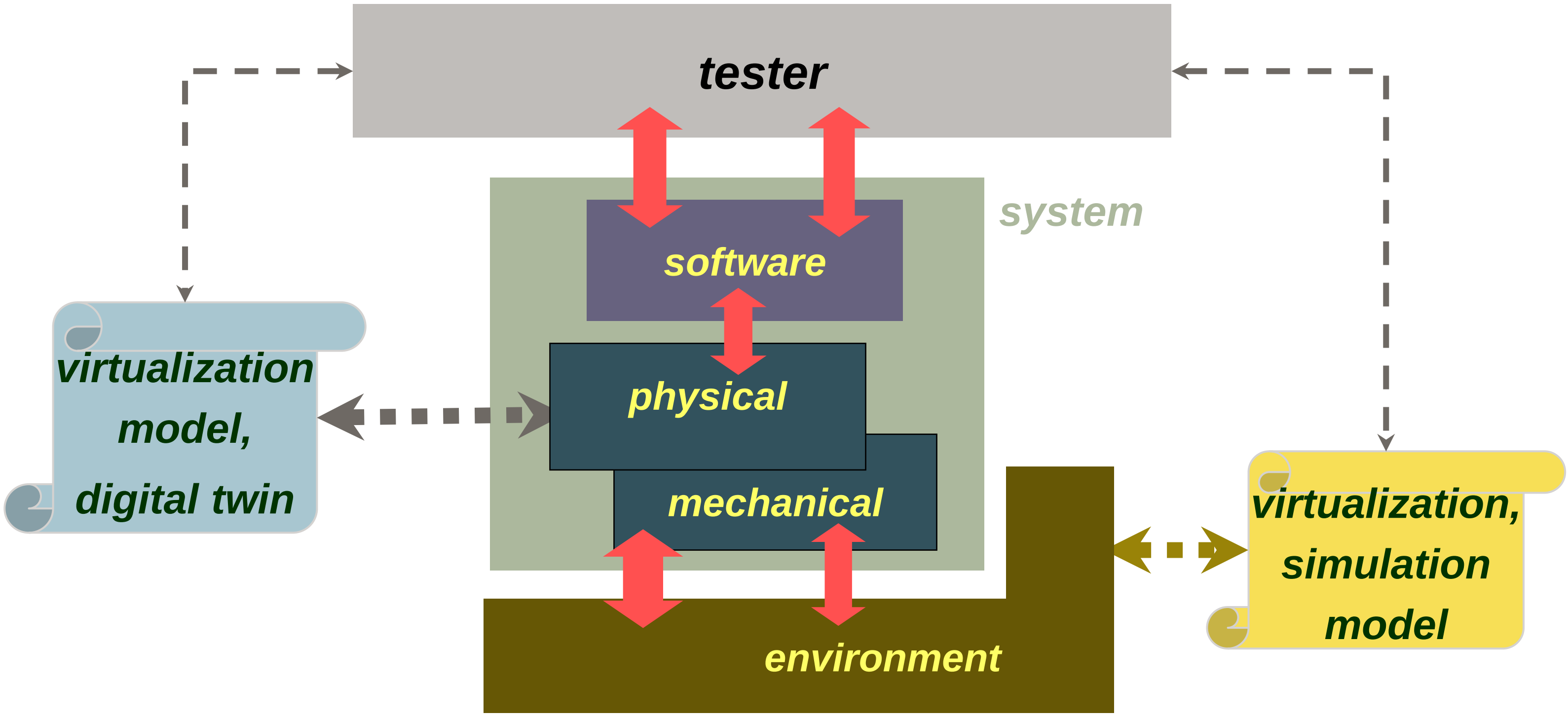
model of $\sqrt{x}$

- *specification of **properties** rather than construction*
- ***under-specification***
- ***non-determinism***

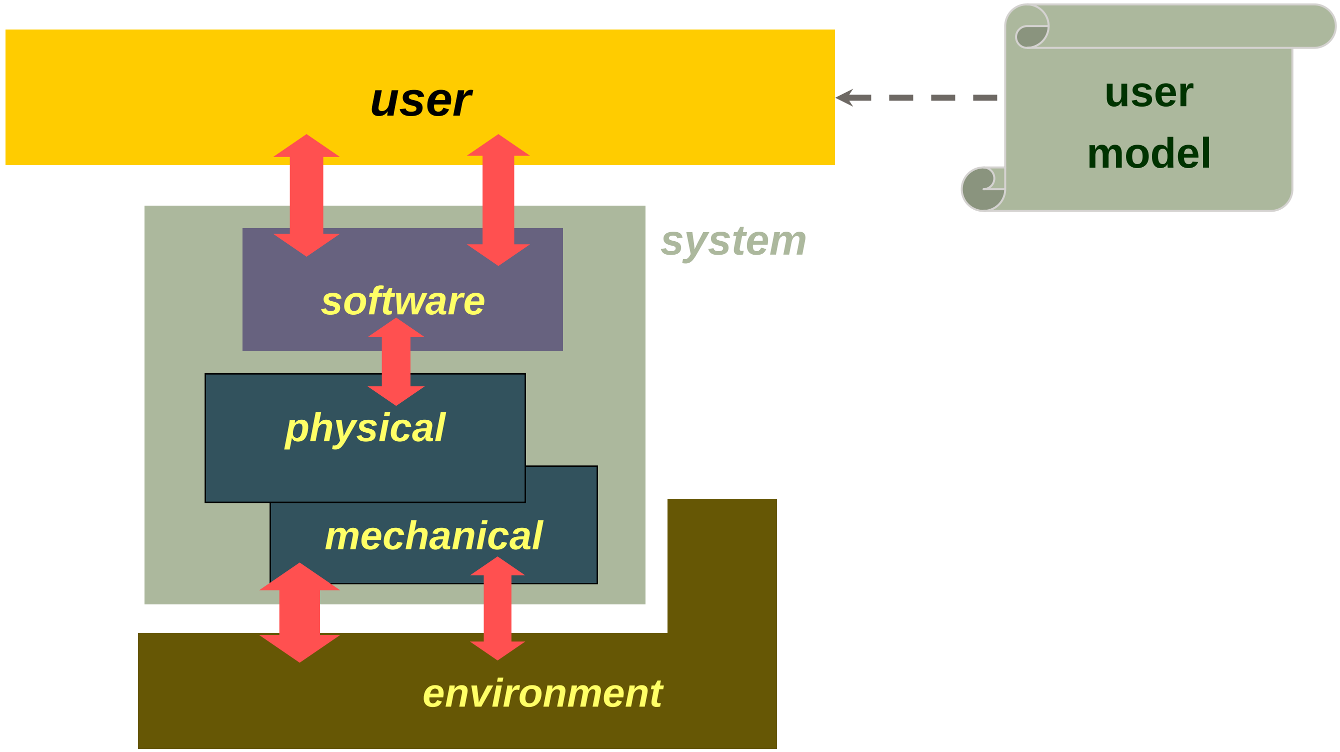
Models for Testing



Models for Testing : Stubs



Models for Testing : Usage Profile



A Choice of Modelling Formalism :

Labelled Transition Systems

Models: Labelled Transition Systems

Labelled Transition System: $\langle$ **S**, **L$_I$**, **L$_U$**, **T**, **s$_0$** $\rangle$

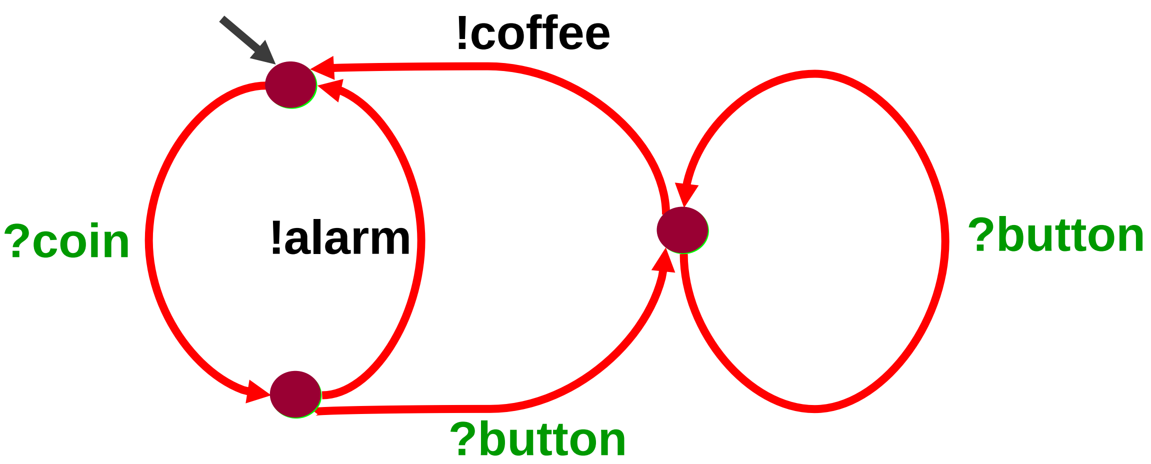
- states
- input labels
- output labels
- transitions $T \subseteq S \times (L \cup \{\tau\}) \times S$
- initial state $s_0 \in S$

? = input

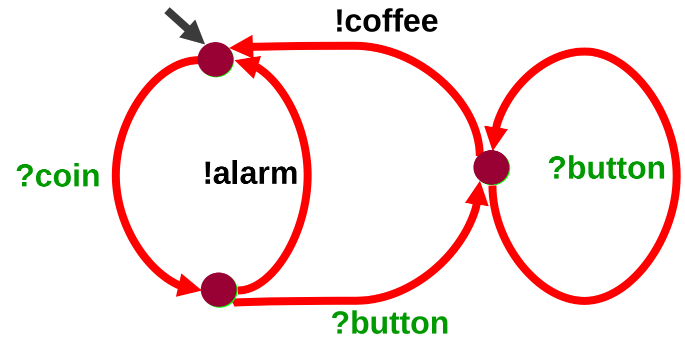
! = output

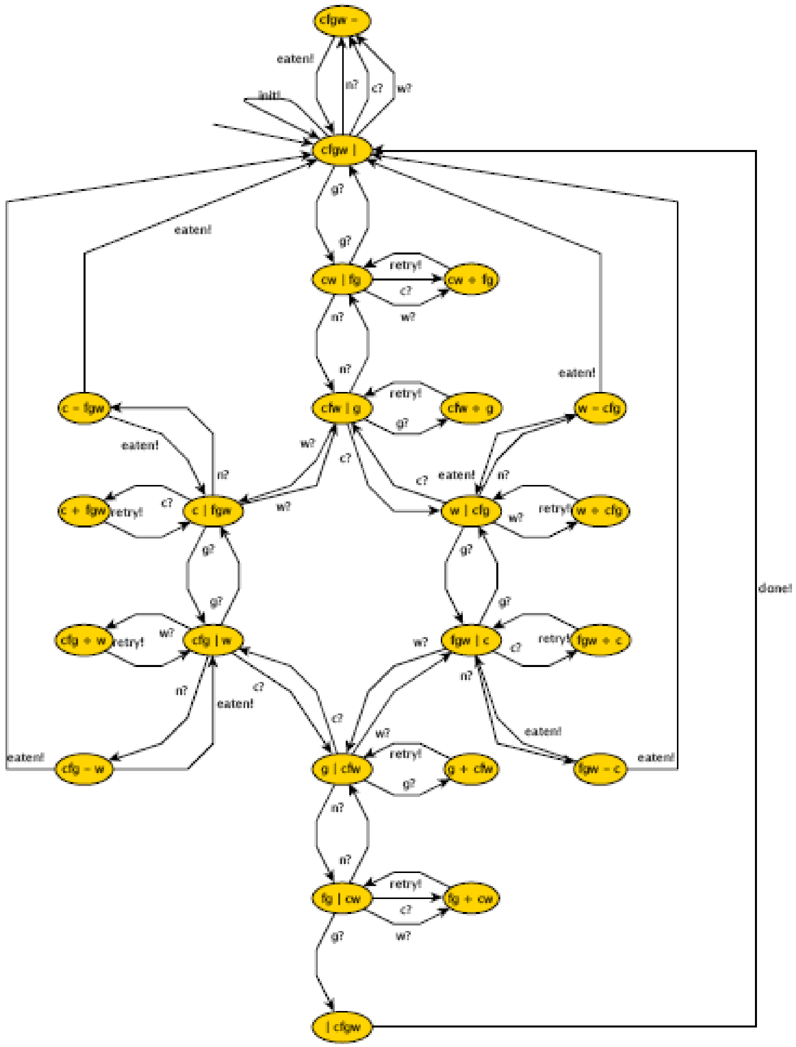
$L = L_I \cup L_U$

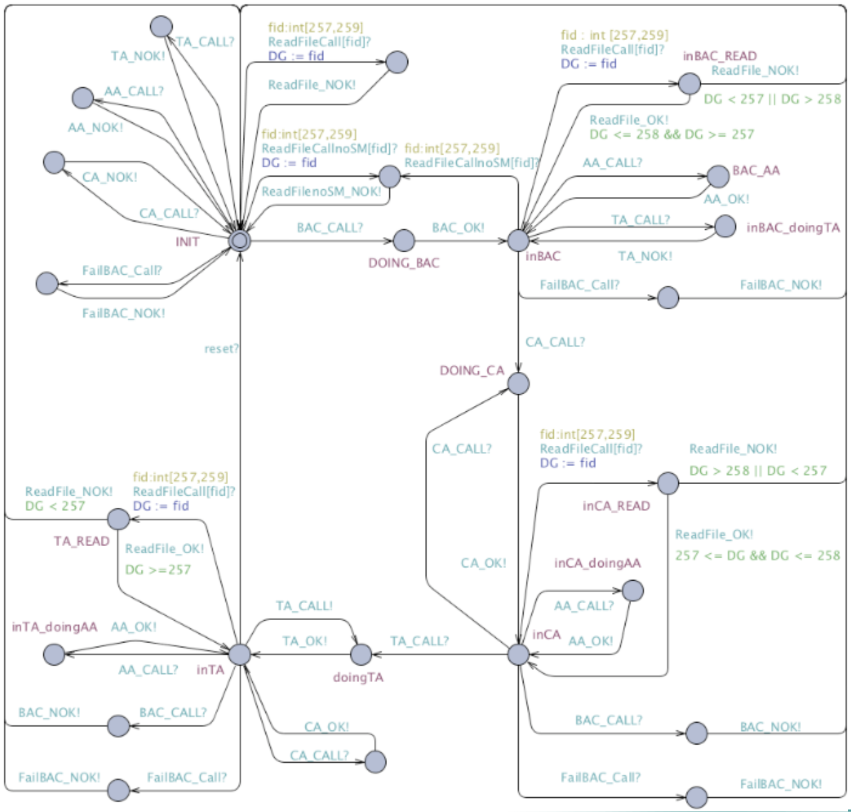
$L_I \cap L_U = \emptyset$



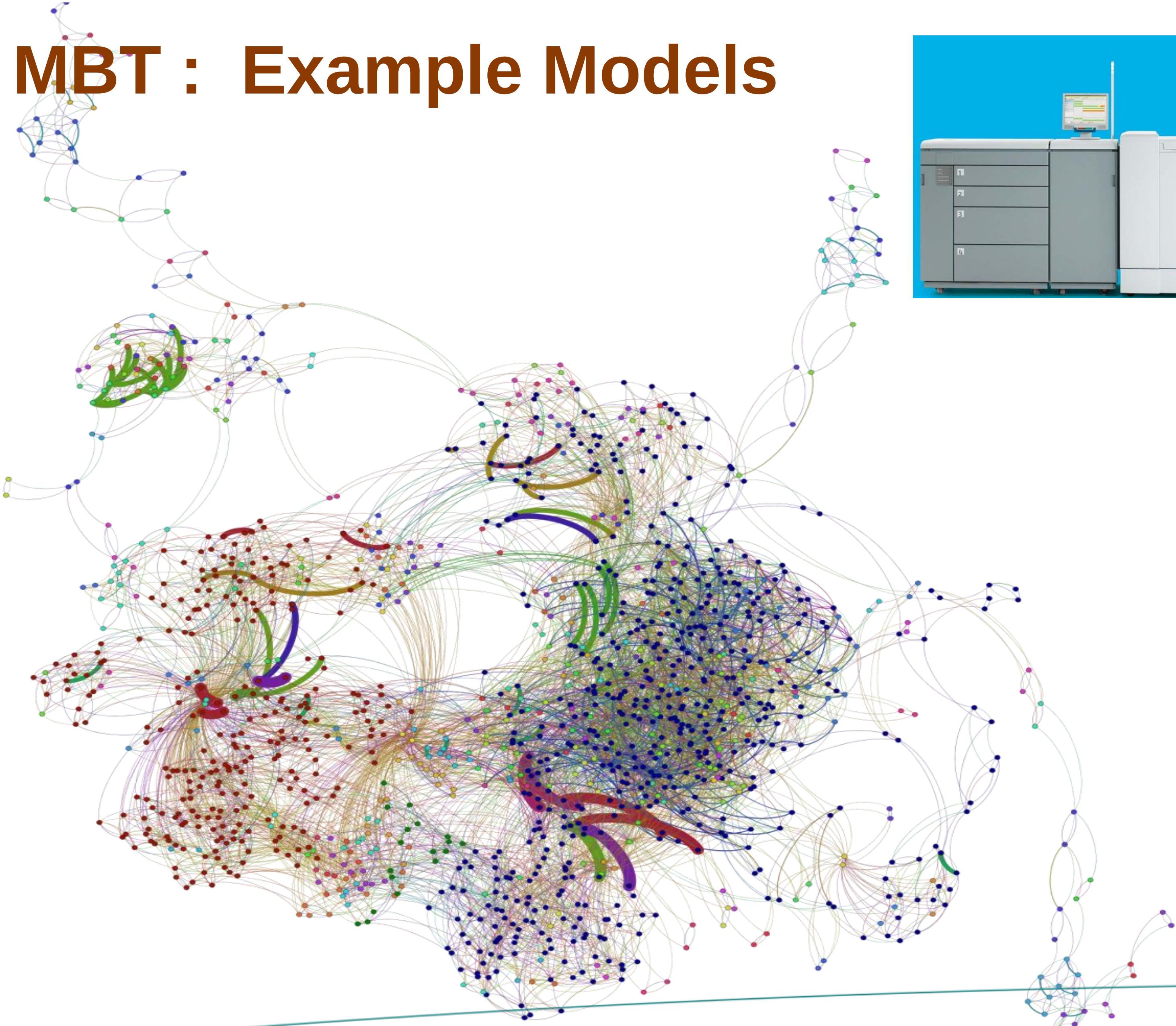
MBT : Example Models



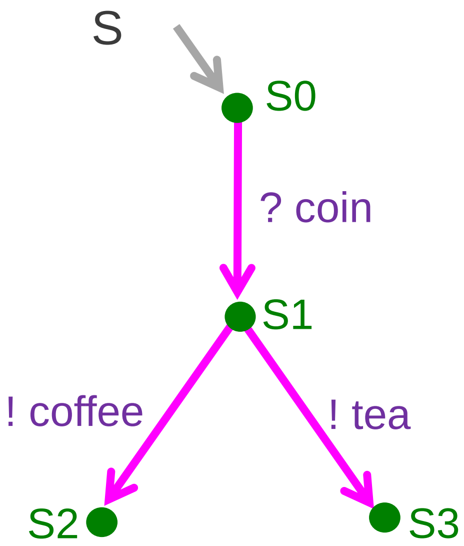




MBT : Example Models



LTS : Representation



- Explicit : ⟨ { S0, S1, S2, S3 } ,
  { ?coin } ,
  { !coffee, !tea } ,
  { (S0, ?coin, S1),
  (S1, !coffee, S2),
  (S1, !tea, S3) } ,
  S0 ⟩

- Transition tree / graph

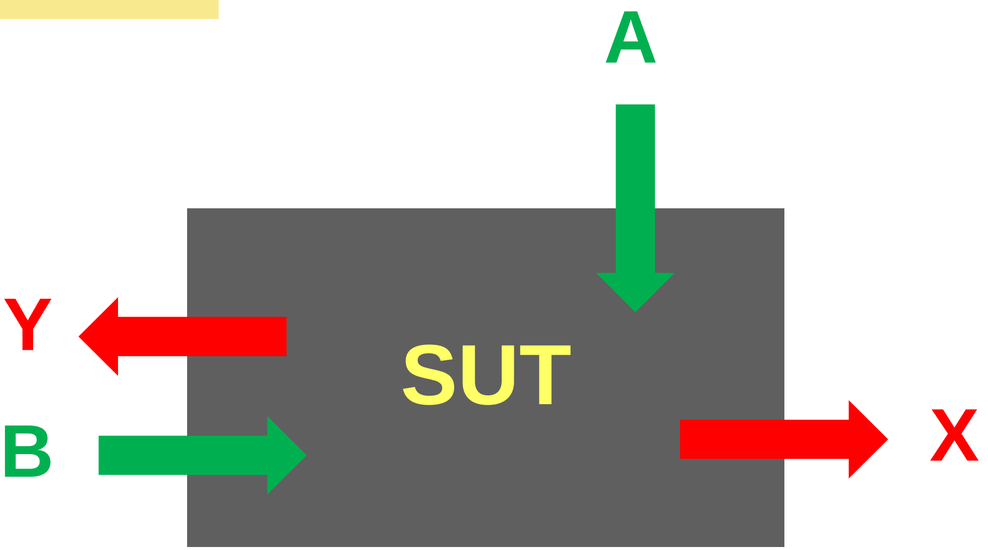
- Language :

S ::= Coin >-> (Coffee ## Tea)

LTS : A Black-Box View on Systems

*A black-box view on a system starts with its **interfaces***

- **Inputs Channels: A; B**
- **Output Channels: X; Y**



STS : A Black-Box View on Systems

Symbolic Transition System

channels with messages :

Symbolic Trabs

- **Inputs Channels:**

  A :: Int; B :: Struct

- **Output Channels:**

  X :: Int; Y :: String

SUT

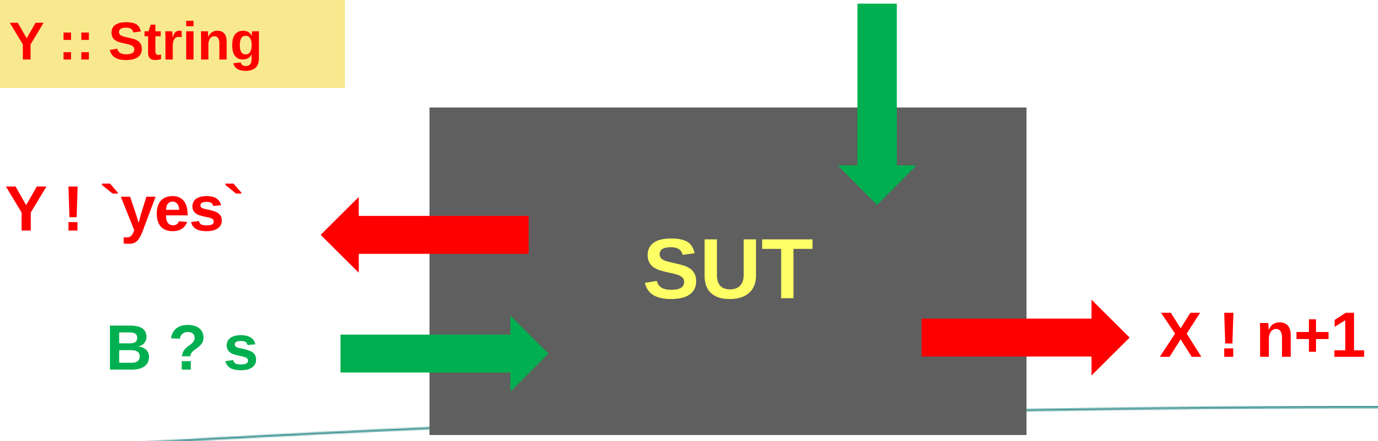
real, black-box system

communicating with its

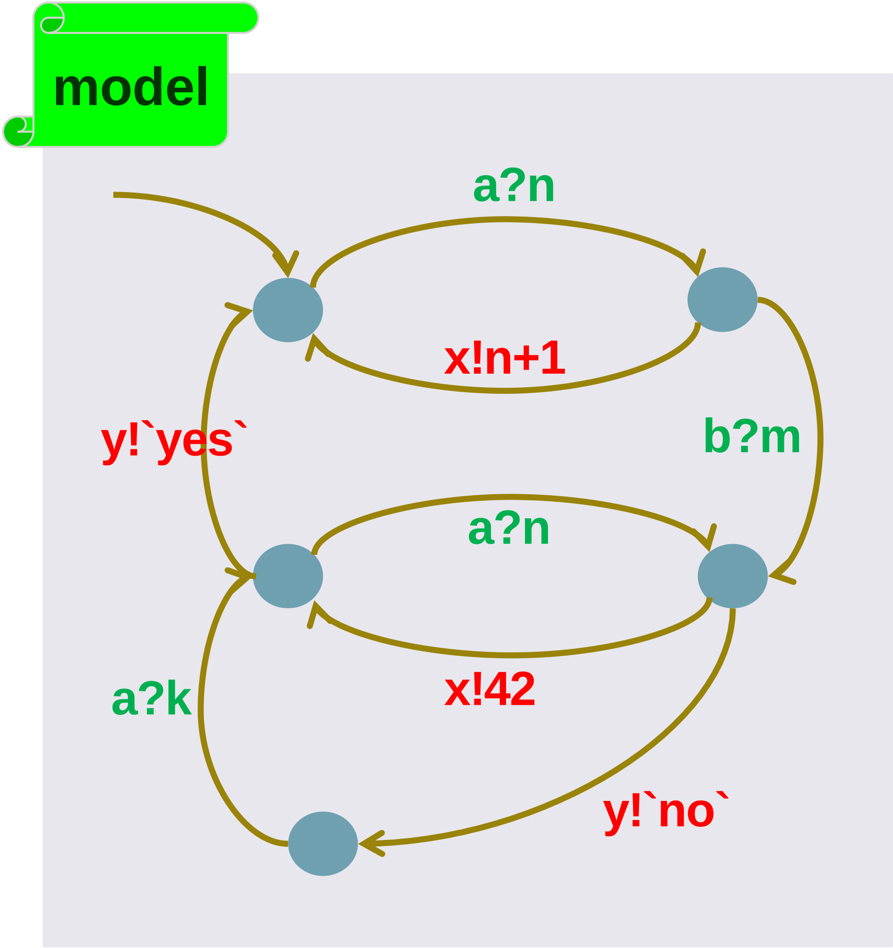
environment

via messages on input- and

output channels



A View on Systems and Models



black-box
system view

Abstracted from :

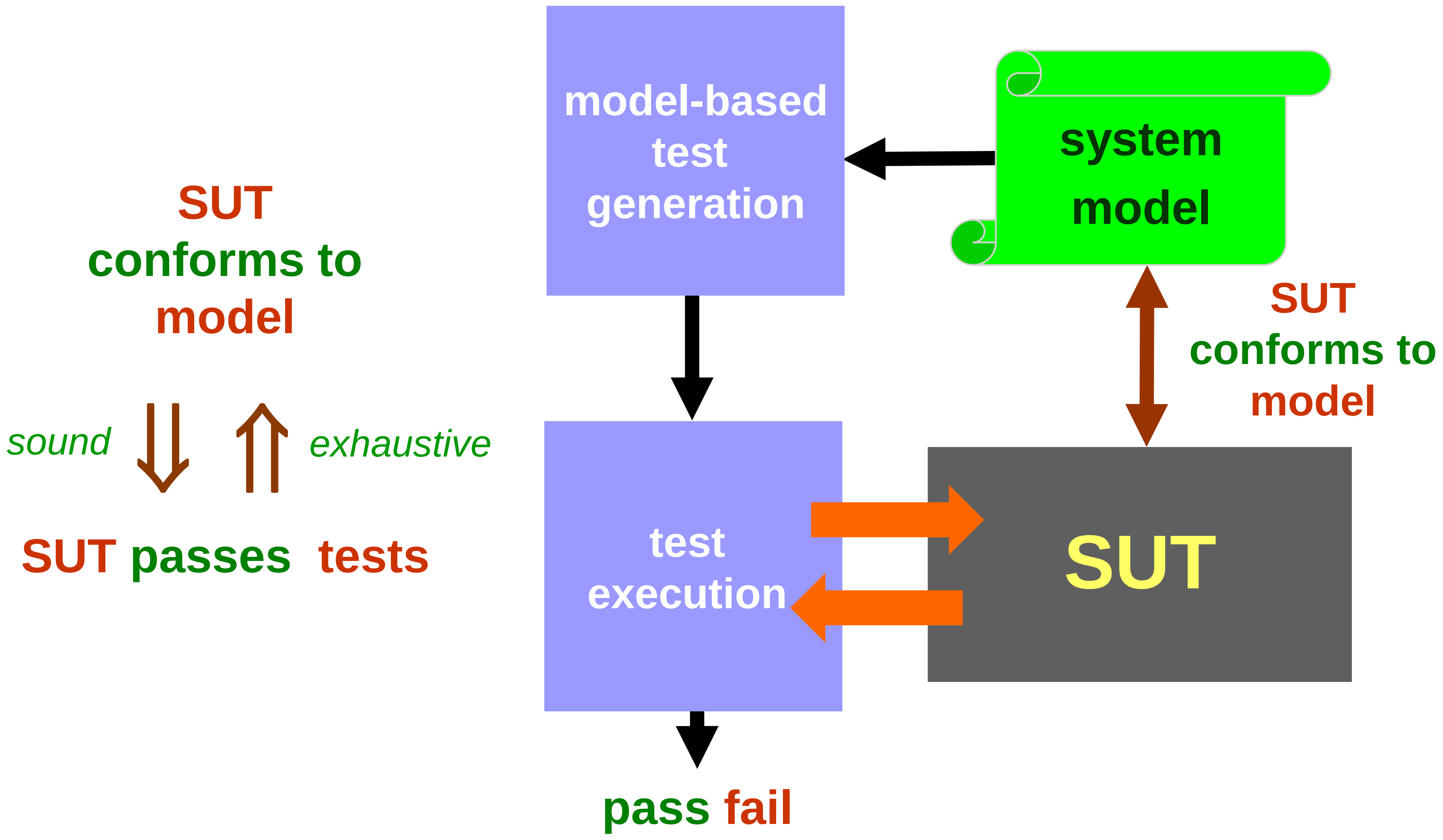
- real-time
- probabilities
- derivatives (hybrid)

modelled as
a symbolic
transition system

Model-Based Testing

with Labelled Transition Systems

Model Based Testing

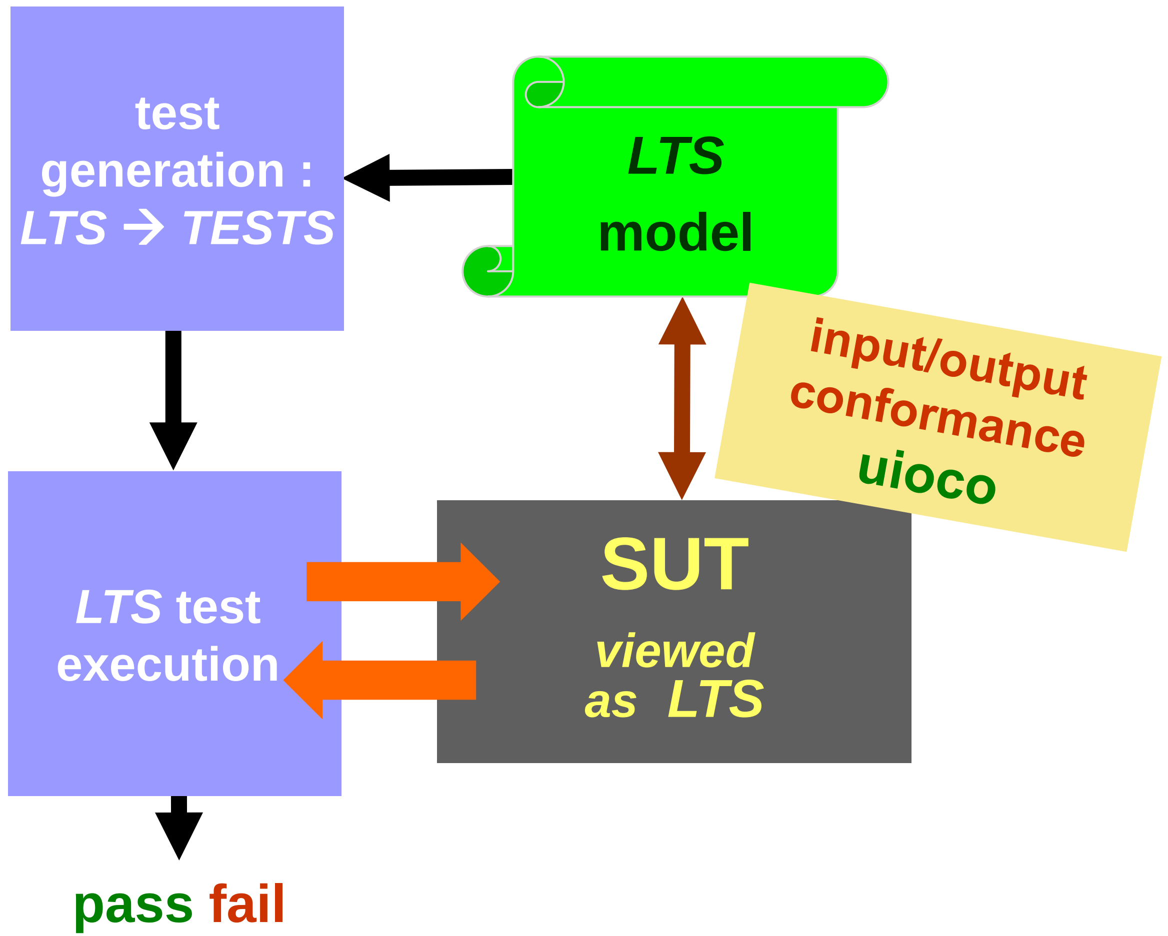


Model Based Testing with LTS

SUT *behaving as LTS*
conforms to
***LTS* Model**

sound ⇓ ⇑ *exhaustive*

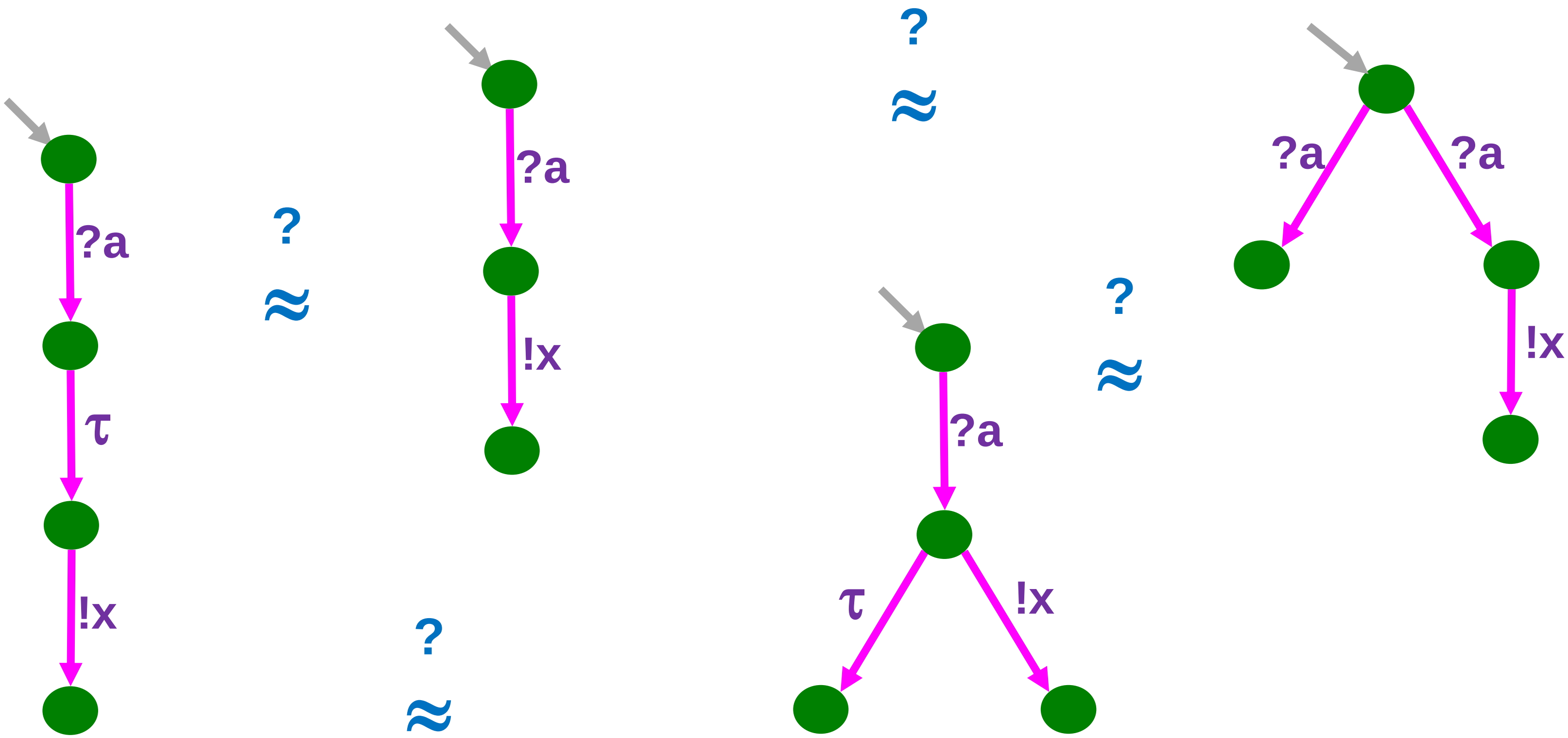
SUT passes generated tests



Equivalences

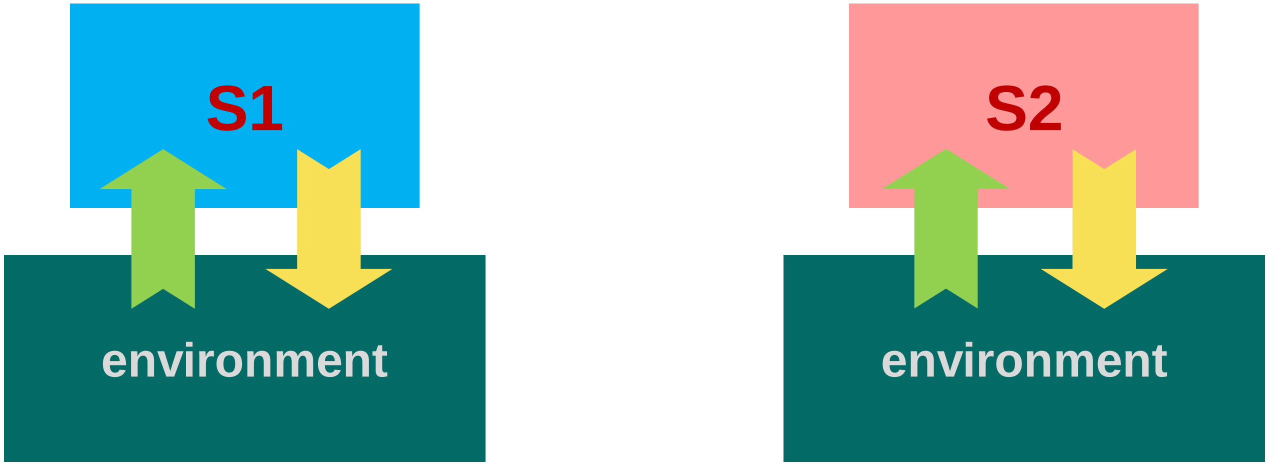
on Labelled Transition Systems

Observable Behaviour



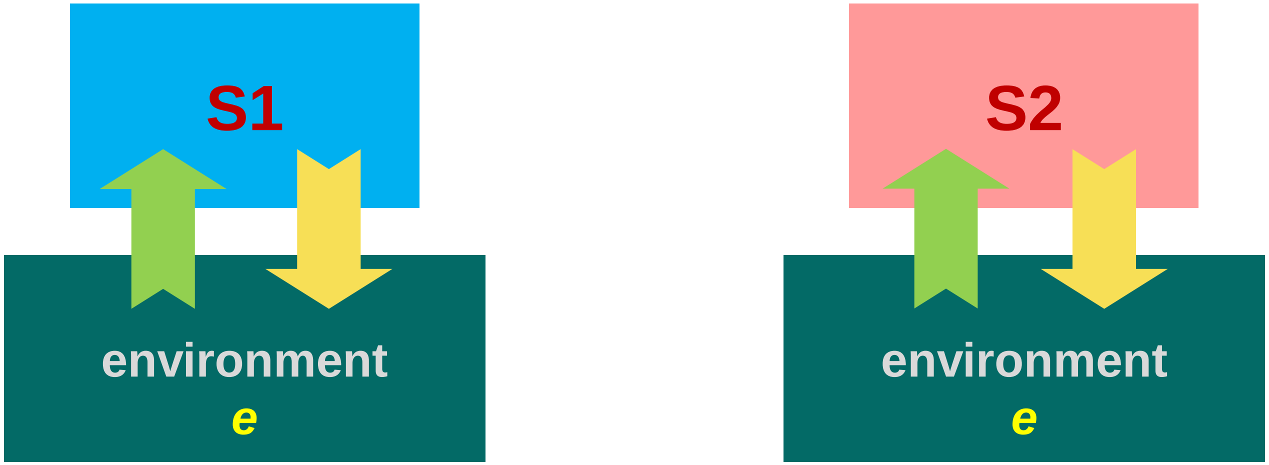
" Some transition systems are more equal than others "

Comparing Transition Systems



- Suppose an environment interacts with the systems:
  - the environment tests the system as black box by observing and actively controlling it;
  - the environment acts as a tester;
- Two systems are equivalent iff they pass the same tests.

Comparing Transition Systems



S1 $\approx$ **S2** $\Leftrightarrow$ $\forall$ ***t*** $\in$ **T .** ***obs*** **(** ***t*, S1)** **=** ***obs*** **(*t*, S2)**

↓ ↓

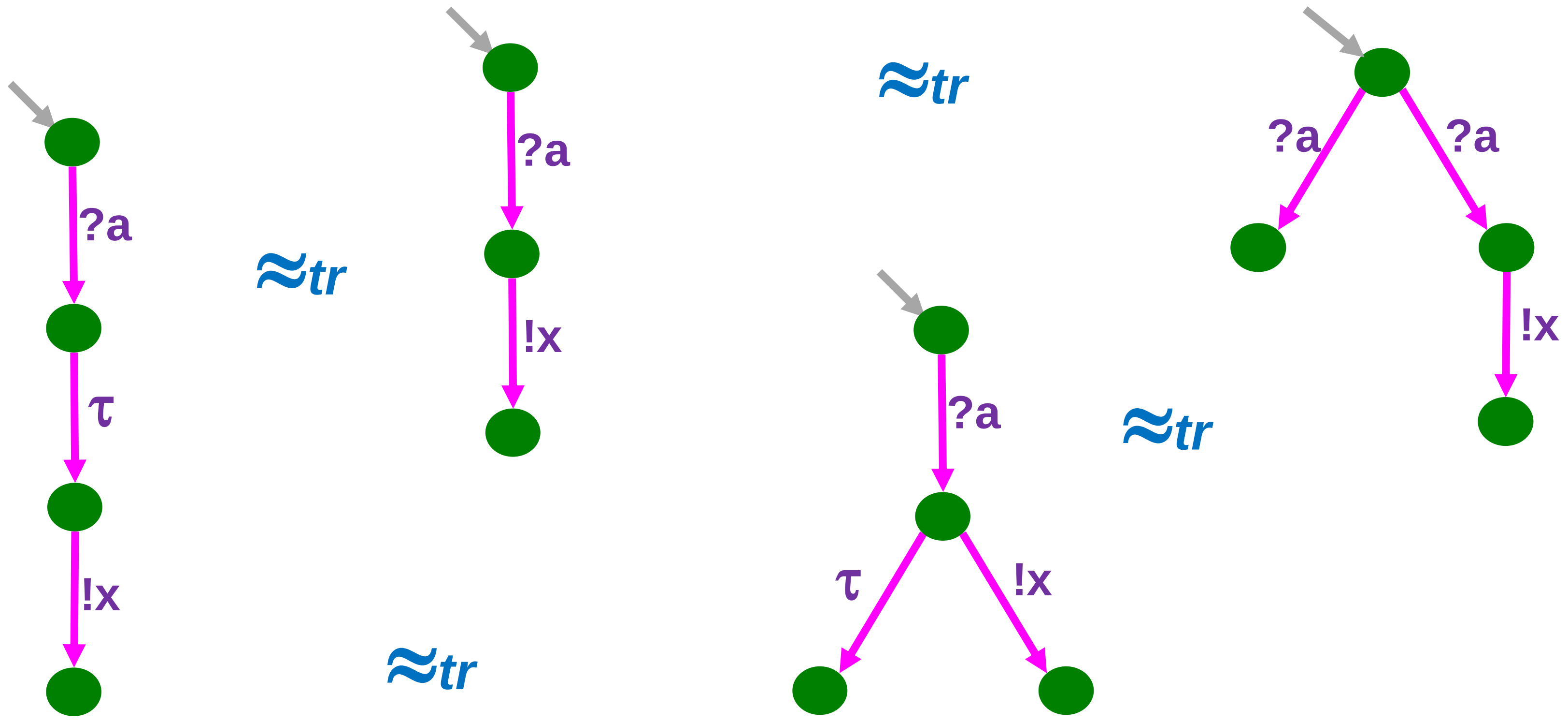
? **?**

Comparing Transition Systems

$$S1 \approx_{tr} S2 \quad \Leftrightarrow \quad \textit{traces}(S1) = \textit{traces}(S2)$$

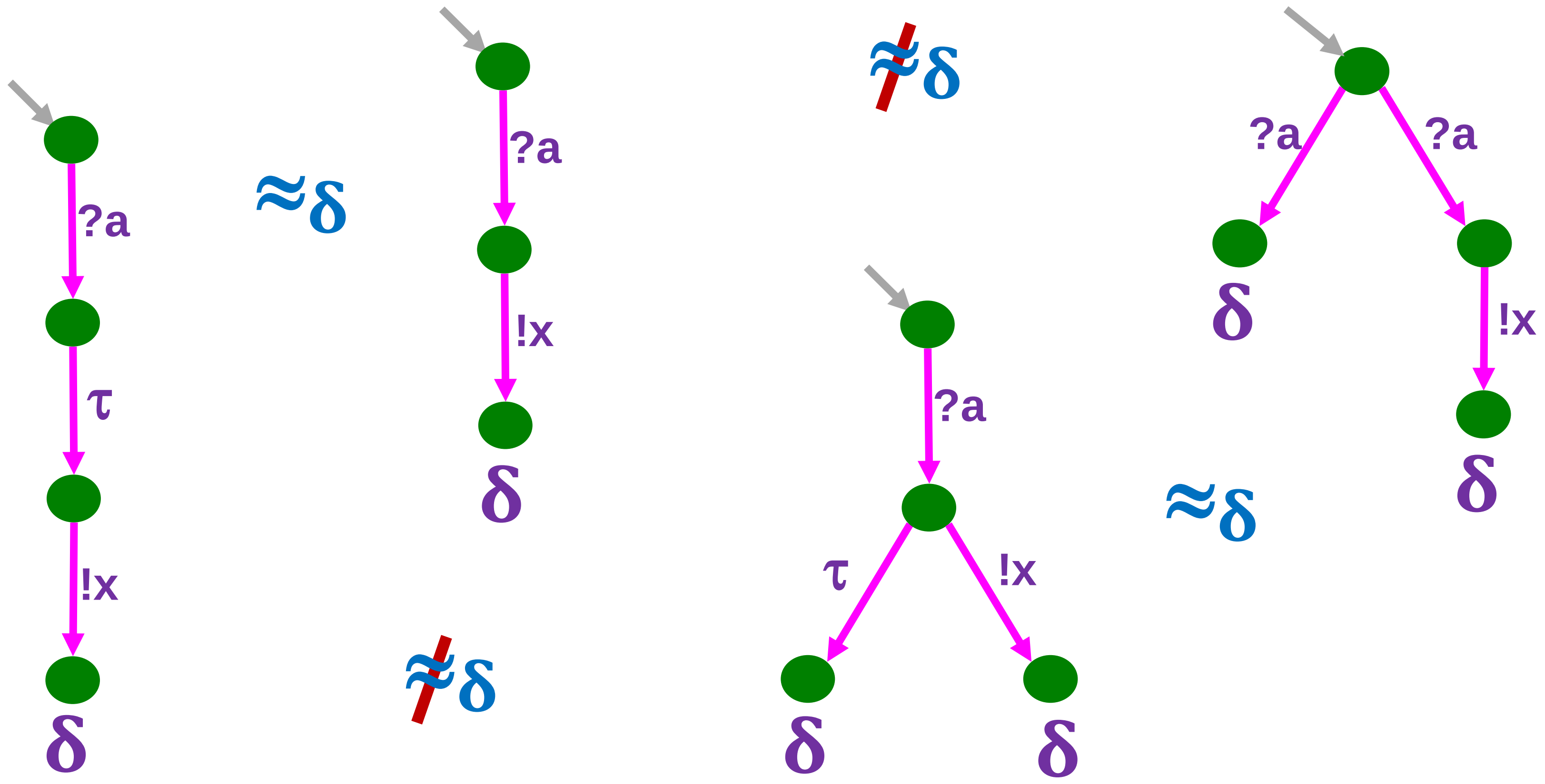
Traces: $$\textit{traces}(s) = \{ \sigma \in L^* \mid s \overset{\sigma}{\Longrightarrow} \}$$

Trace Equivalence



for all: ***traces* (.) = { ε , ?a , ?a.!x }**

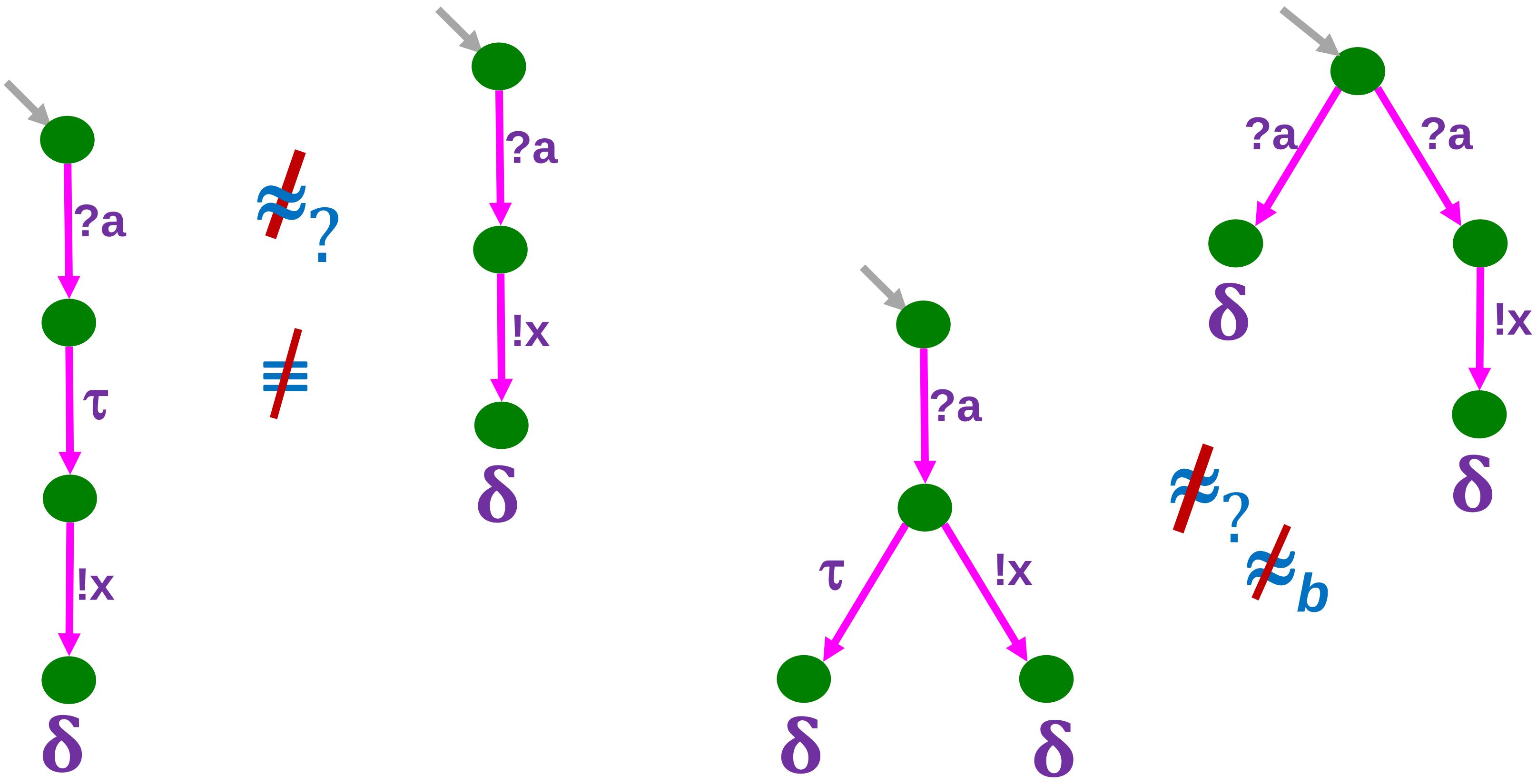
Equivalence with δ



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

$Straces\,(\,\mathbf{s}\,) \quad = \quad \{\ \sigma \in (L \cup \{\delta\})^* \mid s \overset{\sigma}{\Longrightarrow}\ \}$

Stronger Equivalences



MBT : Equivalences

1. *equivalent if they have the same behaviours*

   $$S1 \approx S2 \iff \mathit{traces}(S1) = \mathit{traces}(S2)$$

   $$S1 \approx S2 \iff \mathit{Straces}(S1) = \mathit{Straces}(S2)$$

2. *equivalent if they pass the same tests*

   $$S1 \approx S2 \iff \forall\, t \in T\,.\; S1 \text{ passes } t \iff S2 \text{ passes } t$$

3. *equivalent if they allow the same implementations*

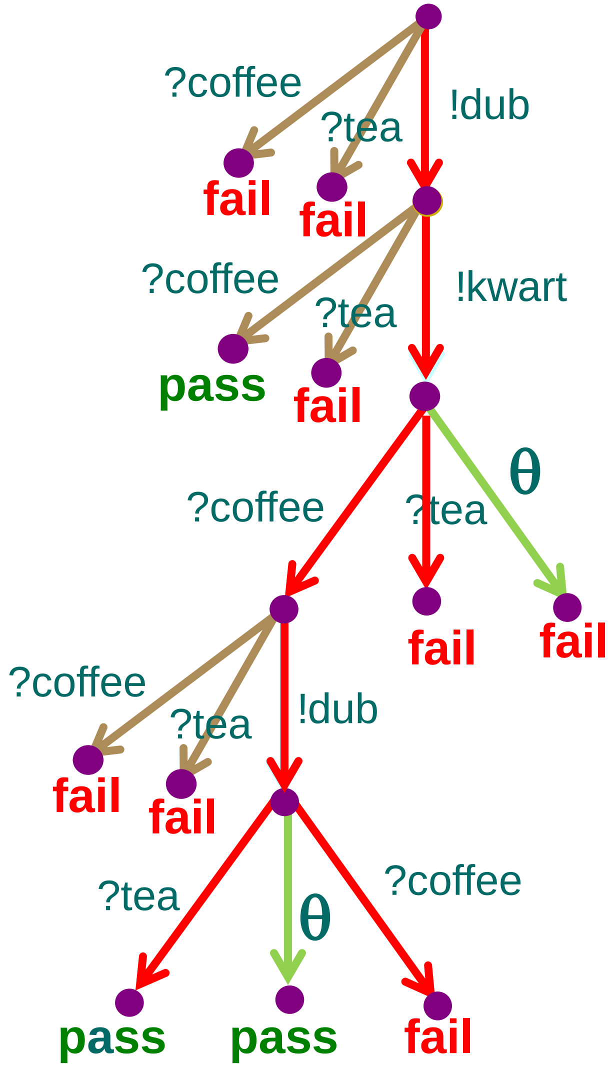
   $$S1 \approx S2 \iff \mathit{Imp}(S1) = \mathit{Imp}(S2)$$

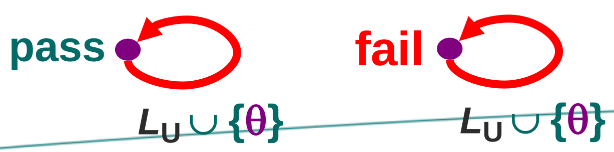
A Choice of Test Cases

model of a test case

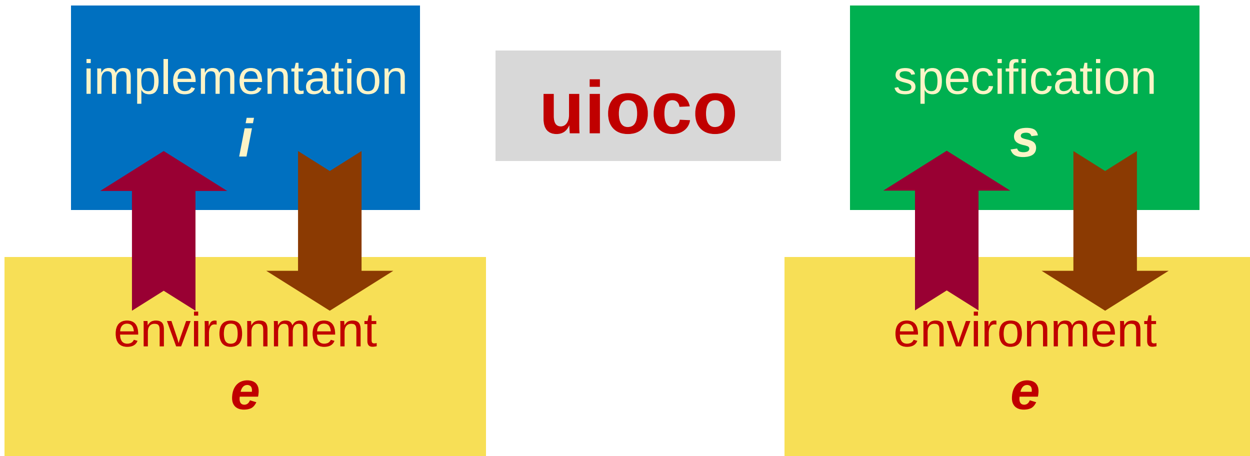
= labelled transition system

- labels in $L_I \cup L_U \cup \{ \theta \}$
- 'quiescence' / 'time-out' label θ
- tree-structured
- finite, deterministic
- sink states **pass** and **fail**
- from each state $\neq$ **pass**, **fail** :
  - either one input **!a** and all outputs **?x**
  - or all outputs **?x** and θ





A Choice of Implementation Relation: *uioco*



i ∈ *IOTS*

s ∈ *LTS*

uioco $\subseteq$ *IOTS* (L_I, L_U) **x** *LTS* (L_I, L_U)

IOTS: input enabled *LTS*

i **uioco** ***s***

i *is a conforming implementation of* **s**

MBT with Labelled Transition Systems

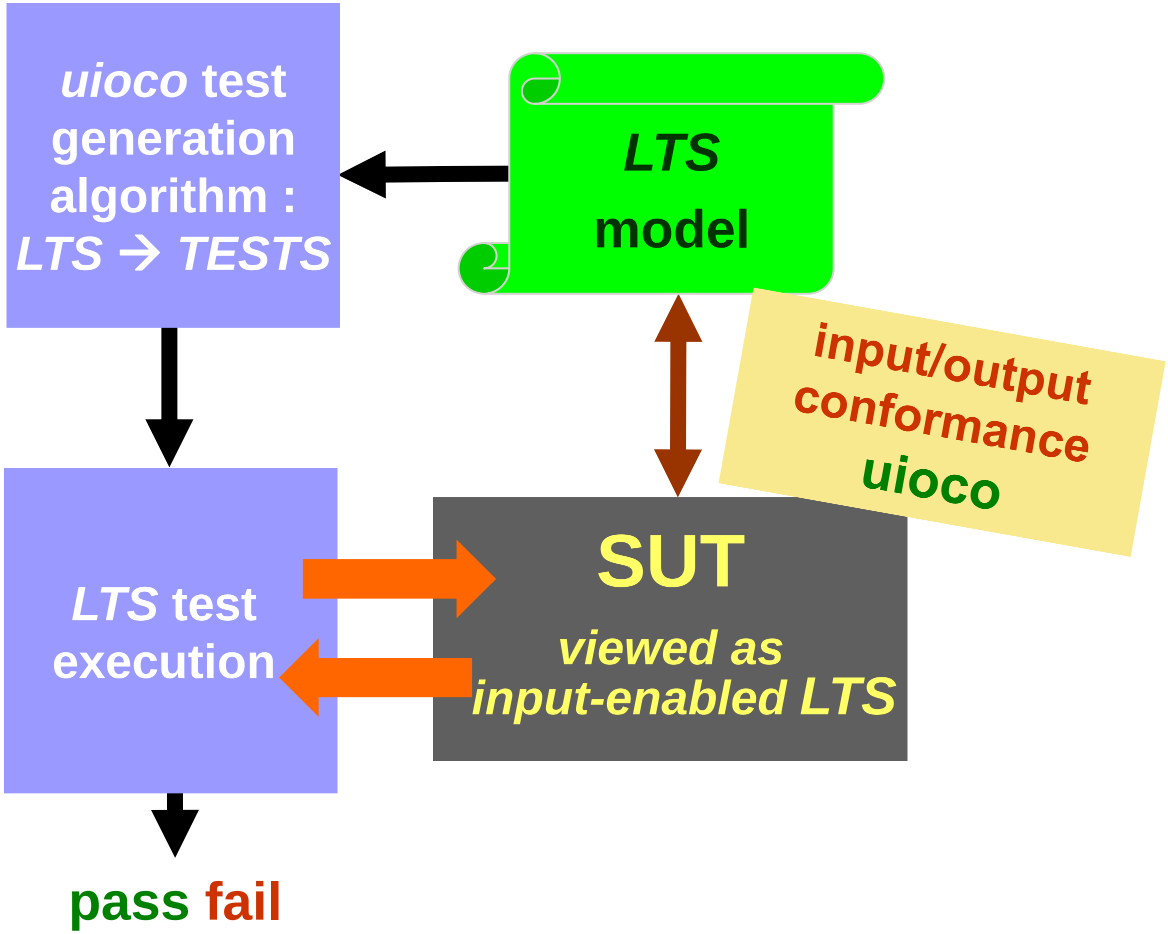
and **uioco**

Model Based Testing with LTS and *uioco*

SUT *behaving as input-enabled LTS*
conforms to
***LTS* Model**

sound ⇓ ⇑ *exhaustive*

SUT passes generated *uioco* tests



Input/Output Conformance : *uioco*

$i \ \textbf{uioco} \ s \ =_{def} \ \forall \sigma \in \mathit{Utraces}(s) : \ \mathit{out}(i \text{ after } \sigma) \subseteq \mathit{out}(s \text{ after } \sigma)$

s is a Labelled Transition System

i is (assumed to be) an input-enabled LTS

$s \text{ after } \sigma \ = \ \{ \ s' \mid s \stackrel{\sigma}{\Longrightarrow} s' \ \}$

$s \text{ refuses } A \ \Leftrightarrow \ \forall \mu \in A \cup \{\tau\} : \ s \stackrel{\mu}{\not\longrightarrow}$

$s \stackrel{\delta}{\longrightarrow} s \ \Leftrightarrow \ s \text{ refuses } L_U$

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

$\mathit{Utraces}(s) \ = \ \{ \sigma \in \mathit{Straces}(s) \mid$
$\forall \sigma_1 \, ?b \, \sigma_2 = \sigma : \ \mathit{not}(\, s \text{ after } \sigma_1 \text{ refuses } \{?b\} \,) \ \}$

$\mathit{out}(P) \ = \ \{ \ !x \in L_U \mid \exists p \in P : p \stackrel{!x}{\longrightarrow} \} \ \cup \ \{ \ \delta \mid \exists p \in P : p \stackrel{\delta}{\longrightarrow} p \ \}$

Input/Output Conformance : *uioco*

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

s is a Labelled Transition System

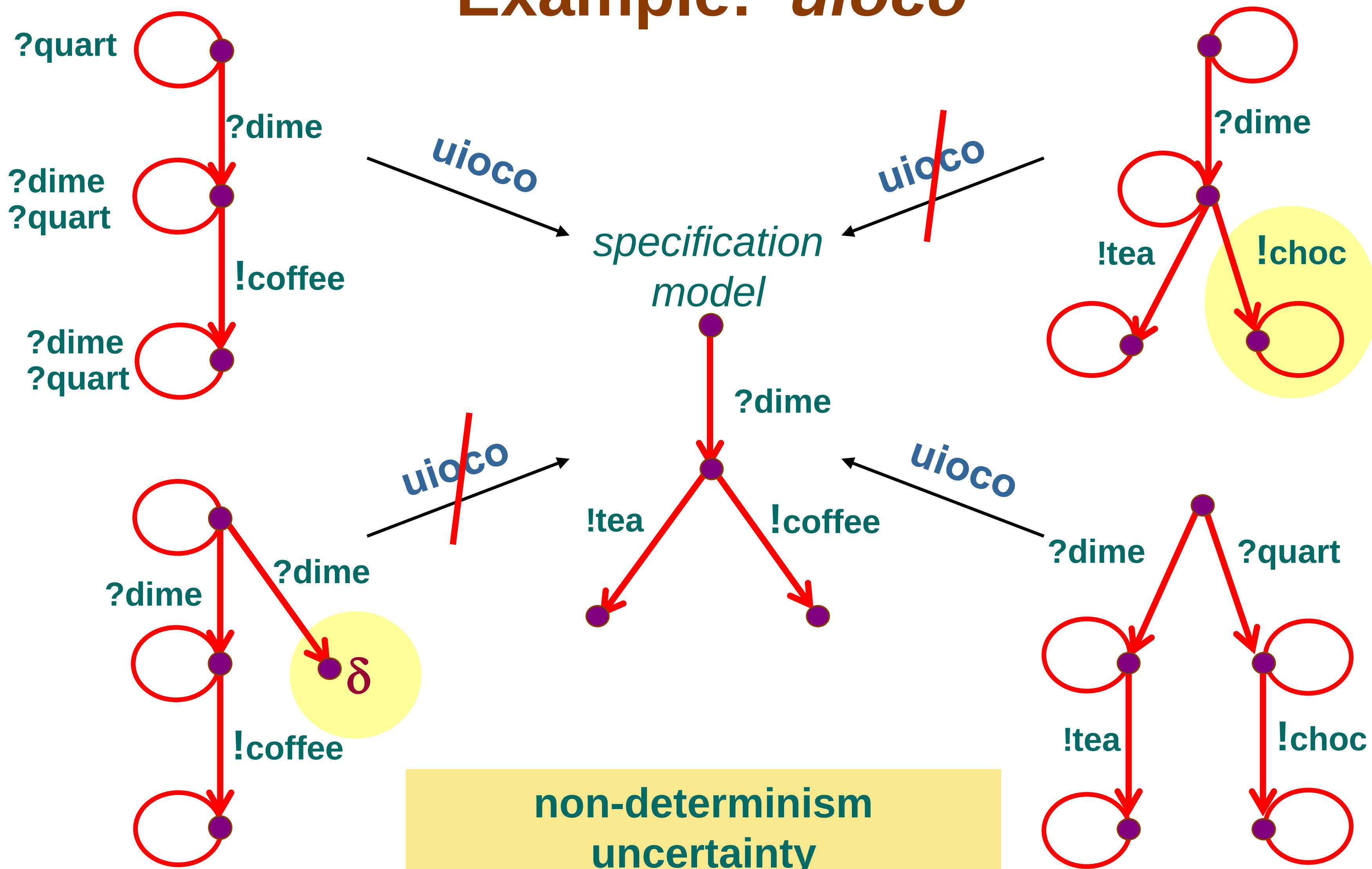
i is (assumed to be) an input-enabled LTS

Intuition:

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

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

Example: *uioco*



non-determinism
uncertainty
under-specification

Test Generation Algorithm : *uioco*

Algorithm to generate a test case ***t*** (***S***)

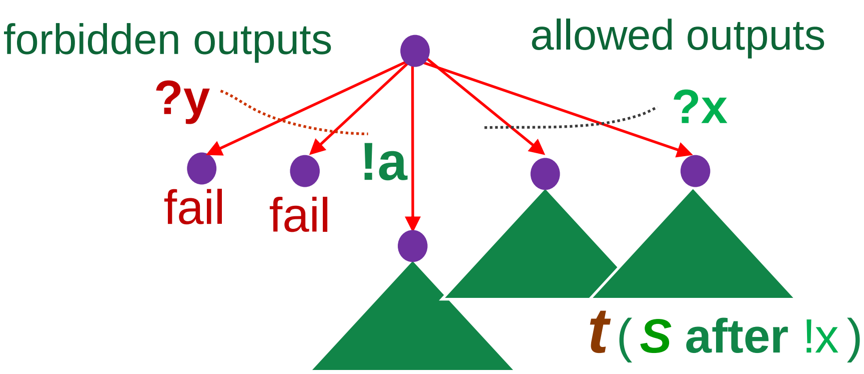
from a transition system state set ***S***, with ***S*** ≠ ∅, and initially` ***S*** = ***s_0*** **after ε**

Apply the following steps recursively, non-deterministically :

1 end test case

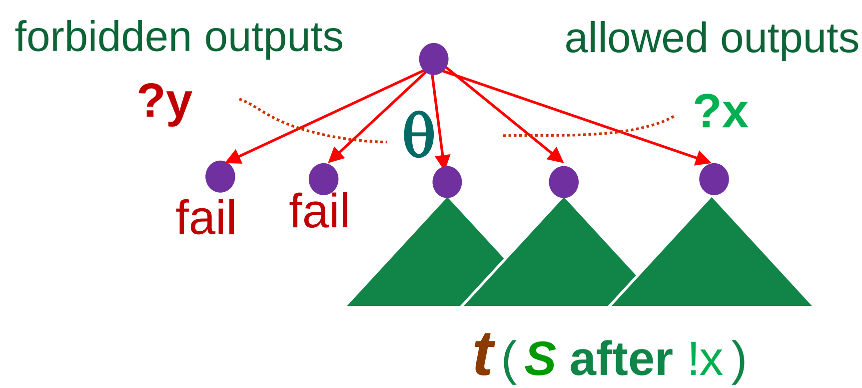
pass

2 supply input !a



t (***S*** **after ?a**), for **?a** ∈ L_I
and *not* ***S*** refuses {**?a**}

3 observe all outputs

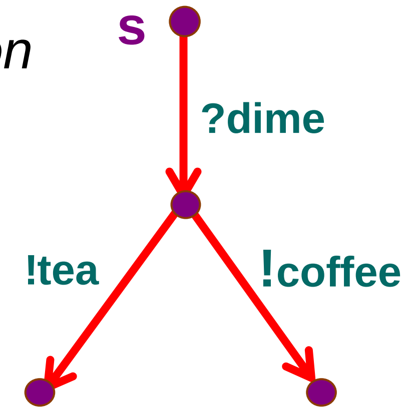


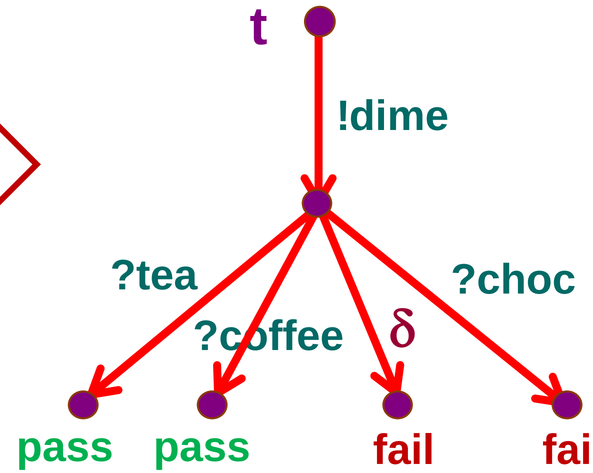
allowed outputs (or δ): **!x** ∈ *out* (***S***)

forbidden outputs (or δ): **!y** ∉ *out* (***S***)

Example: *uioco* Test Generation

specification model



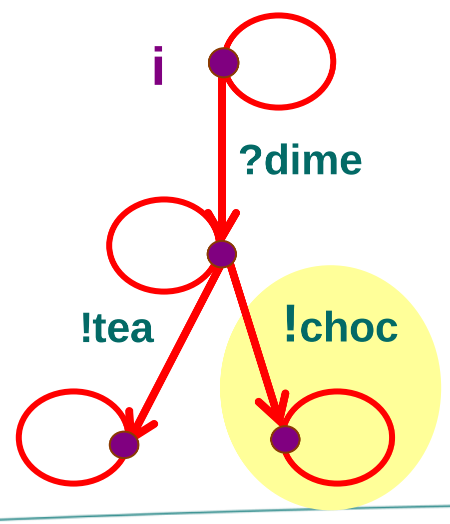


generated test case

i ~~uioco~~ s

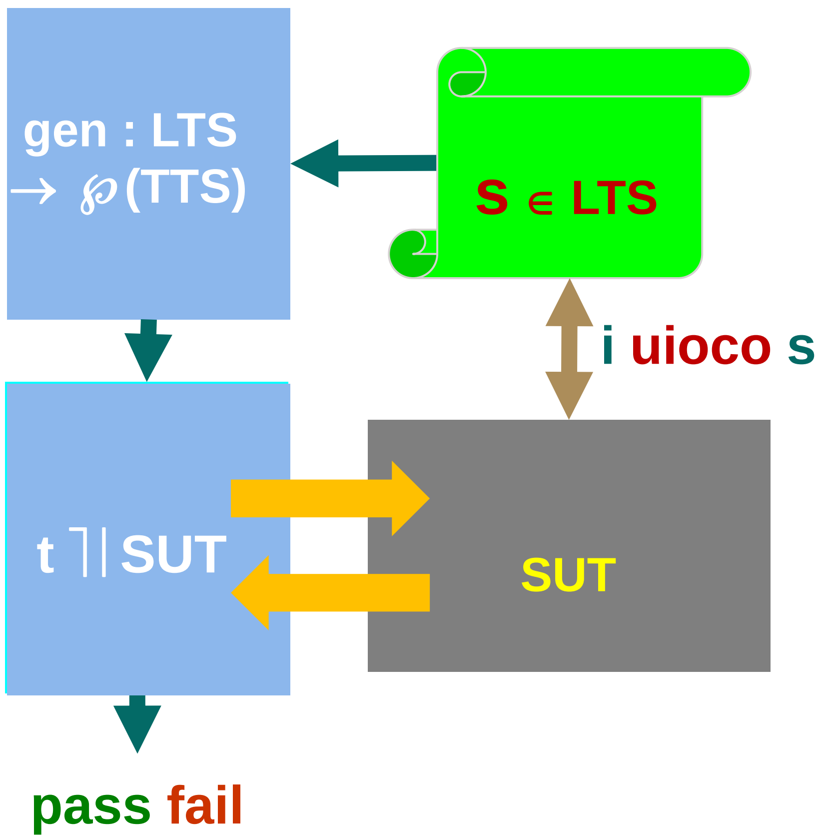
i fails t

implementation



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

MBT with *uioco* is Sound and Exhaustive



Testability assumption :

$\forall SUT \in IMP \,.\, \exists m_{SUT} \in IOTS \,.$

$\forall t \in TESTS \,.$

SUT passes $t \Leftrightarrow m_{SUT}$ passes t

Prove soundness and exhaustiveness:

$\forall i \in IOTS \,.$

$(\forall t \in gen(\mathbf{s}) \,.\, i \text{ passes } t)$

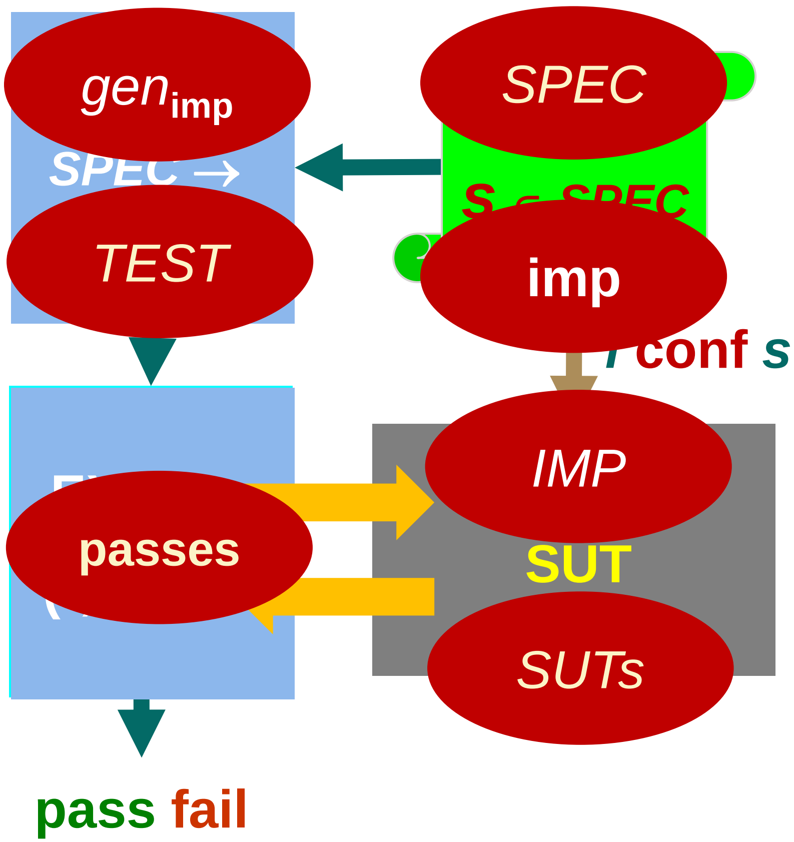
$\Leftrightarrow$ ***i* uioco *s***

SUT comforms to s

exhaustive ⇑ ⇓ *sound*

SUT passes gen(s)

MBT : Formal Framework Overview



Testability assumption :

$\forall \textbf{SUT} \,.\, \exists\, \boldsymbol{i}_{\textbf{SUT}} \in \mathit{IMP} \,.$

$\forall \boldsymbol{t} \in \mathit{TEST} \,.$

SUT **passes** $\boldsymbol{t} \Leftrightarrow \boldsymbol{i}_{\textbf{SUT}}$ **passes** $\boldsymbol{t}$

Prove soundness and exhaustiveness:

$\forall \boldsymbol{s} \in \mathit{SPEC} \,.\, \forall \boldsymbol{i} \in \mathit{IMP} \,.$

$(\forall \boldsymbol{t} \in \mathit{gen}_{\textbf{imp}}(\boldsymbol{s}) \,.\, \boldsymbol{i}$ **passes** $\boldsymbol{t})$

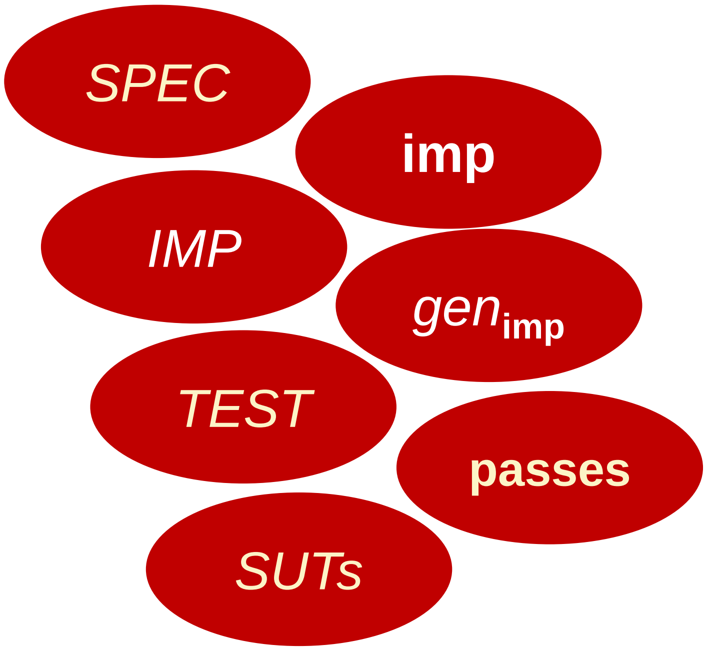
$\Leftrightarrow \boldsymbol{i}$ **imp** $\boldsymbol{s}$

SUT comforms to *s*

exhaustive ⇑ ⇓ *sound*

SUT passes $\mathit{gen}_{\textbf{imp}}(\boldsymbol{s})$

Formal Framework : Instantiations



Giving different interpretations to these abstract concepts, different model-based testing theories are obtained :

- *Labelled Transitions System*
- *Finite-State Machines*
- *Formal Language acceptance*
- *Abstract Data Type testing*
- *Property-Based Testing*
- . . . . .

Instantiation : Property-Based Testing

SPEC
- *relations over typed values*

IMP
- *functions*

SUTs
- *side-effect free computations*

TEST
- *sets of typed values*

imp
- *function ⊆ relation*

gen_{imp}
- *random value generation*

passes
- *constraint satisfaction on results of function computations*

You can use MBT

without knowing all this

MBT : Many Tools

- AETG
- Agatha
- Agedis
- Autolink
- Axini Test Manager
- Conformiq
- Cooper
- Cover
- DTM
- fMBT
- G∀st
- Gotcha
- Graphwalker
- JTorX
- MaTeLo
- MBTsuite
- M-Frame
- MISTA
- NModel
- OSMO
- ParTeG
- Phact/The Kit
- PyModel
- QuickCheck
- Reactis
- Recover
- RT-Tester
- SaMsTaG
- Smartesting CertifyIt
- Spec Explorer
- StateMate
- STG
- tedeso
- Temppo
- TestGen (Stirling)
- TestGen (INT)
- TestComposer
- TestOptimal
- TGV
- Tigris
- TorX
- TorXakis
- T-Vec
- Tveda
- Uppaal-Cover
- Uppaal-Tron
- . . . .

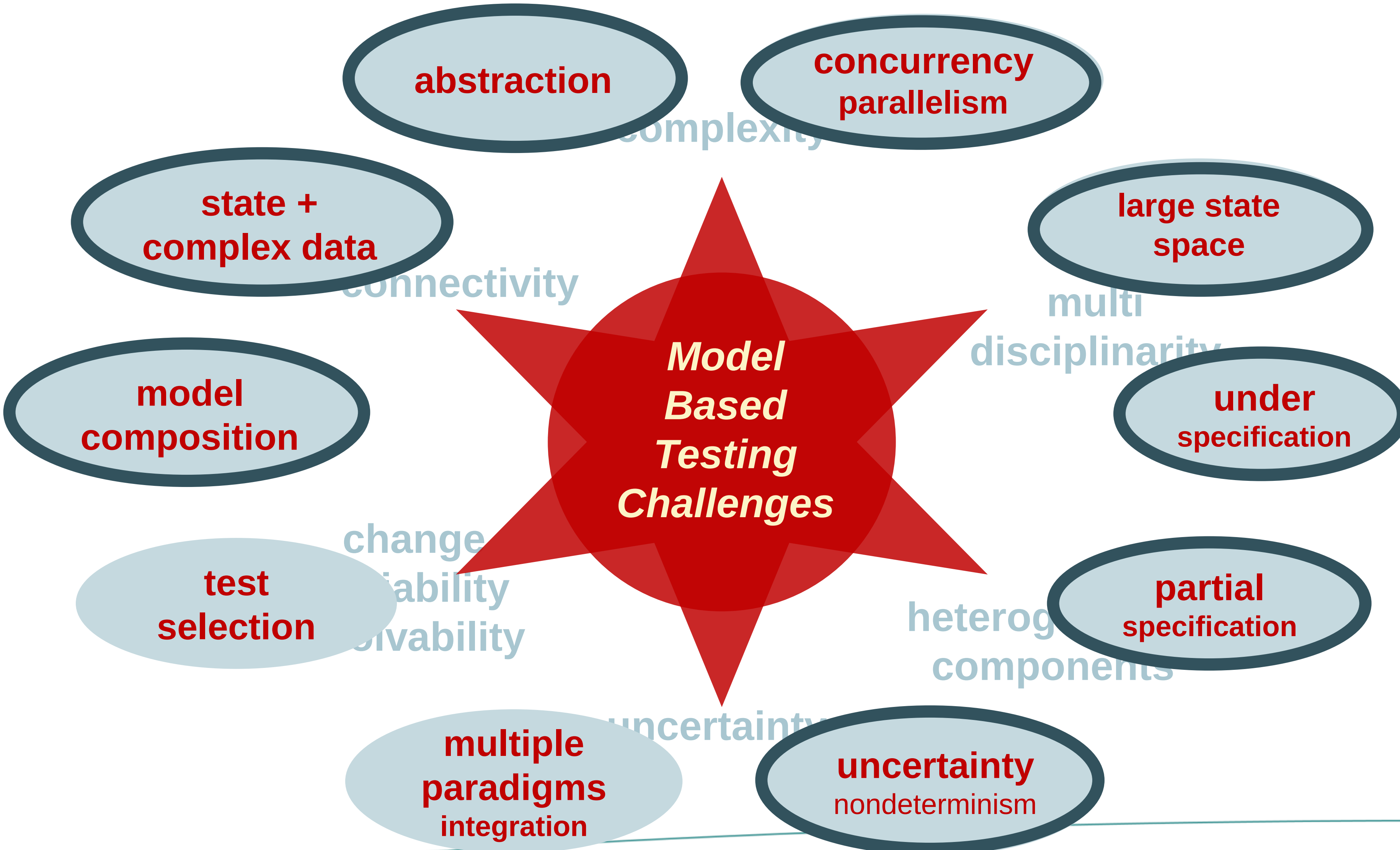
MBT Tools *u/ioco*

- AETG
- Agatha
- Agedis
- Autolink
- Axini Test Manager
- Conformiq
- Cooper
- Cover
- DTM
- fMBT
- G∀st
- Gotcha
- Graphwalker
- JTorX
- MaTeLo
- MBTsuite
- M-Frame
- MISTA
- NModel
- OSMO
- ParTeG
- Phact/The Kit
- PyModel
- QuickCheck
- Reactis
- Recover
- RT-Tester
- SaMsTaG
- Smartesting CertifyIt
- Spec Explorer
- StateMate
- STG
- tedeso
- Temppo
- TestGen (Stirling)
- TestGen (INT)
- TestComposer
- TestOptimal
- TGV
- Tigris
- TorX
- TorXakis
- T-Vec
- Tveda
- Uppaal-Cover
- Uppaal-Tron
- . . . .

Yet Another MBT Tool

- AETG
- Agatha
- Agedis
- Autolink
- Axini Test Manager
- Conformiq
- Cooper
- Cover
- DTM
- fMBT
- G∀st
- Gotcha
- Graphwalker
- JTorX
- MaTeLo
- MBTsuite
- M-Frame
- MISTA
- NModel
- OSMO
- ParTeG
- Phact/The Kit
- PyModel
- QuickCheck
- Reactis
- Recover
- RT-Tester
- SaMsTaG
- Smartesting CertifyIt
- Spec Explorer
- StateMate
- STG
- tedeso
- Temppo
- TestGen (Stirling)
- TestGen (INT)
- TestComposer
- TestOptimal
- TGV
- Tigris
- TorX
- **TorXakis**
- T-Vec
- Tveda
- Uppaal-Cover
- Uppaal-Tron
- . . . .

MBT : Next Step Challenges



TorXakis : Overview

Applications
- several high-tech systems companies
- experimental level

Models
- state-based control flow and complex data
- support for parallel, concurrent systems
- composing complex models from simple models
- non-determinism, uncertainty
- abstraction, under-specification

But ....
- research prototype
- poor usability

Tool
- on-line MBT tool

Under the hood
- powerful constraint/SMT solvers (Z3, CVC4)
- well-defined semantics and algorithms
- **ioco** testing theory
  for symbolic transition systems
- algebraic data-type definitions

Current Research
- test selection
- partial models & composition

Principles of

Model-Based Testing

There is nothing more practical than a good theory

Jan Tretmans
jan.tretmans@tno.nl

Radboud Universiteit

ESI

TNO innovation for life