A Test Generation Solution to Automate Software Testing

F. Bouquet
University of Franche-Comté
LIFC, 16 route de Gray
25030 Besançon, France
bouquet@lifc.univ-fcomte.fr

C. Grandpierre, B. Legeard, F. Peureux
LEIRIOS
18 rue Alain Savary
25000 Besançon, France
{grandpierre, legeard, peureux}@leirios.com

ABSTRACT

This paper describes the LEIRIOS Smart Testing™ approach to the functional validation of an application by way of an example that illustrates the process from start to end: from use cases and functional requirements to the publication of generated tests in a test repository and automatic execution of scripts with a test execution robot. For this paper this testing solution is applied on particular software to take part in a specific case study: *how to automate the testing of UML/MDA platform StarUML*.

Categories and Subject Descriptors

D.2.1 [Requirements/Specifications]
D.2.5 [Testing and Debugging]

General Terms

Reliability, Verification.

Keywords

Model-Based Testing, UML, OLC, software testing

1. INTRODUCTION

The growing complexity of software applications and the necessity of retaining an overall view of software development entail the implementation of high-performance application validation strategies. Functional testing represents a keystone for risk management, quality management and time-to-market constraints. Model-Based Testing (MBT [1]) is a solution that implements functional tests based on business requirements and test design automation (test cases, executable scripts, test plan coverage) while guaranteeing functional coverage completeness. This solution encompasses the processes, tools and best practices to improve conformity, traceability and risk management.

Many approaches and/or tools implement model-based testing with their own properties. The taxonomy [2] classifies some of them according to different dimensions (model paradigm, test selection criteria…). These dimensions are also used to classify some MBT tools. TorX [3], TGV [4] and AutoLink [5] generate automated test case using on-the-fly state space exploration techniques from Input-Output Labelled Transition System models. JUMBL [6] and Matelo [7] are model-based statistical testing tools based on Markov chain usage models. AETG [8] is dedicated to the automated generation of test inputs using n-way search algorithms from a static, environment model.

The LEIRIOS Smart Testing™ solution is a tooled testing approach to generate and manage functional tests from UML /OCL models. [9] gives the UML/OCL subset available to define behavioral models used in this testing solution. The main principle of the LEIRIOS Smart Testing™ solution is:

- Modeling of a UML/OCL behavioral model. The model is an abstraction of the system under test (SUT),
- Automated generation of test cases from the UML behavioral model,
- Publishing of the generated tests into well-known test repositories,
- Generation of executable scripts to automate the test execution on the SUT.

The main objective of this paper is to demonstrate how LEIRIOS Smart Testing™ can be applied to test a software application like StarUML. The carrying out of this solution makes it possible to highlight the problematic aspects lied to the test generation and automation of such application.

This paper is organized as follows. Section 2 explains step by step the solution applied on the defined case study. Section 3 proposes discussion points lied to the automation of software testing. Finally section 4 concludes the paper.

2. CASE STUDY

In this section, we develop the LEIRIOS Smart Testing™ solution on the specific case study. The objective is to demonstrate how the solution can be use to automate the testing of any software application. To reduce the scope, we propose to study the testing of a particular functionality of the StarUML framework: the *StarUML project management (SPM).*

2.1 Scope definition

Functional testing of a software application requires that the system under test is checked according to predetermined and expected behavior under specific circumstances. This objective thus raises the question of defining functionalities in an explicit and detailed manner so that the testing team can develop a consistent and measurable test plan (measurable especially in terms of traceability and functional coverage metrics).

The StarUML framework provides numerous features to develop UML/MDA platform. Among these ones, we propose to automate the testing of the project management inside StarUML.

Project management includes all the functionalities tied to creation, editing, and opening of StarUML projects.

The current practices in the field of functional requirement definition are based on defining use cases and on characterizing functional requirements that are defined, maintained and sometimes traced throughout the lifecycle of the application:

- The use case approach is a method of capturing and describing functional requirements of a system. A use case contains one or more scenarios that define the way in which the system must interact with users (called actors) to achieve a goal or to cover a specific function of the application. In a use case, the actor can either be human or another system.
- Requirement management helps reference "atomic" functional requirements that are testable and which are maintained, i.e. updated, when the application is modified.

In practice, these two approaches complement each other; use cases provide an overall vision of representative user scenarios for the application, and detailed requirements facilitate the identification of key points in the software expected behavior. Of course, these two elements (use cases and functional requirements) are some of the elements used for analysis along with the explanation of terminology, business entities, and the definition of the roles of the users of the application.

In this case study we defined the testing scope via the following use cases.

All the use cases define scenarios for the same actor: the StarUML platform user. The underlined actions refer to defined use cases.

Use Case 1: *Create a project*	
Precondition	-
Postcondition	A new project was created
Called by the use cases	-
Nominal scenario	1. The actor creates a new project 2. The actor can edit the project 3. The actor can save the project 4. The actor can close the project

Use Case 2: *Edit a project*	
Precondition	The project is open
Postcondition	-
Called by the use cases	Create a project, Open a project
Nominal scenario	1. The actor changes the project title 2. The actor can save the project 3. The actor can close the project

Use Case 3: *Open a project*	
Precondition	The project exists
Postcondition	The project is open
Called by the use cases	-
Nominal scenario	1. The actor opens a valid StarUML project from a pathname 2. The actor can save the project 3. The actor can close the project

Use Case 4: *Save a project*	
Precondition	The project is open
Postcondition	The project is saved
Called by the use cases	Create a project, Open a project
Nominal scenario	1. The actor save the project as a valid pathname 2. The actor can close the project

Use Case 5: *Close a project*	
Precondition	The project is open
Postcondition	The project is closed
Called by the use cases	Create a project, Open a project
Nominal scenario	1. The actor close the project

Notice that these use cases don't give the different error cases which would be tied to these actions (saving as an existing pathname, opening a no project file…)

These use cases make it possible to make explicit the scope of functional testing of our example. In addition some functional requirements can complete the functional definition of the validation campaign. Table 1 provides some informal requirements used and traced throughout the testing process.

Table 1. Informal requirements for case study

Identifier	**Requirement definition**
PRJ_CREATED	A new project is created
MODIF_OK	The project modifications are effective
PRJ_OPEN	The project opening successes
NOT_PROJECT	The project opening fails: file is not a project
PRJ_SAVED	The project saving successes
FILE_ALRDY_EXISTS	The project saving fails: file already exists
PRJ_CLOSED	The project is closed

These requirements identifiers are directly used for the test modeling step then throughout the validation process for traceability.

2.2 Modeling for Test Generation

2.2.1 *Test material*

LEIRIOS Smart Testing™ test generation is based on exploration of a behavioral UML model. Concretely three UML diagrams are available to design such test model.

UML *class diagram* is the static view of the model. It describes the abstract objects of the system and their dependencies. The available UML elements are *classes*, *associations*, *enumerations*, class *attributes* and *operations*.

UML *object diagram* models the initial state of the SUT. *Objects* and *links* compose such diagram.
UML *state-machine* is used to model the dynamic SUT behaviors as a finite state transition system. State-machines may contain simple and composite states and any transition with event/guard/action format.

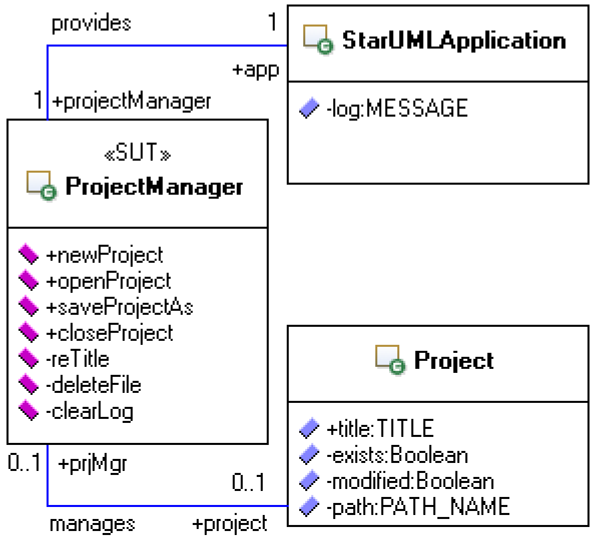
Object Constraint Language (OCL [10]) is used to formally express the SUT behaviors. OCL is used in class diagrams, to formalize the expected behavior of class operations. It is also used within state-machines to formalize transitions between states – the guards and the effects of transitions are expressed as OCL predicates.

This test material provides the elements to design the behavioral model for testing SPM system. This test model is composed of the three diagrams above-mentioned. The following subsection presents these UML elements.

2.2.2 *The* SPM *model*

2.2.2.1 *Class diagram*

Figure 1 presents the class diagram of the SPM model. It depicts the different objects of the SUT and the dependencies between them.
We have three object types in SPM. The class under test is the project manager. This is provided by a StarUML application and manages a project at once.



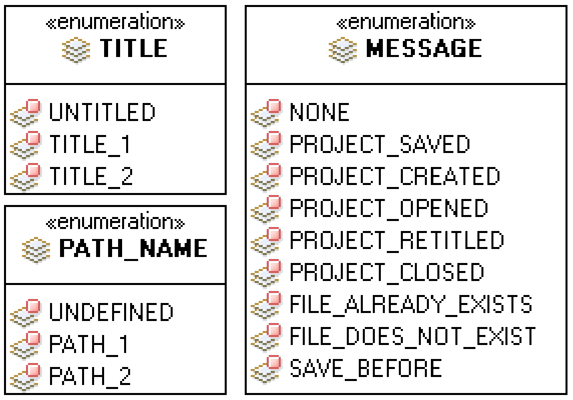


Figure 1. Class diagram for SPM model

The class attribute *Project::path* abstracts the path of a saved project. A new project has the default value UNDEFINED.

The others class attributes and associations are explicit.

The operation *reTitle(newTitle)* is an event simulating a user modification of the current project title.

The operations *deleteFile(path)* and *clearLog* are testing-dedicated operations to simulate the deletion of a file (basically a project) and the clean of the application log.
These two operations are defined with the following OCL expressions.

```
context:
  ProjectManager::deleteFile(path:PATH_NAME)
pre:
  project.oclIsUndefined() and
  path <> PATH_NAME::UNDEFINED and
  Project.allInstances()->exists(p.path=path)
post:
  Project.allInstances()->any(p|p.path=path).
path = PATH_NAME::UNDEFINED


context: ProjectManager::clearLog()
post:
  app.log = MESSAGE::NONE
```

All the other operations of *ProjectManager* class are events used in the state-machine defined in the sequel.

2.2.3 *Object diagram/Initial state*

Figure 2 presents the object diagram that depicts the initial state of SPM.

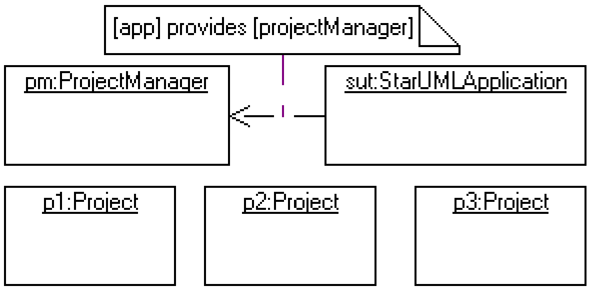


Figure 2. Initial state of SPM

We consider no project is open at the initial state of SPM.

2.2.4 *State-machine*

Figure 3 presents the state-machine used to describe the different dynamic states of the SUT. Our SPM system is very simple, so the corresponding state-machine is clearly comprehensive: a project is open or not. At the initial state no project is open.

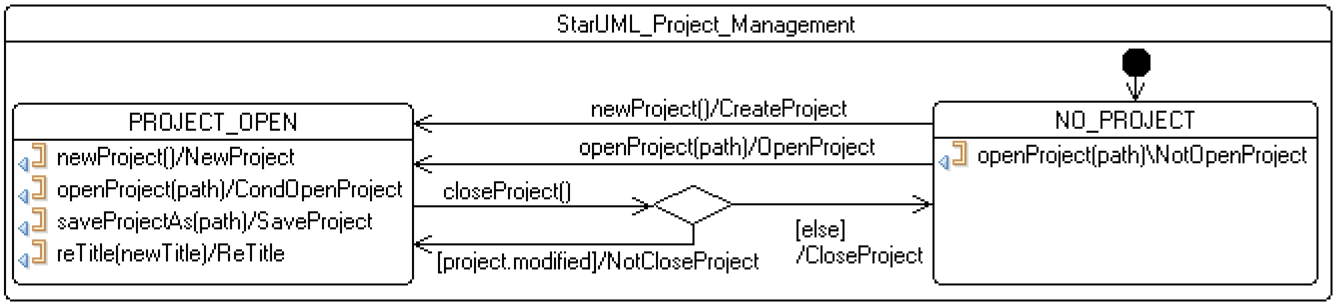

StarUML_Project_Management
PROJECT_OPEN
newProject()/NewProject
openProject(path)/CondOpenProject
saveProjectAs(path)/SaveProject
reTitle(newTitle)/ReTitle
newProject()/CreateProject
openProject(path)/OpenProject
closeProject()
[project.modified]/NotCloseProject
[else]
/CloseProject
NO_PROJECT
openProject(path)\NotOpenProject

Figure 3. State-machine of SPM

User events and linked actions realize the state modifications and the corresponding behaviors. The transition actions are defined in OCL as follows:

```
action NewProject
post:
  if not project.modified then
    let p = Project.allInstances()->
    any(p|p.path=PATH_NAME::UNDEFINED) in
        p.modified = true and
        project = p and
        app.log = MESSAGE::PROJECT_CREATED
        --@REQ: PRJ_CREATED
  else
    app.log = MESSAGE::SAVE_BEFORE
    --@REQ: SAVE_PRJ_BFR
  endif


action CreateProject
post:
  let p = Project.allInstances()->
  any(p|p.path=PATH_NAME::UNDEFINED) in
    p.modified = true and
    project = p and
    app.log = MESSAGE::PROJECT_CREATED
    --@REQ: PRJ_CREATED


action OpenProject
pre:
  path <> PATH_NAME::UNDEFINED
post:
  project = Project.allInstances()->
        any(p|p.path=path) and
  app.log = MESSAGE::PROJECT_OPENED
  --@REQ: PRJ_OPEN
```

```
action NotOpenProject
pre:
  path <> PATH_NAME::UNDEFINED and
  not Project.allInstances()->
        exists(p|p.path=path)
post:
  app.log = MESSAGE::FILE_DOES_NOT_EXIST
  --@REQ: NOT_PROJECT


action CondOpenProject
pre:
  path <> PATH_NAME::UNDEFINED
post:
  if Project.allInstances()->
exists(p|p.pathName=pathName) then
    project = Project.allInstances()->
    any(p|p.pathName=pathName) and
    app.log = MESSAGE::PROJECT_OPENED
    --@REQ: PRJ_OPEN
  else
    app.log = MESSAGE::FILE_DOES_NOT_EXIST
    --@REQ: NOT_PROJECT
  endif


action SaveProject
pre:
  path <> PATH_NAME::UNDEFINED
post:
  if Project.allInstances()->
exists(p|p.pathName=pathName) then
    app.log = MESSAGE::FILE_ALREADY_EXISTS
    --@REQ: FILE_ALRDY_EXISTS
  else
    project.modified = false and
    project.pathName = pathName and
    app.log = MESSAGE::PROJECT_SAVED
    --@REQ: PRJ_SAVED
  endif
```

```
action ReTitle
post:
  project.title = newTitle and
  app.log = MESSAGE::PROJECT_RETITLED
  --@REQ: MODIF_OK
action CloseProject
post:
  project.oclIsUndefined() and
  app.log = MESSAGE::PROJECT_CLOSED
  --@REQ: PRJ_CLOSED


action NotCloseProject
post:
  app.log = MESSAGE::SAVE_BEFORE
  --@REQ: SAVE_PRJ_BFR
```

Notice the specific tags to link functional requirements (Table 1) to corresponding behaviors. These tags make it possible to trace given informal requirements.

2.3 Test Generation and Automation

From a UML behavioral test model LEIRIOS Smart Testing™ extracts test objectives. A test objective is a pair *context-effect*, where *effect* is the system behavior to test and *context* the condition to fire it.

From these test objectives LEIRIOS Smart Testing™ carries out a theorem prover to automatically generate test cases. This prover is used to search for a path from the initial state to a test objective, and data values satisfying all the constraints along this path.

A test case is composed of:

- a *preamble* (potentially empty); the sequence of operations or events called to reach the behavior to test,
- a *body*; the execution of the tested behavior,
- a *postamble* (potentially empty); the sequence of operations to return to the model initial state. Postambles are used to chain the execution of many tests without reinitializing the system after each test. The generation with postamble is optionally.

Table 2 presents some generated test cases from the SPM model.

Table 2. Tests cases from SPM model

Id	**Tested UML element**	**Test definition**			**Tested requirement**
		preamble	*body*	*postamble*	

1	Internal transition
	PROJECT_OPEN
	newProject()
	/NewProject
		newProject()	newProject()	closeProject()
				clearLog()
					PRJ_CREATED

2		newProject()
		reTitle(TITLE_1)
			newProject()	saveProject(PATH_1)
				closeProject()
				deleteFile(PATH_1)
				clearLog()
					SAVE_PRJ_BFR

3	Transition
	NO_PROJECT $\rightarrow$
	PROJECT _OPEN
	newProject()
	/CreateProject
		-	newProject()	closeProject()
				clearLog()
					PRJ_CREATED

4	Internal transition
	PROJECT_OPEN
	openProject(path)
	/CondOpenProject
		newProject()
		saveProject(PATH_2)
			openProject(PATH_2)	closeProject()
				deleteFile(PATH_2)
				clearLog()
					PRJ_OPEN

5		newProject()	openProject(PATH_1)	closeProject()
				clearLog()
					NOT_PROJECT

6	Transition NO_PROJECT $\rightarrow \Diamond \rightarrow$ NO_PROJECT openProject(path) /NotOpenProject	-	openProject(PATH_1)	clearLog()	NOT_PROJECT

7	Transition
	NO_PROJECT $\rightarrow \Diamond \rightarrow$
	PROJECT_OPEN
	openProject(path)
	/OpenProject
		newProject()
		saveProject(PATH_2)
		closeProject()
			openProject(PATH_1)	closeProject()
				deleteFile(PATH_2)
				clearLog()
					PRJ_OPEN

8	Internal transition
	PROJECT_OPEN
	reTitle(newTitle)
	/ReTitle
		newProject()	reTitle(TITLE_1)	saveProject(PATH_1)
				closeProject()
				deleteFile(PATH_1)
				clearLog()
					MODIF_OK

9	Internal transition
	PROJECT_OPEN
	saveProjectAs(path)
	/SaveProject
		newProject()	saveProject(PATH_1)	closeProject()
				deleteFile(PATH_1)
				clearLog()
					PRJ_SAVED

10		newProject()
		saveProject(PATH_2)
			saveProject(PATH_2)	closeProject()
				deleteFile(PATH_2)
				clearLog()
					FILE_ALRDY_EXISTS

11	Transition PROJECT_OPEN → ◊ → NO_PROJECT closeProject() /CloseProject	newProject()	closeProject()	clearLog()	PRJ_CLOSED

12	Transition
	PROJECT_OPEN → ◊ →
	PROJECT_OPEN
	closeProject()
	/CloseProject
		newProject()
		reTitle(TITLE_1)
			closeProject()	saveProject(PATH_1)
				closeProject()
				deleteFile(PATH_1)
				clearLog()
					SAVE_PRJ_BFR

The LEIRIOS Smart Testing™ solution provides adapters and exporters to manage and/or execute the generated test cases.

For instance test cases can be published to HTML/XML languages. For an execution on the SUT, test cases can be translated to test scripts in any language. Adapters are also provided to export test cases in test management and execution tools such as HP/Mercury Quality Center.

For our case study, many ways are available to automate the execution of test scripts.

StarUML is written in Delphi but it is *multi-lingual project.* So we can translate generated test cases to test script in the appropriate language and execute them from the application sources.

StarUML application owns a graphical user interface (GUI). This GUI normally offers all the functionalities available to manually design functional tests. Automated testing tools – like HP/Mercury WinRunner – simulates a human user by moving the mouse cursor over the application, clicking GUI objects, and entering keyboard input. Such tool enables the writing of scripts performing such process. These test scripts can be automatically generated by LEIRIOS Smart Testing™ from the generated test cases.

StarUML exposes open API to outside to access most programs that is UML meta-model, application object and so on. For our SUT SPM, we can translate test cases to JScript tests to execute them. For instance Table 3 presents a translation of the test case 5 (Table 2). Each UML operation/event is translated to a JScript function. These JScript methods are depicted in the adaptation layer. This one is written once and shared among all scripted tests.

Table 3. Example of JScript test

Test case		Corresponding test script

preamble	newProject()	`var app`
		`var prjmgr`
		`var PATH_1`
		
		`Init() //SETUP VARS`
		
		`//PREAMBLE`
		`NewProject()`

body	openProject(PATH_1)	`OpenProject(PATH_1)`
		`//CHECK ORACLE`

postamble	closeProject()
	clearLog()
		`CloseProject()`
		`ClearLog()`

Adaptation layer

```
function Init() {
  app = new
  ActiveXObject("StarUML.StarUMLApplication")
  prjmgr = application.ProjectManager
  PATH_1 = "..\\projects\\p1.uml"
}
function NewProject() {
  prjmgr.NewProject()
}
function OpenProject(path) {
  prjmgr.OpenProject(path)
}
function CloseProject() {
  prjmgr.CloseProject()
}
function ClearLog() {
  app.Log(“”)
}
```

In every instance the main issue met while the automation is the determination of the test verdict. Which StarUML elements are observable? How link the values of StarUML objects and test model elements? The following section notably discusses this crucial point.

3. DISCUSSION

In this section we discuss about the different issues met while this experimentation.

Scope definition

The UML test model is based on the use cases and/or functional requirements. In our case study we arbitrarily decide the perimeter. Then we gave some use cases potentially realizable on the application. Finally we wrote some functional requirements regarding the functional scope. For an industrial case study it is preferable to decide all these scope elements with the validation manager or equivalent.

UML modeling

The modeling step doesn't bring major issue if the previous step is right completed. The idea is to abstract and reduce the SUT to design a compliant test model. Concretely operations/events correspond to user functionality and attributes precise the different system behaviors.

However the test model is a behavioral model of the SUT. So the test model designer has to know which behaviors are expected regarding the user actions. In this case study, we don't precisely know how the StarUML interacts. So use cases and functional requirements given in this paper are proper to authors.

Test verdicts and observation points

The test verdict (pass or fails) is determined by comparing some values returned by the SUT and the equivalent values expected in the test model.

In this case study we have real problem to observe the behaviors of the StarUML application while the test execution. For instance the execution of the test script example (Table 3) triggers an exception like "File not found". But this information can not be catched by the test script. To workaround this problem, we have to manage this exception type in the script (or in its adaptation layer). For instance the code in Figure 4 manages the exception and gives the corresponding information in the log. So the test verdict can be determined by comparing the StarUML log (if observable!) and the attribute *log* designed in the test model.

```
function OpenProject(path) {
  var fs = new
ActiveXObject("Scripting.FileSystemObject"  )
  if (fs.FileExists(path))
  {
    prjmgr.OpenProject(path)
  }
  else
  {
    application.Log("File not found")
  }
}
```

Figure 4. Example of an exception observation

StarUML API contents

We use StarUML API to concrete the generated tests to JScript script. To determine the verdict of a test, we have to observe the system via this API. So we strongly depend upon the fullness of this API. For instance to determine the verdict of the test case example (Table 3), we can compare the *log* values. For that the API has to offer an access to the value of application log.

4. CONCLUSION

In this paper we illustrate the LEIRIOS Smart Testing™ approach on a given case study. This solution automates the process of conception, generation, management and execution of a functional test suite.

Such experimentation highlights some issues tied to the automation of application testing. In this precise case study, the scope of the functional testing and the system observation points turn out critical to success such validation campaign.

The LEIRIOS Smart Testing™ solution is currently deployed on large applications in the domains of Enterprise IT information systems and eTransactions systems (banking, ticketing or e-Admin applications).

5. REFERENCES

[1] M. Utting and B. Legeard. *Practical Model-Based Testing - A tools approach.* Elsevier Science/Morgan&Kaufmann, 2007. 454 pages, ISBN 0-12-372501-1.

[2] M. Utting, A. Pretshner and B. Legeard. *A taxonomy of Model-Based Testing*. Working Paper, April 2006. ISSN 1170-487X.

[3] TorX – Test Tool Information. http://fmt.cs.utwente.nl/tools/torx.

[4] C. Jard and T. Jéron. *TGV : theory, principles and algorithms*, J. Software Tools for Technology Transfer, 2005.

[5] B. Koch, J. Grabowski, D. Hogrefe and M. Schmitt. *Autolink – A Tool for Automatic Test Generation from SDL Specifications*. in Proc. IEEE International Workshop on Industrial Strength Formal Specication Techniques (WIFT98), Boca Raton, Florida, Oct. 1998.

[6] S. J. Prowell, *JUMBL: A Tool for Model-Based Statistical Testing.* in Proc. HICSS'03, IEEE , p. 337c, 2003.

[7] Dulz W. and Fenhua Z. *MaTeLo - statistical usage testing by annotated sequence diagrams, Markov chains and TTCN-3.* in Proc. Third International Conference on Quality Software, pp 336-342, Nov. 2003.

[8] D. M. Cohen, S. R. Dalal, M. L. Fredman and G. C. Patton. *The AETG System: An Approach to Testing Based on Combinatorial Design.* vol. 71, no. 3, pp. 41-47, Mar. 1992.

[9] F. Bouquet, C. Grandpierre, B.Legeard, F. Peureux, N. Vacelet and M. Utting. *A subset of precise UML for model-based testing*. A-MOST 2007: 95-104, Jul. 2007.

[10] J. Warmer and A. Kleppe. *The Object Constraint Language Second Edition: Getting Your Models Ready for MDA*. Addison-Wesley, 2003.