



D 7.3: A MODEL-BASED TESTING APPROACH FOR EVOLUTION

Fabrice Bouquet, Frederic Dadeau, Stephane Debricon, Elizabeta Fournernet, Pierre-Alain Masson (INR), Zoltan Micksei, Daniel Varro (BME), Berthold Agreiter, Michael Felderer (UIB), Bruno Legeard, Olivier Albiez, Julien Botella, Olivier Bussenot, Ed-die Jaffuel, Christophe Grandpierre, Jean-Luc Hamot, Aurelien Masson, Dooley Nsewolo (SMA), Elisa Chiarani, Federica Paci, Fabio Massacci (UNITN), Jan Jur-jens (OU/TUD)

Document Information

Document Number	D7.3.
Document Title	A model-based testing approach for evolution
Version	1.5
Status	Draft
Work Package	WP 7
Deliverable Type	Report
Contractual Date of Delivery	M24
Actual Date of Delivery	xx xx 2010
Responsible Unit	SMA
Contributors	SMA, INR, UIB, BME, TUD, UNITN
Keyword List	Model-based testing, Software Evolution, Security Properties
Dissemination	level PU

Document change record

Version	Date	Status	Author (Unit)	Description
0.1	10.09.3	Draft	S. Debricon (INR) B. Legeard (SMA)	Summary - 1st draft
0.2	10.09.24	Draft	S. Debricon (INR) J. Botella (SMA)	Architecture
0.3	10.10.1	Draft	F. Bouquet, E. Fournieret, P-A. Masson (INR)	MBT Process
0.4	10.10.16	Draft	F. Bouquet, F. Dadeau (INR)	Links WPs
0.5	10.10.22	Draft	B. Agreiter, M. Felderer (UIB)	TTS
0.6	10.11.2	Draft	B. Legeard (SMA)	Introduction - Key challenges
0.7	10.11.10	Draft	B. Agreiter, F. Innerhofer-Oberperfler (UIB)	WP2–WP5–WP7 links
0.8	10.11.18	Draft	Julien Botella, Olivier Bussenot, Eddie Jaffuel, Christophe Grandpierre, Jean-Luc Hamot, Aurelien Masson (SMA)	WP7 Demonstrator - Test generation technologies
0.9	10.11.25	Draft	B. Agreiter, M. Felderer (UIB)	update TTS
1.0	10.11.25	Draft	F. Bouquet, F. Dadeau (INR)	Links WPs & typo
1.1	10.12.09	Draft	B. Agreiter (UIB), S. Debricon (INR)	General update
1.2	10.12.10	Draft	Z. Micksei (BME), S. Debricon (INR)	General update
1.3	10.12.15	Draft	J. Bernet (GTO), S. Debricon (INR)	General update
1.4	11.01.05	Draft	S. Debricon (INR)	New Executive Summary
1.5	11.01.31	Final	F. Bouquet	Integration of quality check

Executive summary

The objective of deliverable D7.3 is to produce a proof-of-concept implementation of a model-based testing tool for evolution. This demonstrator provides a tool-set to ensure the preservation of security properties for long-living evolving systems using software testing. Therefore, the deliverable D7.3 is made of two components: a software prototype (called WP7 Demonstrator) and this document that presents the research results, the demonstrator architecture and a sample case study.

The main results are twofold:

- an approach for testing security properties, based on the use of test schemas that formalize testing needs. Security properties are covered by a test generation process using a behavioral model of the SUT and associated test schemas.
- an approach for change management by means of model comparison. Our objective is to ensure the important criteria defined in D7.1: test repository stability, traceability of changes, impact analysis and ability to automatically structure the test repository into evolution, regression and stagnation test suites.

D7.3 in the Project Timeline

This deliverable is the result of task 7.3 'Model-based testing for Evolution' (M12-M24), and is the milestone M24 for workpackage 7. We provide a demonstrator based on two elements. On one hand, it uses the Smartesting test generation technologies (see D7.1 section 4 - WP7 Background), augmented with new algorithms for test generation. We also provide an adapted architecture to help handling change analysis and a new packaging of a standalone model animator and test generator. On the other hand, the demonstrator is made up of new components developed within task 7.3:

1. an interpreter of test schemas for testing security properties,
2. an impact analyzer (that compares two versions of a model in order to identify the changes and produce a change items file),
3. a test classification component (based on the change items file), that uses the model animator and the test generator to produce tests that are organized into test suites,
4. a test publisher that manages a test repository. It keeps tracks of previous tests status and minimizes repository changes.

This document presents the rationale of the approaches both for security property testing and change management and describes in details the demonstrator (architecture and components). We propose an example to illustrate the main usage scenarios and added values for change management and testing security properties. Finally we position WP7 model-based testing methods in the overall SecureChange process.

Validation

This deliverable contains several artefacts that can be evaluated wrt validation criteria defined by workpackage 1 in deliverable D1.2.

Scientific criteria

- The language described in chapter 3 falls within the criteria 'Evidence of efficiency'.
- Chapter 4 contains a description of WP7 test classification algorithm that can be associated to the criteria 'Precise computation'.
- The demonstrator described in chapter 5 will be evaluated by criteria 'Computer-aided computation'.

Industrial criteria

- Evaluation of the industrial criteria will be the subject of deliverable D7.4 due at M36.

Integration

Integration links among the different workpackages are discussed in chapter 8. Figure 1 gives an overview of partners' interactions with WP7. The workpackage has interfaces to 4 other work packages. The first is WP3. WP3 provides requirements and their evolutions. The second is WP4. WP4 provides a validated security model and their evolution. The third is WP5. WP5 provides risks associated to security properties. The last is WP6. It is a special link because there is not direct interaction between WP6 and WP7 but they are complementary for security verification.

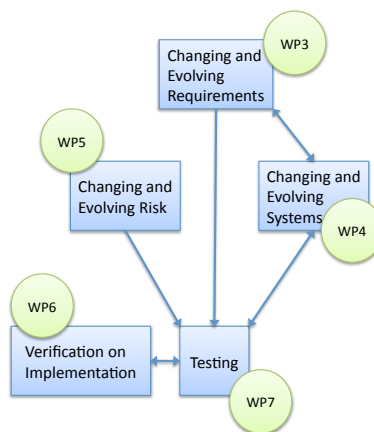


Figure 1: Links between WP7 and others WPs

WP3-WP7 The integration between requirements engineering and testing engineering is done at processes level. The two processes should still be understood as separate processes with their own iterations, activities and techniques for managing change. The integrated process explains at which steps of the respective processes that the conceptual level interface can or should be invoked. We distinguish between what we refer to as shared elements on the one hand and mappable elements on the other hand. The

shared elements are concepts that are common to requirements engineering and testing, with the same semantics in both domains. The mappable elements are concepts from one domain that are not shared by the other, but are nevertheless related to the other domain and can be mapped to concepts of the other domain.

WP4-WP7 Based on the Global Platform life-cycle (POPS), this link shows how model-based testing for evolving systems can benefit from the techniques developed in WP4. The general requirement considered is 'Specification Evolution' and the common property is 'Life-cycle consistency'.

WP5-WP7 The integration between the risk assessment methodology of WP5 and the security testing approach of WP7 is outlined in the Deliverable D.5.3 in Chapter 8. It is also shortly reported on in the present deliverable to highlight how the solutions of WP5 and WP7 fit in the overall Integrated SecureChange Process methodology. The integration between WP5-WP7 is in terms of mapping artifacts from the risk model domain to the test model domain and vice versa. Based on these options for mapping of model artifacts, the risk assessment activities and the testing activities are integrated so as to allow the two domains to leverage on each other. The integration is demonstrated in the HOMES case study, addressing the change requirement of bundle lifecycle operations and the security properties of policy enforcement and security expandability.

WP6-WP7 Workpackages can address separate issues for the protection against threats. In fact, the interest of a connection between these two workpackages is that each hypothesis proposed by one workpackage is tackled by the other. So the completeness of validation can only be done if the workpackages interact as proposed in SecureChange process.

TABLE OF CONTENTS

Document information	1
Document change record	2
Executive summary	3
Abbreviations and Glossary	11
1 Introduction	13
2 Demonstrator overview	16
3 Approach for testing security properties	18
3.1 Principle	18
3.2 TestDesigner Schema Language	19
3.2.1 Presentation	19
3.2.2 Language Key Words	19
3.2.3 Language Syntax	20
3.3 Examples of Test Schemas	20
3.3.1 Example 1	21
3.3.2 Example 2	22
3.4 Related Works and Originality	22
4 Approach for change management in the MBT process	24
4.1 Impact of evolution on Model	24
4.1.1 SecureChange MBT approach	24
4.1.2 Evolution of Test Cases	26
4.1.3 New selective test generation method	29
4.1.4 Evolution in Test Suites	30
4.2 Impact of evolution for security properties	33
4.2.1 Evolution of schemas and requirements	33
4.2.2 Evolution in Test Suites with respect to Security Testing	33
5 Demonstrator	35
5.1 Architecture	35
5.1.1 SeTGaM	35
5.1.2 Smart Publisher	39
5.2 Test generation improvements	40
5.2.1 Model animation API	40
5.2.2 Test generation API	42

5.2.3	Schema-based test generator	42
5.3	Graphical User Interface	45
5.3.1	Chart panel	46
5.3.2	Model selection panel	46
5.3.3	Chart selection panel	46
5.3.4	SeTGaM process	47
5.3.5	Test generation process	47
5.3.6	Test publication	47
6	Example	49
6.1	Example	49
6.1.1	General description of eCinema application	49
6.1.2	Test Generation with <i>TestDesigner</i> for eCinema	50
6.2	Evolution of the eCinema's system	51
6.2.1	Changes in requirements	51
6.2.2	Selective test generation using SeTGaM for eCinema	52
6.3	Security Properties Testing on eCinema	55
6.4	Comparison of SeTGaM with two other approaches	56
7	Telling TestStories: Another Point of View	57
7.1	Evolutions	57
7.1.1	Requirements Evolution	57
7.1.2	Evolution of the System or Environment	58
7.2	Methods and Techniques	58
7.2.1	Telling TestStories Metamodel	58
7.2.2	Test Life Cycle	58
8	Integration in SecureChange process	61
8.1	WP3 – WP7	62
8.1.1	Traceability between functional requirements extracted from specifica- tion and generated tests	62
8.1.2	Upgrading a test model by requirements models comparison	63
8.2	WP4 – WP7	63
8.2.1	General process	64
8.2.2	Concrete integration	65
8.3	WP5 – WP7	66
8.4	WP6 – WP7	67
8.4.1	Interest of the link between WP6 and WP7	67
8.4.2	Concrete scenario on case study	68
9	Conclusion	70
A	Requirements for eCinema	72
B	Requirements for eCinema Evolution	75
C	Transitions for eCinema	77
D	Transitions for eCinema Evolution	78

LIST OF FIGURES

1	Links between WP7 and others WPs	4
1.1	Process supported by SecureChange project	13
2.1	Overall process	16
2.2	Demonstrator main components	17
3.1	Syntax of the TestDesigner Schema Language	20
3.2	Evolution in the GP Card Life Cycle statechart	21
4.1	The Model-Based Testing Process	25
4.2	Smart publisher	26
4.3	Test case life-cycle	27
4.4	Process to determine the status of the tests	28
4.5	Test suites composition	31
4.6	Process for testing security properties	33
4.7	Process for testing security properties with respect to evolution	34
5.1	SeTGaM overview	35
5.2	The process activity diagram	36
5.3	Test cases generation process	37
5.4	The SeTGaM activity diagram	37
5.5	Classification activity diagram	38
5.6	Re-testable tests processing	39
5.7	Smartesting Test Designer GUI	40
5.8	Smartesting Test Designer Animation API usage	41
5.9	Smartesting Test Designer Generation API usage	42
5.10	Smartesting Schema Editor GUI	45
5.11	The EvoTest panel	45
5.12	Test suites chart panel	46
5.13	Test status chart panel	47
5.14	TestLink tool	48
6.1	Class diagram of eCinema	49
6.2	Statechart diagram of eCinema.	50
6.3	Class diagram of eCinema	52
6.4	Statechart of eCinema, evolved version	52
7.1	Telling TestStories Metamodel.	59
7.2	State machine describing the life cycle of Test elements.	60

8.1	Process supported by SecureChange project	61
8.2	Links between WP7 and others WPs	61
8.3	Using requirement model to create the test model	62
8.4	Test model upgrade process	63
8.5	Integration between WP7 and WP4	64
8.6	Change story of the HOMES case study	66

LIST OF TABLES

1	Abbreviations used in the document	11
2	Glossary	12
3.1	Keywords for the TestDesigner Schema Language	19
6.1	Number of tests for eCinema - evolution using the Retests-all, Regenerate-all and SeTGaM methods	56
A.1	Requirements of eCinema.	72
A.2	Test targets of eCinema part 1/2.	73
A.3	Test targets of eCinema part 2/2.	74
B.1	New Requirements of eCinema, evolution.	75
B.2	Impacts in test targets of eCinema.	75
B.3	New test targets of eCinema, evolution.	76

Abbreviations and Glossary

Abbreviations

Abbreviations	References
API	Application Programming Interface
FSM	Finite State Machine
ISTQB	International Software Testing Qualifications Board
MBT	Model-Based Testing
REQ	Requirement
SBTG	Schema-Based Test Generator
SeTGaM	Selective Test Generation Method
SUT	System Under Test
TTS	Telling TestStories

Table 1: Abbreviations used in the document

Glossary

Term	Definition
Adapter	Piece of code to concretize logical tests into physical tests
Deletion Test Suite	Test suite gathering tests from previous versions of the software that are outdated or failed in the current version.
Evolution Test Suite	Test suite targeting SUT evolutions
Logical Test	See Test Case
Model Layer	Link of model's operations in Test cases
Model-Based Testing	Process to generate tests from a behavioural model of the SUT
Status of Test Case	New, obsolete (outdated, failed), adapted, reusable (re-executed, unimpacted)
Physical Test	See Test Script
Requirements	Collection of functional and security requirements
Regression Test Suite	Test suite targeting non-modified part of the SUT
Schema	See Test Schema
Stagnation Test Suite	Test suite targeting removed part of the SUT
System Model	Model of the SUT used for development
Test Case	A finite sequence of test steps
Test Intention	User's view of testing needs
Test Model	Dedicated model for capturing the expected SUT behaviour
Test Suite	A finite set of test cases
Test Script	Executable version of a test case
Test Schema	A regular-based expression to drive automated test generation for testing security properties
Test Sequence	See Test Case
Test Step	Operation's call or verdict computation
Test Strategy	Formalization of test generation criteria
Test Objective	High level test intention

Table 2: Glossary

1 Introduction

This deliverable summarizes the results of the work carried out in task 3 of Work Package 7: A model-based testing approach for evolution in the context of security engineering for lifelong Evolvable Systems.

In task 7.1 we have provided the state of the art presented in deliverable D7.1. In task 7.2 we have developed original methods and algorithms for handling model-based testing for evolution and security testing (presented in Deliverable D7.2). The work achieved in task 7.3, presented in this document, consists in prototyping a proof-of-concept implementation of model-based testing tool for evolution (called WP7 Demonstrator). This demonstrator is based on Smartesting MBT technologies and integrates the WP7 results regarding change analysis, test generation based on evolution, test generation for security properties, classification and publication of the tests in a test repository depending of the test status wrt evolution.

Therefore, the goal of this document is to introduce the main concepts that underlie the WP7 Demonstrator and to detail the architecture and implementation of this Demonstrator. The evaluation of these new methods and tools is part of WP7 task 7.4 in the context of real case studies provided by the SecureChange consortium and connected with WP1.

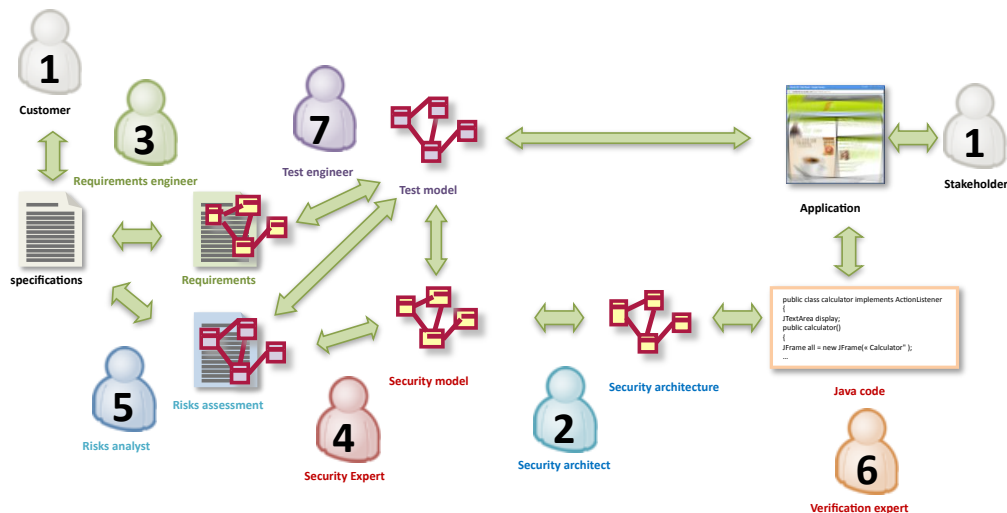


Figure 1.1: Process supported by SecureChange project

Figure 1.1 gives an overall picture of the relationship between the main phases of the SecureChange Security Engineering Process. Model-based testing for evolution is a black-box testing phase, targeting both functional testing and testing that the SUT conforms to

some security properties, and that this conformance is preserved throughout successive evolutions. The inputs of the model-based testing process for evolution are the various versions of the requirements, the risk assessment and the security model. Model-based testing is the automation of test design and test maintenance of models used for automated test generation (called test models). Therefore, the MBT process ensures the continuous update of the *test repository* used for checking the conformance of the SUT with its requirements (both functional and security properties). Test Engineers (Number 7 in the Figure) drive this process on the basis of test models and test selection criteria.

As shown in D7.1 and D7.2, WP7 research work is conducted to provide solutions (at scientific and technology levels) for solving key challenges to ensure the preservation of system security for lifelong evolving systems by means of model based tests. In a model-based testing perspective, these key challenges concern the following four main dimensions:

- **The coverage of security properties by the MBT process** - How to support the validation engineer to produce adequate test suites wrt security properties and to maintain this security test repository in the context of evolution of requirements, usages and attacks.
- **The stability of the test suite through evolution** - Model-based test generation may produce lots of tests, and a completely new test set for each new generation. But, the convergence of the validation process is based on correcting both the SUT (in case of implementation error) and the test repository (when tests are wrong). The stability of the generated suites through re-generation is a key value to support a rapid convergence of this validation process.
- **The organization of the test repository wrt evolution** - Time to market is always a strong pressure for the validation team. Therefore, it is very important to separate between deletion tests, evolution tests, regression tests and more originally, stagnation tests, to help the validation team to prioritize test execution.
- **The efficiency of this test generation process** - Exploiting change impact analysis results for test generation helps to provide an efficient test generation process in terms of test generation time, and then shortening the test generation phase.

The main results, behind the state of the art, obtained by WP7 task 7.2 (approaches and solutions) and task 7.3 (implementation of the Demonstrator) are the followings:

- **A new schema language for driving test generation from security properties** - This schema language supports the formalization of test intention linked to each security property to drive automated test generation.
- **New approach for the classification of generated test cases wrt evolution** - The classification of the test repository after each evolution and test generation in evolution, regression and stagnation test suites help the validation team to prioritize test execution. The concept of stagnation testing is original in the MBT area.
- **New selective test generation method** - The method is based on the research done on impact analysis based on dependence results when considering test model evolutions. It automatically produces new test suites to ensure that the system still conforms its requirements.

All these technical results are fully integrated in a ready-to-use prototype implementation. This Demonstrator will be evaluated in task 7.4 "Test generation for case studies" on SecureChange HOMES and POPS case-studies.

The sequel of this document is structured as follows. Section 2 introduces the overall architecture of WP7 Demonstrator. Section 3 presents the original concepts that characterize the approach for testing security properties implemented in the WP7 Demonstrator. Section 4 presents the original concepts introduced to manage evolution and implemented in WP7 Demonstrator. Section 5 describes in depth the implementation of the Demonstrator, and Section 6 illustrates this new MBT process on a sample example. Section 7 presents another point of view studied within WP7 and based on Telling TestStories scenario-based test approach. Section 8 shows how WP7 Demonstrator is related to other parts of SecureChange process, particularly regarding requirements management (WP3), modeling for security analysis (WP4), risk assessment (WP5) and verification (WP6). And finally, Section 9 sums up the results and introduces further works to be addressed in task 7.4.

2 Demonstrator overview

The demonstrator associated to this deliverable is part of an overall process depicted in Figure 2.1. This figure is divided into 3 lanes, describing the sequence of processes applied for 3 versions of the requirements for a specific software.

The first lane (Version 1) describes a classical model-based testing process. A test engineer takes as input the requirements to model the System Under Test. This test model gathers static and dynamic views of the system. A class diagram is used to capture the points of control and observation, and initial state of objects. A state machine with OCL code describes the system's expected behavior. Schemas are also used to capture some properties or specific behaviors of the system.

This model of the system is used by the test generation technology (based on Smartesting

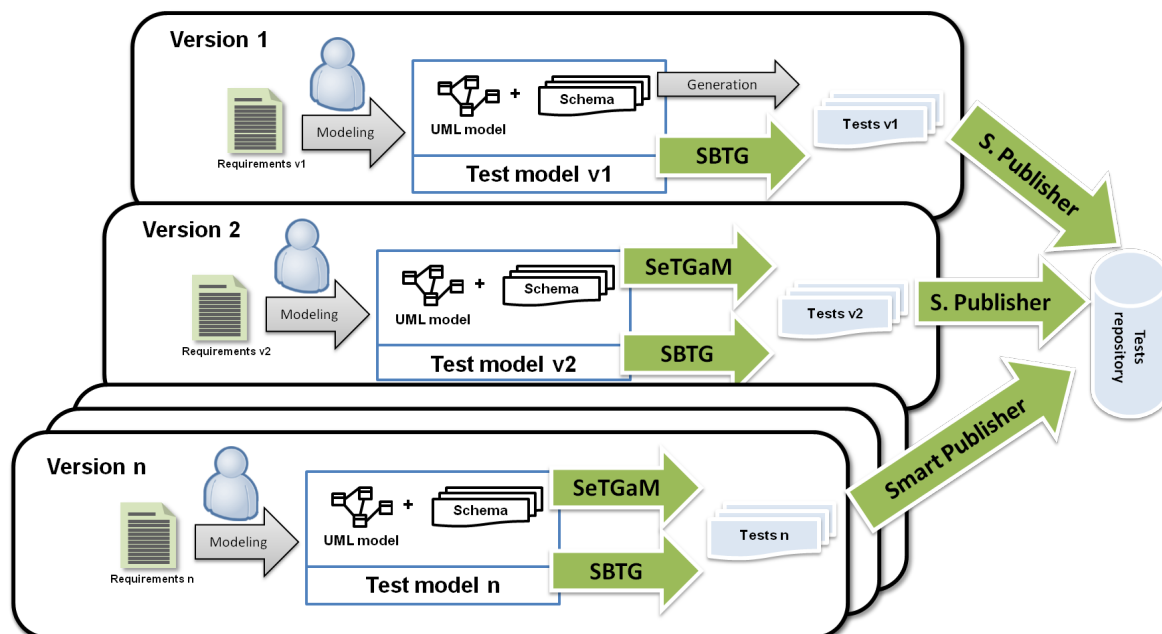


Figure 2.1: Overall process

Test Designer) to compute test cases. More precisely, test cases are generated by covering paths in the state machine. A test verdict is computed based on the OCL code collected on the path. Test cases are sequence of actions associated to expected results at each step of the test. The SBTG (schema-based test generator) component uses test schemas and the UML model to produce test cases. A test repository is used to store the actual test cases, together with the results of their execution on the SUT when done.

This general MBT process, based on a symbolic model animation has been extensively described in SecureChange deliverable D7.1 - section 4. The behavior of the SBTG com-

ponent is described in Section 5.2.3.

The following lanes (Version 2 and Version n) describe the process for the next 2 evolutions of the system. In these cases, the test engineer updates the test model with the modifications available from the new version of the requirements. New test cases will be produced by SeTGaM (selective test generation method) and SBTG components. Then, a smart publisher component is used to store and to organize test cases. The test repository is updated with test cases and a test status is used to trace the history of a test case. Test cases are gathered in new test suites to enhance functional and security testing.

The process introduced in Figure 2.1 is supported by an Eclipse stand-alone application called *EvoTest* available for IBM Rational Software Architect 7.5.0 models.

We have produced several components that are presented in Figure 2.2. The *EvoTest* interface is composed by 3 components:

- a component for the selective test generation method (SeTGaM),
- a component for the schema-based test generator (SBTG),
- a component to upgrade the test repository (Smart Publisher).

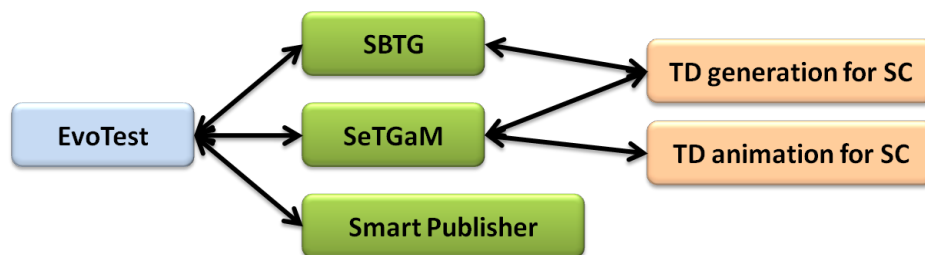


Figure 2.2: Demonstrator main components

The SeTGaM component is described in detail in Section 4.1.3 and 5.1 and uses specific components developed for the project:

- Test Designer animation for SecureChange
- Test Designer generation for SecureChange

Those components take into account specificities introduced by our approach and are presented in Section 5.2.1 and in Section 5.2.2.

The schema-based test generator is described in Section 5.2.3 and is composed of an editor to create schema and a specific test generator.

3 Approach for testing security properties

In this section we explain the principle that we adopt for testing security properties. We use for that a schema language that is described in Section 3.2. Further, we point out the originality of our approach w.r.t. existing approaches.

3.1 Principle

Model Based Testing makes use of selection criteria that indicates how to *select* the tests to be computed from the model. These criteria usually ensure a given structural coverage of the model, such as *all the states*, or *all the transitions*, etc. Each test is a sequence of operation calls with parameter values, which yields a distinguished execution of the model. Their results are predicted by the model. Our approach for testing security properties relies on defining additional selection criteria in the shape of *test schemas*. A test schema is a high level expression that formalizes a test intention linked to a security property to drive the automated test generation on the behavioral model.

Indeed, structural testing essentially provides control- and/or data-flow coverage of the model. The tests exercise the functionalities of the system by directly activating and covering the corresponding operations. Industrial studies have proven the efficiency of the method to detect faults in an implementation (see for example [10, 4]). Nevertheless, structural selection criteria may become insufficient to exercise the SUT in tortuous situations. We think for example of particular scenarios where a security property could become violated due to an unusual sequencing of the operation calls. These scenarios can be described by means of a *test schema*, which we consider as a dynamic selection criteria in the sense that it orchestrates the successive calls of the operations of the model. The tests extracted from the model by means of a test schema are sequences of operation calls corresponding to the unfolding of that schema.

In our approach, the security requirements that a system must fulfil are expressed as a set of *security properties*. We propose test schemas as a means to exercise the system for validating that it behaves as predicted by the model w.r.t. these security properties. Based on his know-how, an experienced security engineer will imagine possible scenarios in which he or she thinks the property might be violated by an erroneous implementation, and then on the basis of this test intention, (s)he will formalize test schemas to drive the automated test generation.

We have defined in [17] the concepts of such test schemas, in the shape of a language. It is based on regular expressions and allows the security testing engineer to conceive its test schemas in terms of states to be reached and operations to be called. The formal semantics of this language has been defined in [18].

Based on this conceptual language, an operational language has been defined within the SecureChange project and implemented as a plug-in to the Test Designer suite, to describe

test schemas in a “textual” way. This language is called *TestDesigner schema language*. We now present it and provide a couple of illustrative examples.

3.2 TestDesigner Schema Language

3.2.1 Presentation

A dedicated schema language editor has been implemented as a plug-in of Test Designer, version 4.1.2. Its aim is to provide a means to express security properties at a high level, close to a textual representation or by using usual computer programming paradigms. The expression of these properties allows for generating test specifications, called *Test Case Specification - TCS*, that are high level scenarios from which tests will be generated by Test Designer.

The language relies on combining keywords, to produce expressions that are both powerful and easy to read by a validation engineer. We first define the keywords of the language and then its syntax.

3.2.2 Language Key Words

In Table 3.1 we list the keywords of the language. For each keyword, we give its intuitive meaning.

for_each	quantifier for an operation or a behaviour
from	to introduce a list of operations or behaviours
then	a separator for sequencing the targets to be reached
use	to introduce an operation, a behaviour or a variable to use
to_reach	to introduce a state to be reached
to_activate	to introduce a behaviour to be activated
state_respecting	to introduce a constraint that characterize a set of states
on_instance	to introduce an instance on which a constraint holds
any_operation	the set of all the operations of the model
any_operation_but	the set of all the operations of the model minus the ones whose list follows
or	for a disjunction of operations or of behaviours
any_behaviour_to_cover	the set of all the behaviours of the model
any_behaviour_to_cover_but	the set of all the behaviours of the model minus the ones whose list follows
behaviour_activating	to introduce a list to be covered of behaviours tagged in the model
behaviour_not_activating	to introduce a list whose complementary must be covered of behaviours tagged in the model
at_least_once	repetition operator indicating to apply at least once the operation or behaviour previously specified
ant_number_of_times	repetition operator indicating to apply any number of times the operation or behaviour previously specified
\$	variable prefix
REQ	to introduce a tag that corresponds to a requirement
AIM	to introduce a tag that corresponds to an aim

Table 3.1: Keywords for the TestDesigner Schema Language

3.2.3 Language Syntax

The syntax of the language is defined by means of the grammar given in Figure 3.1. Roughly speaking, the language makes it possible to design test schemas as a sequence of steps, each step being composed of a set of operations (possibly iterated at least once, or many times) and aiming at reaching a given target (a specific state, the activation of a given operation, etc.).

SCHEME	::=	(QUANTIFIER_LIST ,)? SEQ
QUANTIFIER_LIST	::=	QUANTIFIER (, QUANTIFIER)*
QUANTIFIER	::=	for_each VAR from (BEHAVIOR_CHOICE OP_CHOICE)
BEHAVIOR_CHOICE	::=	any_behaviour_to_cover any_behavior_to_cover_but BEHAVIOR_LIST
BEHAVIOR_LIST	::=	BEHAVIOR (or BEHAVIOR)*
BEHAVIOR	::=	behavior_activating TAG_LIST behavior_not_activating TAG_LIST
TAG_LIST	::=	{ TAG (, TAG)* }
TAG	::=	REQ: <u>tag name</u> AIM: <u>tag name</u>
OP_CHOICE	::=	any_operation OP_LIST any_operation_but OP_LIST
OP_LIST	::=	OPERATION (or OPERATION)*
OPERATION	::=	<u>operation name</u>
SEQ	::=	<u>BLOC (then BLOC)*</u>
BLOC	::=	use CONTROL (RESTRICTION)? (TARGET)?
CONTROL	::=	OP_CHOICE BEHAVIOR_CHOICE VAR
VAR	::=	<u>\$variable name</u>
RESTRICTION	::=	at_least_once any_number_of_times
TARGET	::=	to_reach STATE to_activate BEHAVIOR to_activate VAR
STATE	::=	state_representing <u>ocl constraint</u> on_instance <u>instance name</u>

Figure 3.1: Syntax of the TestDesigner Schema Language

3.3 Examples of Test Schemas

To illustrate the use of the language, we give in this section two examples of security properties and their associated schemas.

These examples refer to the POPS case study. They are related to the card life cycle. As show in Figure 3.2, a card have several states, the last one being the TERMINATED state. The *Card Terminate privilege* for an application allows to set the card to the TERMINATED state, therefore killing the card by permanently disabling all card content management and life cycle functions. In other words, once in the TERMINATED state, a card cannot revert to another state.

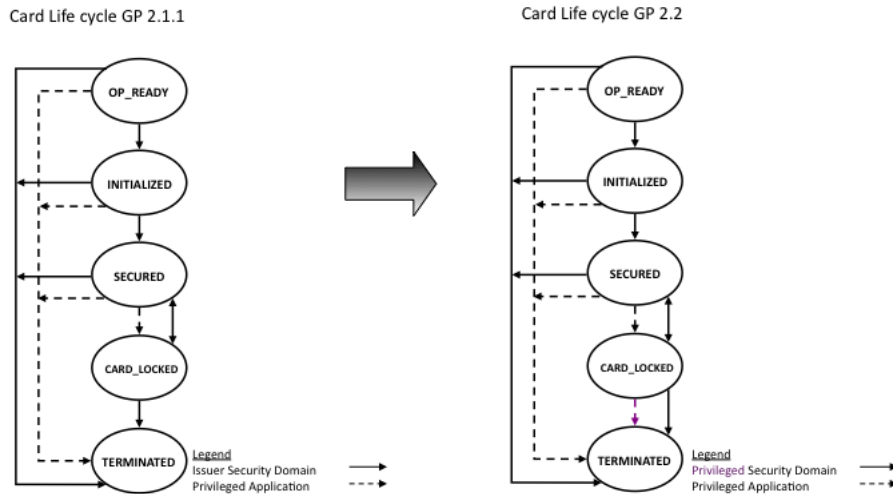


Figure 3.2: Evolution in the GP Card Life Cycle statechart

3.3.1 Example 1

The first security property for which we exhibit test schemas is expressed informally (i.e. in the natural language) as: *For any execution, whenever the card is put in the TERMINATED state by means of a set_status issued by a privileged application, then it should not be possible to revert to another state.*

Test Intention A scenario to test this security property can be described informally as:

- select an application with the Card Terminate Privilege;
- set the status of the card to TERMINATED;
- try all operations (to see if they behave as predicted by the model, i.e. by returning a status word of error).

Test schema With the TestDesigner Schema Language of Section 3.2, we express it as:

```

for_each $X from any_operation,
for_each $Y from any_operation,
  use $X at_least_once to_reach state_respecting (self.selectedApp.cardTermPriv = true)
on_instance "card" then
  use set_status to_reach state_respecting (self.state = TERMINATED) on_instance "card"
then
  use $Y

```

3.3.2 Example 2

The second security property that we exhibit is expressed informally as: *It should not be possible for an application that doesn't have the Card Terminate privilege to switch the card life cycle state to TERMINATED, whether via a set_status command (if the application is an SD) or via the invocation of the GPSystem.terminateCard method.*

Test Intention A scenario to test the nominal case of failure of this security property can be described informally as:

- select any application without the Card Terminate Privilege;
- set the status of the card to TERMINATED or invoke the GPSystem.terminateCard method.

Test Schemas This scenario gives two schemas in the TestDesigner schema Language. The first one is:

```
for_each $X from any_operation,  
  use $X at_least_once to_reach state_respecting (self.selectedApp.cardTermPriv ≠ true)  
on_instance "card" then  
  use set_status to_reach state_respecting (cardState = TERMINATED) on_instance "card"
```

and the other one is:

```
for_each $X from any_operation,  
  use $X at_least_once to_reach state_respecting (self.selectedApp.cardTermPriv ≠ true)  
on_instance "card" then  
  use GPSystem.terminateCard
```

3.4 Related Works and Originality

Other formalisms have already been used to drive the test generation from a property, or by means of a test purpose. According to these formalisms, the test intentions are expressed either as particular sequencings of the actions of the system (temporal view) or as properties of the data of the system (spatial view).

Temporal logics, such as the Linear Temporal Logic (LTL) [23] allow for specifying state properties w.r.t. several successive moments in the life of the system. Tests can be obtained by means of a model-checker in the shape of traces of a model that contradict the properties (see [14, 2] for example). M. Dwyer, to facilitate the use of temporal properties by validation engineers, has identified in [9] a set of design patterns that allow for expressing as temporal properties a set of temporal requirements frequently met in industrial studies.

Input/Output Labelled Transition System (IOLTS) and Input/Output Symbolic Transition System (IOSTS) have frequently been used to specify test purposes [15, 13]. These formalisms specify sequencing of actions by using the same set of actions as the model, and possess two trap states named *Accept* and *Refuse*. The *Accept* states are used as end states for the test generation while the *Refuse* states allow for cutting the traces not wanted in the generated tests. These formalisms are for example used in tools such as TGV [15], STG [8], TorX [24], Agatha [6].

An approach described in [1] is to generate a trace by model-checking from a model specified as an IOLTS, in which a fault have been injected by a mutation operator, according to a fault model. The trace is then used as a test purpose for TGV.

Some approaches are based on the definition of scenarios for the test, e.g. in [5, 3], where test cases are issued from UML diagrams as a set of trees. The scenarios are extracted by a breadth-first search on the trees. A similar approach is that implemented in the tool *Telling TestStories* [11], based on defining a test model from elementary test sequences made of an initial state, a *test story* and test data.

Let us also cite the Tobias tool [20, 19] that provides a combinatorial unfolding of some test schemas. The schemas are sequences of patterns made of operation calls and parameter constraints. The schemas are unfolded independently from any model, thus the tests obtained have to be instantiated from a model. In [22], a connection between Tobias and the UCASTING tool is studied to produce instantiated tests. UCASTING [26] allows for valuating sequences of operations that are not or only partially instantiated from an UML model.

The originality of our Schema Language with respect to these related approaches can be summarized in three points. Firstly, from a scientific point of view, the language that is described in Section 3.2 makes it possible for the validation engineer to express his or her test schemas by combining both the temporal and spatial views. He or she is allowed to reason in terms of sequences of actions of the system to be called, along with the description by means of predicates of the states to be reached by these sequences of calls. Secondly, from a technological point of view, the language is designed to be easy to use by a validation engineer. He or she writes test schemas in a high level language, in a textual manner, with constructions that are close to usual computer programming paradigms. That frees him from manipulating mathematical notations such as in the temporal logics. Thirdly, by its expressivity, the language is designed as a mean for a validation engineer to describe his test intentions w.r.t. a property that has to be tested. This feature strongly helps to monitor the coverage of the properties to be tested.

The conceptual language of [17] from which the TestDesigner Schema Language originates was designed during a project (RNTL POSE) dedicated to testing the conformance of a system to a security policy. Its conception has been guided by the experience of security practitioners, resulting in a language that well serves the aim of testing security properties. Indeed, considering both actions to perform and states to reach is the way a security engineer thinks of testing a security issue.

As for the SecureChange project, the conceptual language of [17] on which the TestDesigner Schema Language is based, was presented in the deliverable D7.1 as part of the background knowledge of WP7 partners in the field. This new version of the language and its concrete implementation are contributions of WP7 for the SecureChange project, and are implemented into the SBTG (schema-based test generator) component of WP7 Demonstrator (see Section 5.2.3 for the description of the SBTG component). This new version of the language allows a validation engineer to benefit plainly from its good knowledge of the model and to explicitly use all artifacts of the model (such as objects names).

4 Approach for change management in the MBT process

In this section, we present the full process to manage security evolution from a testing perspective. This work is based on previous research described in sections 5 and 6 of deliverable D7.2.

We decompose this chapter in two parts. The first part presents the needed elements to manage evolution into a Model-Based Testing (MBT) approach. We describe the test's life cycle with regards to the system's evolution. We define eight different status for the test's life cycle. The tests are then gathered into four dedicated sets (stagnation, evolution, regression and deletion). Each set is used to improve an aspect of the evolution. The second part presents the extension of the approach in order to take into account the evolution for security properties.

4.1 Impact of evolution on Model

We start this chapter with a recall of MBT in Section 4.1.1 on the test generation process we consider, based on UML model coverage. Then, we present, in Section 4.1.2, the concept of test life cycle in the context of evolving software. We introduce then in Section 4.1.3 the dependency analysis that is used to measure the impact of evolutions on a given test suite. The classification of the tests into various test suites is given in Section 4.1.4.

4.1.1 SecureChange MBT approach

The test generation process that we consider in our approach relies on the use of a formal description of the expected behavior of the SUT using UML diagrams (see Deliverable D7.1 - Section 4, for more details). These diagrams are used to generate tests with the Test Designer (TD) technology, which applies model coverage criteria. First we present the type of diagrams that are taken as input by Test Designer, before explaining how tests are generated.

The Test Generation Process

The Model-Based Testing process we consider in Figure 4.1, starts by the design of the test model (in our case, in UML) by the test architect. Then, the test model is given as an input to the test generator that automatically produces the test cases and the coverage matrix, relating the tests with the covered model elements. The tests are then exported, or published, in a test repository from which test scripts can be adapted. The execution of the tests on the System Under Test (SUT) can be automated: the actions composing each test are associated to a concrete command of the SUT. After the test execution, test results and metrics are provided.

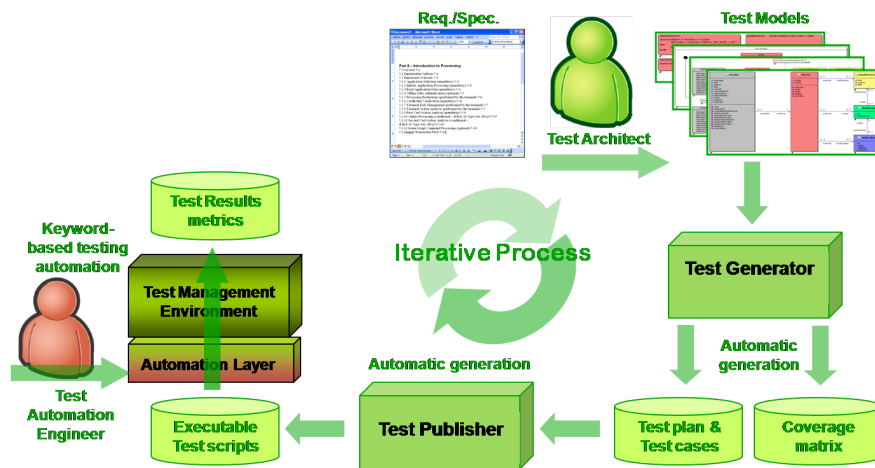


Figure 4.1: The Model-Based Testing Process

Test Designer considers a subset of UML, named UML4ST [7], focusing on three kinds of diagrams. First, a *class diagram* describes the data model, namely the set of classes that represent the entities of the system, with their attributes and operations. Second, an *object diagram* provides a given instantiation of the class diagram together with the test data (i.e. the objects) that will be used as parameters for the operations composing the tests. Finally, the behavior of the system is described by two (complementary) means: a *statechart* diagram, and/or OCL constraints associated to the operations of the class diagram.

The test generation process of TD aims at achieving some given model coverage criterion, according to the test generation presentation made in Section 3.1.2 of Deliverable D7.1. More precisely, TD targets the activation of all the *behaviors* defined in the test model, each behavior being either a transition in the statechart, or a path in the control-flow graph of the OCL post-condition of each operation. In addition, TD relies on the use of tags that annotate the dynamic parts of the model (i.e. transitions of the statechart, OCL code of the operations). We distinguish two kinds of tags: **@AIM** and **@REQ**, both followed by an identifier, issued from the informal specifications of the system (resp. the test targets that the test engineer wants to define).

We see a TD test case as a sequence of steps, in which each step is a call to an operation of the model (described in the class diagram) with a given set of inputs (originating from the object diagram) and expected outputs. Each test covers a given number of requirements (@REQ) and aims (@AIM).

Test Designer allows to automate the computation of the test cases by using dedicated symbolic model state exploration algorithms (see Deliverable D7.1 - Section 4, for more details). Once generated, the tests are classified according to the tags they cover, providing a bi-directional traceability between requirements and generated test cases.

Test Generation and Evolutions

With the current Test Designer technology, the evolution is not taken into account, meaning that an evolution of the system is translated at the model level, and the validation engineer has to regenerate all the tests for the new version of the model. The test repository, which stores the tests that have been produced, is then updated with the new test suite. When facing large models from industrial case studies and if no classification is provided for the test cases regarding the considered evolution, regenerating all the tests can be costly. For

example, as stated by Jiang et al. in [16] complete regeneration of the test suite for the model of Microsoft protocol documentation testing project may take hours or even a full day. Then, test engineers must check all non affected parts, which may take several days, up to several weeks, depending on the model scale.

The objective of our work is to overcome these issues, by first analysing the impacts of the evolutions on the model, and then, classifying the tests so as to be able to distinguish: (i) which parts of the systems did not change (meaning that the tests covering these parts do not need to be regenerated), (ii) which parts of the system did change (meaning that new tests have to be generated specifically for them, or previous versions of the tests have to be modified).

In the next section, we present a solution on how to deal with evolving test cases and we give an overview of our test generation method for evolving systems.

Test Publication

Generated tests are exported into a database to be stored, namely a test repository. The reusability and the future references are important for the maintenance process. We have created a **Smart publisher** for the *TestLink* open-source web-oriented test management system ¹, to organize the test suites and gather test cases information. *TestLink* is a tool for the management and the monitoring of test suites, based on PHP and a MySQL database. Thus, to display the information when updating the repository, we added the panel *Test history*, as shown in Figure. 4.2.

The screenshot shows a web interface for 'Version 2' created on 27/08/2010. It features a 'Summary' section with a 'Test history' box indicating it was 'OUTDATED 2010-08-27 16:55:11'. Below this is a table with columns: Operation, Description, Exigence, and But. The table contains rows for 'setup', 'login', and 'teardown'. The 'login' row has 'ACCOUNT_MNGT/LOG' under Description and 'LOG_Invalid_User_Name' under But. Below the table are sections for 'Execution type: Automated', 'Keywords: None', and 'Requirements: [Exigences fonctionnelles] ACCOUNT_MNGT/LOG:ACCOUNT_MNGT/LOG'. At the bottom, there is an 'Attached files' section showing a file 'login (f2-56-ff).bsh' with a size of 454 bytes, a text/bsh type, and a date of 08/03/2010. An 'Upload new file' button is also present.

Operation	Description	Exigence	But
setup			
login	ACCOUNT_MNGT/LOG		LOG_Invalid_User_Name
teardown			

Figure 4.2: Smart publisher

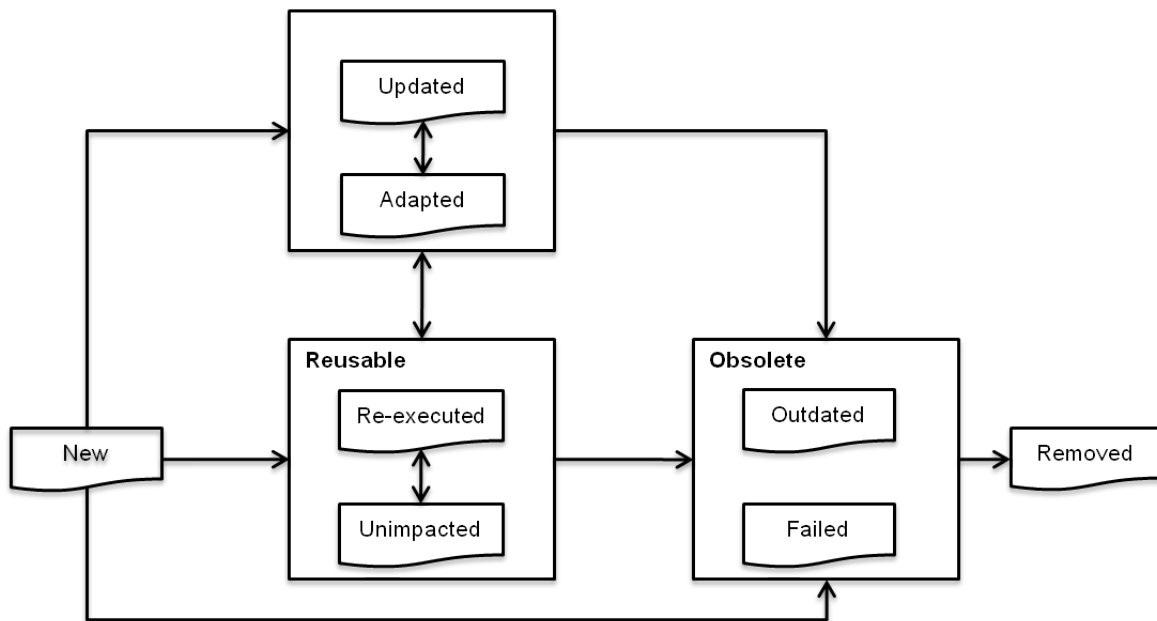
4.1.2 Evolution of Test Cases

We present in this part the notion of evolving test cases, in which all the tests are versioned and associated to a given status. Then, we introduce the method on which we rely in order to obtain the test status.

Test Status

In the context of evolving systems and, thus, evolving test cases, we associate to each test a *status*, that indicates its state in the life-cycle depicted in Figure 4.3. Evolution of

¹<http://www.teamst.org>



status is defined by considering two versions of the model, \mathcal{M} and \mathcal{M}' , in which addition, modification or deletion of model elements (operations, behaviors, transitions, etc.) have been performed. As explained in Section 4.1.1, each test is dedicated to a given test target, that can be either the activation of a transition in the statechart, or the coverage of a behavior extracted from the OCL code of an operation. Evolution of the test status may be defined in \mathcal{M}'' if the validation engineer decide to remove invalid tests from the repository.

Test may have a status **new** in case of a newly generated test for a newly introduced target. At the first test generation stage of a project, all generated tests have the status new.

When an evolution occurs, the state of the test changes depending on the impact of the evolutions. If none of the model elements covered by the test are impacted, it is ready to be run as is on the new version of the model \mathcal{M}' , without modifying the test sequence. The test is thus said to be **reusable**. More precisely, there are two cases: unimpacted and re-executed. It is **unimpacted** if the test sequence is identical to its previous version, and the covered requirements still exist. The test is **re-executed** if it covers impacted model elements, but it can still be animated on the new version of the model without any modification (the expected outputs –on which the oracle is based– are still the same).

If a test covers model elements impacted by the evolution from \mathcal{M} to \mathcal{M}' , and if the test cannot be animated on \mathcal{M}' the test becomes **obsolete**. There are two cases: either the test target represents deleted model elements, and thus the test does not make any sense on \mathcal{M}' and it is said to be **outdated**, or, the test fails when animated on model \mathcal{M}' (e.g. due to a modification of the system behaviour), it is then **failed**. When the test case operations can be animated but produce different outputs, a new version of the test is created in which the expected outputs (i.e. the oracle) are updated w.r.t. \mathcal{M}' . In this case the tests have the status **updated**. When the test case operations can not be animated as is in the first version of the test, a new operation sequence has to be computed to cover the test target. In the latter case, tests have status **adapted**.

In order to discard invalid tests for \mathcal{M}'' , the validation engineer has the possibility to

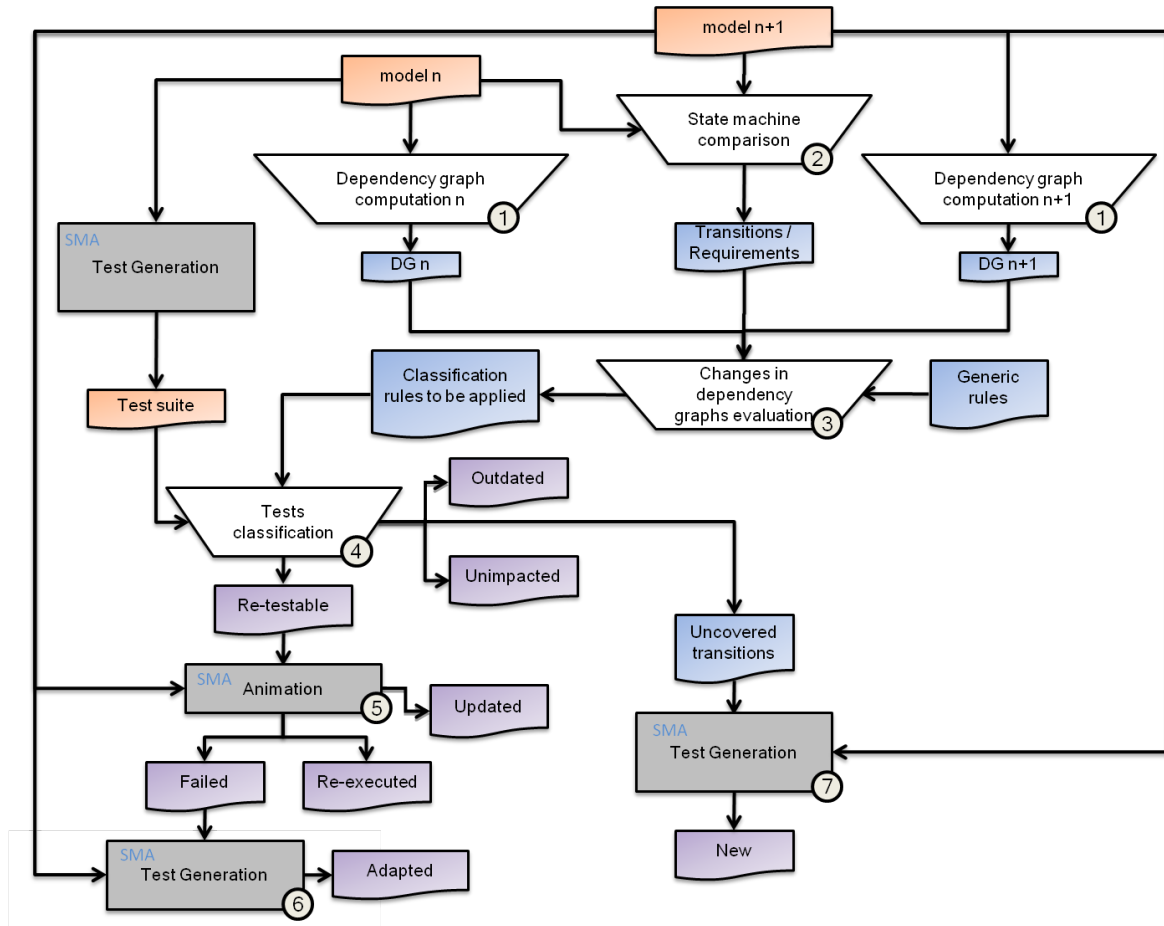


Figure 4.4: Process to determine the status of the tests

change the status of **outdated** and **failed** tests into **removed**. These tests are irrelevant for the version \mathcal{M}'' , because they cover already changed or disappeared elements for \mathcal{M}' and thus, it has no more sense to consider them for \mathcal{M}'' .

Definition 1 (Evolving Test Cases) An evolving test case tc^n , where n is the version of the model, is characterized by a pair $\langle tc, status \rangle$ where tc is the test case and $status$ is its associated status: $status \in \{new, updated, adapted, unimpacted, re-executed, failed, outdated, removed\}$. We will denote $status(tc^n)$ the status associated to the evolving test case tc^n .

Test Status Assignment Process

Now, we introduce the process that will be used to determine the status of a test from a version of the model to another. This process is depicted in Figure 4.4 and relies on the use of a high-level analysis of the impacted model elements on the statecharts. It aims at assigning a status to each test, possibly creating additional tests to cover added or modified targets.

As illustrated on Figure 4.4, the process is always based on two test models (the original one (n) and the evolved one ($n + 1$)) and a test suite from the original model, produced from the model-based test generation using the *Test Designer* technology. We first use a dependency analysis, tag ① on the figure, in order to produce the dependency graphs (DG) for both models that indicate how the model elements depend from each other. Then, we

compare the state machines, tag ②, to identify the transitions or requirements that have been impacted between the two versions. DG and modified elements are put together in step ③ which consists in evaluating the dependencies on the impacted elements. Considering this information and the test suite originally created from model n , it is possible to do a preliminary classification of the tests, tag ④, splitting them into three categories: **outdated**, **unimpacted**, and an intermediate category named *re-testable*. Each re-testable test is then animated, tag ⑤, on the new version of the model ($n + 1$). When a test is animated and produces the same expected outputs, its status is set to **re-executed**. When the operations of the tests can be animated on model ($n + 1$) but produce different outputs, the test case is **updated**. When the test can not be animated, because one of the operations can not be activated at a given step, it is assigned a **failed** status. The test sequence is then modified, tag ⑥, so as to reach the considered target (if it still exists in model ($n + 1$)). The resulting test is **adapted**.

In addition, with the test classification step it is possible to identify parts of model ($n + 1$) that have not been covered by tests, corresponding to new elements that did not exist at version n . In this case, the uncovered transitions of the model ($n + 1$) become test targets for the test generation process, tag ⑦ on the figure.

Notice that for another evolved model ($n+2$) it does not make sense to keep **outdated** and **failed** tests, so the validation engineer can choose to set them as **removed** and drop them into the **Deletion test suite**.

We now give more details of the upstream part of the process, namely, the dependency analysis that is performed to compute the differences between the models, and their impacts on test cases.

4.1.3 New selective test generation method

The selective test generation guides us to select tests from the model before the change according to different techniques. This work is based on research done on impact analysis based on dependence results for UML/OCL statechart diagrams in Deliverable 7.2 and in [12]. The data dependency algorithm is based on the pairs of definitions and uses of variables in a graph, respecting the property specifying the absence of redefinition of variable between a pair definition/use. The control dependence algorithm is based on the property that one execution of transition in the graph depends from another execution. We consider that when two transitions are data dependent, then a test covering the behavior of these transitions may exist. If two transitions are control dependent, then a test exists and it covers their behaviors.

We guide the selective test generation, with the impact analysis results. Thus, our goal here is not only to guide the regression testing, but also to manage the test suites after evolution.

From the test model, a state diagram and its dependency graph are used for test selection. Transitions of the statechart represent only one behavior of a given operation expressed using OCL (see Section 4.1.1). An evolution of the system can be unique or made of several modifications at the same time. We consider three types of elementary modifications to represent evolution: addition, modification, or deletion of a transition for which rules are created separately. By "modification of a transition", we mean a modification of the OCL code (in the guard/effect of the transition or in the pre/post condition of its associated operation). The composition of these actions is called a *complex modification* and the dependency analysis is done separately for each component [25].

For each elementary modification, we can have *creation* or *deletion* of *data* and/or *control*

dependencies between two transitions. According to the changes in the dependency graph, we classify or animate tests in order to gather them in the correct test suites. Thus, we are able to add new tests, as well as to delete, modify existing ones or reuse them (see Section 4.1.2).

Applying this technique, we are able to verify that: (i) evolved parts behave as we expected, (ii) evolved parts did not affect the unchanged parts and (iii) what is changed in the model has been actually changed in the implementation.

4.1.4 Evolution in Test Suites

We describe in this section the composition of test suites we consider in order to test the evolutions of the system. We classify the tests into four test suites: Evolution, Regression, Stagnation and Deletion, whose specificities are now explained.

Test Suites

We consider four kind of generated test suites, each one having a specific purpose. They are denoted with Γ_X , where Γ is the notation for a test suite and X is its type. We give here their names and an informal description of their purposes.

Evolution test suite. Γ_E contains tests exercising the novelties of the system (new requirements, new operations, new behaviors etc.).

Regression test suite. Γ_R contains tests exercising the unmodified parts of the system. These tests aim at ensuring that the evolutions did not impact parts of the SUT that were not supposed to be modified. The particularity of the tests contained in Γ_R is that they have been computed from a former version of the model, that is prior to the current model version.

Stagnation test suite. Γ_S contains invalid tests w.r.t. the current version of the system. These tests aim at ensuring that the evolution did actually take place and changed the behaviour of the system. Notice that, these tests have been computed from a former version of the model and contrary to regression tests, they should be invalid for the current version. They are expected to fail when executed on the SUT (either because they cannot be executed, or because they detect a non-conformance of the SUT w.r.t. the expected results).

Deletion test suite. Γ_D contains tests, that come from the Stagnation test suite from the previous version of the model. Moreover, as they are obsolete for the ancient version, they may be considered as irrelevant for the current one. Thus the test engineer has the possibility to dump them in the *Deletion* test suite.

We now describe how the test suites are filled w.r.t evolutions and test life cycle. This description takes into account the test status defined in Section 4.1.2. Each test suite contains a set of tests for a given version of the system. Our definitions rely on test cases tc evolving from a version n to a version $n + 1$.

Composition of the Test Suites

We present here the rules that are used to distribute the test cases into the three kinds of test suites. Figure 4.5 depicts how the tests are gathered in the respective test suites.

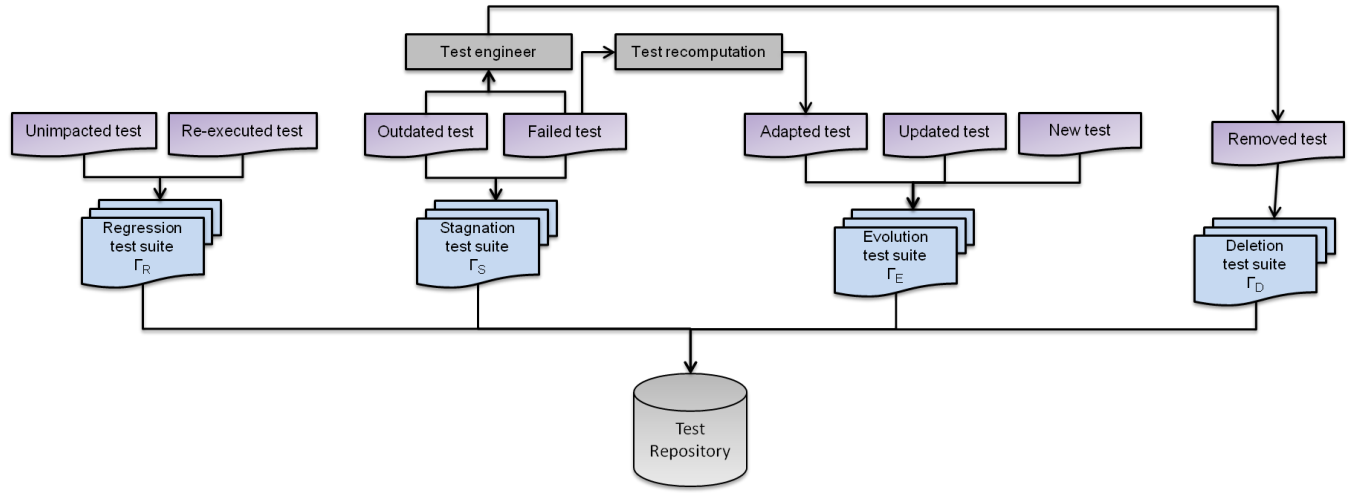


Figure 4.5: Test suites composition

Basically, evolution is addressed by new and adapted tests, regression is addressed by reusable tests, and, stagnation is addressed by outdated tests along with previous versions of the now, adapted tests (the failed tests).

We provide here the rules, based on the test status, that are used to compute the composition of versions $n + 1$ of the test suites.

Rule 1 (New tests) *A new test exists only at tc^{n+1} version. All new tests are added in the Evolution Test Suite.*

$$status(tc^{n+1}) = new \rightsquigarrow tc^{n+1} \in \Gamma_E^{n+1}$$

Rule 2 (Reusable tests) *A reusable test (either unimpacted or re-executed) comes from an existing test suite $tc^n \in \Gamma_E^n \cup \Gamma_R^n$ and it is unchanged $tc^{n+1} = tc^n$. All these reusable tests are added in the Regression Test Suite.*

$$status(tc^{n+1}) \in \{unimpacted, re - executed\} \rightsquigarrow tc^{n+1} \in \Gamma_R^{n+1}$$

Rule 3 (Obsolete tests) *An obsolete test comes from an existing test suite (possibly obsolete) $tc^n \in \Gamma_E^n \cup \Gamma_R^n \cup \Gamma_S^n$. All tests that have been declared as obsolete are added in the Stagnation Test Suite.*

$$status(tc^{n+1}) = \{outdated, failed\} \rightsquigarrow tc^{n+1} \in \Gamma_S^{n+1}$$

Notice that the *failed* tests have also been recomputed to be used as *adapted* for the same version.

Rule 4 (Updated tests) *An updated test comes from an existing test suite $tc^n \in \Gamma_E^n \cup \Gamma_R^n$. All tests which results have been updated are added in the Evolution Test Suite.*

$$status(tc^{n+1}) = updated \rightsquigarrow tc^{n+1} \in \Gamma_E^{n+1}$$

Rule 5 (Adapted tests) *An adapted test comes from an existing test suite $tc^n \in \Gamma_E^n \cup \Gamma_R^n$. All tests that have been adapted are added in the Evolution Test Suite.*

$$status(tc^{n+1}) = adapted \rightsquigarrow tc^{n+1} \in \Gamma_E^{n+1}$$

Notice that the previous versions (*failed* tests) are added in the Stagnation Test Suite.

Rule 6 (Removed tests) *A removed test comes from the Stagnation test suite $tc^{n+1} \in \Gamma_S^{n+1}$. All tests that have been declared as removed by the validation engineer in $(n+2)$ are added in the Deletion Test suite.*

$$status(tc^{n+2}) = \{removed\} \rightsquigarrow tc^{n+2} \in \Gamma_D^{n+2}$$

We now detail how the SeTGaM is used for computing the impact of evolution when testing security properties in a system by the mean of test schemas (see Chapter 3).

4.2 Impact of evolution for security properties

In this section we present how to manage the impact analysis when testing security properties. On one side, in the MBT process, we create test models to cover requirements originating from the system specification. On the other side, we have the security properties expressed in textual form, used to design **test schemas** that capture the test intentions for each security property, with a formal language (see Chapter 3).

Using the test schemas and the data given in the test model, it is possible to automatically unfold schemas into several Test Case Specifications (TCS), as described in a previous part 3.2. Then TCS are sent to the test generator to automatically generate the corresponding tests, as presented on Figure 4.6. An experienced testing engineer can create complex and powerful test schemas to test security properties. By powerful, we mean here that the number of TCS obtained from a test schema may be very high.

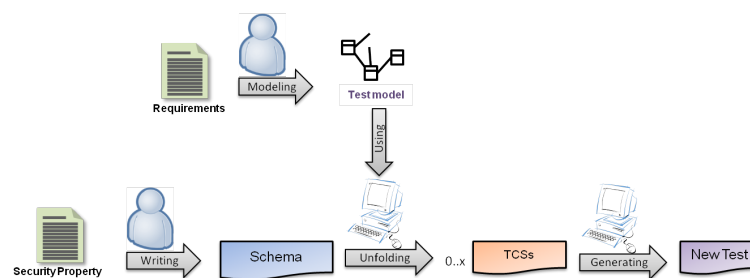


Figure 4.6: Process for testing security properties

4.2.1 Evolution of schemas and requirements

In the evolution process, we can consider three kinds of changes: (i) the test schema can evolve, (ii) the requirement, and thus the model, can evolve and we can use the same test schemas for security testing or (iii) they both can evolve. A schema modification leads to the addition or the removal of the existing one. Thus, we consider three status of schemas:

- Unchanged
- New
- Deleted

As depicted on Figure 4.7 a schema with *New* status leads to generation of a new test cases.

The *Deleted* status, implies the removal of the associated test cases. This set of actions can be done using the traceability feature. For an *Unchanged* schema, if an evolution occurs in the model, we gather the associated tests in a test suite and we then apply the SeTGaM process as described in Section 4.1.

4.2.2 Evolution in Test Suites with respect to Security Testing

In this section, we detail the composition of test suites that are considered for evolution management in security testing. We classify tests into four test suites: Evolution, Regression, Stagnation and Deletion Test Suite (see Figure 4.5). This latter contains all **removed** test

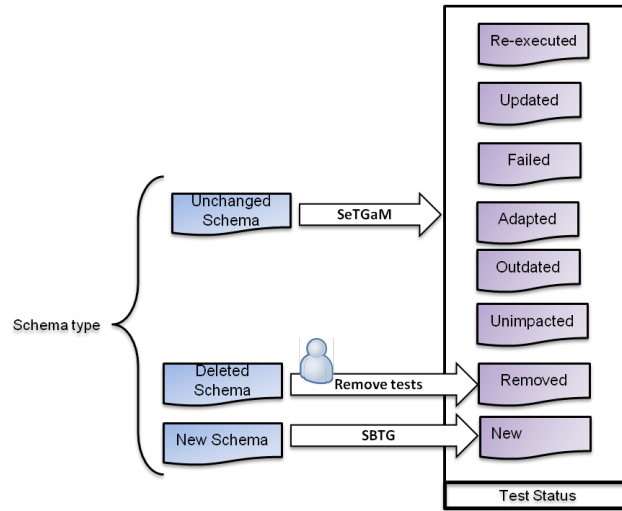


Figure 4.7: Process for testing security properties with respect to evolution

cases, that have no more sense to keep them in the history, as described in Section 4.2.1. The approach for management evolution for security properties extends the definition of **removed tests**, to test that may come from a deleted schema, as given in the rule below.

Rule 7 (Removed tests - extension) *In the process of security properties evolution we consider that the associated schema is deleted and then we create a new one. Using the traceability link between schema and tests, the test engineer can gather corresponding tests and set their life cycle to removed. Thus, we give here the extended definition of this type, that a test can be set to **removed** as result of schema deletion. All tests that have been declared as removed are placed into the Deletion Test Suite of the current version.*

$$status(tc^n) = \{removed\} \rightsquigarrow tc^n \in \Gamma_D^n$$

5 Demonstrator

This section presents the demonstrator associated to this deliverable. The developments associated with the methodological process are described in the following sections. First, the conceptual architecture of this research prototype is detailed in Section 5.1. Then, the SBTG (schema-based test generator) component and some new features for structuring the test model provided for the SecureChange project are presented in Section 5.2. Finally user interface screen captures are shown in Section 5.3.

5.1 Architecture

The architecture of the demonstrator is divided into two components that we describe in the following sections.

5.1.1 SeTGaM

This section introduces the conceptual view of the SeTGaM process and internal components. They are divided into an impact analyzer and a test classification algorithm. Components are shown in Figure 5.1. The impact analyzer uses two versions of a model

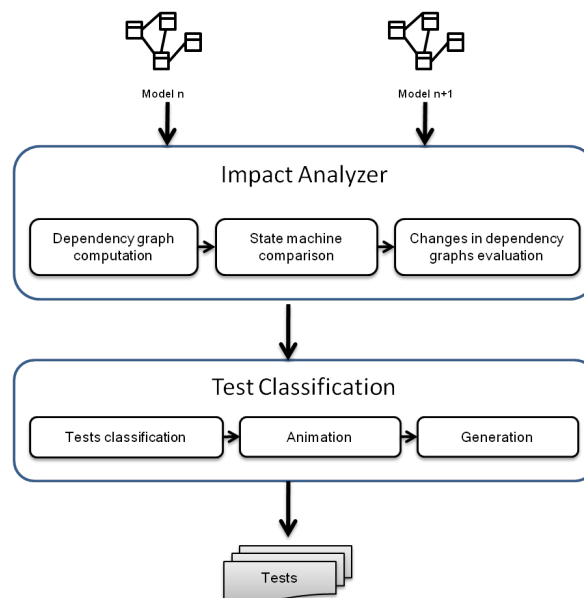


Figure 5.1: SeTGaM overview

(i) to compute a dependency graph for each version, (ii) to compare state machines and (iii) to evaluate changes between versions. This analyzer produces a file that will be used

by the test classification algorithm. This process is described in details in Section 4.1.3.

Figure 5.2 shows the activity diagram for the demonstrator process (for an initial and updated version of the test model). This process has two test models as inputs: `in_model_n` and `in_model_n+1`.

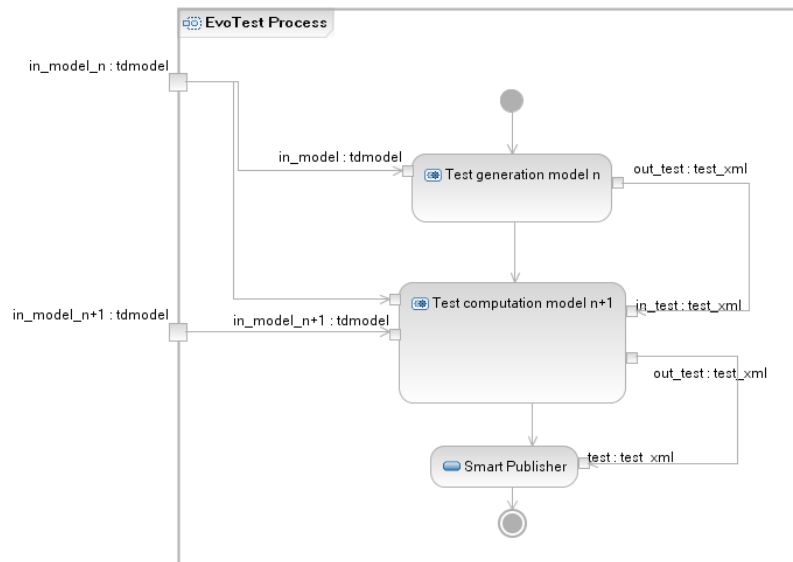


Figure 5.2: The process activity diagram

The first activity, depicted in Figure 5.3, is the initial test cases generation. Test cases are sequences of actions associated to an expected result after the test execution on the SUT. The test model of the system is used by Test Designer to compute test cases. As explained previously, test cases are generated by covering paths in the state machine. A test verdict is computed based on the OCL code collected on the path.

The next step of our process is the test cases computation for the newest model, also called SeTGaM. This activity takes as inputs the two test models and the list of tests produced by the previous tests generation. As a result, the activity produces a new list of tests. The list of tests is given in an XML file.

The SeTGaM process is presented in Figure 5.4. The first step is the transformation of our test model files into files with specific format that are used internally. The transformation allows the process to be as independent as possible of the file format used as input. Then, they are used as parameters for the state machine comparison that produces an XML structure containing all detected changes in the models.

Concurrently, control and data dependencies are computed for each version of the model. The results of this processing are two XML structures containing the current detected dependencies.

To summarize, the produced structures by the previous steps are:

- changes in the models,
- data and control dependencies for model `n`,

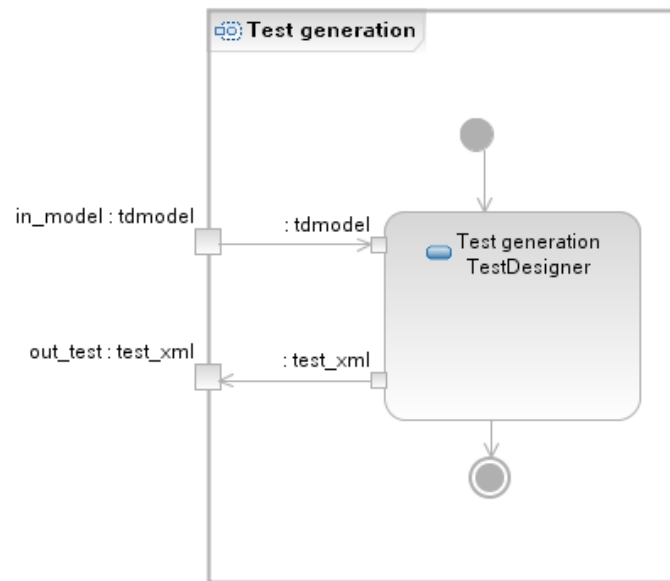


Figure 5.3: Test cases generation process

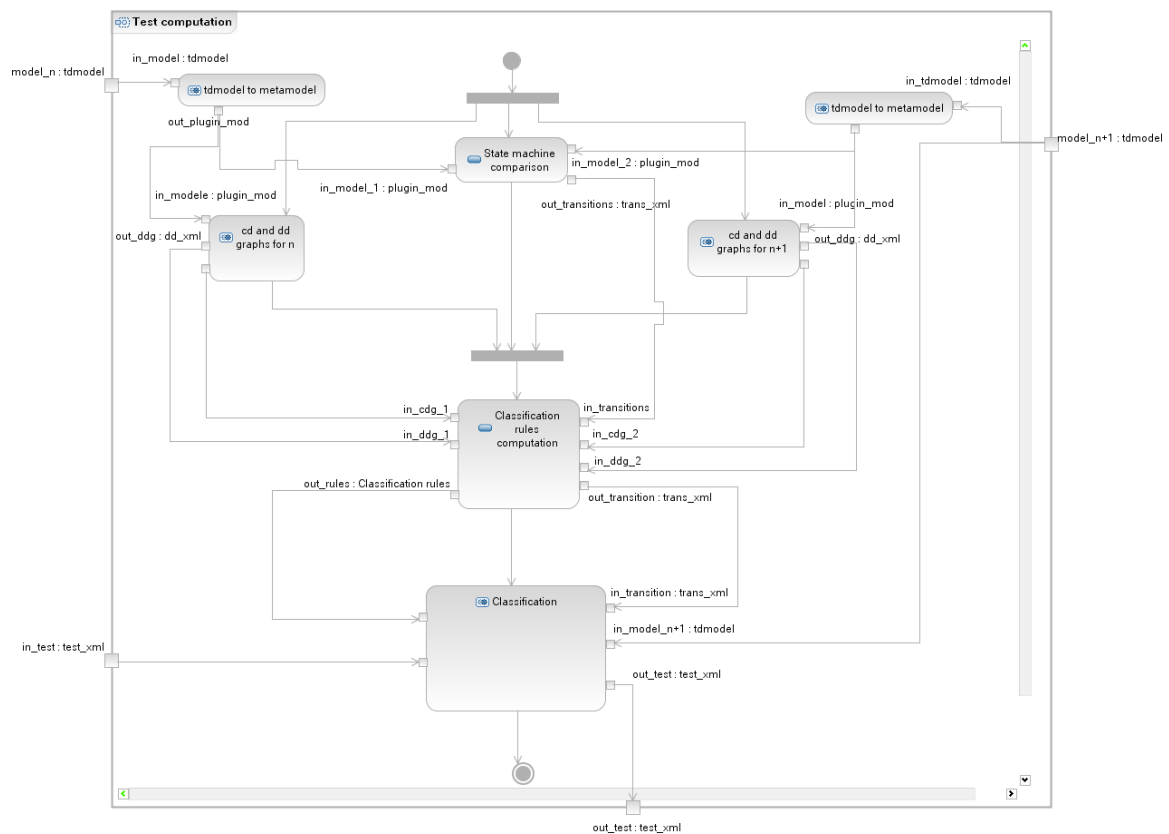


Figure 5.4: The SeTGaM activity diagram

- data and control dependencies for model $n+1$.

They are the input for the test classification activity. This activity will assign a status to each detected change used in the next activity: the actual classification.

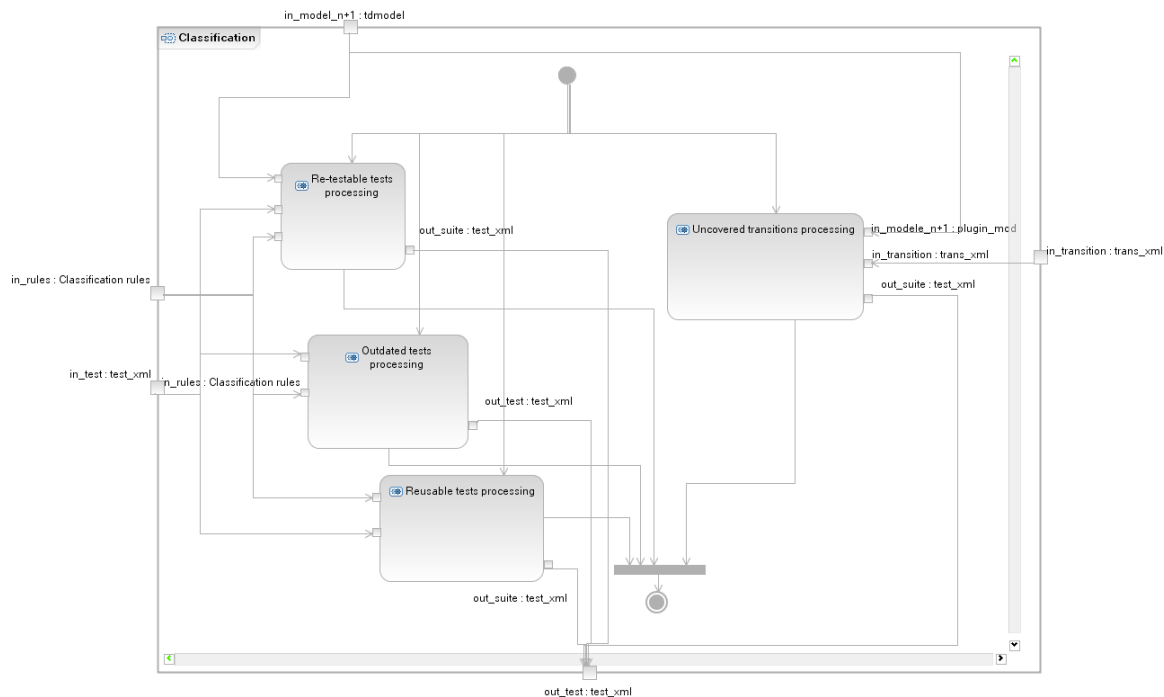


Figure 5.5: Classification activity diagram

The classification, as shown in Figure 5.5, deals with four main processes:

- Re-testable tests processing,
- Outdated tests processing,
- Reusable tests processing,
- Uncovered transitions processing.

For these activities, the new version of the model is used with the list of previously generated tests and detected changes in models. The result is a new XML file used by the Smart Publisher component to update the test repository, which contains details about tests lifecycle.

The most complex activity is the re-testable tests processing, depicted in Figure 5.6. Based on the classification rules provided by the previous activity and the list of tests produced by the previous generation, the re-testable tests are identified. The test animation component, presented in Section 5.2, is used to define if a test can be re-executed as is or if it needs to be regenerated. That is done by the generation component.

Outdated and reusable tests are processed in the same way by creating a new test suite, filled by the identified tests.

Finally, uncovered transitions are used by the test generation component to produce new tests. These tests are then added to a new dedicated test suite. All test suites previously created from existing or new tests are written in a new file that is used by the following step

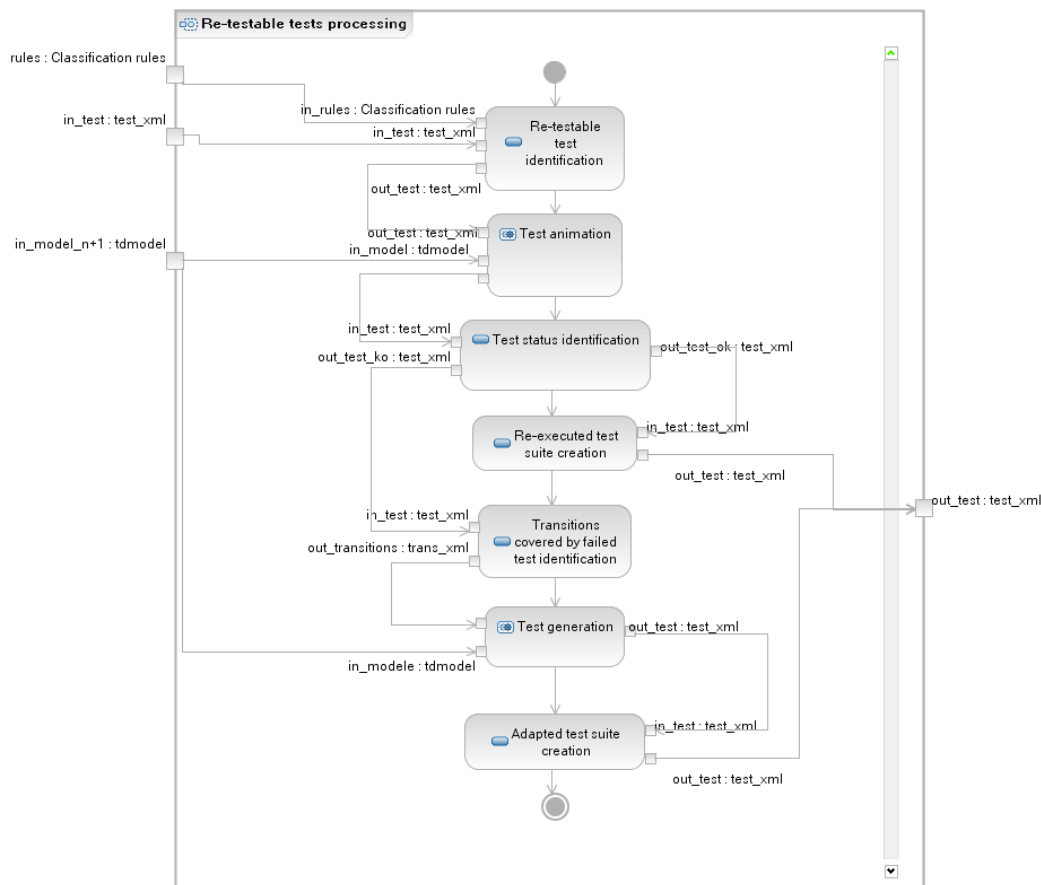


Figure 5.6: Re-testable tests processing

of the process.

We have presented in this section a set of conceptual elements that were used to guide the development of the SetGaM component sub-part of the deliverable. The next component is the Smart Publisher that takes the tests and stores them in a test repository.

5.1.2 Smart Publisher

The generated tests resulting from the previous step will be used to upgrade a test repository. This tool is useful for a test engineer to manage test cases. It is also used to track test execution on the implementation and to store results of this execution. A link is also possible with a bug tracker to give some feedbacks to the development team. For the SecureChange project, we use an Open Source test repository called TestLink in version 1.8. TestLink is a GPL licensed tool that manages test cases organized in test plans. After a test execution on the system's implementation, the results (failures or successes) are stored and published in execution reports. They are used to check the requirements coverage.

TestLink is an independent tool with a PHP based interface using a MySQL database that can be linked to bug tracker tools (such as Bugzilla).

The first test publishing in TestLink produces a new project in the test repository. In the root folder, two folders are created: one for functional tests and the other for security tests.

This structure will remain stable throughout the following test publishing. The Smart Publisher creates several folders based on the SeTGaM method (Evolution, Stagnation and Regression) inside the functional tests folder which can contain several other folders based on the model structure.

During a system life, the test engineer produces several test models based on the evolution of a given specification. A new publishing is done for each version after a test generation. Its impact should be as small as possible because it is a real challenge for the test engineer to keep track of test cases. To do so, the test repository structure should not change and each change should be traceable. For each publication if a test has new status according to SeTGaM then the current version of the test is deactivated in the test repository and its previous status is stored with the previous execution results. If a test moves into a new test suite then the suite is stored in a history panel, helping the test engineer to keep track of the previous position of the test.

5.2 Test generation improvements

After the test classification, in order to provide new test suites corresponding to the new model version, SetGaM uses two components provided by Smartesting that make it possible to animate the test sequences on the model, and to generate new tests if needed. We further present the SBTG (schema-based test generator) component in this section, which is the demonstrator tool chain implementing Test Generation for Security Properties and Test Schema Management.

5.2.1 Model animation API

The Smartesting animation component enables to play sequences of valued operations on the test model, in order to compute the expecting results and check what behaviours represented in the model have been covered by the sequence. At the beginning of the project, this component was only accessible from the Test Designer GUI as shown in Figure 5.7.

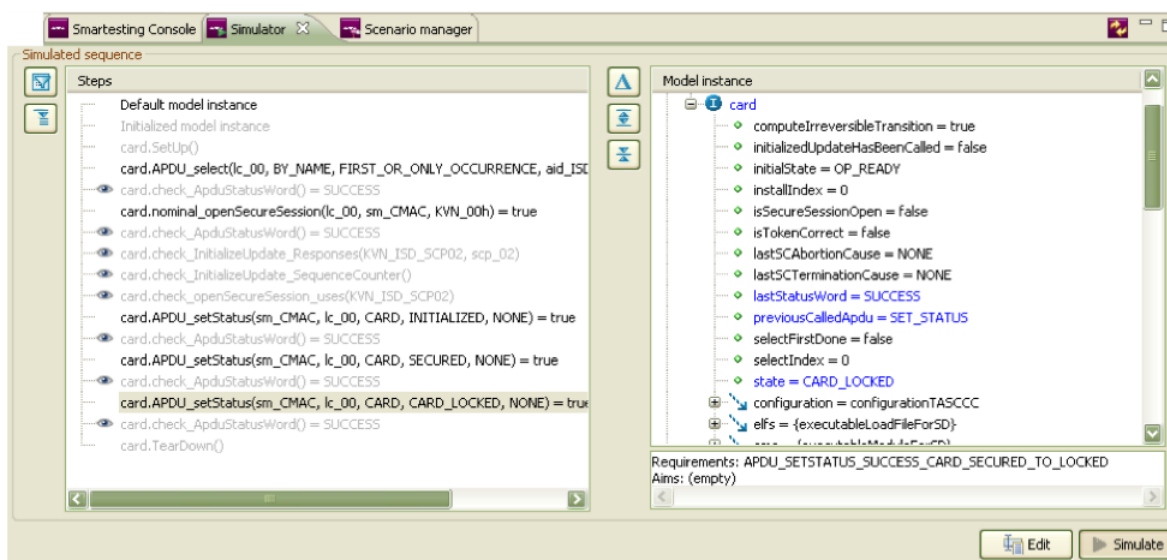


Figure 5.7: Smartesting Test Designer GUI

SeTGaM has its own GUI (see Section 5.3), that is independent from the Smartesting Test Designer tool. A Java API has been created in order to enable the animation process to be used by SeTGaM, without opening the Test Designer tool. It is now a standalone Java library.

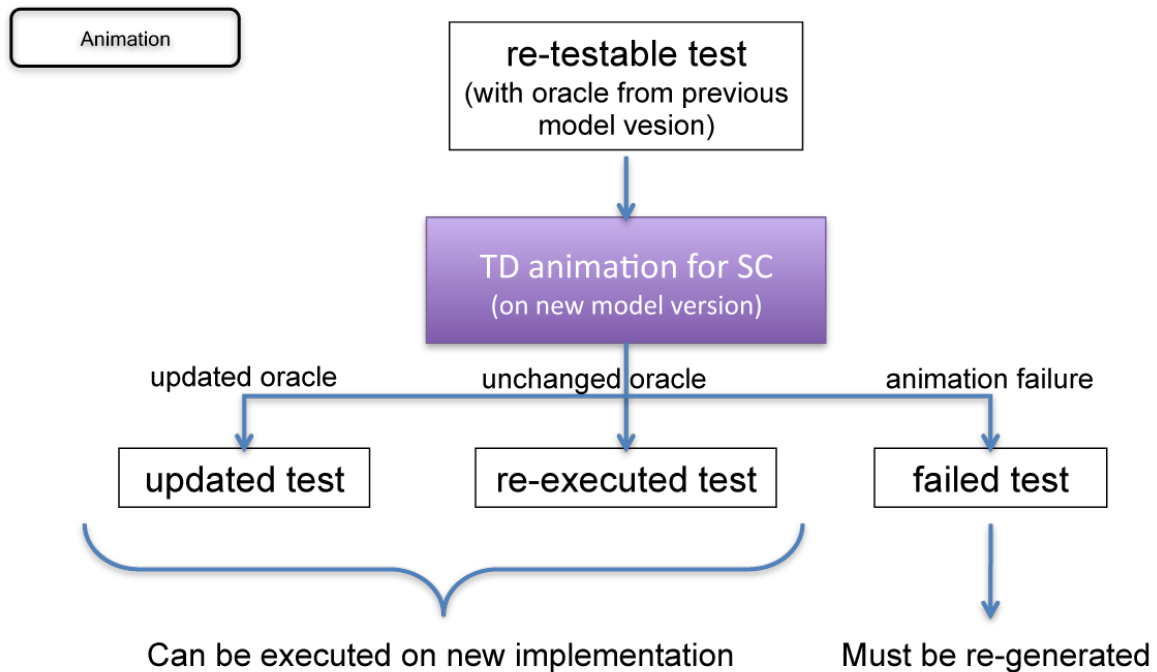


Figure 5.8: Smartesting Test Designer Animation API usage

This library enables SeTGaM to classify the re-testable tests (as shown in Figure 5.8).

- Each re-testable test is animated on the new model version.
- Each re-testable test life cycle is updated. It can be :
 - **re-executed** if the animation of the sequence is equivalent to the previous test
 - **failed** if the animation of the sequence fails on the new test model. A new test must be generated to replace this sequence.
 - **updated** if the animation of the sequence succeeds, but expected results must be updated
- If needed the re-testable test expected results are updated

In addition to the actual Smartesting Test Designer animation engine, specific features have been developed for the needs of the Secure Change project. These features are:

- the capability to easily **compare** the animation result **with a previous generated test**
- the capability to **identify the changes** between an animated sequence on **two different model versions** (used for test classification). At each step of the test, different changes can be identified:
 - the **expected result** after the step
 - the **requirements** covered by the step

5.2.2 Test generation API

The generation component allows to create tests sequences that cover specific behaviours (transitions) represented into the test model. As for the model animation, the test generation feature of Smartesting was only accessible from the Test Designer GUI.

A Java API has been created in order to enable the generation process to be used by SetGaM. It is used at different steps of the SetGaM process as shown on Figure 5.9 :

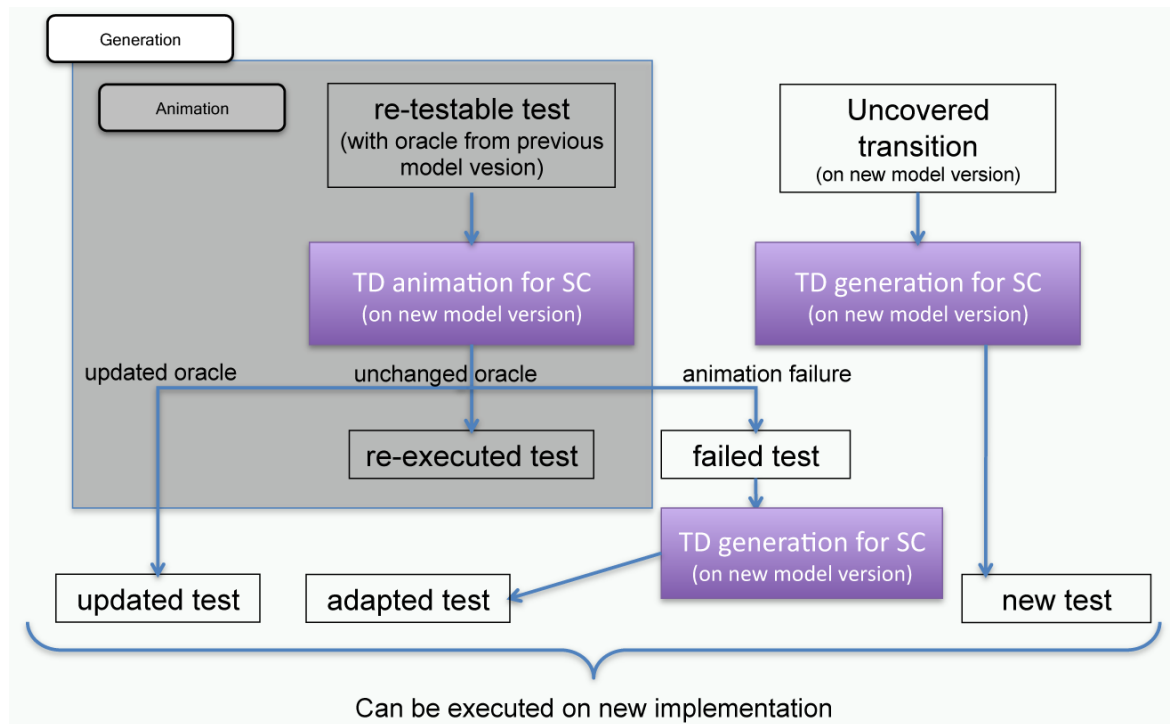


Figure 5.9: Smartesting Test Designer Generation API usage

- It is used to generate a test for each uncovered transition of a given model version. The new test is classified as **new**.
- It also makes it possible to generate a new test for each test that failed during the animation process. This test must cover the same transition as the previous test version. The new test is classified as **adapted**.

5.2.3 Schema-based test generator

For the Secure Change project, one goal is to be able to generate tests from Security Properties. This is the role of the schema-based test generator (SBTG) component. This component is a research prototype developed to extend Test Designer generation engine. The new engine allows the Test Generation based on Test Schemas, that are created from the Security Properties (see Section 3.3). A new Test Schema editor has been created as well as new test generation facilities.

SBTG

Initially, Test Designer focuses only on the functional test generation based on a structural coverage of the test model (mainly behavioural coverage). For instance, on the POPS case

study, the test generation engine was able to generate a test that covered

- *"terminate the card with setStatus command"*

But because of the need to test from Security Properties (see Section 3), the ability to generate a test case that covers a specific sequence of several functional behaviours and/or states has been implemented. The new engine is now able to generate a test that covers:

- *"lock the card"*
- THEN *"unlock the card"*
- THEN *"terminate the card with setStatus command"*

The main effort for the SBTG component was to improve the generation engine in that way. In order to take into account the SBTG component into the Smartesting Test Designer software, the internal architecture has been deeply changed. This represented a new challenge that was necessary because the treatment of Test Schemas for the Test Generation.

Test Schema treatment for Test Generation

A test Schema is not used as is to generate Test Cases. It must be pre-processed to create Test Objectives that will be used by the new generator engine to create the Test Cases. For each Test Schema, several test cases can be generated. For instance, consider the following Test Schema:

```
for_each $X from setStatus or storeData or installForInstall,  
use any_operation at_least_once to_reach "selectedApp.cardTermPriv = true" then  
  use set_status to_reach "cardState = TERMINATED" then  
    use $X
```

Here, one test will be generated for each operation of the model that can be used to instantiate the "X" variable, so three tests will be generated. Those tests must respectively cover the following sequences:

```
use any_operation at_least_once to_reach "selectedApp.cardTermPriv = true" then  
use set_status to_reach state_respecting "cardState = TERMINATED"then  
use setStatus
```

```
use any_operation at_least_once to_reach "selectedApp.cardTermPriv = true" then  
use set_status to_reach state_respecting "cardState = TERMINATED"then  
use storeData
```

```
use any_operation at_least_once to_reach "selectedApp.cardTermPriv = true" then  
use set_status to_reach state_respecting "cardState = TERMINATED"then  
use installForInstall
```

These 3 test objectives are sent to the new Test Designer generation engine, that produces the following Test Cases:

```

Test 1: card.nominal_openSecureSession(lc_00, sm_CMAC, KVN_00h)
- SUCCESS
card.nominal_setUpISDKeys()
- SUCCESS
card.APDU_setStatus(sm_CMAC, lc_00, CARD, INITIALIZED, aid_ISD)
- SUCCESS
card.nominal_setUpCASDandVASD()
- SUCCESS
card.APDU_setStatus(sm_CMAC, lc_00, CARD, SECURED, aid_ISD)
- SUCCESS
card.APDU_setStatus(sm_CMAC, lc_00, CARD, TERMINATED, aid_ISD)
- SUCCESS
card.APDU_setStatus(sm_CMAC, lc_00, CARD, SECURED, aid_ISD)
- ERROR_CARD_TERMINATED

```

```

Test 2: card.nominal_openSecureSession(lc_00, sm_CMAC, KVN_00h)
- SUCCESS
card.nominal_setUpISDKeys()
- SUCCESS
card.APDU_setStatus(sm_CMAC, lc_00, CARD, INITIALIZED, aid_ISD)
- SUCCESS
card.nominal_setUpCASDandVASD()
- SUCCESS
card.APDU_setStatus(sm_CMAC, lc_00, CARD, SECURED, aid_ISD)
- SUCCESS
card.APDU_setStatus(sm_CMAC, lc_00, CARD, TERMINATED, aid_ISD)
- SUCCESS
card.APDU_storeData()
- ERROR_CARD_TERMINATED

```

```

Test 3: card.nominal_openSecureSession(lc_00, sm_CMAC, KVN_00h)
- SUCCESS
card.nominal_setUpISDKeys()
- SUCCESS
card.APDU_setStatus(sm_CMAC, lc_00, CARD, INITIALIZED, aid_ISD)
- SUCCESS
card.nominal_setUpCASDandVASD()
- SUCCESS
card.APDU_setStatus(sm_CMAC, lc_00, CARD, SECURED, aid_ISD)
- SUCCESS
card.APDU_setStatus(sm_CMAC, lc_00, CARD, TERMINATED, aid_ISD)
- SUCCESS
card.APDU_installForInstall()
- ERROR_CARD_TERMINATED

```

Those tests cases can be executed on the system under test. They are automatically generated from the Test Model and the Test Schemas, that are manually edited from Security Properties.

Test Schema editor

In order to be able to create the Test Schema, an editor has been created. It is an Eclipse plugin as shown on Figure 5.10.

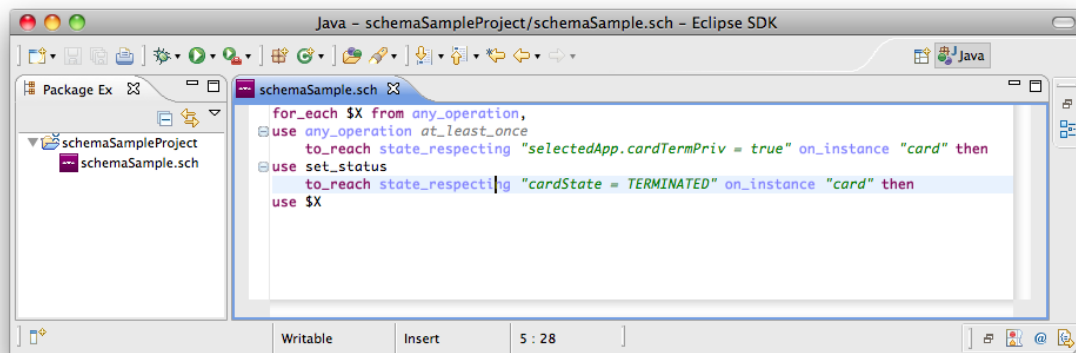


Figure 5.10: Smartesting Schema Editor GUI

This editor offers syntax highlighting and code completion for Test Schema files (with the ".sch" extension). As a schema file is saved, it can be used as it to generate the tests.

5.3 Graphical User Interface

This section describes the demonstrator implementation. An Eclipse stand-alone application called *EvoTest* was developed to provide a graphical interface to the project. Application is based on IBM Rational Software Architect (RSA) UML models. This modeler is used for test models design. Figure 5.11 shows the EvoTest panel and its components which are:

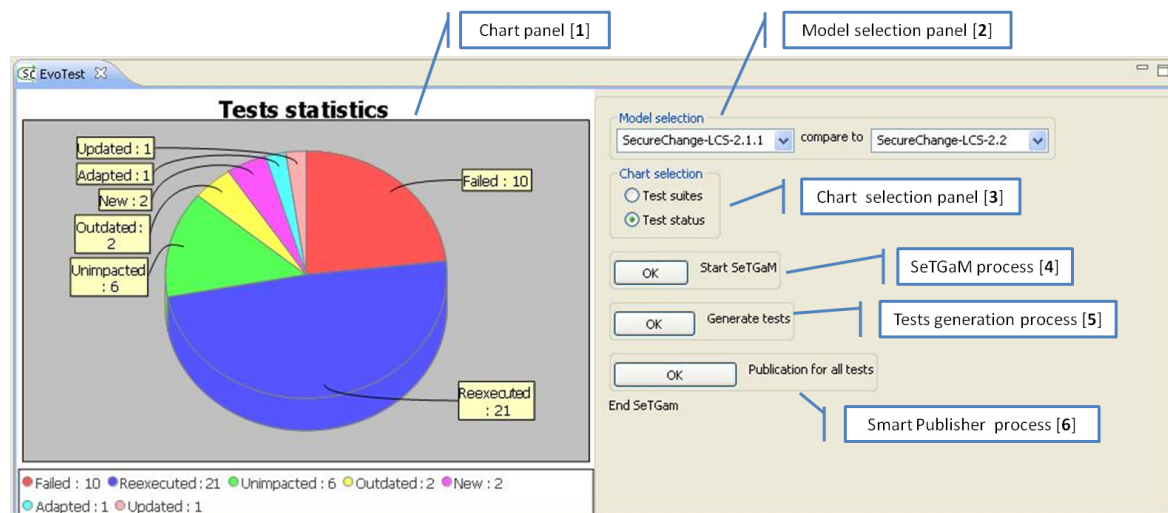


Figure 5.11: The EvoTest panel

- A chart panel containing a pie chart or a bar chart for test suites or test status
- A model selection panel to select two model version to use with SeTGaM

- A chart selection panel to configure the chart panel
- A button to start the SeTGaM process
- A button to start the test generation process
- A button to start the test publication in the test repository

These components are detailed in the next section.

5.3.1 Chart panel

This panel is only available after the SeTGaM process is called. It shows a graphical representation of the test suites composition.

5.3.2 Model selection panel

This panel is composed of two combo-boxes to select two test model versions. All available and opened UML models are filtered and can be selected. These fields are mandatory to start the SetGaM process. The process uses XML files with **.tdmodel** extension produced by a Smartesting Export plugin.

5.3.3 Chart selection panel

There are two charts available:

- Test suites and
- Test status

The **test suite** view is a bar chart (Figure 5.12) showing the number of test created in each test suite type (Evolution, Stagnation and Regression).

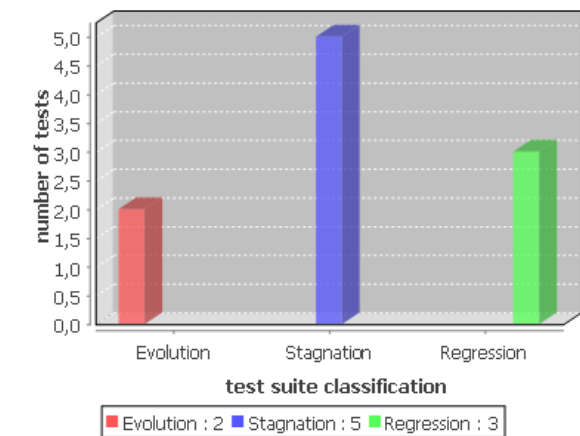


Figure 5.12: Test suites chart panel

The **test status** chart offers a pie chart (Figure 5.13) of the available tests for each status defined by the SeTGaM process (see Section 5.1.1). It gives an overview of what was done by the process, and how many tests will be generated. We now describe the behavior of the three buttons available on the EvoTest.

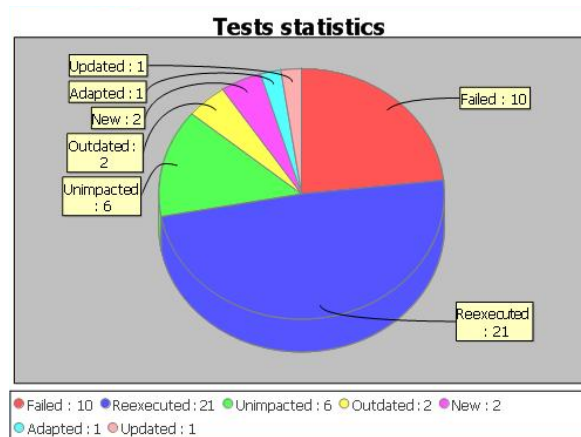


Figure 5.13: Test status chart panel

5.3.4 SeTGaM process

The button starts the first part of the process, described in Section 5.1.1 and stop after the tests classification. The chart panel is then updated to reflect the results obtained by SeTGaM.

5.3.5 Test generation process


This button starts the process that generates tests with Failed or New status after the SeTGaM process. They must be re-generated by the Smartesting component presented in Section 5.2.2. The generation process is not included in SeTGaM to let the user choose whether or not he wants to generate.

5.3.6 Test publication

The final step of the process is the publication in a test repository. The publisher is a java library called by the EvoTest.

The component uses two XML files as parameters. The first file contains the publisher configuration, such as database name and user identification. The second file is the XML file produced by SeTGaM and containing tests. Figure 5.14 shows the tool interface. On the left panel we can see the structure of the project. The root folder is the model used for testing. Then we can see the functional test suite (extracted from the test model) containing the three test suites introduced by SeTGaM. Each folder contains tests that can be loaded in the right panel.

This panel contains a *history* panel that traces the status of the selected test during the previous publication. This is an iterative process so the status changes during the tests life cycle.



TestLink 1.8 RC2 : admin [admin]

Test Project
CardL

[Home](#) | [Specification](#) | [Execute](#) | [Results](#) | [Users](#) | [Events](#) | Test Case ID: | [-documentation-](#) | [Personal](#) | [Logout](#)

Navigator - Test Specification

Filter & Settings ...

Test Suite:

Update tree after every operation ☒

Update tree

- CardLifeCycle-22 (4)
 - OPREADY(4)
 - Evolution(0)
 - Stagnation(2)
 - Car-1.lcs_APDU_setStatus (60-13-6f)
 - Car-1.lcs_APDU_setStatus (60-79-71)
 - Regression(2)
 - Car-1.lcs_APDU_setStatus (60-1f-59)
 - Car-1.lcs_APDU_setStatus (60-d9-cf)

Test Case

Car-1.lcs_APDU_setStatus (60-79-71)

[Edit](#) [Delete](#) [Move / Copy](#) [Delete this version](#) [Create a new version](#) [Deactivate this version](#) [Export](#)

Version 6

Created on 04/10/2010 16:21:54 by admin

Summary

Test history

OUTDATED 2010-10-04 14:47:34
OUTDATED 2010-10-04 16:08:14
OUTDATED 2010-10-04 16:08:27

BODY

Opération	Description	Exigence	But
sm_APDU_initializeUpdate			
lcs_APDU_setStatus		APDU_SETSTATUS_SUCCESS_SD_CARD_OP_READY_TO_INITIALIZED	
ccm_APDU_installForInstallAndMakeSelectable			
lcs_APDU_setStatus		APDU_SETSTATUS_SUCCESS_SD_CARD_INITIALIZED_TO_SECURED	
cm_APDU_selectPartialAid			
lcs_APDU_setStatus		APDU_SETSTATUS_SUCCESS_SD_CARD_SECURED_TO_LOCKED	
lcs_APDU_setStatus		APDU_SETSTATUS_ERROR_CARD_MUST_BE_ISD	FROM_CARD_LOCKED

Steps

Expected Results

Execution type : Automated

Keywords : None

Requirements : None

Attached files :

lcs_APDU_setStatus (60-79-71).bsh - lcs_APDU_setStatus (60-79-71).bsh (2102 bytes, text/bsh) 13/09/2010

Upload new file

Other versions

Figure 5.14: TestLink tool

6 Example

In this chapter, we describe the use of the presented SeTGaM approach on a simple case study and we give a comparison with two other possible approaches for test generation when dealing with evolution.

6.1 Example

In this section, we describe the running example based on an *eCommerce* application *eCinema*. First, we give a general description of the application and its model. Next, we present the generated tests for the model with Smartesting *TestDesigner*.

6.1.1 General description of eCinema application

The **eCinema** is an application aiming at booking movie tickets. The UML class diagram we have designed contains the objects managed by the application: eCinema, movies, tickets and users. The *eCinema* class models the system under test (SUT) and provides the API operations offered by the application (see Figure 6.1).

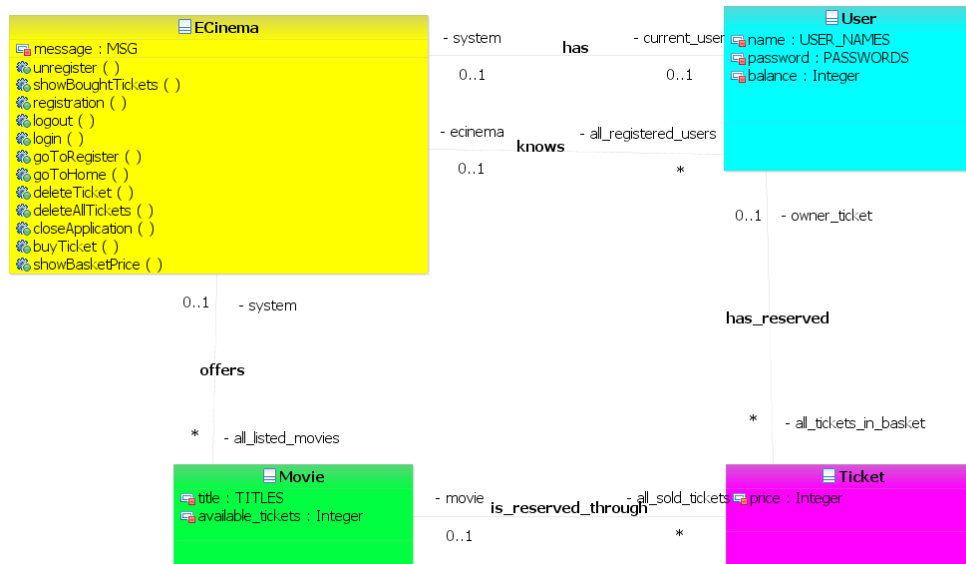


Figure 6.1: Class diagram of eCinema

This case study allows to users to buy cinema tickets, to display bought tickets for movies as well as to subscribe and log in to the application. Several requirements have been identified, such as: (1) the user must be registered and connected to access the proposed

services, (2) after registration level, the user may deposits money on its cinema account, (3) the registration is valid only if the user's name and password are valid and if the user is not already registered, (4) the user must be connected in order to buy tickets, (5) the user can display all selected tickets for movies, (6) the user can delete one, several or even all of the cinema tickets, (7) cinema tickets are prices according to cinema's decision board and the price is the same for each movie.

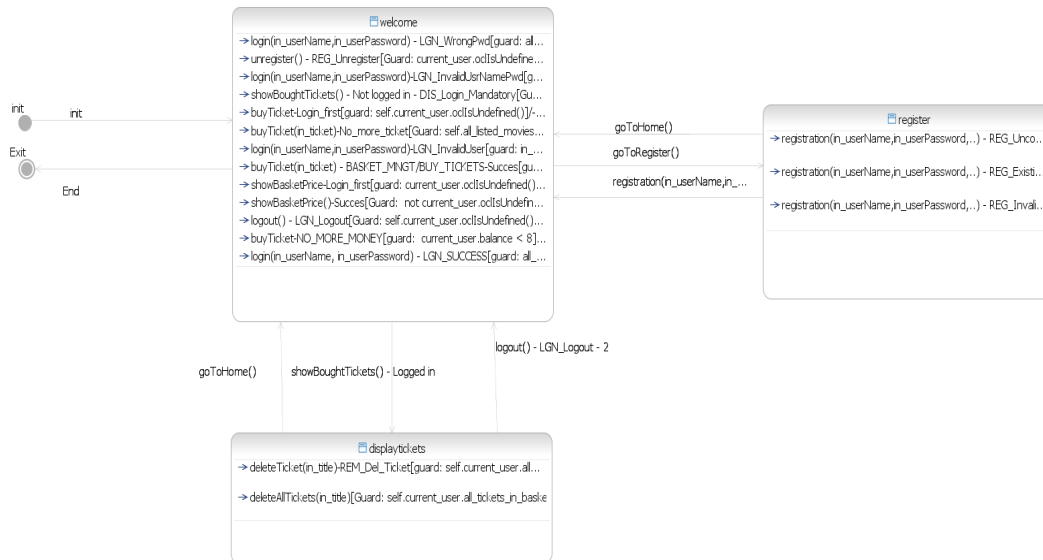


Figure 6.2: Statechart diagram of eCinema.

The SetGaM methodology is applicable on behaviors, which are expressed by using OCL code in guards or actions in transitions of an UML statechart (see Figure 6.2). You can find the list of transitions of the eCinema statechart in Appendix C. Each transition name starts by the event operation, followed by a short name of the transition's performed behavior.

We tag requirements by using the keyword **REQ**. Each requirement is separated in several behaviors tagged by the keyword **AIM**. The test target is one behavior of a requirement expressed by OCL code in a transition. To manage traceability between the requirements and the behaviour on one side and the test targets on the other side, each test target is represented by a couple of **REQ/AIM**. For *eCinema* we have identified **seven** requirements and **twenty-five** test targets (see Section 4.1.1 for more details on this method). For the list of these elements that are covered in *eCinema*, refer to Appendix A.

6.1.2 Test Generation with *TestDesigner* for eCinema

As depicted on Figure 2.1 first we use the Smartesting TestDesigner technology to compute test cases by covering paths and behaviors in the statechart, which are expressed by actions on transitions. The test generation process of Test Designer will target the coverage of the path of the model and/or the behaviors of the OCL constraints describing operation. For example, the following test case can be produced, aiming at testing the successful buying of ticket, or the buyTicket command.

```
ECinema::sut.login(REGISTERED_USER, REGISTERED_PWD);
```

```
ECinema::sut.buyTicket(TITLE1);
```

This test case covers the test target: @REQ: BASKET_MNGT/BUY_TICKETS and @AIM: BUY_Success of the `buyTicket` command (see Appendix A), by activating the action of the transition `buyTicket(in_ticket) - BASKET_MNGT/BUY_TICKETS`.

Notice that this test case activated also the action of the transition `login(in_userName, in_userPassword) - LGN_SUCCESS` and thus it covers also the test target: @REQ: ACCOUNT_MNGT/LOG and @AIM: LOG_Success of the `login` operation.

For our model, to cover all the targets, 20 tests were computed. Each test is described by the name of the target operation and in parenthesis the associated test's id is written. Generated tests for *eCinema* are listed below:

- * buyTicket (f2-31-6d), buyTicket (f2-36-46), buyTicket (f2-fc-81)
- * closeApplication (f2-bc-1c)
- * deleteAllTickets (f2-3d-8a)
- * deleteTicket (f2-65-d1)
- * goToHome (f2-09-fa), goToHome (f2-0b-a0)
- * login (f2-cd-a5), login (f2-e1-71), login (f2-fc-c8)
- * logout (f2-9b-b1), logout (f2-ab-dd)
- * registration (f2-1e-13), registration (f2-26-df), registration (f2-e2-64)
- * showBasketPrice (f2-32-8a), showBasketPrice (f2-e8-51)
- * showBoughtTickets (f2-6a-c2)
- * unregister (f2-06-f5)

6.2 Evolution of the eCinema's system

In this section we describe changes applied to the *eCinema* application, the impacts on the model and then how we have applied SetGaM methodology to produce new test suite.

6.2.1 Changes in requirements

For the study, we have made an evolution of the system and added the notion of cinema subscriptions for users (e.g. for senior costumers) and cinema's account management.

More precisely, for this evolution, the following changes are introduced: (i) at the registration level, the user chooses its type of subscription, (ii) a subscription can be changed at any moment, (iii) the proposed subscriptions are: child, student, normal, senior and there is one free ticket for ten tickets bought, (iv) ticket fees are linked to the selected subscription, and (v) the user can manage its cinema account (see Figure 6.3). The user's subscription and the account management are modeled with a newly added operations.

Each transition represents one requirement's behaviour and we add a pair of **REQ/AIM** to each transition. When taking into account the system's evolution, **two** new states: *BalanceManagement* and *SubscriptionManagement* (see Figure 6.4), **twenty-four** new transitions were added to the statechart, **seven** were modified and **two** were deleted (detailed in Appendix D).

To illustrate the approach we are going to take into account each type of change: deletion, addition and modification of transition and thus of test target.

Remark: To make the reading and the comprehension of SeTGaM approach on the example easier, we define short names for the test targets used in the description below:

- @REQ: BASKET_MNGT/BUY_TICKETS | @AIM: BUY_Success is replaced by *Buy_success*
- @REQ: BASKET_MNGT/BUY_TICKETS | @AIM: BUY_Success/NORMAL is replaced by *Buy_normal_Subscription*
- @REQ: BASKET_MNGT/BUY_TICKETS | @AIM: BUY_NO_MORE_MONEY is replaced by *Buy_no_more_money*
- @REQ: BASKET_MNGT/BUY_TICKETS | @AIM: BUY_Sold_Out is replaced by *Buy_no_more_ticket*
- @REQ: BASKET_MNGT/REMOVE_TICKETS | @AIM: REM_Del_Ticket is replaced by *Delete_ticket*
- @REQ: BASKET_MNGT/REMOVE_TICKETS | @AIM: REM_Del_All_Tickets is replaced by *Delete_all_tickets*
- @REQ: ACCOUNT_MNGT/REGISTRATION | @AIM: REG_Unregister is replaced by *Unregister_success*
- @REQ: ACCOUNT_MNGT/REGISTRATION | @AIM: REG_Success is replaced by *Register_success*

Deletion : with the model comparison we have found that there are two deleted test targets: *Delete_ticket* and *Buy_success*. For each deleted target we classify as **outdated** the corresponding tests, given below:

- deleteTicket (f2-65-d1)
- buyTicket (f2-31-6d)

Each modification may induce changes in activated paths in the statechart, thus one state can become inaccessible from another and some tests can become irrelevant. When taking into account dependencies in both models we can say which elements and tests are impacted by the deletion. For instance, the *Delete_ticket* target affects the following behaviors:

- Buy_success
- Buy_no_more_money
- Buy_no_more_ticket
- Delete_all_tickets

Intuitively, we want to check tests for each target and animate them to obtain a verdict. If we take a look at the made evolution (see Appendix B), we can conclude that in this case, *Buy_success* and *Buy_no_more_money* are already part of the evolution i.e *Buy_success* is deleted and *Buy_no_more_money* is modified. To decide about the tests, that they cover,

in this case we apply rules for deletion and modification respectively (for further information see also paragraph *Modification*).

When using SeTGaM we may face the problem that one test covers several targets that refer to a evolved transition. For example, in this case, the test *buyTicket (f2-31-6d)* is covering the target *Buy_no_more_ticket*, nevertheless it covers the deleted *Buy_success*. Also, the test *deleteAllTickets (f2-3d-8a)*, which covers the target *Delete_all_tickets*, covers also the deleted *Buy_success*. Both tests are already classified as **outdated**. The problem that appears here is that the targets: *Buy_no_more_ticket* and *Delete_all_tickets* are no longer covered by any test.

To respond this problem, the approach takes into account that we may loose tests covering existing test targets and thus requirements. So, at the end of the process, if needed, tests are generated for all uncovered targets.

Addition: We have added twenty-four new test targets in the new model. To illustrate the method when addition of test targets occurs, we take as sample one added test target, for instance *Buy_normal_Subscription*.

As for removed targets, we are interested in discovering affected elements and select corresponding tests. Then, we need to symbolically animate selected tests and obtain a verdict about their validity. In our case, the following targets are impacted by the added *Buy_normal_Subscription*:

- *Buy_no_more_money*
- *Buy_no_more_ticket*
- *Delete_all_tickets*
- *Unregister_success*

For added test target, the process SeTGaM needs to deal with tests that cover several targets. One test might have been already animated. It has no sense to repeat the action, the verdict is the same. Thus, tests covering: *Buy_no_more_money*, *Buy_no_more_ticket*, *Delete_all_tickets* are not selected again.

The test *unregister (f2-06-f5)* covering the tagret *Unregister_success* is then selected as **retestable**. The symbolic animation process did not made any updates on the test, thus it is classified as **Re-executed**.

Modification: We illustrate the process on the test target *Register_Success*. Its corresponding action is modified, because the registration must take into account users with particular subscriptions. According to our results we give below the impacted elements:

- *Unregister_success*
- *Delete_all_tickets*

In this case we need to select tests for the modified target and for each impacted element. Similarly to previous cases, tests for *Unregister_success* and *Delete_all_tickets* are already selected.

The test *buyTicket (f2-36-46)*, covers the modified target *Register_success*. The existing test consists in registering a new user, with initially no credits on the account and then the user tries to buy a ticket. The operation *register* has an additional parameter, to specify the subscription. It is impossible to animate the test sequence as it is, so the test execution fails

and it is classified as **failed**. To cover the target we needed to create another test, that we classify it as **adapted**.

The selective testing process for the other changes is very similar to the given samples. That is why we are not going to give separate details about each modification.

We have previously stated that we can identify impacted elements when using data and control dependencies and make a reference to the table of changed elements in the model (see Appendix B). But, how are we going to select elements that may perform actions differently than expected, after model's evolution? To solve the problem, we are comparing data and control dependencies and we separate deleted or added dependencies, that do not concern directly changed targets. Then, we check and classify tests for each dependent element, according to the obtained verdict issued from the symbolic test animation.

We have minimized the number of new generated tests to **nineteen**, to cover the new and uncovered requirements and test targets. We have animated the selected tests on the new version: **four** have failed and **eleven** have passed. The failed tests, because they cover still existing behaviors, had to be adapted. Thus, we have produced **four** adapted tests using the test generator.

In this section we have illustrated the process of SeTGaM approach on eCinema. In the next section we introduce the evolution management for security properties on the running example.

6.3 Security Properties Testing on eCinema

The system has to satisfy the following security property:

Security Property 1: If the cinema user's database is empty, the only accessible operation is *goToRegister*.

Indeed, if this property was to be violated due to an erroneous implementation of the system, some unauthorized accesses could possibly be granted and remain undetected, since no user is supposed to access the system. This property deals with access control.

Schema 1:

```
for_each $X from any_operation_but goToRegister,
use any_operation any_number_of_times
  to_reach state_respecting "self.all_registered_users->size() > 1" on_instance "sut"
then
use any_operation any_number_of_times
  to_reach state_respecting "self.all_registered_users->size() = 0" on_instance "sut"
then
use $X
```

In order to verify the security property for the running example, using **SBTG** we have created **eleven** tests, one per operation different from *goToRegister*, which end with an error result code, because it is not authorized.

The property and the schema are the same for the eCinema model and its evolution. In order to check the property for the evolved model we are going to consider tests, unfolded from the schema, as a test suite for the model. Then, we classify tests using the SeTGaM

approach and benefit from its results, but for security properties testing. For all, outdated or failed tests, if needed to generate more tests to check the security property, the **SBTG** is used.

6.4 Comparison of SeTGaM with two other approaches

We now compare the SeTGaM method proposed in this deliverable with other test generation techniques for managing evolutions: retest-all and regenerate-all. Retest-all is a commonly used regression testing technique which consists in re-executing all tests from the old test suite on the evolved model and then generate tests for uncovered requirements' behaviors. The regenerate-all technique regenerates tests for each behavior of the changed model without any impact analysis on the evolution wrt the test model. We denote the first evolution as model (n) and the second as model ($n+1$), formalizing the expected behaviors, corresponding to a total of 25 and 47 test targets, respectively.

Status	Test Life Cycle	Retest-all	Regenerate-all	SeTGaM
Invalid tests	Outdated	7	-	3
	Failed		-	4
Total invalid	-	7	-	7
Valid tests	Re-executed	14	-	11
	Updated		-	-
	Unimpacted	-	-	2
	Adapted	-	-	4
	New	22	36	19
Total valid	-	36	36	36

Table 6.1: Number of tests for eCinema - evolution using the Retests-all, Regenerate-all and SeTGaM methods

Results, illustrated in Table 6.1, show that the two other methods lose an important information during testing. The retest-all method considers only three categories of tests: obsolete (outdated and failed), reusable (re-executed and unimpacted) and new (new and adapted). So, it is not possible to deduce their origin (e.g. if the test is reusable because it covers an unmodified part of the system or because it is not impacted by the change). Moreover, our method makes it possible to know at any moment the origin of the test and its classification in the corresponding test suite. In addition, as shown in Figure 4.5, we gather tests into Stagnation, Evolution, Regression and Deletion test suite. Thus, we respectively ensure that the evolutions did really take place, that system novelties are correctly implemented, and that the evolution did not impact parts of the system that were not supposed to be modified.

7 Telling TestStories: Another Point of View

This section explains another view for managing evolution in the MBT process based on the scenario-based approach *Telling TestStories* (TTS). The advancements for this second test approach is part of WP7 T7.3 results, but not part of WP7 Demonstrator.

The basics of the TTS approach, and associated concepts for evolution, are described in the previous WP7 deliverable (see Section 6.4 of Deliverable D7.2).

The anticipated results we aim for, is that the testing workflow is optimized in such a way that after an evolution step only the relevant test cases have to be re-executed to ensure proper system functionality, instead of the whole set of defined test cases. Furthermore, test cases are organized in a way that gives information about the expected result of their test runs.

Since TTS is not aiming at automatic generation of tests, but is a framework for their *manual* definition, every evolution step, regardless of its origin, is reflected in the model and an indication to the respective stakeholder is given when manual intervention is likely required.

The input of TTS is a system model and a set of requirements. Since TTS is a framework for testing service-centric systems, the system model is supposed to describe these services and their interfaces. Besides the static structure, a system model can further describe the behavioral aspects of the system, e.g., the activity sequence of certain use cases. The input by the system's model and the requirements is used by a test designer to model *test stories*.

7.1 Evolutions

Generally, the evolution of certain parts of the system and the propagation to affected parts is covered by the SecureChange Integrated Process. Here, we shortly describe how TTS deals with different kinds of evolution. The TTS framework provides the facilities for traceable associations among tests and requirements, and among tests and system model elements.

7.1.1 Requirements Evolution

Requirements can be subdivided into functional and non-functional requirements. Every test story is always associated to at least one requirement. When a requirement is *changed*, the respective tests can be determined by this association. Since the primary focus of TTS is not the test generation, a test engineer may need to review the corresponding tests and adapt them according to the changed requirements. When a new requirement is *added*, new tests have to be defined by the test engineer. Also if a requirement is *removed*, the connection to test stories is used to determine affected tests. In this case, the impacted tests can be used as negative tests to validate that the functionality is not available anymore.

7.1.2 Evolution of the System or Environment

Also the system or the environment may evolve over time. Evolutions of the system are normally triggered by the requirements if the evolution step was planned. In this case, the impact on the tests is determined as described in the previous Section 4.1.3. On the other hand, it can also happen, that an unplanned change is observed at the system, e.g., the redeployment of a running service because a new version is available. In this case, the corresponding tests are determined, once more, by the association among system model elements and test model elements. Depending on the modification, the status of the affected tests changes so that the tests are either re-executed in the next test run, or marked as being not executable.

Evolutions to the environment are very similar to evolutions of the system. Such typical evolution step is for instance an update of underlying software, e.g., the operating system. If the system model captures the elements of the system impacted by this change implied the identification of impacted tests.

7.2 Methods and Techniques

To handle change in TTS, we attach a state machine to each changeable artefact, that defines its actual state, and triggers resp. receives events to compute new states. Changes in requirements are indicated by triggers on requirement elements and changes of the infrastructure or the system are indicated by triggers on service elements. Consequently, tests have a state. Following the widely used classification in [21], the *type of a test* can be *evolution*, for testing novelties of the system, *regression*, for testing non-modified parts and ensuring that evolution did not unintentionally take place on other parts, *stagnation*, for ensuring that evolution did actually take place and changed the behavior of the system, and *discard*, for tests which are not relevant any more. Based on this test type, which is computed by states of model elements and a test requirement, a test suite for regression testing is determined.

In the following, we first explain the underlying metamodel, and then the overall evolution process and its core process action of change propagation.

7.2.1 Telling TestStories Metamodel

The metamodel is depicted in Figure 7.1. The package `Test` defines all elements needed for system testing service centric systems. A `TestSuite` is a collection of `Test` elements. It has a number of `TestRequirement` elements, e.g., to select tests or define test exit criteria, and a number of `TestRun` elements – assigning `Verdict` values to assertions – attached. A `Test` has a `Type`, which can be either *evolution*, *stagnation*, *regression*, or *deletion*, and consists of `SequenceElement` artifacts. A `SequenceElement` is either an `Assertion` defining how a verdict is computed or a `Call` element invoking a service operation. It has some data assigned to the free variables of assertions or calls attached.

7.2.2 Test Life Cycle

Telling TestStories implements the same test suites as described in deliverable D7.2. These are *evolution*, *regression*, *stagnation*, and additionally *deletion*. As mentioned in the description of the metamodel, the element `Type` determines to which test suite a test belongs. In this section we explain the different states a test can have, and how the test suite is

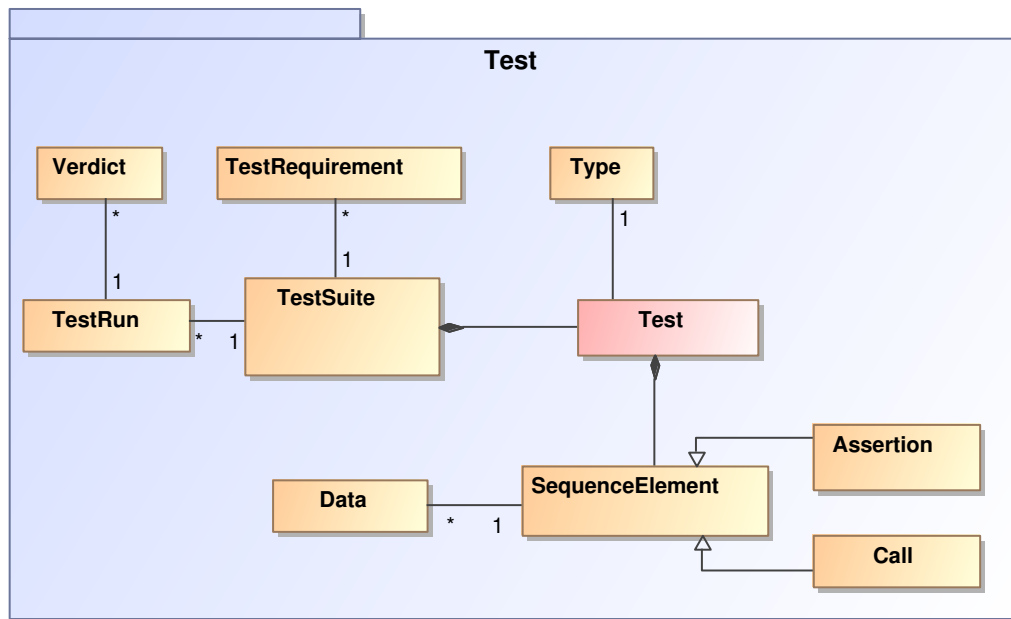


Figure 7.1: Telling TestStories Metamodel.

computed out of the test state or transitions among states. The state machine in Figure 7.2 depicts all possible test states and their transitions.

State transitions are not only initiated by direct modifications of *Test* elements but also triggered by requirements and services. The event *modifyRequirement* is triggered by changes of requirements and the event *modifyService* is triggered by changes of services.

A test is in state *new* as long as it has no requirements assigned by the operation *assignTest*. Note that the same operation is also a trigger in the *FunctionalRequirement* state machine, i.e., this assignment causes a state transition for both, tests and requirements. After the assignment of a requirement, a test is in state *notExecutable* until all *assigned* Service elements are in state *executable*. When a test or an *assigned* service is modified such that all *assigned* services of a test are in state *executable*, the test also gets the state *executable*. A guard condition checks for adherence to this rule.

Test elements have an attribute *Type* assigned which can be used for test selection. Depending on the modifications to the model, the *Type* of a test is updated. If the state of a test goes from *notExecutable* to *executable* its *Type* is set to *evolution* because the test is the result of an evolution step. The same is true if a test is currently in state *executable* and one of its assigned services is subject to a modification. On the other hand, if a service is modified but the current test under consideration is not assigned with that service, the *Type* of the test is changed to *regression* because the test should not be affected by this modification.

Also the modification of requirements influences the *Type* attribute of executable tests. When a requirement assigned to the current test under consideration is modified such that the test and the requirement are incompatible, the type is set to *stagnation* because the test should fail now. If, on the other hand, the requirement and the test are still compatible, the type is set to *regression*. The assessment whether a test validates a requirement is a manual task because the requirements are specified in an informal way. However, the compatibility only needs to be checked when a new test–requirement assignment is created, or when one of these elements is directly changed.

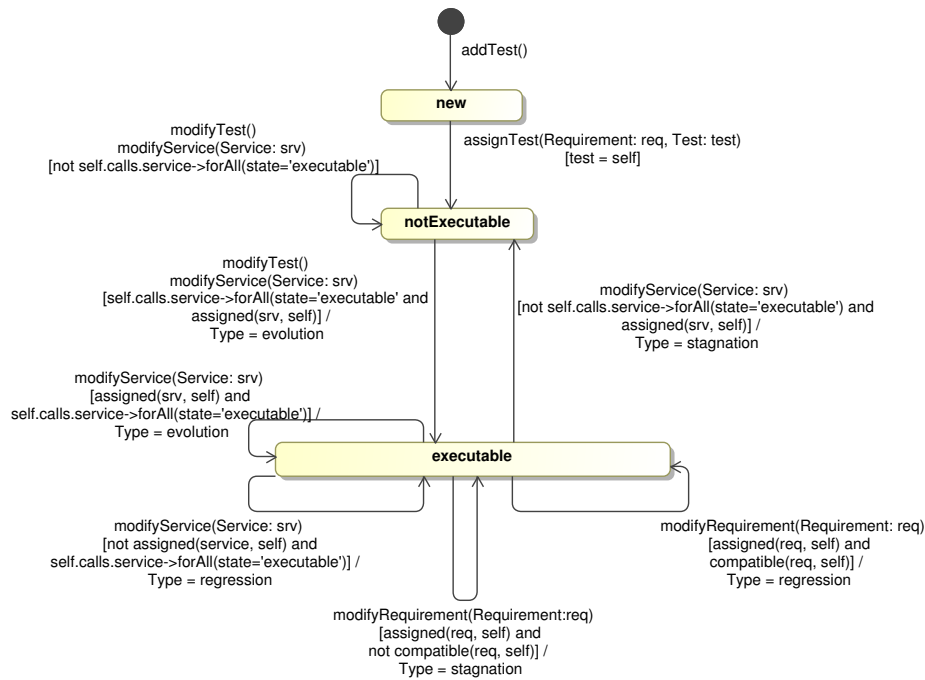


Figure 7.2: State machine describing the life cycle of Test elements.

Test suites are not only implicitly defined by the type of a test but can also be computed by test selection criteria in OCL. For instance, the following test selection criteria selects all tests that are supposed to pass, i.e., tests of type *evolution* or *regression*.

```

context Model:
Test::allInstances->
  select{t | t.type='evolution' or t.type='regression'}
  
```

8 Integration in SecureChange process

This section presents the relation of the Work package 7 and the other technical work packages of SecureChange project. In the process of the SecureChange project (cf. Figure 8.1,

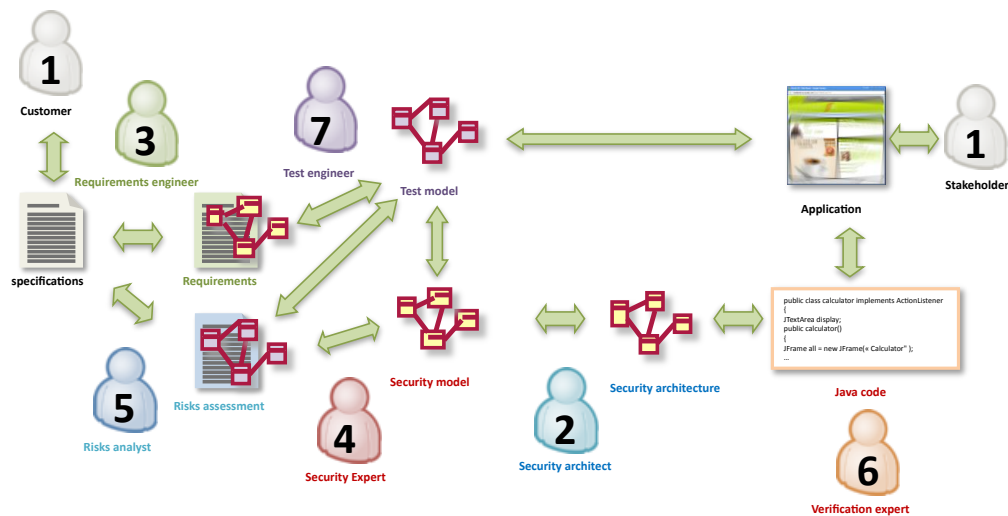


Figure 8.1: Process supported by SecureChange project

each WP is identified by a numbered person), WP7 takes security model as input. In fact, this security model is composed by three entities. The first entity is the requirements. So

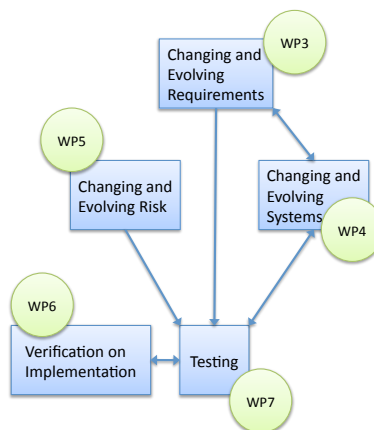


Figure 8.2: Links between WP7 and others WPs

WP7 can assume traceability between requirements and generated tests. The second entity is the model of behavior of the system. In order to provide a test oracle, the test model can predict the expected values of the system. So that we can refine the security model for the

test model. The third entity is security properties and their evolution. At this project time, WP7 deals directly with the origin work package that provide informations. The Figure 8.2 gives an overview of partners' interactions with WP7. The workpackage has interfaces to 4 other work packages. The first is WP3. WP3 provides requirements and their evolutions. The second is WP4. WP4 provides a validated security model and their evolution. The third is WP5. WP5 provides risks associated to security properties. The last is WP6. It is a special link because there is not direct interaction between WP6 and WP7 but they are complementarity for security verification.

8.1 WP3 – WP7

SecureChange WP3 is dedicated to Evolving Security Requirements Engineering. So, in this section we show how the bi-directional traceability between Requirements and generated test cases is managed and maintained through the SecureChange process.

8.1.1 Traceability between functional requirements extracted from specification and generated tests

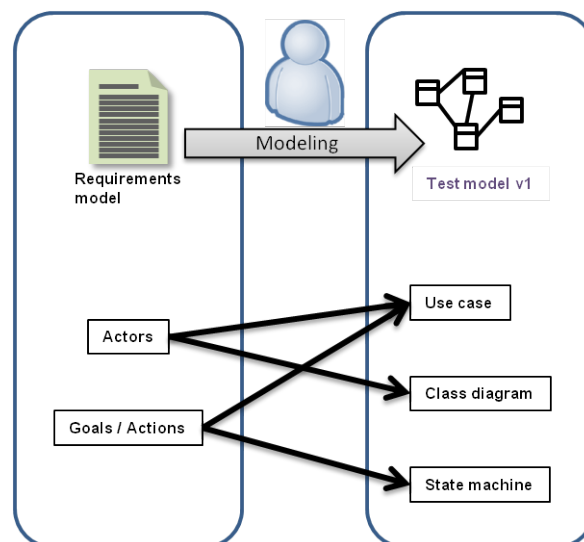


Figure 8.3: Using requirement model to create the test model

The requirements model provided by WP3 gives us a list of Actors, Goals and Actions to be used in the test model. Actors are used in a use case diagram to define what is the system under test. Goals and actions are also added in a use case diagram that is used to create the state machine of test model.

These components are available with a unique short name that is used to tag the OCL code. This name will be used to enforce the traceability between functional requirements extracted from specification (by WP3) and the generated tests. Test description contains this name and a matrix can be created to identify test coverage of the specification.

Each different actor is extracted from the requirements model provided by WP3 and added to a use case model to help identify which one is the system under test. The *Is-A* relationship guides us a way to specialize actors. Figure 8.3 shows the link between components of the requirement model and diagrams of the test model.

Actions and Goals are used to create a use case diagram that will be detailed by a state machine. The *decompose* relationship is used to define the structure of each goal. The *Do-dependency* relationship guides us to identify which actor is actually concerned by a goal. Actions are not sufficiently detailed in the requirement model to be used for the test model state machine. We need to find more specific information in the specification to be able to create a usable behavior. The state machine and its OCL code is annotated with tags that are short names provided by WP3.

The requirement model is used to help the test engineer to create a test model. It contains tags that are also used in requirements, requirement model, test model and tests. Traceability is improved by a link from specification to generated tests.

In the following section, we introduce how the requirement model evolution can be used to help the work of the test engineer.

8.1.2 Upgrading a test model by requirements models comparison

Requirements models can be used to upgrade a test model based on identified evolution. The defined structure of a requirements model makes the comparison of two versions of the model possible. The comparison produces a file that is used by the test engineer to upgrade the test model (see Figure 8.4).

The XML file contains, for each component previously mentioned (Actors, Goal and Action),

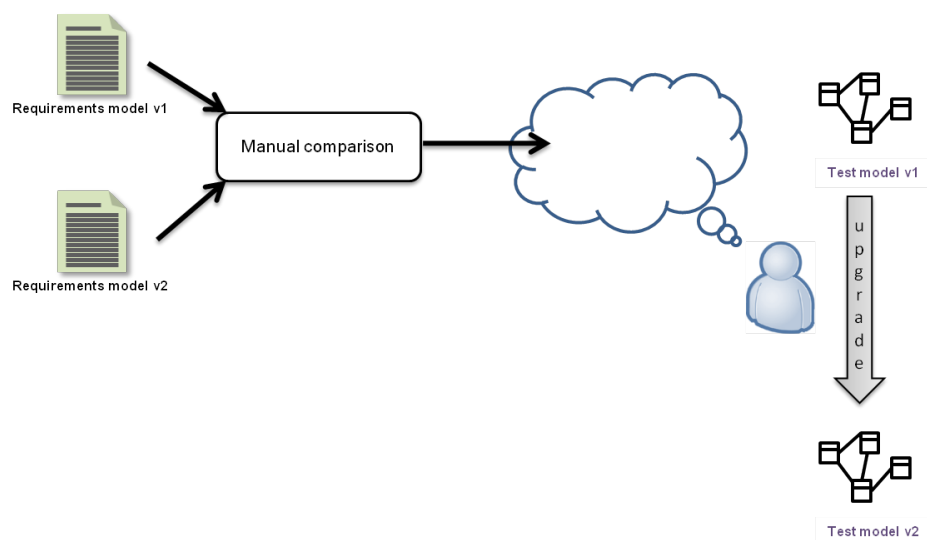


Figure 8.4: Test model upgrade process

a list of short names (requirement tag) and a status: new, deleted or modified. The status is used by the test engineer to focus on parts of the test model that need to change. Interactions between WP3 and WP7 will be applied on the GP case study. More details are available in deliverable D3.2.

8.2 WP4 – WP7

We describe in this section the connection between WP4, aiming to model adaptive security designs and verification of the security models, and WP7 aiming at test generation using

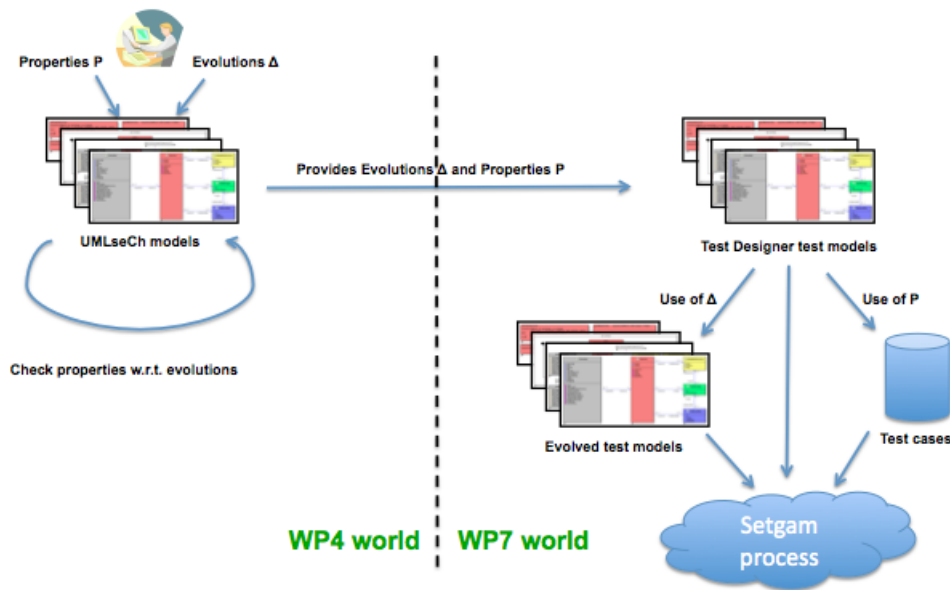


Figure 8.5: Integration between WP7 and WP4

models. From a methodological point of view the two approaches are complementary, and address a good practice of software engineering: in order to build relevant model based tests, it is mandatory to dispose of a model that has been extensively checked.

8.2.1 General process

The general process is reminded in Figure 8.5. On one hand, the WP4 world is represented by the UMLseCh approach/tool that is able to check a set of potential model evolutions against security properties. The considered security properties are those initially considered by UMLsec improved by some dedicated properties, designed for the case study.

On the other hand, the WP7 world is based on the TestDesigner technology that is able to generate model-based test cases from UML models with OCL constraints. TestDesigner considers a class diagram (that provides the logical data model), an object diagram (that provides an initial state of the considered system in terms of existing objects and relations between them), and a statechart diagram with OCL code (that provides a view of the dynamics of the system under test in terms of successive states reached by the invocation of points of control and observation).

In the context of the SecureChange project, the WP7 aims at providing a means for taking into account the evolutions and the security in the software testing process. To achieve that, WP7 proposes two solutions to address these issues. Firstly, a dedicated test generation process, based on user-defined test schemas that formalize test intentions related to security properties in order to drive automated test generation, is defined. Secondly, a special process based on an differential analysis of models makes it possible to focus the test generation effort on a subset of the software without sacrificing the overall validation of the whole software.

We now present in the following section the concrete integration between WP7 and WP4.

8.2.2 Concrete integration

We present the integration between WP7 and WP4 w.r.t. the two issues of security properties validation and evolution testing. For each issue, we present the solution proposed by WP7, and we show how to integrate this solution with the approach developed in WP4.

Integration w.r.t. security properties

The first integration possibility concerns the validation of security properties. Security properties are verified on the model using UMLseCh, and have to be considered for testing.

This integration relies on methodological aspects. Indeed, to be used for test generation, a formal model first has to be verified and validated. Our integration proposal is to be able to consider in our two approaches (WP4 and WP7) the same security properties, that will be:

- first, checked on the model so as to ensure that it satisfies the properties, and
- second, used as a basis for model based test generation in order to produce test cases that exercise (or at least cover) the considered security property.

This verification step is mandatory. If the model does not satisfy a given security property, and if this security property is involved in several test cases, then the tests may fail on a correct implementation, since this latter does not behave as (wrongly) expected by the model, thus falsifying the validation of the software.

Integration w.r.t. evolutions

The second integration point concerns the evolutions of the model. Whereas the UMLseCh approach considers a set of possible evolutions for which the preservation of security properties has to be ensured.

The integration proposed with UMLseCh is as follows.

On one hand, UMLseCh considers a set of possible evolutions for a considered model. These possible evolutions concern:

- the addition of a new model entity (class, state, etc.)
- the deletion of an existing model entity

These additions/deletions are specified by dedicated stereotypes in the corresponding UML diagram.

On the other hand, WP7 evolution testing approach considers two models and computes their differences.

Notice that the two approaches consider models that are designed using different UML modelling tools (ArgoUML for UMLseCh vs. IBM Rational Software Architect for TestDesigner). The switching from one of the notations to the other is not considered, since such activity is highly time-consuming and would not respond to a concrete need for integration. Thus, our integration solution has to be the less invasive possible in terms of adaptation of existing developments.

Our integration proposal consists in:

- using UMLseCh to specify one evolution of the model (a Δ between the original and evolved model depicted in Figure 8.5).

- export this evolution to provide it as an input for the WP7 process.

Having Δ it will be possible for WP7 to respectively:

- build the new model resulting from the evolution described,
- avoid computing the difference between the model, since it will be directly provided by Δ ,
- apply the rest of the methodology without any interference.

More details are available in deliverable D4.2.

8.3 WP5 – WP7

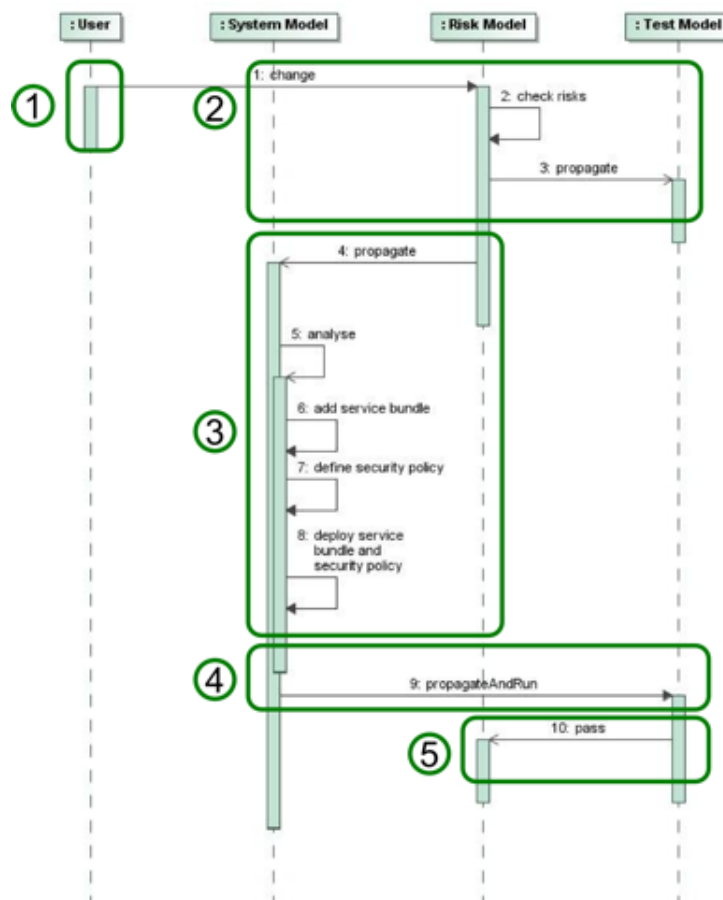


Figure 8.6: Change story of the HOMES case study

This section presents the link with the Work package 5 and the Work package 7. Figure 8.6 depicts the overall change story and the relation between the various technical solutions within the SecureChange Project. In the overall treatment of the change requirement the solutions of WP5 Risk assessment methodologies, WP2 SeAAS Architecture and WP7 TTS (Telling TestStories) will be applied.

The change story is divided in five segments, each with its own technical solutions to the change requirement and the security properties listed in the previous sections. The five segments correspond to the following points in time:

1. At this point in time a simple Home Gateway is already deployed and working successfully. The simple Home Gateway is already SeAAS capable and equipped with the simple security service "Confidentiality Service". The Home Gateway is analysed, tested and running without security problems. We have a system model and a risk model for that point in time.
2. At a certain point in time, the Operator notes increasing customer and third party service provider complaints. The Operator (depicted as User in the change story) orders the risk analysis team to update the existing risk analysis and find reasons and causes for the increasing number of complaints. The risk analysis team applies a risk assessment from a maintenance perspective and provides an updated risk picture, including newly identified threat scenarios and proposed treatments. The treatment is considered as an actual change to the system and therefore analysed from a before-after perspective by the risk analysis team. The resulting risk models depict the risk before the application of the treatment and the risk after the application of the treatment. In addition the risk to change is analysed. The results of step 2 are threat scenarios that serve as input to the test engineers in order to derive test cases. In addition the risk to change diagram provides the basis for the test engineers to derive regression tests, which are run after the application of the system change.
3. In step 3 the SeAAS Architecture is configured to deploy an additional security service, namely a "Non-Repudiation Service" to implement the treatment proposed by the risk analysis team.
4. In step 4 the SeAAS Architecture has been reconfigured and the test engineers start to conduct a series of tests on the new system. These tests include a functional security test to check whether the proposed treatment works as expected. In addition regression tests are run to check against the potential scenarios identified as a risk to change.
5. In step 5 the test engineers feedback the results of the various tests to the risk model, in which the reduction of risk values and the elimination of threats can now be confirmed as the treatment has been tested successfully.

Interactions between WP3 and WP7 will be applied on the HOMES and the GP case studies. More details are available in deliverable D5.3.

8.4 WP6 – WP7

We describe in this section the link between WP6, aiming to verify applet regarding security properties, and WP7 aiming at test generation using models. From a methodological point of view the two approaches are complementary just like verification and validation in software engineering.

8.4.1 Interest of the link between WP6 and WP7

We identified two artefacts shared by WP6 and WP7.

The first one is the system's implementation. WP6 analyses code and realizes static or dynamical verification on the implementation. WP7 provides tests to be executed on the system's implementation. So, these workpackages deal with a link between the high level of a conceptual approach and the concrete level of the implementation.

The second shared artefacts are security requirements. These requirements are translated into security properties that can be checked on the implementation.

In fact, WP6 and WP7 have no direct communication in the SecureChange process (depicted in Figure 8.1). The interest of an interaction between these work packages is to extend the verification and validation process. WP6 uses some hypothesis but cannot verify it, WP7 can validate these assertions using some dedicated tests. Conversely for WP7, some properties cannot be validated with an MBT approach but they can be verified by the WP6 approach.

8.4.2 Concrete scenario on case study

We instantiate the process presented in previous section, on the Global Platform (GP) case study and its evolution between specification version 2.1.1 and 2.2. For security requirements, we choose the security property *"information protection"* defined in deliverable D1.2. This property is decomposed in two parts: control flow of applet and access control. The access control focuses on the data of APDU. An APDU is a command that can be called by different components of the GP specification, for example it can be used by an applet or a Security Domain (SD).

Verification

Two kinds of concrete verification can be done. The first verification is the access control. In this case, we want to verify that protected information cannot be accessed without acquiring specific rights.

The second verification is the control flow, we want to ensure that there is no possible access to protected information during execution of different applets. That is, some protected data (attributes of an applet A) must not be read by an applet B. So during execution of applet B, there is no possible access to attributes of applet A. Information protection rules are represented by the smart card security policy.

The verification process is executed during the installation (or update) of an application by the card content manager. The verifier analyzes the bytecode of the applet in order to capture the details of information exchange (calls to other applets' methods, methods of the applet implementing shareable interfaces, possibly details on information exchange between variables) and checks if these details are compliant with the security policy of the smart card system.

Collaborations

We provide two kinds of scenarios. One for the illegal access to security domain services which decompose into 4 sub-scenarios (see deliverable 6.4). And the second for elicit information flow which decompose into 5 sub-scenarios. These scenarios demonstrate how WP6 and WP7 together provide protection against these attack scenarios and the advantages of the connection. On the nine scenarios defined, WP6 and WP7 can address two separate issues for the protection against threats of the Scenarios 1 and 2 except for the 1.d and 2.e. In fact, the interest of a connection between these two workpackages is that each hypothesis proposed by one workpackage is tackled by the other. So the completeness

of validation can only be done if the workpackages interact as proposed in SecureChange process. More details are available in deliverable D6.4.

9 Conclusion

This document gives a summary of the main results of SecureChange WP7 Task 7.3, which concern a proof-of-concept implementation of a model-based testing for evolution tool. This demonstrator is based on the original methods and algorithms for handling model-based testing for evolution and security testing provided in WP7 Task 7.2 and presented in Deliverable 7.2.

The main results address the two dimensions of the problem:

- an approach for testing security properties, based on the use of test schemas that formalize test needs. Security properties are covered by a test generation process using a behavioral model of the SUT and associated test schemas.
- an approach for change management by means of model comparison. Our objective is to ensure the important criteria defined in D7.1: test repository stability, traceability of changes, impact analysis and ability to automatically structure the test repository into evolution, regression and stagnation test suites.

Advantages of our approach lean on the same improvement: keeping trace of test cases through evolution. This allows a better stability of test suites, it also improves the organization of the test repository wrt evolution. This is a key factor to help the validation team to prioritize test execution. We also provide a more efficient test generation process that takes into account changes in requirements and previous test status to enhance the generation. This prototype implementation is based on existing technology for automated test generation (from Smartesting Test Designer tool suite), and goes one step further on functional evolution management. Moreover it ensures that security properties are well implemented in the SUT. The results obtained in this research target three distinct activities:

- **For model-based security testing** by driving test generation from the test model by test schemas that formalize the testing needs extracted by the validation engineer from the security properties.
- **For test generation based on model evolution analysis** in order to generate efficiently an accurate test repository.
- **For implementing the different new algorithms and techniques** in one demonstrator prototype usable for further experimentation on SecureChange case studies.

The next step of SecureChange WP7, within task 7.4 - Test generation for case studies, is to experiment the demonstrator in the context of project case studies, and to assess it with respect to the criteria assessment defined in deliverable 7.1:

- Stability of test repository;

- Traceability of changes;
- Impact analysis;
- Test suite qualification based on changes;
- Traceability of security properties;
- Completeness of security testing.

A Requirements for eCinema

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

Table A.1: Requirements of eCinema.

REQ Nb	AIM	Name	Description
1	LOG_Success	Login	To login the user must give a valid user name and password
1	LOG_Empty_User_Name	Login_Empty_User_Name	Impossible to connect with empty user name
1	LOG_Invalid_Password	Login_Invalid_Password	Impossible to connect with invalid password
1	LOG_Invalid_UserName	Login_Invalid_UserName	Impossible to connect with invalid user name
1	LOG_Logout	Logout	A user can logout from the main page
1	LOG_Logout_from_display	Logout_from_display	A user can logout from the display monitor page
2	REG_Empty_User_Name	Empty_User_Name	An empty user name is not allowed in registration process
2	REG_Empty_Password	Empty_Password	An empty password is not allowed in registration process
2	REG_Go_To_Register	Register	The user is allowed to create an account on the web site
2	REG_Login_Already_Exists	Login_Already_Exists	During registration user must chose a name not already existing in the database
2	REG_Success	Valid_Registration	After successful registration the user is transferred at the home welcome page
2	REG_Unregister	Unregister	A user can delete its account
3	BUY_Login_Mandatory	Buy_Login_Mandatory_to_buy	to buy tickets the user must login
3	BUY_No_More_Money	No_More_Money	To buy tickets the user must have enough resources on his account
3	BUY_Sold_Out	Ticket_Sold_Out	If no available tickets for the movie a message "sold out" is displayed to the user
3	BUY_Success	Buy	The user can buy tickets, if there are any and then data about the available tickets must be updated

REQ Nb	AIM	Name	Description
4	DIS_Check_Basket	Check_Basket	A connected user must be able to display its booked tickets
4	DIS_Login_Mandatory	Login_Mandatory_to_show	The user must login at the web site to display its booked tickets
4	DIS_PRICE_Login_Success	Login_Success	A connected user must be able to display the total price of his basket
4	DIS_PRICE_Login_First	Login_First	The user must login first to display the total price of his basket
5	REM_Del_All_Tickets	Del_All_Tickets	A user can empty his basket and then the booked places refunded to the user
5	REM_Del_Ticket	Del_Ticket	A user can chose the ticket he want to delete, which is refunded to the use
6	Close_application	Close_application	The user is close the application from the home page
7	NAV_GO_To_Home	Go_To_Home	The user is able to return to the home page
7	NAV_GO_To_Home_from_registration	Go_To_Home_from_registration	The user is able to return to the home page from the registration menu

Table A.3: Test targets of eCinema part 2/2.

B Requirements for eCinema Evolution

Nb	REQ	Name	Description
8	BALANCE_MNGT	BalanceManagement	The system must be able to manage one or several account
9	SUBSCRIPTION_MNGT	SubscriptionManagement	The system must be able to manage the login process and allow only registered user to login

Table B.1: New Requirements of eCinema, evolution.

REQ Nb	AIM	Name	Evolution
2	REG_Success	Valid_Registration	Change
2	REG_Unregister	Unregister	Change
2	REG_Empty_User_Name	Empty_User_Name	Change
2	REG_Login_Already_Exists	Login_Already_Exists	Change
2	REG_Empty_Password	Empty_Password	Change
3	BUY_No_More_Money	No_More_Money	Change
3	BUY_Success	Buy	Deletion
5	REM_Del_All_Tickets	Del_All_Tickets	Change
5	REM_Del_Ticket	Del_Ticket	Deletion

Table B.2: Impacts in test targets of eCinema.

REQ Nb	AIM	Name	Description
2	REG_ERROR_SUBSCRIPTION	Error_Subscription	
3	BUY_Success/STUDENT	Buy_Success/Student	
3	BUY_Success/CHILD	Buy_Success/Child	
3	BUY_Success/SENIOR	Buy_Success/Senior	
3	BUY_Success/NORMAL	Buy_Success/Normal	
3	BUY_Success/ONEFREE	Buy_Success/OneFree	
3	BUY_Success/ONEFREE-NORMAL	Buy_Success/OneFree-Normal	
5	REM_UPDATE_UserBalance_DEL-ONEFREE	Del_Ticket_OneFree	
5	REM_UPDATE_UserBalance_DEL-ONEFREE-NORMAL	Del_Ticket_OneFree-Normal	
6	CL_SUBSCRIPTION_MNGT	Close_Application	
6	CL_BALANCE_MNGT	Close_Application	
8	BAL_REG_RETRIEVE_MONEY-LGN_FIRST	Retrieve_Money_Lgn	
8	BAL_REG_RETRIEVE_MONEY-OK	Retrieve_Money_Ok	
8	BAL_REG_ADD_MONEY_OK	Deposit_Money_Ok	
8	BAL_REG_NEGATIVE_BALANCE	Deposit_Money_Negative_Balance	
8	BAL_REG_RETRIEVE-KO	Retrieve_Money_Ko	
8	BAL_REG_RETRIEVE_Money-BalanceInsufficient	Retrieve_Money_BalanceInsufficient	
8	BAL_RETRIEVE_MONEY-OK	Retreive_money_OK	
8	BAL_RETRIEVE_MONEY-LGN_FIRST	Retreive_money_login_first	
8	BAL_RETRIEVE_MONEY-KO	Retreive_money_KO	
9	SUB_GO_To_SUBSCRIPTION	Go_To_Subscription	
9	SUB_REG_SET_UP	Set_Subscription_Ok	
9	SUB_REG_Error_Sub	Set_Subscription_Error	
9	SUB_REG_LOGIN_FIRST	Set_Subscription_Login_First	

Table B.3: New test targets of eCinema, evolution.

C Transitions for eCinema

- t0 - closeApplication() - CLOSE_APPLICATION
- t1 - unregister() - REG_Unregister
- t2 - login(in_userName, in_userPassword) - LGN_SUCCESS
- t3 - login(in_userName,in_userPassword) - LGN_InvalidUsrNamePwd
- t4 - login(in_userName,in_userPassword) - LGN_WrongPwd
- t5 - login(in_userName,in_userPassword) - LGN_InvalidUser
- t6 - buyTicket(in_ticket) - BASKET_MNGT/BUY_TICKETS
- t7 - buyTicket(in_ticket) - Login_First
- t8 - buyTicket(in_ticket) - No_more_ticket
- t9 - logout() - LGN_Logout
- t10 - logout() - LGN_Logout - 2
- t11 - showBoughtTickets() - Logged in
- t12 - showBoughtTickets() - Not logged in - DIS_Login_Mandatory
- t13 - goToHome() - NAV_Go_To_Home
- t14 - goToHome() - NAV_Go_To_Home - 2
- t15 - deleteTicket(in_title) - REM_Del_Ticket
- t16 - deleteAllTickets(in_title)
- t17 - goToRegister()
- t18 - registration(in_userName,in_userPassword) - REG_Success
- t19 - registration(in_userName,in_userPassword) - REG_InvalidUserName
- t20 - registration(in_userName,in_userPassword) - REG_Existing_UserName
- t21 - registration(in_userName,in_userPassword) - REG_InvalidPwd
- t22 - showBasketPrice()- Succes
- t23 - showBasketPrice() - Login_first
- t24 - buyTicket(int_ticket)-No_more_money

D Transitions for eCinema Evolution

- t0 - closeApplication() - CLOSE_APPLICATION
- t1 - unregister() - REG_Unregister
- t2 - login(in_userName, in_userPassword) - LGN_SUCCESS
- t3 - login(in_userName, in_userPassword)-LGN_InvalidUsrNamePwd
- t4 - login(in_userName, in_userPassword) - LGN_WrongPwd
- t5 - login(in_userName, in_userPassword)-LGN_InvalidUser
- t6 - *buyTicket(in_ticket) - BASKET_MNGT/BUY_TICKETS - deleted*
- t7 - buyTicket(in_ticket)-Login_First
- t8 - buyTicket(in_ticket)-No_more_ticket
- t9 - logout() - LGN_Logout
- t10 - logout() - LGN_Logout - 2
- t11 - showBoughtTickets() - Logged in
- t12 - showBoughtTickets() - Not logged in - DIS_Login_Mandatory
- t13 - goToHome() - NAV_Go_To_Home
- t14 - goToHome() - NAV_Go_To_Home - 2
- t15 - *deleteTicket(in_title) - REM_Del_Ticket - deleted*
- t16 - deleteAllTickets(in_title)
- t17 - goToRegister()
- t18 - registration(in_userName, in_userPassword) - REG_Success
- t19 - registration(in_userName, in_userPassword) - REG_InvalidUserName
- t20 - registration(in_userName, in_userPassword) - REG_Existing_UserName
- t21 - registration(in_userName, in_userPassword) - REG_InvalidPwd
- t22 - showBasketPrice() - Succes

- t23 - showBasketPrice() - Login_first
- t24 - buyTicket(int_ticket) - No_more_money
- t25 - setSubscription() - OK
- t26 - setSubscription() - KO-Sub_Invalid
- t27 - setSubscription() - KO-Err_user
- t28 - goToSubscription()
- t29 - goToHome() - Subscription
- t30 - End() - Subscription
- t31 - deposit() - OK
- t32 - deposit() - KO
- t33 - End() - AM
- t34 - goToHome() - AM
- t35 - registration() - REG_ERROR_SUB
- t36 - buyTicket(in_ticket) - Sub-Child
- t37 - buyTicket(in_ticket) - Sub-Student
- t38 - buyTicket(in_ticket) - Sub-Normal
- t39 - buyTicket(in_ticket) - Sub-Senior
- t40 - buyTicket(in_ticket) - Sub-OneFree
- t41 - buyTicket(in_ticket) - Sub-OneFree_Normal
- t42 - retrieve() - OK
- t43 - retrieve() - BalanceInsufficient
- t44 - retrieve() - KO
- t45 - retrieve() - LGN_FIRST
- t46 - deleteTicket(in_title) - REM_Del_Ticket-Sub
- t47 - deleteTicket(in_title) - REM_Del_Ticket-OneFree
- t48 - deleteTicket(in_title) - REM_Del_Ticket-OneFree-Normal

Bibliography

- [1] B. K. Aichernig, M. Weiglhofer, and F. Wotawa. Improving fault-based conformance testing. *Electron. Notes Theor. Comput. Sci.*, 220:63–77, December 2008.
- [2] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *ICFEM'98, 2nd IEEE Int. Conf. on Formal Engineering Methods*, pages 46–54. IEEE Computer Society Press, December 1998.
- [3] F. Basanieri, A. Bertolino, and E. Marchetti. The Cow_Suite approach to planning and deriving test suites in UML projects. In *UML'02, 5-th int. conf. on the UML language*, volume 2460 of *LNCS*, pages 383–397, London, UK, 2002.
- [4] E. Bernard, B. Legeard, X. Luck, and F. Peureux. Generation of test sequences from formal specifications: GSM 11-11 standard case study. *Software: Practice and Experience*, 34(10):915–948, 2004.
- [5] A. Bertolino, E. Marchetti, and H. Muccini. Introducing a reasonably complete and coherent approach for model-based testing. *Electron. Notes Theor. Comput. Sci.*, 116:85–97, January 2005.
- [6] C. Bigot, A. Faivre, J.-P. Gallois, A. Lapitre, D. Lugato, J.-Y. Pierron, and N. Rapin. Automatic test generation with AGATHA. In H. Garavel and J. Hatcliff, editors, *TACAS 2003, Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference*, volume 2619 of *LNCS*, pages 591–596. Springer, 2003.
- [7] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. A subset of precise UML for model-based testing. In *A-MOST'07, 3rd int. Workshop on Advances in Model Based Testing*, pages 95–104. ACM Press, 2007.
- [8] D. Clarke, T. Jérón, V. Rusu, and E. Zinovieva. STG: A symbolic test generation tool. In *TACAS'02, Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 151–173. Springer, 2002.
- [9] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE'99, 21st international conference on Software engineering*, pages 411–420, Los Angeles, California, United States, 1999.
- [10] E. Farchi, A. Hartman, and S. S. Pinter. Using a model-based test generator to test for standard conformance. *IBM Systems Journal*, 41(1):89–110, 2002.
- [11] M. Felderer, R. Breu, J. Chimiak-Opoka, M. Breu, and F. Schupp. Concepts for Model-based Requirements Testing of Service Oriented Systems. In *Proceedings of the IASTED International Conference*, volume 642, page 018, 2009.

- [12] E. Fourneret and F. Bouquet. Impact Analysis for UML/OCL Statechart diagrams based on Dependence Algorithms for Evolving Critical Software. Technical Report RT2010-06, LIFC - Laboratoire d'Informatique de l'Université de Franche Comté, September 2010.
- [13] L. Frantzen, J. Tretmans, and T.A.C. Willemse. Test generation based on symbolic specifications. In J. Grabowski and B. Nielsen, editors, *FATES 2004, Formal Approaches to Software Testing*, volume 3395 of *LNCS*, pages 1–15. Springer, 2005.
- [14] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *SIGSOFT Softw. Eng. Notes*, 24(6):146–162, 1999.
- [15] C. Jard and T. Jéron. Tgv: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 7(4):297–315, 2005.
- [16] Bo Jiang, T. H. Tse, Wolfgang Grieskamp, Nicolas Kicillof, Yiming Cao, and Xiang Li. Regression testing process improvement for specification evolution of real-world protocol software. In *Proceedings of the 10th International Conference on Quality Software*, pages 62–71, 2010.
- [17] J. Julliand, P.-A. Masson, and R. Tissot. Generating security tests in addition to functional tests. In *AST'08, 3rd Int. workshop on Automation of Software Test*, pages 41–44, Leipzig, Germany, May 2008.
- [18] J. Julliand, P.A. Masson, R. Tissot, and P.C. Bué. Generating tests from B specifications and dynamic selection criteria. *FAC, Formal Aspects of Computing*, 2009.
- [19] Y. Ledru, F. Dadeau, L. Du Bousquet, S. Ville, and E. Rose. Mastering combinatorial explosion with the TOBIAS-2 test generator. In *ASE'07: Procs of the 22nd IEEE/ACM int. conf. on Automated Software Engineering*, pages 535–536, 2007.
- [20] Y. Ledru, L. du Bousquet, O. Maury, and P. Bontron. Filtering tobias combinatorial test suites. In Michel Wermelinger and Tiziana Margaria, editors, *Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004*, volume 2984 of *Lecture Notes in Computer Science*, pages 281–294. Springer, 2004.
- [21] H. K. N. Leung and L. White. Insights into regression testing [software testing]. In *Proceedings of Conference on Software Maintenance*, pages 60–69. IEEE, 1989.
- [22] O. Maury, Y. Ledru, and L. du Bousquet. Intégration de TOBIAS et UCASTING pour la génération des tests. In *ICSSEA'03, 16th Int. Conf. on Software and Systems Engineering and their Applications*, Paris, France, 2003.
- [23] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.
- [24] G. J. Tretmans and H. Brinksma. TorX: Automated model-based testing. In *First European Conference on Model-Driven Software Engineering*, pages 31–43, Nuremberg, Germany, December 2003.
- [25] H. Ural, R. L. Probert, and Y. Chen. Model based regression test suite generation using dependence analysis. In *Proceedings of the third international workshop on Advances in model-based testing*, pages 54–62, 2007.

- [26] L. Van Aertryck and T. Jensen. UML-CASTING: Test synthesis from UML models using constraint resolution. In *AFADL'03*, 2003.