Model Based Testing (MBT)

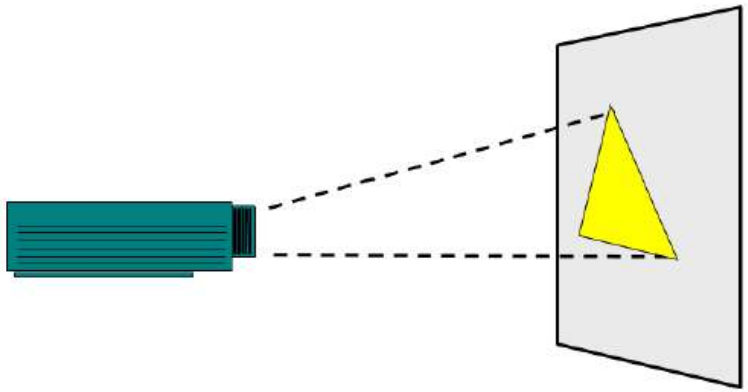
Ana Paiva

apaiva@fe.up.pt

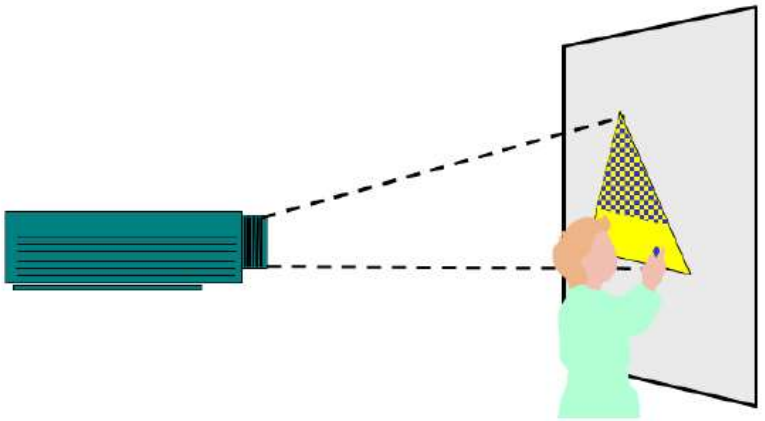
Universidade do Porto
FEUP Faculdade de Engenharia

INESCTEC
TECNOLOGIA E CIÊNCIA | LABORATÓRIO ASSOCIADO | COORDENADO POR INESCPORTO

"Traditional" Automated Testing

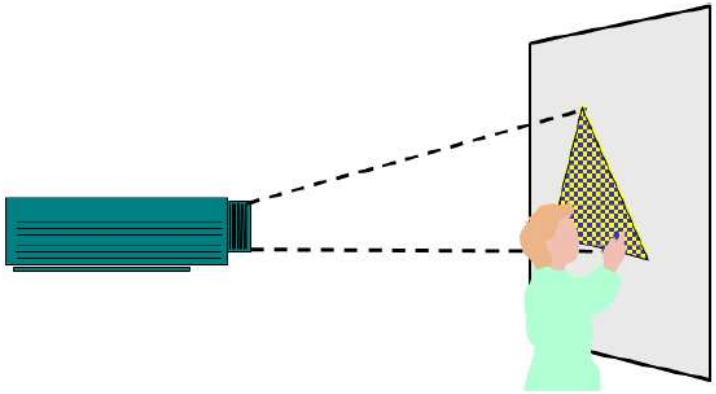


Imagine that this projector is the software you are testing.

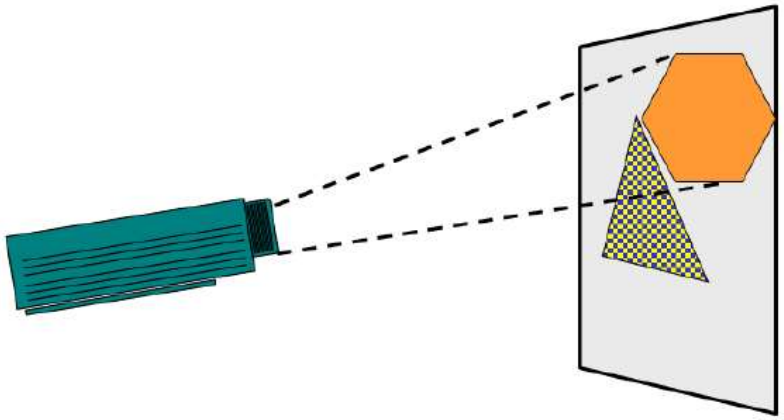


Typically, testers automate by creating static scripts.

"Traditional" Automated Testing

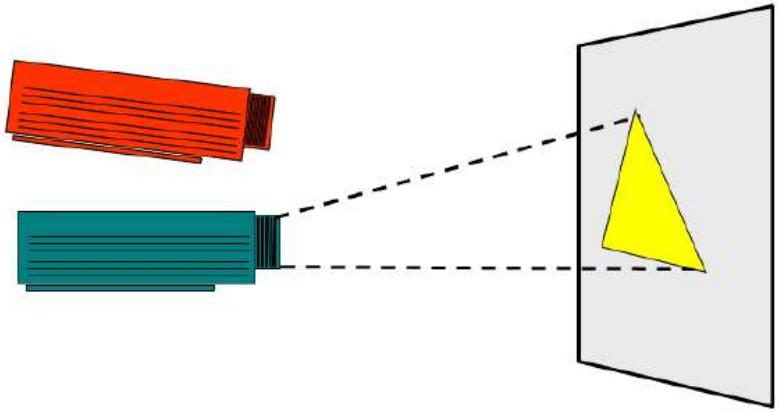


Given enough time, these scripts will cover the behavior.

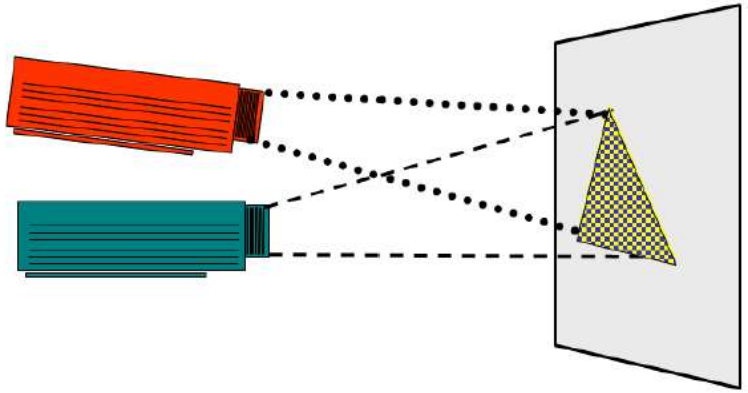


But what happens when the software's behavior changes?

Model Based Testing

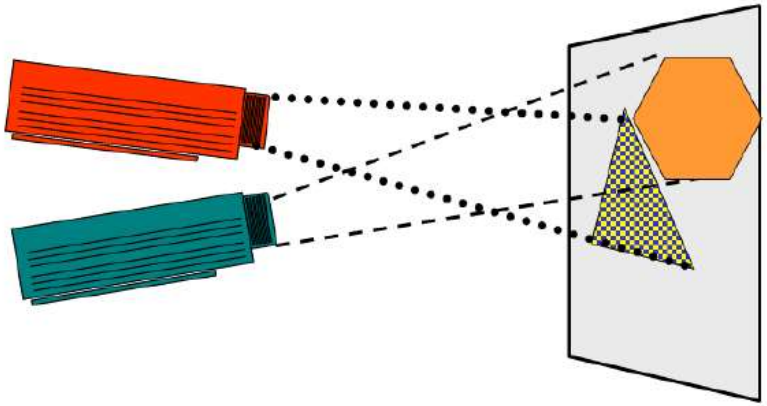


Now, imagine that the top projector is your model.

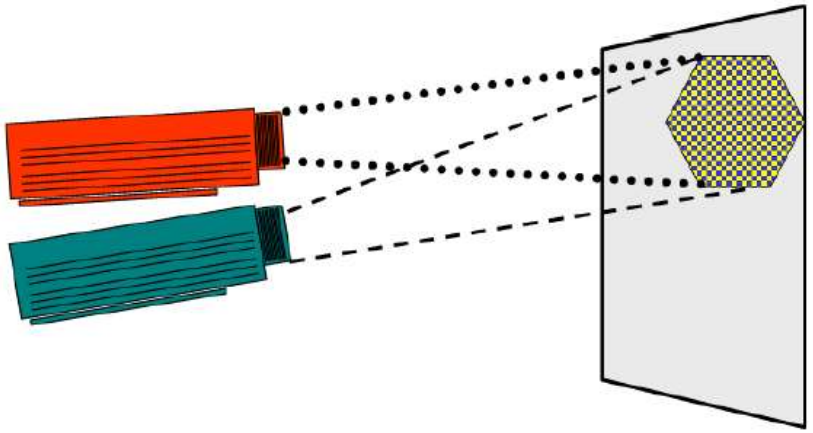


The model generates tests to cover the behavior.

Model Based Testing



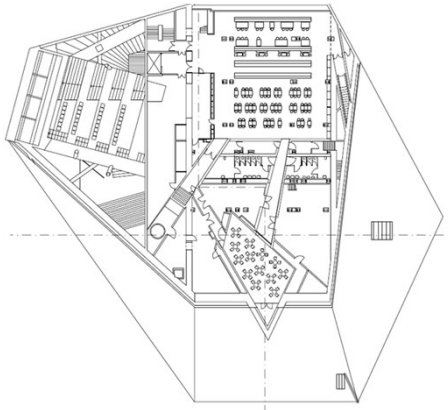
… and when the behavior changes…



… so do the tests.

What is a model?

- A Model is an abstract representation of the system.
- Models are simpler than the systems they describe.
- Models help us understand and predict the system's behavior.



Model

Real

Definition

- Model-based testing is software testing in which test cases are derived in whole or in part from a model that describes some (usually functional) aspects of the System Under Test (SUT).

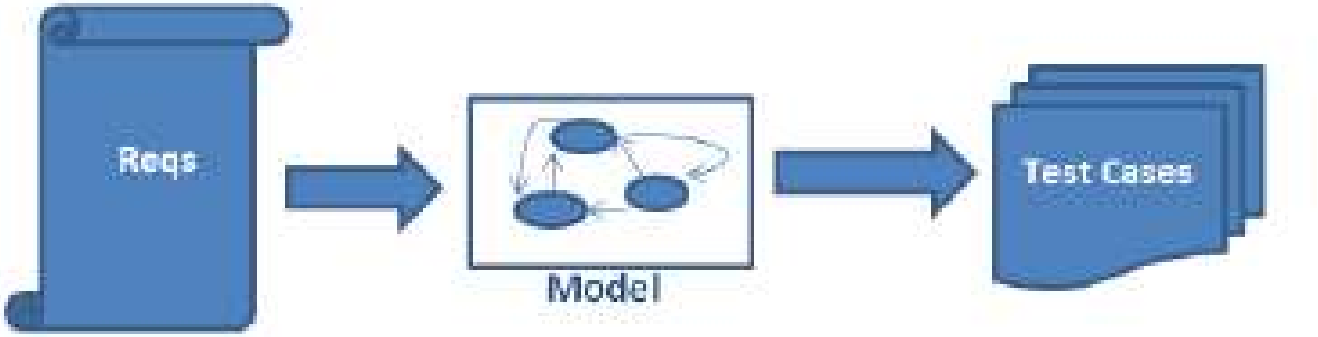
- Model-based testing uses a model program to generate test cases or act as an oracle.

Definition

- An oracle is the authority which provides the correct result used to make a judgment about the outcome of a test case - whether the test passed or failed

Definition

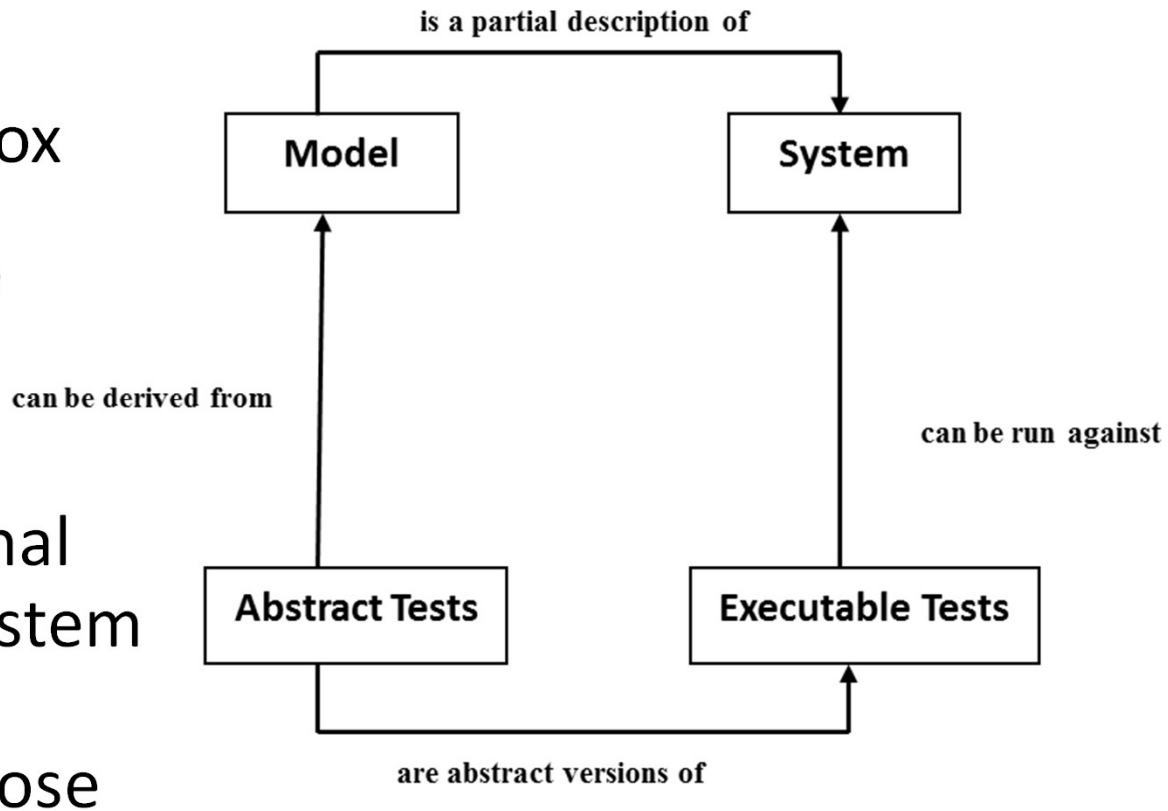
- Model Based Testing is a testing technique where the **runtime behavior** of an implementation under test is checked againts predictions made by formal specification, or **model**." – Colin Campbell, Microsoft Research



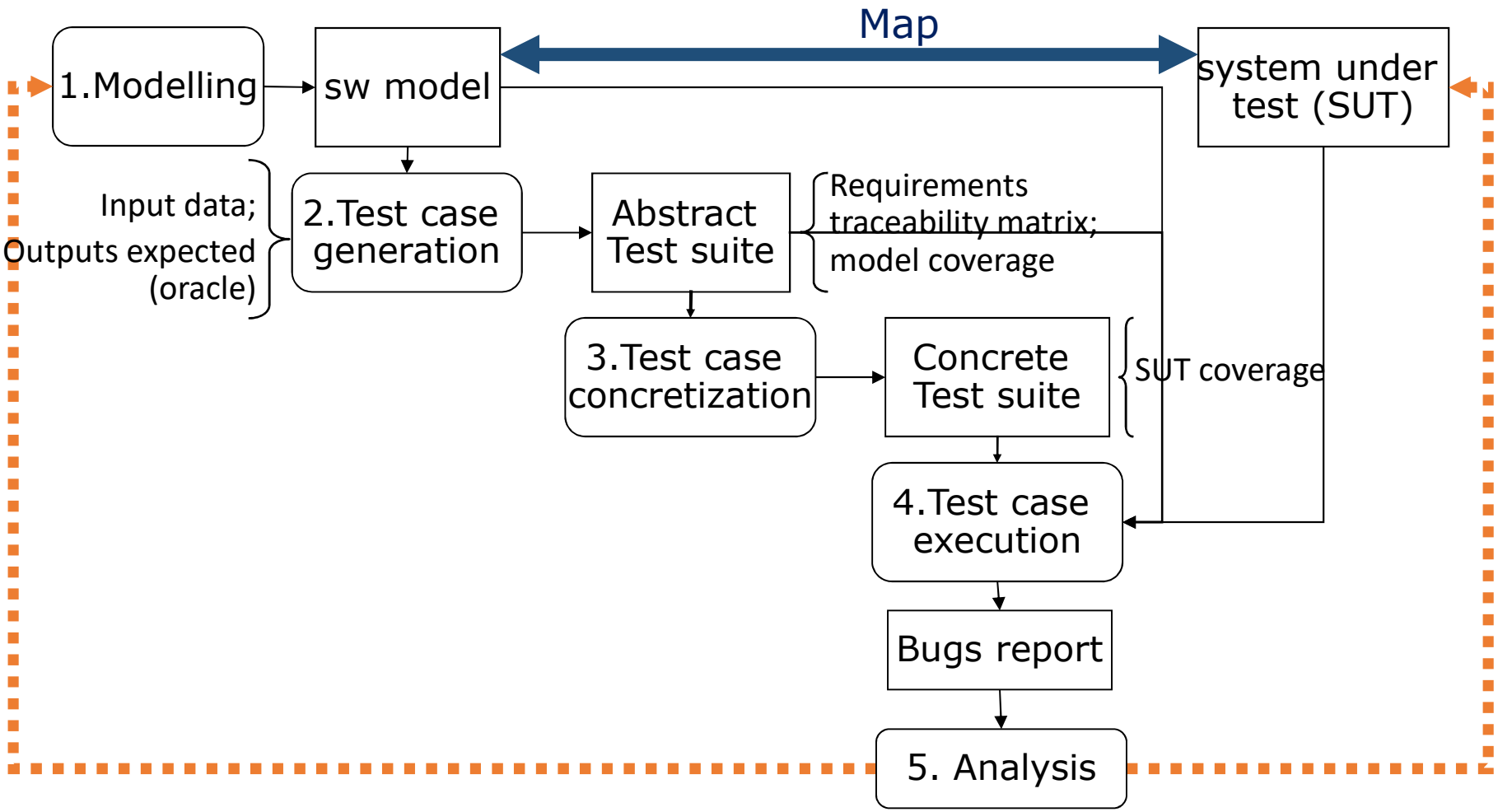
- One way to generate test cases automatically is "model based testing" where a model of the system is used for test case generation.

MBT Process

- "Model-based testing is black box testing process for derivation of test cases from a model that describe functional aspects of the system under test and executing those test cases."



MBT Process


Map
1.Modelling
sw model
system under test (SUT)
Input data;
Outputs expected (oracle)
2.Test case generation
Abstract Test suite
Requirements traceability matrix; model coverage
3.Test case concretization
Concrete Test suite
SUT coverage
4.Test case execution
Bugs report
5. Analysis

General Test Strategies

- Manual vs automatic
- Static vs dynamic
- White vs Black-box
- Error/Fault based vs specification-based
- API vs UI testing
- Functional vs non-functional (Usability, Performance, Robustness)

Advantages and Disadvantages

- Advantages
  - Improved quality of the product
  - Better quality of the product
  - Higher degree of automation (test case generation)
  - Allows more exhaustive testing
  - Good for correctness/functional testing
  - Model can be easily adapted to changes
  - Early exposure of ambiguities in specification and design
  - Generate variety of test suites from the same model by different test selection criteria

Advantages and Disadvantages

- Disadvantages
  - Requires a formal specification/model
  - Test case explosion problem
  - Test case generation has to be controlled appropriately to generate a test case of manageable size
  - Small changes to the model can result in a totally different test suite
  - Time to analyse failed tests (model, SUT, adaptor code)

Some tools

- ModelJUnit
- SpecTest
- Mulsaw
- Nmodel and SpecExplorer
- PBGT
- GUITAR
- ...

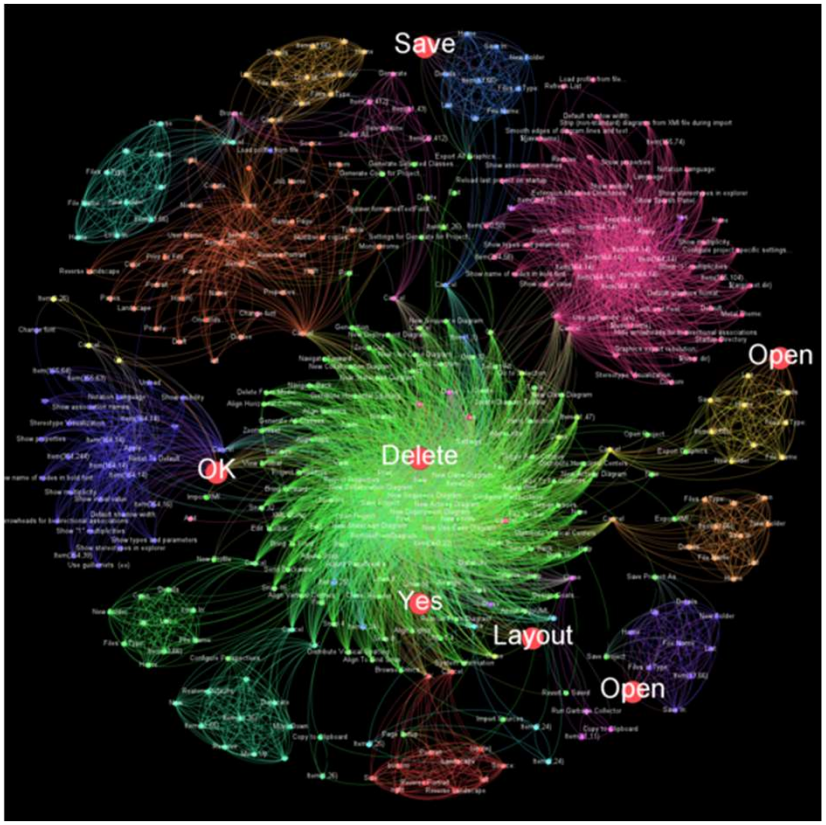
Spec Explorer

- Is developed by Windows (so support is assured);
- Not many tutorials…
- But tons of references.
- Uses .NET language.

- The modeling is made by code, only later can the visual model be observed.

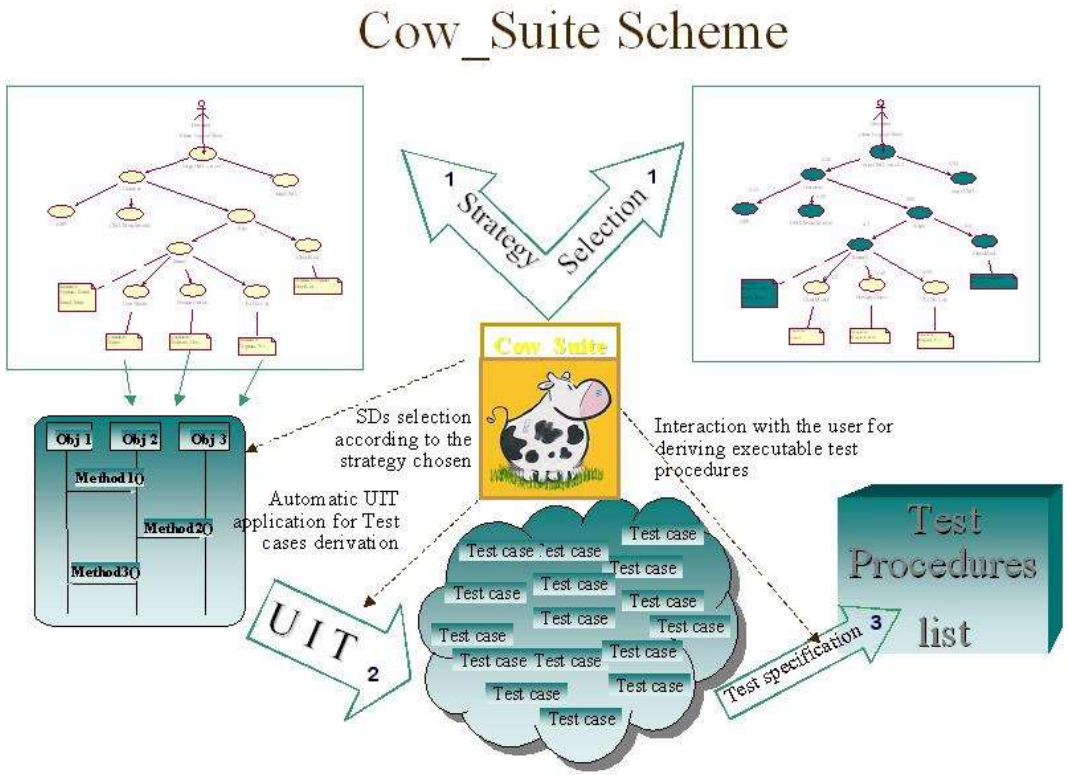
GUITAR

- Nice looking models! -----> EFG
- Can test GUI interfaces
- Very few tutorials
- Difficult to use



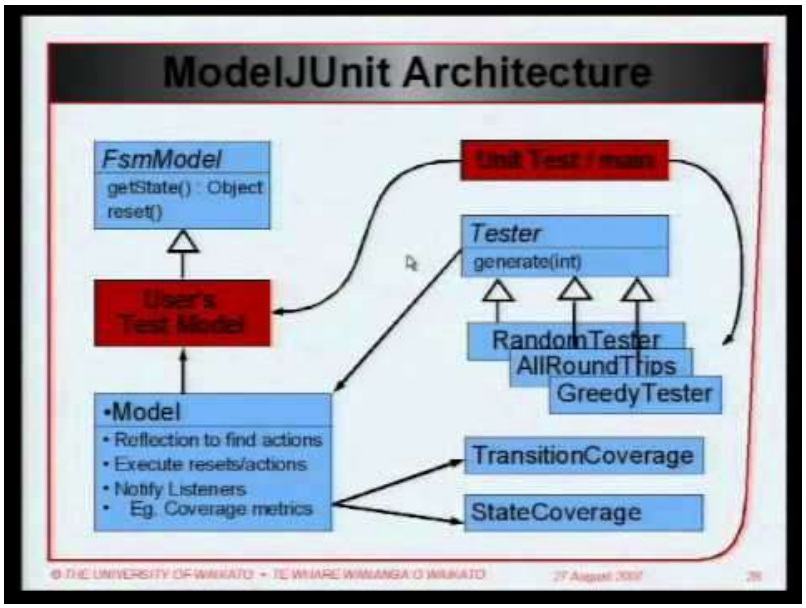
CowSuite

- An automated test strategy based on UML models



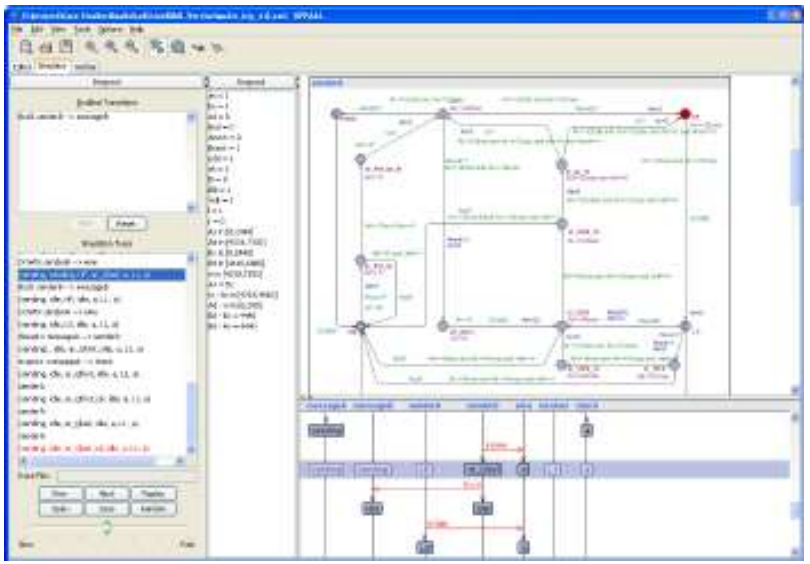
ModelJUnit

- ModelJUnit is a Java library that extends JUnit to support model-based testing. Models are extended finite state machines that are written in a familiar and expressive language: Java. ModelJUnit is an open source tool, released under the GNU GPL license.



T-UPPAAL

- T-UPPAAL - a tool for model based testing of embedded real-time systems that automatically generates and executes tests "online" from a state machine model of the implementation under test (IUT) and its assumed environment which combined specify the required and allowed observable (realtime) behavior of the IUT



Tools survey ...

Tool Name	Type	Modeling Language	Usability	Automatic generation of test cases	Export models to other formats	Model Simulation	Platform
UPPAAL	Academic	UML	Yes	No	Can export to XML file	Yes	Windows
CowSuite	Research	UML	Yes	Yes	-	Yes	Windows
JUMBL	Academic	TML (Markov Chain Models)	No	Yes	-	No	Windows /Unix
Enterprise Architect	Commercial	UML	Yes	Only for Modeling and does not support model based testing			
Spec Explorer	Research	SPEC # and ASML	Yes	Yes	Can export to XML file	No	Windows
Model Junit	Academic	Java	No	Yes	Can export to XML file	No	Windows

Tools survey ...

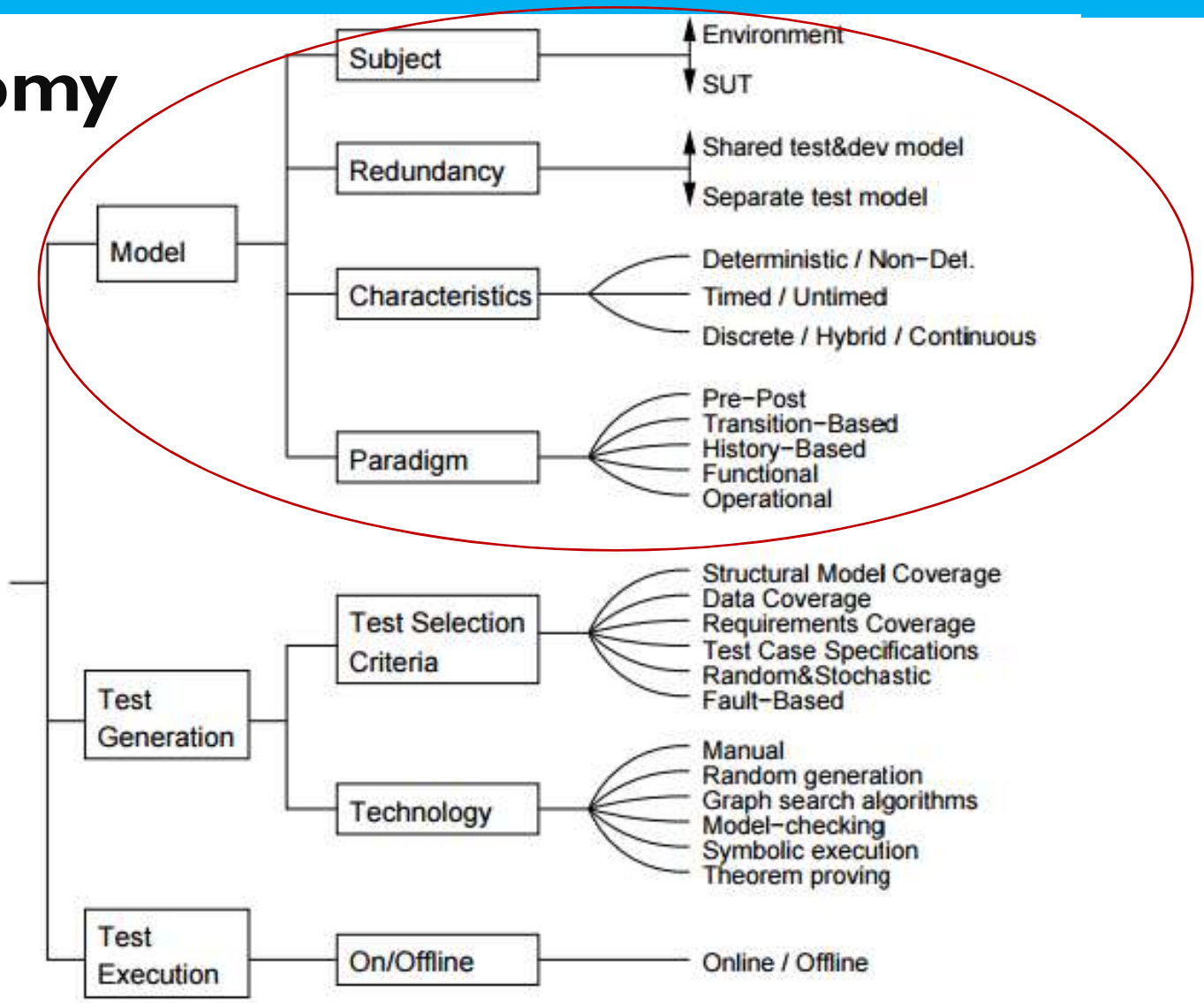
Tool	Modified	Ref.	Input format	Type	Description
BPM-Xchange	2014	[FS14]	BPMN, UML...	Commercial	BPM-Xchange creates test cases from business process models based on different criteria (statement, branch, path, condition). It can import models from several modeling tools, and can export test cases to Excel, HP Quality Center, etc.
Conformiq Designer	2014	[H07]	UML State Machines, QML	Commercial	Models can be created as UML State Machines and in Qtronic Modeling Language (QML). Tests can be exported to test management tools or TTCN-3. (Previously was Conformiq Qtronic.)
DTM	2013		custom activity model	Commercial	The DTM (Dialogues Testing Method) tool uses a custom activity model (actions, decisions...), and selects tests based on structural coverage (multiple condition or all path).
fMBT	2014		Custom (AAL)	Open source	fMBT (free Model-Based Testing) generates test cases from models written in the AAL/Python pre/postcondition language using different heuristics (random, weighted random, lookahead...).
GraphWalker	2014		FSM	Open source	Test generation from Finite State Machines. Search algoritms: A* or random, with a limit for various coverage criteria (state, edge, requirement). Formerly called as mbt.
JSXM	2014	[DBI12]	EFSM (Stream X-machines)	Academic	JSXM is model animation and test generation tool that uses a kind of EFSMs as its input. The generated tests can be transformed to JUnit test cases.
JTorX	2014	[B10]	LTS	Open source	JTorX is a reimplementation of TorX in Java with additional features. The LTS specification can be given in multiple format, and it can interact on-the-fly with the implementation under test.
					MaTeLo (Markov Test Logic) is a commercial

Tools survey …

- Go to Google and look for
  - Model based testing tools survey
  - Model based testing tools comparison
  - …

MBT Phases - Modeling

Taxonomy



[Utting, Pretschner and Legeard]

Modeling the system

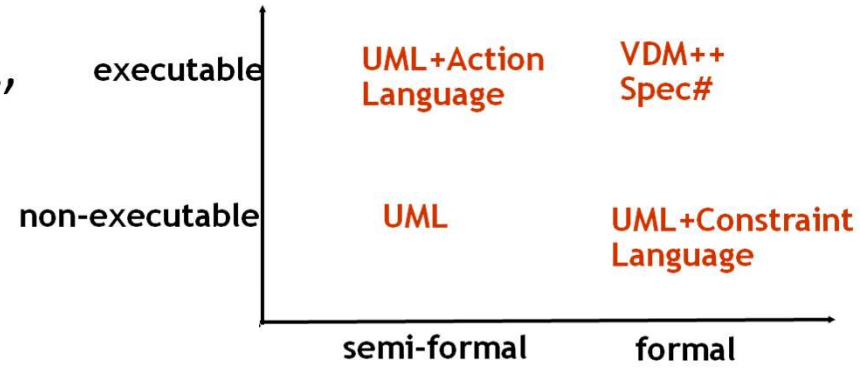
- Design the model to meet your testing goals
- Choose the right level of abstraction (which aspects of SUT you want to test)
- Choose a notation for modelling
- Once the model is written, ensure that it is accurate (verify and validate your model)

Built or Borrow?

- Reuse the development models – 100% reuse
  - Too much detail
  - Usually do not describe the *dynamic* behaviour of the SUT
  - It is neither abstract enough nor precise
- Models to generate code or reverse engineered
  - Lack of independence (implementation and test cases are derived from the same source code)
- Develop test models from scratch – 0% reuse
  - Maximum level of independence

Different modeling notations

- Visual (ex.: UML diagrams) or textual
- Formal models (ex.: Z, VDM, OCL, …)
- Executable (ex.: XTUML)
- Non-executable models (ex.: implicit)
- Deterministic vs non-deterministic
- Timed vs non timed
- Discrete vs continuous
- DSL



Formal notation

- Pre/Post (or Model-based). Ex.: VDM, Z, Spec#.
- Transition-based. Ex.: FSM, Petri nets.
- Behavior-based (or history-based). Ex.: CSP.
- Property-based (or functional-based). Ex.: OBJ.
- Hybrid approaches. Ex.: RAISE.
- UML with OCL
- GUI modeling: Abstract models (e.g., PiE e RED-PiE), grammars, FSM, Property based, Behavioral based, hybrid, EFG, Spec#, etc.

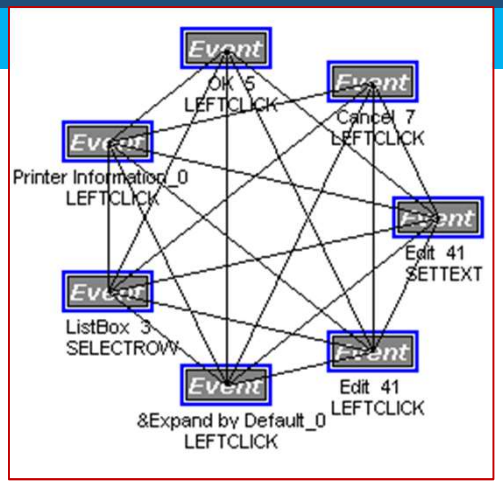
UML

- ["29 New Unclarities in the Semantics of UML 2.0 State Machines" by H. Fecher, J. Schönborn, M. Kyas, W. Roever]

- Unclarities categorized as
  - Incompleteness
  - Inconsistency
  - Ambiguity
  - Equivocality

GUI modeling

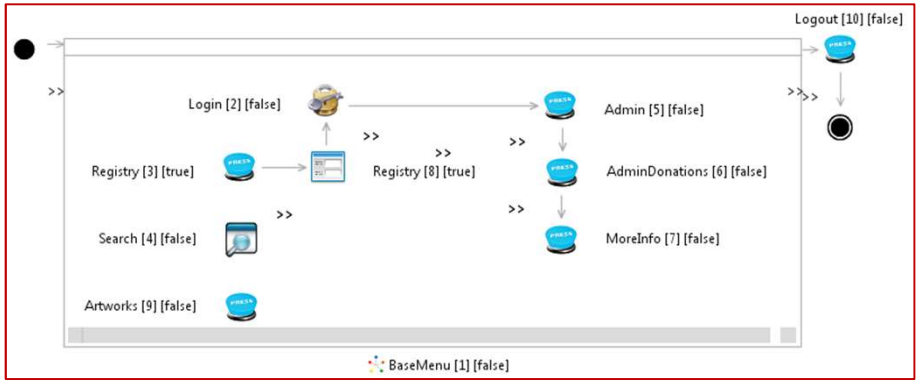
- EFG (GUITAR)



- Spec# (Mapping tool)

```
namespace MyNotepad;

//State variables
string text="",selText="";
bool dirty=false;
int posCursor=0;

// Start and close the Notepad application
[Action] void LaunchNotepad()
requires !IsOpen("Notepad"); {
  AddWindow("Notepad","",false);
  //... state variables initialization ...
}

[Action] void Close()
requires IsEnabled("Notepad"); {
  if (dirty) AddWindow("MsgClose","Notepad",true);
  else { RemoveWindow("Notepad");
        //... reset variables to initial values
       }
}
```

- UI Test Patterns (PBGT)

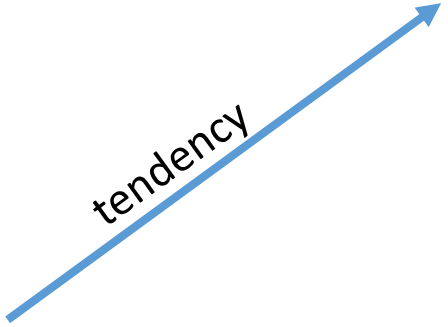


Testing models

- Tendency ?
  - Modelling testing goals

  instead of

  - Modelling SUT behavior



Choosing a modeling notation

- Which characteristics may be important?
  - Expressive power
  - Abstraction level
  - Can it be accepted in industry or not?
  - Simple enough?
  - Tool support

Choosing a modeling notation

- Pre/Post (for modelling complex data) and transition-based (for modelling control) are the **most common** notations used in model-based testing processes

- Whatever notation you choose, it has to be a *somehow* a **formal** language with precise semantics in order to write accurate models so as to be used as test oracles

Testing model

The characteristics of the modeling language will lead to different

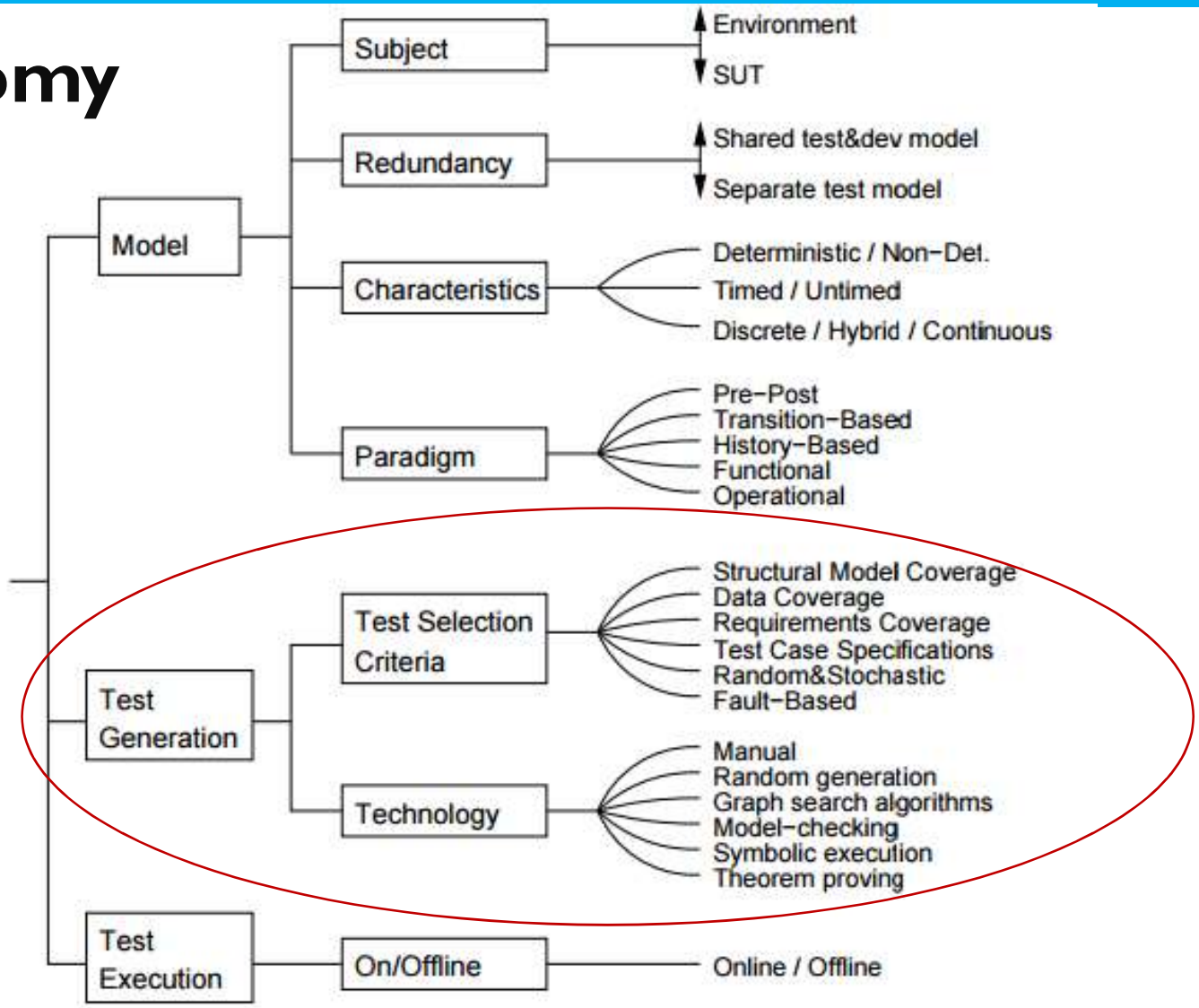
- test case generation techniques

and

- different kinds of test oracles (assertions versus executable specification)

MBT - Test case generation

Taxonomy



[Utting, Pretschner and Legeard]

Challenge

"To design a test generator that addresses the whole test design problem from

choosing input values

and

generating sequence of operation calls

to generate executable tests that include verdict information"

Challenge

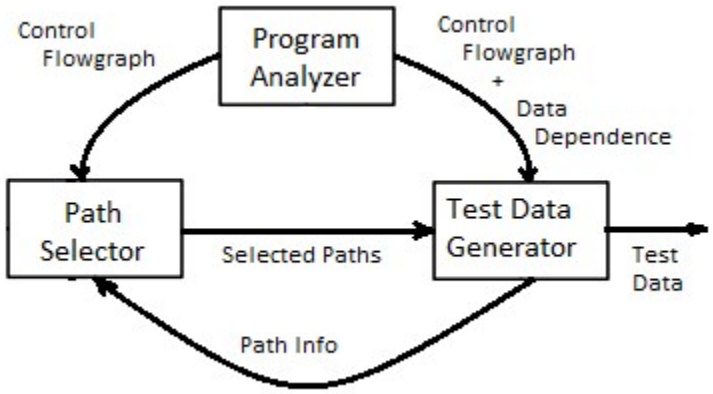
Test case explosion or combinatorial explosion

When SUT have:

- **Several parameters**, each with so many possible values that testing everything is infeasible
- **Many configuration parameters**, each with several possible values that testing each configuration is infeasible, e.g., versions of sw or hw component

Test data generators

- Based on the Mathematical Modelling above we can simply state the Test Data Generator Problem as:
- Given a program **P** and a path **u**, generate input **x** ∈ **S**, so that **x** traverses path **u**.
- A Test Data Generator follows the following steps
  - Program Control Flow Graph Construction
  - Path Selection
  - Generating Test Data



Test data generators

- The path selector identifies the paths. Once a set of test paths is determined the test generator derives input data for every path that results in the execution of the selected path. This is done in two steps:
  - Find the predicate for the path
  - Solve the path predicate
- The solution will ideally be a system of equations which will describe the nature of input data so as to traverse the path. In some cases the generator provides the selector with feedback concerning paths which are infeasible etc.

Test data generators

- Random test data generators
  - Probably the simplest method
  - Can be used to generate input for any type of program
  - Does not generate quality test data as it does not perform well in terms of coverage
- Pathwise data generators
  - A computer program and a testing criterion (e.g., total path coverage, branch, etc.) as input and then generate test data that meet the selected criterion. Typical steps are 1) program control flow graph construction, 2) path selection, and 3) test data generation

Test data generators

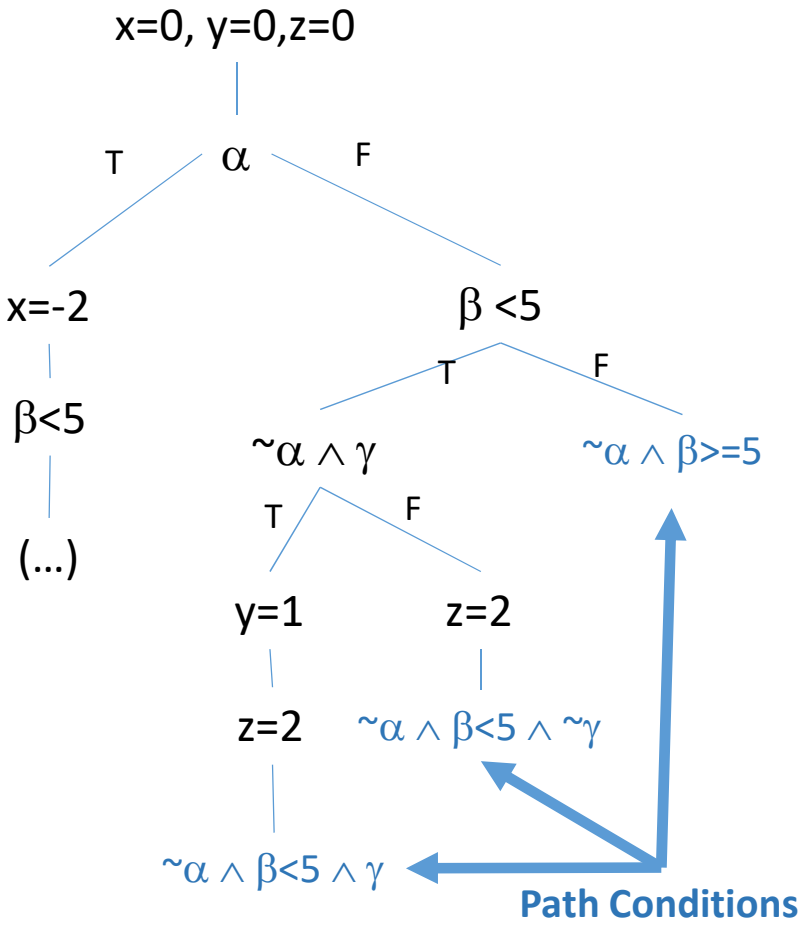
- Goal oriented generators
  - Channing approach tries to identify the nodes of the control flow graph that are vital to the execution of the goal node
  - Assertion oriented tries to find any path to an assertion that does not hold
- Intelligent test data generators
  - depend on sophisticated analysis of the code to guide the search of the test data. Intelligent Test Data Generators are essentially utilize one of the test data generation method coupled with the detailed analysis of the code.

Symbolic execution

- Symbolic execution (also symbolic evaluation) is a means of analyzing a program to determine what inputs cause each part of a program to execute.
- An interpreter follows the program, assuming symbolic values for inputs rather than obtaining actual inputs as normal execution of the program would, a case of abstract interpretation.
- It thus arrives at expressions in terms of those symbols for expressions and variables in the program, and constraints in terms of those symbols for the possible outcomes of each conditional branch.

Symbolic execution

1. int a=α, b=β, c=γ // symbolic
2. int x=0, y=0, z=0
3. if (a) {
4. x = -2;
5. }
6. if (b<5) {
7. if (!a && c) {y=1;}
8. z=2;
9. }
10. assert (x+y+z!=3)



Symbolic execution

When you get the path condition
you can then use a SAT (Satisfiability solver)
to find the values for the symbols
that satisfy such expression

- Some examples of SAT solvers are
  - Z3, STP and Yices

- SAT solvers are automated theorem provers

Combination strategies

Different combination strategies support different levels of coverage

- **1-wise**: requires that every value of every IPM parameter is included in, at least, one test case in the test suite
- **2-wise**: (also known as pair-wise) requires that every possible pair of values of any two IPM parameters is included in some test case

IPM – input parameter value

A (start): 1:negative, 2: zero, 3: positive
B (length of sequence): 1:zero, 2:one, 3: >1
C (direction of sequence): 1:negative, 2:positive

Test case	A	B	C
TC1			
TC2			
TC3			

1-wise

Test case	A	B	C
TC1			1
TC2			2
TC3			1
TC4	2		
TC5	2		
TC6	2		
TC7		1	
TC8	3	2	2
TC9		3	

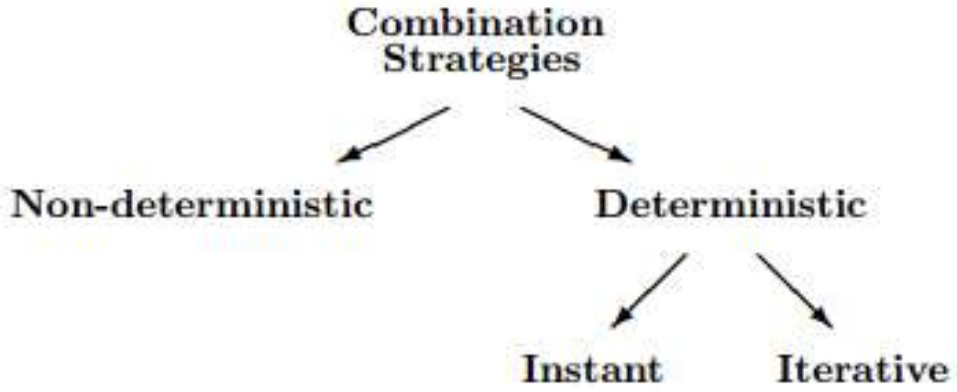
2-wise

Combination strategies

- **t-wise**: every possible combination of values of the IPM parameters to be included in some test case in the test suite
- **N-wise**: requires a test suite to contain every possible combination of the IPM parameter values in the IPM. Often too large to be practical

Combination strategies

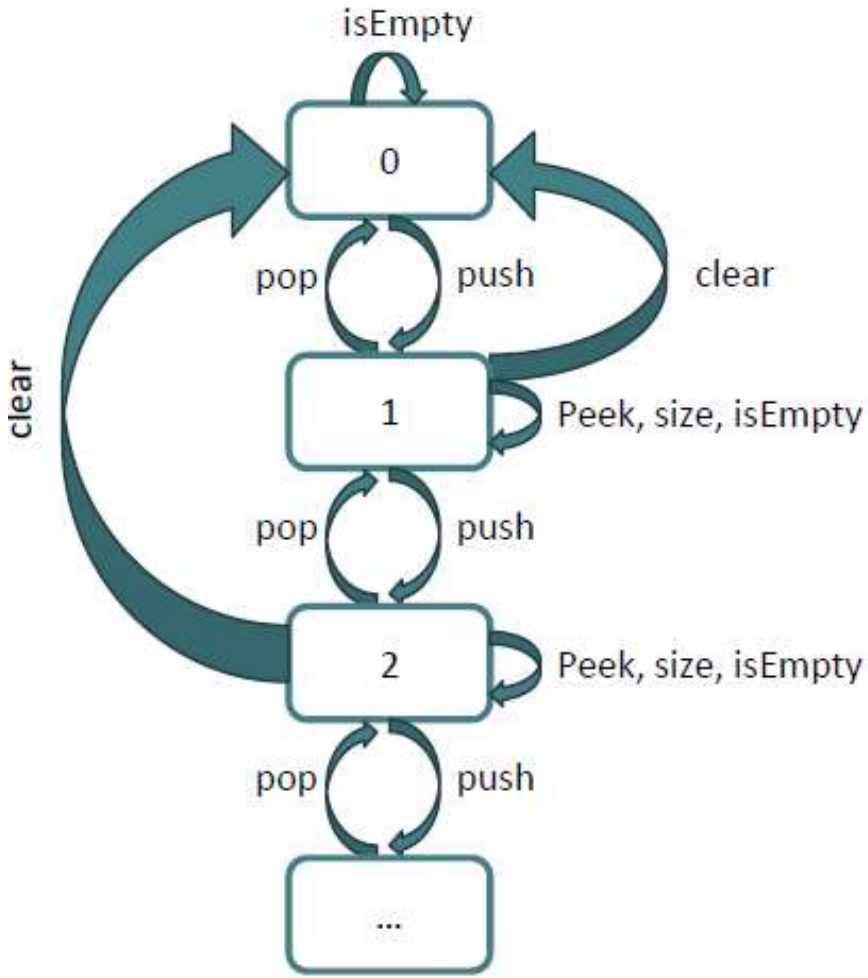
- **Non-deterministic** – rely to some degree of randomness. These solution may produce different solutions to the same problem at different times
- **Deterministic** – always produce the same solution for a given problem
- **Instant** – produce all combinations in an atomic step
- **Iterative** – build the solution one combination at a time



Generic Strategies – navigation through a FSM

- **Random** – traverses randomly the FSM
- **Greedy** – similar to random but give priority to paths not yet covered
- **AllRound** – similar to Greedy but prevents go through cycles of states more than once
- **Lookahead** – analyses the paths that departure from current state and chooses the one that goes into more states and new transitions (extremely slow)

Example - stack



FSM – State explosion problem

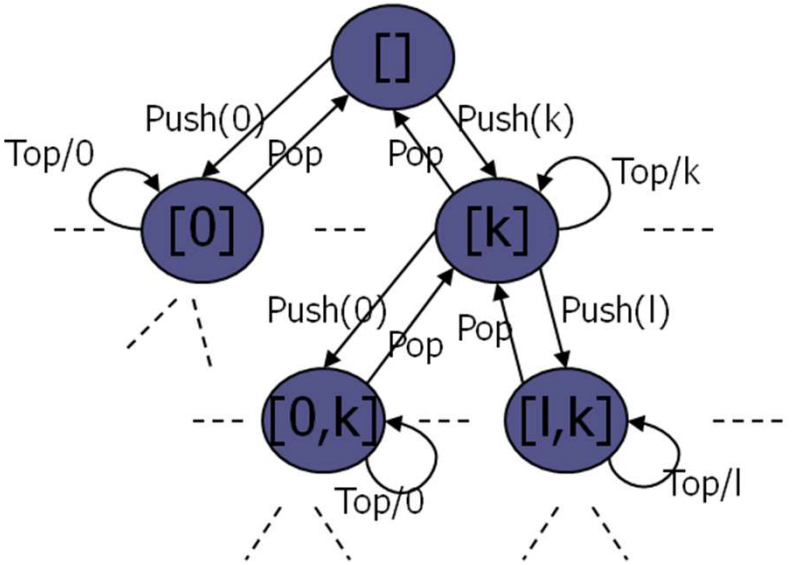
- In general the transition system describing a software system is infinite, but we can impose limits
- Goal
  - Create a state space of manageable size that satisfies a given testing goal
- Two main tasks
  - Restrict action parameter domains to interesting values
  - Restrict the state space to interesting states

- Note:
  The two tasks are not necessarily independent!

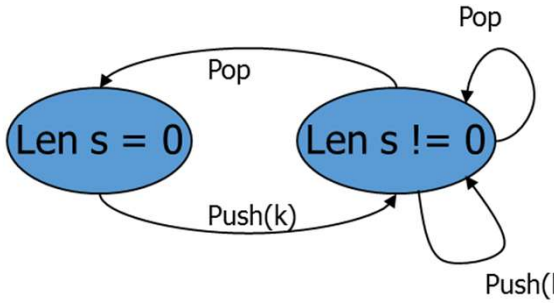
State space of a Stack model - SpecExplorer

Stack Spec# model

```
var Seq<int> content = Seq{};

[Action]
public void Push(int x) {
  content = Seq{x} + content;
}

[Action]
public void Pop() {
  requires !content.IsEmpty;
  content = content.Tail;
}

[Action]
public int Top() {
  requires !content.IsEmpty;
  return content.Head;
}
```



Projection on length



State explosion problem

- The following techniques are used in Spec Explorer
  - State filters
  - Stopping conditions
  - State groupings
  - Search order
  - Enabling conditions on actions

- Usually a combination of all (or some) of the techniques is needed to achieve desired effect

State Grouping

One or more state groupings may be defined

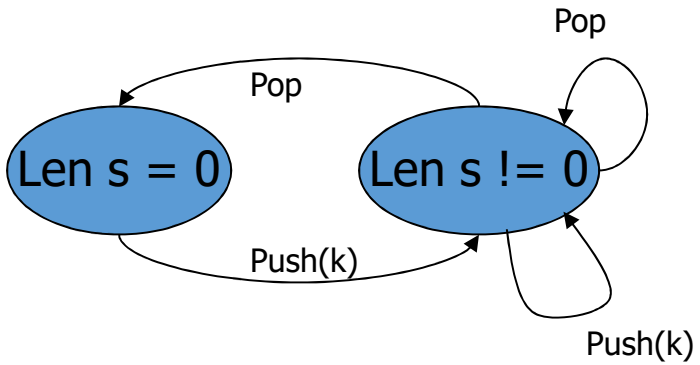
- A grouping G is a sequence of state based on expressions $g_1,\ldots,g_k$
- Two states *s* and *t* are in the same group G or G-equivalent if
  - $g_i^s = g_i^t$ for $1 \leq i \leq k$
- A G-group is the set of all G-equivalent states

Also used in viewing

Main purpose of groupings

- View all states in a G-group as being the same
- A way to define “what is an interesting state” from some testing point of view

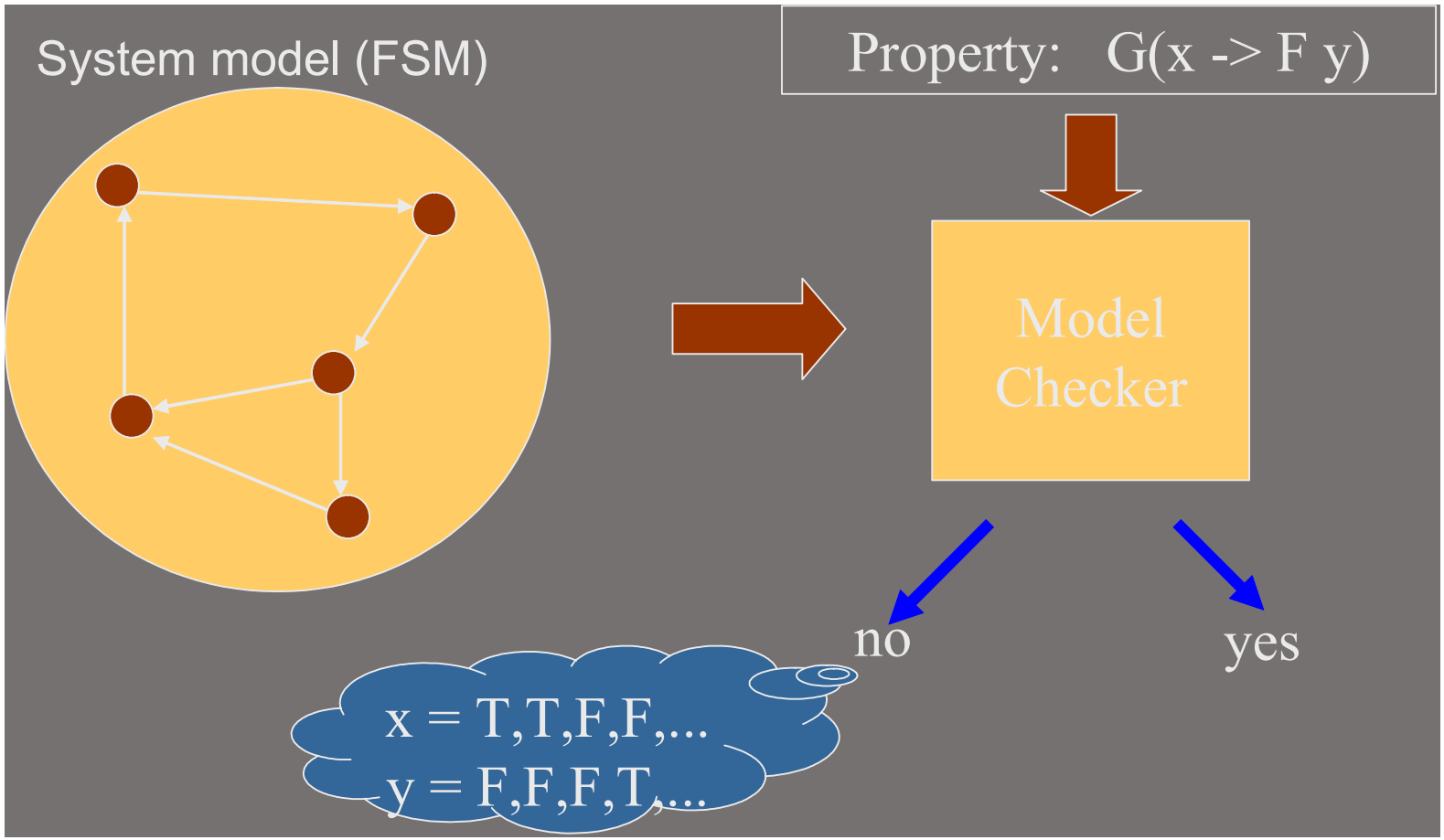
- Example: content.Size in the stack model



Techniques

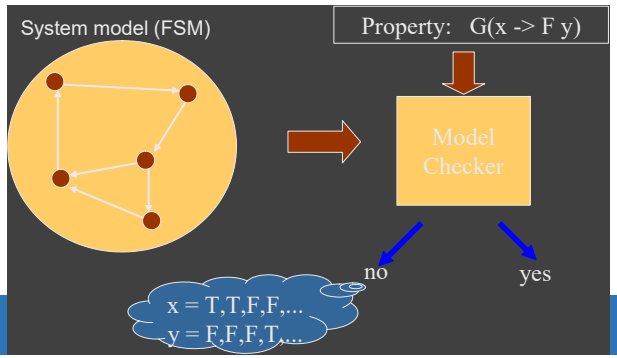
- Depends on the specification characteristics:
  - Pre/Post. Ex.: VDM, Z, Spec#
    - Structural converage
    - Partition testing
    - Generate FSM from the model. Ex.: Spec Explorer.
  - State/Transition based. Ex.: FSM, EFSM, Statecharts.
    - Traversal algorithms (state and transition coverage)
    - Model checking (algorithms based on state exploration; mutation analysis)
  - Property-based. Ex.: Obj
    - Rewriting rules
    - Constraint solving
  - Behaviour-based. Ex.: CSP
    - Trace analysis

Model Checking



Model Checking

- Whenever a property, expressed in temporal logic, does not hold in a system described as a FSM, model-checking tries to generate a counter-example
- When a counter-example is produced, it can be used as a test case – sequence of transitions in the FSM with inputs and expected outputs
- To be effective as a test-case generation technique, the properties about the system should be described in such a way that counter-examples produced by them can be used as test cases



Now that you have tests ...

- How do you know they are good enough?

How do you measure the quality?

- Capability to find defects
  - Particularly defects with higher risk
  - Risk = frequency of failure * impact of failure ≈ cost (of post-release failure)
- Capability to exercise multiple aspects of the system under test
  - Reduces the number of test cases required and the overall cost
- Low cost
  - Development: specify, design, code
  - Execution (fast)
  - Result analysis: pass/fail analysis, defect localization
- Easy to maintain
  - Reduce whole life-cycle cost
  - Maintenance cost ≈ size of test artefacts

(See also: "What Is a Good Test Case?", Cem Kaner, Florida Institute of Technology, 2003)

Test adequay/coverage criteria

- Adequacy criteria - Criteria to decide if a given test suite is adequate, i.e., to give us “enough” confidence that “most” of the defects are revealed
  - Used in the evaluation and in the design/selection of test cases
  - In practice, reduced to coverage criteria
- Coverage criteria
  - Requirements/specification coverage (black-box)
    - At least one test case for each requirement/specification statement
  - Model coverage
    - State-transition coverage
    - Use case and scenario coverage
  - Code coverage (white-box)
    - Control flow coverage (statement, decision, MC/DC coverage …)
    - Data flow coverage
  - Fault coverage / mutation testing

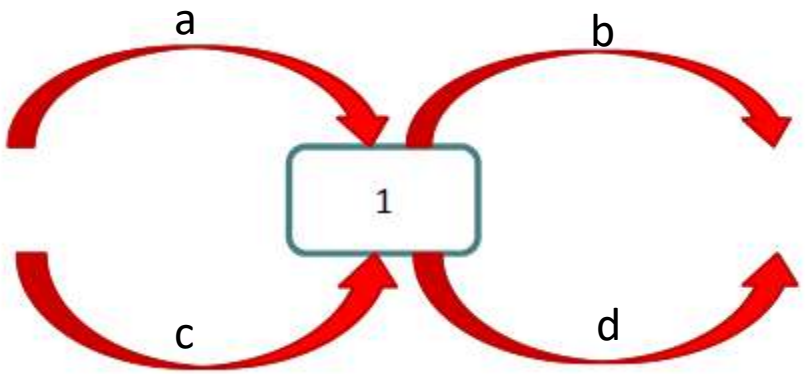
See also: “Software Unit Test Coverage and Adequacy”, Hong Zhu et al, ACM Computing Surveys, December 1997

Coverage analysis

- 1.Structural coverage criteria – aims to exercise code or model concerning some coverage goal
- 2.Data coverage criteria – aims to cover the input data space of an operation or transition
- 3.Fault-based criteria – aims to generate test suites appropriate for detecting specific kinds of faults
- 4.Requirements coverage criteria – aims to ensure that each requirement will be tested
- 5.Others
- …

Coverage analysis – FSMs

- Action coverage
- State coverage
- Transition coverage
- Transition pair coverage
  - Example with four possible pairs (ab,cd,ad,cb)



Coverage analysis – Code

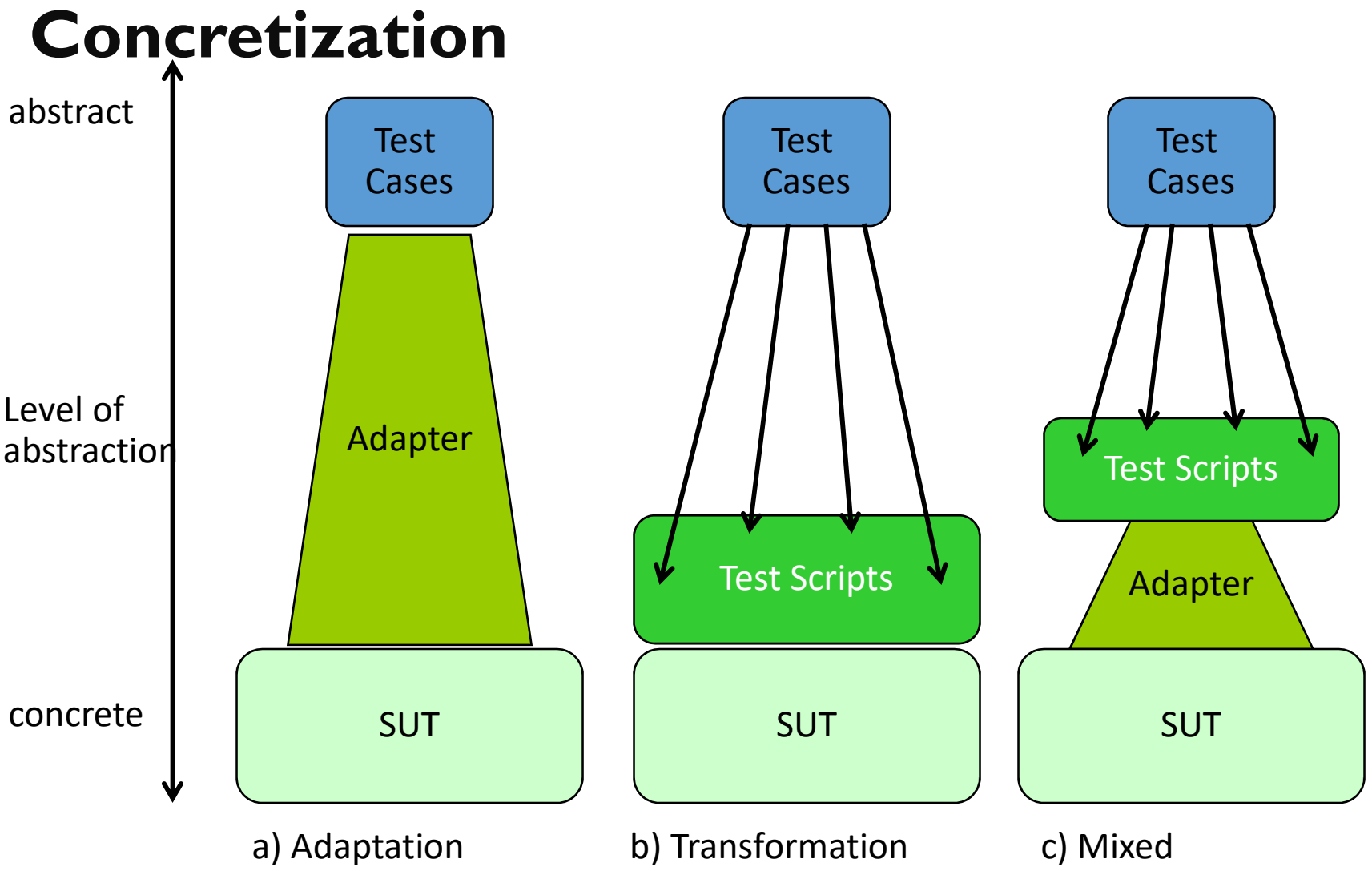
- Statement converage
- Decision coverage
- Condition coverage
- Condition and decision coverage
- MCDC
- …

MBT - Test case concretization

Mapping

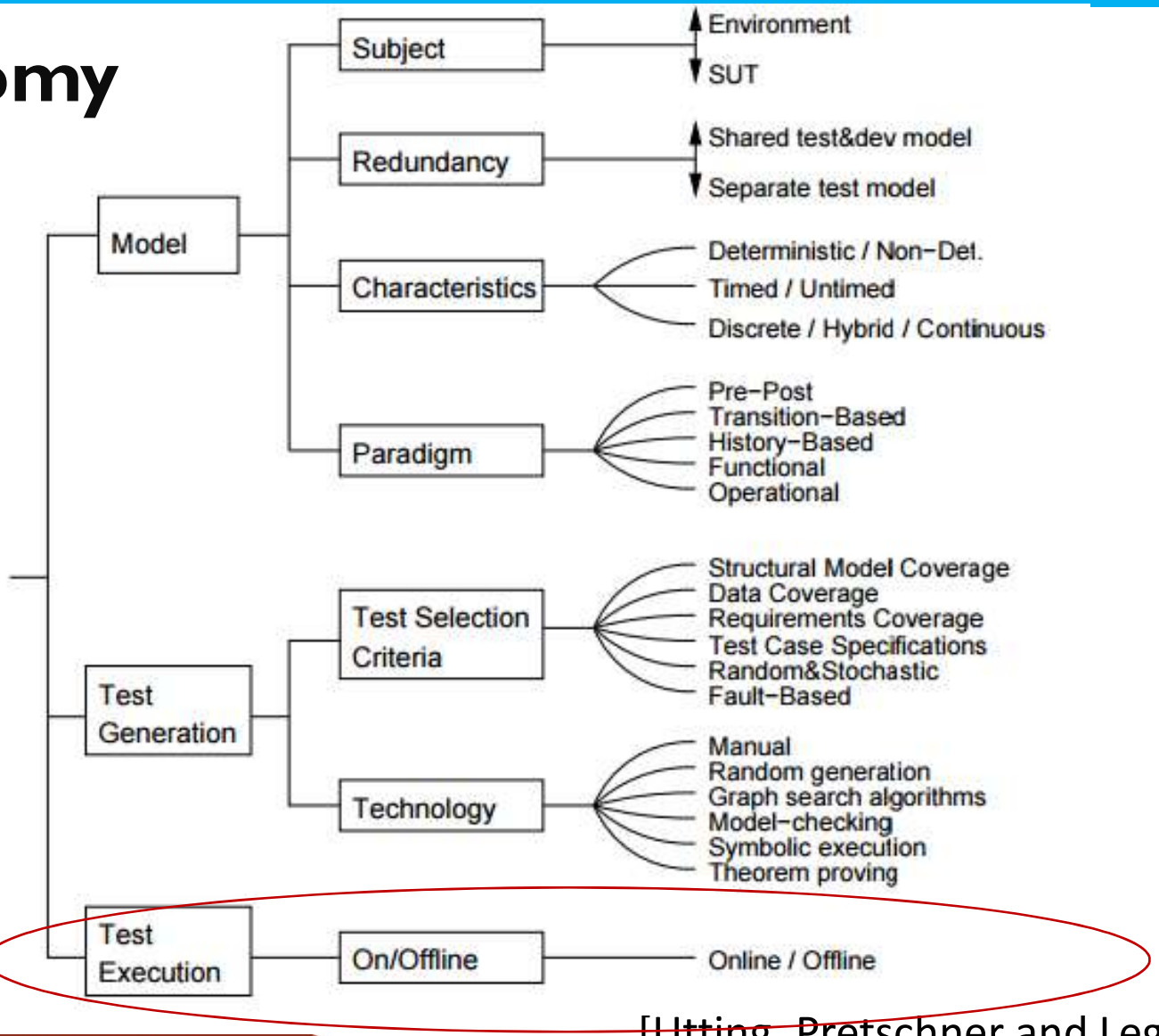
- Translate abstract test cases into concreate test cases
- Establish a mapping between abstract and concreate
- **API**
  - Map methods from the model to methods of the SUT
- **GUI**
  - Map methods of the API to Graphical User Interface objects
  - But then you have to identify them during test case execution!

Test case "concretization"



MBT - Test case execution

Taxonomy



[Utting, Pretschner and Legeard]

Execution

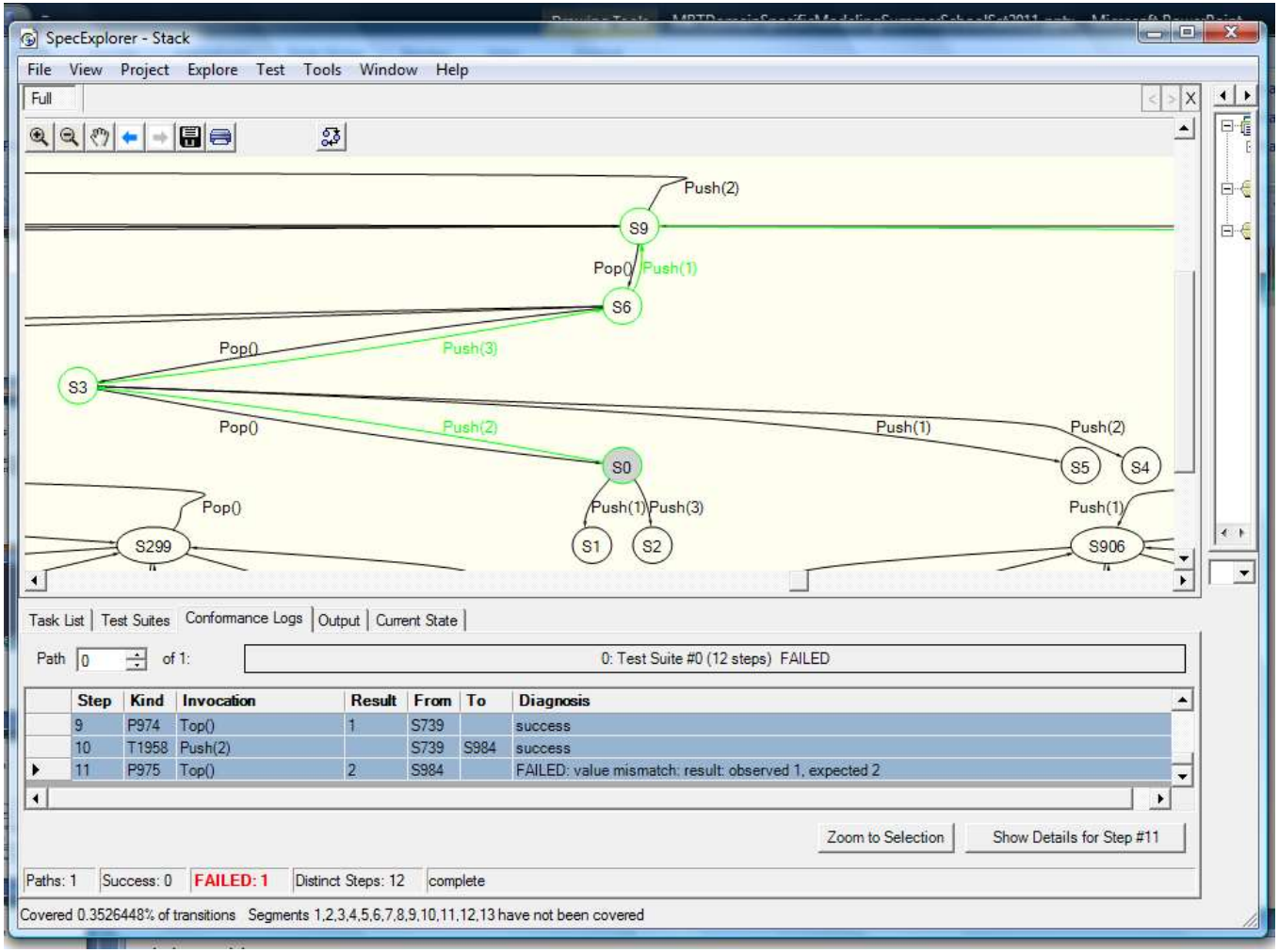
- **Lock-step** mode: results are compared after each step
- **Batch-oriented** mode: the test suite is run as a whole in the specification level, and expected results are kept in memory for later comparison with the results obtained from the execution of the implementation (which is performed in a different execution time instant)
  - One advantage of the batch-oriented way is the need to execute the model only once and not every time test cases are executed. The main drawback is the additional need of memory to keep the results expected
- **On-the fly** combines in a single algorithm the test case generation and execution and executes each operation as a “lock-step” in each level comparing results after each of those execution steps

MBT - Analysis

Locate the faults

- Mainly a manual activity
- Some tools provide information that may be useful to diminish the effort to find the origins of the failures detected
- 50% of the faults are in the code and 50% in the model!

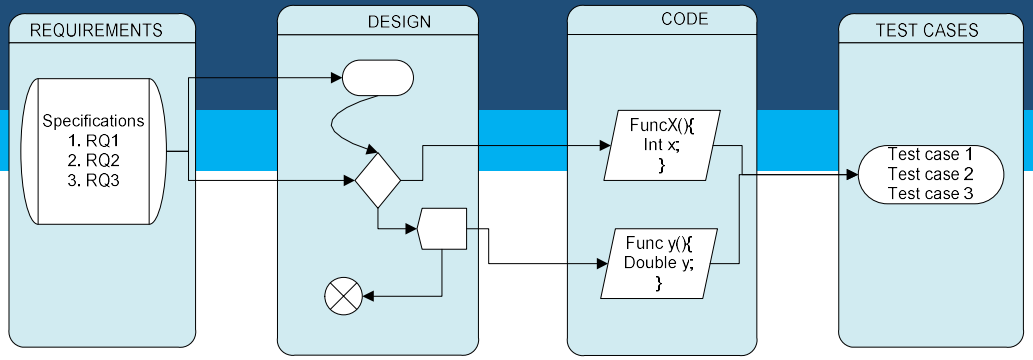
Spec Explorer example



Traceability

Problem statement

- "Achieving traceability between the test cases generated and requirements in a ***model based testing*** (MBT) scenario"



Traceability

- Ensure interrelation between different artifacts involved software engineering
- Enable to perform quick and easy impact analysis
- Enhance the product quality

SDLC Phase	**Artifacts**
• Specification	- requirements
• Design	- design models
• Implementation	- code (functions, libraries)
• Validation	- test case

Traceability would enable

- Measure relevance of the test case generated from model
- Assess how a modification in the model would effect the generated test case
- Provides the effective coverage

Summary

- It was more common on API testing
- Now we see GUI testing and mobile testing
- It is more common in industry nowadays
- It is a hot research topic
- There are some research problems still open