Model-Based Testing for Functional and Security Test

Fabrice BOUQUET

Summer School FOSAD
3 - 7th July 2012

OUTLINE

I. Introduction of test
II. Functional Testing
III. Model-Based Testing
IV. Model-Based Testing and Security
V. Evolution of system

OUTLINE

I. Introduction of test
   i. Software engineering
   ii. What is the test?
   iii. Kinds of test
II. Functional Testing
III. Model-Based Testing
IV. Model-Based Testing and Security
V. Evolution of system

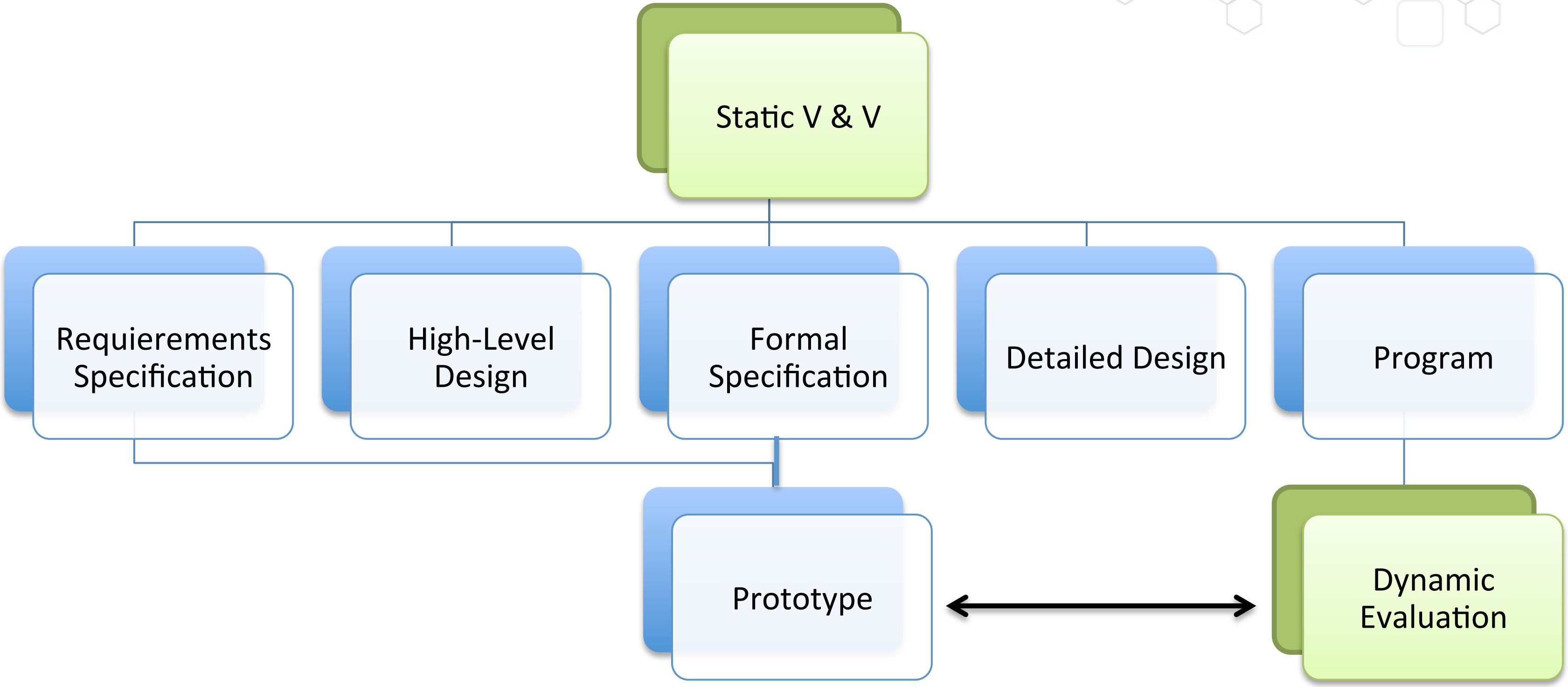
Validation & Verification

V & V:

- Validation:
  - Are we building the right product?
  - The software should do what the user really requires (conform to the requirements)

- Verification:
  - Are we building the product right?
  - The software should conform to its model artifacts

Validation & Verification



V & V Methods:

- Static Test: reviews of code, of specification, of design documentation
- Dynamic Test: execute code to ensure correctness of functionality
- Symbolic Verification: Run-time checking, Symbolic execution…
- Formal Verification: Proof, model-checking from formal model

Introduction of software testing

Motivation of the Test:

- Cost of bug (Ariane 5, German smart card, Orange cell phone network…)
- Complexity of the behaviors

Error
Specification, Design, Development
↓
Default in software
↓
Failure operation

Some numbers *(source - National Institute of Standards and Technology)*:

- Cost of computer bugs: ~60 billion $ / year
- 22 billion must be save if test process is increase

Testing focus areas of IT organizations

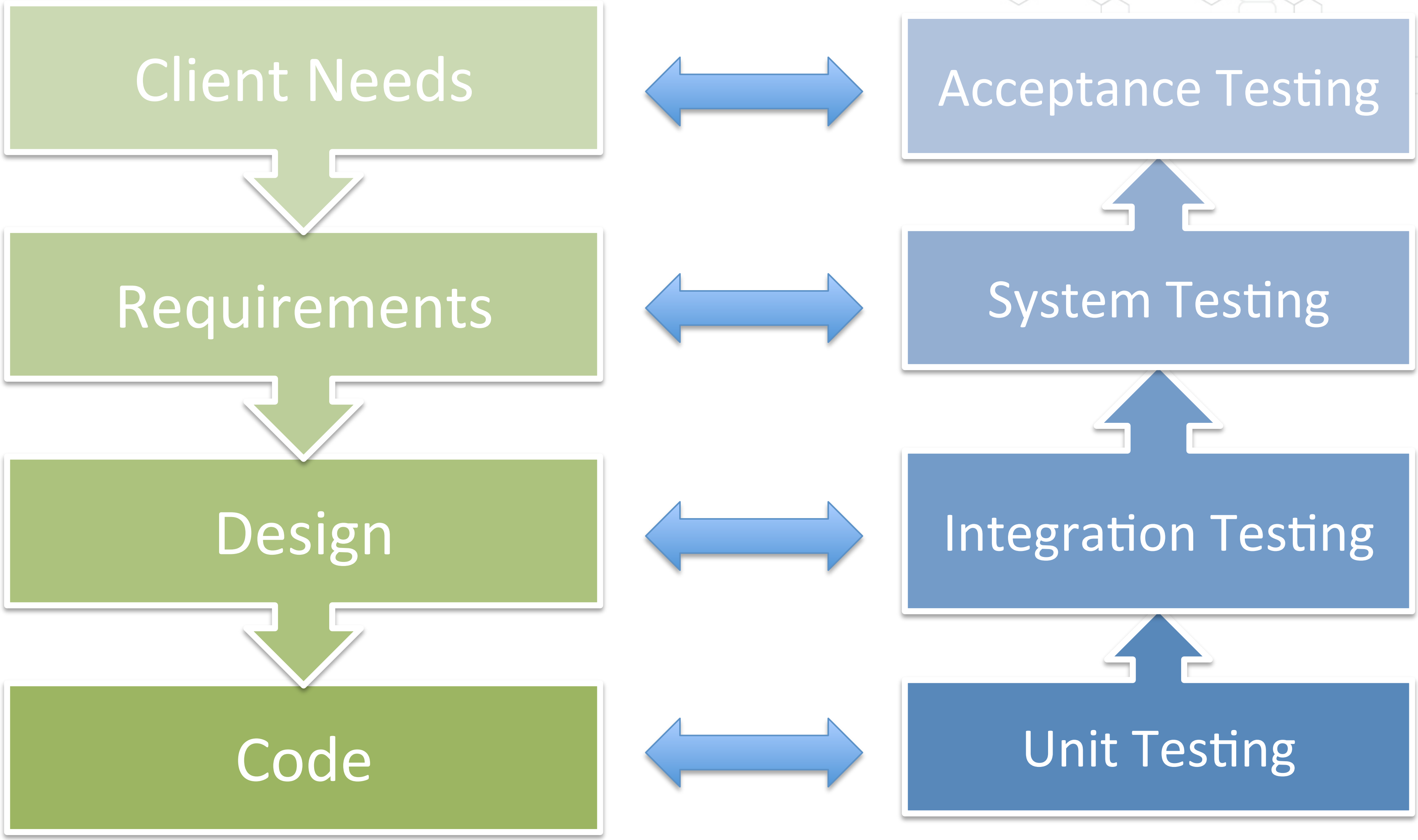
QA/Test function maturity: shift from tactical ad hoc process to a more strategic & centralized approach

Top Four Focus Area – Across Western Europe

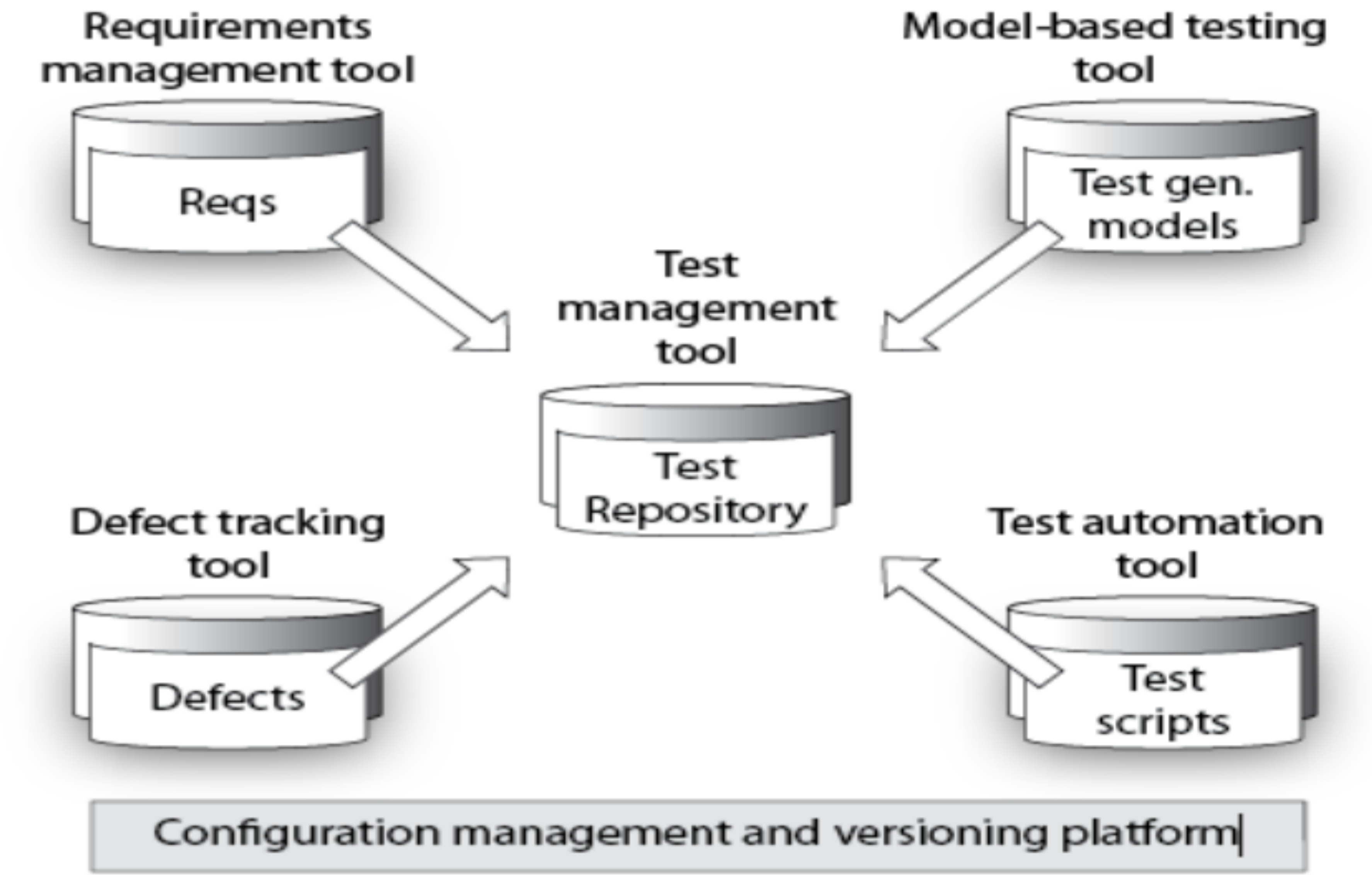
1. Choosing a testing methodology to address agile/component based development life cycle
2. Provide automated test coverage to build agility in testing
3. More focus on the non-functional aspects like performance, availability, security, etc.
4. Having a test strategy that optimizes use of testing services (traditional and cloud based)

Source IDC - European Services, Enterprise Application Testing Survey, March 2011

Development Life Cycle & Testing Levels



Testing tools



Test Definition

IEEE Std 829:

- “The process of analyzing (execution or evaluation) a software item to detect the differences between existing and required conditions (that is, bugs), and to evaluate the features of the software item.”

G. Meyers (The Art of Software testing):

- “Test is the process of executing a program (or part of a program) with the intention of finding errors.”

Edsgar W. Dijkstra (Notes on Structured Programming)

- “Testing can reveal the presence of errors but never their absence.”

Test Practices

Test is an activity of test validation:

- "Does Software realize the right thing and thing right?"

Not very popular activity in company

Difficulties of a psychological or "cultural" nature:

- The test is a destructive process: a good test is a test which finds an error
- The activity of programming is a constructive process - one seeks to establish correct results

However, the test is a central activity:

- it is the principal vector of the improvement of the quality of the software
- Can be represent until 60% of complete effort for software development
  - 1/3 during software development
  - 2/3 during software maintenance

Example – Register form

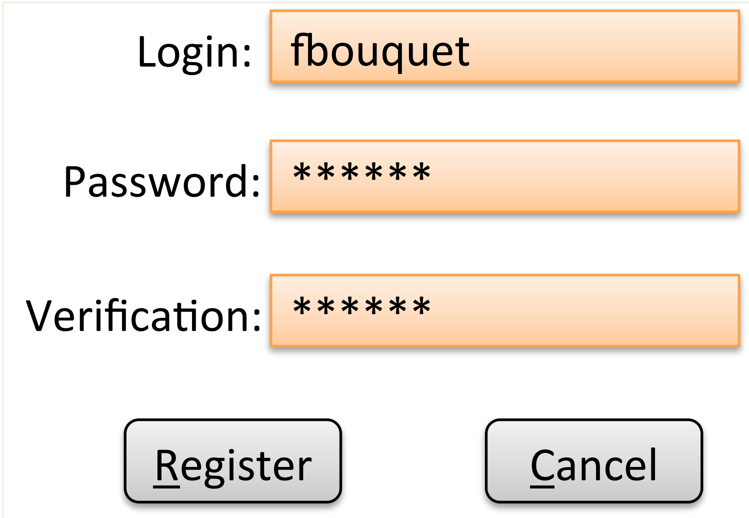
Specification:

- "Let realize a secured register form for a web site"

Attempt:

- Provides the test cases for this specification

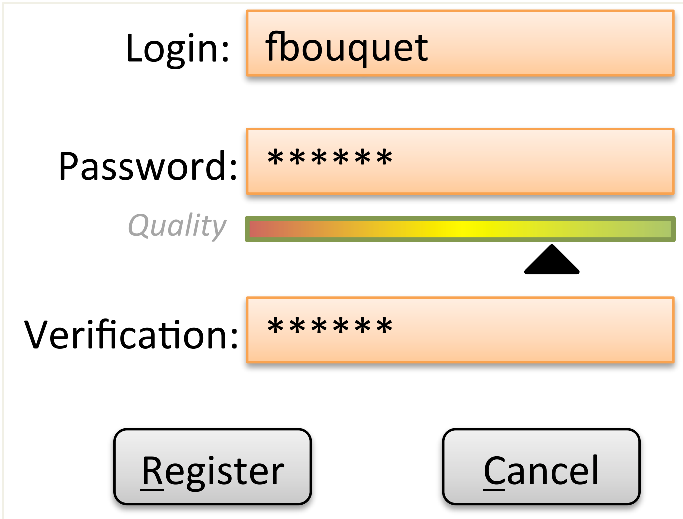
Example – Register – Level 0



5 Cases: *10 tests*

- Login (not) empty (2)
- Login (doesn't) exist (2)
- Password (not) empty (2)
- Password and Verification (retyped password) is (not) the same (2)
- Protocol http(**s**) (2)

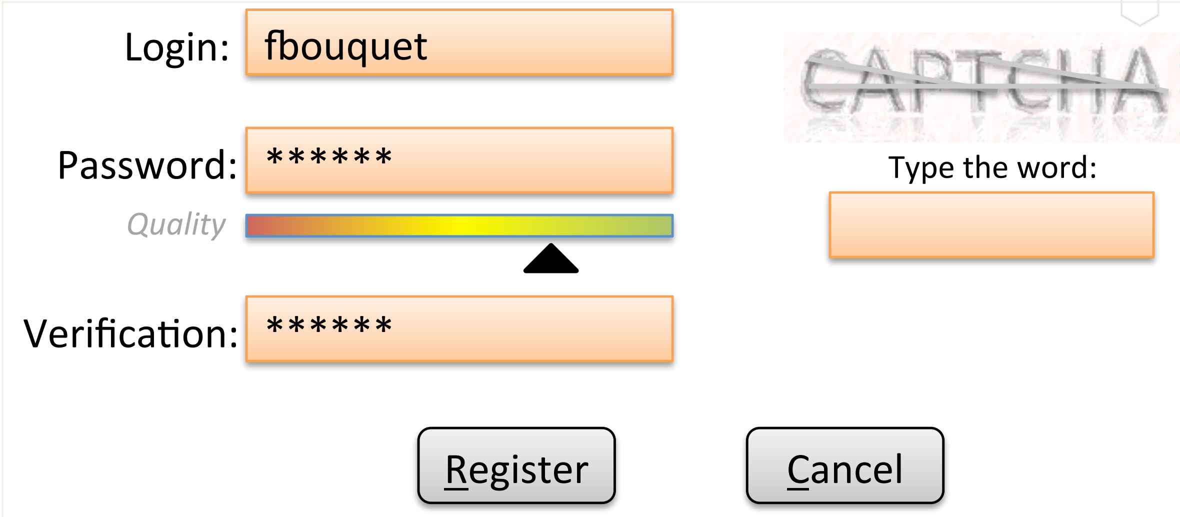
Example – Register – Level 1



6 Cases: *13 tests*

- Login (not) empty (2)
- Login (doesn't) exist (2)
- Password (not) empty (2)
- Password and Verification (retyped password) is (not) the same (2)
- Protocol http(**s**) (2)
- Verify quality / Robustness of the password (1 by level) *<- secure the account*
  - poor, average, good…

Example – Register – Level 2



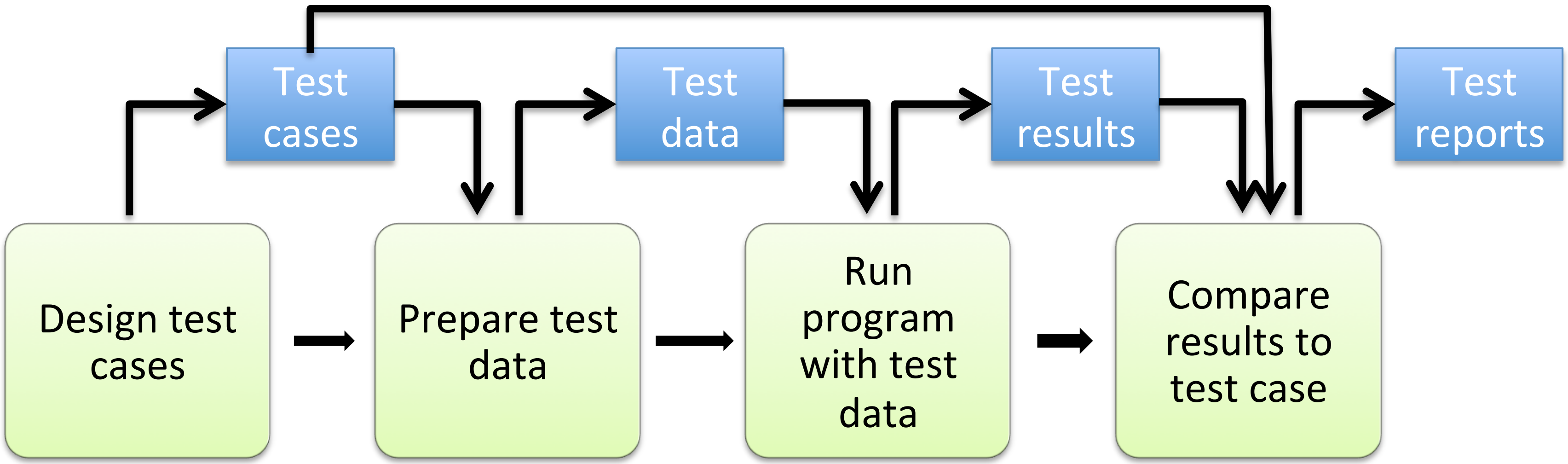
7 Cases: *15 tests*

- Login (not) empty (2)
- Login (doesn't) exist (2)
- Password (not) empty (2)
- Password and Verification (retyped password) is (not) the same (2)
- Protocol http(**s**) (2)
- Verify quality / Robustness of the password (3)
- Verify the (no) human registering (2) *<- secure from robot*

Activities of dynamic test

4 fallowing activities:

- Select a test: choose a subs-set of all possible inputs of the software
- Submit the test to the software
- Analyze the result: Decide if test is successed or failed (verdict):
  - Fail
  - Pass
  - Inconclusive
- Evaluate the quality and the relevance of the tests (to determine the end of test phase).



Structural Test (white box)

Data:

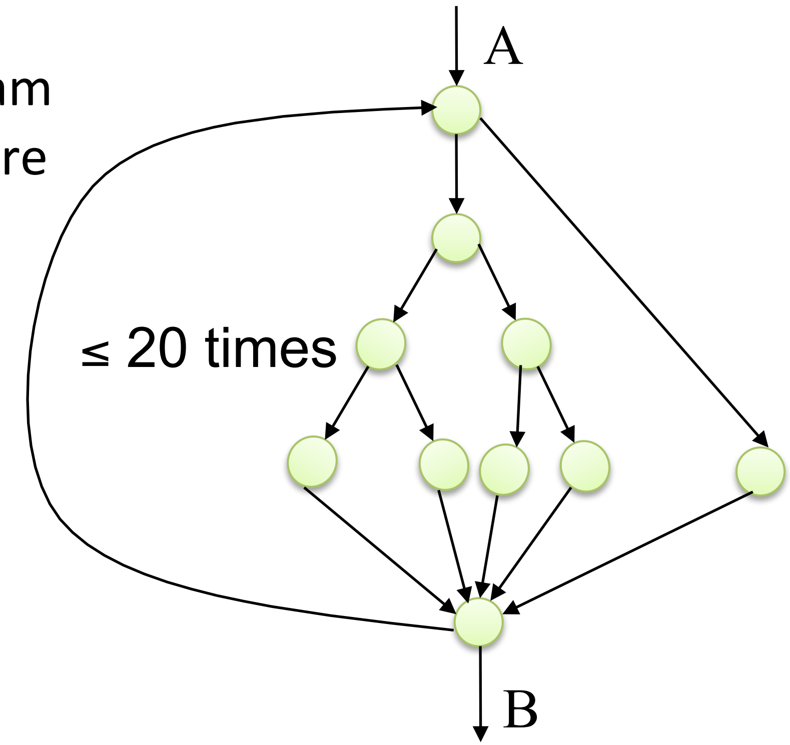
- Test data is produce from source code analyses

Actions:

- Examines the internal design of the program
- Requires detailed knowledge of its structure

Coverage criteria:

- Path,
- Branch (condition),
- Statement (node, instructions…)
- …



Control Flow Graph of program

Functional Test (black box)

Data:

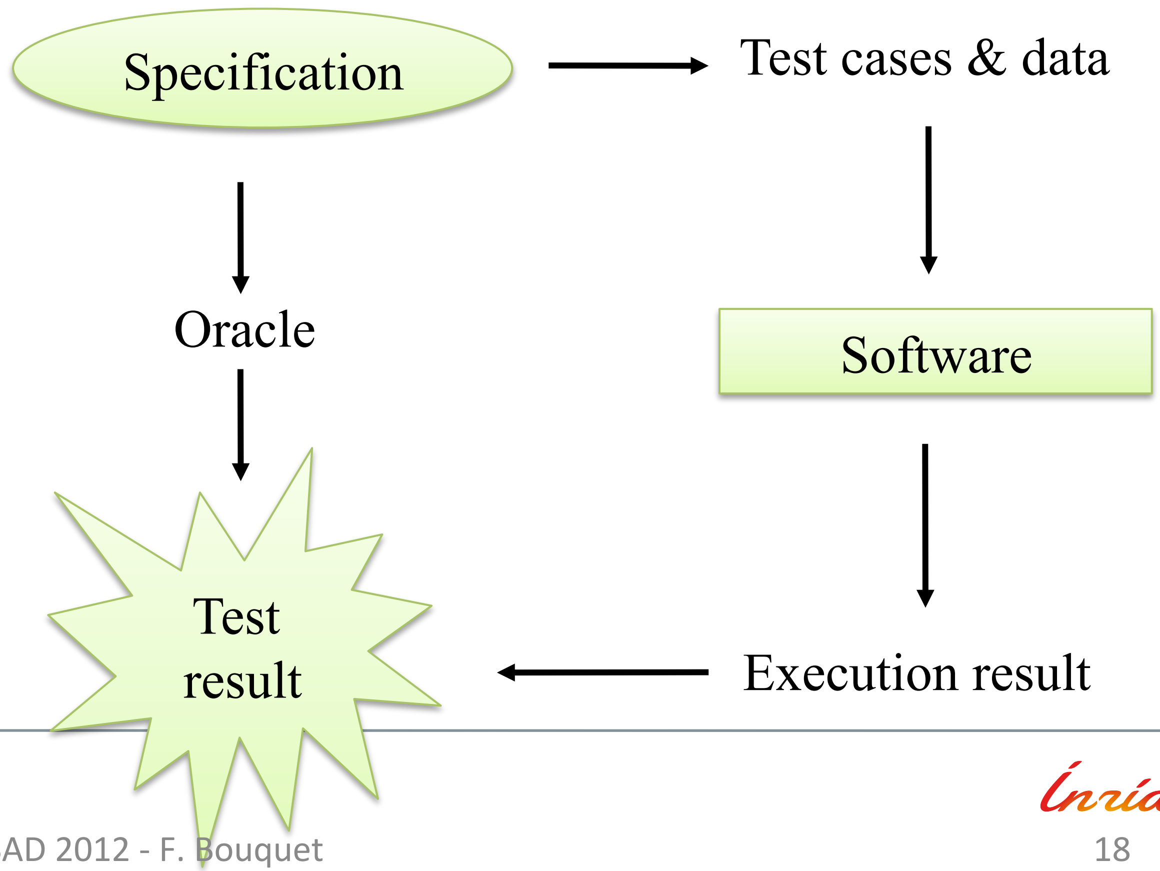
- Test data is extract of specification

Actions:

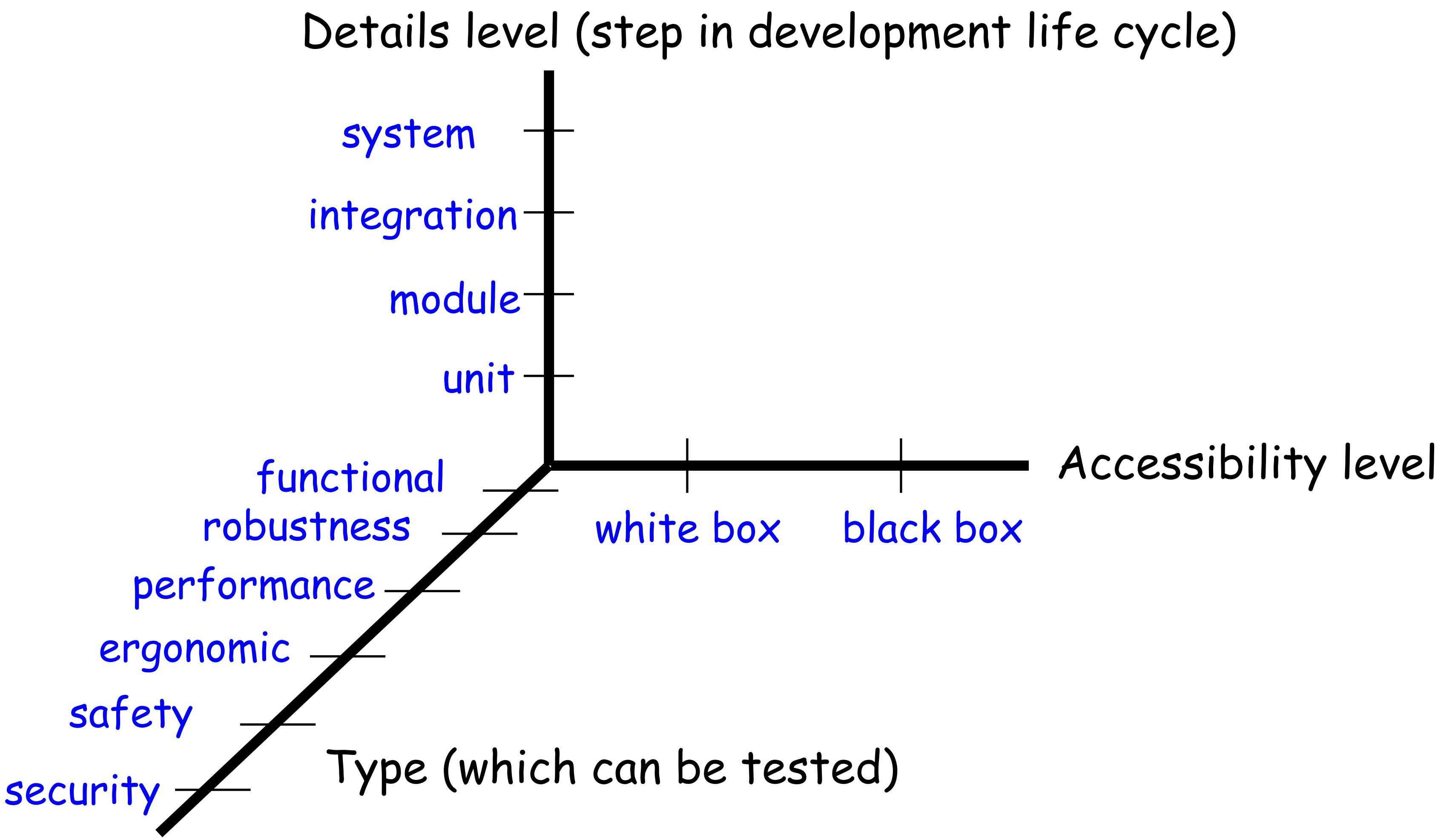
- Designed without knowledge of the program's internal structure and design
- Based on functional requirements

Coverage criteria:

- Requirements
- Functionality



Kinds of Test [J. Tretmans]



OUTLINE

I. Introduction of test
II. Functional Testing
    i. Partition Analysis of input domains
    ii. Combination test
    iii. Random and stochastic Test
    iv. Model-Based Testing
III. Model-Based Testing
IV. Model-Based Testing and Security
V. Evolution of system

Methods of Functional Testing

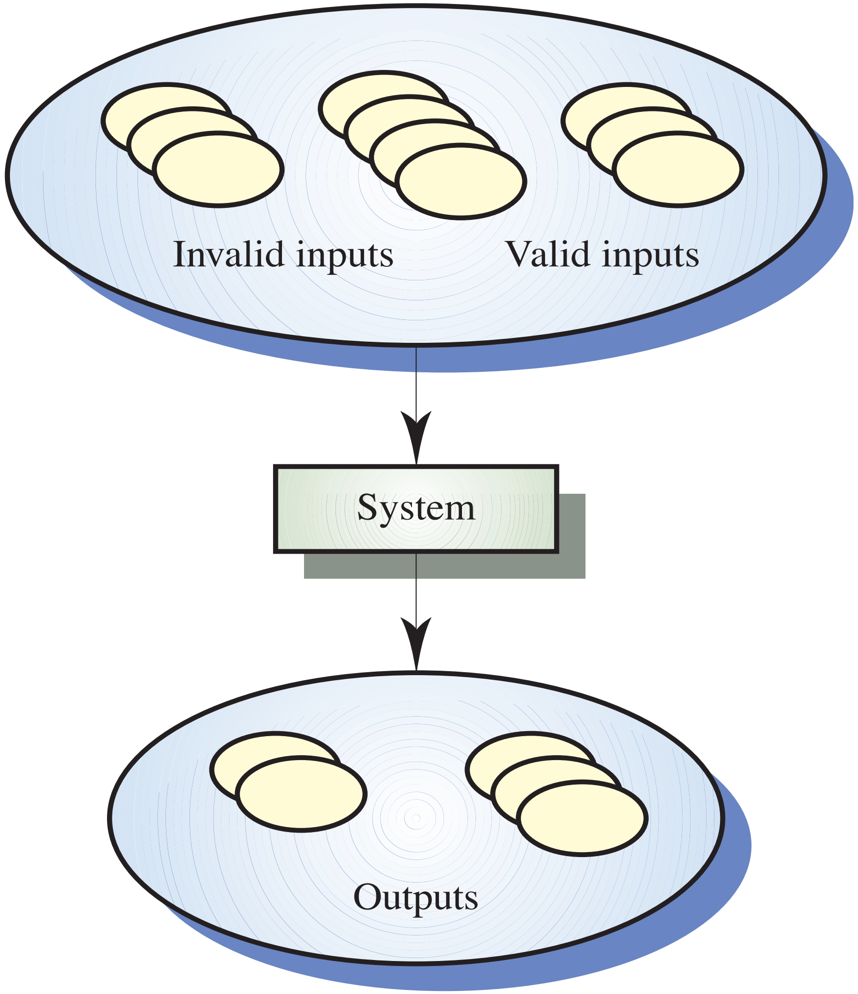
Subject:

- The functional test aims at examining the functional behavior of the software and its conformity with the specification of the software
- To Found / Select the Test Data (TD)

Methods for functional testing

- Partition Analysis of input domains
  - Reduce the number of possible values
  - Strategy to choice a representative value (bounded, middle, random…)
- Combination test
- Random and stochastic Test
- Model-Based Testing

Equivalence Partitioning



Class of equivalence:

- A class of equivalence corresponds to a set of data of tests supposed to test the same behavior, i.e. to activate it
- The definition of equivalent classes allows to translate an infinity number of input data to a finite number and limited test data

Partition Analysis – Target

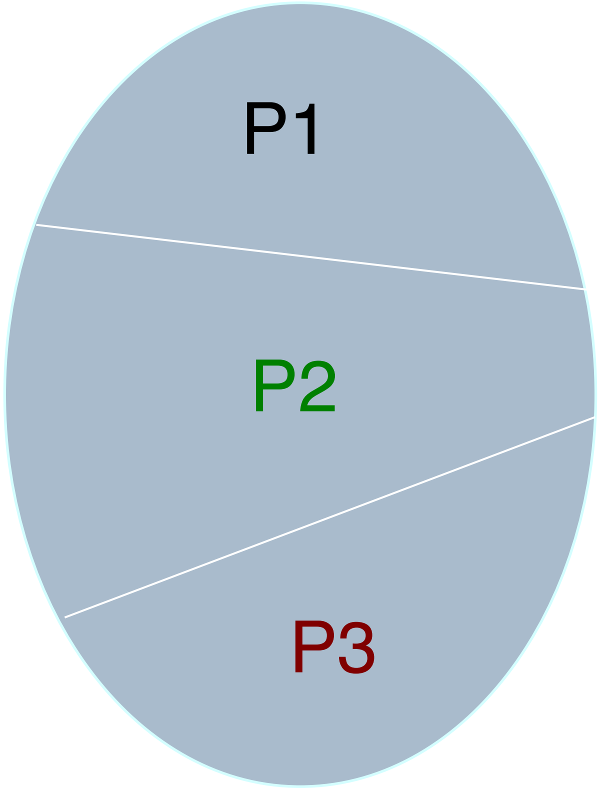
- Computation of the test targets by coverage …
  - … of behaviors
    - Extraction of the behavior from a method
    - Target = before part of the behavior – activation part
    - Under hypothesis of the classes' invariants

Partition Analysis – Example

Invariant
$x \in -1000 .. 1000$
Pre_{op}
$x \leq 100$
$Post_{op}$
IF $x \leq 0$ **THEN** y := default
ELSE IF $x \leq 40$
THEN y := low
ELSE y:= high
END END

Behavior classes

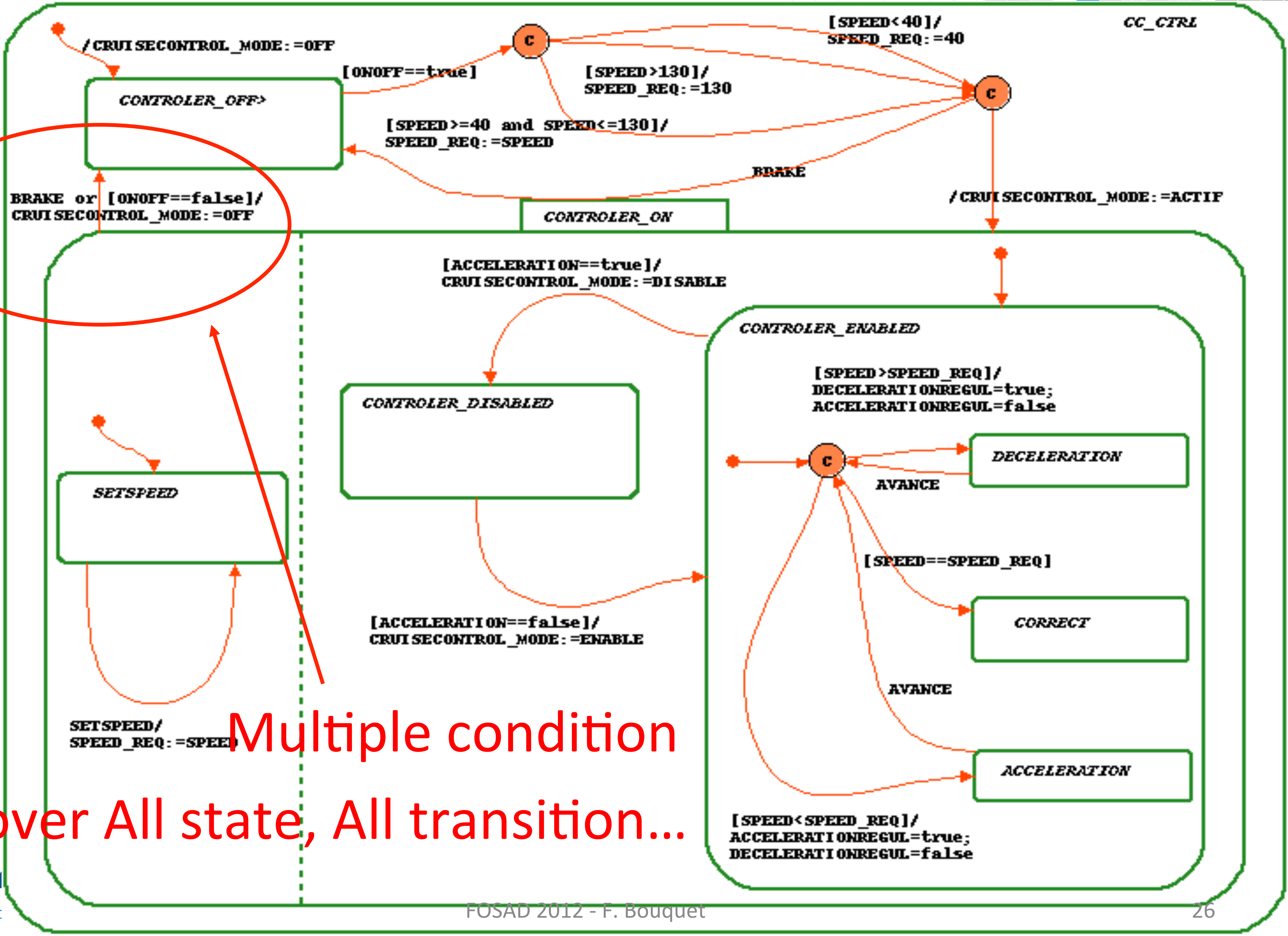
P1: $x \leq 0$
P2: $x > 0 \wedge x \leq 40$
P3: $x > 40 \wedge x \leq 100$



Set of system states which allows to activate the operation

Partition Analysis – Target

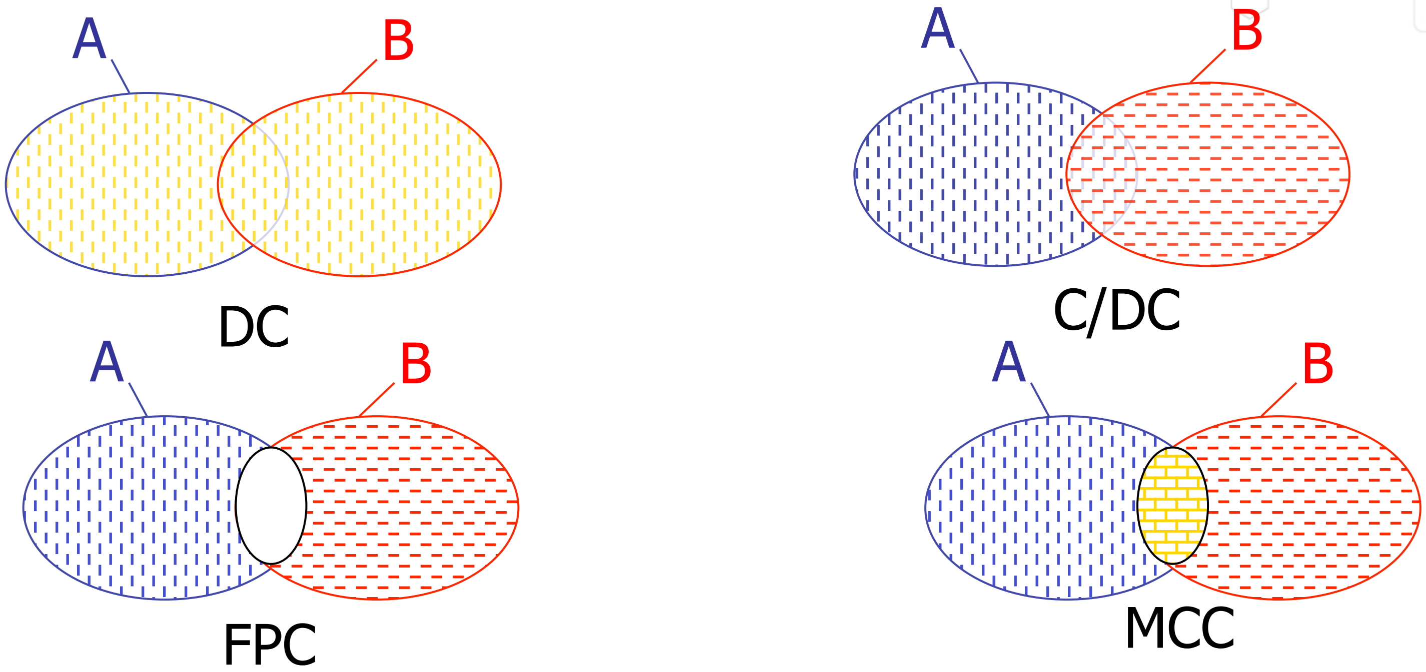
- Computation of the test targets by coverage …
  - … of behaviors
    - Extraction of the behavior from a method
    - Target = before part of the behavior – activation part
    - Under hypothesis of the classes' invariants
  - … of decisions
    - Rewriting of the disjunctions in the Target



Multiple condition

Cover All state, All transition…

Partition Analysis – Decision coverage

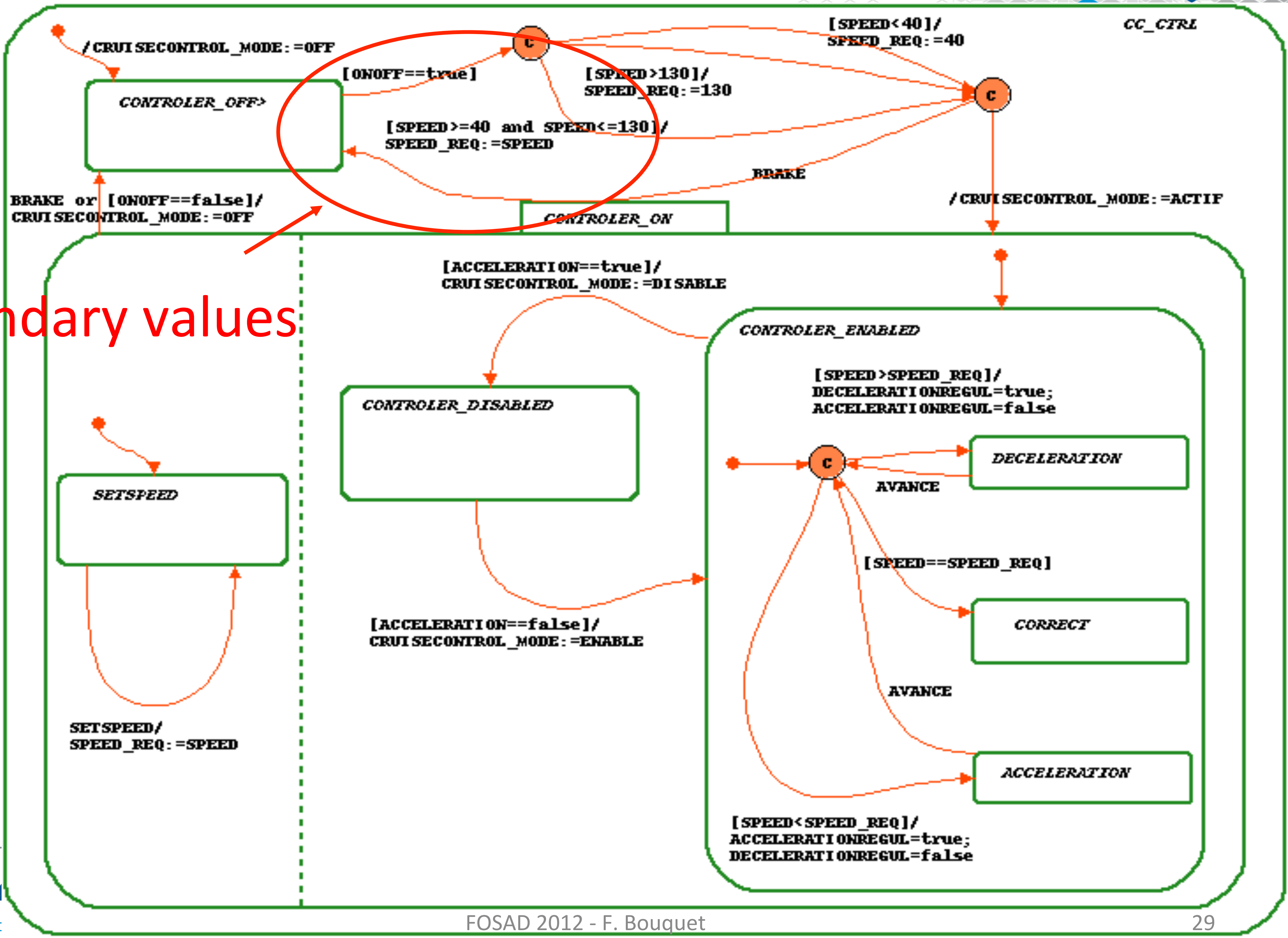


Condition (A ∨ B)

- Statement Coverage & Decision Coverage → A ∨ B
- Decision/Condition Coverage → A ✤ B
- Full Predicate Coverage → A ∧ ¬B ✤ ¬ A ∧ B
- Multiple Condition Coverage → A ∧ B ✤ A ∧ ¬ B ✤ ¬ A ∧ B

Partition Analysis – Target

- Computation of the test targets by coverage …
  - … of behaviors
    - Extraction of the behavior from a method
    - Target = before part of the behavior – activation part
    - Under hypothesis of the classes' invariants
  - … of decisions
    - Rewriting of the disjunctions in the Target
    - Satisfy criteria (SC/DC, FPC, MCC)
  - … of data
    - Boundary values



Boundary values

Partition Analysis - Method

3 Steps:

- For each input, calculation datum of classes of equivalence on the fields of values
- Choice of a representative value of each class of equivalence
- Composition by Cartesian product on the whole of the data input to establish the TD.

Rules:

- If the data is defined with an interval, It build:
  - One class for lower values (out of interval)
  - One class for upper values (out of interval)
  - N classes for valid values
- If the data is defined with a set of values, it build:
  - One class with empty set
  - One class with value outside of set
  - N classes for valid values
- If the data is defined with the constraints, it build :
  - One class without the constraints satisfied
  - One class with the constraints satisfied

Bounded test data

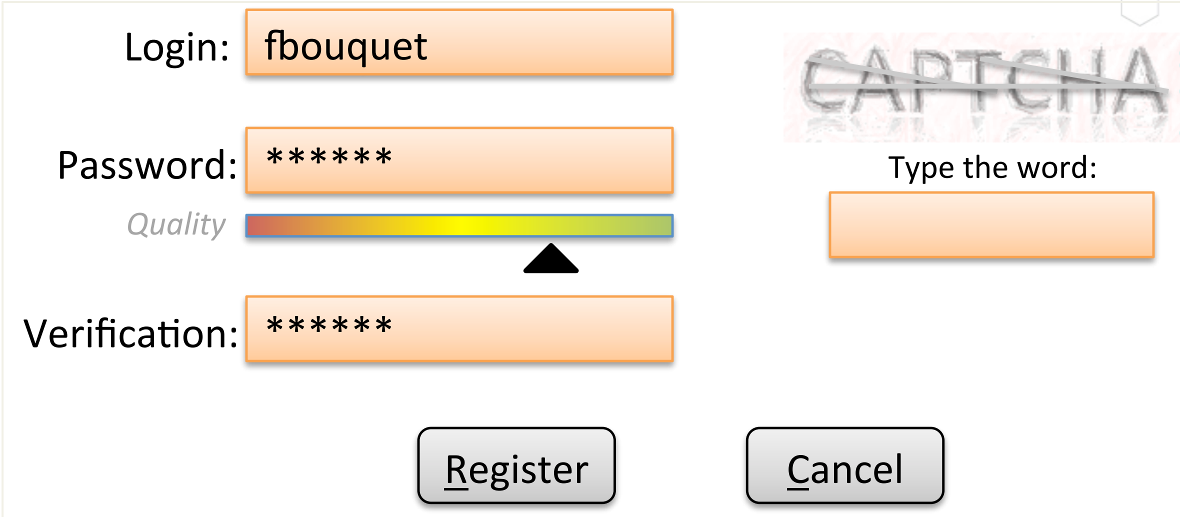
Idea:

- It is interested at the boundaries of the intervals for the domain of the inputs

Examples:

- for each interval, one keeps the 2 values corresponding to the 2 limits, and N values corresponding to the values of the limits with minus/plus delta:
  - $n \in 3 .. 15 \Rightarrow v1 = 3, v2 = 15, v3 = 2, v4 = 4, v5 = 14, v6 = 16$
- if the variable belongs to an ordered set of values, we choose the first, the second, before the last and the last and one outside:
  - $n \in \{-7, 2, 3, 157, 200\} \Rightarrow v1 = -7, v2 = 2, v3 = 157, v4 = 200, v5 = 300$
- if a condition on input choice the minimum and maximum number of values, and test for numbers of values except limits invalid:
  - Input file contains 1 to 255 records, we built files with: 0, 1, 255 and 256 records.
- Object typed data *p* with a static type *C* :
  - *null* reference
  - *this* reference (if *\typeof(this)* <: *\type(C)*)
  - Object *p* such that : *p != null && p != this && \typeof(p) == \type(c)*
  - Object *p* such that : *p != null && p != this && \typeof(p) < : \type(c)*
  - Object *p* such that : *p == p'* with *p'* an other compatible object

Bounded test on example



Register form variables:

- Login: empty, very long name (more than 256), existing login, 'valid' login
- Password: empty, very long string, same of login, poor, average, good login
- Password verification: not the same of Password, the same
- Captcha: the good string, not the good

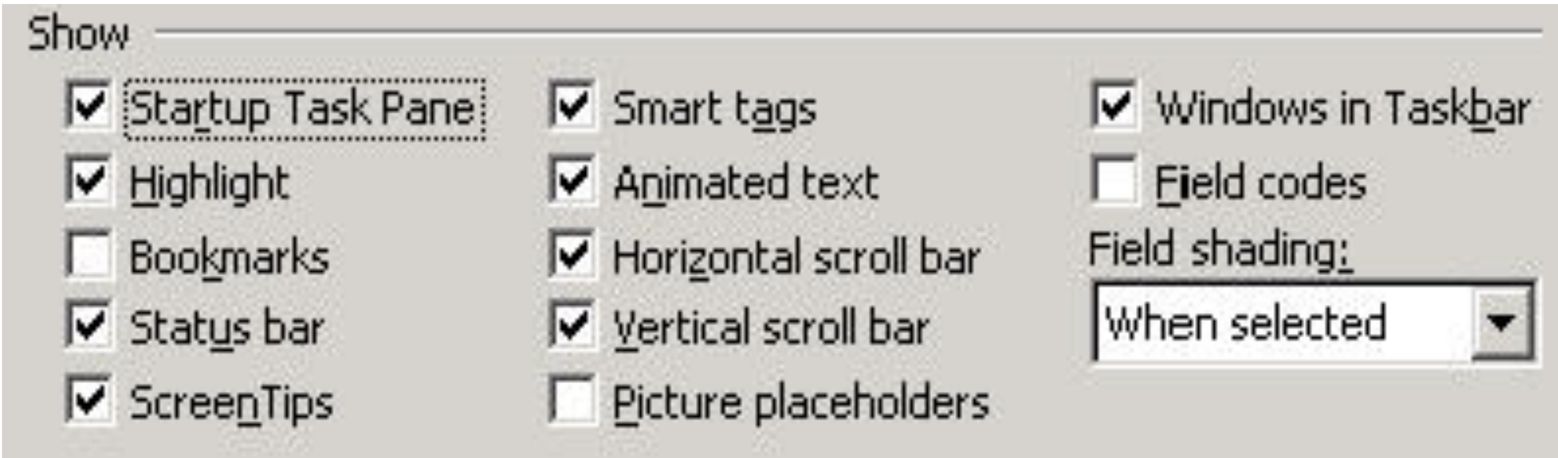
Combination test

Risk:

- The combinations of all input values give place to combinative explosion.

Example:

- 2 Inputs defined as integer: $2^{32}*2^{32} = 2^{64} =$ 18 000 000 000 000 000 000
- Preference parameter in MS Office



➔ 2^{12} * 3 (12 check boxes and 3 entries (in pull-down menu) = 12 288

Combination test (Pair-wise)

Aim:

- To test a fragment of the combinations of values which guarantee that each combination of 2 variables is tested.
- Majority of fault can be detect with combination of 2 values.

Example:

- 4 variables with 3 values

OS	Network	Printer	Application
XP	IP	HP35	Word
Linux	Wifi	Canon900	Excel
Mac OS	Bluetooth	Canon-EX	Powerpoint

All combination: 81

All pairs: 9

Pairwise

9 test cases:

- Each combination of 2 values is tested:

Case	OS	Network	Printer	App
1	XP	Bluetooth	Canon-EX	Powerpoint
2	Mac OS	IP	HP35	Powerpoint
3	Mac OS	Wifi	Canon-EX	Word
4	XP	IP	Canon900	Word
5	XP	Wifi	HP35	Excel
6	Linux	Bluetooth	HP35	Word
7	Linux	IP	Canon-EX	Excel
8	Mac OS	Bluetooth	Canon900	Excel
9	Linux	Wifi	Canon900	Powerpoint

Combination test (Pair-wise)

N-wise:

- This approach Pair-wise can be use with triplets, quadruples,…
  But the number of tests increase quickly

Reference site (articles and tools):

- http://www.pairwise.org/default.html

Lacks:

- the choice of the combination of values is perhaps not that which detects the bug…
- the expected result of each test must be provided manually

Random Test

Principle:

- Use a function to choose the TD to select :
  - Random function to take a value in the domain of the input
  - Statistic law to take the value

Example:

- Sampling of 5 in 5 for an input datum representing a distance,
- Use of a law of Gauss for a data representing the size of the individuals,

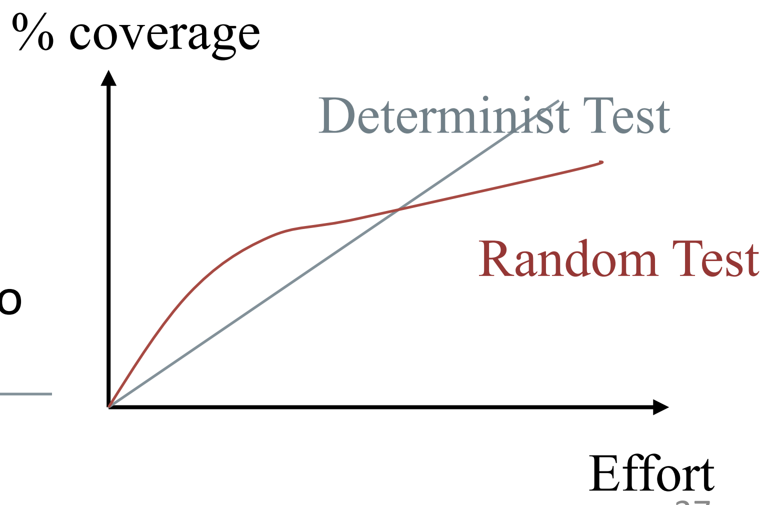
Pros:

- easily automatable for the selection of the cases of test (more difficult for the expected result)
- objectivity of the DT.

Cons:

- Blinding research,
- Difficulties to produce very specific behaviors

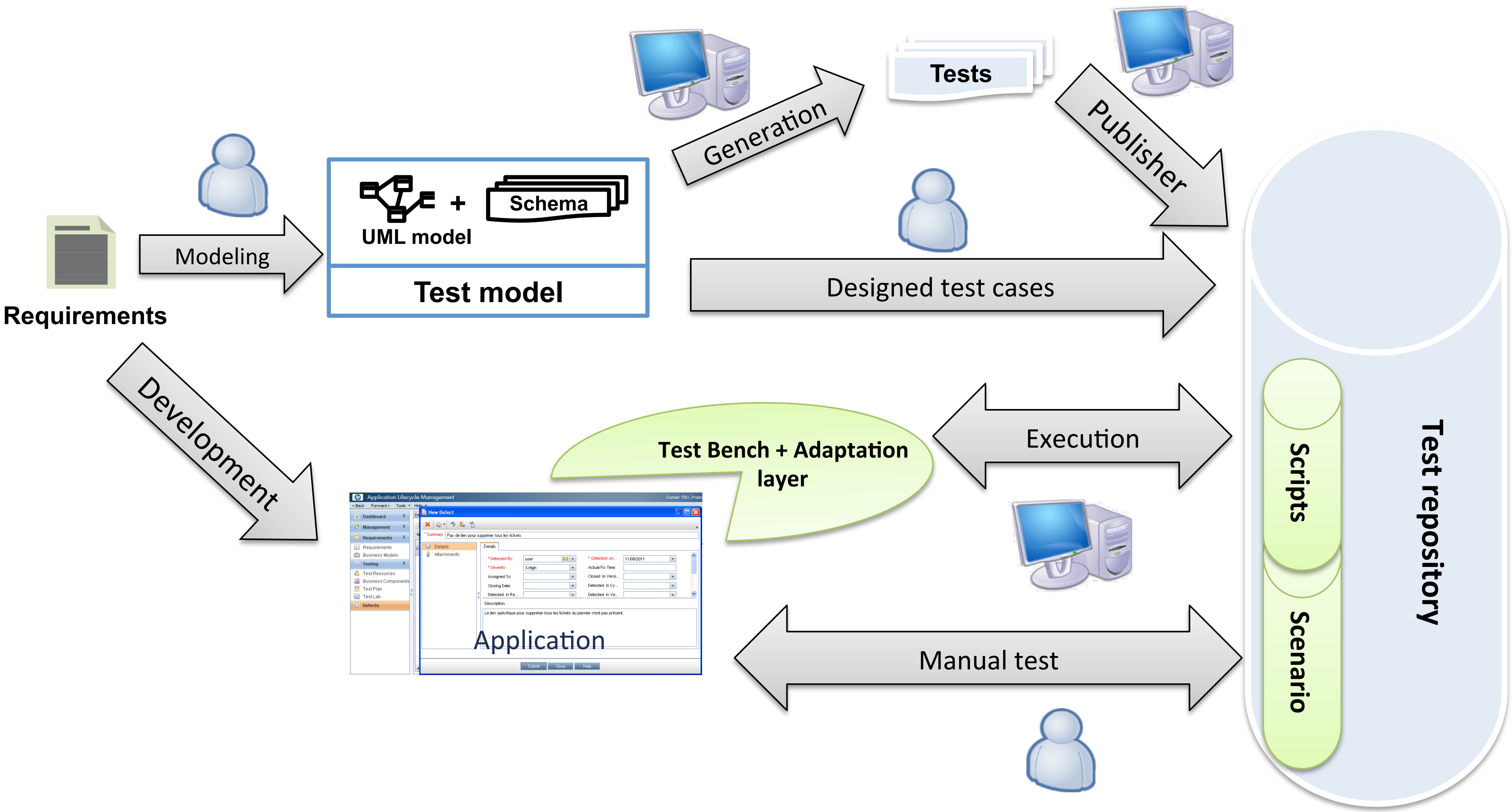
- The studies show that the statistical test makes it possible to quickly reach 50% of the objective of test but which it has tendency to reach a maximum then.



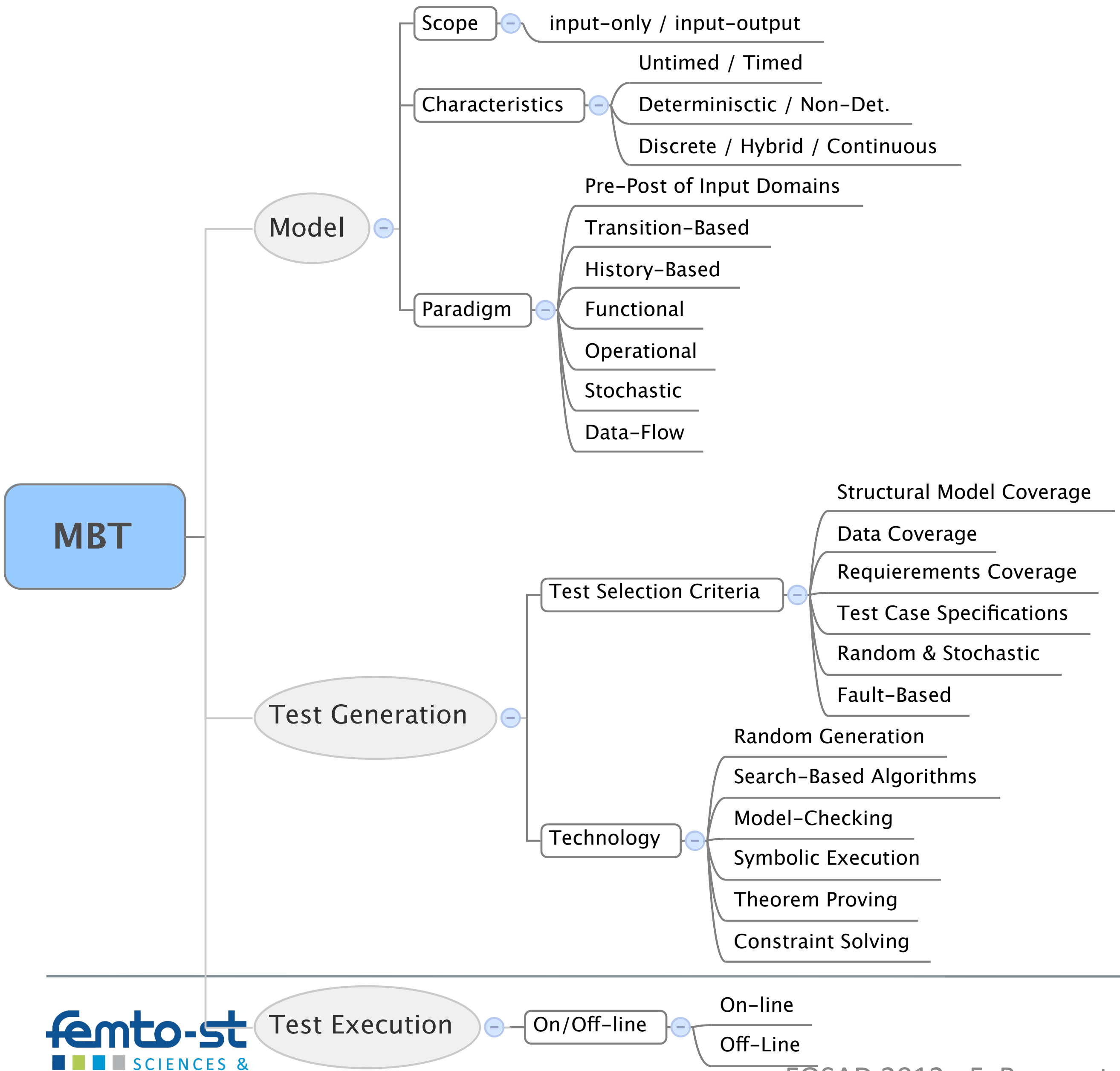
OUTLINE

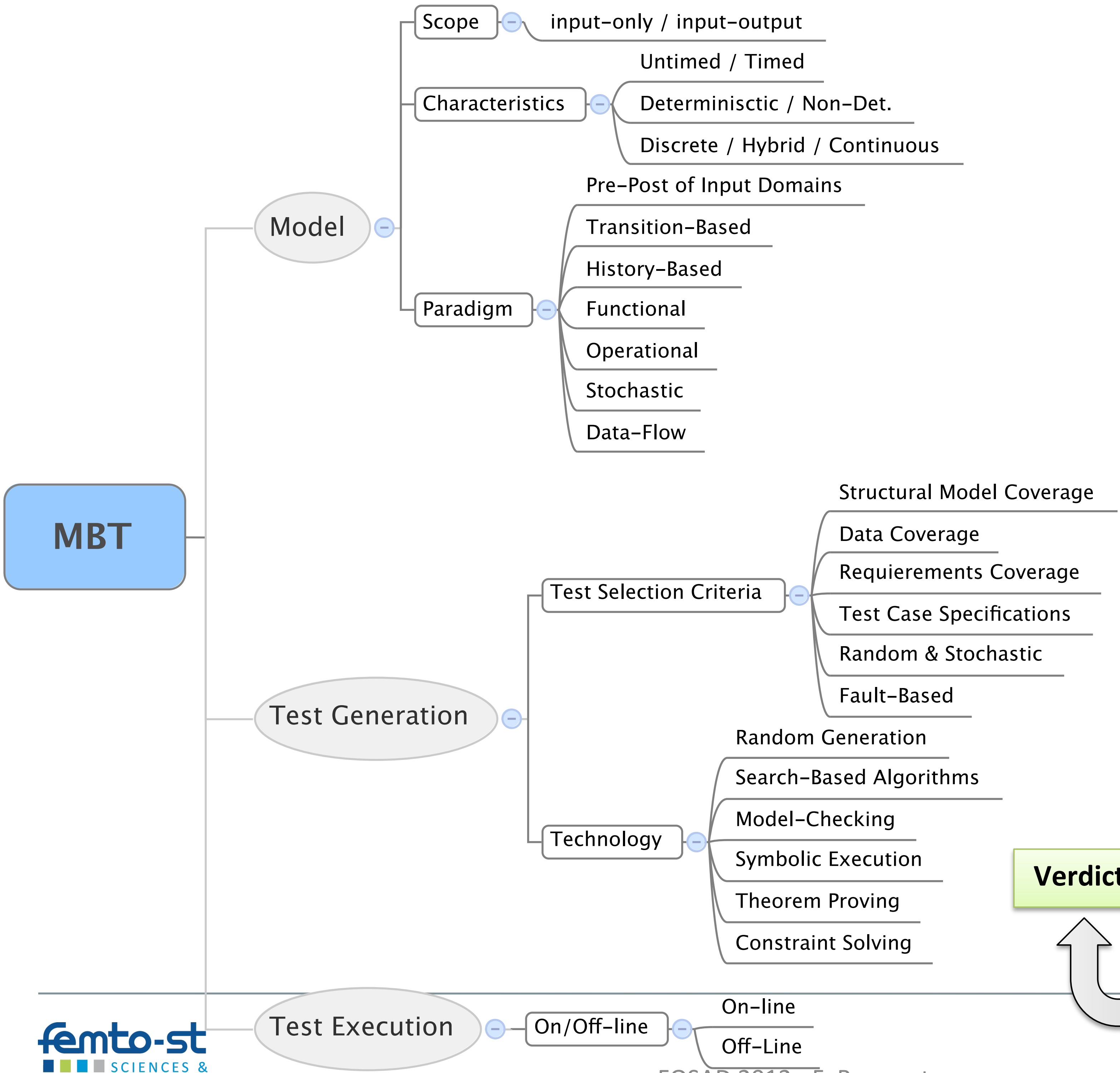
I. Introduction of test
II. Functional Testing
III. Model-Based Testing
    i. Process
    ii. Off-line
    iii. On-line
IV. Model-Based Testing and Security
V. Evolution of system

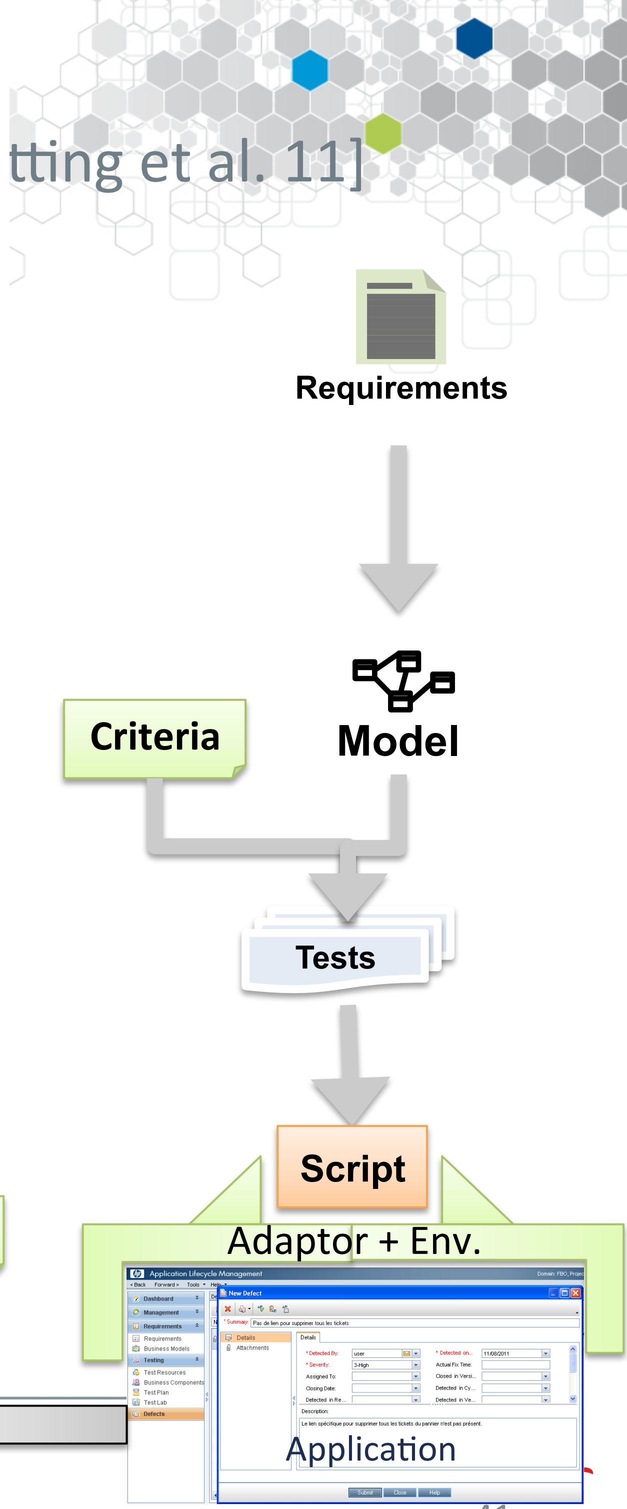
Model-Based Testing architectures



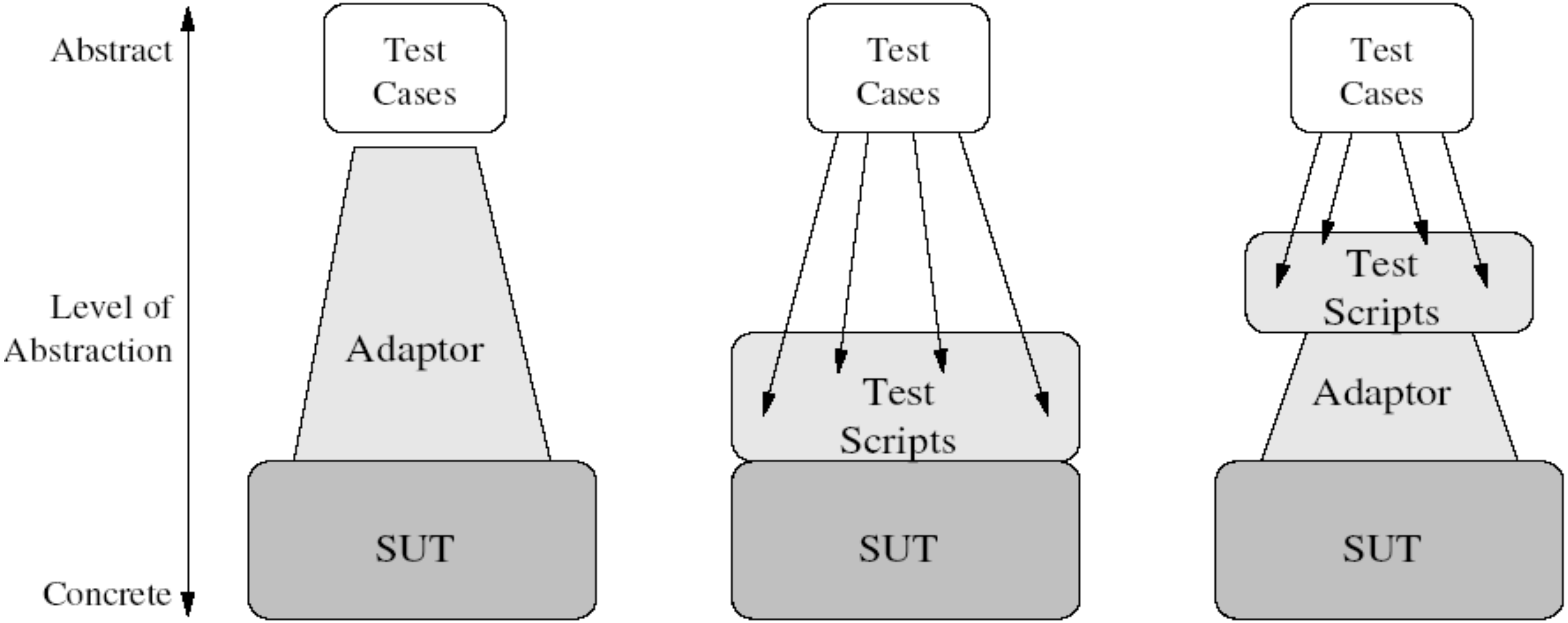
Model-Based Testing – Taxonomy [Utting et al. 11]




MBT
Model
Scope
input-only / input-output
Characteristiscs
Untimed / Timed
Determinisctic / Non-Det.
Discrete / Hybrid / Continuous
Paradigm
Pre-Post of Input Domains
Transition-Based
History-Based
Functional
Operational
Stochastic
Data-Flow
Test Generation
Test Selection Criteria
Structural Model Coverage
Data Coverage
Requierements Coverage
Test Case Specifications
Random & Stochastic
Fault-Based
Technology
Random Generation
Search-Based Algorithms
Model-Checking
Symbolic Execution
Theorem Proving
Constraint Solving
Test Execution
On/Off-line
On-line
Off-Line


tting et al. 11]
Requirements
Criteria
Model
Tests
Script
Verdict
Adaptor + Env.
Application

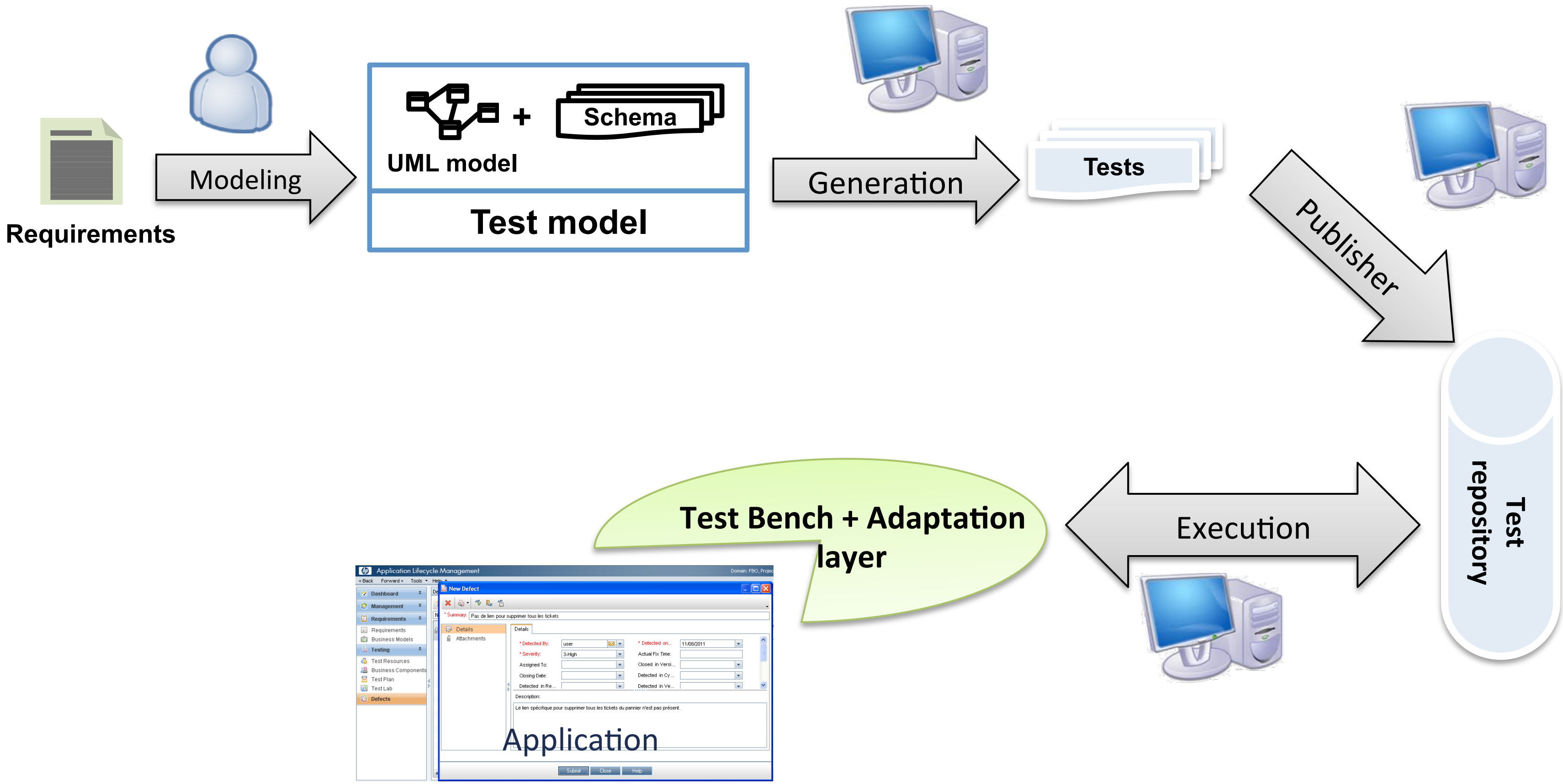
Model-Based Testing – Test adaptation

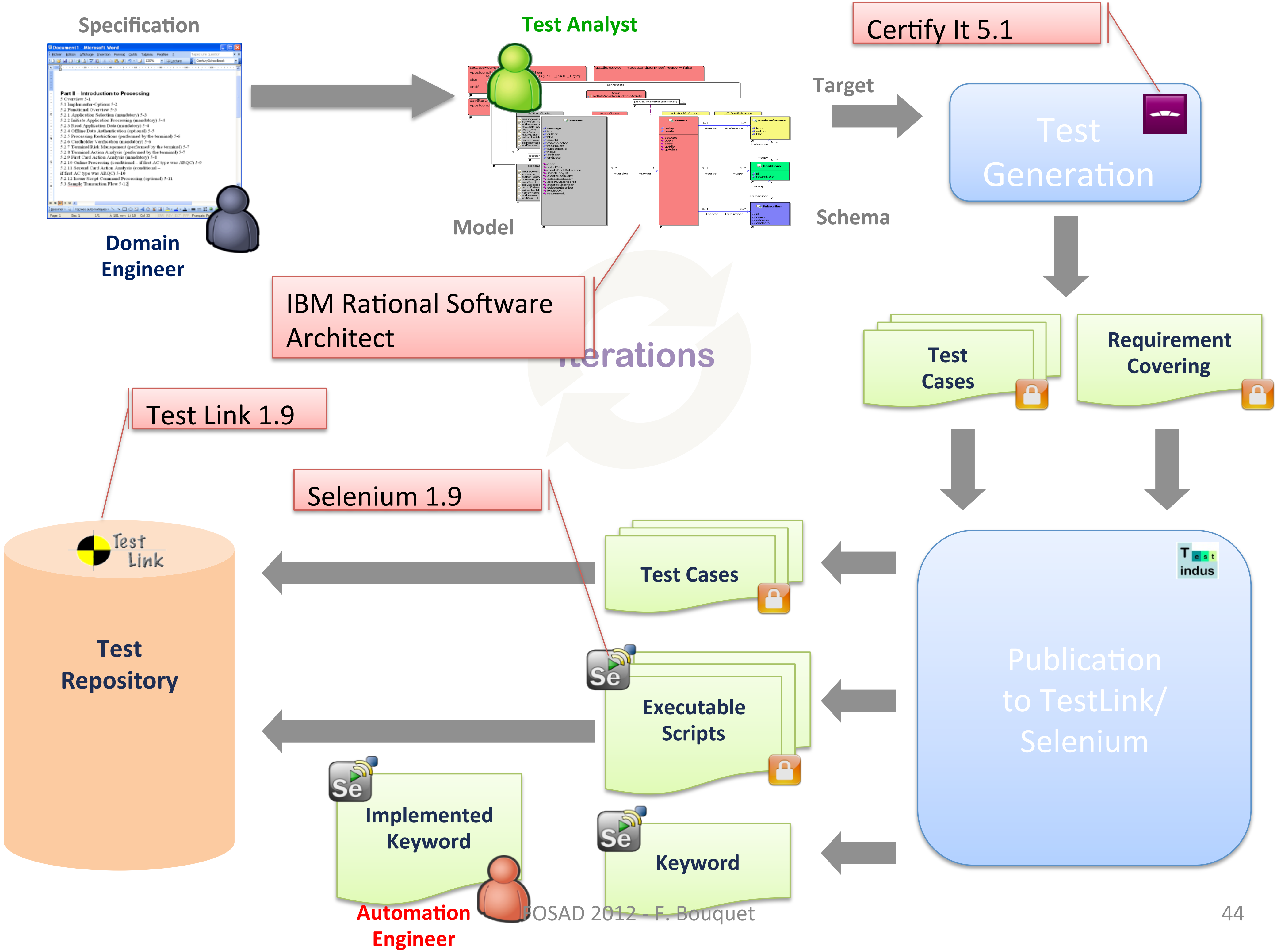


(a) Adaptation (b) Transformation (c) Mixed

Three classical approaches to test case adaptation into executable test scripts

Off-line architecture




Specification
Domain Engineer
Test Analyst
Model
Certify It 5.1
Target
Test Generation
Schema
IBM Rational Software Architect
Iterations
Test Cases
Requirement Covering
Test Link 1.9
Selenium 1.9
Test Link
Test Repository
Test Cases
Executable Scripts
Implemented Keyword
Keyword
Test indus
Publication to TestLink/ Selenium
Automation Engineer

Example – eCinema Requirements 1/8

Specification:

- “The **eCinema** is an application aiming at booking movie tickets. ”

Nb	Requierment	Name	Description
1	ACCOUNT_MNGT/LOG	Log	The system must be able to manage the login process and allow only registered user to login.
2	ACCOUNT_MNGT/ REGISTRATION	Registration	the system must be able to manage the user’s accounts.
3	BASKET_MNGT/ BUY_TICKETS	BuyTickets	The system be able to allow users to buy available tickets.
4	BASKET_MNGT/ DISPLAY_BASKET and DISPLAY_BASKET_PRICE	Display_Basket and Display_Basket_Price	The system must be able to display booked tickets and the total basket’s price for a connected user.
5	BASKET_MNGT/ REMOVE_TICKETS	Remove_Tickets	The system must be able to allow deletion of all tickets for a given user.
6	CLOSE_APPLICATION	Close_Application	The system can be shut down
7	NAVIGATION	Navigation	In the system is possible to navigate from one state to another

Example – eCinema 2/8

Username

Password

Login

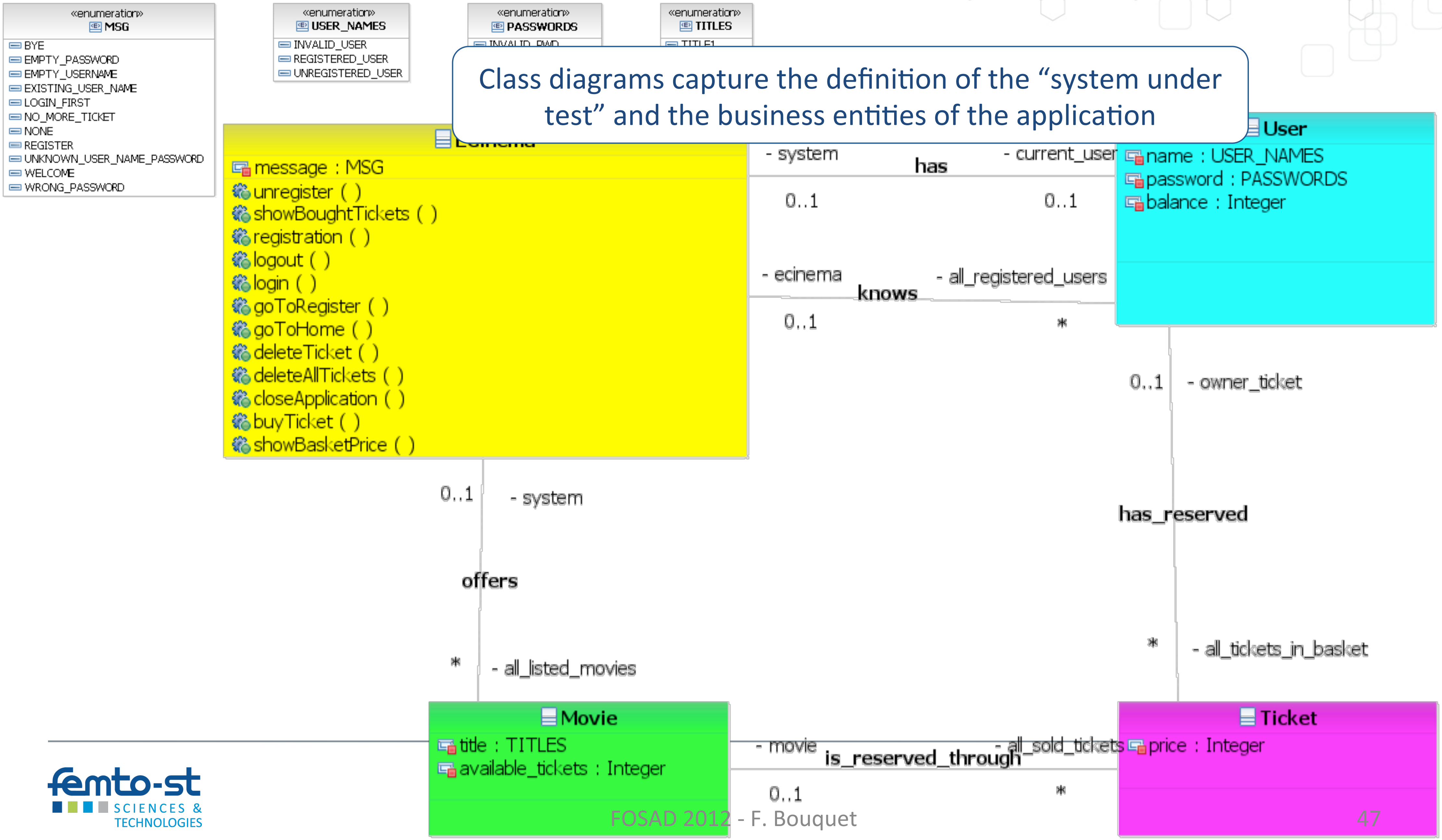
Register

Films list for 2012-09-03.

03 09 2012 go!

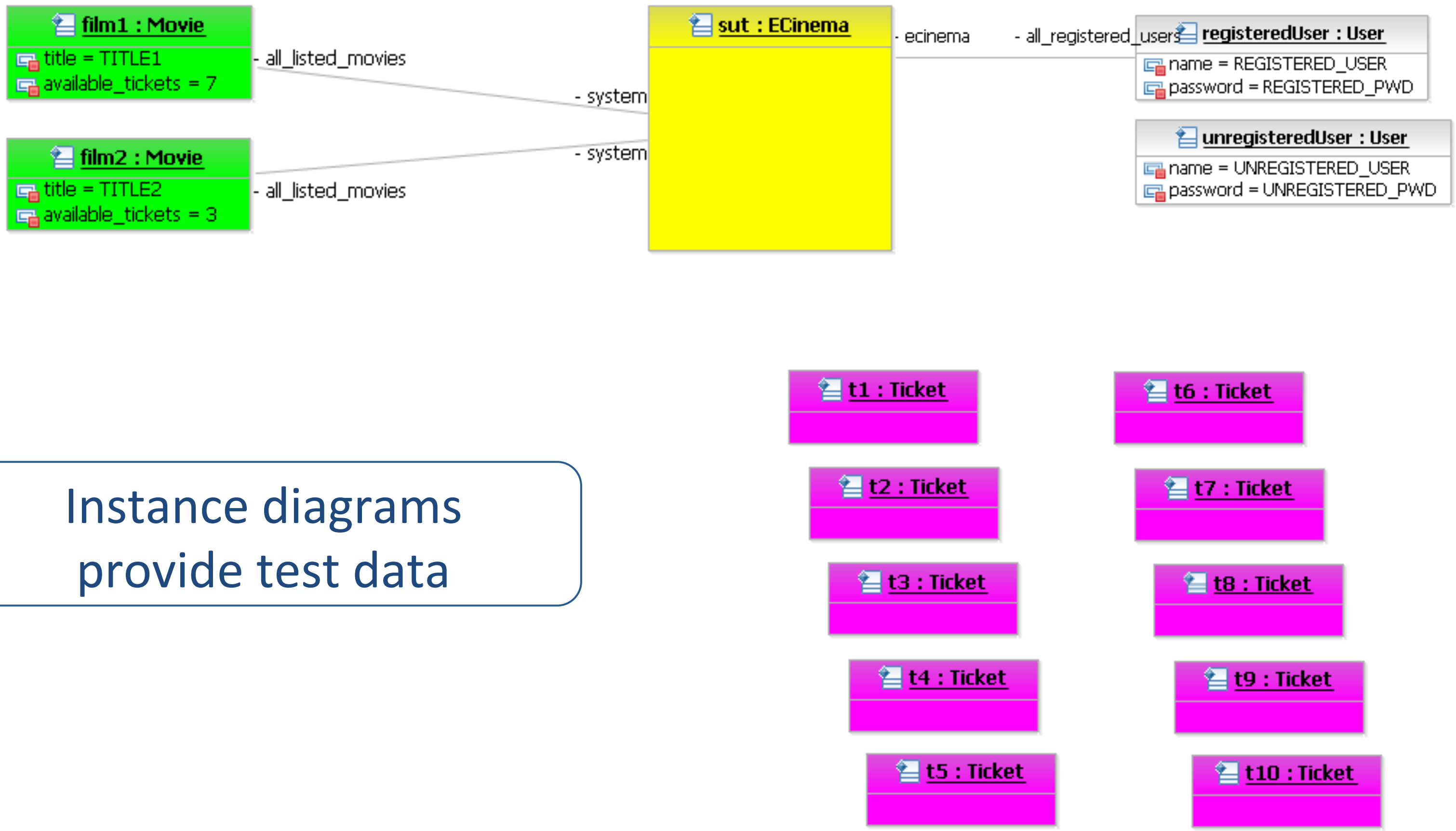
Title	Time	Duration	Tickets	
Rambo 1	20h00	120	7	Buy
Rambo 2	20h00	110	3	Buy
Rocky 7	20h00	210	5	Buy

Example – eCinema Diag. Classes 3/8



Class diagrams capture the definition of the "system under test" and the business entities of the application

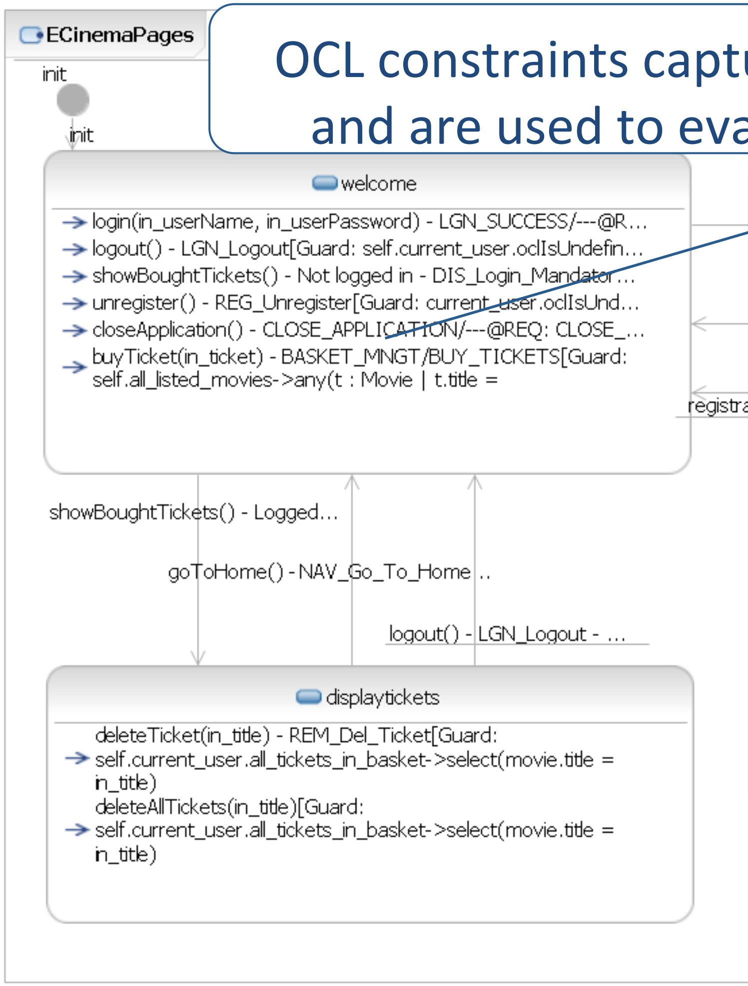
Example – eCinema Diag. Objects 4/8



Instance diagrams provide test data

Example – eCinema StateCharts – OCL code 5/8

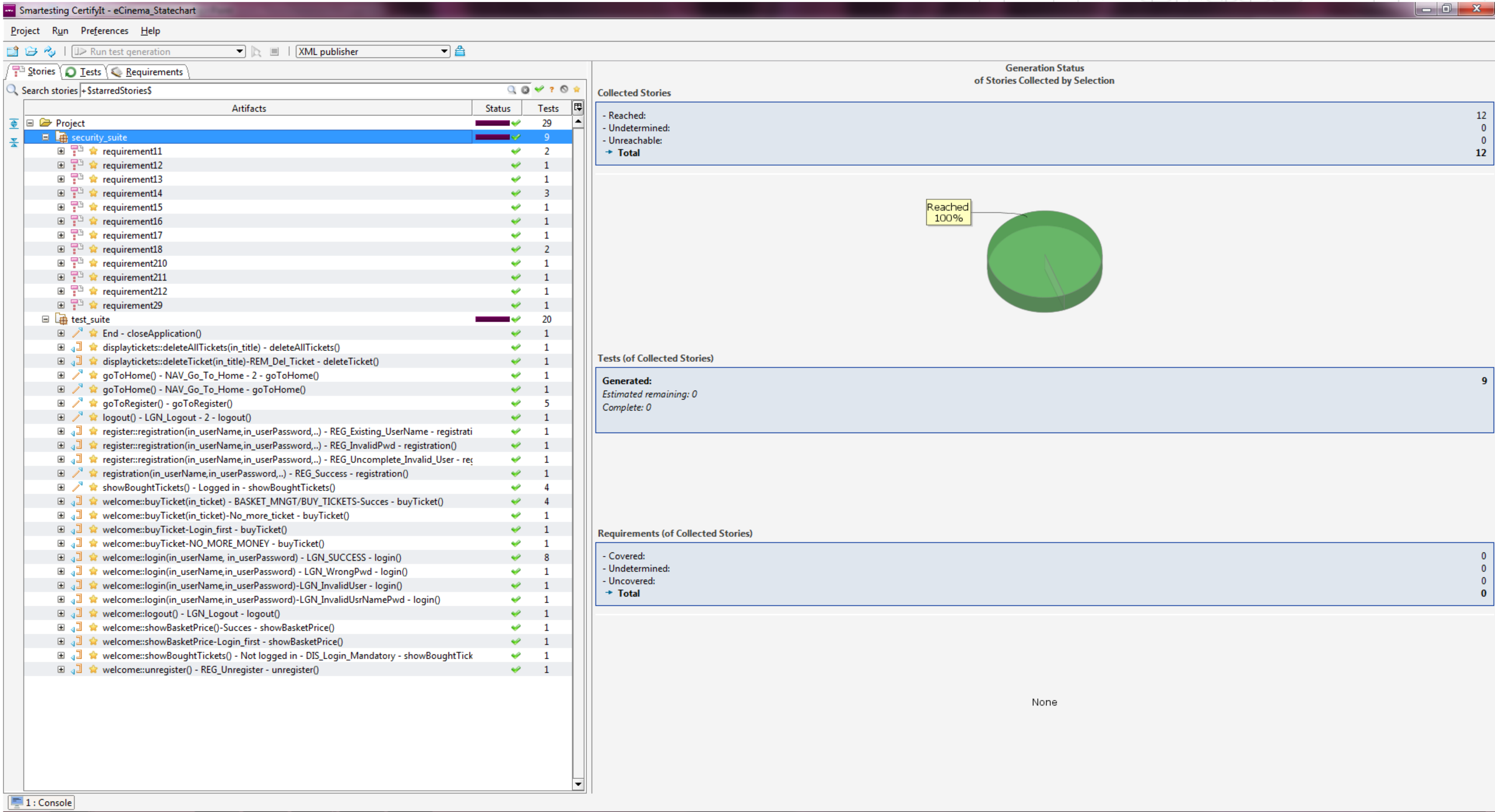
OCL constraints capture business rules, and are used to evaluate transitions



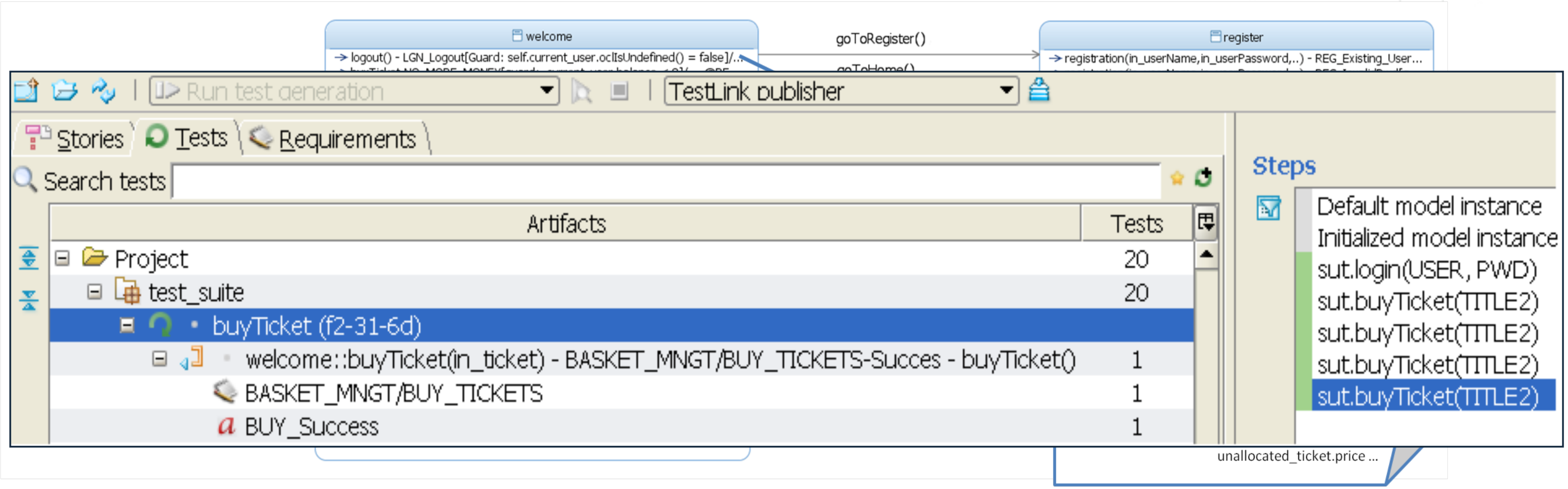

```
context login(in_userName,in_userPassword)::effect:

---@REQ: ACCOUNT_MNGT/LOG
if in_userName = USER_NAMES::INVALID_USER then
  ---@AIM: LOG_Empty_User_Name
  message= MSG::EMPTY_USERNAME
else
  if not all_registered_users->exists(name = in_userName) then
    ---@AIM: LOG_Invalid_User_Name
    message= MSG::UNKNOWN_USER_NAME_PASSWORD
  else
    let user_found:User = all_registered_users->any(name = in_userName) in
      if user_found.password = in_userPassword then
        ---@AIM: LOG_Success
        self.current_user = user_found and
        message = MSG::WELCOME
      else
        ---@AIM: LOG_Invalid_Password
        message = MSG::WRONG_PASSWORD
      endif
  endif
endif
```

State machine diagrams capture the behavior of the system

Example – eCinema Cover 6/8



Example – eCinema test cases 7/8



- We use the couple of keywords: @REQ & @AIM to represent the requirement and use it to tag the specification, the model and the generated test.
- Requirements are extracted from the specification and used in the model within post conditions of statechart transitions.
- The requirement is a target to be reached by a test. Thus, at any moment we have a link between requirement and test.

Example – eCinema Scenario 8/8

Navigator - Test Specification

Filter & Settings ...

Test Suite

Update tree after every operation

Update tree

- eCinema (20)
  - test_suite(20)
    - Evolution(0)
    - Stagnation(7)
    - Regression(13)
      - eCi-1:closeApplication (f2-bc-1c)
      - eCi-1:login (f2-fc-c8)
      - eCi-1:login (f2-e1-71)
      - eCi-1:buyTicket (f2-fc-81)
      - eCi-1:login (f2-cd-a5)
      - eCi-1:showBoughtTickets (f2-6a-c2)
      - eCi-1:showBasketPrice (f2-e8-51)
      - eCi-1:goToHome (f2-1b-fa)
      - eCi-1:logout (f2-9b-b1)
      - eCi-1:showBasketPrice (f2-32-8a)
      - eCi-1:unregister (f2-06-f5)
      - eCi-1:logout (f2-d4-dd)
      - eCi-1:goToHome (f2-0b-a0)
    - Deletion(0)

Test Case

eCi-1:unregister (f2-06-f5)

Edit | Delete | Move / Copy | Delete this version | Create a new version | Deactivate this version | Add to Test Plans | Export

Version 2

Created on 16/03/2011 16:55:20 by admin

Summary

Actual state : Re-executed

Test history
New 2011-03-16 16:45:25

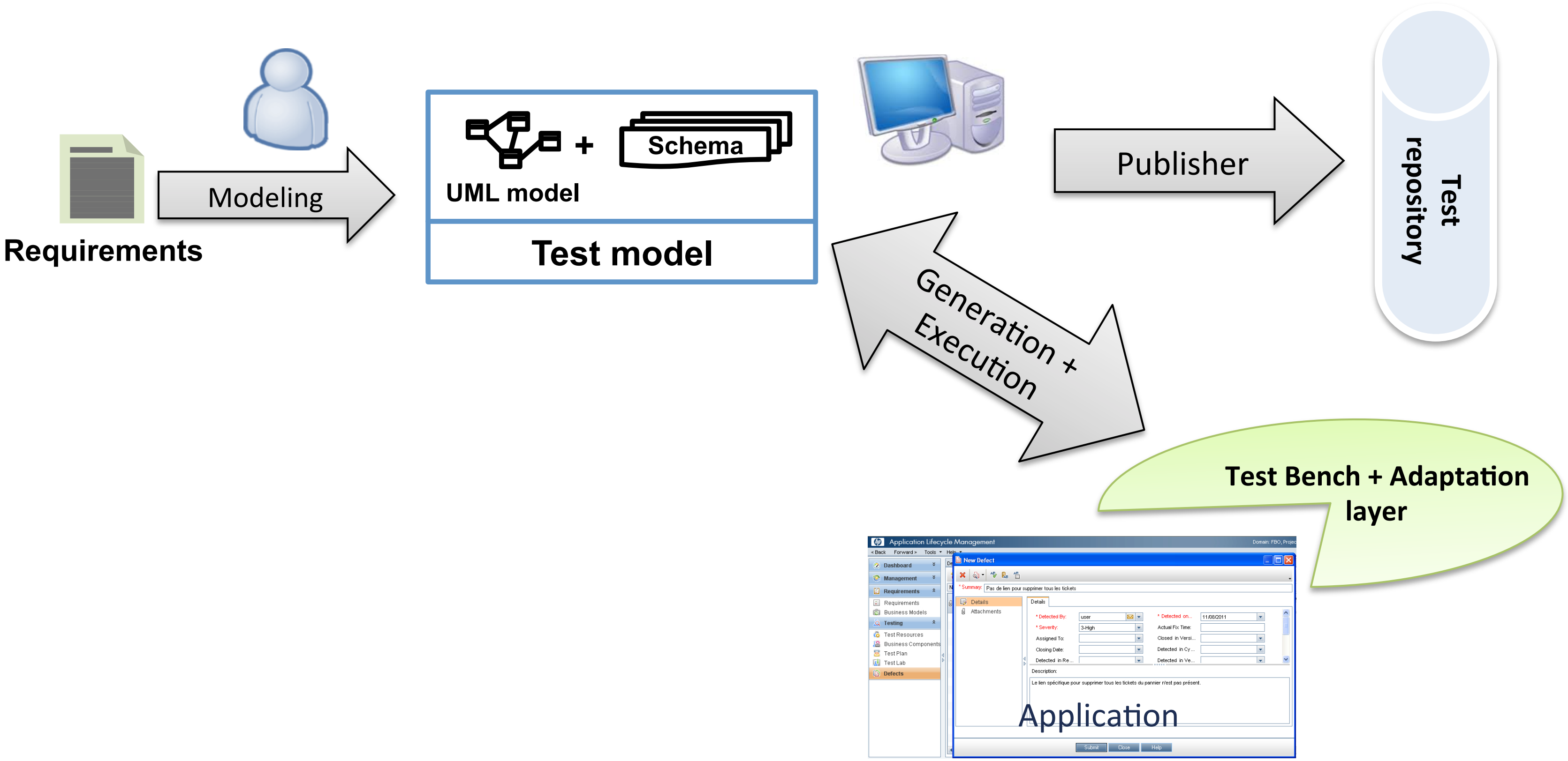
BODY

No	Description	But	Exigence	Operation
1	Fill the name field with : in_userName Fill the password field with : in_userPassword Click the `login` link With: • in_userName = `USER` (e.g. `ERIC`) • in_userPassword = `PWD` (e.g. `ETO`)	LOG_Success	ACCOUNT_MNGT/LOG	login
2	Click the `unregister` link	REG_Unregister	ACCOUNT_MNGT/REGISTRATION	unregister

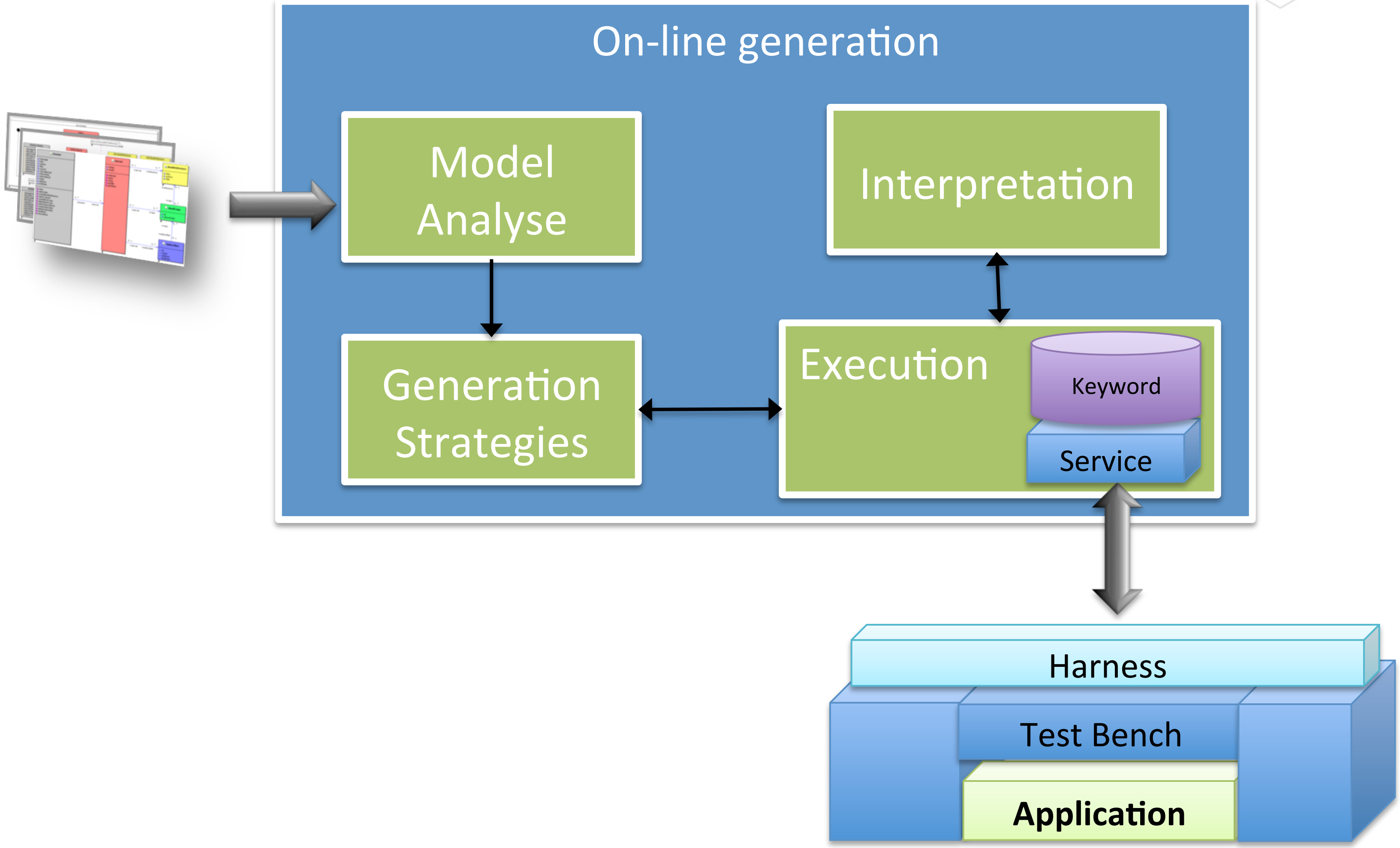
Steps Expected Results

No	Description	Operation
	Fill the name field with : in_userName	

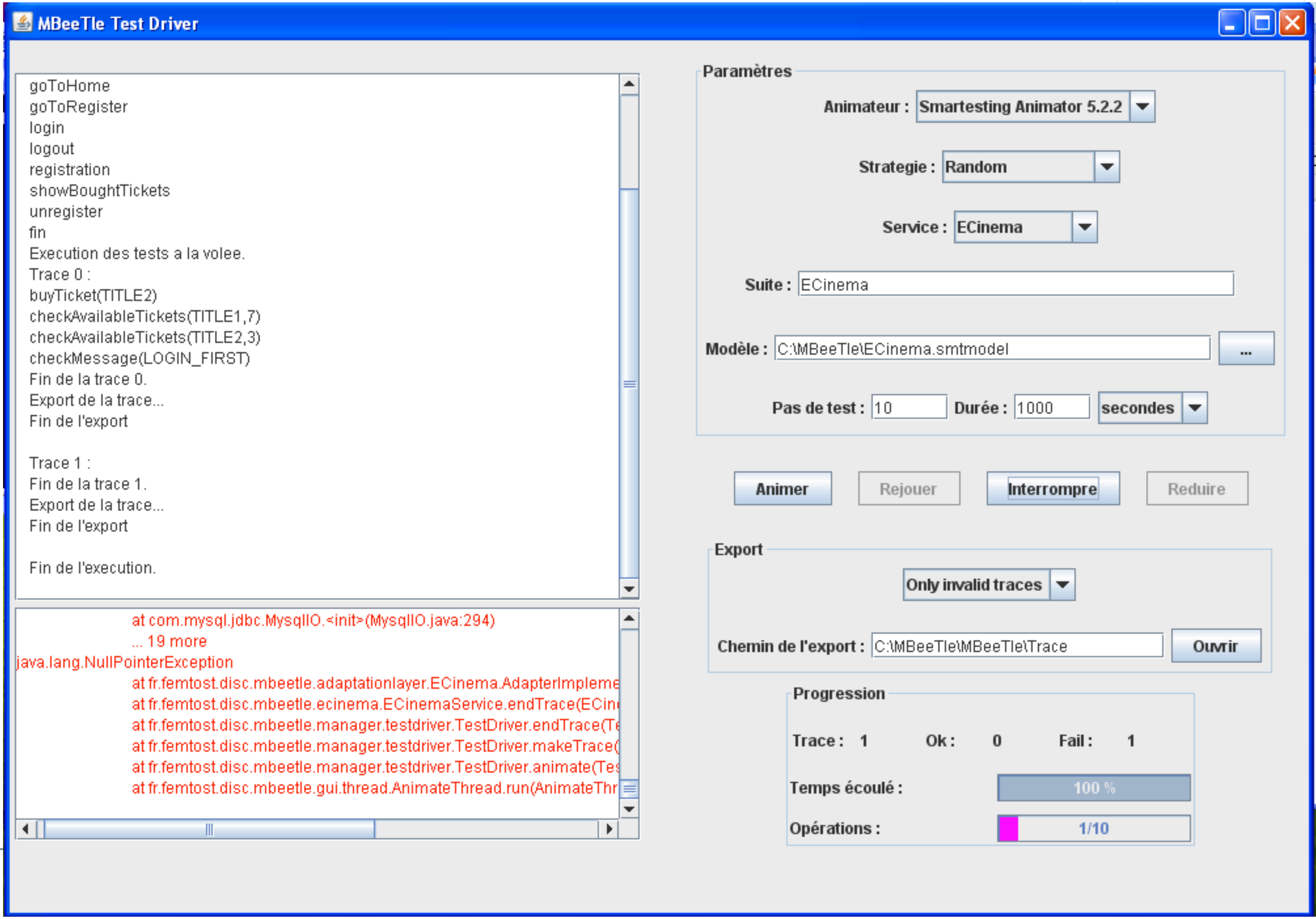
On-line architecture (ModelJUnit, GraphWalker, TGV...)



On-line MeBeeTle Architecture



On-line MeBeeTle demonstration



MBT Bilan

Main benefits of MBT:

- Easier test suite maintenance
- “Automated” test design:
  - Remove ambiguities of requirements
  - Save effort
- Better test quality (coverage):
  - no human forgetting
  - Computer can find more combinations for complex systems than human brain
- Online MBT provides also
  - Testing nondeterministic systems
  - Infinite test suite

Tools:

- MaTeLo
- Qtronic – Designer (Conformiq)
- Reactis
- Spec Explorer (Microsoft)
- Certify It (Smartesting)
- STG - TGV (IRISA)
- HydraCore – BZTestingTools (INRIA)
- GraphWalker (Tigris)
- ModelJUnit (CSZ)

More than 36 tools…

To be continued Wednesday…

I. Introduction of test
II. Functional Testing
III. Model-Based Testing
IV. Model-Based Testing and Security
V. Evolution of system

Thanks for your attention

Do you prefer to use a system
formally proved or tested ?

i.r. the artifacts referred …

Model-Based Testing for Functional and Security Test

- Part 2 -

Fabrice BOUQUET

Summer School FOSAD
3 - 7th July 2012

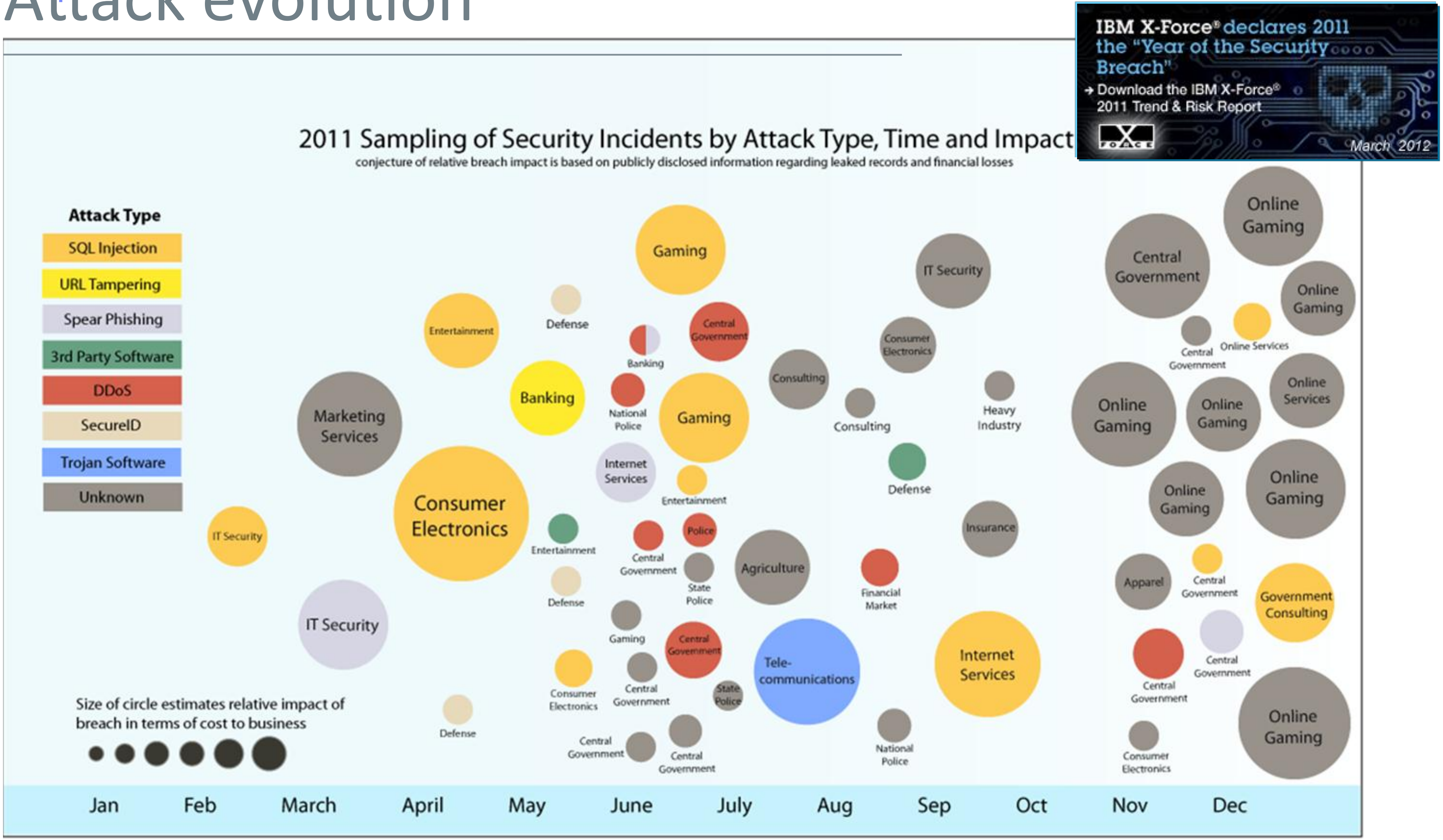
OUTLINE

I. Introduction of test
II. Functional Testing
III. Model-Based Testing
IV. Model-Based Testing and Security
   i. Model for Security Testing
   ii. Security test objectives
   iii. (Security) Test purposes
V. Evolution of system

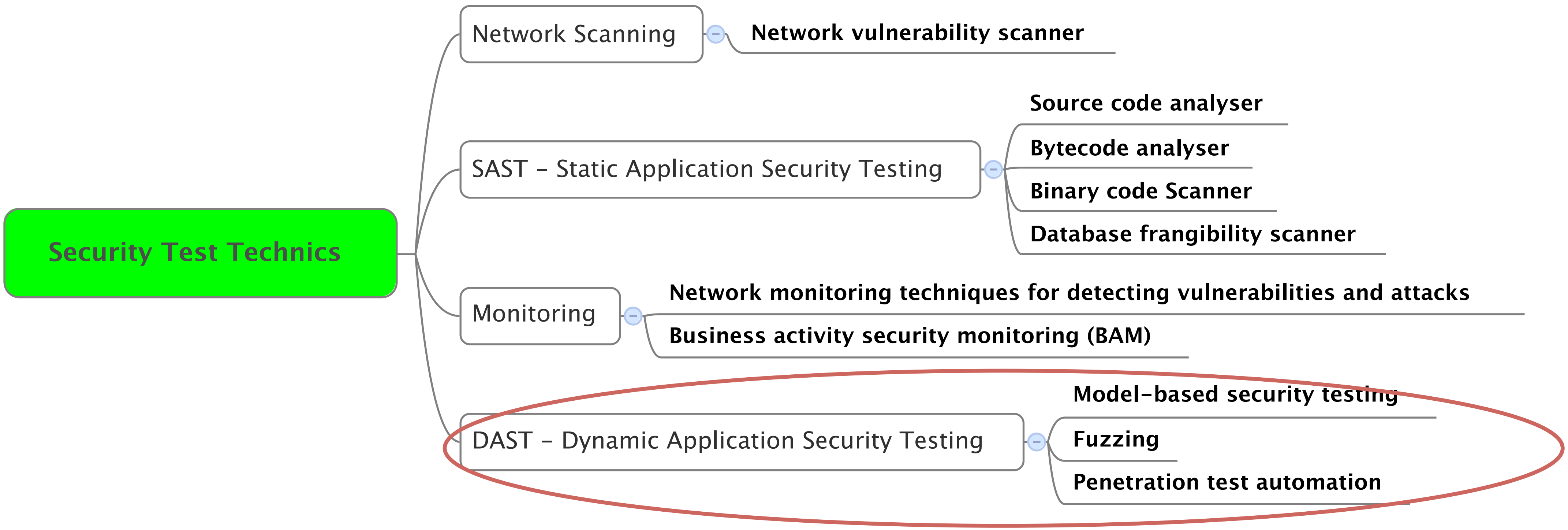
Security Testing Overview

- **Security Test Technics**
  - Network Scanning
    - **Network vulnerability scanner**
  - SAST – Static Application Security Testing
    - **Source code analyser**
    - **Bytecode analyser**
    - **Binary code Scanner**
    - **Database frangibility scanner**
  - Monitoring
    - **Network monitoring techniques for detecting vulnerabilities and attacks**
    - **Business activity security monitoring (BAM)**
  - DAST – Dynamic Application Security Testing
    - **Model-based security testing**
    - **Fuzzing**
    - **Penetration test automation**

Attack evolution



Security Testing Overview



Classification for Model-Based Security Testing [Felder et Al 11]

Individual Knowledge:

- individual knowledge determines the design of security tests
- Selection of function and data

(Adapted) Risk-Based Testing:

- Using threat models
- Prioritization of Test

Scenario-Based MBT:

- Complete Model (of MBT) with scenario

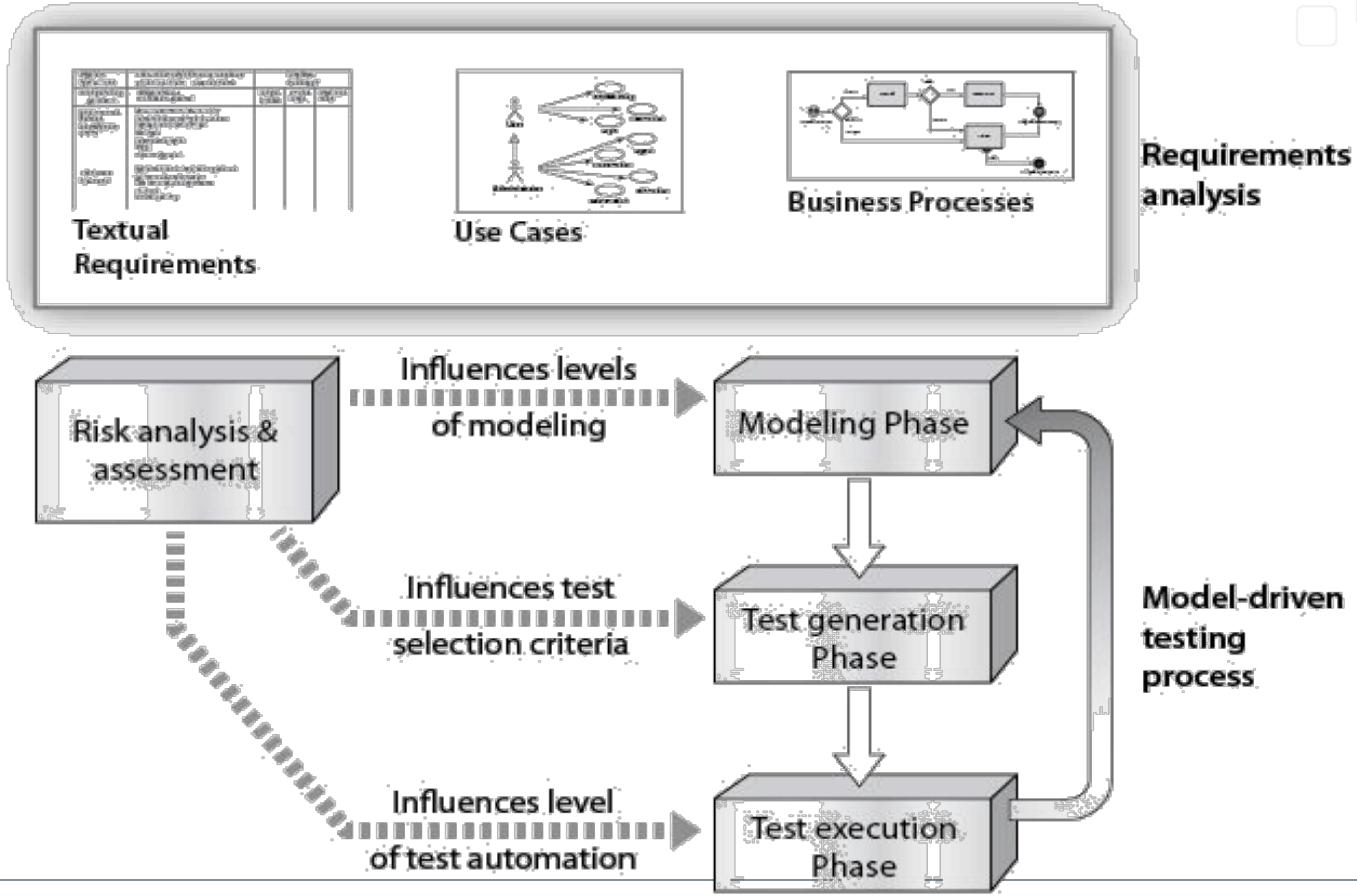
Risk Enhanced Scenario-Based MBT :

- Complete Model
    - with scenario
    - Risk information

Adapted MBT:

- Dedicated model for security (many works in access control policies)

MBT and Risk-based testing



How to define Security Test Objective

From Dedicated Security Model:

- UMLSech [Jurgens 2008] [Fourneret et al 2011]
- SecureUML [Lodderstedt et Al. 2002]
- Protocole [Probert and Guo 91], [Darmaillacq et a 2006l], [Dadeau et al 11]
- Threat model / Tree / Automata [XU et al. 2005], [Wang et al. 2007] + Workshops
- Fuzz testing or Fuzzing [Sutton et Al. 2007]

From Properties or Schema languages:

- General purpose [Jeron et al. 92], … [Karbrera et AL 2011]…
- Dedicated:
  - *Many by* Advanced Open Standards of the Information Society (OASIS) :
    - *eXtensible Access Control Markup Language* (XACML)
    - *Security Assertion Markup Language* (SML)
  - Open Web Application Security Project (OWASP)
  - OrBac [Abou El Kalam et al. 03]
  - Many kinds of automata, Regular expression, Logic formula…

Mutation-Based Test Generation from Security Protocols [Dadeau et al. 11]

Security protocols:

- are an important issue in system security designs (integrity, authenticity, secrecy, etc.)
- represent exchanges of messages (possibly encrypted, hashed, signed, etc.) between agents
- aim at establishing a trustful communication link between agents (e.g. authentication, sensitive information exchange)

The AVISPA project and tool-set :

European project aiming at the verification of security protocols (2003–2006).

- Definition of a common language High-Level Protocol Specification Language (HLPSL)
- Tool-set for verifying the protocols and finding attack traces when declared *unsafe*
- Set of modeled protocols from real-world applications
- It relies on 4 back-ends: CL-AtSe, TA4SP, SATMC, OFMC

For more details: http://avispa-project.org

Mutation-Based Test Generation from Security Protocols [Dadeau et al. 11]

Validation ≠ Verification:

- Verifying the safety of a protocol modeled in HLPSL does not guarantee its
- correct implementation

  Various examples of "safe" protocols wrongly implemented

  ⇒ OpenSSL package of Debian (bug in random generator generating SSL and SSH keys)

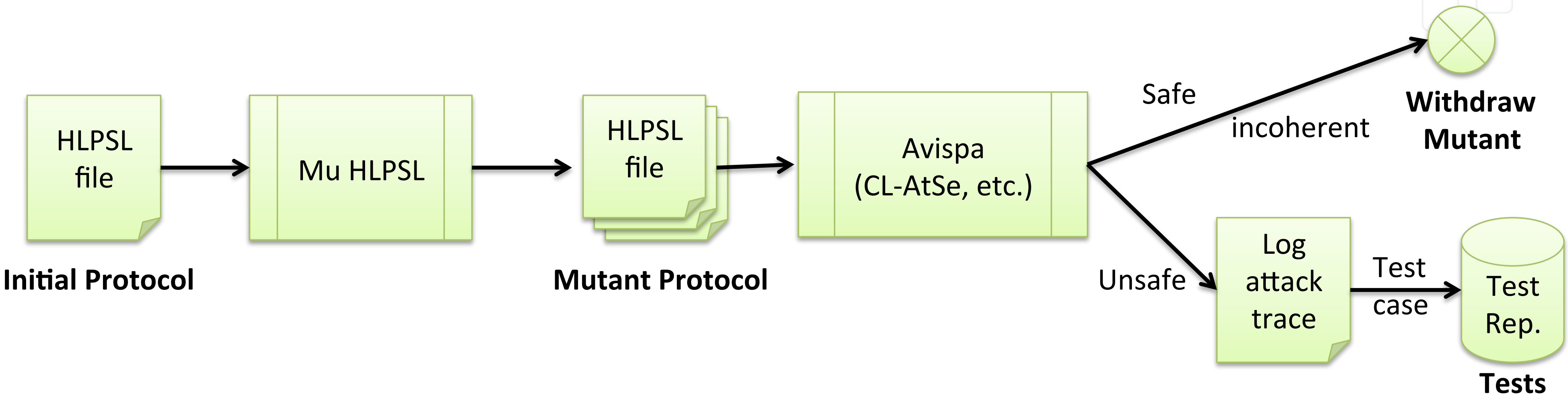
  ⇒ Omission of a part of the specification of the SSO protocol of Google Applications

Mutation Testing:

Mutation testing consists in introducing a single fault in a correct program/model:

- an *error* in the coding, is expressed by a *fault* that, when executed, reveals a *defect*
- can be used to evaluate the quality of a test suite (program mutation) but also used to generate tests, that aim at revealing the defect (model mutation)
- definition of a relevant *fault model* to produce mutants

Mutation-Based Test Generation from Security Protocols [Dadeau et al. 11]

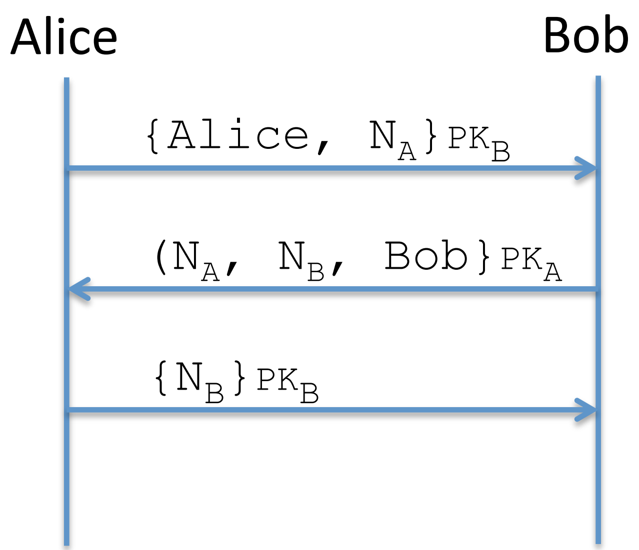


Process:

- Definition and application of a fault model for security protocols in HLPSL
- Verify protocols using the AVISPA tool-set
- They use *CL-AtSe,* a symbolic model-checker that is able to compute attack traces and supports XOR and EXP operators.
- Unsafe mutants yield to the building of an attack trace that can be used as a test case

HLPSL with an example

Needham-Schröder Public Key protocol: Alice <-> Bob



$Alice \rightarrow Bob \quad : \{Alice,\ N_A\}PK_B$

$Bob \rightarrow Alice \quad : (N_A,\ N_B,\ Bob\}PK_A$

$Alice \rightarrow Bob \quad : \{N_B\}PK_B$

HLPSL with an example

Needham-Schröder Public Key protocol: Alice <-> Bob

```
role alice (A, B: agent, Ka, Kb: public_key, SND, RCV: channel (dy))

played_by A def=
    local State: nat, Na, Nb: nat

    init State := 0

    transition
        0. State = 0 /\ RCV(start) =|>
           State':= 2 /\ Na' := new() /\ SND({Na'.A}_Kb)

        2. State = 2 /\ RCV({Na.Nb'.B}_Ka) =|>
           State':= 4 /\ SND({Nb'}_Kb)
end role
```

HLPSL with an example

Needham-Schröder Public Key protocol: Alice <-> Bob

```
role bob(A, B: agent, Ka, Kb: public_key, SND, RCV: channel (dy))

played_by B def=
    local State : nat, Na, Nb: nat

    init State := 1

    transition
        1. State = 1 /\ RCV({Na'.A}_Kb) =|>
           State':= 3 /\ Nb' := new() /\ SND({Na'.Nb'}_Ka)

        3. State = 3 /\ RCV({Nb}_Kb) =|>
           State':= 5
end role
```

HLPSL with an example

Needham-Schröder Public Key protocol: Alice <-> Bob

```
role session(A, B: agent, Ka, Kb: public_key) def=
    local SA, RA, SB, RB: channel (dy)

    composition
        alice (A,B,Ka,Kb,SA,RA) /\ bob (A,B,Ka,Kb,SB,RB)
end role

role environment() def=
    const a, b : agent, ka, kb, ki : public_key

    intruder_knowledge = {a, b, ka, kb, ki, inv(ki)}

    composition
        session(a,b,ka,kb) /\ session(a,i,ka,ki) /\ session(i,b,ki,kb)
end role
```

Mutation operators

Motivations:

- exploiting the expressiveness of HLPSL
- inspired from real-world errors that may introduce flaws in security protocols

6 Operators:

- XOR and EXP
- Homomorphism
- Public Key
- Hash Functions
- Substitution
- Permutation

XOR and EXP Operator

Principe of XOR:

- exclusive-OR encryption: $\{M\}_K \Leftrightarrow M \oplus K$
- used in the Wired Equivalent Privacy (WEP) Protocol
- subject to man-in-the-middle attacks due to the XOR operator properties of commutativity, idempotence, and associativity

Man-in-the-middle on the Shamir Three-Pass protocol:

Alice → Bob : M xor K_A
Bob → Alice : (M xor K_A) xor K_B
Alice → Bob : M xor K_B

A simple passive observation makes it possible for an intruder to know (and combine) :

$$M \text{ xor } K_A \text{ xor } (M \text{ xor } K_A) \text{ xor } K_B \text{ xor } M \text{ xor } K_B \Leftrightarrow M$$

In HLPSL code:

```
1. State = 1 /\ RCV({Na'.A}_Kb) =|>
     State':= 3 /\ Nb' := new() /\ SND({Na'.Nb'}_Ka)
3. State = 3 /\ RCV({Nb}_Kb) =|>
     State':= 5
```

XOR and EXP Operator

Principe of XOR:

- exclusive-OR encryption: $\{M\}_K \Leftrightarrow M \oplus K$
- used in the Wired Equivalent Privacy (WEP) Protocol
- subject to man-in-the-middle attacks due to the XOR operator properties of commutativity, idempotence, and associativity

Man-in-the-middle on the Shamir Three-Pass protocol:

Alice → Bob : M xor K_A
Bob → Alice : (M xor K_A) xor K_B
Alice → Bob : M xor K_B

A simple passive observation makes it possible for an intruder to know (and combine) :

$$M \text{ xor } K_A \text{ xor } (M \text{ xor } K_A) \text{ xor } K_B \text{ xor } M \text{ xor } K_B \Leftrightarrow M$$

In HLPSL code:

```
1. State = 1 /\ RCV({Na'.A}_Kb) =|>
    State':= 3 /\ Nb' := new() /\ SND({Na'.Nb'}_Ka)
3. State = 3 /\ RCV({Nb}_Kb) =|>
    State':= 5
```

XOR and EXP Operator

Principe of XOR:

- exclusive-OR encryption: $\{M\}_K \Leftrightarrow M \oplus K$
- used in the Wired Equivalent Privacy (WEP) Protocol
- subject to man-in-the-middle attacks due to the XOR operator properties of commutativity, idempotence, and associativity

Man-in-the-middle on the Shamir Three-Pass protocol:

Alice → Bob : M xor K_A
Bob → Alice : $(M$ xor $K_A)$ xor K_B
Alice → Bob : M xor K_B

A simple passive observation makes it possible for an intruder to know (and combine) :

$$M \text{ xor } K_A \text{ xor } (M \text{ xor } K_A) \text{ xor } K_B \text{ xor } M \text{ xor } K_B \Leftrightarrow M$$

In HLPSL code:

```
1. State = 1 /\ RCV(xor({Na'.A},Kb)) =|>
     State':= 3 /\ Nb' := new() /\ SND(xor({Na'.Nb'},Ka))
3. State = 3 /\ RCV(xor({Nb},Kb)) =|>
     State':= 5
```

Homomorphism

Principe of homomorphism:

- Encryption function that satisfies property: $\{X.Y\}_K \Leftrightarrow \{X\}_k . \{Y\}_k$
- Block-by-block encryption
- Used in the Electronic Code Book

NSPK attack due to homomorphism:

Consider the corrected version of NSPK

Alice → Bob : $\{Alice, N_A\}PK_B$
Bob → Alice : $(N_A, N_B, Bob\}PK_A$
Alice → Bob : $\{N_B\}PK_B$

It can become unsafe with homomorphism in the encryption scheme:

Alice → Bob : $\{Alice, N_A\}PK_B$
Bob → Charlie : $\{Alice, N_A\}PK_C$
Charlie → Bob : $(N_A, N_C, Charlie\}PK_A$ **(interception)**
Bob → Alice : $(N_A, N_C, Bob\}PK_A$
Alice → Bob : $\{N_C\}PK_B$
Bob → Charlie : $\{N_C\}PK_C$

Homomorphism

Principle of homomorphism:

- Encryption function that satisfies property: $\{X .Y\}_K \Leftrightarrow \{X\}_k . \{Y\}_k$
- Block-by-block encryption
- Used in the Electronic Code Book

NSPK attack due to homomorphism:

Consider the corrected version of NSPK

Alice → Bob : $\{Alice, N_A\}PK_B$
Bob → Alice : $(N_A, N_B, Bob\}PK_A$
Alice → Bob : $\{N_B\}PK_B$

In HLPSL code:

```
1. State = 1 /\ RCV({Na'.A}_Kb) =|>
    State':= 3 /\ Nb' := new() /\ SND({Na'.Nb'}_Ka)
3. State = 3 /\ RCV({Nb}_Kb) =|>
    State':= 5
```

Homomorphism

Principe of homomorphism:

- Encryption function that satisfies property: $\{X .Y\}_K \Leftrightarrow \{X\}_k . \{Y\}_k$
- Block-by-block encryption
- Used in the Electronic Code Book

NSPK attack due to homomorphism:

Consider the corrected version of NSPK

Alice → Bob : $\{Alice, N_A\}PK_B$
Bob → Alice : $(N_A, N_B, Bob\}PK_A$
Alice → Bob : $\{N_B\}PK_B$

In HLPSL code:

```
1. State = 1 /\ RCV({Na'.A}_Kb) =|>
     State':= 3 /\ Nb' := new() /\ SND({Na'.Nb'}_Ka)
3. State = 3 /\ RCV({Nb}_Kb) =|>
     State':= 5
```

Homomorphism

Principe of homomorphism:

- Encryption function that satisfies property: $\{X.Y\}_K \Leftrightarrow \{X\}_k . \{Y\}_k$
- Block-by-block encryption
- Used in the Electronic Code Book

NSPK attack due to homomorphism:

Consider the corrected version of NSPK

Alice → Bob : $\{Alice, N_A\}PK_B$
Bob → Alice : $(N_A, N_B, Bob\}PK_A$
Alice → Bob : $\{N_B\}PK_B$

In HLPSL code:

```
1. State = 1 /\ RCV({Na'}_Kb.{A}_Kb) =|>
    State':= 3 /\ Nb' := new() /\ SND({Na'}_Kb.{Nb'}_Ka)
3. State = 3 /\ RCV({Nb}_Kb) =|>
    State':= 5
```

Public Key

Principe of public key:

- protocol participants use the same public key
- may happen after a bad manipulation of **.ssh** files
- mutation introduced in the **role** environment

In HLPSL code:

```
role environment() def=
    const a, b : agent, ka, kb, ki : public_key

    intruder_knowledge = {a, b, ka, kb, ki, inv(ki)}

    composition
        session(a,b,ka,kb) /\ session(a,i,ka,ki) /\ session(i,b,ki,kb)
end role
```

Public Key

Principe of public key:

- protocol participants use the same public key
- may happen after a bad manipulation of **.ssh** files
- mutation introduced in the **role** environment

In HLPSL code:

```
role environment() def=
    const a, b : agent, ka, kb, ki : public_key

    intruder_knowledge = {a, b, ka, kb, ki, inv(ki)}

    composition
        session(a,b,ka,kb) /\ session(a,i,ka,ki) /\ session(i,b,ki,kb)
end role
```

Public Key

Principe of public key:

- protocol participants use the same public key
- may happen after a bad manipulation of **.ssh** files
- mutation introduced in the **role** environment

In HLPSL code:

```
role environment() def=
    const a, b : agent, ka, kb, ki : public_key

    intruder_knowledge = {a, b, ka, kb, ki, inv(ki)}

    composition
        session(a,b,ka,ka) /\ session(a,i,ka,ki) /\ session(i,b,ki,ka)
end role
```

Substitution

Principe of Substitution:

- replace computational useless pieces of messages by arbitrary values
- catches absences of semantic verification of messages content

Substitution on NSPK:

A flaw may arise if Alice does not check X, but only the message is well formed.

Alice → Bob : $\{Alice, N_A\}PK_B$
Bob → Alice : $(N_A, N_B, \mathbf{X}\}PK_A$
Alice → Bob : $\{N_B\}PK_B$

In HLPSL code:

```
role alice (A, B: agent, Ka, Kb: public_key, SND, RCV: channel (dy))

played_by A def=
     local State: nat, Na, Nb: nat

     init State := 0

     transition
          0. State = 0 /\ RCV(start) =|>
             State':= 2 /\ Na' := new() /\ SND({Na'.A}_Kb)

          2. State = 2 /\ RCV({Na.Nb'.B}_Ka) =|>
             State':= 4 /\ SND({Nb'}_Kb)

end role
```

Substitution

Principe of Substitution:

- replace computational useless pieces of messages by arbitrary values
- catches absences of semantic verification of messages content

Substitution on NSPK:

A flaw may arise if Alice does not check X, but only the message is well formed.

Alice → Bob : $\{Alice, N_A\}PK_B$
Bob → Alice : $(N_A, N_B, \mathbf{X}\}PK_A$
Alice → Bob : $\{N_B\}PK_B$

In HLPSL code:

```
role alice (A, B: agent, Ka, Kb: public_key, SND, RCV: channel (dy))

played_by A def=
    local State: nat, Na, Nb: nat

    init State := 0

    transition
        0. State = 0 /\ RCV(start) =|>
           State':= 2 /\ Na' := new() /\ SND({Na'.A}_Kb)

        2. State = 2 /\ RCV({Na.Nb'.B}_Ka) =|>
           State':= 4 /\ SND({Nb'}_Kb)
end role
```

Substitution

Principe of Substitution:

- replace computational useless pieces of messages by arbitrary values
- catches absences of semantic verification of messages content

Substitution on NSPK:

A flaw may arise if Alice does not check X, but only the message is well formed.

Alice → Bob : $\{Alice, N_A\}PK_B$
Bob → Alice : $(N_A, N_B, \mathbf{X}\}PK_A$
Alice → Bob : $\{N_B\}PK_B$

In HLPSL code:

```
role alice (A, B: agent, Ka, Kb: public_key, SND, RCV: channel (dy))

played_by A def=
     local State: nat, Na, Nb: nat, X: agent

     init State := 0

     transition
          0. State = 0 /\ RCV(start) =|>
             State':= 2 /\ Na' := new() /\ SND({Na'.A}_Kb)

          2. State = 2 /\ RCV({Na.Nb'.X}_Ka) =|>
             State':= 4 /\ SND({Nb'}_Kb)
end role
```

Hash Functions

Principe of hash functions:

- one-way compression functions
- frequently used for numerical signatures (SSL protocol) but also used for numerical registration
- in security protocols: guarantee of security

Remove the hash function

In HLPSL code:

```
role chap(B, A: agent, Kab: symmetric_key, H: hash_func, SND, RCV: channel (dy))
played_by B def=
    local State: nat, Na, Nb: nat
    init State := 0
    transition
        0. State = 0 /\ RCV(A') =|>
           State':= 1 /\ Nb' := new() /\ SND(Nb')
        2. State = 1 /\ RCV(Na'.H(Kab.Na'.Nb.A)) =|>
           State':= 2 /\ SND(H(Kab.Na'))
end role
```

Hash Functions

Principe of hash functions:

- one-way compression functions
- frequently used for numerical signatures (SSL protocol) but also used for numerical registration
- in security protocols: guarantee of security

Remove the hash function

In HLPSL code:

```
role chap(B, A: agent, Kab: symmetric_key, H: hash_func, SND, RCV: channel (dy))

played_by B def=
    local State: nat, Na, Nb: nat

    init State := 0

    transition
        0. State = 0 /\ RCV(A') =|>
           State':= 1 /\ Nb' := new() /\ SND(Nb')

        2. State = 1 /\ RCV(Na'.H(Kab.Na'.Nb.A)) =|>
           State':= 2 /\ SND(H(Kab.Na'))
end role
```

Hash Functions

Principle of hash functions:

- one-way compression functions
- frequently used for numerical signatures (SSL protocol) but also used for numerical registration
- in security protocols: guarantee of security

Remove the hash function

In HLPSL code:

```
role chap(B, A: agent, Kab: symmetric_key, H: hash_func SND, RCV: channel (dy))

played_by B def=
     local State: nat, Na, Nb: nat

     init State := 0

     transition
          0. State = 0 /\ RCV(A') =|>
               State':= 1 /\ Nb' := new() /\ SND(Nb')

          2. State = 1 /\ RCV(Na'.Kab.Na'.Nb.A) =|>
               State':= 2 /\ SND(Kab.Na')
end role
```

Permutation

Principe of permutation:

- Applicable to messages composed of several blocks
- Exchange/commutation of blocks inside a message
- Very common non-conformance between implementation and specification, that may lead to security flaws

Operation:

- applied on pairs of messages (sending/reception)
- combinatorial explosion: for messages of *n* blocks, *n!* permutations
  ⇒ for the experiments: move last block to first place

In HLPSL code:

```
role alice (A, B: agent, Ka, Kb: public_key, SND, RCV: channel (dy))
played_by A def=
     local State: nat, Na, Nb: nat
     init State := 0
     transition
          0. State = 0 /\ RCV(start) =|>
               State':= 2 /\ Na' := new() /\ SND({Na'.A}_Kb)

          2. State = 2 /\ RCV({Na.Nb'.B}_Ka) =|>
               State':= 4 /\ SND({Nb'}_Kb)
end role
```

Permutation

Principe of permutation:

- Applicable to messages composed of several blocks
- Exchange/commutation of blocks inside a message
- Very common non-conformance between implementation and specification, that may lead to security flaws

Operation:

- applied on pairs of messages (sending/reception)
- combinatorial explosion: for messages of *n* blocks, *n!* permutations

  ⇒ for the experiments: move last block to first place

In HLPSL code:

```
role alice (A, B: agent, Ka, Kb: public_key, SND, RCV: channel (dy))
played_by A def=
     local State: nat, Na, Nb: nat
     init State := 0
     transition
          0. State = 0 /\ RCV(start) =|>
             State’:= 2 /\ Na’  := new() /\ SND({Na’.A}_Kb)

          2. State = 2 /\ RCV({Na.Nb’.B}_Ka) =|>
             State’:= 4 /\ SND({Nb’}_Kb)
end role
```

Permutation

Principe of permutation:

- Applicable to messages composed of several blocks
- Exchange/commutation of blocks inside a message
- Very common non-conformance between implementation and specification, that may lead to security flaws

Operation:

- applied on pairs of messages (sending/reception)
- combinatorial explosion: for messages of *n* blocks, *n!* permutations

  ⇒ for the experiments: move last block to first place

In HLPSL code:

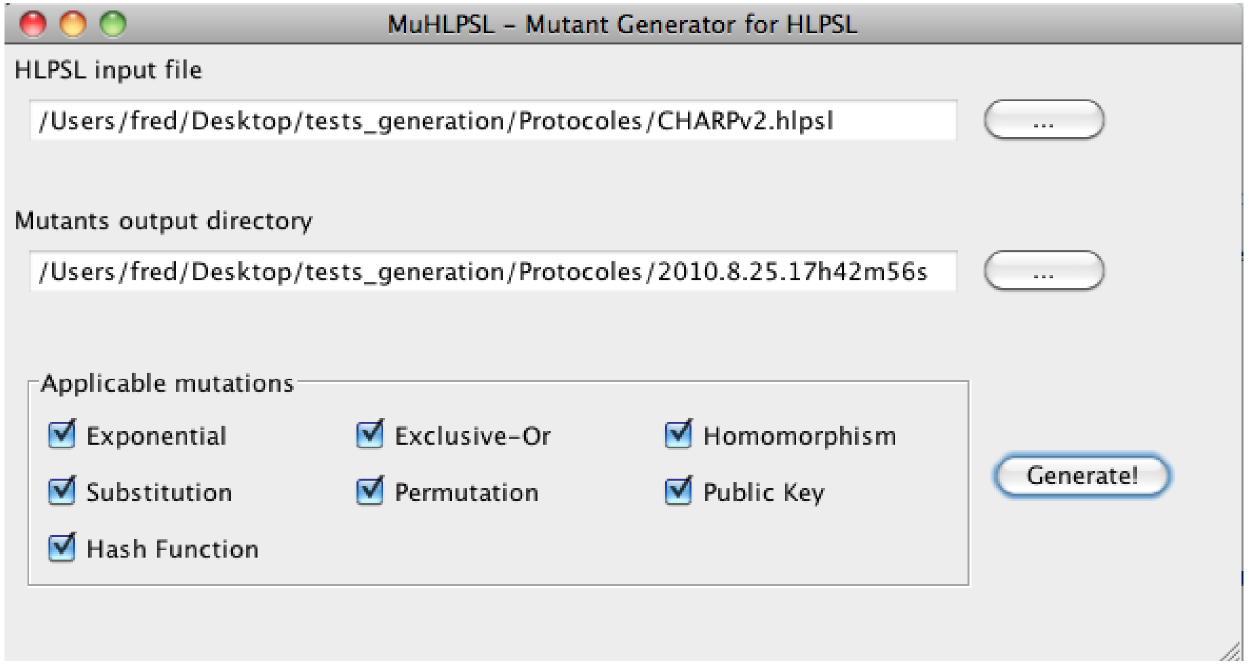
```
role alice (A, B: agent, Ka, Kb: public_key, SND, RCV: channel (dy))
played_by A def=
     local State: nat, Na, Nb: nat
     init State := 0
     transition
          0. State = 0 /\ RCV(start) =|>
              State':= 2 /\ Na' := new() /\ SND({A.Na'}_Kb)

          2. State = 2 /\ RCV({Na.Nb'.B}_Ka) =|>
              State':= 4 /\ SND({Nb'}_Kb)
end role
```

Experimentation

Principe:

- GUI for applying selected mutation operators to HLPSL models
- Produces the mutants that can be then checked by AVISPA

Freely available at
`http://disc.univ-fcomte.fr/˜fdadeau/tools/jMuHLPSL.jar`
To be completed by the AVISPA tool-set available at
`http://avispa-project.org` (web version available)



Experimentation

Application to security protocols:

- Application to a bench of 50 protocols available from the AVISPA website
- Computation of 1075 mutants in a few seconds
- Analysis of the mutants in less than one hour

Mutation	Safe	Unsafe	Incoherent	Total
Exponential	30 (90%)	0	3	**33**
Exclusive or	17	13 (39%)	3	**33**
Homomorphism	18	15 (45%)	0	**33**
Public key	45	2	0	**47**
Substitution	286	3	134 (31%)	**425**
Hash Functions	47	24 (30%)	8	**79**
Permutation	420	1	4	**425**
Total	**863**	**60**	**152**	**1075**

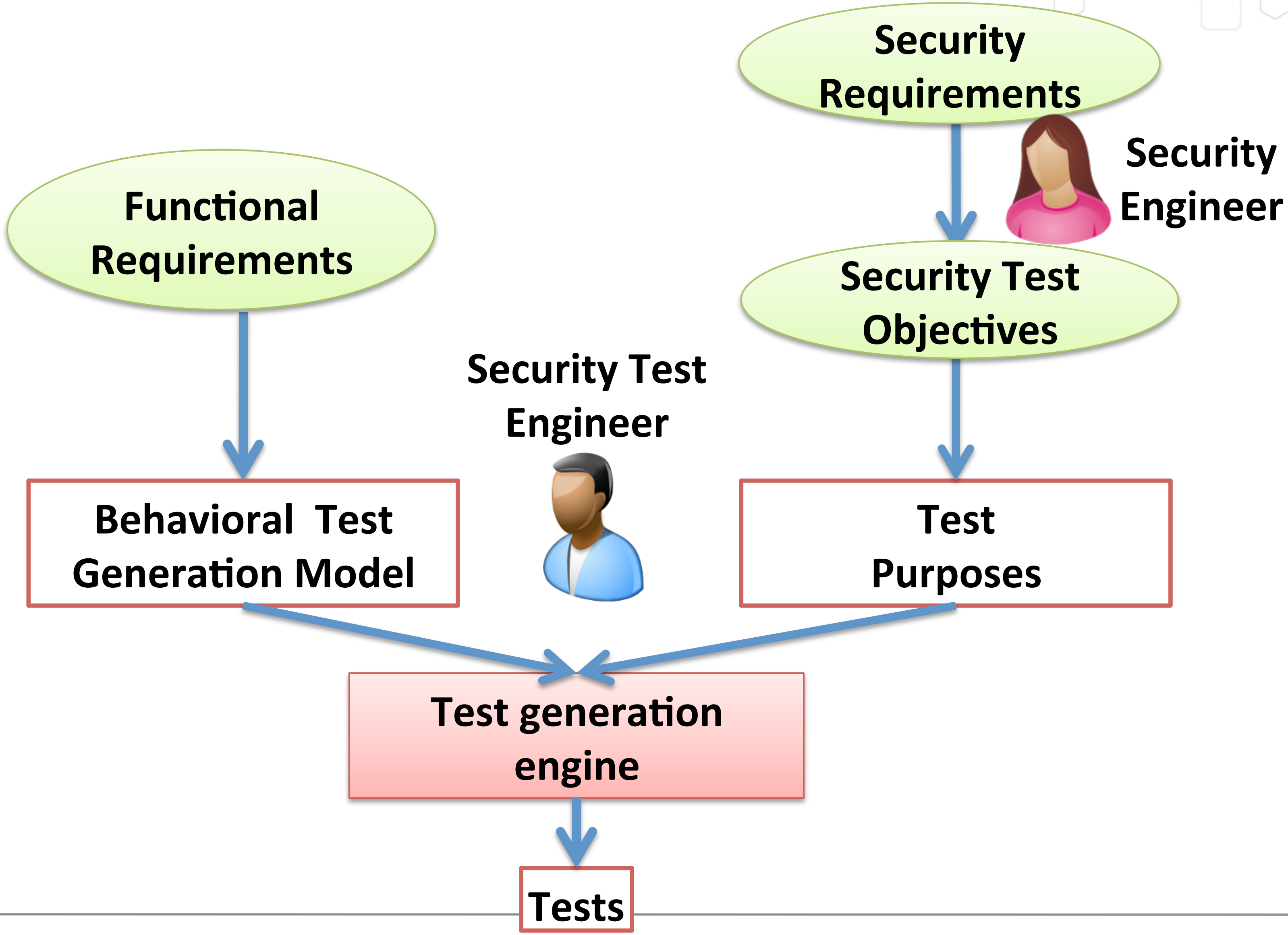
Safe: no attack — Unsafe: attack trace found — Incoherent: unexecutable protocol

An example of attack trace

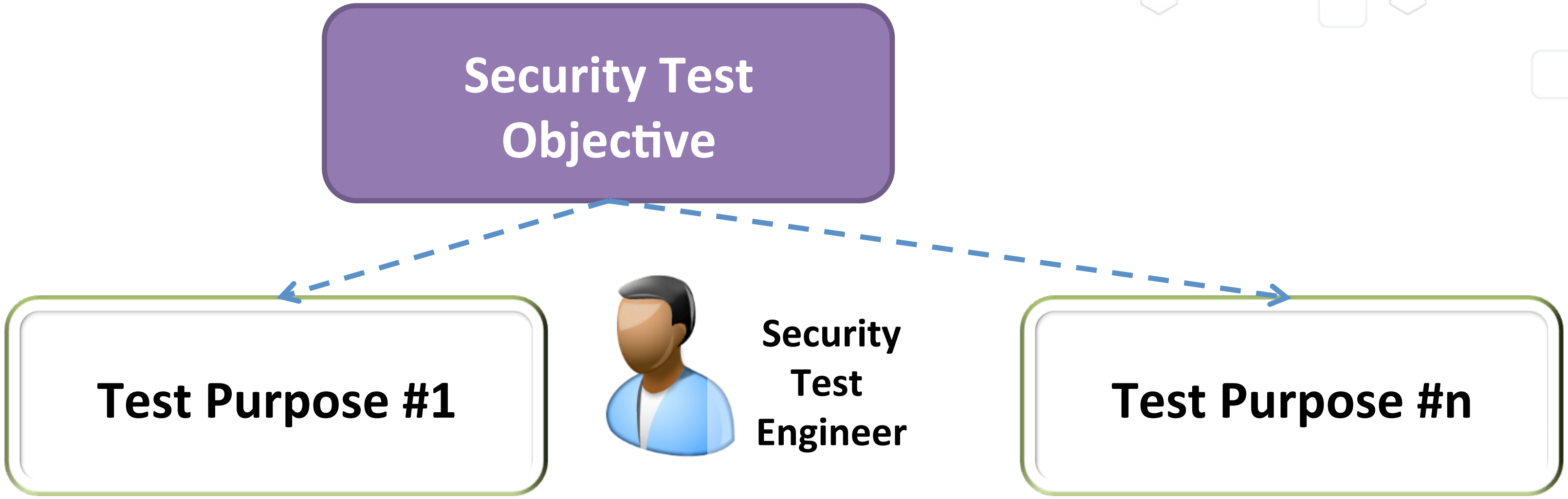
```
SUMMARY
UNSAFE

DETAILS
ATTACK_FOUND
TYPED_MODEL

PROTOCOL
PBK-fix-weak-auth_permut_2.if

GOAL
Authentication attack on
(a,a,msg,tag2)

BACKEND
CL-AtSe

STATISTICS
Analysed : 246 states
Reachable : 106 states
Translation: 0.00 seconds
Computation: 0.00 seconds
```

```
ATTACK TRACE
i -> (a,12): start
(a,12) -> i: i.{tag1.n21(Msg)}_(inv(pk_a)).
{pk_a}_f & Witness(a,a,msg,n21(Msg));

i -> (a,3): start
(a,3) -> i: b.{tag1.n1(Msg)}_(inv(pk_a)).
{pk_a}_f & Witness(a,a,msg,n1(Msg));

i -> (a,12): tag1
(a,12) -> i: {tag1.tag2}_(inv(pk_a))

i -> (b,4): b.{tag1.tag2}_(inv(pk_a)).
{pk_a}_f (b,4) -> i: n5(Nonce)

i -> (a,3): n5(Nonce)
(a,3) -> i: {n5(Nonce).tag2}_(inv(pk_a))

i -> (b,4): {n5(Nonce).tag2}_(inv(pk_a))
(b,4) -> i: ()
& WRequest(a,a,msg,tag2);
```

MBT for Security requirements testing



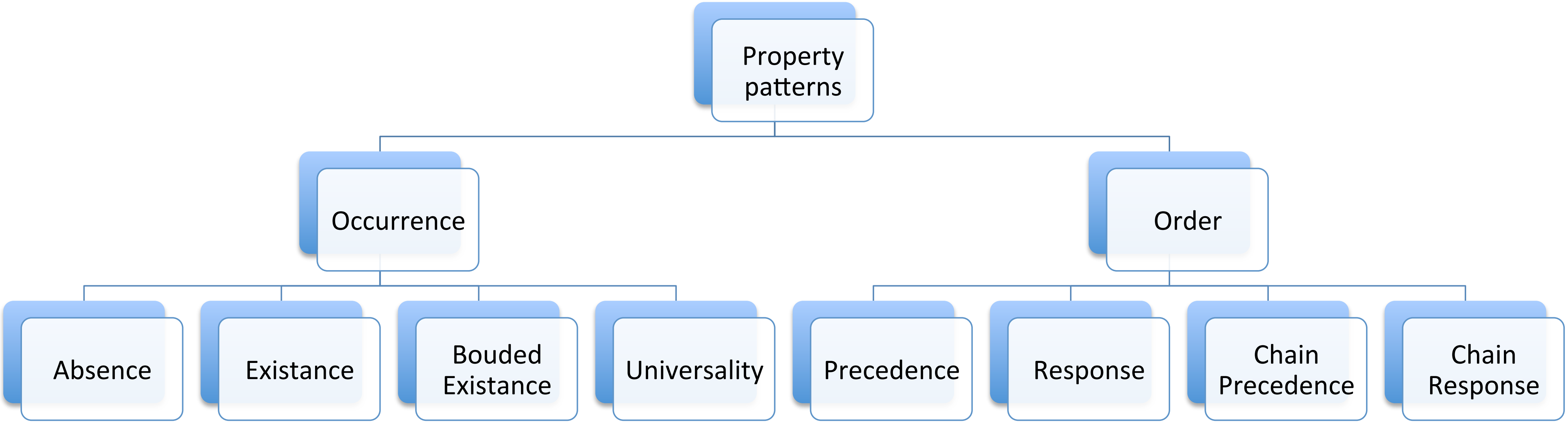
From test purposes to test case specs



Define Properties [Dwyer et al. 99]

Works:

- Analysis 555 properties (not only security) and define simple language to cover 92%
- 8 Patterns organized under a semantic classification:

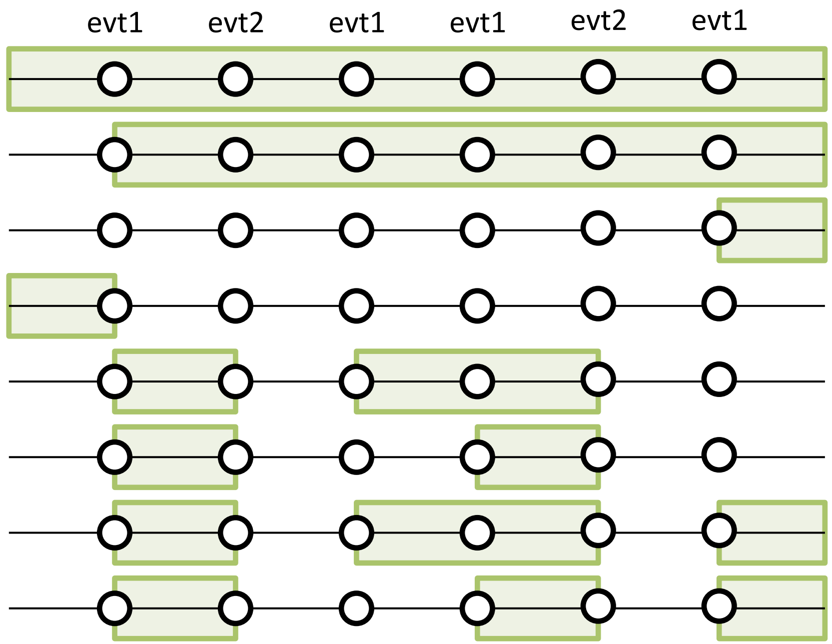


Define Properties Tocl [Cabrera et al. 11]

Property decomposed by: scope + pattern + events

- Scope: is part of pattern observation
- 5 scopes:

globaly
after [last] evt1
variante 'last'
before evt1
between [last] evt1 *and* evt2
variante 'last'
after [last] evt1 *until* evt2
variante 'last'



Define Test objective with Tocl [Cabrera et al. 11]

Test objective is decomposed by: scope + pattern + events

- Pattern: element to verify (under the scope)
- 5 Patterns:

```
never evt
always pred
evt1 [directly] follows evt2
evt1 [directly] precedes evt2
eventually [ (exactly/at least/ at most) k times]evt
```

- Events: all methods of UML/OCL model

Examples

A ticket purchase may be performed only by an existing and successfully logged user:

```
eventually isCalled(buyTicket, including:{@AIM:BUY_Success})
at least 0 times
between becomesTrue(not(self.current_user.isOclUndefined())) and
becomesTrue(self.current_user.isOclUndefined())
```

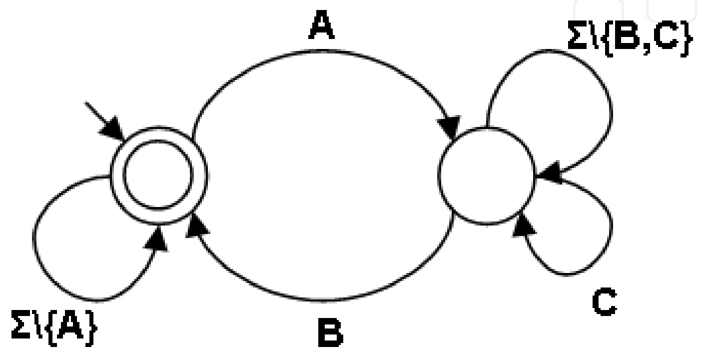
- Or with as a robustness point of view

```
never isCalled(buyTicket, including:{@AIM:BUY_Success} after
isCalled(logout, including:{@AIM:LOGOUT_Success}
unless isCalled(login, including:{@AIM:LOGIN_Success})
```

Examples

Automata representation:

- A = login
- B = logout
- C = buyTicket



Cover automata to produce test purpose:

- Σ\{login}*.login.(Σ\{logout,buyTicket}|buyTicket).logout

- 2 Test cases to cover all node and all edge coverage criterion:

```
1. init; sut.registration(USER1,PASSWD1} ;
    sut.login(REGISTERED_USER,REGISTERED_PWD);
    sut.showBoughtTickets(); sut.logout().
2. init;
    sut.login(REGISTERED USER,REGISTERED PWD);
    sut.buyTicket(TITLE1); sut.logout();
```

Language of TEST SCHEMA

Simple sequence (Bloc):

use Op1 then use Op2 then use Op3

Sequence of operations with state to reach:

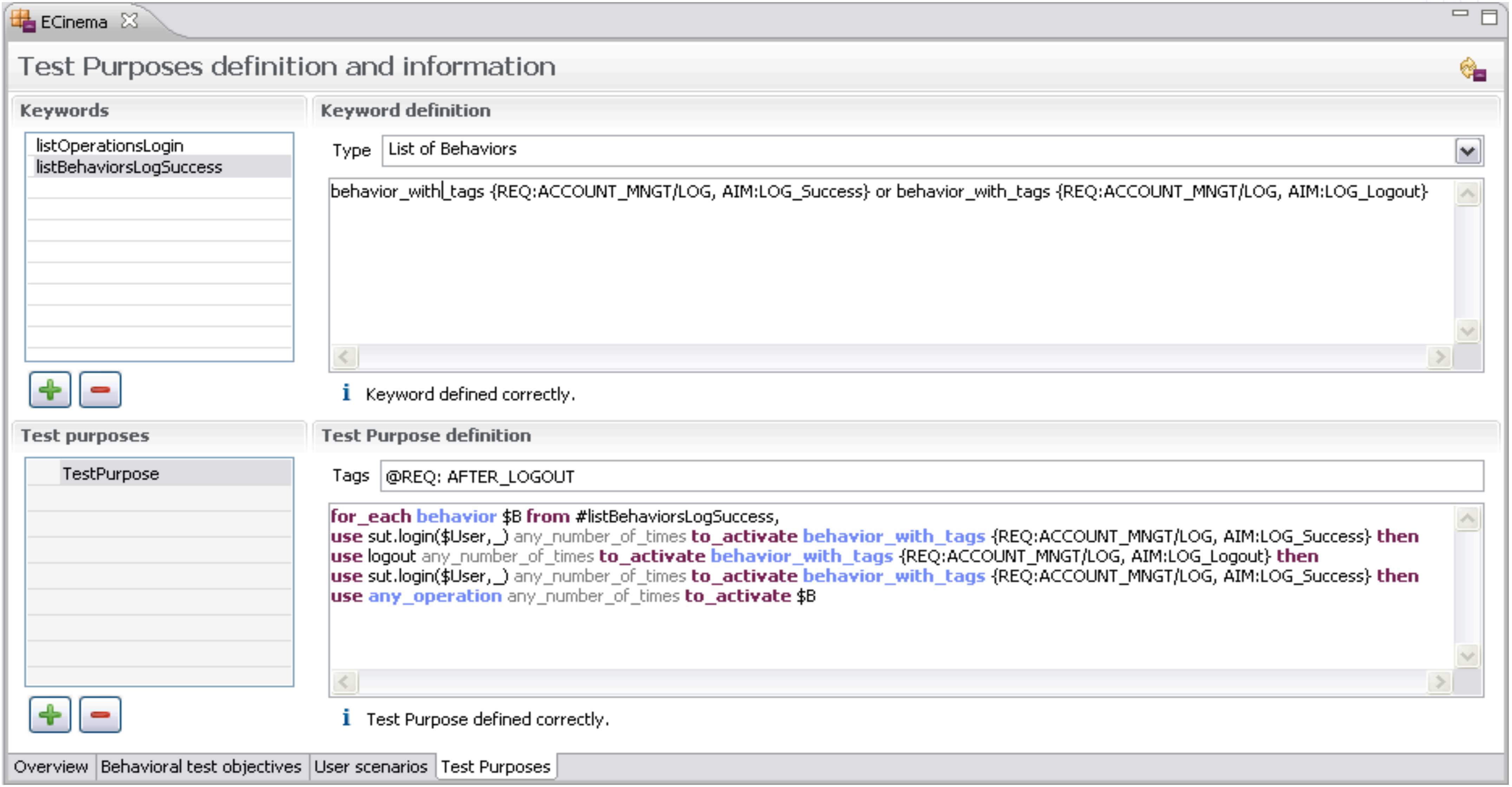
use Op1 then use any_operation to_reach state_respecting *"self.var1 = value2"* on_instance *"object1"*

Quantified Sequence of operations with state to reach:

for_each $Ops from Op1 or Op2 or Op3, use Op1 then use any_operation to_reach state_respecting *"self.var1 = value2"* on_instance *"object1"* *then* use $Ops

Language for Test Case Specification

"Try to login / logout at least twice."



From test purposes to test case specs

"Try to login / logout at least twice."

Security Test Objective

Test Purpose #1

Security Test Engineer

Test Purpose #n

Unfolding

Test Purpose definition

Tags: @REQ: AFTER_LOGOUT

```
for_each behavior $B from #listBehaviorsLogSuccess,
use sut.login($User,_) any_number_of_times to_activate behavior_with_tags {REQ:ACCOUNT_MNGT/LOG, AIM:LOG_Success} then
use logout any_number_of_times to_activate behavior_with_tags {REQ:ACCOUNT_MNGT/LOG, AIM:LOG_Logout} then
use sut.login($User,_) any_number_of_times to_activate behavior_with_tags {REQ:ACCOUNT_MNGT/LOG, AIM:LOG_Success} then
use any_operation any_number_of_times to_activate $B
```

Automated Test Generation based on the behavioral model

TC_1 | Unreachable | … | TC_n

Test cases

Unfolding of one test purpose

Test Purpose definition

Tags: @REQ: AFTER_LOGOUT

```
for_each behavior $B from #listBehaviorsLogSuccess,
use sut.login($User,_) any_number_of_times to_activate behavior_with_tags {REQ:ACCOUNT_MNGT/LOG, AIM:LOG_Success} then
use logout any_number_of_times to_activate behavior_with_tags {REQ:ACCOUNT_MNGT/LOG, AIM:LOG_Logout} then
use sut.login($User,_) any_number_of_times to_activate behavior_with_tags {REQ:ACCOUNT_MNGT/LOG, AIM:LOG_Success} then
use any_operation any_number_of_times to_activate $B
```

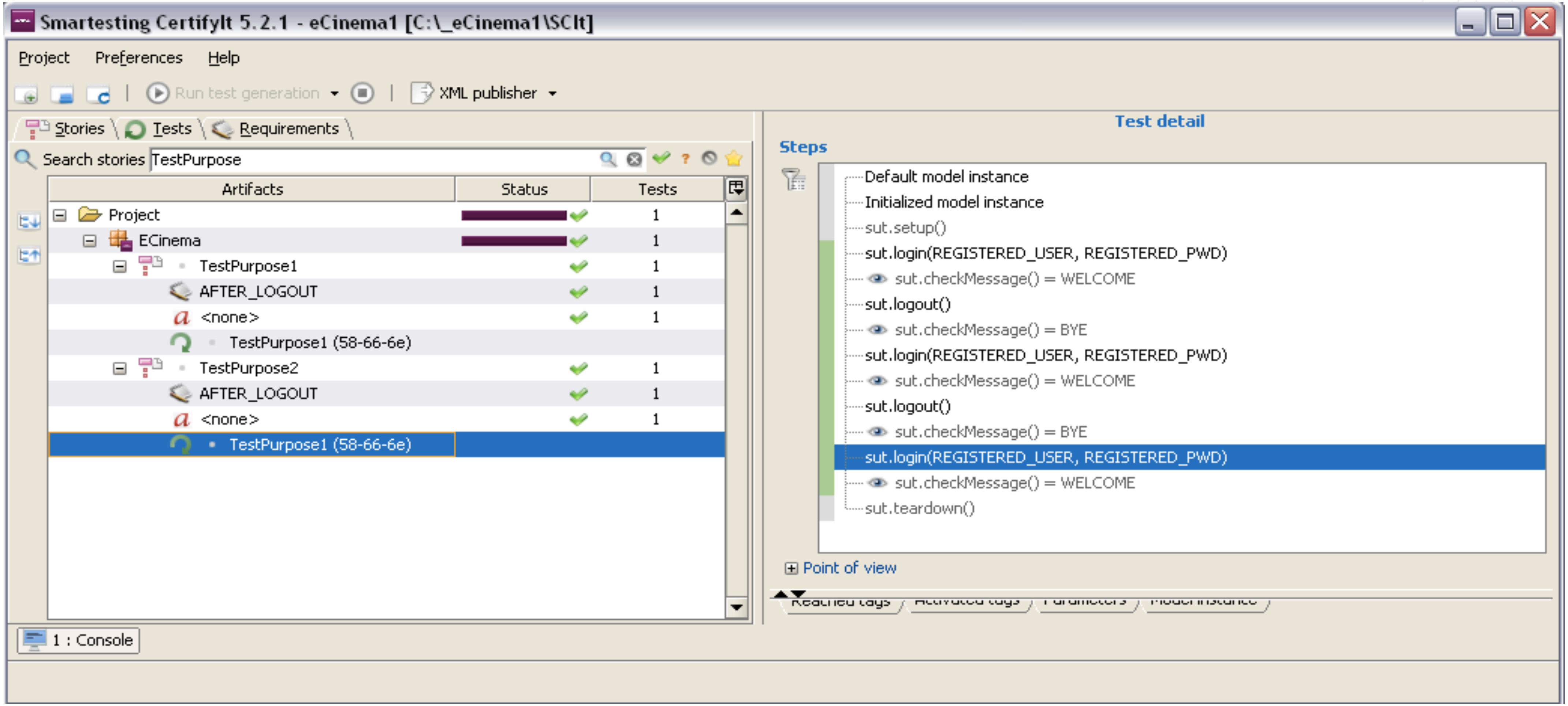
Unfold

use sut.login($User,_) *any_number_of_time*
to_activate behavior_with_tags {REQ:ACCOUNT_MNGT/LOG,AIM:LOG_Success} *then*
use logout*any_number_of_time*
to_activate behavior_with_tags {REQ:ACCOUNT_MNGT/LOG,AIM:LOG_Logout} *th*
use sut.login($User,_) *any_number_of_time*
to_activate behavior_with_tags {REQ:ACCOUNT_MNGT/LOG,AI
Use any_operation any_number_of_time to_activate beh … _MNGT/LOG,AIM:LOG_Logout}

use sut.login($User,_) *any_number_of_tim*
to_activate behavior_with_tags … IM:LOG_Success} *then*
use logout*any_number_of*
to_activate behavior … NT_MNGT/LOG,AIM:LOG_Logout} *then*
use sut.login($User,_) *an* … *_time*
to_activate behavior_w … tags {RAQ:ACCOUNT_MNGT/LOG,AIM:LOG_Success} *then*
Use any_operation any_number_of_time to_activate behavior_with_tags {RAQ:ACCOUNT_MNGT/LOG,AIM:LOG_Success}

Test Case Specifications

Test generated



MBT Functional and Security

Presented approach interest:

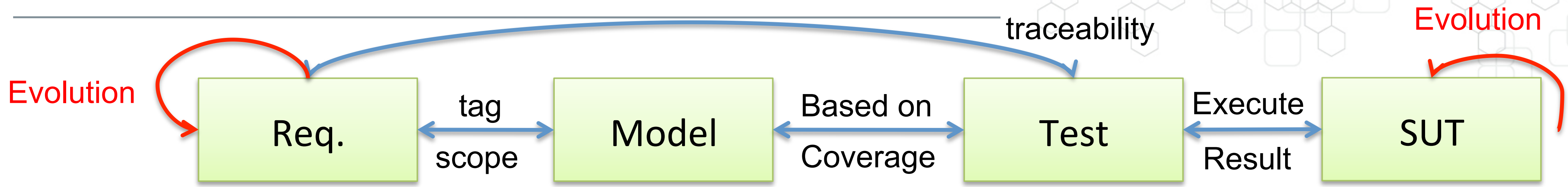
- Dedicated model for special aspect of security provides good results

- Structural coverage of the functional model is not sufficient for Security Reqs Testing:
  - Validate the correct implementation of security requirements by testing security functions
  - Need an adequate language to describe Security Test Objectives
  - Need for test selection criteria for this language

OUTLINE

I. Introduction of test
II. Functional Testing
III. Model-Based Testing
IV. Model-Based Testing and Security
V. Evolution of system
   i. What is an evolution?
   ii. How to compute evolution?
   iii. EvoTest an integrate framework

MBT and evolution



Two goals:

1. Ensure change parts of the System Under Test (SUT) behave as intended
2. Ensure unchanged parts of SUT are not affected adversely

Three tasks:

- Test interactions between code and model modification (evolution)
  1. Test model modification
  2. Test code modification
- Test interactions between unchanged parts of model and code (non regression)
  3. Test side-effects of modification on unchanged parts of model

Evolution impacts

Change management in the MBT process for functional requirements testing and security requirements testing

- Using model version comparison (test models evolve to reflect the change)
- Bidirectional traceability between generated tests and requirements
- Optimizing the MBT process (avoiding systematic re-generation)
- Structuring the test repository wrt change in the requirements

Automated test generation for security requirements testing

- From security requirements to the formalization of testing needs using Test Purposes
- Ensuring security requirements coverage
- Bidirectional traceability between generated tests and security requirements

State of the art 1/2

B. Korel, L. H.Tahat, and B. Vaysburg, "*Model based regression test reduction using dependence analysis*," in IEEE ICSM'02, 2002

- Three types of testing in case of modification: testing the **modified part**, test the **remaining part** of the system, test the **side-effects**
- Use of interaction patterns to reduce the original test suite

Chen et Al. "*Model-based Regression Test Suite Generation Using Dependence Analysis*", AMOST'07.

- Make clear distinction between **effects** and **side-effects** due to changes
- Encapsulate possibly impacted elements due to changes using **data** and **control dependence analysis**
- The existing test suite is **not reused**, the Regression test suite is created from the machine model and the set of modified elements.

State of the art 2/2

B. Jiang, T. H. Tse, W. Grieskamp, N. Kicillof, Y. Cao, and X. Li, "*Regression testing process improvement for specification evolution of real-world protocol software*" in Proceedings of the 10th International Conference on Quality Software, 2010, pp. 62–71.

- Regenerate full **new suite is time consuming**! For Microsoft's Protocol documentation project test generation took nearly one full day.
- Goal: **Maximally reuse** the existing test cases and generate tests only for the new elements
- Technique: based on exploring and **comparing the model** graphs (original and evolved)
- Stress on the importance to **maintain the test suite** for further specification evolutions

KEY CHALLENGES

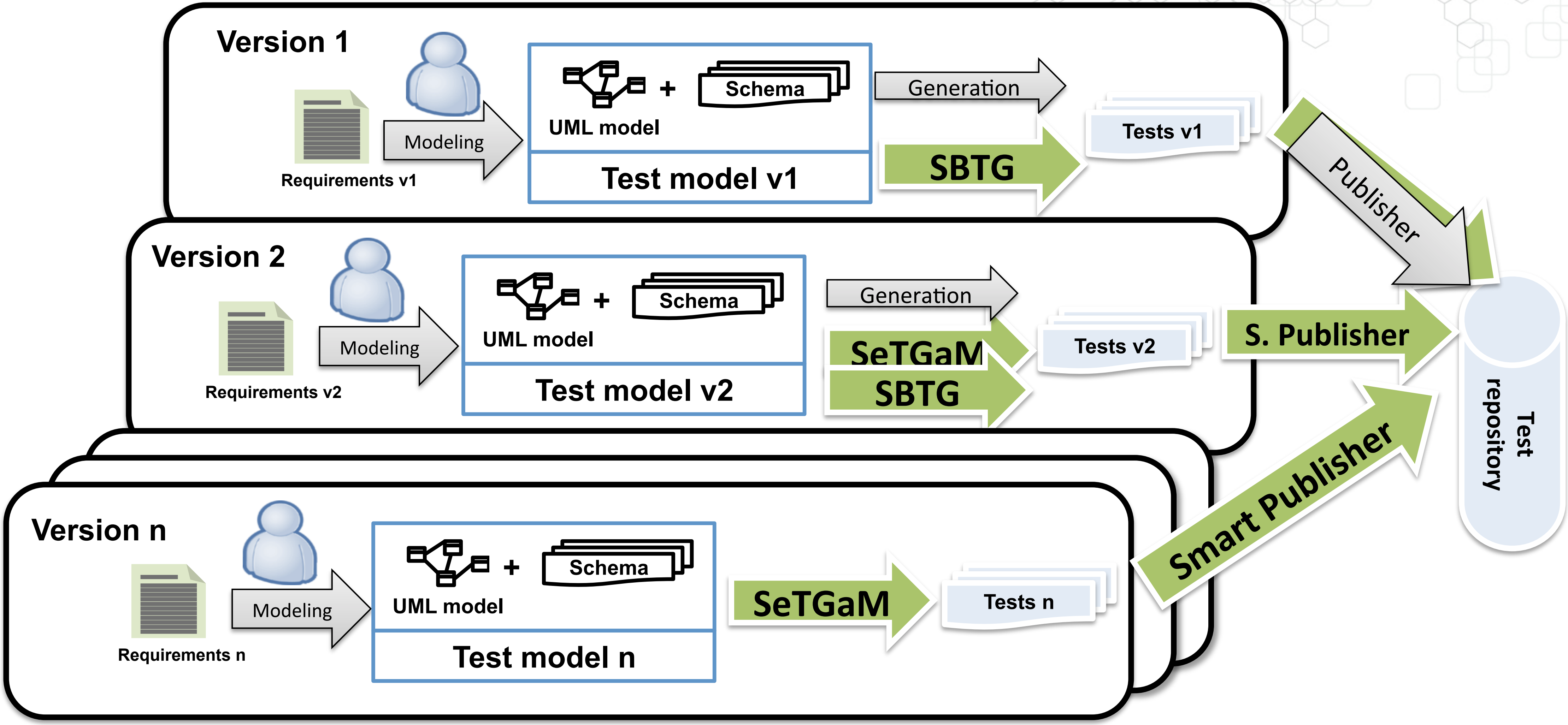
In a model-based testing perspective, key challenges are:

- The coverage of security requirements by the MBT process;
- The stability of the generated tests through evolution;
- The organization of the generated tests w.r.t. evolution;
- The efficiency of this test generation process.

Those challenges are connected to the following fields:

- Model-based testing
- Security requirements testing from behavioral test generation models and test purposes
- Regression testing

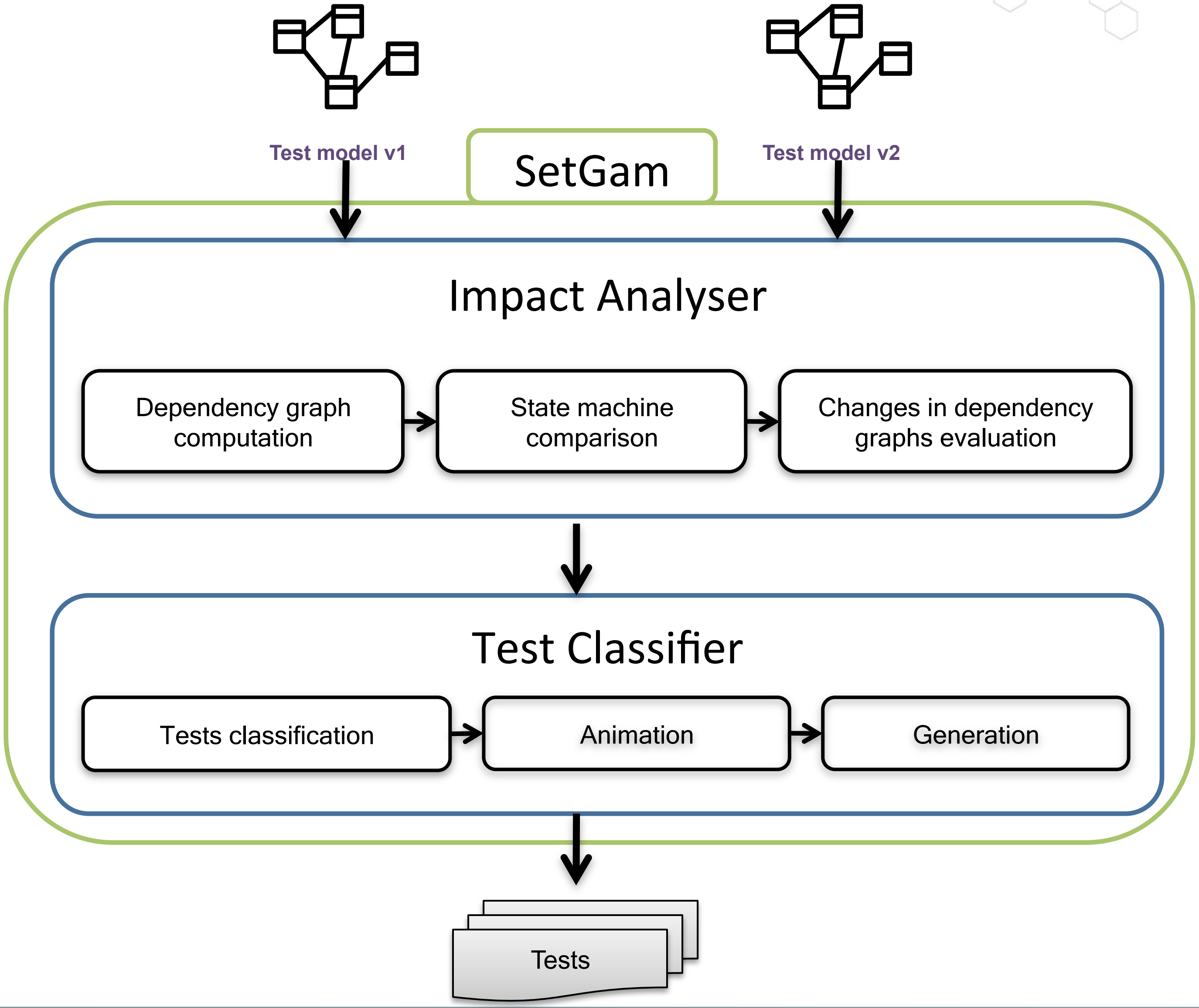
MBT Process for Evolving system



SeTGaM Selective Test Generation Method

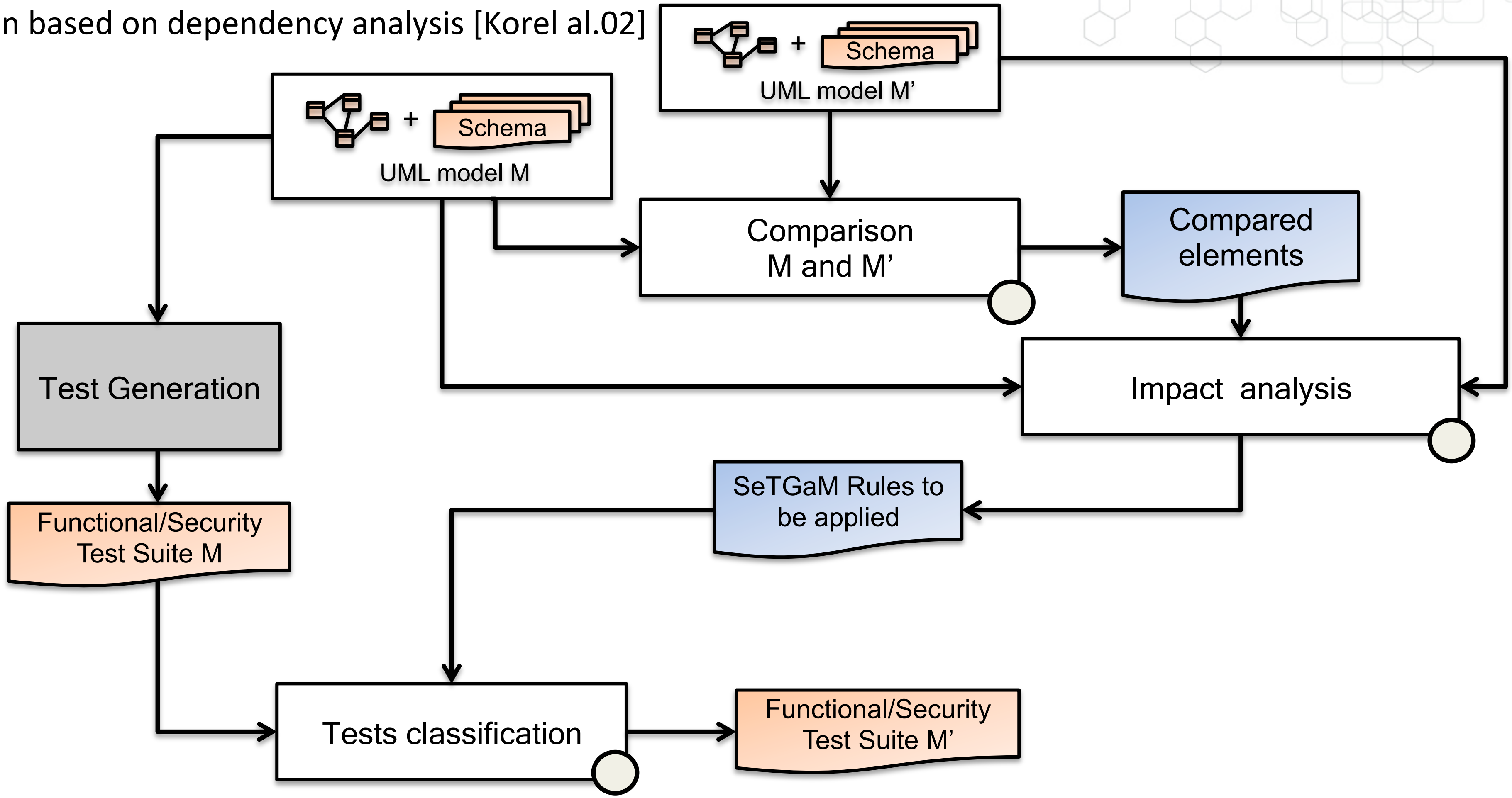
SBTG Schema-Based Test Generation

SETGAM Component Overview



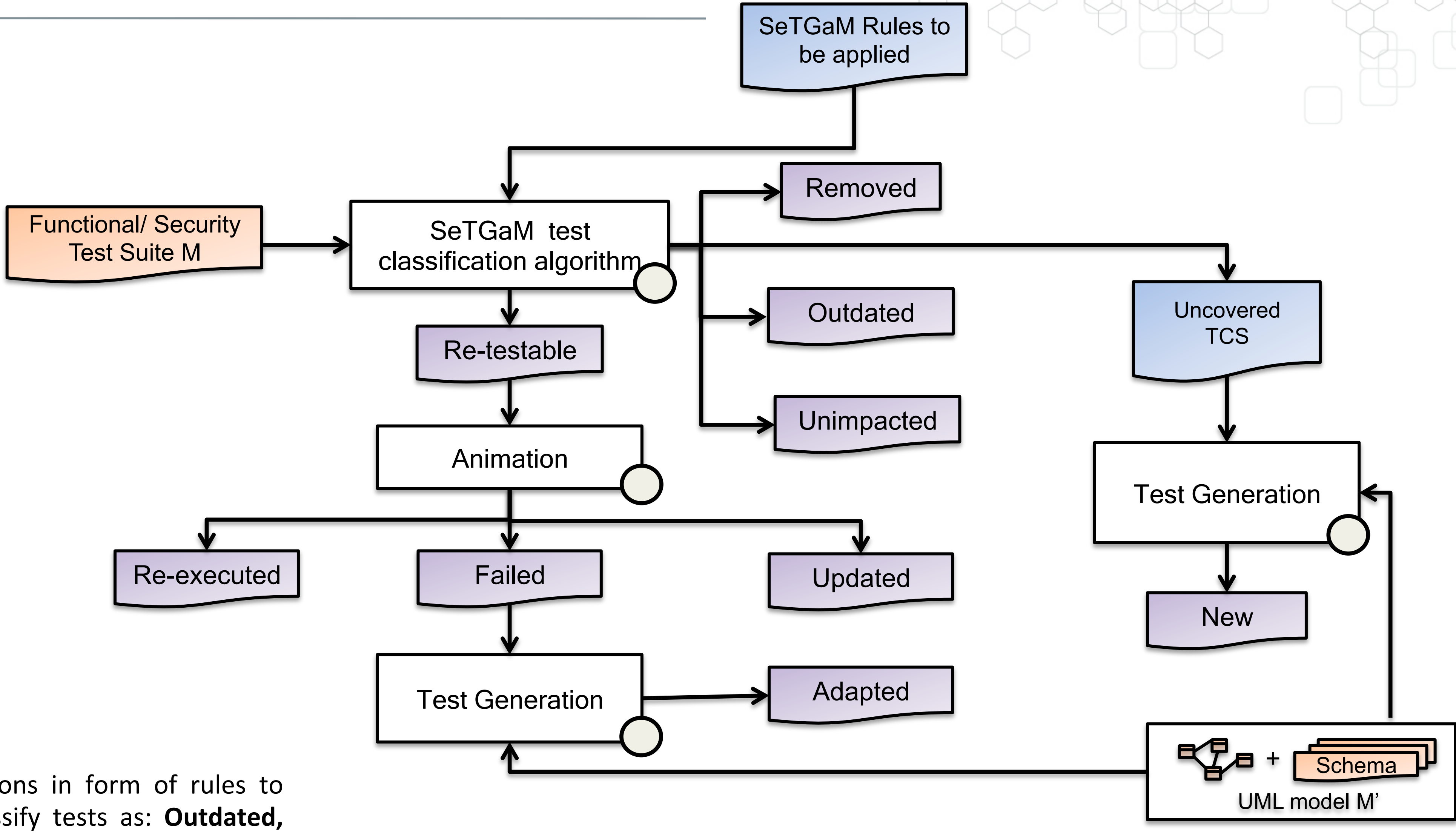
SeTGaM Process

Detection based on dependency analysis [Korel al.02]



- For the considered MBT approach we generate tests to cover requirements expressed in the initial model.
- In MBT a specification evolution leads to the creation of a new model version
- In order to test the evolved software, we proceed by detecting affected requirements
- we compare the models to obtain **new**, **deleted** and **modified** requirements
- we perform impact analysis based on **dependence graphs** for both versions
- we define actions in form of rules to select and classify tests.

Classification of test cases



- Defined actions in form of rules to select and classify tests as: **Outdated, Unimpacted, Removed and Re-testable**

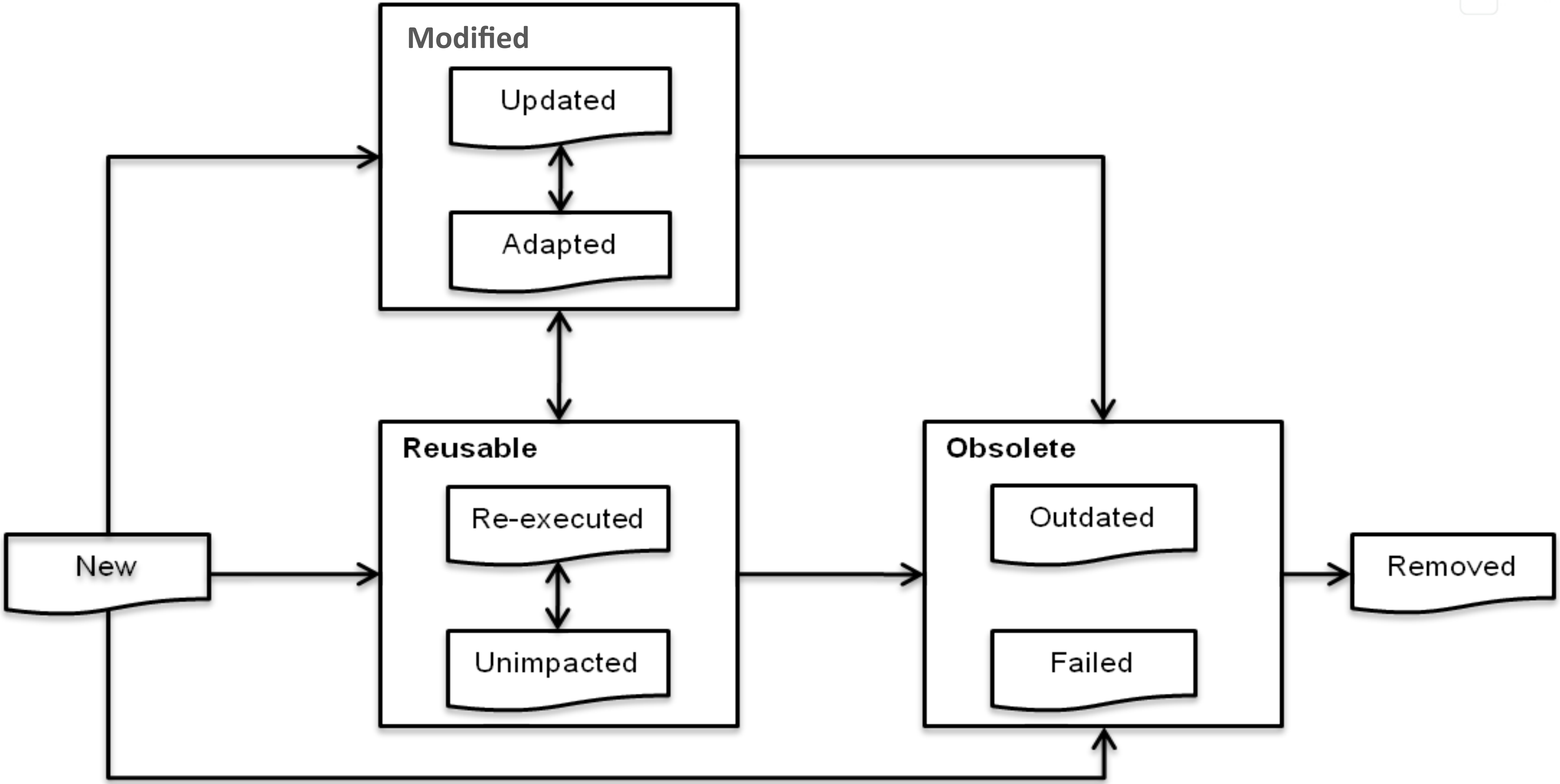
- To conclude about the final status: **Updated, Re-executed or Failed**, we animate Failed tests on the model.

- Failed tests cover existing requirements, to reach max. requirements coverage we need to **adapt** these tests.

- We generate tests for all **new** uncovered requirements

Test STATUS Life Cycle / Evolution

Status extends [Leung al.89]



Running example – eCinema

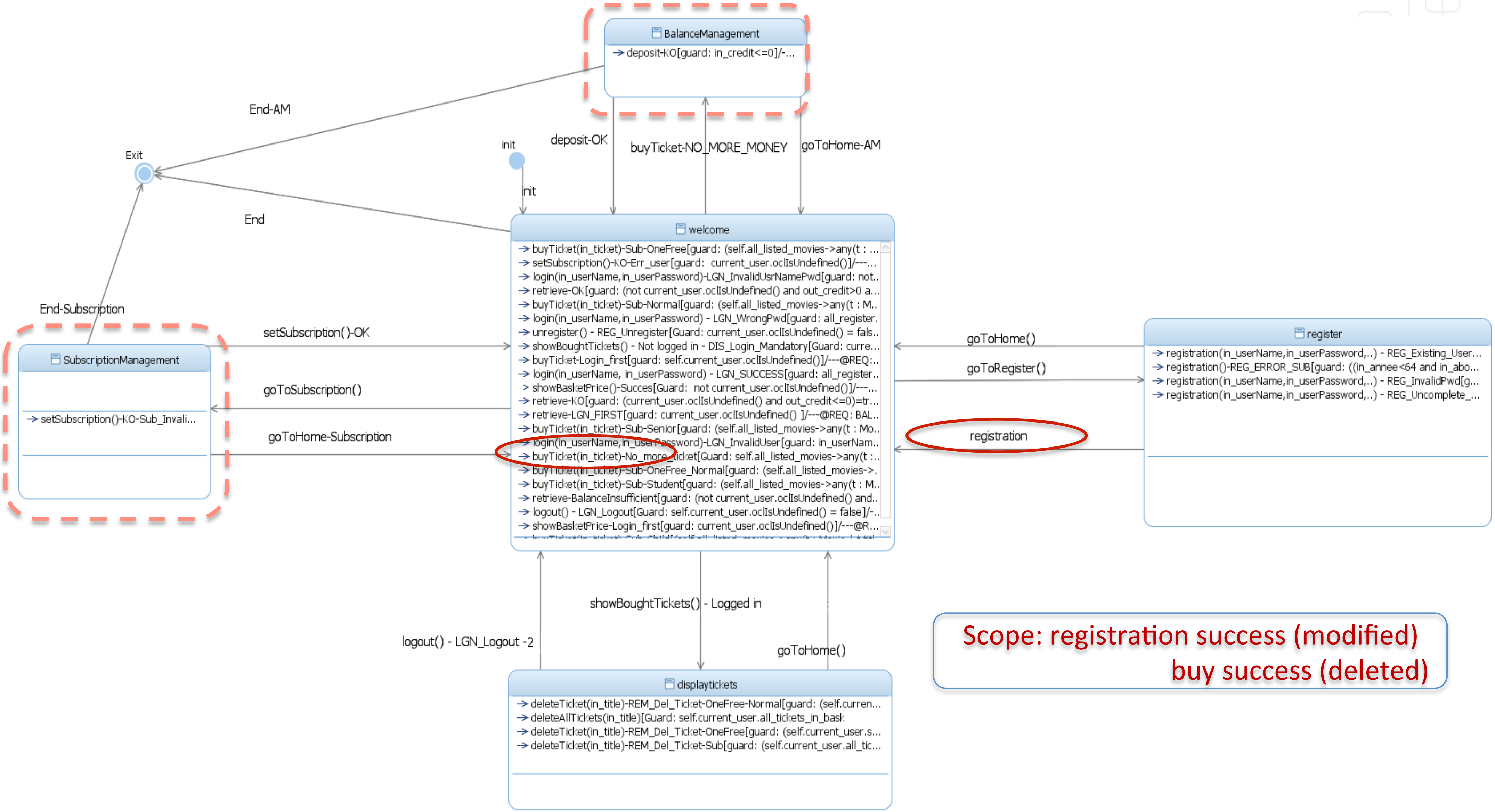
Specification:

- eCinema application aiming at booking movie tickets.

eCinema evolutions:

- Initial version: register, login, logout, buy ticket, remove ticket...
- Evolved version, having two major novelties:
  - type of subscription
  - account management
- In summary: 25 and 47 requirements were modelled for the initial and the evolved version, respectively

Running example – eCinema

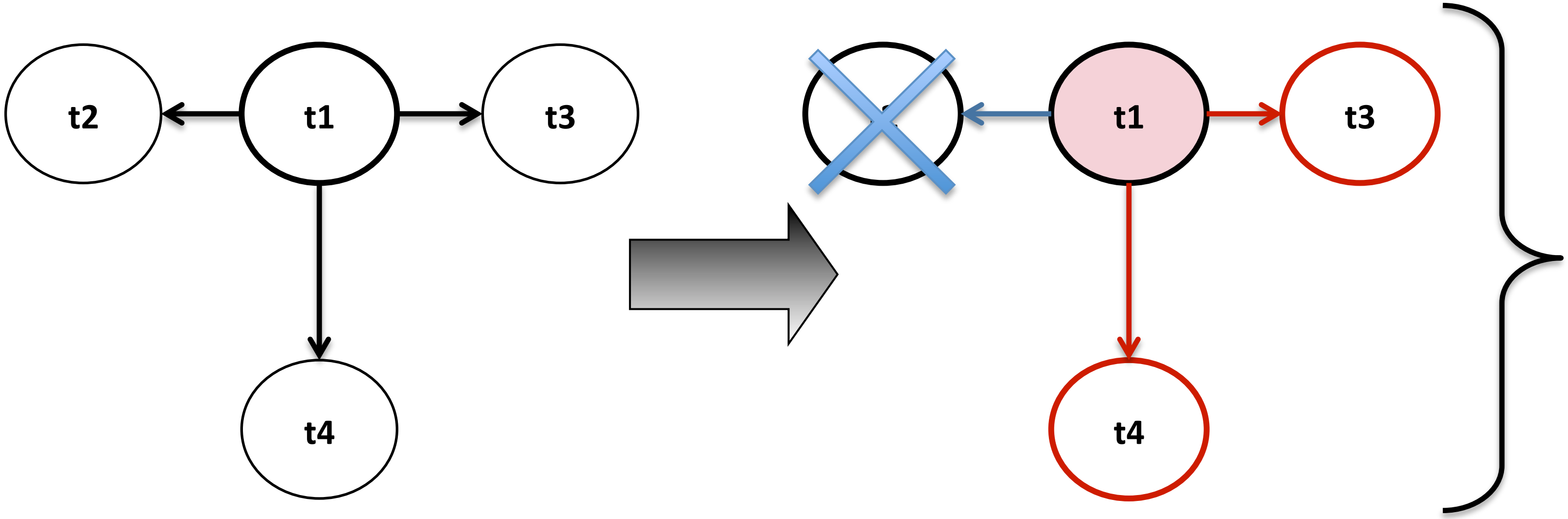


Scope: registration success (modified)
buy success (deleted)

eCinema -Dependence Analysis

- With the dependency analysis we can find impacted elements and thus create rules to apply for each case.

Impacted elements of the graph after the evolution



eCinema initial DDG

eCinema evolution DDG

- *Select tests covering requirements expressed in t3, t4*
- *Classify as outdated tests covering requirement in t2*

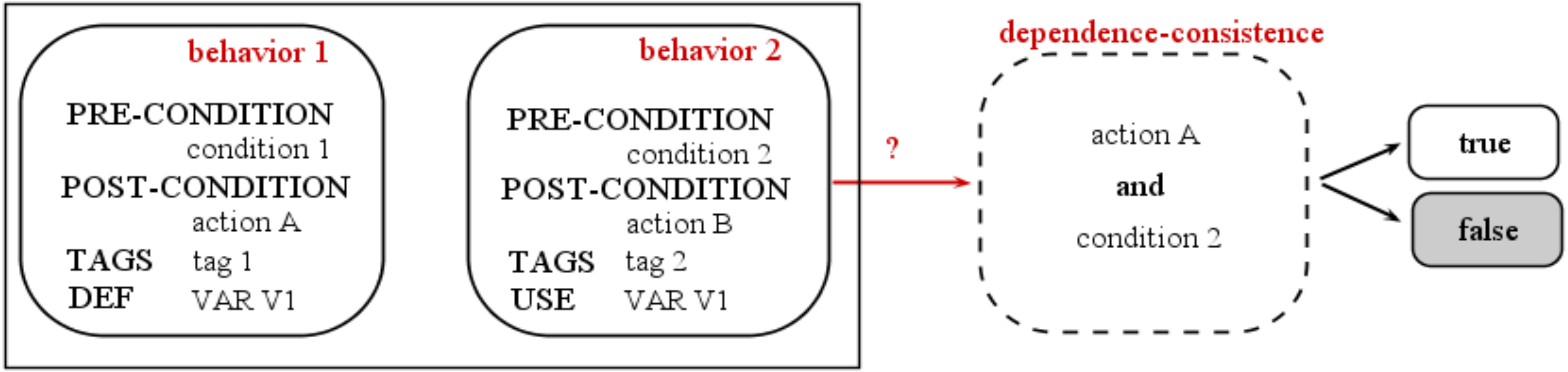
Legend:
t1: register_sucess t2: buy_ticket
t3: unregister t4: deleteAllTickets

Deleted data dep. →
Impacted transition →

SeTGaM – without statechart

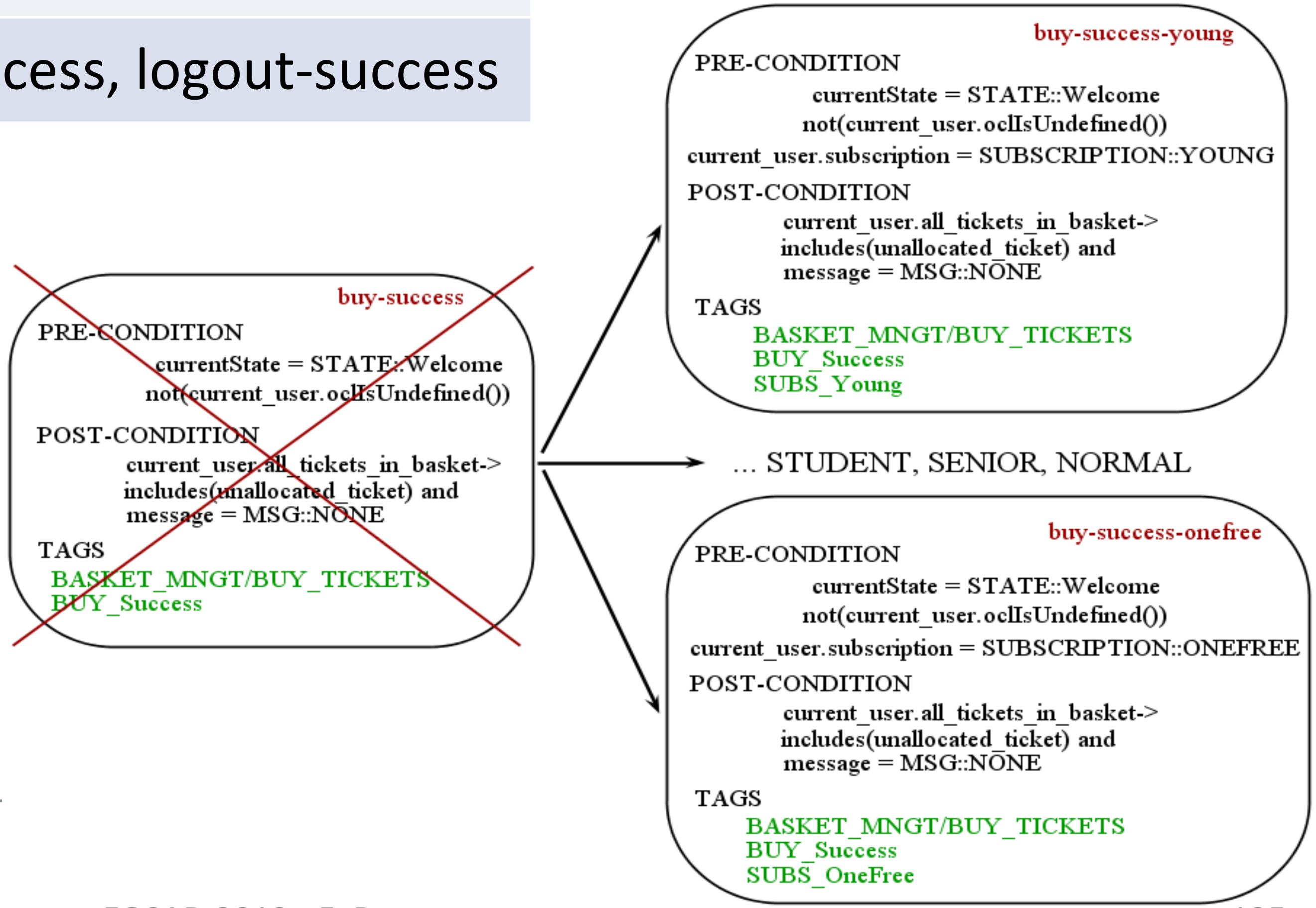
- No Statecharts for (data/control) dependencies computation.

- Based on def/use of the variables in operation behavior (@PRE,@POST,ΣTAGS)
  - Behavorial Data dependence: *"A behavior* 1 *is data-dependent from another behavior* 2 *w.r.t. variable* v1 *if and only if* v1 *is defined in* behavior 1 *and used in* behavior 2 *and there exists dependence-consistence w.r.t.* v *between behavior 2 and* behavior 1 "
  - Dependence Consistence: *"A behavior 2 is dependence-consistent w.r.t behavior 1 iff (behavior1.@POST && behavior2@PRE) is consistent."*

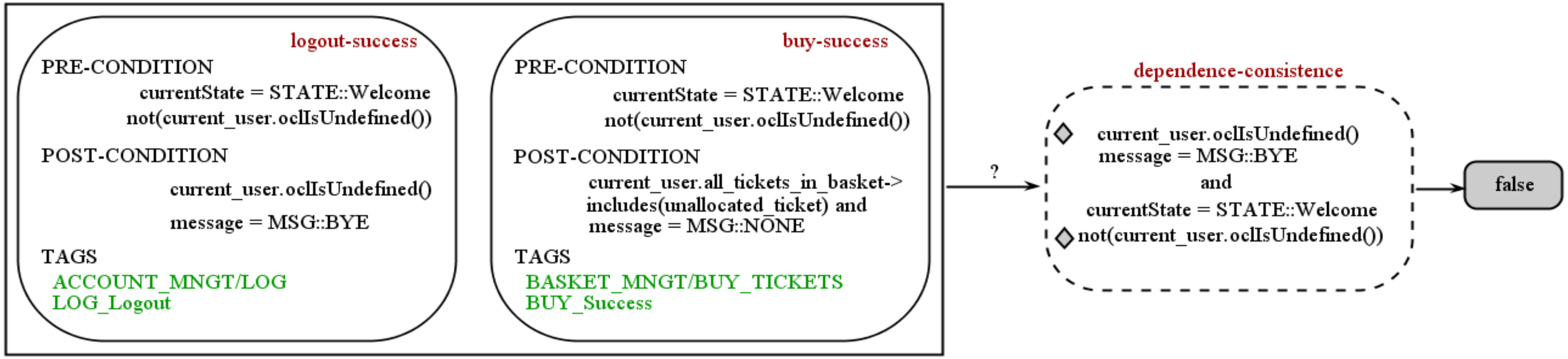


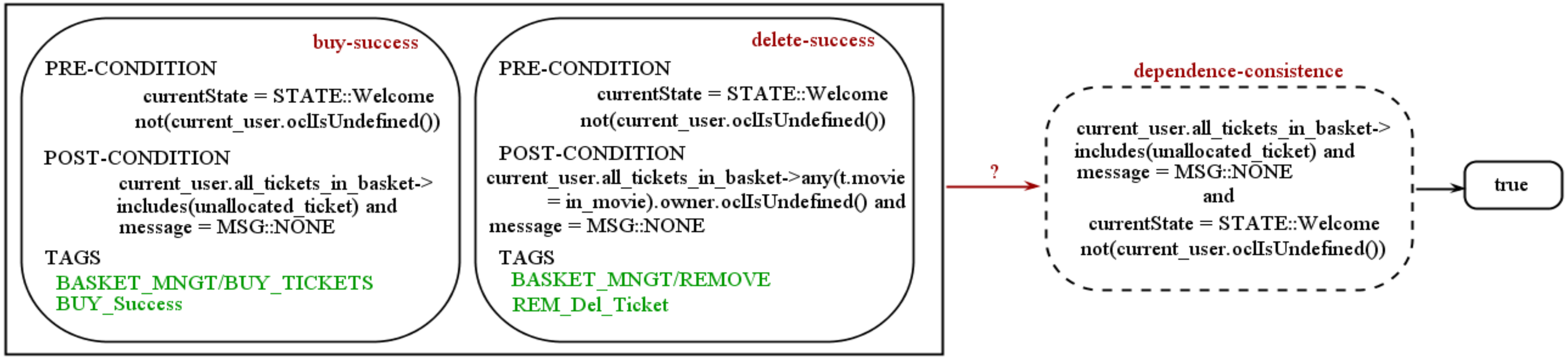
Running example – eCinema

Set of tests	
Test	**Covered behaviors**
T1	login-sucess, buy-success, delete-ticket
T2	login-sucess, buy-success, logout-success



Running example – eCinema

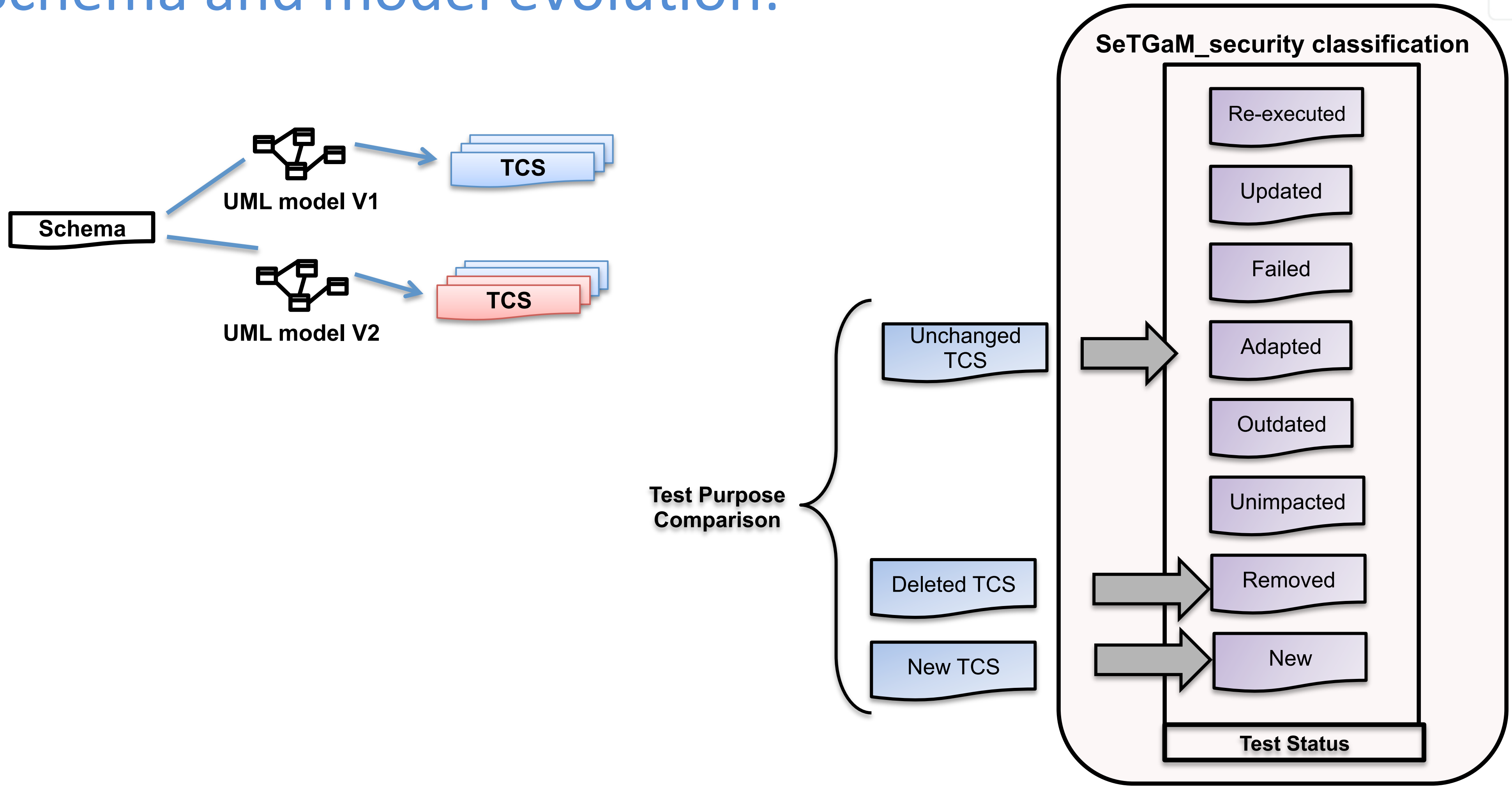




Set of tests		Classification
Test	**Covered behaviors**	**Status**
T1	login-sucess, buy-success, delete-ticket	Outdated -> Adapted
T2	login-sucess, buy-success, logout-success	Outdated

SeTgaM for security requirements

Schema and model evolution:



Running example – eCinema

- Security requirement : A ticket purchase may be performed only by an existing and successfully logged user.
- The associated schema, defined below produced 4 test case specifications and thus 4 tests (for the eCinema models 1 and 2, with and without statechart respectively).

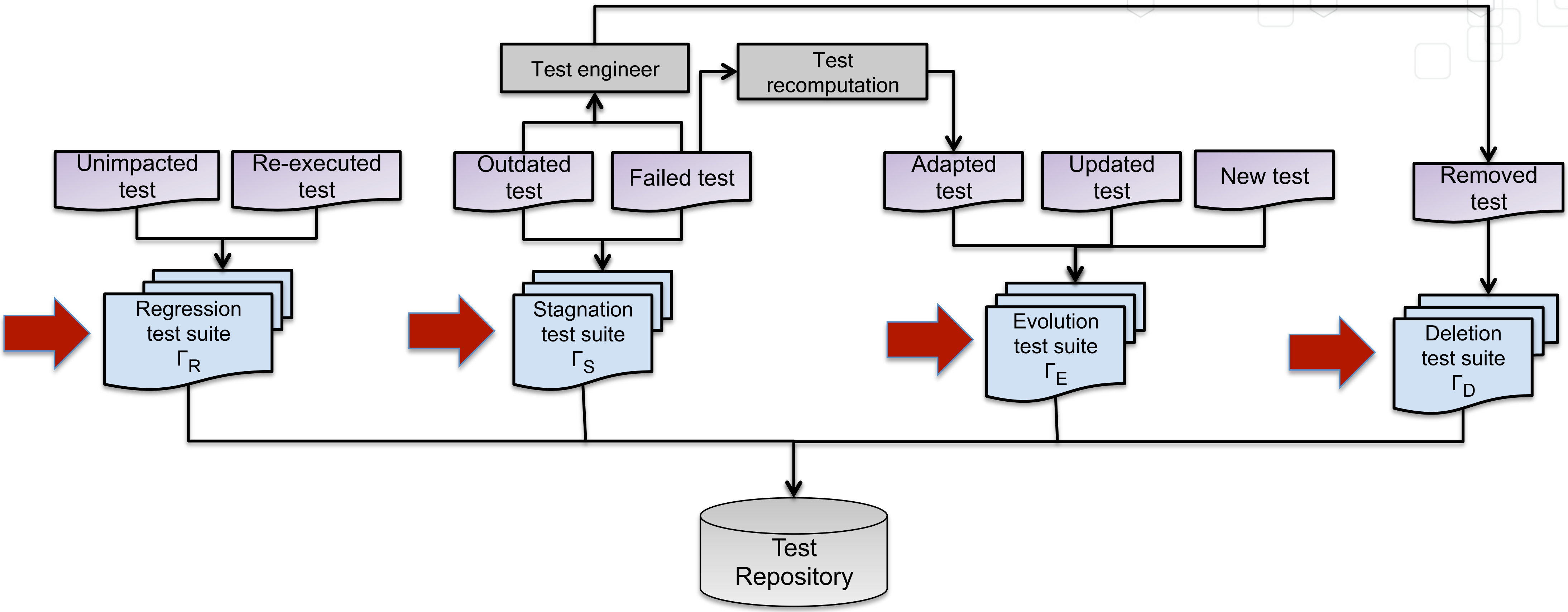
```
for_each operation $X from registration or login,
for_each operation $Y from unregister or logout,
for_each operation $Z from goToRegister,
use $Z 0..1 on_instance sut then
use $X at_least_once to_reach_state
"self.message = MSG : :WELCOME" on_instance sut then
use buyTicket at_least_once to_reach_state
"self.message = MSG : :NONE" on_instance sut then
use $Y at_least_once to_reach_state
"self.message = MSG : :BYE" on_instance sut then
use buyTicket at_least_once to_activate_behavior
without_tags {AIM :BUY_Success}
```

Outdated

Adapted

TCS	$Z	$X	$Y	Test - model 1	Test - model 2
TCS 2.1	goToRegister	registration	unregister	requirement28(bb-0c-3d)	requirement28(bb-f1-e3)
TCS 2.2	goToRegister	registration	logout	requirement210(bb-ca-6c)	requirement210(bb-72-5c)
TCS 2.3	-	login	unregister	requirement29(bb-c0-2c)	requirement29(bb-5c-3f)
TCS 2.4	-	login	logout	requirement211(bb-24-b0)	requirement211(bb-be-43)

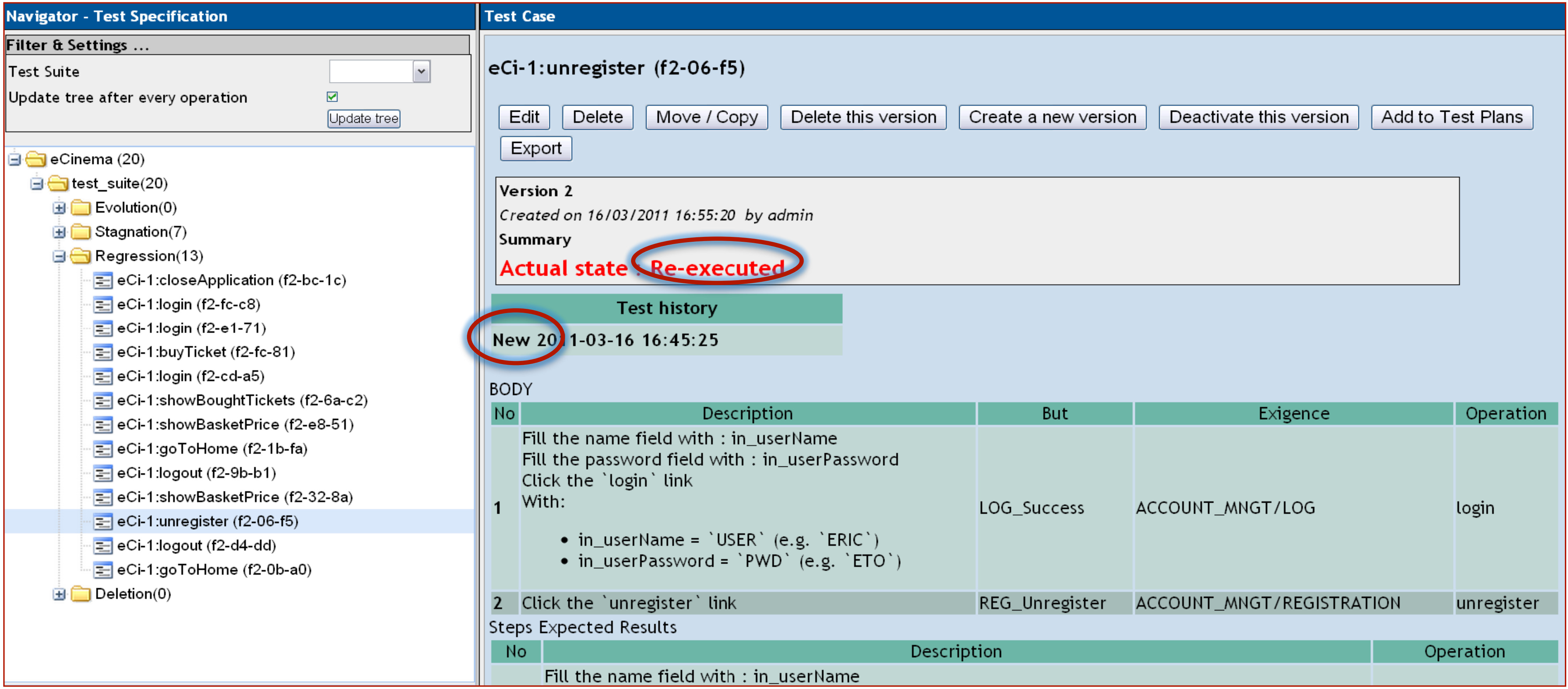
Test repository management



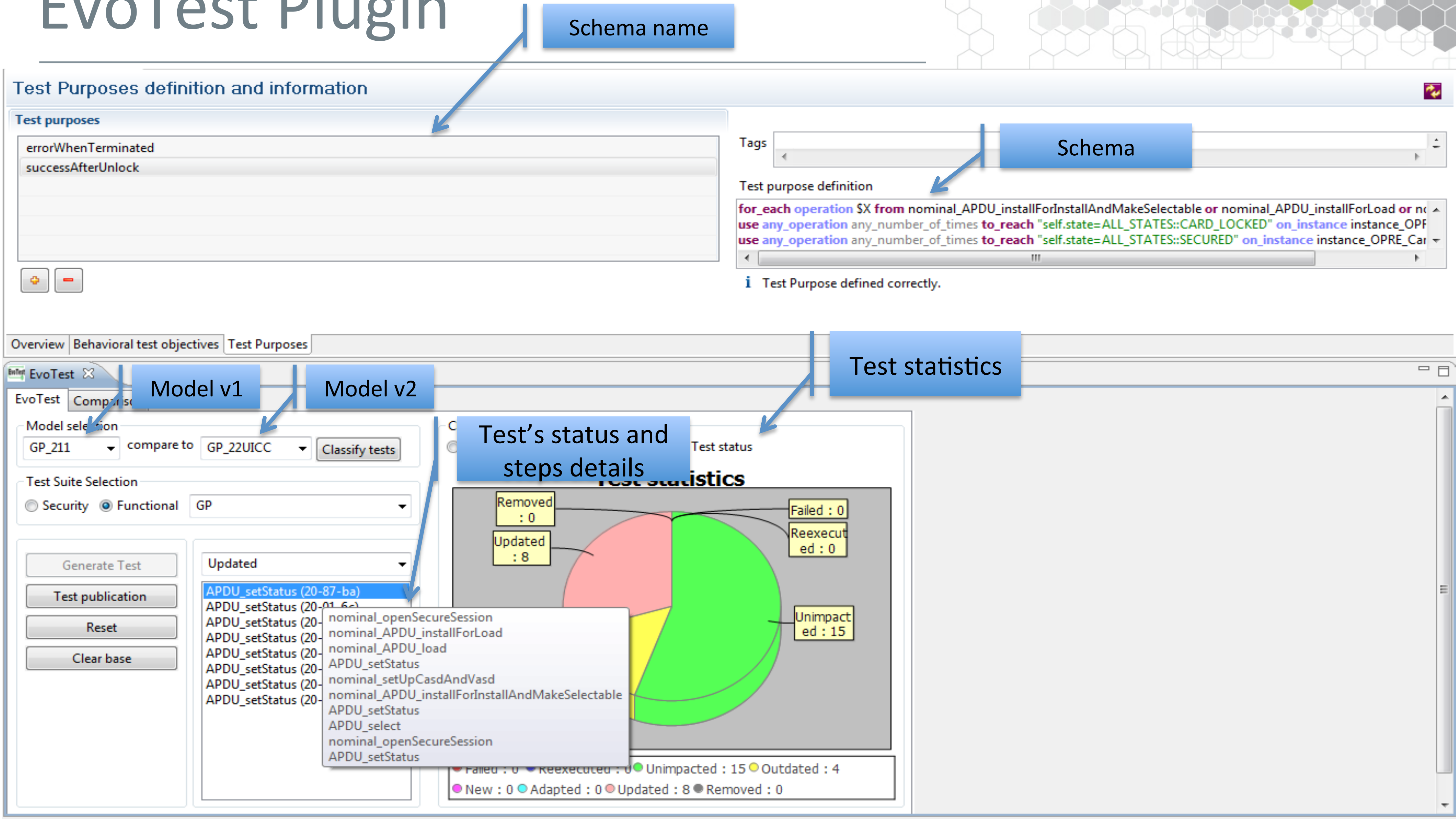
- Ensure that the evolution did not impact unmodified parts.
- Ensure that the evolution changed the system behaviour.
- Ensure that the system novelties behave correctly.
- Ensure maximal test history and contains tests from the Stagnation test suite of previous version.

Smart Publisher

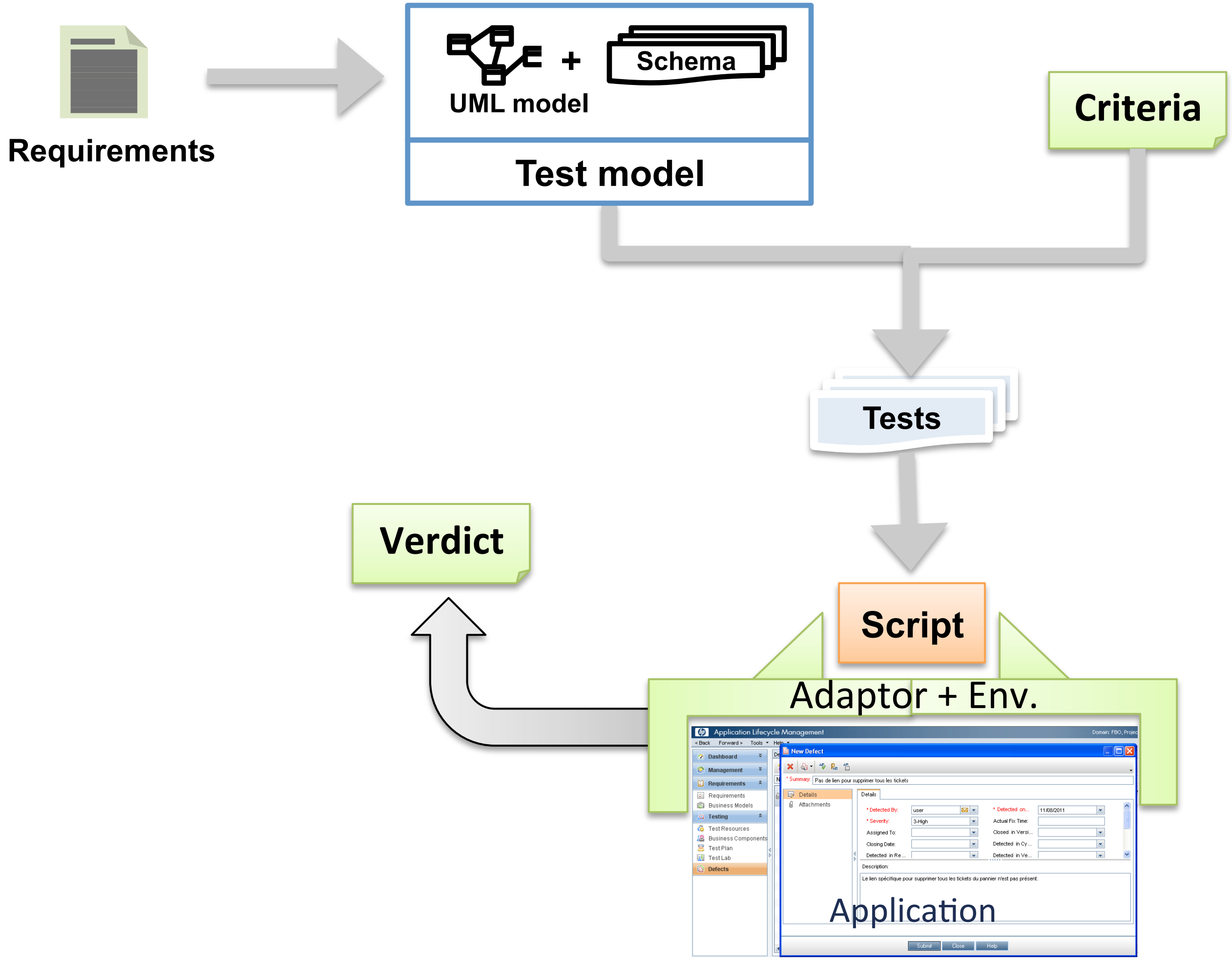
We export the test classification into a Test repository using the Smart Publisher



EvoTest Plugin

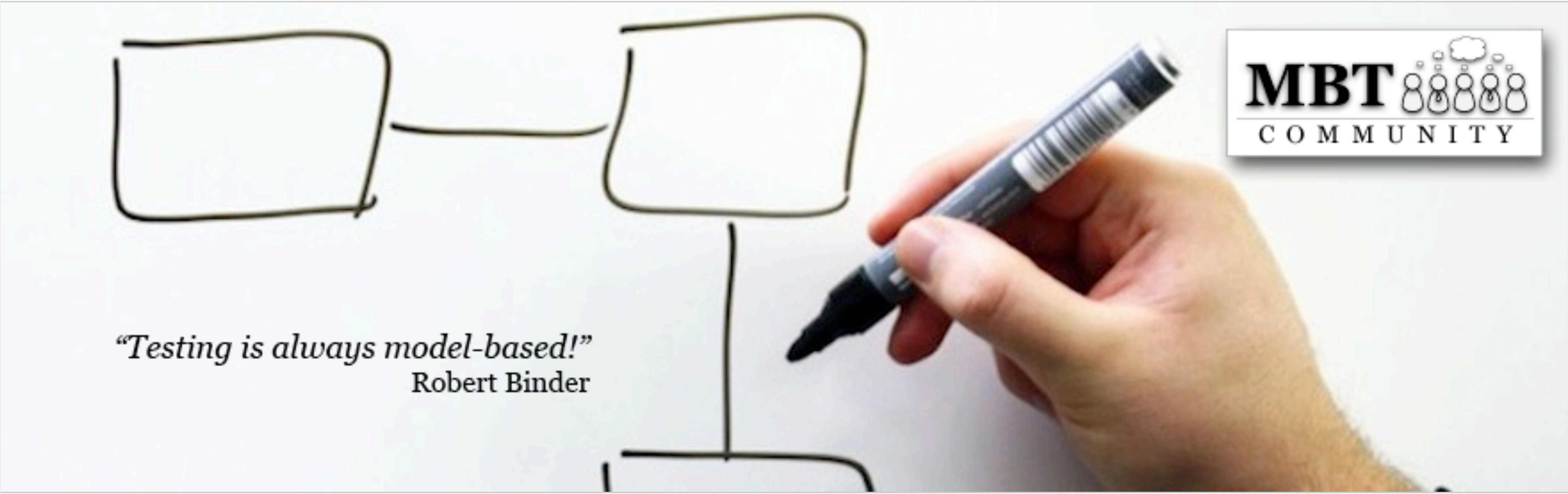


Synthesis



« They are many works and results but everything remains to be done »

Thanks for your attention



Source - http://model-based-testing.info