Model-Based Testing for Functional and Security Test

Fabrice BOUQUET

Summer School FOSAD
3 - 7th July 2012
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

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

Client Needs

Requirements

Design

Code

Acceptance Testing

System Testing

Integration Testing

Unit Testing

FOSAD 2012 - F. Bouquet
Testing tools

Requirements management tool

Model-based testing tool

Test management tool

Test gen. models

Test Repository

Test automation tool

Test scripts

Defect tracking tool

Defects

Configuration management and versioning platform
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
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...

Login: fbouquet
Password: ******
Verification: ******

Register Cancel
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 following 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...)
- ...
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

Diagram:
- Specification -> Test cases & data
- Oracle -> Software
- Test result -> Execution result

FOSAD 2012 - F. Bouquet
Kinds of Test [J. Tretmans]

Details level (step in development life cycle)
- system
- integration
- module
- unit

Accessibility level
- functional
- robustness
- performance
- ergonomic
- safety
- security

Type (which can be tested)
- white box
- black box

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
\[ x \in \{-1000, \ldots, 1000\} \]

Pre_{\text{op}}
\[ x \leq 100 \]

Post_{\text{op}}
\[
\text{IF } x \leq 0 \text{ THEN } y := \text{default}
\text{ELSE IF } x \leq 40
\text{THEN } y := \text{low}
\text{ELSE } y := \text{high}
\text{END END}
\]

Behavior classes

- **P1**: \( x \leq 0 \)
- **P2**: \( x > 0 \land x \leq 40 \)
- **P3**: \( x > 40 \land x \leq 100 \)

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 \((A \lor B)\)

- **Statement Coverage & Decision Coverage** \(\Rightarrow A \lor B\)
- **Decision/Condition Coverage** \(\Rightarrow A \not\leftrightarrow B\)
- **Full Predicate Coverage** \(\Rightarrow A \land \neg B \not\leftrightarrow \neg A \land B\)
- **Multiple Condition Coverage** \(\Rightarrow A \land B \not\leftrightarrow A \land \neg B \not\leftrightarrow \neg A \land B\)
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 ∈ 3 .. 15 ⇒ 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 ∈ {-7, 2, 3, 157, 200} ⇒ 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) <: typeof(C))
    • Object p such that : p != null && p != this && typeof(p) == typeof(c)
    • Object p such that : p != null && p != this && typeof(p) < : typeof(c)
    • Object p such that : p == p’ with p’ an other compatible object
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: $2^{32} \times 2^{32} = 2^{64} = 18\ 000\ 000\ 000\ 000\ 000\ 000$
- Preference parameter in MS Office

$2^{12} \times 3$ (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

<table>
<thead>
<tr>
<th>OS</th>
<th>Network</th>
<th>Printer</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>XP</td>
<td>IP</td>
<td>HP35</td>
<td>Word</td>
</tr>
<tr>
<td>Linux</td>
<td>Wifi</td>
<td>Canon900</td>
<td>Excel</td>
</tr>
<tr>
<td>Mac OS</td>
<td>Bluetooth</td>
<td>Canon-EX</td>
<td>Powerpoint</td>
</tr>
</tbody>
</table>

All combination: 81
All pairs: 9
Pairwise

9 test cases:
• Each combination of 2 values is tested:

<table>
<thead>
<tr>
<th>Case</th>
<th>OS</th>
<th>Network</th>
<th>Printer</th>
<th>App</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>XP</td>
<td>Bluetooth</td>
<td>Canon-EX</td>
<td>Powerpoint</td>
</tr>
<tr>
<td>2</td>
<td>Mac OS</td>
<td>IP</td>
<td>HP35</td>
<td>Powerpoint</td>
</tr>
<tr>
<td>3</td>
<td>Mac OS</td>
<td>Wifi</td>
<td>Canon-EX</td>
<td>Word</td>
</tr>
<tr>
<td>4</td>
<td>XP</td>
<td>IP</td>
<td>Canon900</td>
<td>Word</td>
</tr>
<tr>
<td>5</td>
<td>XP</td>
<td>Wifi</td>
<td>HP35</td>
<td>Excel</td>
</tr>
<tr>
<td>6</td>
<td>Linux</td>
<td>Bluetooth</td>
<td>HP35</td>
<td>Word</td>
</tr>
<tr>
<td>7</td>
<td>Linux</td>
<td>IP</td>
<td>Canon-EX</td>
<td>Excel</td>
</tr>
<tr>
<td>8</td>
<td>Mac OS</td>
<td>Bluetooth</td>
<td>Canon900</td>
<td>Excel</td>
</tr>
<tr>
<td>9</td>
<td>Linux</td>
<td>Wifi</td>
<td>Canon900</td>
<td>Powerpoint</td>
</tr>
</tbody>
</table>
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

- Requirements
- Modeling
- Test model
- Development
- Test Bench + Adaptation layer
- Test repository
- Scripts
- Scenario
- Application
- Manual test
- Execution
- Designed test cases
- Generation
- Publisher
- Test repository

UML model + Schema

FOSAD 2012 - F. Bouquet
Model-Based Testing – Taxonomy [Utting et al. 11]
**Model-Based Testing – Taxonomy**

### Model
- **Scope**: input-only / input-output
  - Untimed / Timed
- **Characteristics**:
  - Deterministic / Non-Det.
  - Discrete / Hybrid / Continuous
- **Pre-Post of Input Domains**
  - Transition-Based
  - History-Based
  - Functional
  - Operational
  - Stochastic
  - Data-Flow

### Test Generation
- **Test Selection Criteria**
  - Structural Model Coverage
    - Data Coverage
    - Requirements Coverage
    - Test Case Specifications
    - Random & Stochastic
    - Fault-Based
  - Random Generation
  - Search-Based Algorithms
  - Model-Checking
  - Symbolic Execution
  - Theorem Proving
  - Constraint Solving
- **Technology**
  - Random Generation
  - Search-Based Algorithms
  - Model-Checking
  - Symbolic Execution
  - Theorem Proving
  - Constraint Solving

### Test Execution
- **On/Off-line**: On-Line
  - Off-Line

### Application
- **Adaptor + Env.**
  - Script
  - Tests
  - Model
  - Requirements
  - Verdict
Model-Based Testing – Test adaptation

Three classical approaches to test case adaptation into executable test scripts

(a) Adaptation
(b) Transformation
(c) Mixed
Off-line architecture

1. Requirements
2. Modeling
3. UML model
4. Schema
5. Test model
6. Generation
7. Tests
8. Test Bench + Adaptation layer
9. Execution
10. Application
11. Publisher
12. Test repository

Off-line architecture flow:
- Requirements are modeled.
- UML models are generated along with schemas.
- Tests are generated from the generated models.
- The test bench and the adaptation layer are involved in the execution phase.
- The final tests are published in the test repository.
### Specification:
- “The **eCinema** is an application aiming at booking movie tickets.”

<table>
<thead>
<tr>
<th>Nb</th>
<th>Requirement</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ACCOUNT_MNGT/LOG</td>
<td>Log</td>
<td>The system must be able to manage the login process and allow only registered user to login.</td>
</tr>
<tr>
<td>2</td>
<td>ACCOUNT_MNGT/REGISTRATION</td>
<td>Registration</td>
<td>The system must be able to manage the user’s accounts.</td>
</tr>
<tr>
<td>3</td>
<td>BASKET_MNGT/BUY_TICKETS</td>
<td>BuyTickets</td>
<td>The system be able to allow users to buy available tickets.</td>
</tr>
<tr>
<td>4</td>
<td>BASKET_MNGT/DISPLAY_BASKET and DISPLAY_BASKET_PRICE</td>
<td>Display_Basket and Display_Basket_Price</td>
<td>The system must be able to display booked tickets and the total basket’s price for a connected user.</td>
</tr>
<tr>
<td>5</td>
<td>BASKET_MNGT/REMOVE_TICKETS</td>
<td>Remove_Tickets</td>
<td>The system must be able to allow deletion of all tickets for a given user.</td>
</tr>
<tr>
<td>6</td>
<td>CLOSE_APPLICATION</td>
<td>Close_Application</td>
<td>The system can be shut down</td>
</tr>
<tr>
<td>7</td>
<td>NAVIGATION</td>
<td>Navigation</td>
<td>In the system is possible to navigate from one state to another</td>
</tr>
</tbody>
</table>
Example – eCinema 2/8

Films list for 2012-09-03.

<table>
<thead>
<tr>
<th>Title</th>
<th>Time</th>
<th>Duration</th>
<th>Tickets</th>
<th>Buy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rambo 1</td>
<td>20h00</td>
<td>120</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Rambo 2</td>
<td>20h00</td>
<td>110</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Rocky 7</td>
<td>20h00</td>
<td>210</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
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
OCL constraints capture business rules, and are used to evaluate transitions.

State machine diagrams capture the behavior of the system.
Example – eCinema Cover 6/8
- 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

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

**Version 2**
CREATED ON 16/03/2011 16:55:20 BY admin

**Summary**
**Actual state : Re-executed**

**New 2011-03-16 16:45:25**

**BODY**

<table>
<thead>
<tr>
<th>No</th>
<th>Description</th>
<th>But</th>
<th>Exigence</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fill the name field with : in_userName</td>
<td>LOG_Success</td>
<td>ACCOUNT_MNGT/LOG</td>
<td>login</td>
</tr>
<tr>
<td></td>
<td>Fill the password field with : in_userPassword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Click the <code>login</code> link With:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• in_userName = <code>USER</code> (e.g. <code>ERIC</code>)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• in_userPassword = <code>PWD</code> (e.g. <code>ETO</code>)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Click the <code>unregister</code> link</td>
<td>REG_Unregister</td>
<td>ACCOUNT_MNGT/REGISTRATION</td>
<td>unregister</td>
</tr>
<tr>
<td></td>
<td>Steps Expected Results</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>No Description</td>
<td>Operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fill the name field with : in_userName</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
On-line architecture (ModelJUnit, GraphWalker, TGV...)

- Requirements
  - Modeling

- Test model
  - UML model
  - Schema

- Publisher

- Test repository

- Generation + Execution

- Test Bench + Adaptation layer

- Application
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 (Smarttesting)
- 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 Techniques

- **Network Scanning**
  - Network vulnerability scanner

- **SAST – Static Application Security Testing**
  - Source code analyser
    - Bytecode analyser
    - Binary code scanner
    - Database fragility 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

---

FOSAD 2012 - F. Bouquet
Evolution des menaces (1): Les récentes attaques amènent à se poser des questions sur l'efficacité des mécanismes de sécurité traditionnels.
Security Testing Overview

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

- Textual Requirements
- Use Cases
- Business Processes

Risk analysis & assessment influences levels of modeling

Modeling Phase

Model-driven testing process

Test generation Phase

Test execution Phase

Influences test selection criteria

Influences level of test automation
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 2006I], [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...
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
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 \textit{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 : \{Alice, N_A\}^{PK_B}
Bob \rightarrow Alice : (N_A, N_B, Bob)^{PK_A}
Alice \rightarrow Bob : \{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 \(\land\) RCV(start) =|>
   State' := 2 \(\land\) Na' := new() \(\land\) SND({Na'.A}_Kb)

2. State = 2 \(\land\) RCV({Na.Nb'.B}_Ka) =|>
   State' := 4 \(\land\) SND({Nb'}_Kb)
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
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 Privacy attacks due to the XOR operator properties of commutativity, idempotence, and associativity

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

Alice $\rightarrow$ Bob : \( M \ xor \ K_A \)
Bob $\rightarrow$ Alice : \( (M \ xor \ K_A) \ xor \ K_B \)
Alice $\rightarrow$ Bob : \( M \ xor \ K_B \)

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

\[
M \ xor \ K_A \ xor \ (M \ xor \ K_A) \ xor \ K_B \ xor \ M \ xor \ K_B \ \equiv \ M
\]

**In HLPSL code:**

1. State = 1 $\lor$ RCV({Na’ . A}_Kb) = $\Rightarrow$
   State’ := 3 $\lor$ Nb’ := new() $\lor$ SND({Na’ . Nb’}_Ka)
2. State = 3 $\lor$ RCV({Nb}_Kb) = $\Rightarrow$
   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 : \( M \oplus K_A \)
Bob → Alice : \( (M \oplus K_A) \oplus K_B \)
Alice → Bob : \( M \oplus K_B \)

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

\[
M \oplus K_A \oplus (M \oplus K_A) \oplus K_B \oplus M \oplus K_B \Leftrightarrow M
\]

**In HLPSL code:**

1. \( \text{State} = 1 \) \( / \backslash \text{RCV}\{\text{Na’.A}\}_Kb\) =|>
   \[ \text{State’} := 3 \] \( / \backslash \text{Nb’} := \text{new()} \) \( / \backslash \text{SND}\{\text{Na’.Nb’}\}_Ka\)
3. \( \text{State} = 3 \) \( / \backslash \text{RCV}\{\text{Nb}\}_Kb\) =|>
   \[ \text{State’} := 5 \]
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 \( \rightarrow \) Bob : \( M \oplus K_A \)
Bob \( \rightarrow \) Alice : \( (M \oplus K_A) \oplus K_B \)
Alice \( \rightarrow \) Bob : \( M \oplus K_B \)

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

\[ M \oplus K_A \oplus (M \oplus K_A) \oplus K_B \oplus M \oplus K_B \leftrightarrow M \]

**In HLPSL code:**

1. State = 1 \( /\ \) RCV(\( \text{xor} \{\text{Na'.A}\},K_b\)) =\( \rightarrow \)
   State' := 3 \( /\ \) Nb' := new() \( /\ \) SND(\( \text{xor} \{\text{Na'.Nb'}\},K_a\))
3. State = 3 \( /\ \) RCV(\( \text{xor} \{\text{Nb}\},K_b\)) =\( \rightarrow \)
   State' := 5
Homomorphism

Principe of homomorphism:
• Encryption function that satisfies property: \( \{X \cdot Y\}_k \Leftrightarrow \{X\}_k \cdot \{Y\}_k \)
• Block-by-block encryption
• Used in the Electronic Code Book

NSPK attack due to homomorphism:
Consider the corrected version of NSPK

Alice \( \rightarrow \) Bob : \{Alice, N_A\}_{PK_B}
Bob \( \rightarrow \) Alice : \{(N_A, N_B, Bob)\}_{PK_A}
Alice \( \rightarrow \) Bob : \{N_B\}_{PK_B}

It can become unsafe with homomorphism in the encryption scheme:

Alice \( \rightarrow \) Bob : \{Alice, N_A\}_{PK_B}
Bob \( \rightarrow \) Charlie : \{Alice, N_A\}_{PK_C}
Charlie \( \rightarrow \) Bob : \{(N_A, N_C, Charlie)\}_{PK_A} \textbf{(interception)}
Bob \( \rightarrow \) Alice : \{(N_A, N_C, Bob)\}_{PK_A}
Alice \( \rightarrow \) Bob : \{N_C\}_{PK_B}
Bob \( \rightarrow \) Charlie : \{N_C\}_{PK_C}
Homomorphism

**Principe of homomorphism:**
- Encryption function that satisfies property: $\{X \cdot Y\}_k \Leftrightarrow \{X\}_k \cdot \{Y\}_k$
- Block-by-block encryption
- Used in the Electronic Code Book

**NSPK attack due to homomorphism:**
Consider the corrected version of NSPK

- Alice $\rightarrow$ Bob : $\{\text{Alice}, N_A\}_{PK_B}$
- Bob $\rightarrow$ Alice : $(N_A, N_B, \text{Bob})_{PK_A}$
- Alice $\rightarrow$ Bob : $\{N_B\}_{PK_B}$

**In HLPSL code:**
1. State $= 1$ \(\rightarrow\) RCV($\{\text{Na’.A}\}_Kb$) = $|> $
   State' := 3 \(\rightarrow\) Nb' := new() \(\rightarrow\) SND($\{\text{Na’.Nb’}\}_Ka$)
2. State $= 3$ \(\rightarrow\) RCV($\{Nb\}_Kb$) = $|> $
   State' := 5
Homomorphism

**Principe of homomorphism:**
- Encryption function that satisfies property: $\{X \cdot Y\}_K \Leftrightarrow \{X\}_K \cdot \{Y\}_K$
- Block-by-block encryption
- Used in the Electronic Code Book

**NSPK attack due to homomorphism:**
Consider the corrected version of NSPK

Alice $\rightarrow$ Bob : $\{\text{Alice}, N_A\}_\text{PK}_B$
Bob $\rightarrow$ Alice : $(N_A, N_B, \text{Bob})_\text{PK}_A$
Alice $\rightarrow$ Bob : $\{N_B\}_\text{PK}_B$

**In HLPSL code:**
1. State = 1 /\ RCV(\{Na'.A\}_Kb) =|>
   State' := 3 /\ Nb' := new() /\ SND(\{Na'.Nb'\}_Ka)
2. State = 3 /\ RCV(\{Nb\}_Kb) =|>
   State' := 5
Homomorphism

**Principe of homomorphism:**
- Encryption function that satisfies property: \( \{ X \cdot Y \}_K \iff \{ X \}_K \cdot \{ Y \}_K \)
- Block-by-block encryption
- Used in the Electronic Code Book

**NSPK attack due to homomorphism:**
Consider the corrected version of NSPK

Alice \( \rightarrow \) Bob : \( \{ \text{Alice}, N_A \}_PK_B \)

Bob \( \rightarrow \) Alice : \( \{ N_A, N_B, \text{Bob} \}_PK_A \)

Alice \( \rightarrow \) Bob : \( \{ N_B \}_PK_B \)

**In HLPSL code:**
1. State = 1 \( \land \) RCV(\( \{ \text{Na'} \}_Kb.\{A\}_Kb \)) =|>
   State' := 3 \( \land \) Nb' := new() \( \land \) SND(\( \{ \text{Na'} \}_Kb.\{Nb' \}_Ka \))

3. State = 3 \( \land \) 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

**Principle 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:**
```hlpsl
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

Principle 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.

\[
\begin{align*}
    \text{Alice} & \rightarrow \text{Bob} : \{\text{Alice}, N_A\}^{PK_B} \\
    \text{Bob} & \rightarrow \text{Alice} : (N_A, N_B, X)^{PK_A} \\
    \text{Alice} & \rightarrow \text{Bob} : \{N_B\}^{PK_B}
\end{align*}
\]

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)
```

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 $\to$ Bob : $\{\text{Alice}, N_A\}^{PK_B}$
Bob $\to$ Alice : $(N_A, N_B, X)^{PK_A}$
Alice $\to$ 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 $\land$ RCV(start) $=>$
            State' := 2 $\land$ Na' := new() $\land$ SND($\{Na'.A\}_{Kb}$)
        2. State = 2 $\land$ RCV($\{Na.Nb'.B\}_{Ka}$) $=>$
            State' := 4 $\land$ SND($\{Nb'\}_{Kb}$)
Substitution

**Principle 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 $\rightarrow$ Bob : $\{\text{Alice, } N_A\}_{PK_B}$
Bob $\rightarrow$ Alice : $(N_A, N_B, X)_{PK_A}$
Alice $\rightarrow$ 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 $\land$ RCV(start) =|$>
   State' := 2 $\land$ Na' := new() $\land$ SND($\{N_a'.A\}_{K_b}$)

2. State = 2 $\land$ RCV($\{N_a.N_b'.X\}_{K_a}$) =|$>
   State' := 4 $\land$ SND($\{N_b'\}_{K_b}$)
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'.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'.Kab.Na'.Nb.A) =>
       State' := 2 \ SND(Kab.Na')
end role
Permutation

**Principle 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
  \[ \Rightarrow \] for the experiments: move last block to first place

**In HLPSL code:**

```hlpsl
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
```

FOSAD 2012 - F. Bouquet
Permutation

Principle 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

**Principle 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
  \[ \Rightarrow \text{for the experiments: move last block to first place} \]

**In HLPSL code:**

```hlpsl
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 \land RCV(start) =|>
       State' := 2 \land Na' := new() \land SND({A.Na'}_Kb)
    2. State = 2 \land RCV({Na.Nb'.B}_Ka) =|>
       State' := 4 \land 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

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

Safe: no attack — Unsafe: attack trace found — Incoherent: unexecutable protocol
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);
From test purposes to test case specs

Security Test Objective

Test Purpose #1

Security Test Engineer

Test Purpose #n
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:

  - **globally**
  - **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 \( \text{evt} \)
  - always \( \text{pred} \)
  - \( \text{evt1} [\text{directly}] \) follows \( \text{evt2} \)
  - \( \text{evt1} [\text{directly}] \) precedes \( \text{evt2} \)
  - eventually \( [(\text{exactly}/\text{at least}/\text{at most}) k \text{ times}]\text{evt} \)

- Events: all methods of UML/OCL model
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:
- \( \Sigma \{\text{login}\}^*.\text{login.}(\Sigma \{\text{logout, buyTicket}\}|\text{buyTicket}).\text{logout} \)

- 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."

Test Purposes definition and information

Keywords

- listOperationsLogin
- listBehaviorsLogSuccess

Keyword definition

Type: List of Behaviors

behavior_with_tags {REQ:ACCOUNT_MNGT/LOG, AIM:LOG_Success} or behavior_with_tags {REQ:ACCOUNT_MNGT/LOG, AIM:LOG_Logout}

Test purposes

Test Purpose

Test Purpose definition

Tags: @REQ: AFTER_LOGOUT

for_each behavior $b$ from listBehaviorsLogSuccess,
use $b$.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 $b$.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 $b$

Test Purpose defined correctly.
"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

TC₁

TCn

Unreachable

Test cases
Unfolding of one test purpose

Test Purpose definition

Tags  @REQ: AFTER_LOGOUT

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

Test Case Specifications

```plaintext
use sut.login($User,_) any_number_of_time
   to_activate behavior_with_tags {REQ:ACCOUNT_MNTG/LOG,AIM:LOG_Success} then
use logoutany_number_of_time
   to_activate behavior_with_tags {REQ:ACCOUNT_MNTG/LOG,AIM:LOG_Logout} then
use sut.login($User,_) any_number_of_time
   to_activate behavior_with_tags {REQ:ACCOUNT_MNTG/LOG,AIM:LOG_Success} then
Use any_operation any_number_of_time to_activate behavior_with_tags {REQ:ACCOUNT_MNTG/LOG,AIM:LOG_Logout}
```

```plaintext
use sut.login($User,_) any_number_of_time
   to_activate behavior_with_tags {REQ:ACCOUNT_MNTG/LOG,AIM:LOG_Success} then
use logoutany_number_of_time
   to_activate behavior_with_tags {REQ:ACCOUNT_MNTG/LOG,AIM:LOG_Logout} then
use sut.login($User,_) any_number_of_time
   to_activate behavior_with_tags {REQ:ACCOUNT_MNTG/LOG,AIM:LOG_Success} then
Use any_operation any_number_of_time to_activate behavior_with_tags {RAQ:ACCOUNT_MNTG/LOG,AIM:LOG_Success} then
```
Test generated
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
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

- 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


- 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.

- 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
CURRENT TYPICAL MBT PROCESS

Version 1

- Requirements v1
- Modeling
- Test model v1
- SBTG
- Tests v1

Version 2

- Requirements v2
- Modeling
- Test model v2
- SBTG
- Tests v2

Version n

- Requirements n
- Modeling
- Test model n
- SBTG
- Tests n

Selective Test Generation Method

Schema-Based Test Generation

Publisher

Smart Publisher

Test repository

FOSAD 2012 - F. Bouquet
SETGAM Component Overview

Impact Analyser

- Dependency graph computation
- State machine comparison
- Changes in dependency graphs evaluation

Test Classifier

- Tests classification
- Animation
- Generation

Tests

Test model v1

Test model v2

FOSAD 2012 - F. Bouquet
• For the considered MBT approach we generate tests to cover requirements expressed in the initial model.

• In order to test the evolved software, we proceed by detecting affected requirements

• we perform impact analysis based on dependence graphs for both versions

• we compare the models to obtain new, deleted and modified requirements

• we define actions in form of rules to select and classify tests.

For the
considered MBT
approach
we generate
tests to cover
requirements expressed in the initial model.
Classification of test cases

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

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

- We generate tests for all new uncovered requirements

- To conclude about the final status: Updated, Re-executed or Failed, we animate Failed tests on the model.
Test STATUS Life Cycle / Evolution

Status extends [Leung al.89]

Modified
- Updated
- Adapted

Reusable
- Re-executed
- Unimpacted

Obsolete
- Outdated
- Failed

New

Removed

FOSAD 2012 - F. Bouquet
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)
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

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

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

Deleted data dep.
Impacted transition
No Statecharts for (data/control) dependencies computation.

- Based on def/use of the variables in operation behavior \((@\text{PRE},@\text{POST},\Sigma\text{TAGS})\)
  - Behavioral Data dependence: ”A behavior 1 is data-dependent from another behavior 2 w.r.t. variable \(v_1\) if and only if \(v_1\) 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 \((\text{behavior1.}@\text{POST} \&\& \text{behavior2}@\text{PRE})\) is consistent.”
## Running example – eCinema

### Set of tests

<table>
<thead>
<tr>
<th>Test</th>
<th>Covered behaviors</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>login-sucess, buy-success, delete-ticket</td>
</tr>
<tr>
<td>T2</td>
<td>login-sucess, buy-success, logout-success</td>
</tr>
</tbody>
</table>

### Running example – eCinema

#### FOSAD 2012 - F. Bouquet
Running example – eCinema

<table>
<thead>
<tr>
<th>Test</th>
<th>Covered behaviors</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>login-success, buy-success, delete-ticket</td>
<td>Outdated -&gt; Adapted</td>
</tr>
<tr>
<td>T2</td>
<td>login-success, buy-success, logout-success</td>
<td>Outdated</td>
</tr>
</tbody>
</table>

Set of tests

Classification

```
PRE-CONDITION
currentState = STATE::Welcome
not(current_user.oclIsUndefined())

POST-CONDITION
current_user.oclIsUndefined()
message = MSG::BYE

TAGS
ACCOUNT_MNGT/LOG_LOG_Logout

PRE-CONDITION
currentState = STATE::Welcome
not(current_user.oclIsUndefined())

POST-CONDITION
current_user.all_tickets_in_basket->includes(unallocated_ticket) and
message = MSG::NONE

TAGS
ACCOUNT_MNGT/BUY_TICKETS
BUY_Success

PRE-CONDITION
currentState = STATE::Welcome
not(current_user.oclIsUndefined())

POST-CONDITION
current_user.all_tickets_in_basket->any(t.movie
= in_movie).owner.oclIsUndefined() and
message = MSG::NONE

TAGS
ACCOUNT_MNGT/BUY_TICKETS
BUY_Success

PRE-CONDITION
currentState = STATE::Welcome
not(current_user.oclIsUndefined())

POST-CONDITION
current_user.all_tickets_in_basket->any(t.movie
= in_movie).owner.oclIsUndefined() and
message = MSG::NONE

TAGS
ACCOUNT_MNGT/BUY_TICKETS
BUY_Success

PRE-CONDITION
currentState = STATE::Welcome
not(current_user.oclIsUndefined())

POST-CONDITION
current_user.all_tickets_in_basket->not(unallocated_ticket) and
message = MSG::NONE

TAGS
ACCOUNT_MNGT/BUY_TICKETS
BUY_Success
```
SeTgaM for security requirements

Schema and model evolution:

UML model V1

TCS

UML model V2

TCS

Test Purpose Comparison

Unchanged TCS

Deleted TCS

New TCS

SeTgaM_security classification

New

Re-executed

Updated

Failed

Adapted

Outdated

Unimpacted

Removed

Test Status
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).

```plaintext
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}
```

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

Outdated ➤ Adapted
Ensure that the evolution did not impact unmodified parts.
Ensure that the evolution changed the system behaviour.
Ensure that the system novelities behave correctly.
Ensure maximal test history and contains tests from the Stagnation test suite of previous version.
We export the test classification into a Test repository using the Smart Publisher.
Synthesis

« They are many works and results but everything remains to be done »
Thanks for your attention

“Testing is always model-based!”
Robert Binder

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