D7.4 RESULTS OF TEST CAMPAIGN ON CASE STUDIES

Fabrice Bouquet (INR), Frederic Dadeau (INR), Stephane Debricon(INR), Elizabeta Fourneret (INR), Pierre-Alain Masson (INR), Zoltan Micksei (BME), Berthold Agreiter (UIB), Frank Innerhofer-Oberperfler (UIB), Bruno Legeard (SMA), Olivier Albiez (SMA), Julien Botella (SMA), Olivier Bussenot (SMA), Eddie Jaffuel (SMA), Christophe Grandpierre (SMA), Jean-Luc Hamot (SMA), Aurelien Masson (SMA), Dooley Nsewolo (SMA), Elisa Chiarani (UNITN), Michela Angeli (UNITN), Fabio Massacci (UNITN), Jan Jurjens and Sven Wenzel (OU/TUD)

Document Information

Document Number	D7.4
Document Title	Results of test campaign on case studies
Version	2.0
Status	Final
Work Package	WP 7
Deliverable Type	Report
Contractual Date of Delivery	M36
Actual Date of Delivery	26 01 2012
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	13.06.11	Draft	F. Bouquet(INR) B. Legeard (SMA)	Outline
0.2	31.10.11	Draft	E. Fourneret (INR), P-A. Masson (INR), F. Bouquet (INR)	Update EvoTest/SeTGaM
0.3	4.11.11	Draft	F. Bouquet (INR), F. Dadeau (INR)	WP7 Results
0.4	10.11.11	Draft	B. Legeard (SMA), J. Botella (SMA)	Evaluation criteria
0.5	17.11.11	Draft	B. Agreiter (UIB), F. Innerhofer-Oberperfler (UIB)	Homes Case study
0.6	28.11.11	Draft	J. Botella (SMA), B. Legeard (SMA), O. Albiez (SMA), O. Bussenot (SMA), C. Grandpierre (SMA), J-L. Hamot (SMA), A. Masson (SMA)	update SBTG
0.7	5.12.11	Draft	J. Botella (SMA), D. Nsewolo (SMA), E. Jaffuel (SMA), J. Bernet (GTO), E. Fourneret (INR)	Pops case study
1.0	15.12.11	Draft	J. Botella (SMA)	Typo.
1.1	3.1.12	Draft	B. Agreiter (UIB)	Review
1.2	7.1.12	Draft	Z. Micksei (BME	Review
1.3	11.1.12	Draft	J. Bernet (GTO)	Review update
1.4	13.1.12	Draft	B. Agreiter (UIB)	Homes update
1.5	11.1.12	Draft	E. Fourneret(INR)	Details for SeTGaM
1.6	18.1.12	Draft	J. Botella (SMA), E. Fourneret(INR)	Review update
1.7	19.1.12	Draft	M. Angeli (UNITN)	1st Quality check
1.8	25.1.12	Draft	E. Fourneret (INR), F. Bouquet (INR), J. Botella (SMA)	Integration of 1st QC
1.9	26.1.12	Draft	M. Angeli (UNITN)	2nd Quality check
2.0	31.1.12	Final	E. Fourneret (INR), F. Bouquet (INR), J. Botella (SMA)	Integration of 2nd QC

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

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
TCS	Test Case Specification
TTS	Telling TestStories
RSA	IBM Rational Software Architect

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 document is the last work package 7 (WP7) deliverable of the SecureChange project. The planned composition of the deliverable is exclusively on experimentation results. However, we have decided to add some complementary elements on research activities. There are two types of added contributions:

1. The first one contains an overview of all obtained results from WP7. We analyse these results with the criteria defined in deliverable D7.1. These criteria were used to analyse the state of the art in regards of the project's statement problem.

2. The second contribution in this deliverable presents the tool's improvements w.r.t the previous deliverable D7.3.

First, in this deliverable we discuss on the contributions. Next, we provide details on obtained results of the SecureChange two case studies: POPS and HOME. The first case study (POPS) is divided in two sub-scopes. Each one addresses a complementary aspect of the evolution and the security of the GlobalPlatform specification. The first sub-scope is the Card Life Cycle. The associated evolution is linked to the specification's evolution. It changes from version 2.1.1 to 2.2 UICC. From security point of view, we work on the translation of security properties into test needs. The second sub-scope is Card Content Management (CCM). It enabled us to focus on the translation of security properties into test needs. The goal of this sub-scope is to express security properties different to those in Card Life Cycle.

The second case study (HOME) addresses the evolution of the home gateway services. The provider can update services but the associated security properties must be guaranteed. WP7 provides an application of a methodology on this case study.

This deliverable is composed as following. In Sec. 2, the results' overview is presented. Sec. 3 is the update of previous deliverable (D7.3). Sec. 4 gives obtained results on POPS case study. Sec. 5 gives the results obtained on HOME case study. Finally, Sec. 6 provides a conclusion and discussion on WP7 results.

2 WP7 Results Summary in regards to Evaluation Criteria

This chapter summarizes the evaluation of SecureChange WP7 results with respect to the evaluation criteria that have been defined at the very beginning of the project and presented in deliverable D7.1. In this section we first recall the evaluation criteria, then we summarize the main WP7 results: on evolution management and on security testing and finally we propose an evaluation of these results w.r.t. to those criteria.

2.1 Reminder of Evaluation Criteria

In the first SecureChange WP7 deliverable (D7.1), we have defined the evaluation criteria to analyse existing model-based testing approaches with respect to the SecureChange project objectives. These criteria are structured into two categories:

- Evaluation criteria w.r.t. the evolution management;
- Evaluation criteria w.r.t. the security testing.

Name	Description	Evaluation
Stability of test repository	Ability to minimize the impact of evolutions on the test repository in term of creation / deletion of tests	scale 1..3 (1: complete re-creation, 3: maximum tests re-use)
Traceability of changes	Ability to trace an evolution from requirements to test repository	scale 1..3 (1: no traceability, 3: full traceability)
Impact analysis	Ability to inform the user on potential impacts of an evolution on the test repository	scale 1..3 (1: no impact analysis, 3: full impact analysis)
Test suite qualification based on changes	Ability to create test suite based on change type	qualification / no qualification

Table 2.1: Evaluation criteria for change

Therefore, we have proposed six criteria specifically dedicated to evaluate MBT approaches. These are gathered into two groups:

- Change: criteria to evaluate how MBT methods deal with evolutions in the context of long-life evolving systems, as presented in Table 2.1. There are four criteria associated to *Change* in order to take into account evolution in the testing process.

- Security: criteria to evaluate how MBT methods deal with security properties in the context of long-life evolving systems as presented in Table 2.2. There are two criteria associated to *Security* in order to establish confidence in the generated specific security tests.

These criteria reflect an industrial point of view, and have been defined with the industrial SecureChange partners involved in WP7. They reflect the practicability and effectiveness of an MBT approach dedicated to deal with continuous security testing of long-life evolving systems.

Name	Description	Evaluation
Traceability of security properties	Ability to provide bi-directional traceability between security properties and generated test cases managed in the test repository	scale 1..3 (1: no traceability, 3: full traceability)
Completeness of security testing	Ability to manage the test of functional security properties and to find security vulnerabilities (threats and attacks)	both / functional security properties only / security vulnerabilities only

Table 2.2: Evaluation criteria for security

After summarizing the main results of SecureChange WP7, we are using these criteria as an "interpretive lens" to analyse these results, based on the experience acquired during project case studies and experimentations.

2.2 WP7 Results

In the SecureChange project, WP7 has three types of results:

1. Conceptual method,
2. Methods with associated tools,
3. Experimentation.

2.2.1 Conceptual method

The conceptual method is composed by two elements. The first element is the integration of the test aspect into the general process of the SecureChange project. The second element is the methodology called Telling TestStories. This methodology can be used to test the services/security requirements of a system.

The SecureChange project provides a process to manage security and change. These two aspects are the key issues in the software engineering process for evolving systems. In this process, WP7 provides a method based on two artefacts. The first artefact is the test model. This model, based on the security model (provided by WP4), embeds a link with requirements (extracted and manage by WP3). The security aspect is given in the test model and can be completed by the test intention. The test intention is a translation of security properties or risk elements (identified by WP5) into test needs. These links are presented in the previous deliverable D7.3 and exposed in the article [9].

Telling Test Stories is a model-driven testing methodology developed in SecureChange project. This methodology provides a framework to maintain evolving test models. The methodology is based on a requirements model, a system model, and a test model. The overall evolution process manages changes of models, selection and execution of tests. The evolution process is initiated by adding, modifying or deleting model elements. State machines attached to various model elements describe the current state of the artefact, and allow a propagation of the change to other relevant model elements. This procedure provides an overview over the state of each test at any time, and automatically assigns test types when a change happens. Furthermore, it supports the creation of tailored test suites for specific needs. In particular, the presented methodology allows an automatic determination of tests affected by a change. This optimises the creation of test suites in two different ways:

1. on one hand, test suites are kept minimal, e.g. by only including tests which were affected by the last change in the model,
2. on the other hand, the test suites are kept up-to-date automatically by the test lifecycle, and additionally test requirement specifications in OCL allow finer-grained determination of test suites.

This approach is applied on the HOME case study (see Section 5).

2.2.2 Method with associated tools

This section gathers a concise description of all tools developed during the SecureChange project by two partners of WP7: INRIA and Smartesting. This development provides an Eclipse plugin called EvoTest. The details of the integration and the application example are represented in Section 3.3. The architecture of the plugin EvoTest is depicted in Figure 2.1.

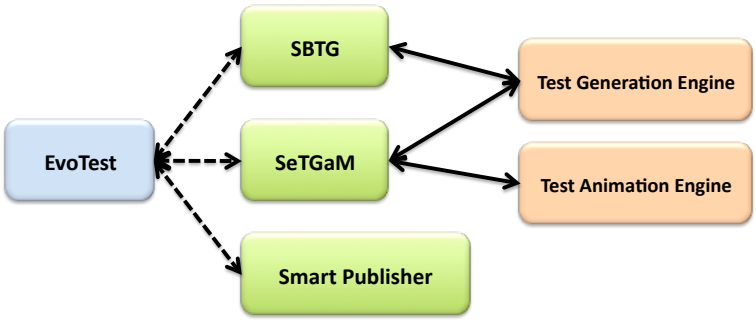

Figure 2.1: EvoTest plugin components

Below the three main components of the EvoTest plugin are detailed:

- **Schema-Based Test Generation** (SBTG), this component provides two features: a scenario editor and a test generator dedicated to scenario. The last improvements of the scenario language are presented in Section 3.1
- **Selective Test Generation Method** (SeTGaM), this component provides two features. The first one is the computation of dependencies in the test model elements and the scenario. This dependency allows to classify tests w.r.t. the evolution. This approach helps to maintain history of test and to reduce the test computation time. This method can take into account security aspects and UML/OCL models with or

without a statechart. The second feature is the test classification i.e. test life cycle management in a repository (for instance TestLink). Last improvements are presented in Section 3.2.

- **Smart Publisher**, this component manages the tests life cycle publication into the repository w.r.t. the classification computed by SeTGaM.

The two first components use a dedicated evaluation module developed by Smartesting for the SecureChange project. This evaluation module is the generation and evaluation engine as depicted in Figure 2.1.

2.2.3 Experimentations

To validate the methods and the tools developed in WP7 during the SecureChange project, an experimentation is made on two different case studies.

The first one, POPS, is proposed by Gemalto. On this case study, we use the EvoTest approach. This case study is composed by two experimentations: one on the Card Life Cycle and the second one on the Card Content Management. Obtained results are respectively presented in Sections 4.1 and 4.2.

The second case study is proposed by Telefonica. We have applied the Telling TestStories approach. Its results are presented in Section 5.

2.3 Evaluation of Results with Respects to Criteria

In this section, we provide an auto-evaluation on WP7 results with respect to the evaluation criteria that have already been presented.

Stability of the test repository This is a strong added value of the SeTGaM algorithm to maximize the stability of the generated test repository. The basic idea, which is to start the process from a previous tests generation, offers by construction this possibility. *Evaluation:* 3 - Maximum test re-use;

Traceability of changes Firstly, the requirement information is associated to a set of tags into the test model. In regards of the evolution, if the management of requirements is correctly done, the information can be propagated into the model via this tag (remove or add tags or the OCL code is modified). Then, the algorithms take into account these tags and propagate the link between test sequences and tags. Next, it is published into the test repository in order to provide a traceability matrix. The *Evaluation* grade is 2 for the first part because it is not an automatic process and 3 - full traceability - for the second part.

Impact analysis This information is done by test classification and four test suite (Evolution, Regression, Stagnation, Deletion). *Evaluation:* 3 - full impact analysis.

Test suite qualification based on changes This information is done also by the test classification. *Evaluation :* qualification.

Traceability of security properties This criterion is well implemented by the SBTG prototype. From Security Properties, the Security Engineer defines Security Test Objectives, and then formalizes them via Test Schemas in order to drive automated test case generation. Through all this process, a bi-directional traceability is managed and supported by the tool. *Evaluation:* 3 - Full traceability.

Completeness of security testing Security Testing of Information Systems is clearly a challenge that required a large set of complementary techniques, such as active and passive testing techniques. Therefore, the techniques developed in SecureChange WP7 mainly focus on a sub part of the challenge: mainly focused on testing the conformance of security functions (such as access control) using black-box active testing techniques. *Evaluation:* Functional security properties only.

The integration of SeTGaM and SBTG in the EvoTest prototype brings the mixed capabilities of managing evolution and driving test generation from security properties.

These results are detailed in the following sections.

3 Update on WP7 scientific and technical results

3.1 Schema-Based Test Generation (SBTG) for security testing

Smartesting provides model-based testing (MBT) solutions for functional testing. In order to address Security Testing, a prototype has been developed based on the Smartesting Test generation engine using dedicated Test Schemas to capture Security Test Objectives.

3.1.1 Overall Process

Model-based security testing from schemas is an extension of model-based testing which targets the conformance testing of security functions with respect to the specifications. Figure 3.1 illustrates this global process of model-based security testing.

This process is based on four main steps, as depicted in Figure 3.1

① Defining Security Test Objectives. (Security Engineer)
② Behavioural modelling. (Security Test Expert)
③ Defining Schemas for Test Generation. (Security Test Expert)
④ Automated test generation.

From the security property analysis, a Security Engineer defines the security test objectives. These expose the test objectives for security testing, in a detailed but informal form, and a possible way of testing them (how to test). These security test objectives define the testing strategy and impact the modelling activities and the driving of automated security test generation.

In the *Modelling* phase, there are two main activities that are delivered by the Security Test Expert in an incremental way. Both, the Behavioural Modelling of the System Under Test (SUT), and the Schemas definition are to be done.

Once the modelling activity is achieved, the Smartesting Schema-Based Test Generation engine prototype generates the test sequences corresponding to the expressed schemas, in regard of the behavioural model.

3.1.2 Defining Security Test Objectives

Security Test Objectives are defined by the Security Engineer from Security Properties concerning the SUT. For example a GP Security Property is an informal statement as illustrated in the following sentence:

For any execution, whenever the card is put in the TERMINATED *state by means of a* APDU_setStatus *issued by a privileged application, then it should not be possible to revert*

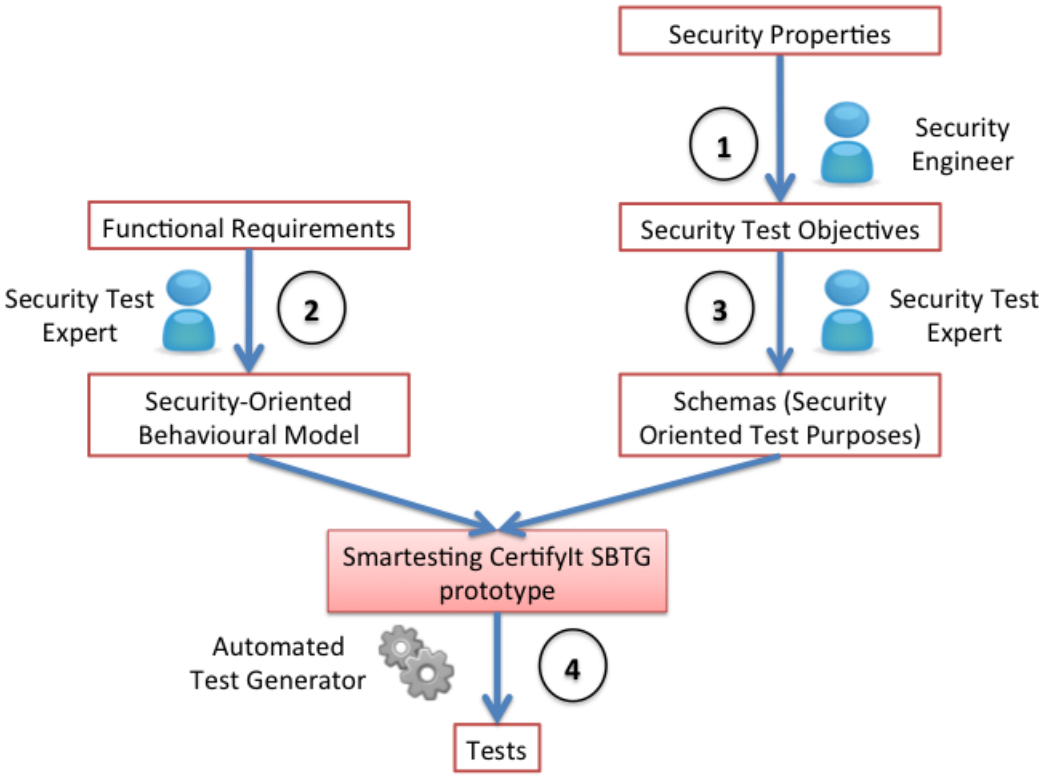

Figure 3.1: Process of Model-Based Security Testing with Schemas

to another state.

Security Properties may be tested in several ways. It depends on the SUT's characteristics and the Security Engineer knowledge and experience in testing such Security Properties. This know-how leads to Security Test Objectives defined in an informal way as illustrated the following statement:

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

3.1.3 Behavioural Modelling

This section summarizes the main characteristics of the behavioural modelling as it is supported by the current SBTG prototype version.

The behavioural model focuses on security features. In fact, the behavioural model is restricted on the features relevant to the security test objectives. The model formalizes the relevant point of control and observation, and the expected behaviours of the SUT. The security-oriented behavioural model is built on the basis of the SUT functional requirements and the security test objectives.

The model is created using the IBM Rational Software Architect v8.x (RSA) tool. A RSA project can contain a UML2 model, which is used by the Smartesting prototype to generate tests. Next we detail the model elements that are used for the test generation with the Smartesting prototype. We present the different modelling activities: data type definition,

class diagram design, initial state creation and statechart definition.

Data types

The Smartesting generation engine is able to manipulate the UML primitive types **Boolean** and **Integer**. Other types can be represented by **Enumerations**. For the GlobalPlatform use case, the generator should be able to access several applications by their identifiers, application and card states, and a status word must be returned after each APDU call. Figure 3.2 presents the Enumerations that represent those different types of data:

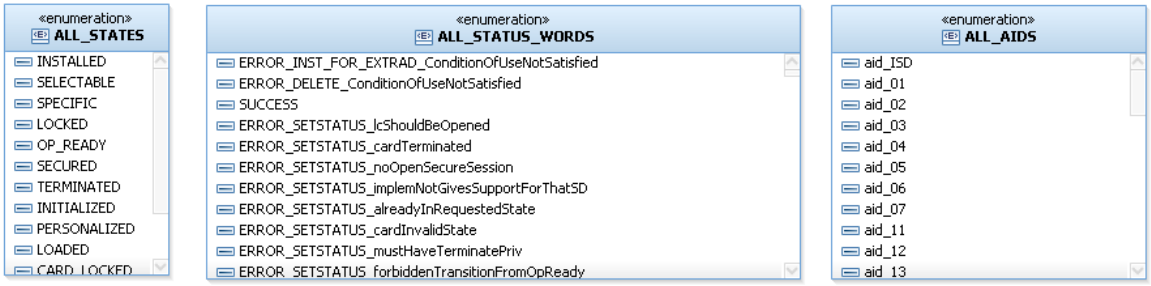

Figure 3.2: GlobalPlatform enumerations

More complex data to be manipulated are represented in a UML class diagram.

Class diagram

The class diagram contains classes of objects used for the test generation. For GlobalPlatform, the SUT is represented by a class (in our case named *Card*). The system must have the knowledge about applications that are installed on it, logical channels that ensure communications, and the secure sessions that can be established on logical channels. Each of these kinds of objects are also represented by classes, that have attributes (i.e. "state" for the card) to store the system state.

Associations represent the relations between the objects. For instance, on the Card system may install several applications. Each object can potentially perform actions on the system, which are represented by UML operations.

Operations

Each control point of the SUT is represented by an operation. Here the Card class contains different operations that represent those control points (APDUs). Figure 3.4 is an example of operation that represent the SetStatus APDU.

When there is a need to observe the result of a SUT stimulation, operation with an "observation" stereotype can be created. Here after each SUT stimulation, the status word returned by the card is checked.

Initial Model Instances

Classes and association instances then represent the initial state of the SUT, with initial values given for each attribute.

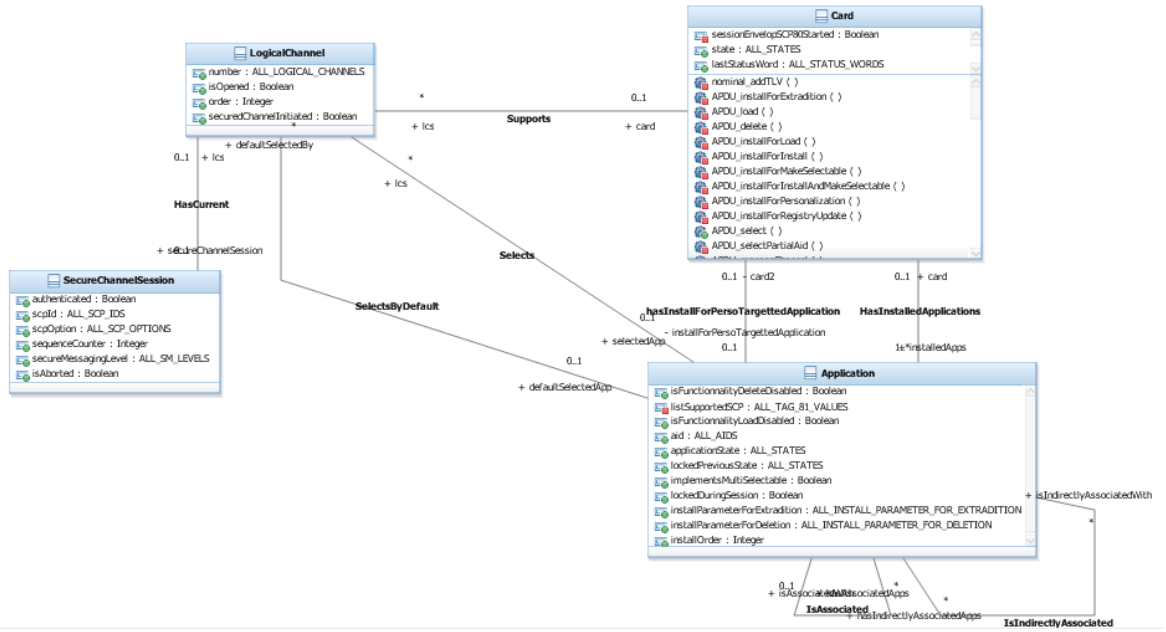

Figure 3.3: Classes, attributes and Associations

```
APDU_setStatus(
    IN_claSMLevel : ALL_SM_LEVELS,
    IN_lcNumber : ALL_LOGICAL_CHANNELS,
    IN_option : ALL_SET_STATUS_OPTIONS,
    IN_state : ALL_STATES,
    IN_appAid : ALL_AIDS
) : ALL_STATUS_WORDS
```

Figure 3.4: Operation Definition

Statechart

A statechart can be created to represent the dynamic part of the SUT. Here each state of the statechart represents a state of the card. Transitions between the states represent the card life cycle transitions. Internal transitions represent actions that can be done on each state (Figure 4.2).

Each transition is triggered by one operation defined in the class diagram. To express the conditions that must be respected to trigger a transition, a guard can be defined. For each transition, an effect can be defined to represent the expected behaviour of the SUT. Both are represented with Object Constraint Language (OCL) code as seen on Figure 3.6. Here the card state transition from OPREADY state to INITIALIZED is concerned.

The OCL code effect may contain annotations that enable to *tag* each branch of the code. Tags is a set of requirements (denoted by the keyword REQ) and functional test objectives (denoted by the keyword AIM), covering the expected behaviour (see Figure 3.6).

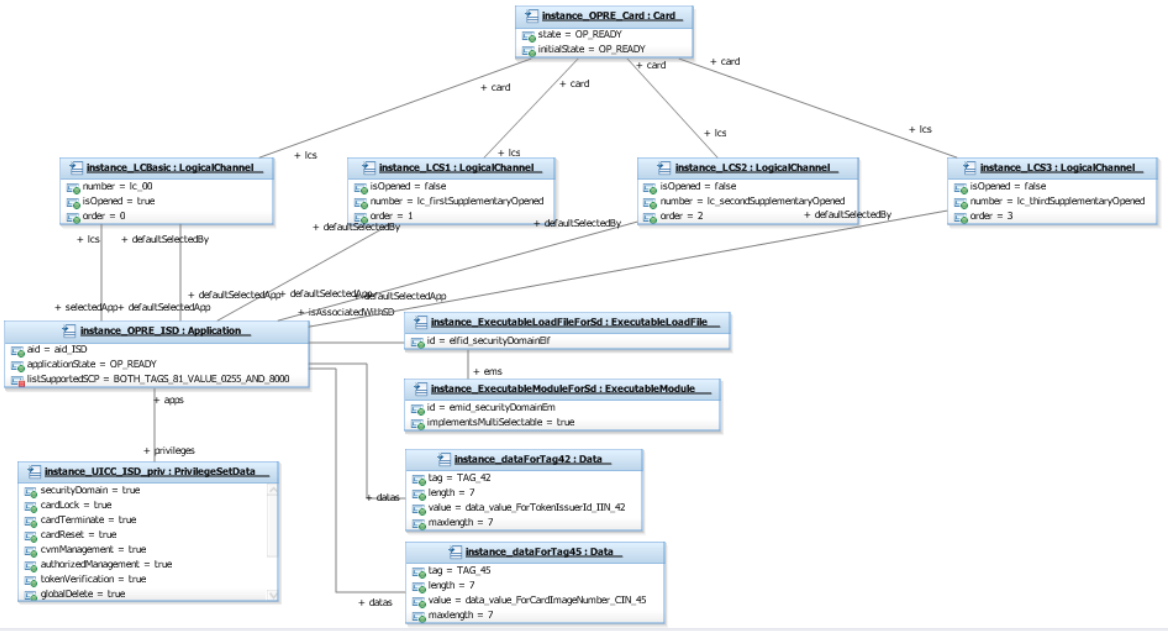

Figure 3.5: Instances

Guard - setStatusOpreadyToInitialized_privilegedSD

```
IN_appAid = ALL_AIDS::aid_ISD and
IN_state = ALL_STATES::INITIALIZED and
IN_option = ALL_SET_STATUS_OPTIONS::CARD and
self.lcs->exists(lc : LogicalChannel |
        lc.number = IN_lcNumber and
        lc.secureChannelSession.secureMessagingLevel = IN_claSMLevel and
        lc.selectedApp.aid <> ALL_AIDS::aid_ISD and
        lc.selectedApp.privileges.securityDomain = true
    ) and
self.lastStatusWord = ALL_STATUS_WORDS::SUCCESS and
self.state = ALL_STATES::INITIALIZED
```

Effect - setStatusOpreadyToInitialized_privilegedSD

```
self.state = ALL_STATES::INITIALIZED
---@REQ: SECURE_CHANGE
---@REQ: SUCCESS
---@REQ: OPREADY_TO_INITIALIZED
---@REQ: PRIVILEGED_SD
```

Figure 3.6: OCL guard and effect

3.1.4 Defining Schema

Schema is a formal definition of the Security Test Objective. It allows to explain the objectives with a dedicated Test Purpose Definition language. So, this formalization enables to express a meta-scenario in terms of states and actions in regards of model elements. Each schema is unfolded into several Test Case Specifications (TCS) that are used to generate

the test cases.

Test Purpose Definition language

The created **Test Purpose Definition language** is an evolution of the language explained in deliverable D7.3 and in the paper [8]. In addition, in Figure 3.7 and Figure 3.8, we give respectively the rules and terminals of the language.

```
test_purpose     ::=  (quantifier_list COMA)? seq EOF;
quantifier_list  ::=  quantifier (COMA quantifier)*;
quantifier       ::=  FOR_EACH BEHAVIOUR var FROM behaviour_choice
                  |   FOR_EACH OPERATION var FROM op_choice
                  |   FOR_EACH LITERAL var FROM literal_choice
                  |   FOR_EACH INSTANCE var FROM instance_choice
                  |   FOR_EACH INTEGER var FROM integer_choice
                  |   FOR_EACH CALL var FROM call_choice;
op_choice        ::=  ANY_OPERATION
                  |   ANY_OPERATION_BUT op_list
                  |   op_list;
call_choice      ::=  call_list;
behaviour_choice ::=  ANY_BEHAVIOUR_TO_COVER
                  |   ANY_BEHAVIOUR_TO_COVER_BUT behaviour_list
                  |   behaviour_list;
literal_choice   ::=  IDENTIFIER (OR IDENTIFIER)*;
instance_choice  ::=  instance (OR instance)*;
integer_choice   ::=  CURLY_OPEN INT (COMA INT)+ CURLY_CLOSE;
var              ::=  DOLLAR IDENTIFIER;
state            ::=  ocl_constraint ON_INSTANCE instance;
instance         ::=  IDENTIFIER;
ocl_constraint   ::=  STRING_LITERAL;
seq              ::=  bloc (THEN bloc)*;
bloc             ::=  USE control restriction? target?;
restriction      ::=  AT_LEAST_ONCE
                  |   ANY_NUMBER_OF_TIMES
                  |   INT TIMES
                  |   var TIMES;
target           ::=  TO_REACH state
                  |   TO_ACTIVATE behaviour
                  |   TO_ACTIVATE var;
control          ::=  op_choice
                  |   behaviour_choice
                  |   var
                  |   call_choice;
call_list        ::=  call (OR call)*;
op_list          ::=  operation (OR operation)*;
operation        ::=  IDENTIFIER;
call             ::=  instance '.' operation parameters;
parameters       ::=  PARENTHESIS_OPEN (parameter (COMA parameter)*)? PARENTHESIS_CLOSE;
parameter        ::=  FREE_VALUE
                  |   IDENTIFIER
                  |   var
                  |   INT;
behaviour_list   ::=  behaviour (OR behaviour)*;
behaviour        ::=  BEHAVIOUR_WITH_TAGS tag_list
                  |   BEHAVIOUR_WITHOUT_TAGS tag_list;
tag_list         ::=  CURLY_OPEN tag (COMA tag)* CURLY_CLOSE;
tag              ::=  REQ COLON IDENTIFIER
                  |   AIM COLON IDENTIFIER;
```

Figure 3.7: Syntax of the Schema Language: Rules

A Smartesting Test Purpose language editor with syntax highlighting and completion has been developed and integrated for IBM Rational Software Architect (RSA) environment (Figure 3.9).

```
                TIMES   ::=   'times' ;
              FOR_EACH  ::=   'for_each' ;
             BEHAVIOUR  ::=   'behaviour' ;
             OPERATION  ::=   'operation' ;
               INTEGER  ::=   'integer' ;
                  CALL  ::=   'call' ;
              INSTANCE  ::=   'instance' ;
               LITERAL  ::=   'literal' ;
                  FROM  ::=   'from' ;
                  THEN  ::=   'then' ;
                   USE  ::=   'use' ;
             TO_REACH   ::=   'to_reach' ;
            TO_ACTIVATE ::=   'to_activate' ;
           ON_INSTANCE  ::=   'on_instance' ;
         ANY_OPERATION  ::=   'any_operation' ;
     ANY_OPERATION_BUT  ::=   'any_operation_but' ;
                    OR  ::=   'or' ;
 ANY_BEHAVIOUR_TO_COVER  ::=   'any_behaviour_to_cover' ;
ANY_BEHAVIOUR_TO_COVER_BUT ::= 'any_behaviour_to_cover_but' ;
 BEHAVIOUR_WITH_TAGS    ::=   'behaviour_with_tags' ;
BEHAVIOUR_WITHOUT_TAGS  ::=   'behaviour_without_tags' ;
       AT_LEAST_ONCE    ::=   'at_least_once' ;
 ANY_NUMBER_OF_TIMES    ::=   'any_number_of_times' ;
                 COMA   ::=   ',' ;
           CURLY_OPEN   ::=   '{' ;
          CURLY_CLOSE   ::=   '}' ;
       PARENTHESIS_OPEN ::=   '(' ;
      PARENTHESIS_CLOSE ::=   ')' ;
                COLON   ::=   ':' ;
               DOLLAR   ::=   '$' ;
                  REQ   ::=   'REQ' ;
                  AIM   ::=   'AIM' ;
           FREE_VALUE   ::=   '_' ;
                  DOT   ::=   '.' ;
           IDENTIFIER   ::=   'identifier' ;
                  EOF   ::=   <EOF> ;
```

Figure 3.8: Syntax of the Schema Language: Terminals

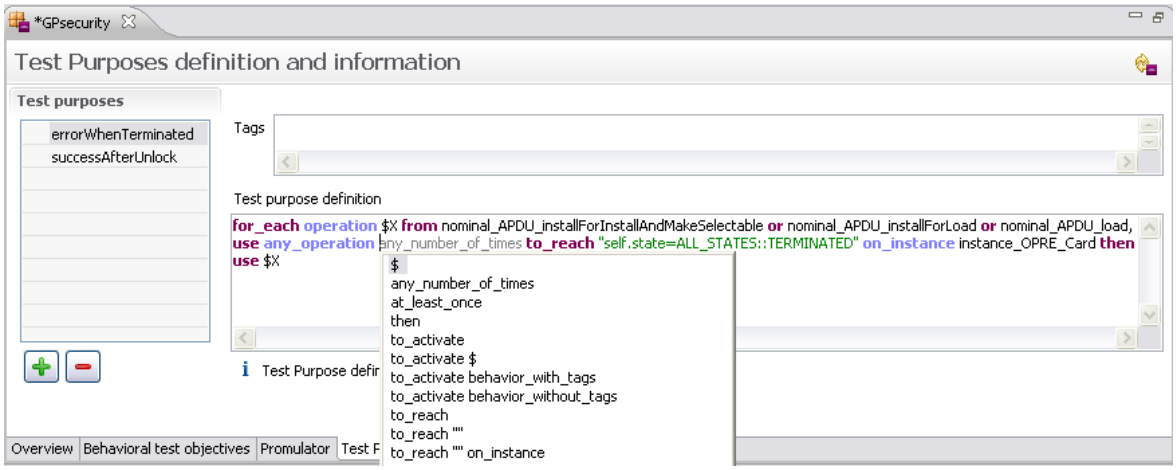

Figure 3.9: Test Purpose Editor

The Schemas are defined as part of dedicated test suites, that can be created to organize tests that will be produced by the Smartesting generation tool. The tests to be generated depend on the test suite definition (initial state, ...).

Test Suite definition

The Smartesting plugin enables the creation of test suites to define several SUT initial states as depicted in Figure 3.10. Here can also be defined Test Fixtures, that are the instances that contain model operations usable by the generator to compute the tests (by default this field is empty, allowing the generator to use any operation present in the model).

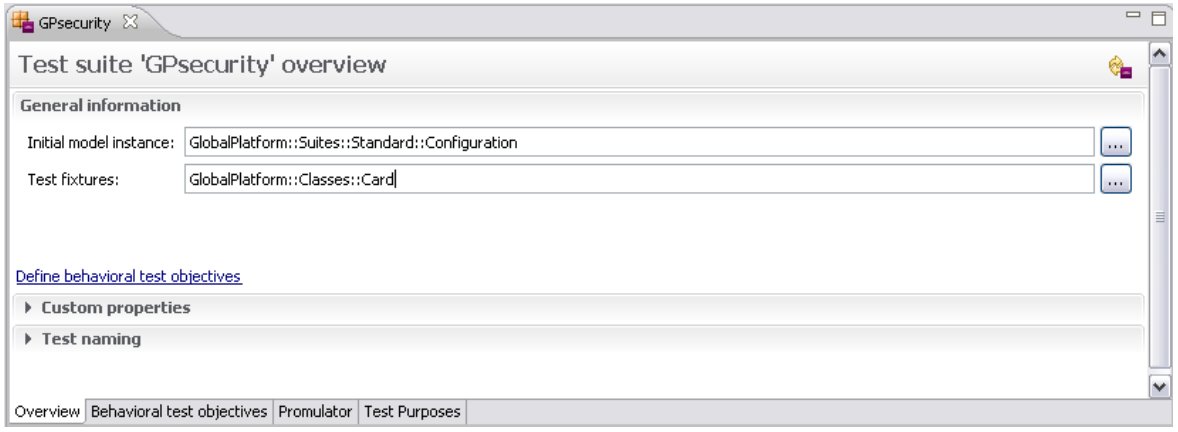

Figure 3.10: Test Suite Definition

For each test suite the behaviours to cover can be filtered (Figure 3.11). Tests are generated to cover only the behaviours filtered by the text given in the field "Definition". The filter keeps the behaviours that have the text in the Definition field contained either by the behaviour name, or by a requirement covered by the behaviour.

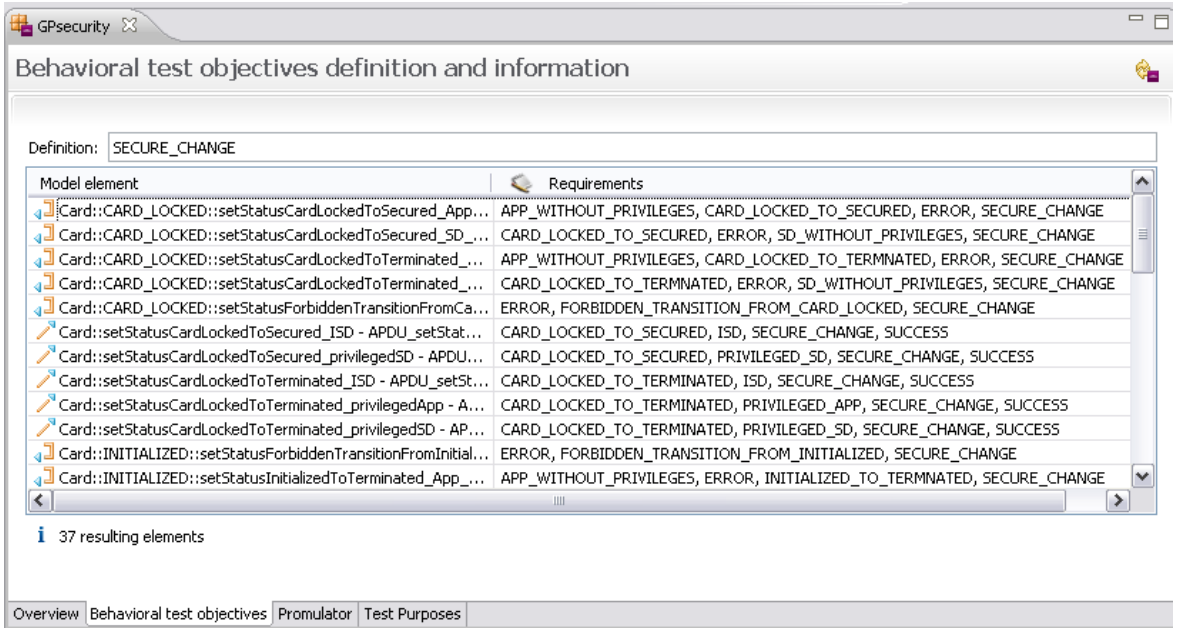

Figure 3.11: Behavioural Objectives Filter

The test generation from test purposes depends on our generation strategy from test schemas. It uses the behaviours defined in the test suite, and the initial state of the suite in order to access the objects to be used for test computation.

Test generation strategy from Schema

Once the Schema is defined, several TCS are created using the SBTG component. Each TCS is an instantiation of the Schema, with a combination of variables, possible values defined in the FOR_EACH part of the Schema.

For instance, consider the following Schema for GP:

```
for_each operation $X
    from nominal_APDU_installForInstallAndMakeSelectable
        or nominal_APDU_installForLoad
        or nominal_APDU_load,
    use any_operation any_number_of_times
        to_reach "self.state=ALL_STATES::CARD_LOCKED" on_instance instance_OPRE_Card then
    use any_operation any_number_of_times
        to_reach "self.state=ALL_STATES::SECURED" on_instance instance_OPRE_Card then
    use $X
```

It expresses that we aim at generating tests for each operation that can be used in the model, where an application with the CARD_TERMINATE privilege is selected, then the status of the card is set to TERMINATED, then an operation is called, represented by the selected one in $X.

The $X variable can be valuated with each operation of the model. The Schema produces as many TCS as the number (n) of possible values for $X. If an other variable is defined in the FOR_EACH part of the Schema, with m possible valuations, the strategy aims at creating a TCS for each possible combination of variable valuations. It would produce n * m TCS. For each TSC zero or one test is computed by Smartesting SBTG prototype depending on the TCS reachability. Figure 3.12 shows one of the tests produced from a TCS. The implementation of the Smartesting SBTG prototype has required a deep adaptation of the Smartesting test generation engine in order to manage efficiently TCS computed from Schemas.

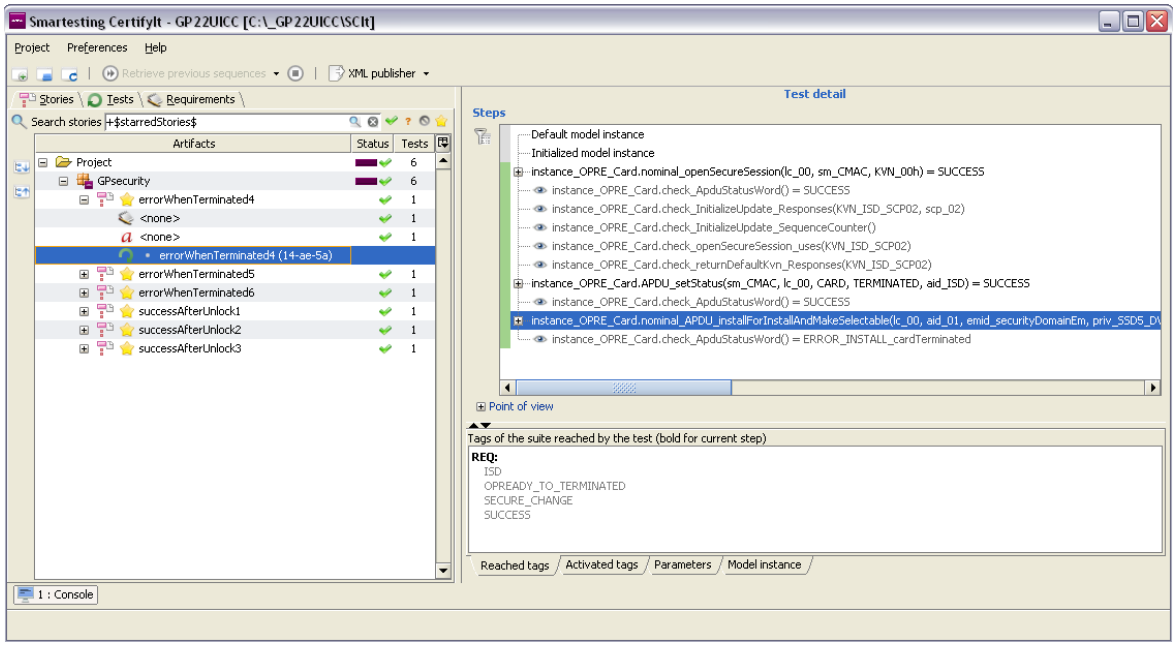

Figure 3.12: Generated Test from a Schema in Smartesting SBTG prototype

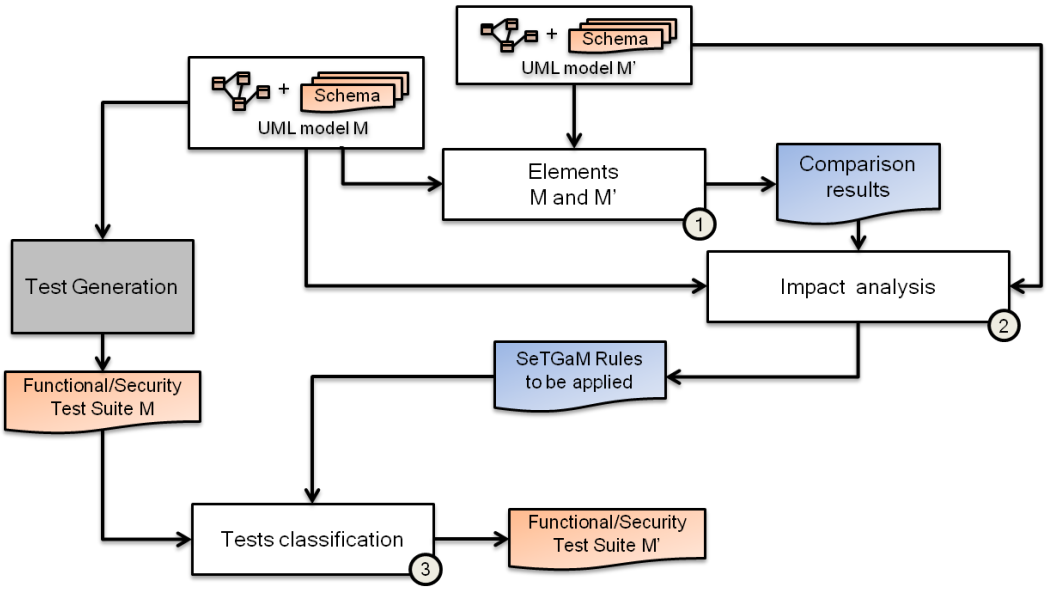

Figure 3.13: SeTGaM overall process

3.2 Selective Test Generation Method (SeTGaM)

In this section we give two extensions of SeTGaM. On one hand we present in details the extension of SeTGaM for security properties. On the other hand we give an extension of SeTGaM's impact analysis. The impact analysis was initially based on dependency algorithms for UML/OCL statecharts. Now, we have developed an impact analysis based on behavioural dependency algorithms for UML class diagrams, with pre/post conditions in operations written in OCL.

3.2.1 Overall process

On Figure 3.13 we summarize the overall process of SeTGaM. It starts by giving as input two models (the original *M* and the evolved one *M'*) including or not the associated schemas and a test suite issued from the original model and the considered schemas (if they exist), generated by Smartesting CertifyIt Tool. We first compare the elements (requirements for functional testing and Test Case Specifications (TCS) for security testing) from the original model *M* and the evolved one *M'* by classifying them into **Unchanged, Deleted, New and Modified**, tag ①. Then, in tag ② we put together on one hand the element's comparison and the evaluation of changes in dependency graphs on the other, to evaluate the impacted tests. Considering this information we can classify them first in step ③ as: *Unimpacted, Re-executed, Outdated, Failed, Adapted, Updated, New and Removed*. The final result is the produced test suite from model *M'*.

In addition, Figure 3.14 depicts how tests are gathered in the respective test suites. The evolution test suite is composed of new and adapted tests. The regression test suite is addressed by reusable tests, stagnation is addressed by outdated tests along with previous versions of the adapted tests (the failed tests). Finally, the deletion test suite is composed with tests issued from the *stagnation test suite* from the previous version.

Next we discuss how this process is extended for security properties testing and when there is no existing statechart diagram.

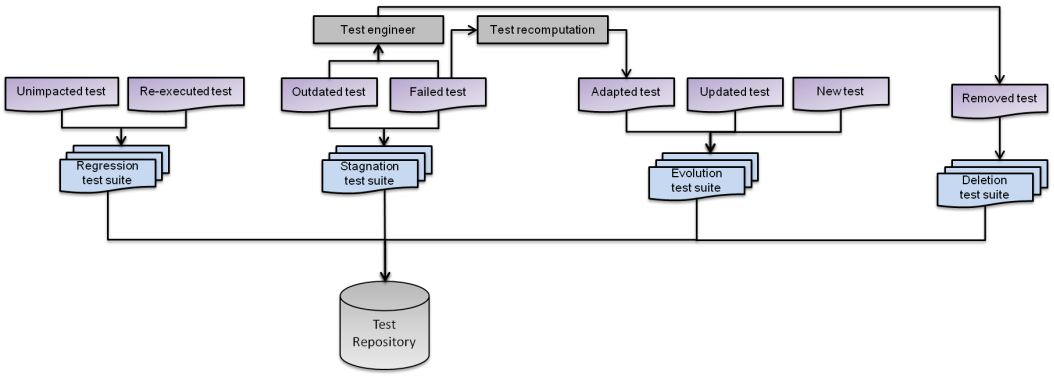

Figure 3.14: Test Suites Composition

3.2.2 Evolution Aspects in Security Testing

We present here how to manage the impact analysis when testing security properties. On one hand, in MBT we create test models to cover requirements originating from the system specification. On the other hand, 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 Section 3.1).

Selective test generation method for security properties

In this subsection we present the selective test generation method, called SeTGaM4Ssecurity, that guides the selection of tests produced before model evolution. As depicted at figure 3.13, it is based on a previous work done for selective test generation from UML/OCL Statechart using impact analysis, only from functional point of view [7]. Here we extend this work to test generation from security properties from UML/OCL Statecharts. In the previous work, cited above, the test intention was defined by requirements expressed in the model and identified as tags REQ/AIM. Here, we are focused on testing from security properties and the test intention is defined by Test Case Specifications (TCSs), obtained by schema unfolding, as discussed in Section 3.1. In the context of evolving systems and, thus, evolving security test cases, we associate to each test a *status*, that indicates its state in the life cycle, as depicted at figure 3.15.

Definition 1 (Evolving Security Test Cases) *An* evolving security test case tc^n *is characterized by a tuple* $\langle tc, \{tcs\}, status\rangle$ *in which* n *is the version of the model on which the test case* tc *applies,* tcs *is the set of test case specifications to which it is associated and* $status$ *is its associated status:* $status \in \{new, adapted, updated, unimpacted, reexecuted, failed, outdated, removed\}$

$status(tc^n)$ *denotes the status associated to* tc^n*.*

The overall process of SeTGaM test classification (see Section3.2.2) with respect to the security properties and the models evolution starts by giving as input two models (the original *M* and the evolved one *M'*) including the associated schemas and a test suite issued from the original model and the considered schemas, generated by Smartesting CertifyIt Tool. According to the process depicted on figure 3.16 we first compare the TCS from the original model *M* and the evolved one *M'* by classifying them into **Unchanged, Deleted, New**. Then, we put together on one hand the TCS comparison and the evaluation of changes in depen-

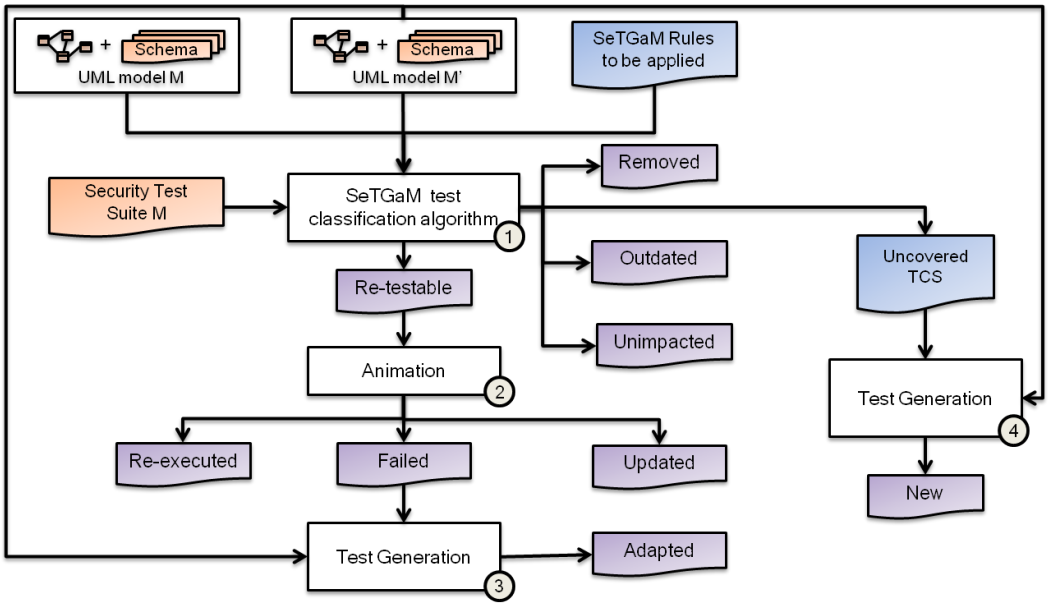

Figure 3.15: Test classification for testing security properties with respect to evolution

dency graphs on the other, to evaluate the impacted tests. Considering this information we can classify them and produce as result the test suite from model *M'*.

In addition, we detail at Figure 3.15 the test classification with respect to the security properties and the models evolution. In step ① tests are classified among four categories: **removed, outdated, unimpacted** and an intermediate category *re-testable*. Each test issued from **unchanged** TCS is classified as **outdated**(if it covers deleted requirements), **unimpacted**(if the covered requirements by the test still exist and they are not impacted) or **re-testable**(if the requirement exist but they are impacted by the change). Each test *re-testable* is animated on the new model version *M'*, tag ②. When the animated tests produces the same expected outputs, its status is set to **re-executed**. If the expected outputs are changed, then the status **updated** is attributed to test. When it is impossible to animate the step, then its status is set to **failed**. However, the test target (TCS) still exists in model *M'* and we need to generate another test to cover it, with the Smartesting test generation tool. The resulting test is set to **adapted**. For all new TCS, we generate new tests using the generation tool. During the test generation time overhead is expected, and thus with the test classification and reuse, the time complexity of the generation is much better than a full generation.

Evolution of schemas w.r.t TCS

At evolution process level, we consider three kinds of changes: (i) the test schema can evolve, (ii) the requirement, and thus the model, can evolve and we use the same test schemas for security testing or (iii) both can evolve. In testing from security properties we consider that both can evolve. However, when a schema evolves, we are interested in changes that happen in the set of produced TCS, as depicted on Figure 3.16, under ①. Thus, we consider three status of TCS: **Unchanged, New, Deleted**. Depending on the evolution, from one single schema we may unfold different sets of TCS for different model versions. TCS may remain unchanged or a new one can be added. In ② the unchanged set of TCS is then used by SeTGaM for security properties' for test classification, as already depicted on Figure 3.15.For each new TCS, the test generator produces *0* or *1* test ④. And

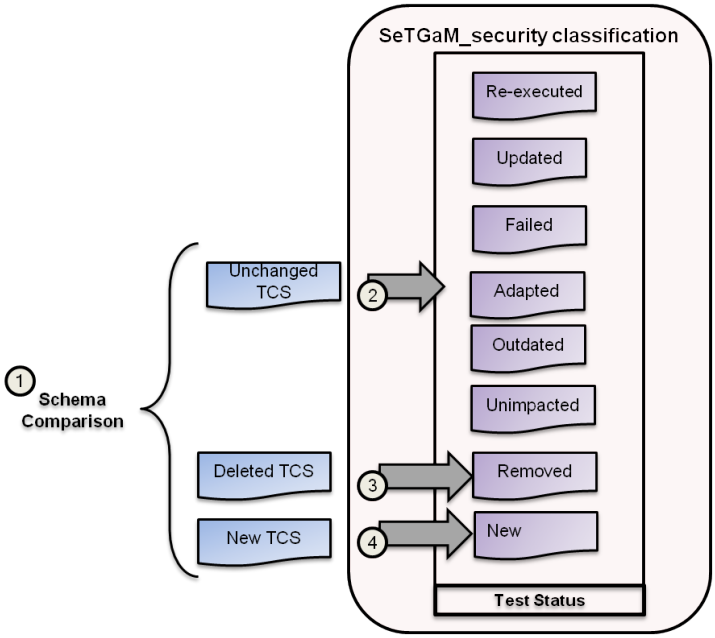

Figure 3.16: Test Status definition w.r.t TCS comparison

a deleted schema implies the removal of its associated TCS, and tests also, tag ③. It is straight-forward to find a test from a TCS and vice-versa, since we ensure their traceability.

In the next subsection we give more details about the test classification and test suite management based on security properties tests.

Evolution in Test Suites with respect to Security Testing

The composition of test suites that are considered for evolution management in security testing is represented by four test suites: Evolution, Regression, Stagnation and Deletion Test Suite (see Figure 3.14). 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. The definitions and rules below are extension to the rules given in our previous work in [7].

***Evolution* test suite** Γ_E contains tests produced from the **new** TCS, which represent the novelties of the system, such as new requirements, new operations, new behaviours etc., related to the security properties we are testing.

***Regression* test suite** Γ_R contains tests produced from **unchanged** TCSs which exercise 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 and that the security properties are still preserved.

***Stagnation* test suite** Γ_S contains invalid tests produced from **unchanged** TCS w.r.t. the current version of the system. 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 produced from **deleted** TCS, from deleted schemas w.r.t the current version.

Composition of security test suites We give here rules used to attribute tests into the previously defined test suites. We recall the notation, tc^n is a test from the initial version of the model and tc^{n+1} a test for the evolved version.

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

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

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: $status(tc^{n+1}) \in \{unimpacted, reexecuted\} \rightsquigarrow tc^{n+1} \in \Gamma_R^{n+1}$*

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.*

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

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.*

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

Notice that the previous versions (*failed* tests) are added in the Stagnation Test Suite. Since, we consider a light merge, a test set as outdated may lead to decreasing the TCS coverage. Thus, for each uncovered TCS by means of test classification of the previous model version, an adapted test is generated.

Rule 6 (Removed tests) *In the process of security properties evolution we consider that when a schema is deleted, by traceability they are gathered into the **Deletion Test Suite** and their life cycle is set to **removed**. We give here the extended definition:*

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

3.2.3 SeTGaM without UML/OCL statechart diagram

Using statechart diagrams helps a lot to identify which requirements are really dependent from another one and which ones are not dependent. Then, by using these dependency relations it is possible to define the impacted requirements. This work is done in our previous work in [6]. But some complex operations to test is often difficult due to several reasons, even impossible, to represent by a statechart. First of all it is not obvious to define the initial state or under which conditions is possible to change a state. Another possibility, there is not possible to identify a cycle process in the system that can be represented by a statechart. Another problem is that the number of states may explode and then the time to process all the data is increasing infinitely long. In this case, it can happen to obtain too many dependencies, which makes the SeTGaM application useless. A possible solution to this problem would be to use model slicing to reduce the number of states. This will create additional work that will slow down the process. Operations have pre and post condition expressed in Object Constraint Language (OCL). In each operation, we express several requirements. Each operation has represented by a behaviour and its associated tags. Thus, we have decided to work on an approach for dependency analysis based only on class diagrams and behaviours defined by OCL in the operations.

Next in this section we detail the extraction of model behaviours and then the calculation of behavioural dependencies using dependence-consistence.

Model behaviours

An OCL operation is defined by a precondition and a post condition. Generally, an operation has different behaviours, depending of the context where the operation is called. A behaviour is defined by a triple of precondition, post-condition and tags. We define formally a model behaviour with the definition 2

Definition 2 (Model Behaviour) *is extracted from an operation and is defined as a triple of precondition, postcondition and tags, denoted* B *:(@PRE,@POST,*Σ*TAGS).*

The behaviours can be identified by the presence of conditions (if, then, else) in the post-condition of the operation. The actions described by the "then" of the condition corresponds to one behaviour and the actions in the else, an another. Further, we depict in the Figure 3.17 the transformation of an operation into behaviour.

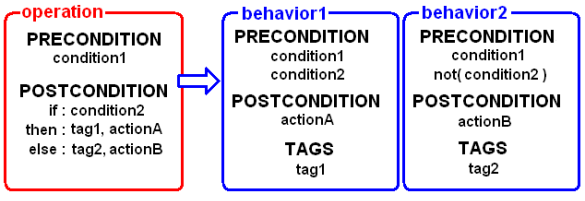

Figure 3.17: Transformation of an operation into behaviours

The post-condition correspond to the actions executed by the behaviour. The precondition is the union of the operation's precondition and the conditions necessary to reach the behaviour's post-condition. If a post-condition of the behaviour is placed in the "else" of a condition, the negation of the condition is stored as a precondition. The tags of the behaviour refer to requirements covered by the behaviour's post-condition.

In the OCL condition we accept unary logical operators such as *not* and binary logical operators such as *<, >, =, >=, <=, <>, and, xor, or*. This latter in expression *"A or B"* makes reference to three possible cases where the expression may be *true*. Thus, once the behaviours are extracted form the operations, the next step is to remove the binary operator *"or"* from the precondition. Indeed, the expression *"A or B"* constitutes three different cases that are: *"A and B"*, *"not A and B"* and *"A and not B"*. So, each behaviour which has an *"or"* operator in the precondition is divided in three behaviours having the same post-condition and tags but different preconditions. Finally, expressions with *not* unary operator, such as "not(A or B)" in the precondition are transformed in "not A and not B".

Impact analysis based on Behavioural Dependence

When discussing on dependences for class diagrams we do not consider control dependencies, since each operation may be called after each operation. If the operation is called with or without success it depends only on the conditions to be reached. Evaluate completely the conditions is time consuming but for dependence graph construction our goal is to reduce the processing time as much as possible. Thus, we suggest to constitute data dependence graph based on model's behaviours, extracted from an operation.

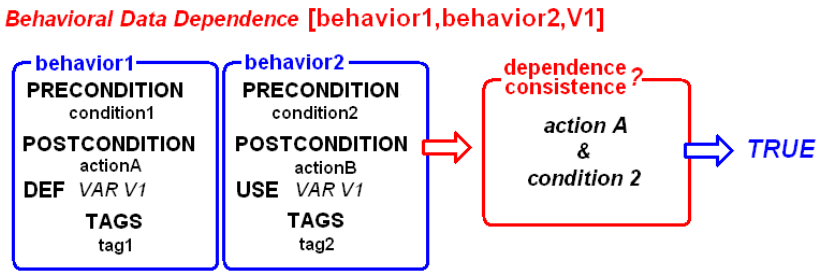

Figure 3.18: Behavioural Data Dependence process

Indeed, the graph based on definitions and uses of variables in behaviours represents more real dependencies than a graph based on definitions and uses in an operation. Moreover, we suggest a technique to reduce false positive data dependencies between behaviours. We depict the behavioural dependence process in Figure 3.18. As shown at this figure *behaviour* ***2*** depends from another *behaviour* ***1*** with respect to variable *V1*, defined in *behaviour 1* and used in *behaviour 2* if the condition composed by the postcondition *action A* and the precondition *condition 2* is *dependence-consistent*. We give below the formal definition of *behavioural-dependence*.

Definition 3 (Behavioural Data dependence) *A behaviour B' is data-dependent from another behaviour B w.r.t. variable v if and only if v is defined in B and used in B' and there exists dependence-consistence w.r.t. v between B' and B.*

We give below the definition for *dependence-consistence* between behaviours.

Definition 4 (Dependence-consistence) *A behaviour B' is dependence-consistent w.r.t behaviour B iff $(B'.@PRE \;\&\&\; B.@POST)$ is consistent.*

To verify the consistence of the condition from UML/OCL models we use a SATisfiability Modulo Theory (SMT) solver for first order logical formulas. A SMT solver can determine if the formula is true or not. However, there are many SMT solvers compatible with SMT-LIB format. In our case, we can use Z3 [3] and CVC3 [2] solvers. Indeed there is not really

difference in performance when using one or the other solver. According to the result's of the SMT solver's competition SMT-COMP[1] we have thus decided to use for SecureChange-project only the Z3 SMT solver.

Example of the approach on GlobalPlatform

To illustrate this approach, we consider an extract of the GlobalPlatform(GP) Card Life Cycle depicted on Figure 3.19. The extract of the OCL code corresponds to the **green** line on the Card Life Cycle statemachine, taking the card's state from **OP_READY** to **INITIALIZED**, and then to **SECURED** or **TERMINATED**. These three steps are considered as different behaviours, denoted from $B1$ to $B3$.

```
if (self.state = ALL_STATES::OP_READY)
then
    if (IN_state = ALL_STATES::INITIALIZED)
    then
         self.state = ALL_STATES::INITIALIZED and
         self.lastStatusWord = ALL_STATUS_WORDS::SUCCESS
     B1 /**@REQ: CARD_OP_READY_TO_INITIALIZED */
    else
        if (self.state = ALL_STATES::INITIALIZED)
        then
            if (IN_state = ALL_STATES::SECURED)
            then
                self.state = ALL_STATES::SECURED and
                self.lastStatusWord = ALL_STATUS_WORDS::SUCCESS
             B2 /**@REQ: CARD_INITIALIZED_TO_SECURED */
            else
                if (IN_state = ALL_STATES::TERMINATED)
                then if (l_lc.selectedApp.privileges.cardTerminate = true)
                     then
                         self.state = ALL_STATES::TERMINATED and
                         self.lastStatusWord = ALL_STATUS_WORDS::SUCCESS
                     B3 /**@REQ: CARD_INITIALIZED_TO_TERMINATED */
                     else ...
```

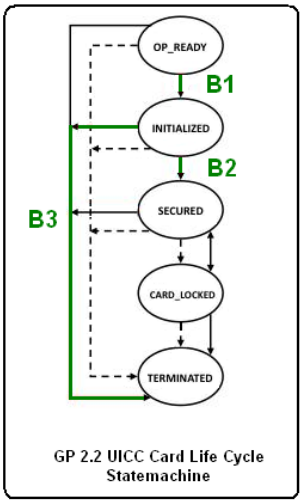

Figure 3.19: Set of behaviours for GP

Each behaviour is identified by the triple (@PRE,@POST,ΣTAGS) w.r.t Definition 2. Next we define the $definitions$ and $uses$ of variables in the behaviours, represented in Table 3.1. We compute data dependencies as triples composed by definition def and use use of variable v noted (def, use, v). According to these results and the def/use pairs, we conclude that all behaviours are mutually data dependent i.r variable `state`, thus we have 6 data dependencies denoted by $(B_i, B_j, state)_{i \neq j}$.

Behaviours	Variables			
	state	IN_STATE	lastStatusWord	cardTerminate
B1	def/use	use	def	
B2	def/use	use	def	
B3	def/use	use	def	use

Table 3.1: Variables definitions and uses for GP behaviours

We consider the set of three tests $t1$, $t2$ and $t3$, covering the behaviours $B1$, $B2$ and $B3$. The tests (decomposed in steps and the behaviours covered) are detailed in the Table 3.2. Now, we consider an evolution (change) in the guard of the behaviour $B3$.

When applying the test selection process with these data i.e. six *def/use* pairs and the change of $B3$, all tests are selected to be as **Retestable** w.r.t the defined test classification

[1] http://www.smtcomp.org

Tests	Steps	B1	B2	B3
t1	setStatus(INITIALIZED)	x		
t2	setStatus(INITIALIZED),setStatus(SECURED)	x	x	
t3	setStatus(INITIALIZED) setStatus(SECURED) setStatus(TERMINATED)	x	x	x

Table 3.2: Illustrative set of test for GP behaviours

(see the overall process above in Section 3.2.1).

In fact, not all of these tests should have been classified as **Retestable**. The *def/use* pairs define a **possible** data dependence between behaviours. We propose to reduce the number of possible data dependencies. For this, we have extended our approach and defined the *dependence-consistence*. When applying Definition 4 (*dependence-consistence*), we reduce the number of possible data dependencies to only two: $(B1, B2, state)$ and $(B1, B3, state)$.

The behaviour $B3$ has changed, so the $t3$ is set as **Retestable**. The data dependencies, w.r.t SeTGaM process, permitted to keep unchanged the tests $t1$ and $t2$ and their status is set as **Reusable**. To conclude, the *dependence-consistence* introduced for SeTGaM allows to reuse maximum of tests from a previous version and decide about their status.

3.3 Integration in EvoTest Plugin

In this section we present our work on software development level. We have created an industrial prototype to experiment our research work and to face real problems of scalability. The software, called EvoTest, is an Eclipse Plugin for IBM Rational Software Architect.

Figure 3.20 depicts a screenshot of the *EvoTest* Plugin. We have provided two main parts for this tool. The first panel *Test Purposes* is dedicated to the schema edition. The second panel *SeTGaM tool* on the page bottom is dedicated for the functional and security test suite management when system's evolution occurs.

The schema allows to produce TCS, in order to generate tests to cover a given security property. We associate the created schema to the model's test suite, called *smartsuite*. Which is then used for test generation by Smartesting CertifyIt generator engine. Moreover, in the industry the number of security properties may be very high. Take for instance the Global Platform project, there are many security properties. To respond to this scalability problem is not optional, we need to deal with many schemas. Thus, as depicted on the panel's left side (Figure 3.20), we give the possibility to create an undefined number of schemas and associate them to a test suite. On the panel's right side, the schema editor is available. It verifies in run time the schema's syntax and it offers to the user an autocompletion and coloring syntax feature. For internal use at the Smartesting company and at our SecureChange's industrial partners it has been shown that a validation engineer produces schemas, and thus tests, with better quality, correctly written and quicker. In addition, the engineers were satisfied using the autocompletion feature, since there was no additional effort for learning the schema's language syntax.

In the tool we address a second automated solution for managing tests dedicated to security properties, with respect to changes on requirements level, as detailed in D7.3. The panel dedicated to evolution is shown at Figure 3.20, allows the user to select two

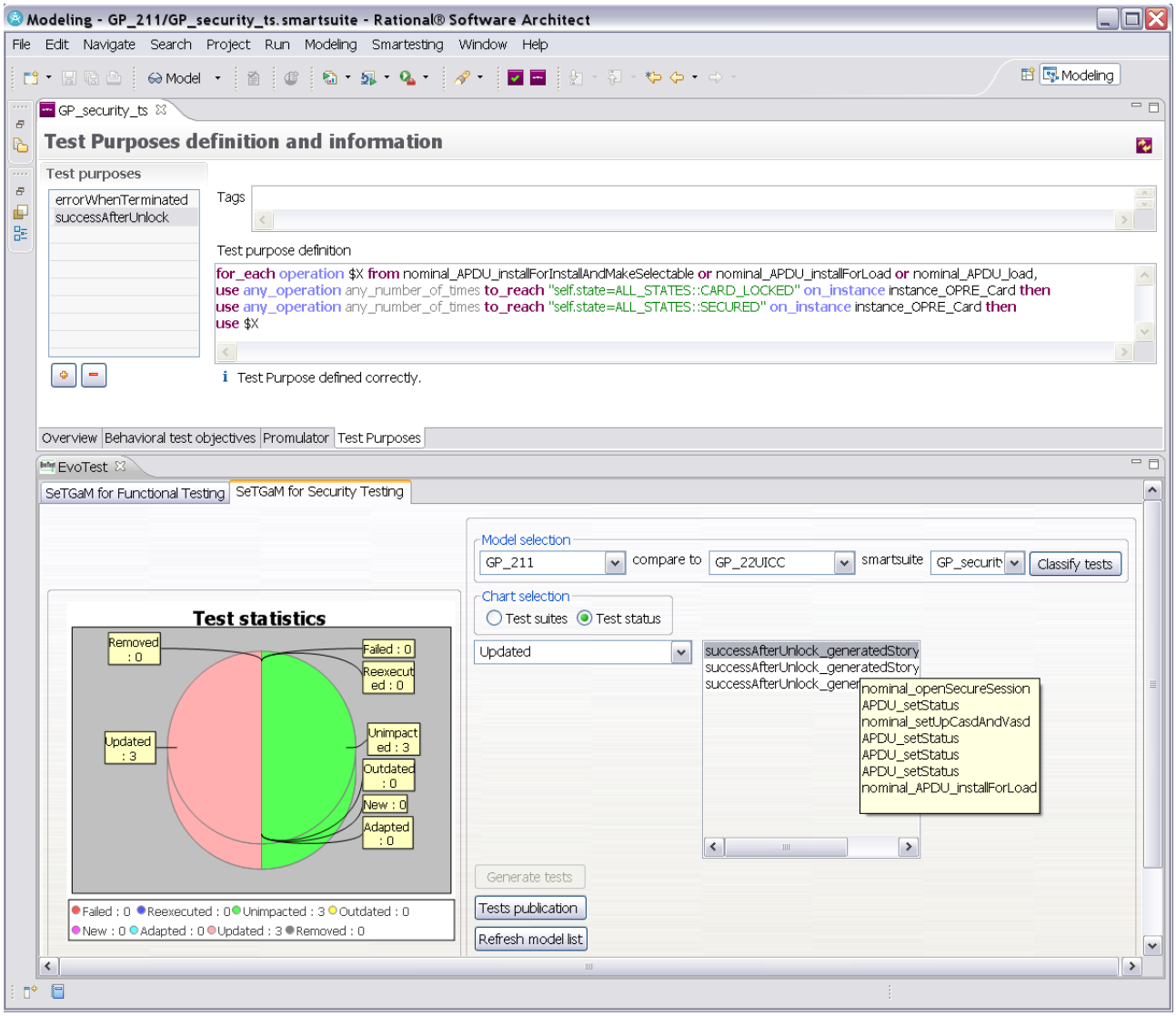

Figure 3.20: EvoTest integrated interface

model versions a the security test suite. Then (s)he can run the classification process on the already existing tests from the initial version and the generation of new and, if needed adapted tests. The different test statuses can be seen on the *piechart* depicted at the Figure 3.20.

The SeTGaM benefits are twofold. Firstly it ensures the test history; Secondly it eases the maintenance process when executing them on the system by:

- helping the validation engineer to understand where the problem comes from. If the system does not behave as expected, (s)he can easily locate the bug.
- running a large number of tests is time consuming. The user needs to prioritize the test execution. The classification we propose permits her/him to prioritize the execution of the tests. Our experience shows that the most important test suite for the user is the Stagnation Test Suite, since it refers to requirements that should not exist any more.

According to the POPS case study's first impressions feedback, we make it possible, after test classification, to select the wanted test suite and observe the added tests. When

setting the cursor on a given test at the right panel a tool-tip text appears containing the test steps, as shown at the Figure 3.20.

Once tests are classified and generated, if necessary, for the new version, the tool permits to export tests and their statuses using the Smart Publisher, in a TestLink mysql database, as depicted in Figure 3.21.

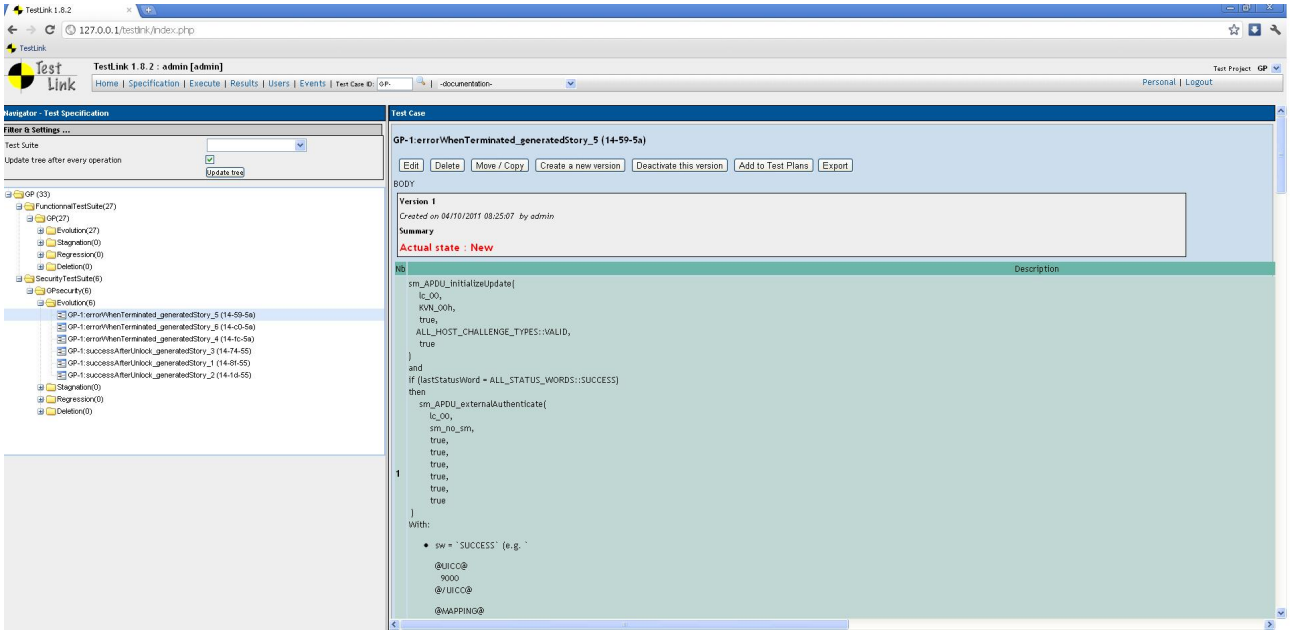

Figure 3.21: TestLink Smart Publisher

4 Results of test campaign on POPS Case study

4.1 GP 2.1.1 and GP 2.2 UICC Card Life Cycle

GlobalPlatform (GP) is a set of smart card management services such as the loading of applications. It provides an interface to communicate in a secure way with the external world, in accordance with GP specifications [1]. In the GP documents life cycle models are detailed in order to control the behaviour and the security of GP components: card, executable load files, executable modules and applications. The scope of our work is the management of the card life cycle, from the card's production until its destruction. We have created models for the two versions on the Card Life Cycle Scope of GlobalPlatform 2.1.1 and 2.2 UICC Configuration, the UICC Configuration is standardizing the minimum interoperability for (U)SIM cards for supporting remote application management.

4.1.1 Functional models

In this section we present the odels for Gp Card Life Cycle scope 2.1.1 and 2.2, UICC configuration.

GP 2.1.1 Card Life Cycle

The GP 2.1.1 Card Life Cycle model focuses on the APDU command SetStatus and its behaviours that manage the Card State transitions. However, in order to generate tests cases, the model must contain enough information to be able to create the necessary contexts to test each Card Life Cycle related behaviours. That is why "nominal" operations have been created. Those operations only contain the subset of behaviours useful for the activation of the dedicated Card Life Cycle behaviours. Some operations that represent a sequence of operations have also been created in order to increase readability of the generated sequences. For example a "nominal_openSecureSession" represents the successful sequence of the INITIALIZE_UPDATE and EXTERNAL_AUTHENTICATE APDUs permitting to establish a secure channel session. Here is the list of operations represented in that model:

- **APDU_setStatus**,
- nominal_APDU_installForInstallAndMakeSelectable,
- nominal_APDU_installForLoad,
- nominal_APDU_load,
- nominal_APDU_putKeyDesLight,

- APDU_manageChannel,
- APDU_select,
- nominal_openSecureSession,
- nominal_setUpCasdAndVasd.

Here are statistics about the model:

number of operations	number of usable behaviours	number of Card Life Cycle behaviours
9	132	32

The Statechart representing the Card Life Cycle for the GP 2.1.1 specification is shown on figure 4.1.

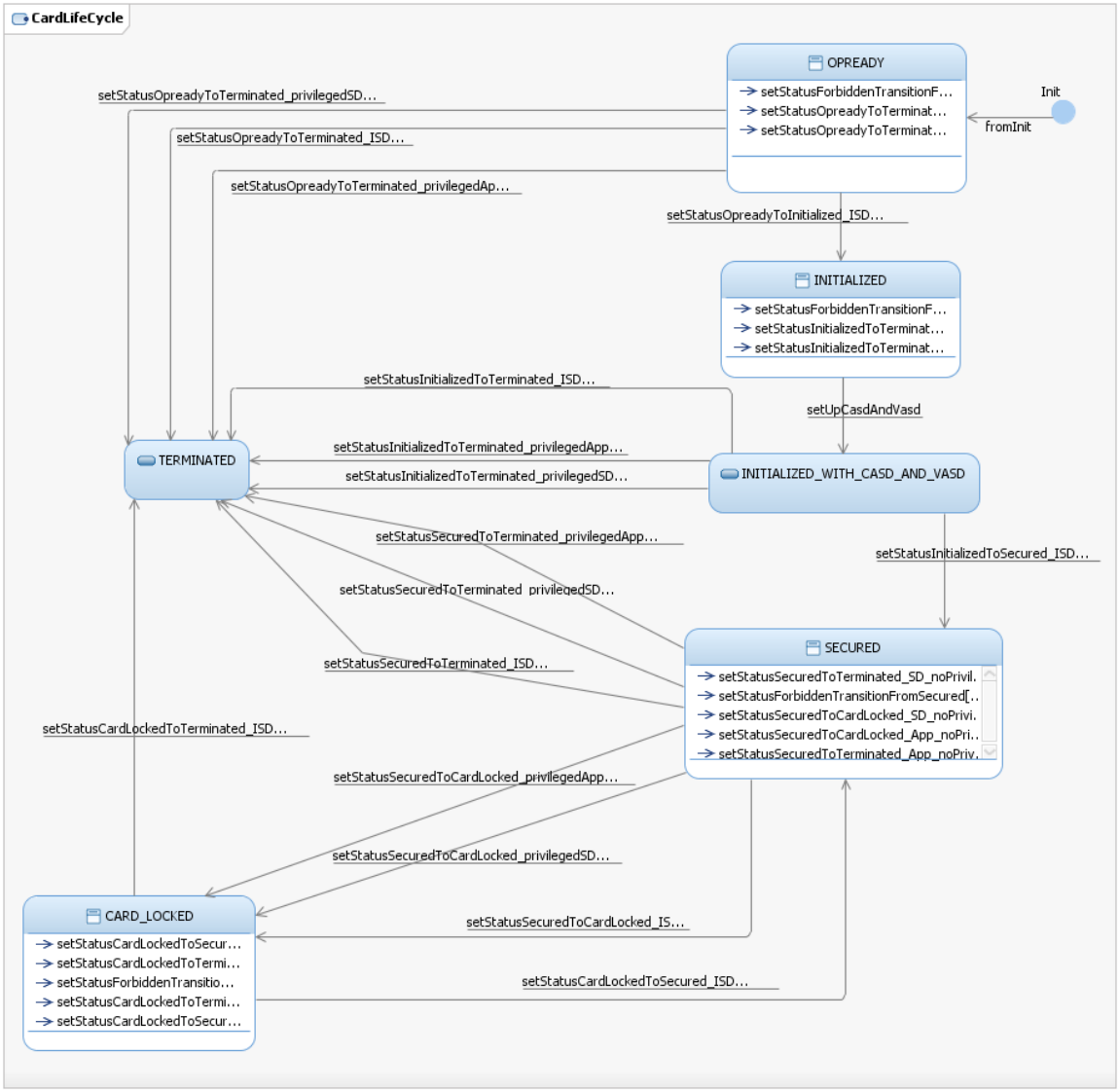

Figure 4.1: Statechart GP 2.1.1

GP 2.2 UICC Card Life Cycle

The scope is the same than the one for the GP 2.1.1 Card Life Cycle model. The same operations are modelled.

The Statechart representing the Card Life Cycle for the GP 2.2 UICC specification is shown on figure 4.2.

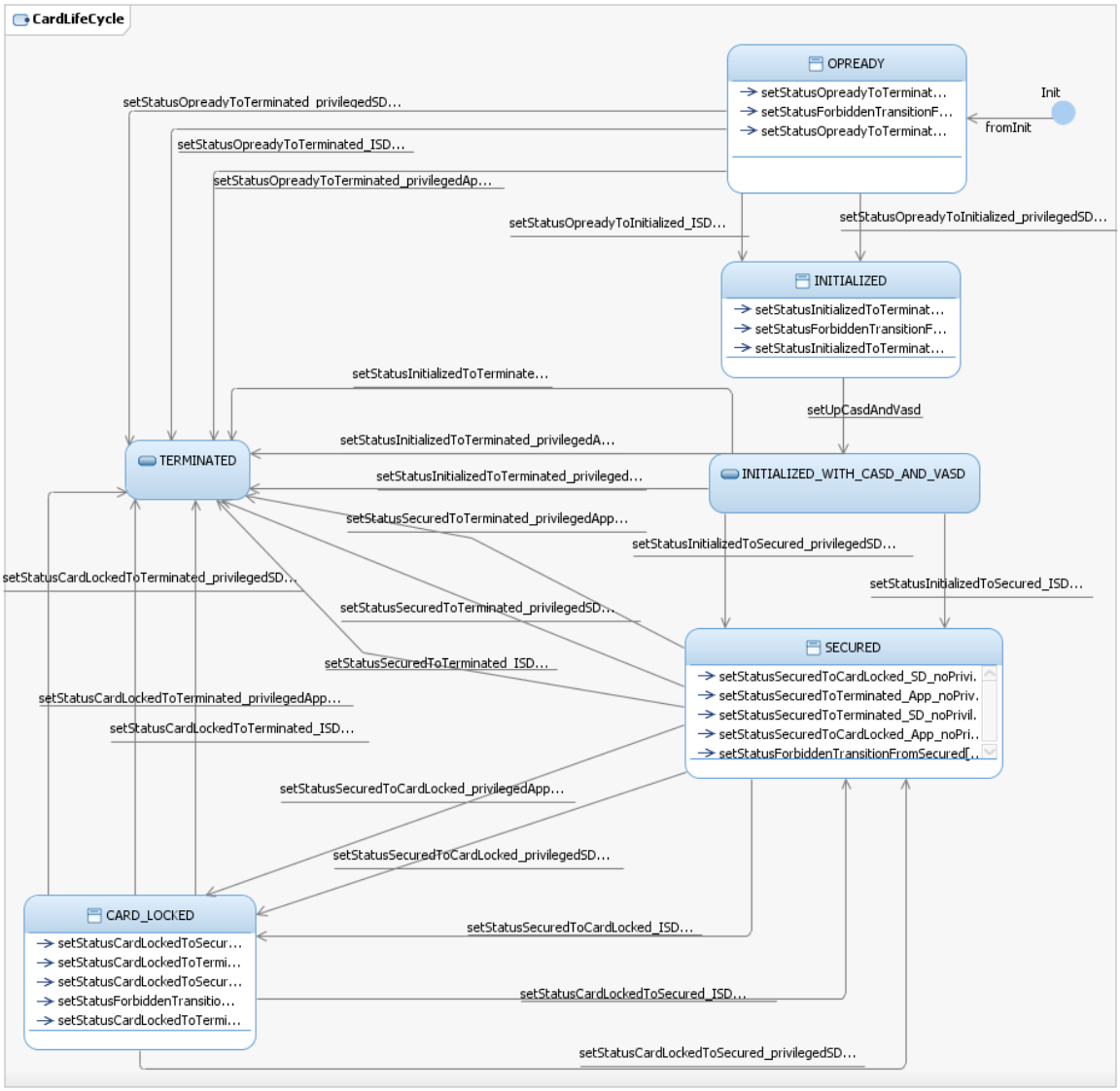

Figure 4.2: Statechart GP 2.2 UICC

Here are statistics about the model:

Nb of operations	Nb of usable behaviours	Nb Card Life Cycle behaviours
9	135	37

4.1.2 Global Platform Security Properties to schemas

In this section we are going to discuss about the security properties we have considered for GP. Due to confidentiality we have chosen general security properties that are valid for all

credit cards, and not only for GP. We have worked on the subscope of GP, Card Life Cycle and the APDU_setStatus operation allowing the card's applications to change the card's status through *OP_READY, INITIALIZED, SECURED, CARD_LOCKED and TERMINATED*. Each application has privileges for changing the card's status. For instance in GP 2.1.1 only the *super application* **ISD** can render the card unusable (state *TERMINATED*). While for the new specification's evolution GP 2.2UICC, an application having the terminate privilege can terminate the card.

Security Property 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* APDU_setStatus *issued by a privileged application, then it should not be possible to revert to another state.*

Security Test Objective A scenario to test this security property can be described informally as: (i) select an application with the Card Terminate Privilege, (ii) set the status of the card to TERMINATED, (iii) try all operations (to see if they behave as predicted by the model, i.e. by returning a status word of error).

Test schema In order drive automatic generation, the Security Test Objective is formalized as a Test Schema thanks to the Smartesting Test Purpose language as follow:

```
for_each operation $OPERATION from any_operation,
    use any_operation at_least_once
      to_reach "self.selectedApp.privileges.cardTerminate = true" on_instance card then
    use APDU_setStatus
      to_reach "self.state=ALL_STATES::TERMINATED" on_instance card then
    use $OPERATION
```

Security Property 2

The second security property that we analyse 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*, via a* APDU_setStatus *command (if the application is an SD).*

Test Intention A scenario to test the nominal case of failure of this security property can be described informally as: (i) try for all card states different from TERMINATED, (ii) select any application without the Card Terminate Privilege, (iii) set the status of the card to TERMINATED.

Test Schema

```
for_each literal $STATE from OP_READY or INITIALIZED or SECURED or CARD_LOCKED,
    use any_operation any_number_of_times
      to_reach "self.selectedApp.privileges.cardTerminate = false
          and self.state = ALL_STATES::$STATE" on_instance card then
    use card.APDU_setStatus(_, _, CARD, TERMINATED, _)
```

4.1.3 POPS Case Study by the numbers

In this section we give results on the GP Card Life Cycle Scope. On one hand we give numerical results for the evolution from functional point of view. On the other hand we detail results w.r.t the tested security properties and the GP evolution.

Evolution in functional test suites for GP Card Life Cycle Scope

We have created models for the two versions on the Card Life Cycle Scope of GlobalPlatform 2.1.1 and 2.2 UICC Configuration. The GP UICC Configuration is a configuration of v2.2, standardizing the minimum interoperability for (U)SIM cards for supporting remote application management. For the first version we have identified **32** test targets. The second version contains **37** test targets. With the comparison module of SeTGaM we have identified **four** deleted, **nine** new, and **twenty-eight** unchanged requirements. For the first version we have obtained **twenty-seven** tests. Next we have applied the *SeTGaM* process on the evolved version, by classifying tests from the test suite of GP 2.1.1 model, which resulted in: **4** obsolete tests, **15** unimpacted and **8** to be re-tested (i.e. **8** updated). Thus, we had to generate tests only for **9** test targets, instead of **37**. Using the generation engine we have generated 9 **new** tests and obtained a total set of **32** tests.

Evolution in security test suites for GP Card Life Cycle Scope

For the first security property **9** tests were generated for GP 2.1.1, one for each operation. Since in the schema we did not specify from which state the generator should attend the state *TERMINATED* it took the first state *OP_READY* and sets the card to *TERMINATED* and then calls each $OPERATION. The second security property produced **4** tests for GP 2.1.1, one for each given state. Contrary to the first schema, here we force the generator to reach the state *TERMINATED* from each other state. When using the plugin EvoTest for selective test generation for testing security properties on GP 2.2UICC, we have obtained satisfying results on these two more general security properties. For the first one, all **9** tests were unimpacted and there were not any new to produce. For the second security property, the process resulted with **1** outdated, **3** unimpacted and **1** adapted test, since the outdated test covers a deleted requirement, but the TCS associated to this test was still existent.

4.2 GP 2.2 UICC Card Content Management

4.2.1 Functional model

The GP 2.2 Card Content Management model focuses on the GlobalPlatform commands behaviours that manage the loading, installation, configuration and deletion of applications on the card. Only one version of the specification (GP 2.2 UICC Configuration) has been modelled. It aims to see if the Test Purpose Language is expressive enough to cover the property and produce interesting tests.

Here is the list of operations represented in that model:

- **APDU_delete**,
- **APDU_installForExtradition**,
- **APDU_installForInstall**,
- **APDU_installForInstallAndMakeSelectable**,
- **APDU_installForLoad**,
- **APDU_installForMakeSelectable**,
- **APDU_installForPersonalization**,
- **APDU_installForRegistryUpdate**,
- **APDU_load**,
- APDU_manageChannel,
- APDU_select,
- APDU_setStatus,
- nominal_openSecureSession,
- nominal_setUpCasdAndVasd,
- nominal_APDU_putKeyDesLight.

Here are statistics about the model:

Nb of operations	Nb of usable behaviours	Nb of Card Content Management behaviours
15	1076	944

4.2.2 GP Card Content Management Security Property to Schemas

The Card Content Management (CCM) model has been done to experiment the Schemas possibilities to express Security Test Objectives issued from a Security Property. The Test Purpose language must be expressive enough to represent the Security Test Objectives. Next is an example of a Card Content Management Security Property, the related Security Test Objectives, and the Schemas representing each one of them.

Security Property

The UICC configuration mandates that if a Security Domain application (SD) has authorized management privilege, then it should be the only one amongst all SD that are indirectly associated to it.

Security Test Objectives

Objective 1 whenever a SD 'A' with authorized management is extradited to another SD 'B' (using an INSTALL[for extradition] command), then neither 'B' nor any SD on the path from 'B' to the root of its hierarchy (i.e. any SD to which 'B' is associated via the transitive closure of the SD association relation) can have authorized management. The test objective is to extradite a SD using INSTALL[for extradition] command, to different SD hierarchies with at least a SD with authorized management, and observe that the command is rejected.

Objective 2 whenever a SD 'A' is given authorized management (using a INSTALL[for registry update] command), then no SD on the path from 'A' to the root of its hierarchy can have authorized management. The test objective consists to give the authorized management to a SD using INSTALL[for registry update] command, this SD belonging to different SD hierarchies with at least a SD with authorized management, and observe that the command is rejected.

Objective 3 whenever a SD 'A' is installed with authorized management (using a INSTALL[for install] command), then no SD on the path from 'A' to the root of its hierarchy can have authorized management. The test objective consists to install a SD with authorized management using INSTALL[for install] command, under different SD hierarchies with at least a SD with authorized management, and observe that the command is rejected.

Schemas

To simplify the readability of the following schemas please consider that HIERARCHY_1, HIERARCHY_2, HIERARCHY_3 are the relevant hierarchies to test the concerned Security Property, with SD_01 as the leaf SD of the hierarchy. The corresponding schemas to the Test Objectives are:

Schema 1

```
for_each state $HIERARCHY from HIERARCHY_1 or HIERARCHY_2 or HIERARCHY_3,
    use any_operation any_number_of_times
      to_reach $HIERARCHY then
    use APDU_installForInstall
      to_reach "self.installedApps->exists(app : Application |
        app.aid = SD_02 and app.privileges.securityDomain = true)"
        on_instance instance_OPRE_card then
    use card.APDU_installForExtradition(SD_02, SD_01)
```

Schema 2

```
for_each state $HIERARCHY from HIERARCHY_1 or HIERARCHY_2 or HIERARCHY_3,
    use any_operation any_number_of_times to_reach $HIERARCHY then
    use card.APDU_installForRegistryUpdate(SD_01, authorized_management)
```

Schema 3

```
for_each state $HIERARCHY from HIERARCHY_1 or HIERARCHY_2 or HIERARCHY_3,
    use any_operation any_number_of_times to_reach $HIERARCHY then
    use card.APDU_installForInstall(SD_02, SD_01, authorized_management)
```

4.3 Feedback on the evaluation

This section provides the Gemalto evaluation and feedback. The first sub-section presents the calendar and elements which were validated. The second is the feedback of the final evaluation step.

4.3.1 Calendar and Evaluation purpose

This information came from the Validation plan defined in Task 1.3 and associated deliverable.

The first version delivers to Gemalto is the version 1.2. It was provided in May 2011. The evaluation purpose is:

- Evaluation of the usability, scalability and relevance of the test model,
- Installation of the tool and training on its usage,
- Preliminary report for this lite version.

The second version delivered to Gemalto is the version 1.3. This version takes into account the first feedback and feature of the roadmap. It was provided in September 2011. The evaluation purpose is:

- Installation of the additional EvoTest components,
- Final evaluation report on the tool and the methodology.

4.3.2 Feedback of the evaluation

This section corresponds to section 2.6.5 of the deliverable D1.3. This section is the conclusion of the WP7 evaluation approach. All details of the evaluation are allowing in the section 2.6 of the deliverable D1.3.

Several experiences has been made using model-based testing for smart cards software. Generally, the major drawback highlighted by the validation teams is the time spent for modeling and the maintenance of the models in case of specification evolutions. Although the organization of the generated test suites and the traceability are appreciable, the validation engineer prefers modifying the tests suites that the model itself.

The SecureChange project confirms these results on the modeling effort but at the same time demonstrates that, in case of change, it is easier to report a minor modification on the models than investigating the test suites to identify the place for modification. This an important feature for smart cards platforms. Generally, a smart card manufacturer develops and maintains few platforms (called baselines), traditionally one per market sector. Then several branches are developed corresponding to family of products. This means that the software that constitutes the platforms, such as Globalplatform implementation, will be concerned by specification evolutions or small modification for customization purpose. Therefore, the SecureChange model-based technology will be helpfully to report the changes on the models developed once that on the million of tests that are maintained in the tests benches. This is why the effort must continue to improve the usability of the modeling and we advocate that this technology must be planned and used early enough in the product life cycle.

Although expertise in UML / OCL seems to be an important requirement, it is rapidly damped in time as with any programming language. Usually the R&D people are either already familiar with these languages or have the scientific background needed to be quickly trained.

One of the main advantage of this technology is its use in the context of Common Criteria certification. Generally, if a product has been CC certified, any modification requires at least a "delta" certification. This requires to the developer providing the evaluator with evidences on the impact of the change and in particular, how he perform the testing on the modified product. It is clear that the SeTGaM methodology and the tool will facilitate this step with the categorization of the test suites and the corresponding reports.

5 Results of test campaign on HOME Case study

This chapter discusses an application of the model-driven testing methodology developed in SecureChange. The methodology is applied to the HOMES case study, where a home gateway is in charge for enforcing various security requirements, and providing services to its user. First, a system model and test model, together with a compact requirements model for the case study are developed. Afterwards, we will check how our method behaves if the system and its requirements evolve, and investigate the expected advantages concerning the evolution steps. The underlying metamodel is explained in deliverable D7.3 and in [4] in more details.

5.1 Business Case

The general environment is a home network wherein any connecting device shall be assessed by the operator (see Figure 5.1). Once the device is accepted we may consider the following interactions: the customer shall access an operator's service store - which is indeed a service installed in the customer's home gateway - and select any service from the catalogue. Services are offered by third party service providers (SP). Once the customer selects a service, it is redirected to the proper service provider to proceed with the purchase. The SP shall deliver the service to the customer's home gateway. The delivered service shall then be deployed as a web service client.

The operator requires a certain level of quality from the services offered by SPs. By default, once the operator and the SP sign a commercial agreement, the operator trusts the SP and its services. This trust is translated into a basic level of control over the SP and its services, i.e., the operator does not impose strict constraints to the services. Nevertheless, this trust might degrade over time. The operator shall degrade the trust on a certain SP because of several reasons:

1. Reports on bad quality of the offered services: some SP may receive a noticeable amount of complaints from customers about malfunctions or low quality of the service, etc.
2. Critical bugs in services or even malware,
3. Non delivery of services.

In the following subsection we present the requirements model, the test model, and the system model for our business case. We follow a test–driven modeling approach where the test model initiates the modification of the system model.

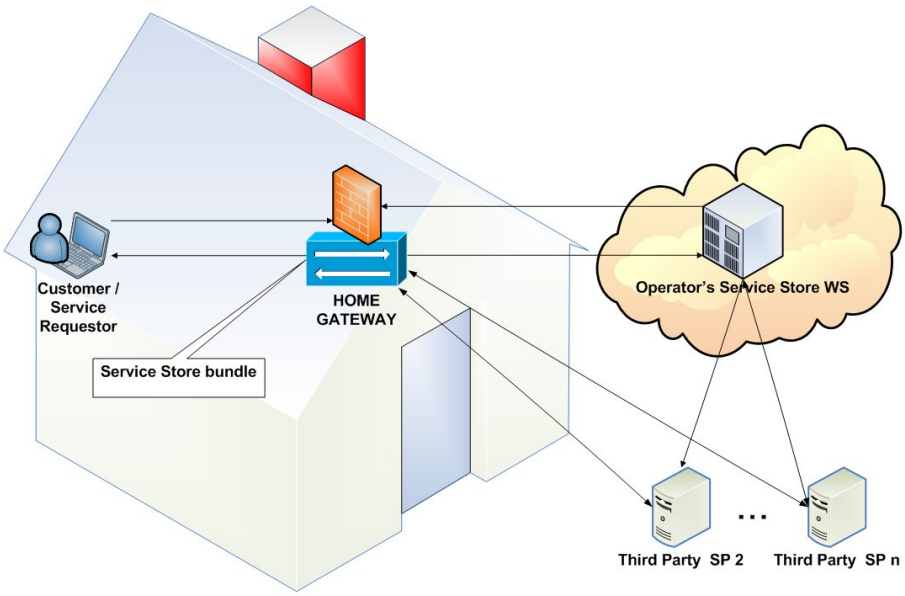

Figure 5.1: Home General environment

5.1.1 Requirements

The requirements model contains a hierarchy of functional requirements and security requirements. Note that this model will be changed at a later point to demonstrate the evolution functionality. Figure 5.2 presents the two base requirements that services can be purchased (`Req_1`), and trust may be degraded by the operator (`Req_2`). Note that we here use our own way of modeling requirements, however, this does not restrict the applicability of the requirements model developed in workpackage 3. For presentational reasons we chose to stick to our own representation.

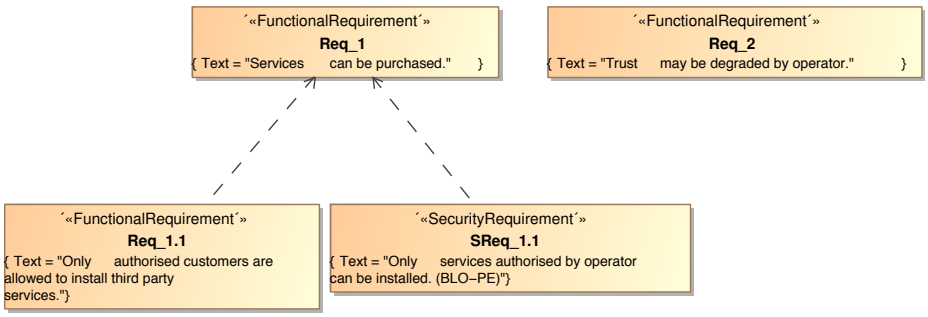

Figure 5.2: Home original requirements model

`Req_1` has two attached sub-requirements which means that the requirement cannot be fulfilled if one of its sub-requirements is violated. The sub-requirement `Req_1.1` describes that only authorised customers should be allowed to install third party services. The security requirement `SReq_1.1` prescribes that only services authorised by the operator should be allowed to be installed. Following our methodology, `SReq_1.1` is checked by executable assertions. In the following section, we define the test model containing tests to check the requirements.

5.1.2 Test Model

The test model contains two tests for verifying the requirements stated before. Note that only the requirements `Req_1`, `Req_1.1` and `SReq_1.1` are covered by the tests, and `Req_2` is omitted until now. The first test is depicted in Figure 5.3. It checks the functionality to browse the service store provided by the operator. It is expected that, on the one hand, not all customers are allowed to purchase services at all, and on the other hand, that only the services which are authorised by the operator can be installed.

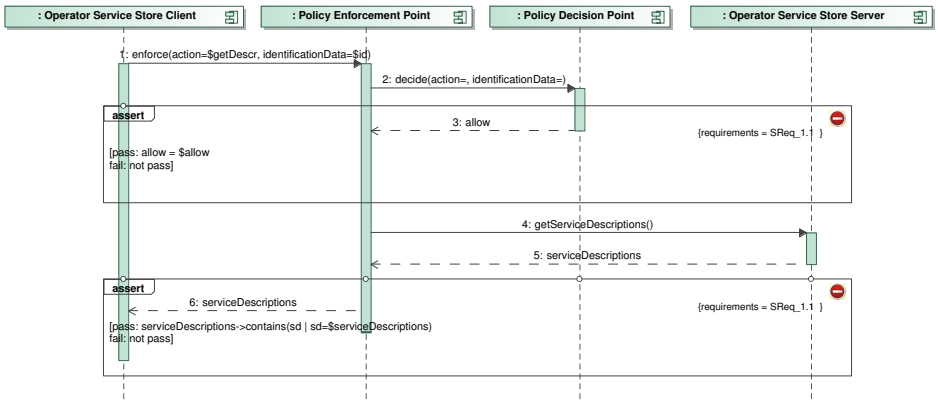

Figure 5.3: Test to browse service store and retrieve descriptions of available services

In the test, we have placed two assertions. The first assertion checks whether the policy decision point correctly permits or denies access to the service store, and the second assertion checks whether the correct service descriptions are returned by the service store. Both assertions are assigned to `SReq_1.1`, and the test itself is assigned to `Req_1.1`. The second test checks whether purchasing a specific service works as expected. The client requests a specific service, but this action first has to be permitted by the policy decision point. In case the installation is permitted, the purchase request is forwarded to the service provider. Figure 5.4 depicts the test.

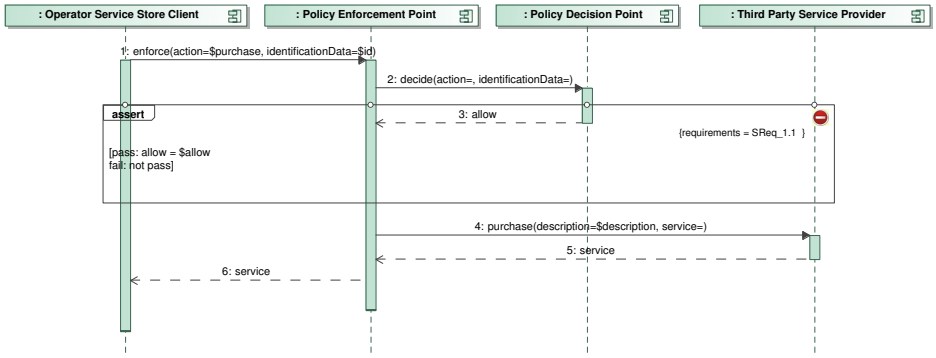

Figure 5.4: Test to purchase a service from a third party service provider

The test contains one assertion which checks whether the decision point permits the purchase as expected or not. This assertion is assigned to `SReq_1.1`. Note that these two tests are not sufficient for an extensive quality assurance of the presented system. We only present these two tests as an example here, however one may consider multiple further tests inspecting other aspects of the system. According to our metamodel (see D7.3 and [5]), also tests itself can be assigned to requirements. The test for browsing the service

store is assigned to `Req_1.1`, and the test for purchasing services is assigned to `Req_1`. This completes the relationships between tests and requirements. In the next section the system model is discussed.

5.1.3 System Model

The system model defines services and the interfaces among them. The services and their relationships are depicted in Figure 5.5. Most of the services were already used in the test model.

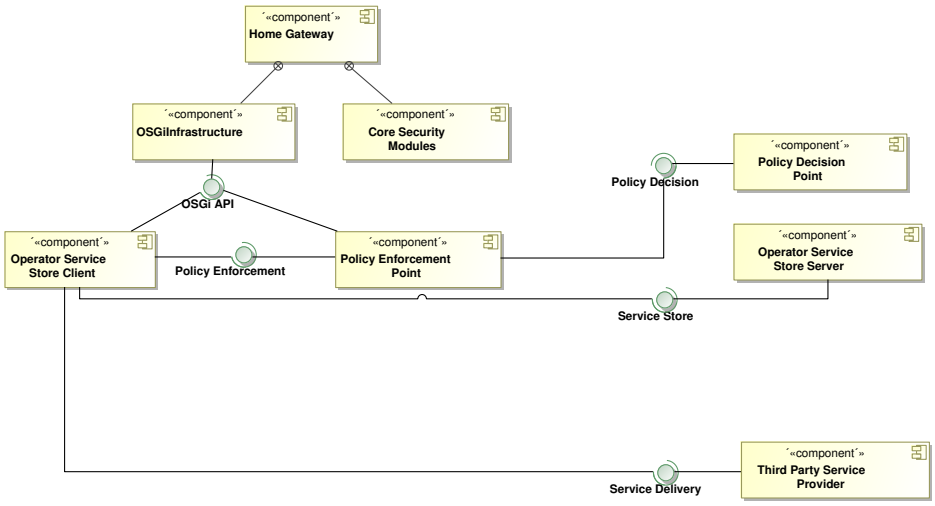

Figure 5.5: Home Services

The most relevant information are the interfaces because interfaces make the interactions among services explicit. This information, in turn, is used for propagating changes to the relevant model elements. Furthermore, since the tests only use services and interfaces which are available in the system model, an immediate consistency check is made while modelling tests.

5.2 Change Requirements

In this section we consider two different changes, and analyse how these changes are handled by the methodology developed in SecureChange. The two changed scenarios we take into account are: "Core security module update" and "Bundle lifecycle operations". The core security modules are services contained within the Home Gateway (cf. Figure 5.5). The function of core security modules is to facilitate basic security functionality for the Home Gateway itself, for instance the security assessment of connected devices. If the core security modules are updated, for instance because a new device security assessment method should be used (`SReq_1.2` in Figure 5.6), this change propagates to all tests which depend on the core security modules. This propagation is used to determine which tests are affected by a change in order to determine exactly those tests which possibly need to be rerun. In the case of adding `SReq_1.2`, there is no test which is directly affected by it. Nevertheless, this does not mean that no action has to be taken. In fact, at least one test should be defined which validates `SReq_1.2` – we omit this step here. Concerning the change requirement "bundle lifecycle operations", we introduce a further security requirement which enforces the execution of a non-repudiation protocol for the service-purchase process (`SReq_2.1`).

The existing functional requirement, that "trust! in a service provider may be degraded by the operator" depends on the new security requirement (see Figure 5.6).

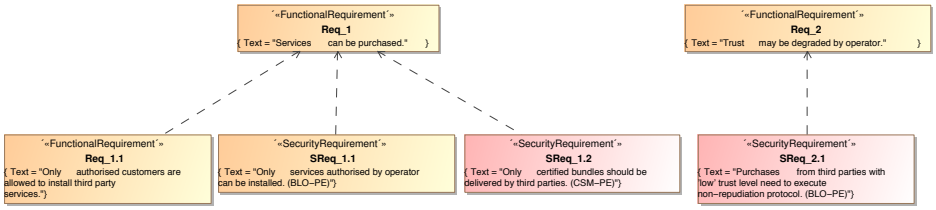

Figure 5.6: Requirements Home model after changes

Until now, no test is defined which checks whether `SReq_2.1` is fulfilled or not. This means that we, first of all, create a new test which includes according checks. In Figure 5.7, a test is displayed which performs a service purchase, and checks the new requirement. The second assertion of that test checks the recipient information which is sent to the trusted third party. This assertion is associated to `SReq_2.1`. Furthermore, the test makes use of two additional services which are not present in the system model so far.

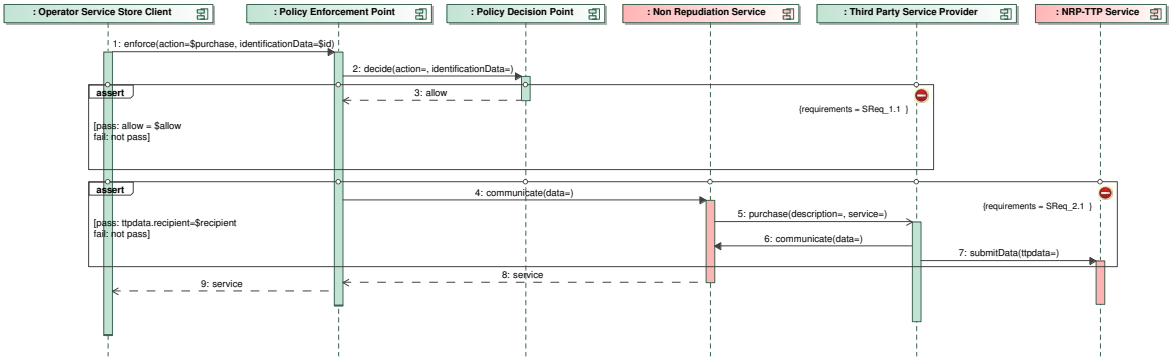

Figure 5.7: Test to purchase a service from a third party service provider and using a non-repudiation protocol – only executed for service providers with a low trust level

Each test can be in one of three possible states described by a state machine (cf. Figure 5.8). This state machine regulates how tests are organised into different test suites by assigning every test a specific type when anywhere in the model an element is added, modified or deleted. The test just added is currently in state **notExecutable**. The next step is to add the missing services to the system model (Figure 5.9). As soon as the services are deployed, the test transitions to state executable (triggered by `modifyService()`). Now, that `SReq_2.1` is in state **underTest**, also `Req_2` can be considered being under test. Furthermore, also for services and requirements a state machine can be defined describing its lifecycle, and receiving and emitting events upon changes. However, in the present description we only focus on the lifecycle of tests. As described in [4, 5] , and deliverable 7.3, the test lifecycle keeps track of changes on various model elements. A specifically important feature is the `Type` of a test. As described in previous deliverables, this type can be either *evolution*, *regression*, *stagnation* or *obsolete*. The Type helps to determine which tests were affected by a change, plus need to be rerun, and which tests can be disregarded. Considering the above mentioned example of the new purchase test, it transitions to state executable as soon as the service is deployed (triggered by `modifyService()`). This transition has the effect to set the `Type` of the test to evolution, which means that it was affected by the last change. The already existing test to browse the service store instead, was already in

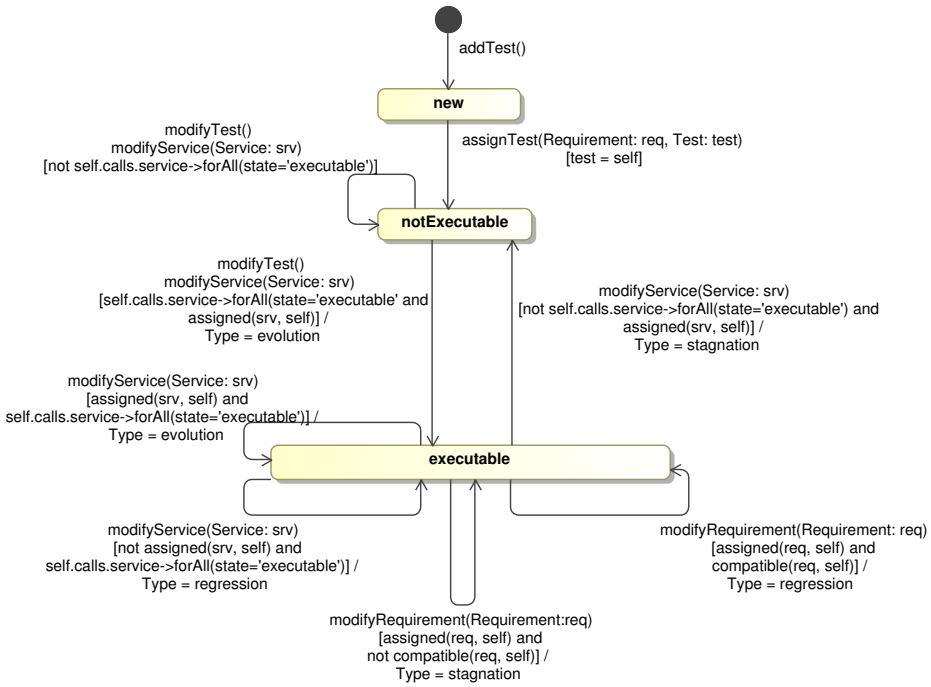

Figure 5.8: Test Lifecycle: depending on what is changed in the model, a test can be affected

state **executable** before. However, the test has no relation to the two new services. This means that the transition *modifyService()* will be triggered for the test when the services are deployed. But, since it is not assigned to one of the two services its `Type` will be set to regression.

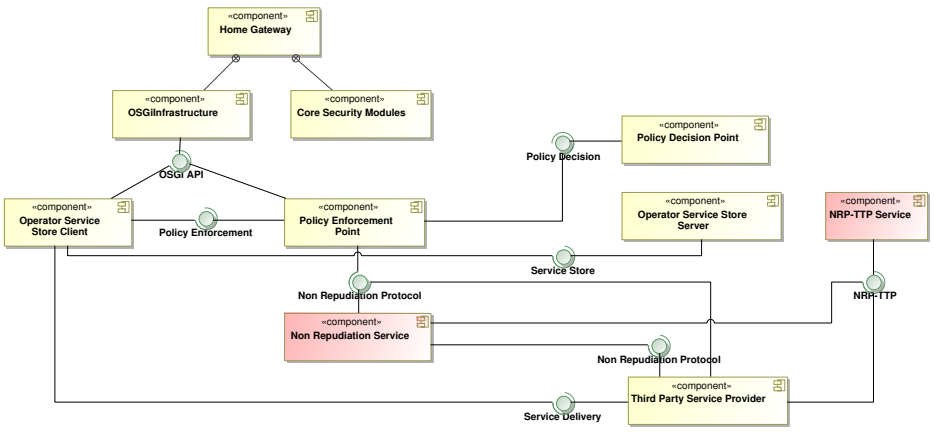

Figure 5.9: HOMES services with additional non-repudiation components

This is described by the transition *modifyService(Service: srv)* and its guard which is as follows:

```
[not assigned(srv, self) and
self.calls.service->forAll(state='executable')]
```

This way, all changes on requirements or services can be reflected accordingly at the tests.

5.3 Evolution Process

The evolution process regulates how changes are propagated, and how test suites are finally created. The process is always initiated by a change of a model element. Changes can be either the creation of a model element (*add-operations*), the deletion of a model element (*remove-operations*), or the modification of a model element (*modify-operations*). Figure 5.10 depicts the process. A change triggers an event involving state transitions in state machines which may generate new events. This may cause state changes of affected model elements, e.g., based on a service modification an assigned test state could transition from executable to **notExecutable**. This induces further changes to obtain consistent and executable models. After the model is changed, the **consistency** and **executability** of the new model need to be checked. This may imply new modifications in case the check does not evaluate to true. The process of changing tests, or any other part of the model is repeated until the model is *consistent*, i.e., it contains no internal contradictions, and can therefore be transformed to executable test code.

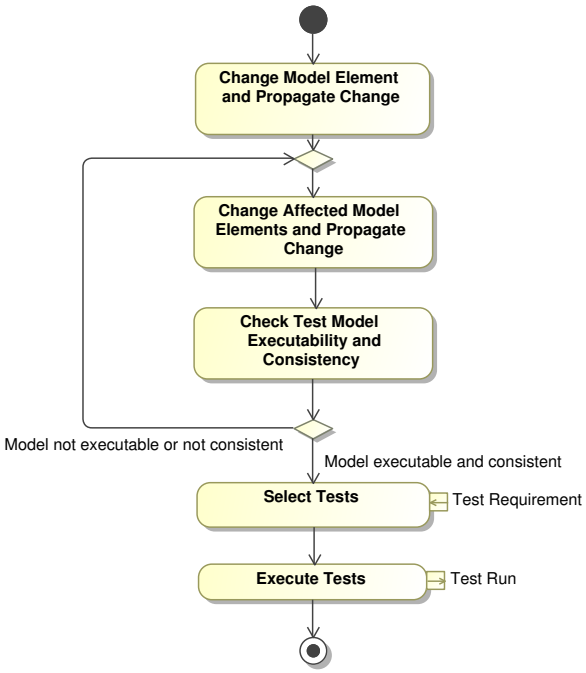

Figure 5.10: Evolution process which handles changes on the model

The executability and consistency can be checked by OCL. The following OCL query checks whether the parameters in calls are compatible with the parameters of the referred operations.

```
context Model:
  Call::allInstances.parameters->
    forAll{ param | param.data.class
      = self.operation.class }
```

Afterwards, tests are selected, i.e., a concrete test suite is computed from the set of all tests, based on test requirements. Test requirements define test selection criteria in OCL and typically consider the type of a test and the state of other model artifacts. A very general regression test selects all tests that are supposed to pass, i.e., tests of type *evolution* or *regression* and is as follows:

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

This way, specific test suites can be created and kept up-to-date automatically. The methodology informs the test engineer which tests were affected by the last change (*evolution*), which were unaffected by the last change (*regression*), and which tests should explicitly fail, e.g. because a service or requirement is not supported any more (*stagnation*). Additionally, all deleted tests are assigned a specific type and form the *obsolete* suite.

6 Conclusion / Discussion

In this deliverable, we have given the final status of the tool's development and methodology done in WP7. These results address a model-based testing approach for evolution in the context of security engineering for lifelong evolving systems. The novelty of the results can be categorized into two levels:

- management of evolutions in the model-based testing process with SeTGaM;
- capability to drive automated test generation from the model in order to ensure Security Test Objectives with SBTG.

These results have been fully supported by a research prototype (EvoTest). This demonstrator is based on Smartesting MBT technologies and integrates the WP7 results. It takes into account change analysis, test generation based on evolution, test generation for security properties, classification and publication of theses tests in a test repository depending of the associated test status. *EvoTest* has been experimented on the POPS case study on a real-size smart card application (GlobalPlatform GP2.2 UICC) with encouraging results.

Moreover, a semi-automatic approach, named TTS has been defined, providing model-driven testing methodology.

These results of WP7 are integrated in a global SecureChange methodology and tool chain, starting from requirements and formal definition of security needs, including model-based testing and verification of the system under test. The re-usability of security model and change analysis is clearly defined and provides a global work-flow for the SecureChange process.

Finally, SecureChange WP7 work leads to several perspectives :

Regarding the exploitation of the work The results obtained for managing evolution in a model-based testing process (SeTGaM) and to drive test generation from security test objectives (SBTG) brings added value to the model-based testing solution. Smartesting intends to industrialize these results in near future in order to integrate them into the Smartesting CertifyIt product. The schedule and road-map are still not defined. However SBTG integration into Smartesting CertifyIt will certainly be planed for the next 12 months after the end of the SecureChange project.

Regarding scientific perspectives The results obtained in SecureChange WP7 lead to interesting perspectives:

1. At evolution management level in the MBT process based on models analyze and tests classification. This analyze is extension of existing works on data and control dependencies. The tests classification is based on regression testing works, which are refined in order to have more precise results.

2. At model-based security testing level, the current approach (SBTG) addresses clearly a restricted part of the problem, focussing mainly on the conformance testing of security functions. An important extension would be to address vulnerability testing based on test pattern and dedicated test generation models. This will be a new research project.

To summarize, SecureChange project, with all its work packages, allowed model-based testing techniques and challenge to strongly cooperate with security and change modelling issues, as well as requirements management issues. For WP7 partners, and considering the results obtain around SeTGaM, SBTG and TTS method, this project is a success and pave the way to better address market needs in the context of security engineering for lifelong evolving systems.

Bibliography

[1] Global platform specification. http://www.globalplatform.org/specificationscard.asp, May 2011.

[2] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19^{th} International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007.

[3] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963/2008 of *Lecture Notes in Computer Science*, chapter 24, pages 337–340. Springer Berlin, Berlin, Heidelberg, April 2008.

[4] Michael Felderer, Berthold Agreiter, and Ruth Breu. Evolution of security requirements tests for service-centric systems. In *ESSoS 2011*, pages 181–194, 2011.

[5] Michael Felderer, Berthold Agreiter, and Ruth Breu. Managing evolution of service centric systems by test models. In *IASTED International Conference on Software Engineering 2011*, 2011.

[6] Elizabeta Fourneret and Fabrice Bouquet. Impact Analysis for UML/OCL Statechart diagrams based on Dependence Algorithms for Evolving Critical Software. In *10th International Conference ETAI 2011*, Ohrid, Macedonia, 2011.

[7] Elizabeta Fourneret, Fabrice Bouquet, Frédéric Dadeau, and Stéphane Debricon. Selective Test Generation Method for Evolving Critical Systems. In Per Runeson and Shin Yoo, editors, *1st International Workshop on Regression Testing*, pages 125 – 134, Berlin, Germany, July 2011.

[8] Elizabeta Fourneret, Martin Ochoa, Fabrice Bouquet, Julien Botella, Jan Jürjens, and Parvaneh Yousefi. Model-based security verification and testing for smart-cards. In *ARES 2011, 6th Int. Conf. on Availability, Reliability and Security*, Vienna, Austria, August 2011.

[9] Fabio Massacci, Fabrice Bouquet, Elizabeta Fourneret, Jan Jurjens, Mass Lund, Sébastien Madelénat, JanTobias Muehlberg, Federica Paci, Stéphane Paul, Frank Piessens, Bjornar Solhaug, and Sven Wenzel. Orchestrating Security and System Engineering for Evolving Systems. In Witold Abramowicz, Ignacio M. Llorente, Mike Surridge, Andrea Zisman, and Julien Vayssière, editors, *4th European Conference, Towards a Service-Based Internet - ServiceWave 2011*, volume 6994 of *Lecture Notes in Computer Science*, pages 134–143, Poznan, Poland, 2011.