Model-Based Testing From UML Models

Bruno Legeard, Leirios, France,
Model-Based Testing from UML models

Bruno Legeard – LEIRIOS Technologies

Manchester – Thursday, 6th December 2006
1. The challenge of functional validation
2. Model-Based Testing (MBT) process
3. Main activities of MBT with UML
4. A case study: eTheater web application
5. Benefits of MBT with UML
A solution to automate **functional** test design

- Does my application respect the requirements?
To specify the most relevant tests
To keep the skyrocketing number of combinations under control
To optimize the functional coverage
To limit the tester’s subjectivity
To efficiently manage the functional changes

Automate test design with Model-Based Testing
Model-based testing process

Use case model

Analysis model

Design model

Implementation model

Test Model

Executable scripts

Adaptation layer

Validation Process

Development Process

Tests generation

<<realize>>

<<refine>>

<<refine>>

<<implement>>

<<implement>>

<<use>>

<<use>>
Model-Based Testing Process

AUTOMATED TEST DESIGN

Business Analyst → Requirements
Analyst → UML Model
Test Manager → Test Scripts
QA Tester → ...

Modeling Tools

LEIRIOS Test Generator

Test Management & Execution Tools
LEIRIOS Test Generator™

Modeling Tools

Automate the validation testing process

Requirements

MBT Tools

Test Management and execution tools

LEIRIOS Test Generator™
Main activities of MBT with UML

1. Refine (or develop) the functional model for test generation

2. Configure automated test generation

3. Adapt generated test cases in executable test scripts
LTG supports a sub-part of UML2.0 for automated test generation purpose:

- **Class diagrams** – static data and operation definition
- **State Machines with OCL** – Dynamic behavior
- **Object diagrams** – Initial state

The test model is developed by refinement of the functional model:
- precise enough
- at the good level of abstraction depending of the test objectives
1. The validation engineer defines test campaigns:
   - test objectives (what should be tested?)
   - coverage criteria (how thoroughly should it be tested?)

2. LTG computes precise targets for which tests have to be generated:
   
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
</tr>
</tbody>
</table>

3. LTG computes the test cases for each target and display the resulting coverage:
   - target coverage
   - requirements coverage
Test Selection Criteria

• 3 families of model coverage criteria
  – Transition coverage (All-Transitions)
  – Decision coverage (Condition/Decision coverage)
  – Data coverage (One_Value, All_Values, Boundary Values)
Test cases

public class TestScript{
    // ${test.getName()}
    public void ${test.name}(){
        // preamble
        ${preamble.getMessages()}
        // body
        ${body.getMessages()}
        // observation
        ${observation.getMessages()}
    }
}

Tree structure patterns
Test script patterns

Main features:
- Test cases are UML Testing Profile compliant
- Tree structure and patterns are very flexible
- Java/ JUnit
- Mercury TD / QTP
- Compuware QARun
- Any language

Generate executable scripts

Executable test scripts
LTG - Test generation process

Requirements

1) Model
   LTG Model Animator

2) Check
   LTG Tests Generator

3) Generate
   LTG Adapter

4) Export
   Test Ex. Env. + Adaptation layer

5) Execute and analyse
   Test results

Test Targets and Test Cases

Executable Test Scripts

UML Model
- Class diagram
- Object diagram
- State Machines
- OCL
Case study – eTheater application

A web-based system for buying movie tickets

1. Modeling in UML
2. Test generation with LEIRIOS Test Generator
3. Adapt generated test cases in executable test scripts
The e-Theater application allows movie-goers to buy movie tickets online, before they go to the theater.
Use cases – Buy Tickets

<table>
<thead>
<tr>
<th>Use Case:</th>
<th>BuyTickets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope:</td>
<td>e-Theater</td>
</tr>
<tr>
<td>Primary Actor:</td>
<td>Customer</td>
</tr>
<tr>
<td>Secondary Actor:</td>
<td>Bank</td>
</tr>
<tr>
<td>Goal:</td>
<td>The customer wants to buy one or more tickets for a given session of a movie.</td>
</tr>
<tr>
<td>Precondition:</td>
<td>The customer is an existing customer of the e-Theater system and has registered a valid payment card.</td>
</tr>
</tbody>
</table>

**Main Success Scenario**

1. Customer enters their username and password.
2. Customer is shown a welcome page, which shows all the tickets they have previously bought (and not yet used) and offers two links: ‘Buy Tickets’ and ‘Register Payment Card’.
3. Customer clicks the ‘Buy Tickets’ link and is taken to the buying page, which displays information about the films that are showing, the date and time of each session of those films and the number of available tickets. It also displays the current contents of their shopping basket at the top of the page.
4. Customer selects the desired number of tickets for a particular session of a particular film. They may repeat this for several different films, adding more tickets to their shopping basket each time.
5. Customer confirms their selection.
6. A payment is made using their credit card, via the bank.
7. A receipt is displayed showing payment details, film name and the session time.
8. Customer logs out from the e-Theater system.
Use cases – Buy Tickets (2)

Extensions

1a: If the customer username or password is wrong then an error message is displayed and they remain in the same login page.

4a: In the buying page, the customer can also delete tickets from their shopping basket, if they change their mind.

6a: If there are not enough available tickets for a given session then an error message is displayed and the user is returned to the buying page.

6b: If the payment fails, an error message is displayed and the customer is returned to the welcome page. From there they can choose to register a different payment card, or go back into the buying page to delete some tickets.
eTheater Model – Class diagram

Class diagrams give static data and operation signatures.
Object diagram defines the initial state for test generation
State machine and OCL definitions define the expected behavior.
Configure test generation

- **Model coverage criteria**
  - Transition coverage (all transitions are activated)
  - Condition/Decision coverage (each condition in each decision)
  - One value per equivalence class, but All values for Card type (Visa, Mastercard, Amex)

→ 22 test cases have been generated
## Test generation results

<table>
<thead>
<tr>
<th>Test Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
</tr>
<tr>
<td>T2</td>
</tr>
<tr>
<td>T3</td>
</tr>
<tr>
<td>T4</td>
</tr>
<tr>
<td>T5</td>
</tr>
<tr>
<td>T6</td>
</tr>
</tbody>
</table>
eTheater Model – State machine
Three classical approaches to test case adaptation into executable test scripts
ETheater – Executable scripts generation

• Use the mixed approach
• Generate Ruby scripts
• Adaptation layer to execute operation calls

```ruby
# Generated e-Theater test T12
sut.start();
sut.newCust(c1, AMANDINE);
sut.logout();
sut.disconnect()

# Ruby code that implements e-Theater test T12
require 'test/unit'
require File.dirname(__FILE__) + '/../..../test_helper'

class ETheater_T12 < Test::Unit::TestCase
  fixtures :customers

  def setup
    @controller = CustomersController.new
    @request = ActionController::TestRequest.new
    @response = ActionController::TestResponse.new
  end

  def test_T12
    # start();
    post :register
    # newCust(c1, AMANDINE);
    customer = @customers['AMANDINE']
    post :save_new_customer, :customer =>
    { :login => customer['login'],
      :password => customer['password'] }
    assert_response :redirect
    assert_redirected_to :action => "list"
    # logout();
    post :logout
    assert_redirected_to :action => "login"
    # disconnect()
  end
end
```
Benefits of MBT with UML

- **Return on Investment**
  - Generate tests and scripts from the model
  - Reuse existing modeling artifacts from requirements analysis

- **Improved test quality**
  - Automated generation from the model using coverage criteria gives a strong rationale for you test

- **Tester keep the control**
  - The tester drive the test generation using model coverage criteria

- **Much easier test maintenance**
  - Evolving requirements: update the model and re-generate

- **Generate test earlier**
  - Generating test from analysis model contributes to reinforce product requirements