D 7.2: How to integrate evolution into a model-based testing approach

Fabrice BOUQUET, Frédéric DADEAU, Stéphane DEBRICON, Elizabeta FOURNERET, Jacques JULLIAND, Pierre-Alain MASSON, Philippe PAQUELIER (INR), Julien BOTELLA, Bruno LEGEARD (SMA), Berthold AGREITER, Michael FELDERER (UIB), Julien BERNET, Boutheina CHETALI, Quang-Huy NGUYEN (GTO), Alvaro ARMENTEROS PACHECO (TID), Zoltan MICSKEI (BME), Fabio MASSACCI and Elisa CHIARANI (UNITN)

Document Information

Document Number	D7.2.
Document Title	How to integrate evolution into a model-based testing approach
Version	1.7
Status	Final
Work Package	WP 7
Deliverable Type	Report
Contractual Date of Delivery	M12.
Actual Date of Delivery	19 January 2010
Responsible Unit	INR
Contributors	INR, UIB, SMA, BME, GTO, TID, UNITN
Keyword List	Model-based, Test
Dissemination	level PU

Document change record

Version	Date	Status	Author (Unit)	Description
0.1	2009/10/12	Working	F. BOUQUET (INR)	Plan and document
0.2	2009/10/30	Working	F. BOUQUET (INR), F. DADEAU (INR), S. DEBRICON (INR), E. FOURNERET (INR), J. JULLIAND (INR), B. LEGEARD (SMA), P.A. MASSON and P. PAQUELIER (INR)	Methods
0.3	2009/11/19	Working	M. FELDERER and B. AGREITER (UIB) and Alvaro ARMENTEROS PACHECO (TID)	Home Gateways Elements & Telling Story
0.4	2009/11/24	Working	J. BOTELLA (SMA) and E. FOURNERET (INR)	GlobalPlateform
0.5	2009/12/14	Working	all authors	Reviews
1.0	2009/12/19	Draft	F. BOUQUET (INR)	First non author Review
1.1	2009/12/20	Draft	J. BERNET, Boutheina CHETALI, Quang-Huy NGUYEN (GTO)	Review
1.2	2009/12/23	Draft	Z. MISCKEI (BME)	Review
1.3	2010/01/11	Draft	F. MASSACCI and E. CHIARANI (UNITN)	First Quality Check; minor remarks
1.4	2010/01/12	Draft	F. BOUQUET (INR)	Changes made according to first quality check and remarks
1.5	2010/01/18	Draft	B. AGREITER (UIB)	Typos
1.6	2010/01/19	Draft	F. MASSACCI and E. CHIARANI (UNITN)	Quality Check; minor remarks
1.7	2010/01/19	Final	F. BOUQUET (INR)	Changes made according to quality check and remarks

Executive summary

This deliverable is the result of the work realized in task 2 of Work Package 7. From the state of the art provided in task 1 and presented in Deliverable 7.1, we propose to extend the methods and the tools in order to take the evolution into account. We will generate tests to emphasize the correctness of the system w.r.t. evolution, based on requirements and model changes.

This document presents the process and the concepts for a Model-Based Testing approach used to compute tests. We introduce a motivating example called Bob's adventure. We propose with this simple example to illustrate several situations of evolution, and their impact on our methods. We conclude with the preliminary results of the collaboration between WP7 and WP1 on two case studies: Home Gateway provided by Telefonica and GlobalPlatform (called POPS) provided by Gemalto.

Index

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
SUT	System Under Test
TTS	Telling TestStories

Table 1: Abbreviations used in the document

Glossary

Term	Definition
Adapter	Piece of code to concretize logical tests into physical tests
Evolution Test Suite	Test Suite targeting SUT evolutions
Logical Test	See Test Case
Model Layer	Link of model's operations in Test Scenario
Model-Based Testing	Process to generate tests from a behavioural model of the SUT
Status of Test Case	new, obsolete (outdated, failed), adapted, reusable (reexecuted, 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
Stagnation Test Suite	Test Suite targeting removed part of the SUT
System Model	Model of the SUT used for development
Test Case	A finite set of test steps
Test Intention	User's view of requirement into Test Scenario
Test Model	Dedicated model for test capturing the expected SUT behaviour
Test Suite	A finite set of test cases
Test Script	Executable version of a test case
Test Scenario	A test generation strategy
Test Sequence	See Test case
Test Step	Operation's call or verdict computation
Test Strategy	Formalization of test generation criteria
Test Objective	High level test intention

Table 2: Glossary

1. Introduction

This deliverable is the result of the work realized in task 2 of Work Package 7. From the state of the art provided in task 1 and presented in Deliverable 7.1, we propose to extend the methods and the tools in order to take the evolution into account. We will generate tests to emphasize the correctness of the system w.r.t. evolution, based on requirements and model changes.

This document begins with the process and the concepts for a Model-Based Testing approach used to compute tests. In Section 3, we present a motivating example called Bob's adventure. We propose with this simple example to illustrate several situations of evolution, and their impact on our methods. The kinds of evolution to consider are explained in Section 4 and we give in Section 5 the impact on test. These two sections provide the vocabulary for testing evolution. Section 6 presents the methods and processes used for testing evolution from two points of view. The first one proceeds by comparing two models that integrate evolution and the second one is based on scenarios for security. In Section 7, we present the preliminary results of the collaboration between WP7 and WP1 on two case studies: Home Gateway provided by Telefonica and GlobalPlatform (called POPS) provided by Gemalto.

2. MBT Process and Concepts

We use this section to define the terminology used throughout the deliverable and describe the general process of model-based testing. See [UPL06] for a detailed description of MBT approaches' taxonomy.

2.1 MBT Process

A *test suite* is a finite set of test cases. A *test case* is a finite structure of input and expected output: a pair of input and output in the case of deterministic transformative systems, a sequence of input and output in the case of deterministic reactive systems, and a tree or a graph in the case of non-deterministic reactive systems. The input part of a test case is called *test input*.

By *model-based testing* we consider: the processes and the techniques for automatic derivation of abstract test cases from abstract formal models, the generation of concrete tests from abstract tests, and the manual or automated execution of the resulting concrete test cases. Models must be formal and complete enough to allow, in principle, a machine to derive tests from these models, which is not the case for use case diagrams, or static business domain models like class diagrams for instance.

The generic process of model-based testing described in Figure 2.1 then proceeds as follows.

Step 1. A model of the System Under Test (SUT) is built from informal requirements or existing specification documents. This model is often called a *test model*, because the abstraction level and the focus of the model is directly linked with testing objectives. In some cases, the test model could be also the design model of the system under test, but it is important to have some independence between the model used for test generation and any development models, so that the kind of errors possibly introduced in the development model are not propagated into the generated tests. For this reason, it is usual to develop a test-specific model directly from the informal requirements, or to reuse just a few aspects of the development model as basis for a test model, which is then validated against informal requirements. Validating the model means that the requirements themselves are scrutinised for consistency and completeness and this often exposes requirements errors.

On the one hand in Model-Based Testing, the necessity to validate the model implies that the model must be simpler (more abstract) than the SUT, or at least easier to check, modify and maintain. Otherwise, the efforts of validating the model would equal the efforts of validating the SUT. Throughout this paper, we will use the term *abstraction* to denote both: the deliberate omission of details in the model and the encapsulation of detail by means of high-level language constructs. The test model can reside at various levels of abstraction. The most abstract variant maps each possible input to the output 'no exception' or 'no crash'.

On the other hand, the model must be sufficiently precise to serve as a basis for the generation of 'meaningful' test cases. This means that tests generated from the model should be complete enough in terms of actions, input parameters and expected results in order to provide real added value. If not, the test design job still has to be done manually, and there is little added value in generating tests from the model.

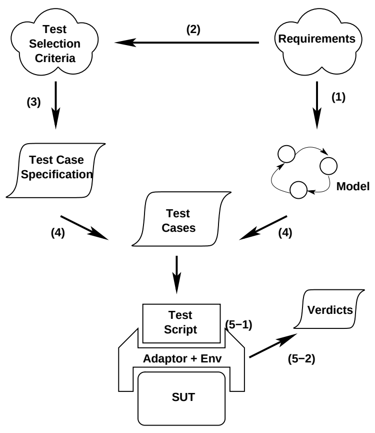


Figure 2.1: The Process of Model-Based Testing

Step 2. Test selection criteria are chosen to guide the automatic test generation so that it produces a 'good' test suite – one that fulfils the test policy defined for the SUT. Defining a clear test policy and test objectives for a system and associated development project is part of all testing methods such as TMap(r)[TMa08] or the ISTQB Manuals [IST10] that are widely used in industry. In such methods, the test policy and test objectives are formalized into Test Plan documents, which define the scope of testing and the various testing strategies and techniques that will be used in the project for each testing level (e.g. unit testing, integration testing, system testing, acceptance testing).

Test selection criteria can relate to a given functionality of the system (requirements-based test selection criteria), to the structure of the test model (state coverage, transition coverage, def-use coverage), to data coverage heuristics (pair-wise, boundary value), to stochastic characterisations such as pure randomness or user profiles, to properties of the environment, and they can also relate to a well-defined set of faults.

Step 3. Test selection criteria are then transformed into *test case specifications.* Test case specifications formalise the notion of test selection criteria and render them operational: given a model and a test case specification, some automatic test case generator must be capable of deriving a test suite (see Step 4). For instance, 'state coverage' of a finite state machine (FSM) might translate into a set of test case specifications such as $\{reach\ s0, reach\ s1, reach\ s2, \ldots\}$, where $s0, s1, s2, \ldots$ are all the states of the FSM. A test case specification is a high level description of a desired test case.

Step 4. Once the model and the test case specification are defined, a test suite is *generated.* The set of test cases that satisfy a test case specification with respect to the model, can be empty, in which case we say that the test case specification is *unsatisfiable.* Usually, there are several test cases that satisfy it, and the test case generator will choose just one of those test cases. Some test generators may spend significant effort in minimizing the test suite, so that a small number of generated test cases cover a large number of test case specifications.

Step 5. Once the test suite has been generated, the test cases are *run.* Test execution may be manual - i.e. by a physical person - or may be automated by a *test execution environment* that provides facilities to automatically execute the tests and record test verdicts. Sometimes, especially for non-deterministic systems, the generation and running of the tests are done-tailed together, which is called *online* testing.

Running a test case includes several steps: recall that model and SUT reside at different levels of abstraction, and that these different levels must be bridged. For example, an abstract test case for a bookshop website might be $checkPrice(WarAndPeace) = \$19.50$, where $checkPrice$ is the name of the webservice to be used, $WarAndPeace$ is the book to be queried, and \$19.50 is the expected result. *Executing a test case* then starts by concretising the test inputs (e.g., to obtain a detailed web services call) and sending that concrete data to the SUT (see step 5-1 in Figure 2.1). Secondly, the resulting concrete output of the SUT (e.g., a page of XML) must be captured and must then be abstracted to obtain the high-level expected result (a price) that can then be compared against the expected result (step 5-2 in Figure 2.1). We call the component that performs the concretization of test inputs and abstraction of test outputs the *adaptor*, because it adapts the abstract test data to the concrete SUT interface.

The verdict can take the outcomes *pass, fail,* and *inconclusive.* A test *passes* if the expected and actual output conform. It *fails* if they do not, and it is *inconclusive* when this decision cannot be made (yet). Building the verdict can be inconclusive in the context of nondetermistic systems where the decision

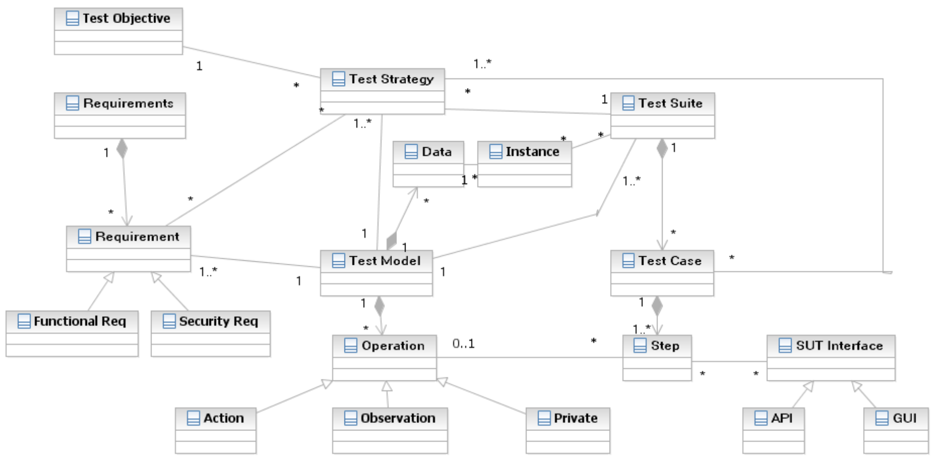


Figure 2.2: Conceptual Model of SecureChange approach of Model-Based Testing

must be postponed. For instance, certain permutations of output signals may be acceptable, and one has to wait until all output is given to judge this.

A *test script* is some executable code that executes a test case, abstracts the output of the SUT, and then builds the verdict. Note that an adaptor can be a concept and it is not necessarily a separate software component—it may be integrated within the test scripts.

To summarize, model-based testing involves the following major activities: building the model, defining test selection criteria and transforming them into operational test case specifications, generating tests, conceiving and setting up the adaptor component (if the generated tests are to be executed automatically, the adaptor is usually a significant proportion of the workload) and executing the tests on the SUT. The model of the SUT is used as the basis for test generation, but also serves to validate requirements and check their consistency.

2.2 MBT Artefacts

This section introduces a conceptual model (see Figure 2.2) that represents the main entities and artefacts that are used in the MBT process in the SecureChange project.

In the following list, we detail the main concepts used in the model-based testing approach in SecureChange:

Requirements They represent a collection of requirements, either functional or security requirements. The functional and security requirements repository is considered to be already present at the start of the test project.

Test Objective This is the high level test objective for testing. Regarding security testing, this includes definition of the testing need for security requirements.

Test Model The test model represents the expected behaviour of the System Under Test (SUT). In the SecureChange MBT approach, the test model is developed using the Unified Modeling Language (UML), with Class Diagrams, Instance Diagrams and State Machine Diagrams with formal behavioural specifications in OCL (Object Constraint Language). The test model is composed of operations (actions, observation or private/internal operation), and data (attributes).

Test Strategy Test strategies formalize the way tests are generated to fulfill the test objectives on the test model basis. Test strategies are coverage criteria (such as decision coverage, transition coverage, data coverage ...) and high level scenarios.

Test Suite A test suite is a collection of generated test cases based on the test model, a test data configuration (instances of test and environment data) and test strategies.

Test Case A test case is composed by steps and concrete test scripts implement generated test cases depending on the SUT interface (API or GUI).

The model-based testing process manages bidirectional traceability between requirements and generated tests. ”Bidirectional traceability” is the ability to trace link between two parts of the software development process with respect to each other. In the present case, this concerns management of the double link: requirements to tests and tests to requirements.

In the next chapter, we propose to illustrate this concept on the running example of this document.

3. Motivating example: Bob's Adventure

Bob's smartphone was bought from a Telefonica store and pre-installed with several applications. These applications were provided and secured by Telefonica, they were uploaded on the SIM card of the phone:

- The first application is a *Bank account manager*. It is allowing Bob to buy on the Internet or to interact directly with the bank account.
- There is also a *Taxi booking application*. As a customer of Telefonica, Bob has access to specific book service for taxi in Europe.
- The last one is an *Instant messaging system on TV*, allowing Bob or a Telefonica application to send messages to Bob's TV using the HOME gateway.
- Bob has also installed another application: *Flight monitor* provided by SkyTeam, allowing Bob to check for his business flights.

3.1 Functional description

In order to facilitate the reader's understanding of the example, the sequel of this section considers only a simplified view of a SIM card and the GlobalPlatform specification.

The SIM card was designed using GlobalPlatform specification and offers the following operations: access and block. These operations cannot be used directly by Bob, but are used by developers of applications that can be found on any smartphone. For instance, the developer of the Bank account manager wants to check if he can access the instant messaging application. Telefonica can decide to block an application like taxi booking if Telefonica don't found an arrangement with a taxi company. The blocked application cannot be used anymore by Bob.

Access is used to define if an application can access another application, the

operation returns 0 if the access is possible. It will return -1 and an error code otherwise.
Block will ask a security domain to block an application and return 0, or return -1 and an error code if not possible. A security domain is an entity containing applications or other security domains.

Following sections will present the detailed functional behavior of both operations as well as the new expected behavior.

3.1.1 Initial behavior

We will begin with the access operation. There are two parameters: caller application and called application, meaning that the caller application wants to access the called application.

Nominal case: return 0 if both applications have the same security domain or if the security domain of the calling application contains the security domain of the called application.

Error case: return -1 with an error code:

- *DIFF_SD* if applications have not the same security domain
- *SAME_APP* if access is called with the same application
- *CALLER_BLOCK* if the caller application is blocked
- *CALLED_BLOCK* if the called application is blocked

The block operation has also two parameters: a security domain and an application. The card asks a security domain to block an application.

Nominal case: return 0 if the security domain manages the application and if the application is not already blocked.

Error case: return -1 with an error code:

- *NOT_OWNER* if the security domain does not contain the application
- *ALREADY_BLOCKED* if the application is already blocked

3.1.2 Evolution

A new version of the GlobalPlatform specification is introduced and the behavior of the access operation changes. There is a revolution: the concept of access right is introduced, and a security domain stores operations' access rights. This is a simplified version of access management that only specify bidirectional access privilege.
The access operation now has three parameters: a security domain, a caller

application and a called application.
Nominal case: return 0 when applications have access rights.
Error case: return -1 with an error code:

- *SAME_APP* if access is called with the same application
- *CALLER_BLOCK* if the caller application is blocked
- *CALLED_BLOCK* if the called application is blocked

But there is also a small functional evolution on the block operation. The new version includes a search for a suitable security domain including children of the security domain given in parameter.

3.2 Functional Requirements

Functional requirements were extracted from the previous section. Requirements are business rules covering nominal and error cases.

Initial behavior

We identify nine requirements from the initial behavior. They are presented in Table 3.1.

id	description
ACCESS_OK_SAME_SD	access succeeds if both application have the same security domain
ACCESS_OK_SD_CONTAIN	access succeeds if the security domain of the calling application contains the security domain of the called application
ACCESS_ERROR_DIFF_SD	access fails if applications have not the same security domain (error DIFF_SD)
ACCESS_ERROR_SAME_APP	access fails if caller and called application are the same (error SAME_APP)
ACCESS_ERROR_CALLER_BLOCKED	access fails if caller application is blocked (error CALLER_BLOCKED)
ACCESS_ERROR_CALLED_BLOCKED	access fails if called application is blocked (error CALLED_BLOCKED)
BLOCK_OK_SD_CONT_APP	block succeeds if the security domain contains the application
BLOCK_ERROR_NOT_OWNER	block fails if the security domain does not contain the application (error NOT_OWNER)
BLOCK_ERROR_ALREADY_BLOCKED	block fails if the application is already blocked (error ALREADY_BLOCKED)

Table 3.1: Initial functional requirements

3.2.1 Evolutions

In Table 3.2, we add a new column to identify new (NEW), unchanged (UN) and removed (REM) requirements. There is one new requirement, seven requirements unchanged and three removed.

id	status	description
ACCESS_OK_SAME_SD	REM	access succeeds if both application have the same security domain
ACCESS_OK_SD_CONTAIN	REM	access succeeds if the security domain of the calling application contains the security domain of the called application
ACCESS_ERROR_DIFF_SD	REM	access fails if applications have not the same security domain (error DIFF_SD)
ACCESS_ERROR_SAME_APP	UN	access fails if caller and called application are the same (error SAME_APP)
ACCESS_ERROR_CALLER_BLOCKED	UN	access fails if caller application is blocked (error CALLER_BLOCKED)
ACCESS_ERROR_CALLED_BLOCKED	UN	access fails if called application is blocked (error CALLED_BLOCKED)
BLOCK_OK_SD_CONT_APP	UN	block succeeds if the security domain contains the application
BLOCK_ERROR_SD_NOT_OWNER	UN	block fails if the security domain does not contain the application (error NOT_OWNER)
BLOCK_ERROR_ALREADY_BLOCKED	UN	block fails if the application is already blocked (error ALREADY_BLOCKED)
BLOCK_OK_SD_CONT_SD	UN	block succeeds if the security domain contains a security domain containing the application
ACCESS_OK_RIGHT	NEW	access succeeds if security domain store access rights for applications

Table 3.2: Functional requirements capturing evolutions

The smart card's expected behavior can be defined by a static and a dynamic view. They are detailed in the following section.

3.3 Static View

The static view defines classes, enumerations and an initial state of the system (the smart card).

3.3.1 Initial behavior

We have created an UML model to describe the initial behavior. The UML class diagram contains three classes (see Figure 3.1):

1. the *Smardcard* class, which is the system under test
2. the *SecurityDomain* class, describing a security domain
3. the *Application* class.

The smart card contains at least one security domain. Each security domain can contain several security domains and several applications.

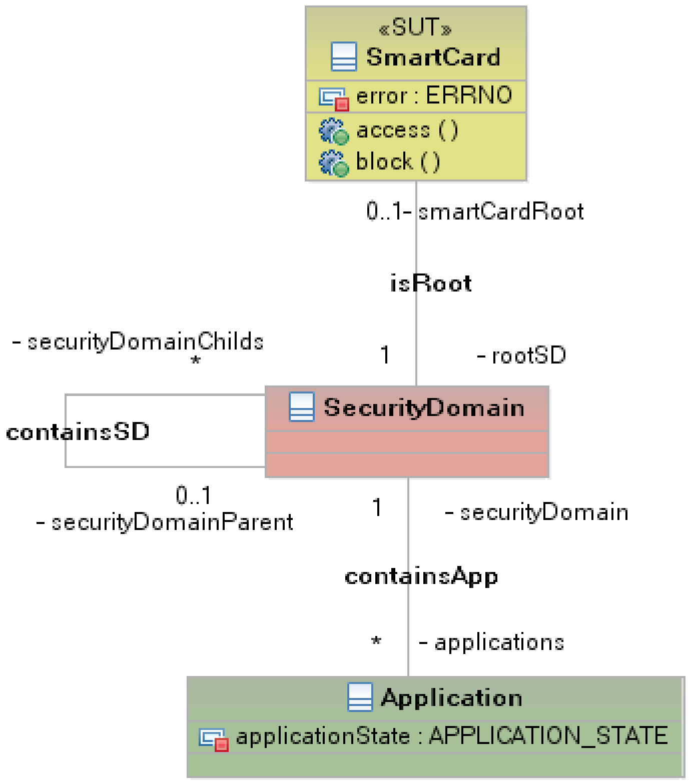


Figure 3.1: Initial class diagram

There are two enumerations described in Figure 3.2: ERRNO describing any error types returned by access and block operations. APPLICATION_STATE describing the current state of an application.

ERRNO:

- VOID: no error code
- DIFF_SD: Applications do not have the same security domain
- SAME_APP: Applications used for access call are the same
- CALLED_BLOCK: Called application is blocked
- CALLER_BLOCK: Caller application is blocked
- NOT_OWNER: The security domain does not contain the application
- ALREADY_BLOCK: The application cannot be blocked because it already is.

APPLICATION_STATE:

- AVAILABLE: Application can be used
- BLOCKED: Application is blocked

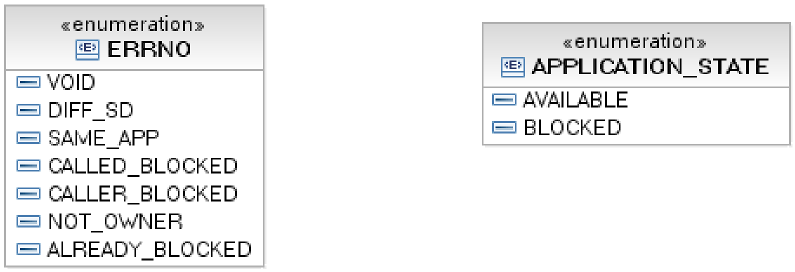


Figure 3.2: Enumerations

Figure 3.3 shows the content of the card. We can find the smart card and its root security domain (SIM security domain). There are also two main security domains: one for Telefonica and one for other provider. The Telefonica security domain contains the Home security domain (dealing with home applications). There are also all the applications listed in the functional description.

3.3.2 Evolution

The new class diagram (see Figure 3.4) contains a new class *AccessRight* storing rights between applications and managed by a security domain.

The ERRNO enumerate contains the new value, ACCESS_REFUSED, as presented in Figure 3.5, meaning that two applications cannot access each other.

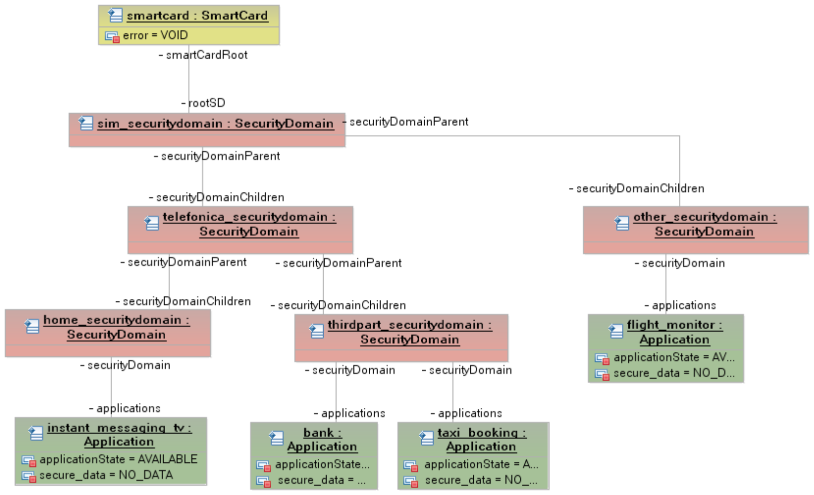


Figure 3.3: Initial smart card content

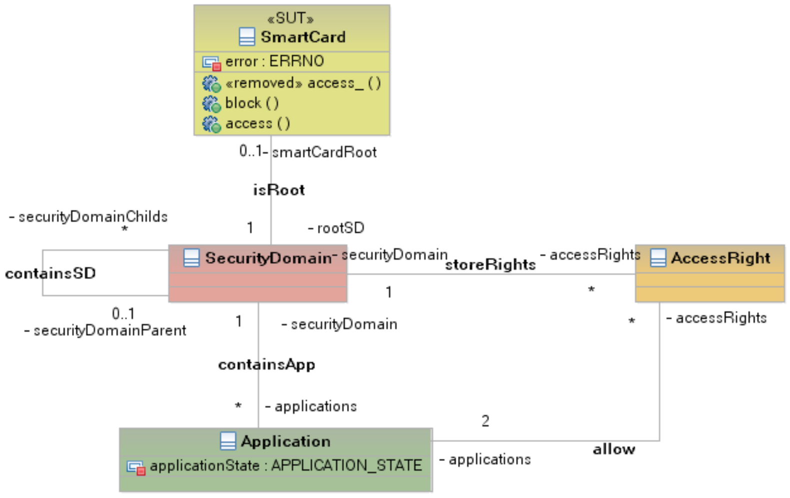


Figure 3.4: New class diagram

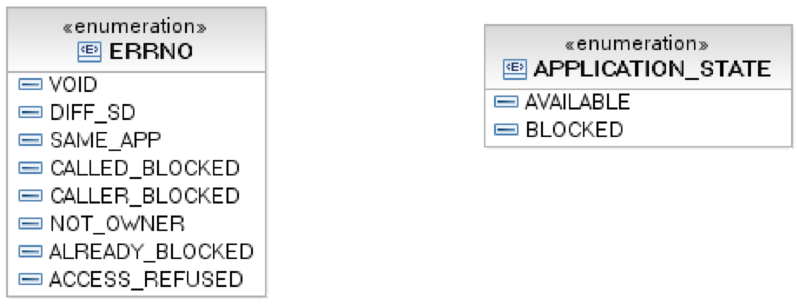


Figure 3.5: New enumerations

Figure 3.6 presents the smart card containing a new object. The object called *AccessRightInstance* is managed by the Telefonica security domain. *AccessRightInstance* is linked to the bank application and the instant TV messaging application.

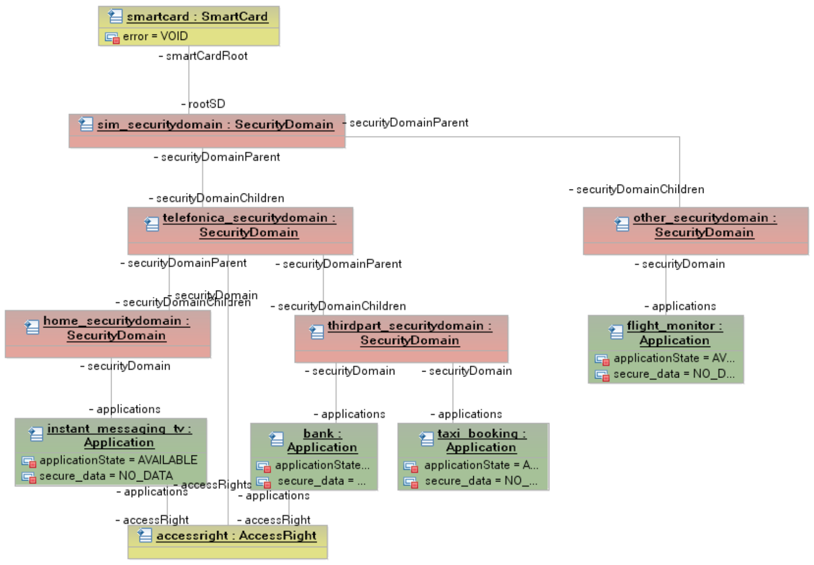


Figure 3.6: New initial smart card content

3.4 Dynamic View

The dynamic view of the system is the actual behavior of the card when dealing with access and block operations.

In Figure 3.7, the *access()* operation is modeled with the OCL language, it contains a tag (REQ) to specify what part of the code corresponds to a requirement.

```
if in_caller_app.applicationState = APPLICATION_STATE::BLOCKED then
        ---@REQ: ACCESS_ERROR_CALLER_BLOCKED
        self.error=ERRNO::CALLER_BLOCKED and
        result = -1
else
        if in_called_app.applicationState = APPLICATION_STATE::
            BLOCKED then
                ---@REQ: ACCESS_ERROR_CALLED_BLOCKED
                self.error = ERRNO::CALLED_BLOCKED and
                result = -1
        else
                if in_caller_app.securityDomain = in_called_app.
                    securityDomain then
                        if in_caller_app = in_called_app then
                                ---@REQ: ACCESS_ERROR_SAME_APP
                                self.error = ERRNO::SAME_APP and
                                result = -1
                        else
                                ---@REQ: ACCESS_OK_SAME_SD
                                result = 0
                        endif
                else
                        if in_caller_app.securityDomain.
                            securityDomainChildren->exists(sd | sd.
                            applications->exists(app | app=
                            in_called_app)) then
                                ---@REQ: ACCESS_OK_SD_CONTAIN
                                result = 0
                        else
                                ---@REQ: ACCESS_ERROR_DIFF_SD
                                self.error = ERRNO::DIFF_SD and
                                result = -1
                        endif
                endif
        endif
endif
```

Figure 3.7: *access()* operation

In Figure 3.8, the behavior of the *block()* operation is also modeled in OCL.

```
if in_app.applicationState = APPLICATION_STATE::BLOCKED
    then
          ---@REQ: BLOCK_ERROR_ALREADY_BLOCKED
          self.error = ERRNO::ALREADY_BLOCKED and
          result = -1
else
          if in_sd.applications->exists(app | app=in_app)
              then
                    ---@REQ: BLOCK_OK_SD_CONT_APP
                    in_app.applicationState =
                        APPLICATION_STATE::BLOCKED and
                    result = 0
          else
                    ---@REQ: BLOCK_ERROR_NOT_OWNER
                    self.error = ERRNO::NOT_OWNER and
                    result = -1
          endif
endif
```

Figure 3.8: *block()* operation

In Figure 3.9, the new *access()* operation behavior contains the *AIM* tag to identify a behavior that is not present in requirements.

```
if in_app1.applicationState = APPLICATION_STATE::BLOCKED then
        ---@REQ: ACCESS_ERROR_CALLER_BLOCKED
        self.error=ERRNO::CALLER_BLOCKED and
        result = -1
else
        if in_app2.applicationState = APPLICATION_STATE::BLOCKED
            then
                ---@REQ: ACCESS_ERROR_CALLED_BLOCKED
                self.error = ERRNO::CALLED_BLOCKED and
                result = -1
        else
                if in_app1 = in_app2 then
                        ---@REQ: ACCESS_ERROR_SAME_APP
                        self.error = ERRNO::SAME_APP and
                        result = -1
                else
                        if in_sd.accessRights->exists(ar | ar.
                            applications->includes(in_app1) and ar.
                            applications->includes(in_app2)) then
                                ---@REQ: ACCESS_OK_RIGHT
                                result = 0
                        else
                                ---@AIM: ACCESS_REFUSED
                                self.error = ERRNO::ACCESS_REFUSED
                                    and
                                result = -1
                        endif
                endif
        endif
endif
```

Figure 3.9: New *access()* operation

In Figure 3.10, the new *block()* operation behavior contains the functional change introduced by the new version of the GlobalPlatform specification.

```
if in_app.applicationState = APPLICATION_STATE::BLOCKED then
          ---@REQ: BLOCK_ERROR_ALREADY_BLOCKED
          self.error = ERRNO::ALREADY_BLOCKED and
          result = -1
else
          if in_sd.applications->exists(app | app=in_app) then
                    ---@REQ: BLOCK_OK_SD_CONT_APP
                    in_app.applicationState = APPLICATION_STATE::
                        BLOCKED and
                    result = 0
          else
                    if in_sd.securityDomainChildren->exists (sd |sd.
                        applications->exists(app | app=in_app)) then
                              ---@REQ: BLOCK_OK_SD_CONT_SD
                              -- Functional change
                              in_app.applicationState = APPLICATION_STATE
                                   ::BLOCKED and
                              result = 0
                    else
                              ---@REQ: BLOCK_ERROR_NOT_OWNER
                              self.error = ERRNO::NOT_OWNER and
                              result = -1
                    endif
          endif
endif
```

Figure 3.10: New *block()* operation

3.5 Tests

Based on this example and on the UML model produced, we used the Test Designer tool (see Figure 3.11) to generate several test campaigns.

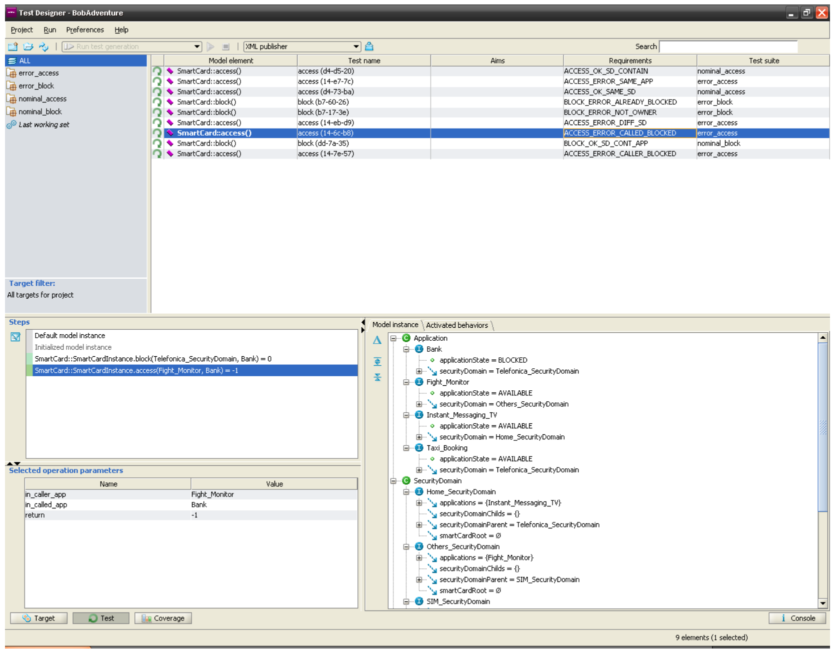
Test suite	Tests	Behaviors	Requirements
All	9	9	9
error_access	4	4	4
error_block	2	2	2
nominal_access	2	2	2
nominal_block	1	1	1

Table 3.3: Statistics on test suites

For each operation, we created two test suites: one for all nominal cases and one for all error cases. Table 3.3 presents statistics about those test suites:

- The column Test suite gives goal of test case,

- The column Tests gives the number of tests,
- The column Behaviors gives the number of Behaviors in model covering goal,
- The column Requirements gives the number of requirements associated to the goal.



Model element	Test name	Aims	Requirements	Test suite
SmartCard::access()	access (d4-d5-20)		ACCESS_OK_SD_CONTAIN	nominal_access
SmartCard::access()	access (14-e7-7c)		ACCESS_ERROR_SAME_APP	error_access
SmartCard::access()	access (d4-73-ba)		ACCESS_OK_SAME_SD	nominal_access
SmartCard::block()	block (b7-60-26)		BLOCK_ERROR_ALREADY_BLOCKED	error_block
SmartCard::block()	block (b7-17-3e)		BLOCK_ERROR_NOT_OWNER	error_block
SmartCard::access()	access (14-eb-d9)		ACCESS_ERROR_DIFF_SD	error_access
SmartCard::access()	access (14-6c-b8)		ACCESS_ERROR_CALLED_BLOCKED	error_access
SmartCard::block()	block (dd-7a-35)		BLOCK_OK_SD_CONT_APP	nominal_block
SmartCard::access()	access (14-7e-57)		ACCESS_ERROR_CALLER_BLOCKED	error_access

Steps

```
Default model instance
Initialized model instance
SmartCard::SmartCardInstance.block(Telefonica_SecurityDomain, Bank) = 0
SmartCard::SmartCardInstance.access(Fight_Monitor, Bank) = -1
```

Selected operation parameters

Name	Value
in_caller_app	Fight_Monitor
in_called_app	Bank
return	-1

Model instance | Activated behaviors

```
Application
  Bank
    applicationState = BLOCKED
    securityDomain = Telefonica_SecurityDomain
  Fight_Monitor
    applicationState = AVAILABLE
    securityDomain = Others_SecurityDomain
  Instant_Messaging_TV
    applicationState = AVAILABLE
    securityDomain = Home_SecurityDomain
  Taxi_Booking
    applicationState = AVAILABLE
    securityDomain = Telefonica_SecurityDomain
SecurityDomain
  Home_SecurityDomain
    applications = {Instant_Messaging_TV}
    securityDomainChilds = {}
    securityDomainParent = Telefonica_SecurityDomain
    smartCardRoot = Ø
  Others_SecurityDomain
    applications = {Fight_Monitor}
    securityDomainChilds = {}
    securityDomainParent = SIM_SecurityDomain
    smartCardRoot = Ø
  SIM_SecurityDomain
```

9 elements (1 selected)

Figure 3.11: Test Designer tool

The same work was done on the evolved model producing the results presented in Table 3.4.

Test suite	Tests	Behaviors	Requirements
All	9	9	8
error_access	3	3	3
error_block	2	2	2
nominal_access	2	2	1
nominal_block	1	1	1

Table 3.4: Statistics on test suites (new model)

Requirement changes (as shown in Table 3.2, page 20) explain the difference between those tables. For the *access()* operation in the nominal case, we add a new requirement and remove two requirements, which leaves us with only one requirement. A specific behavior, not part of the initial requirements, is tagged on the OCL code (see Figure 3.9) with *AIM*, and it produces a test. This explains why we have nine tests for nine behaviors but for only eight requirements. Considering error case, we removed a requirement and its OCL code. We now have only three tests left.

This example is good to illustrate method but in Section 7, we will present preliminary works of our Work Package on case studies of SecureChange project provided by Work Package 1.

We use this example in the next chapters. We will begin with some definition and vocabulary for testing evolution.

4. Several Kinds of Evolution for Test

Usually, the first step towards the development of a software system consists in collecting as a document an informal description of what the system is supposed to do. This constitutes what we call the *requirements document*. It is the description of the expected functionalities of the system. The document can contain, or be accompanied by a set of behavioral and/or security properties descriptions. Then the implementation is written, and has to be tested in order to validate the fact that it correctly implements all of these requirements. This results in functionality testing, behavioral properties testing and security testing. We examine in this chapter a list of needs for evolution that may have an impact on the MBT methodology, and we discuss how these needs can, or cannot, be addressed in the scope of an MBT approach.

4.1 Evolution of the requirements

We consider in this section two kinds of requirements: the *functional* requirements, that we distinguish from the *behavioral* requirements.

We call functional the requirements concerned with the expected functionalities of the system. They are modelled by the operations of the model. Tests of functional requirements will be obtained by structural coverage criteria of the model's operations.

Besides, the behavioral requirements are specifications of the expected behavior of entire executions of the system, regardless of the functionalities exercised. They are modelled as properties (invariant, safety, ...) that accompany the model. The tests of such behavioral requirements will be obtained by using the properties as selection criteria, i.e. by selecting executions of the model whose shape conform to the one described by the properties.

4.1.1 Impact of a functional requirement evolution

A change in functional requirements will have a direct impact on the model. Each functionality is supposed to be modelled as an operation in the model.

Thus we can report an evolution of the functional requirements as a modification of the model's operations.

Addition of a functionality The introduction of a new functionality will be reported as the definition of a new operation in the model. New tests have to be extracted to structurally cover the new operations.

Suppression of a functionality If a functionality is no longer expected to be provided by the system, then the tests that had been computed from them become totally obsolete. They can nevertheless be used as negative test cases, to validate the fact that the functionality is no longer available.

Modification of a functionality A modification of a functionality occurs when an existing feature must still be provided by the system, but differently. More precisely, the before/after specification of the functionality changes. Tests computed from the previous version of the model become "partially" obsolete. This means that the oracle for these tests has to be recomputed from the new model.

4.1.2 Impact of a behavioral requirement evolution

A change in the behavioral requirements will be reflected as a change in the properties that accompany the model. There again, changes can occur in terms of addition, suppression or modification of the properties. New tests have to be computed by using the new properties as selection criteria. The tests issued from the removed properties become obsolete. As for the modified properties, the tests issued from the property before its modification become obsolete, while new tests have to be computed to exercise the property in its modified shape.

4.2 Evolution of the model

A change in the model can occur, even in absence of a change in requirements. We do not discuss here if this is desirable or not. For example, a wrong interpretation of the informal requirements can be discovered in the model, which motivates its correction. A more radical change is to completely rewrite the model, for example in a different modelling language.

The impact of a correction of the model is the same as if the modification was imposed by a change of requirements. New tests have to be computed to exercise the added parts, and the tests that covered the modified parts become partially or totally obsolete.

As for a complete change of the model, all the former tests become obsolete, and must be replaced by new ones extracted from the new model. Notice that we could imagine relating the two models through a formal conformance relationship, from which we could deduce that the old tests are still valid w.r.t.

the new model. But we think that this question is out of scope of the Secure Change project.

4.3 Evolution of the IUT

Changes in the IUT that we consider in this section are evolutions, not revolutions. We typically think of new versions of an implementation. We distinguish it from a complete re-implementation of the system, which is obviously more a revolution than an evolution. Such radical changes in the IUT will be considered in the next section (see Section 4.4).

A modification of the system's implementation is not supposed to have a direct impact on tests, that are computed from the model and not from the IUT. If the modification was motivated by an evolution of requirements, then the model is also supposed to have evolved, according to what is discussed at Section 4.1 on the changes in the requirements. In this case, the tests have already been modified. But the modification does not necessarily result from an evolution of requirements. It can result for example from the correction of a bug. In this case, the tests are unchanged.

Thus a modification of the IUT does not have a direct impact on tests. But it has an impact on the concretization layer, which relates the tests issued from the model to the IUT. Each transformation of the IUT has to be reported into the concretization layer.

4.4 Evolution of the environment

Changes of the environment in which the system is deployed are also to be considered.

4.4.1 Technological Evolution

A complete rewriting of the IUT, for example in another programming language, or by using a new technology, can be considered as a change of the system's environment. It is in fact a change of its technological environment. Such changes are out of scope of the testing activity in the project. Once again, tests are computed from a model, that is a conceptual one. This means that technological considerations and detailed implementation choices are voluntarily not taken into account in the model. That guarantees that the model is generic enough to be used to compute tests whatever the implementation details are. Thus, these tests will remain the same, but they will be translated differently by the concretization layer, that has to be re-written in such a situation.

4.4.2 Data Evolution

The data on which the system operates is abstracted into the model. The mapping between the concrete data (manipulated by the IUT) and the abstract data (the representation in the model) is implemented by the concretization layer. Therefore there are two cases to consider if the concrete data evolve.

Abstract Data not Impacted The change does not necessarily impact the abstract data. Think for example of an error code that is abstracted as OK (no error) or KO (error) in the model. If in the concrete code the value of an error code has changed from -1 to -2, it can still for example be related to the KO value of the model. In this case, the abstract model does not change and the abstract tests remain valid. But the change has to be taken into account by the concretization layer.

Abstract Data Impacted A more substantial change of the concrete data may have an impact on the way it is modelled, and thus be reported as a modification of the model. To take such changes into account, it is necessary to identify the operations of the model that manipulate the modified data. The tests invoking these operations become partially obsolete. Their oracle has to be recomputed according to the new abstract values of the data.

4.4.3 New Vulnerabilities

In the case of tests dedicated to security, another change to be considered is the discovery of new vulnerabilities in the system, or new attacks. This raises the need for new tests, dedicated to exercise the system w.r.t. these vulnerabilities or attacks. The possibility to compute such tests by MBT depends on the ability to model vulnerabilities or attacks, in the shape of properties or by modifications of the model. The addition of new properties or the modification of the model falls into the case described at Section 4.1, of changes in requirements. This will be illustrated it in Section 6.4.

In this chapter, we have presented a list of needs for evolution that may have an impact on the MBT methodology. In the next chapter, we will present the impact for life cycle of tests and test suites.

5. Tests and Test Sequences Life Cycles

This part deals with the definition of the notion of test life cycle and test suite life cycles.

5.1 Preliminary Definitions

We give in this first section formal definitions of the test generation process that we consider.

Definition 1 (Test Generation Process) *A test generation process TG is a deterministic function, taking as input a model and a test intention, and resulting in a set of test sequences:*

$$TG : \mathcal{M} \times scen \rightarrow 2^{seq}$$

in which:

- *$\mathcal{M}$ is a test model, that describes how the system evolves,*
- *scen defines the test generation strategy, it is expressed as a test scenario that is applied to the model so as to produce a set of test sequences.*
- *seq is a test sequence, namely a sequence of steps. Each step is defined by:*

  $$\vec{o}, s \leftarrow op(\vec{i})$$

  an operation call with specific inputs $op(\vec{i})$, and its expected results, the output parameter values $\vec{o}$ and the resulting state s.

We assume that there is only one test generation process. In our approach, the test generation approach consists in animating the model, guided by the scenario, so as to produce the test sequences.

Definition 2 (Test Scenario) *A test scenario is defined as a specification of operation sequences potentially reaching specific states.*

```
SEQ     ::=  OP1 | "(" SEQ ")"                 OP      ::=  operation_name
         |   SEQ "." SEQ                                |   "$OP"
         |   SEQ REPEAT ALL_or_ONE                      |   "$OP \ {" OPLIST "}"
         |   SEQ CHOICE SEQ
         |   SEQ "⇝(" SP ")"                    OPLIST  ::=  operation_name
                                                        |   operation_name"," OPLIST
REPEAT  ::=  "?" | "{" n "}" | "{," n "}"
         |   "{" n "," m "}"                    SP      ::=  state_predicate
```

Figure 5.1: Syntax of the sequence and model layers

It is a textual description whose syntax, organised in three layers, is given by the grammar given in Figure 5.1 and Figure 5.2, and described hereafter.

The *sequence* layer (Figure 5.1, left column) is based on regular expressions that make it possible to define test scenarios as operation sequences (repeated or alternated) that may possibly lead to specific states. The *model* layer (Figure 5.1, right column) describes the operation calls at the model level and constitutes the interface between the model and the scenario.

Rule SEQ describes a sequence of operation calls as a regular expression. A step in the sequence is either a simple operation call, denoted by OP1, or an operation call that leads to a state satisfying a state predicate, denoted by SEQ ⇝(SP). This latter represents provides a useful help for the design of the scenarion since it makes possible to define the target of an operation sequence, without necessarily having to enumerate all the operations that compose this sequence. Scenarios can be composed of the concatenation of two sequences, the repetition of a sequence, or the choice between two or more sequences. In practice, we use bounded repetition operators: 0 or 1, exactly n times, at most m times, between n and m times. Rule SP describes a state predicate, whereas OP is used to describe the operation calls that can be (i) an operation name, (ii) the $OP keyword, meaning "any operation", or (iii) $OP\{OPLIST} meaning "any operation except those of OPLIST".

The test generation directive layer makes it possible to drive the step of test generation, when the tests are unfolded. We propose three kinds of directives that aim at reducing the research for the instantiation of a test scenario. This part of the language is given in Figure 5.2.

Rule CHOICE introduces two operators denoted $|$ and $\otimes$, for covering the branches of a choice. For example, if S_1 and S_2 are two sequences, $S_1 | S_2$ specifies that the test generator has to produce tests that will cover S_1 and

```
CHOICE      ::=  "|"                 OP1      ::=  OP | "["OP"]"
             |   "⊕"                          |   "[" OP "/w" CPTLIST "]"
                                              |   "[" OP "/e" CPTLIST "]"
ALL_or_ONE  ::=  "_one"
             |   ϵ                   CPTLIST  ::=  cpt_label  ("," cpt_label)*
```

Figure 5.2: Syntax of the test generation directive layer

other tests that will cover schema S_2, whereas $\mathtt{S}_1 \otimes \mathtt{S}_2$ specifies that the test generator has to produce test cases covering either S_1 or S_2. Rule ALL_or_ONE makes it possible to specify if all the solutions of the iteration will be returned (ϵ – default option) or if only one will be selected (_one).

Rule OP1 indicates to the test generator that it has to cover one of the behaviors of the OP operation. The test engineer may also require all the behaviors to be covered by surrounding the operation with brackets. Two variants make it possible to select the behaviors that will be applied, by specifying which behaviors are authorized (/w) or refused (/e) using labels that are associated to the behaviors of the model operations.

Example 1 (Scenario and Test Generation Directives) *Consider the following piece of scenario, that can be associated to the case study presented in Chapter 3.*

$$[\mathtt{Smartcard::block}\ /w\ \{\mathtt{BLOCK\_OK\_SD\_CONT\_APP}\}]\{1,3\}\_one \\ \leadsto (\mathtt{Bank.applicationState\ =\ BLOCKED})$$

This scenario tries to repeat from 1 to 3 times the invocation of the `block` *operation with a nominal behavior until the* `Bank` *application instance is in a blocked state. In practice, any sequence of length 1, 2 or 3, that ends with the blocking of the banking application will satisfy this scenario. However, the test generation directive (_one) will only keep the first one that succeeds (i.e. the shortest sequence).*

The scenarios are currently designed manually. They address a specific test intention that originates from the know-how of the validation engineer that designs the scenario. Each test intention is generally associated to a given requirement.

Definition 3 (Test Intention) *A test intention is a user-defined function intent of profile:*

$$intent : req \rightarrow scen$$

taking as input a given requirement req and producing a test scenario scen that describes how to exercise the considered requirement.

Finally, we define a test case, using the previous definitions.

Definition 4 (Test Case) *A test case is a quadruplet $\langle \mathcal{M}, req, intent, seq \rangle$. It results of the application of the intention intent for requirement req on model $\mathcal{M}$ with $seq \in TG(\mathcal{M}, req(intent))$.*

Test cases are gathered into test suites, that are defined as sets of test cases.

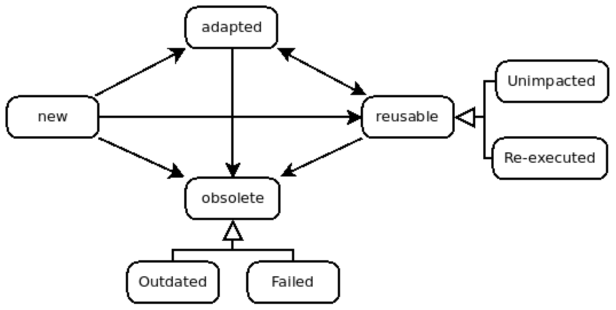


Figure 5.3: Test Life Cycles

5.2 Tests life cycle

In the context of evolving systems and, thus, evolving test suites, we associate to a test a *status*, that indicates its state in the life cycle depicted in Figure 5.3.

All tests start by being **new**. When an evolution occurs, the state of the test changes depending on the impact of the evolutions on the model elements covered by the test. If none of the covered model elements are impacted, the test may be replayed as it is, without modifying the test sequence. The test is thus said to be **reusable**, more precisely, it is *unimpacted.* If any covered model element turns out to be impacted, there are two possibilities:

- The test intention is still relevant: the test can be replayed to check if it is executable on the new model. If so, the test is said to be *re-executed.* Otherwise, it then has to be adapted so as to update the test sequence w.r.t. the evolution (i.e. update the oracle and/or modify the steps composing the test sequence). Once changed, the test becomes **adapted**. Its old version is *obsolete* and more precisely, it is *failed.*

- The test intention is no more relevant, meaning that covered model elements have disappeared or the intention does not make any sense regarding the result of the evolution. In this case, the test becomes **obsolete**, more precisely, it is *outdated.*

We assume that a high level analysis process is able to retrieve the differences between the models, and identify the modified model elements between models $\mathcal{M}^N$ and $\mathcal{M}^{N+1}$. Furthermore, we assume that we are able to detect if a test sequence *seq* covers parts of $\mathcal{M}^N$ that have been modified, or have disappeared, in $\mathcal{M}^{N+1}$.

A technical solution, based on model content analysis is provided in the next chapter.

The definition of the evolving test cases revisits the definition of the test case, and introduces the test status.

Definition 5 (Evolving Test Cases) *An* Evolving Test Case *is characterized by a triplet* $\langle req, intent, vd \rangle$ *in which*

$$vd : \mathbb{N} \rightarrow \mathcal{M} \times seq \times status$$

gives the version dependant informations of the test, namely the model from which it is computed, the test sequence and the associated status ($status \in \{new, adapted, reusable, obsolete\}$*)*

Notice that, if tc is an evolving test case, then tc^N is the test case that contains all the informations related to the version N of the system.

5.3 Test suites life cycle

We describe in this part the evolution of the test suites contents w.r.t. the evolutions that are performed. We consider three tests suites.

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

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

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

We now describe how the test suites are filled w.r.t. the evolutions and the test life cycles. This description takes into account the test status defined in Sect. 5.2. Each test suite contains a set of tests for a given version of the system. We speak about tc be an evolving test case with tc^N for previous version (when it exist) and tc^{N+1} for new version (evolving).

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

$$status(tc^{N+1}) = new \text{ and } tc^{N+1} \in \Gamma_E^{N+1}$$

Rule 2 (Reusable tests) *A reusable test 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 reusable tests are put in the Regression Test Suite:*

$$status(tc^{N+1}) = reusable \text{ and } tc^{N+1} \in \Gamma_R^{N+1}$$

Rule 3 (Adapted tests) *An adapted tests comes from an existing test suite $tc^N \in \Gamma_E^N \cup \Gamma_R^N$. All tests that have been adapted are put in the Evolution Test Suite. Their previous versions are put in the Stagnation Test Suite.*

$$status(tc^{N+1}) = adapted \text{ and } tc^{N+1} \in \Gamma_E^{N+1} \ \wedge \ tc^N \in \Gamma_S^{N+1}$$

Rule 4 (Obsolete tests) *An obsolete tests comes from an existing test suite (eventually obsolete) $tc^N \in \Gamma_E^N \cup \Gamma_R^N \cup \Gamma_S^N$. All tests that have been declared as obsolete are put in the Stagnation Test Suite.*

$$status(tc^{N+1}) = obsolete \text{ and } tc^{N+1} \in \Gamma_S^{N+1}$$

This chapter described the life cycle of tests and test suites. In the next chapter, we will present process to compute each kind of tests.

6. Methods and techniques

In Section 2.1, we have presented the Model-based Testing technique (MBT) as one possible way to test formally modeled systems. We propose in this chapter two complementary techniques of MBT. The first one focuses on evolution of the SUT and the second one on security.

6.1 Introduction

This technique as mentioned above is implemented using Rational Software Architect (RSA) based on the Test Designer (TD) Smartesting tool. For our method we have used these tools and the Unified Modeling Language (UML), in particular class and state machine diagrams. Functional requirements are expressed with the constraint language OCL. Bearing in mind the systems evolution and in order to respond to the project's aims, we have developed our method for testing evolving critical systems.

The method, we have created, is taking into account functional tests and requirements expressed in the model. On one hand, we should verify that all tested unchanged parts of the system are not affected by the modification. This subset of tests is used for regression testing. On the other hand, we should verify modifications, if they are made in implementation. We suppose that each modification in the application is passed in the model and vice-versa. In this case we have always two models:

- the first one is the reference model - the one before the evolution.
- the second one is the evolved model - including modification.

Thus, regression testing (see Deliverable 7.1) consists in re-executing well chosen test sequences in order to validate a change. In order to cover all application's behaviors, it might be necessary to generate new test sequences.

With reference to regression testing techniques we have based our technique on selective test generation strategy for UML/OCL using behavioral coverage and dependence analysis.

So, in this chapter, you will find the computation algorithms for data and control dependencies from state chart diagrams and the study carried out on selective test generation technique. In the second approach, we take into account the security properties by Telling TestStories.

6.2 Transformation into dependency graph

The selective test generation guides us to select tests from the model before the change according to different techniques. This work is inspired from the dependency analysis, as defined by Chen and al. [UPC07]. They gave rules for Extended Finite State Machines, to select tests from test suites using dependency analysis.

Notice that an UML state chart diagram can be considered as an automaton. We consider only a simple state machine without any hierarchy. So we can build our dependence graph from it. First of all, we are going to present the data dependence graph and then the control dependence graph construction. Finally, we will merge them in order to obtain the dependency graph.

6.2.1 Data dependence graph

The data dependence is based on the definition and use of variables in the graph respecting the property of def-clear path. You can find below a brief recall of the data dependence definition, which we deduced from the data dependence definition for automata (see Definition 6).

Definition 6 (Data dependent) *We define that one transition `T'` is data dependent from another transition `T` with reference to a variable `v` if and only if `v` is defined in `T` and is used in `T'` and def-clear path exists with reference to `v` between `T'` and `T`. A def-clear path is between the definition and the use of the variable and there should not be any other definition between, otherwise the path is not def-clear.*

Referring to the definition of data dependence, we have created our algorithm for state chart diagrams.

Used vocabulary for the algorithm:
`StatechartDiagram` - the state chart diagram.
`T`, `T'` - transitions belonging to the state chart diagram.
`dataDependence(X,Y)` - Boolean matrix, indicating *true* if Y is data dependent on X, and *false* if it is not.

```
Data dependence algorithm
Forall transition T ∈ StatechartDiagram do
  Forall transition T' ∈ StatechartDiagram where T' ≠ T do
    if the path between T and T' is def-clear then
        dataDependence(T,T') = true
    else dataDependence(T,T') = false
    endif
  Done
Done
```

Using this algorithm, we are defining the def/use pairs, which represent the data dependence graph.

6.2.2 Control dependencies graph

To define control dependencies, we are going to make a short recall of its definition for state chart diagrams, which we deduced from the automata control dependence definition (cf. Definition 7).

Definition 7 (Control Dependence) *Transition T has control dependence on transition T' if and only if:*

- *The state from which T' leaves does not post-dominate the state from which T leaves.*
- *The state from which T' leaves post-dominates the transition T.*

From this definition, we can find two notions that we should clarify: the post-dominance between states (see Definition 8) and the post-dominance between state and transition (see Definition 9).

Definition 8 (Post-Dominate State) *One state Z post-dominates another state Y of the state chart diagram if and only if the state Z is on each path from Y to the exit state.*

Definition 9 (Post-Dominate Transition) *One state Z post-dominates one transition T of the state chart diagram if and only if each path passing through T to the exit state passes through the state Z.*

In the next section, we are going to present each algorithm separately.

A reflexive transition (transition that is leaving and arriving at the same state) during graph visiting can create viscous circles (a viscous circle is a sequence of causes and effects in a loop). That is why we are going to consider the principle of graph coloring. We are going to mark each transition that we pass through, and if needed we are going to backtrack and mark it as a non visited transition.

Control dependence algorithm

We are proposing a simple structure for the control dependence graph. We are presenting the graph as a Boolean matrix representing whether one transition has control dependence on another transition.

`Used vocabulary for the algorithm`
`StatechartDiagram` - the state chart diagram.
`T`, `L` - transitions from the state chart diagram.
`Et` - state from the state chart diagram from which the transition T is leaving.
`El` - state from the state chart diagram from which the transition L is leaving.

`postDominanceStateState` - Boolean matrix representing whether one state post-dominates another state.
`postDominanceStateTransition` - Boolean matrix representing whether one state post-dominates one transition.
`controlDependence` - Boolean matrix representing whether one transition has control dependence on another transition.

```
Forall transition T ∈ StatechartDiagram do
  Forall transition L ∈ StatechartDiagram where L ≠ T do
    State Et = departure(T)
    State El = departure(L)
    if postDominanceStateState(El,Et) = false and
      postDominanceStateTransition(El,T)=true then
        controlDependence(T,L) = true
    else
        controlDependence(T,L) = false
    endif
  Done
Done
```

Post-Dominance State/State algorithm

Used vocabulary for the algorithm
`StatechartDiagram` - the state chart diagram.
`E1`, `E2` - states from the state chart diagram.

```
Forall state E1 ∈ StatechartDiagram do
  Forall state E2 ∈ StatechartDiagram where E1 ≠ E2 do
    Mark E2 and it's neighbours as visited
    postDominanceStateState(E1,E2) = pd(E1,E2, emptyList,
emptyList)
  Done
Done
```

The pd(S,V,neighbourV,visitedStates) operation is an auxiliary operation allowing us to visit all paths from V to the exit state and to determine whether all paths from V to the exit state pass through S by using the list of closest neighbours.

Function pd(S,V,neighbourV,visitedStates)

Used vocabulary for the algorithm
`StatechartDiagram` - the state chart diagram.
`S` - the post-dominant state.
`V`- the post-dominated state.
`neighbourV` - the list of immediate V state's neighbours.
`visitedStates` - the list of visited states during the search.

`res` - Boolean variable, returned as result at the end of the search.

We consider four trivial operations which we are not going to describe further:

- `nbStatesNonVisited(X)` - operation returning us as result the number of non-visited neighbour states for one state X of the state chart diagram.

- `neighbourNonVisited(X)` - operation returning the first non-visited neighbour state of the state X.

- `add(X,Y)` - operation adding the pair X,Y i.e. that the neighbour state Y of the state X has been visited.

- `add(X)` - operation adding the state X to the list of visited states.

```
res = true
if V = S then
    return res
else
    if neighbourV.nbStatesNonVisited(V) = 0 and
        V ∉ visitedStates then
        return false
    else
        if V ∈ visitedStates then
            return true
        else
            While neighbourV.nbStatesNonVisited(V) > 0
                and res = true do
                 State B = neighbourNonVisited(V)
                 neighbourV.add(V,B)
                 res = res and
                 pd(S,B,neighbourV,visitedStates)
            Done
            visitedState.add(V)
            return res
        endif
    endif
```

Post-Dominance State/Transition algorithm

Used vocabulary for the algorithm
`StatechartDiagram` - the state chart diagram .
`T` - transition from the state chart diagram.

`E` - state from the state chart diagram.

```
forall state E $\in$ StatechartDiagram do
  forall transition T $\in$ StatechartDiagram do
    if arriving(T) $=$ E and departure(T) $\neq$ E then
      postDominanceStateTransition(E,T) $=$ true
    else
      if postDominanceStateState(E,arriving(T)) $=$ true then
        postDominanceStateTransition(E,T) $=$ true
      else          postDominanceStateTransition(E,T) $=$ false
      endif
    endif
  Done
Done
```

We are going to create the control dependence graph using the post-dependence algorithms. Figure 6.1 presents an example of dependence graph. Furthermore, you can notice that the full line represents the data and the dotted line the control dependencies. Thus, to create the dependence graph we are going to gather together the data and the control dependence graphs.

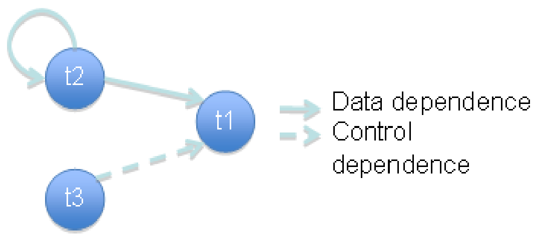


Figure 6.1: Dependence graph.

6.3 Selective test sequence generation method - SeTGaM

The test generation method created by *Smartesting*, presented in the previous paper (see Deliverable 7.1) is used for evolvable systems. This method is systematic, but we lose important information concerning tests. After we have regenerated all the tests we are not able to say how tests are classifyed w.r.t. the test sequence life cycle described in Section 5.2. That is why we propose to improve this approach by more finely classifying test sequences issued from the test suite.

As we can see in Figure 6.2, we are proposing to use state chart diagrams and dependence analysis [KHV06] on UML4ST diagrams. The approach is always based on the reference test suite and two models: the reference one and the evolved one. Then, by comparing models and using the dependence analysis we are going to be able classify a test as Unimpacted, Re-executed, Adapted, Failed, Outdated and New (see Section 5.2). Thus, we are going to keep the trace of what evolved and what did not. You can also notice in the figure that the test generation process for the reference test suite already exists. For that, we are using the *Test Designer* tool. The other part of the test generation process is proper to the SecureChange project. For now, it is done manually, but it is in our perspectives to make a tool implementing this method.

This work represents the first step of the test suite management. However, the goal is to maximize the number of tests to be reused and minimize the generation of new tests, because the generation cost can seem prohibitive for big systems. We are aiming at getting as much confidence as possible in the changed model, making sure that the evolved parts have not been affected the unchanged ones. Using our approach and rules based on dependency analysis (see Section 6.3.1), the new test suite will be created step by step. Rules are going to establish the link with models and test sequences to be selected. They are defined for each elementary modification. When we speak about elementary modification we consider addition, deletion and modification of a transition in the state chart diagram. We consider update in the OCL code inside the transition as transition's modification. Furthermore, we can have *complex modification* composed of several elementary modifications.
In the next section, we are going to detail our rules issued from the dependence analysis with reference to both models and the test suite.

6.3.1 Rules for the selective test generation with reference to model's evolution

Test sequences selection is based on state chart diagrams (the reference and the evolved one), their dependence graphs and the initial test suite. As we noted previously, we are considering three types of elementary modifications (add, modify, delete) for which we consider separately the created rules. The composition of these actions is what we call *complex modification* and the analysis will

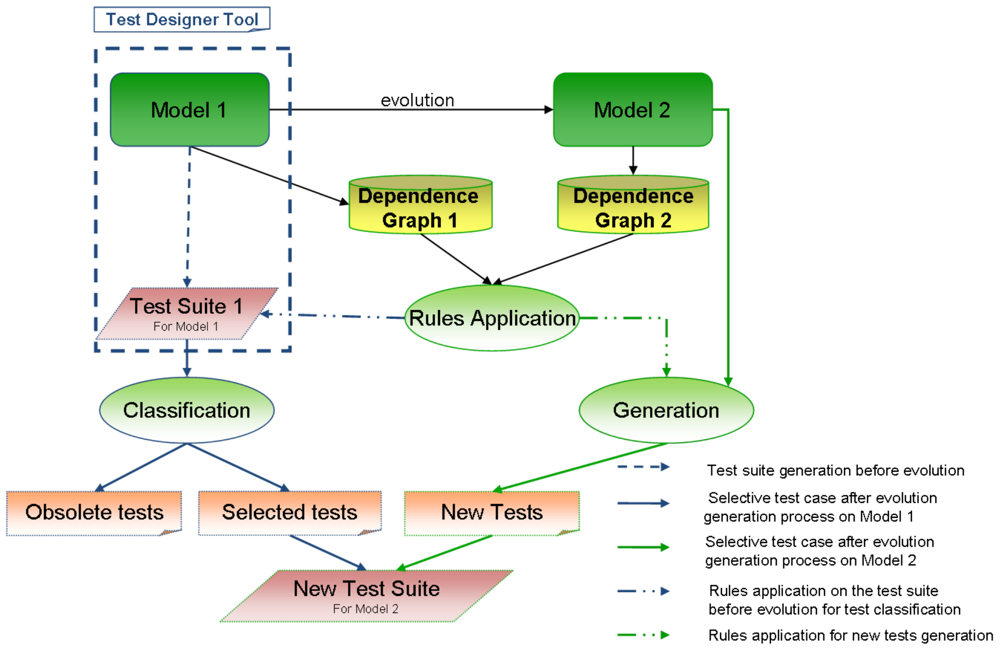

Test Designer Tool
Model 1
evolution
Model 2
Dependence Graph 1
Dependence Graph 2
Test Suite 1
For Model 1
Rules Application
Classification
Generation
Obsolete tests
Selected tests
New Tests
New Test Suite
For Model 2
Test suite generation before evolution
Selective test case after evolution generation process on Model 1
Selective test case after evolution generation process on Model 2
Rules application on the test suite before evolution for test classification
Rules application for new tests generation

Figure 6.2: Selective test generation.

be done separately for each component. Each modification has its own rules presented in Section 5.2, so we are able to add new tests, as well as to delete or to modify existing ones.

Thus, applying this technique will allow us to classify tests and compute the new test suite as well as to verify that the evolved parts did not affect the unchanged parts, that unchanged part did not affect the evolution and that what is changed has been changed in the implementation.

Used vocabulary:

- **NewTransition** - new transitions added into the evolved state chart diagram.
- **DelTransition** - deleted transition from the reference state chart diagram.
- **ModifTransition** - modified transition in the evolved state chart diagram w.r.t. the reference one. Modification can be the modification of a variable's value or a new variable into an OCL code.
- **TS** - the test suite before evolution.
- **NTS** - the new test suite, after evolution.
- **t'** - transition that concerns an elementary modification.
- **t,t1,t2** - transitions linked to the test suite TS.
- **T** - test sequence from the test suite, covering the behavior of the transition *t*.
- **T'**- test sequence covering the behavior of the transition *t'*.
- **T1** - test sequence covering *t* in case of evolution.
- **T1'** - test sequence covering *t'* in case of evolution.
- **T2** - test sequence from the test suite, covering the behavior of the transition *t2*.
- **T2'** - test sequence covering *t2* in case of evolution passing through *t1*.
- **cd(x,y)** - function returning true if the transition "y" is control dependent from the transition "x", and false otherwise.
- **dd(x,y)** - function returning true if the transition "y" is data dependent from the transition "x", and false otherwise.

In case of an elementary modification in the dependence graph according to Chen and al. [UPC07] we can have creation and/or deletion of data and/or control dependencies. But, it is also possible that there are dependencies that will not change. However, we are going to use the vocabulary of *creation* and *elimination* of a transition. With respect to the results that they have obtained, we have created our own rules for state chart diagrams. We use the notation $\cup_=$ with the following meaning: $X \cup_= Y$ means $X = X \cup Y$.

1. **New transition** $\forall t' \in$ NewTransition
   (a) Calculate the test sequence T', in order to cover the new transition t':

   $$\text{New} \cup_= \text{T'}$$

   (b) $\forall T \in$ TS
      i. if creation of cd(t',t) then
      The test sequence T is classified as outdated, we should calculate a new test sequence T1:

      $$\text{Outdated} \cup_= \text{T and}$$
      $$\text{New} \cup_= \text{T1}$$

      ii. if creation of dd(t',t) then
      Re-run the test sequence T covering the behavior of t:

      $$\text{Re-testable} \cup_= \text{T}$$

      iii. It is possible to have any creation or elimination of other cd(t1,t2) or an elimination of other dd(t1,t2).
      if creation or elimination of other cd(t1,t2) or dd(t1,t2) then
      The test sequence T2 becomes outdated, we should then calculate a new test sequence T2':

      $$\text{Outdated} \cup_= \text{T2 and}$$
      $$\text{New} \cup_= \text{T2'}$$

2. **Deleted transition** $\forall t' \in$ DelTransition and
   (a) $\forall$ T' $\in$ TS covering the behavior of t' becomes outdated.

   $$\text{Outdated} \cup_= \text{T'}$$

   (b) $\forall T \in$ TS
      i. if elimination of cd(t',t) then
      The test T covering the behavior of the transition t is no more relevant, it becomes outdated and we should calculate a new test sequence *T1*:

      $$\text{Outdated} \cup_= \text{T and}$$
      $$\text{New} \cup_= \text{T1}$$

      ii. else if elimination of dd(t',t) then
      The transition t could be data dependent of the transition *t'*, that is why we should rerun the sequence test T covering the behavior of t:

Re-testable $\cup_=$ T

iii. taking into account modifications it is possible to have other creation or elimination of data/control dependencies for other transitions.

- if creation or elimination of other cd(t1,t2) then

  Outdated $\cup_=$ T2 and
  New $\cup_=$ T2'

- if creation of dd(t1,t2) then

  Re-testable $\cup_=$ T2

- if elimination of other dd(t1,t2) then

  Outdated $\cup_=$ T2 and
  New $\cup_=$ T2'

3. **Modified transition**

   When we speak about modification of transition, we cannot have any modification in the control dependence graph, because it cannot be changed in any case. So, we consider in this case only changes in the data dependence graph. $\forall t' \in$ ModifTransition and $\forall T \in$ TS

   (a) forall dd(t',t)

   i. if creation of dd(t',t) then
   Add the test sequence T covering the behavior of t into *Re-testable*:

   Re-testable $\cup_=$ T

   ii. if elimination of dd(t',t) then

   Outdated $\cup_=$ T and
   New $\cup_=$ T1

   (b) forall dd(t,t')

   i. if creation of dd(t,t') then
   Add the test sequence *T'* covering the behavior of *t'* into *Re-testable*:

   Re-testable $\cup_=$ T'

   ii. if elimination of dd(t,t') then

   Outdated $\cup_=$ T' and
   New $\cup_=$ T1'

   (c) forall dd(t1,t2)

   i. if creation of dd(t1,t2) then
   Add the test sequence *T2* covering the behavior of *t2* into *Re-testable*:

   Re-testable $\cup_=$ T2

ii. if elimination of dd(t1,t2) then

$$\text{Outdated} \cup_{=} \text{T2 and}$$
$$\text{New} \cup_{=} \text{T2'}$$

All other tests which are not concerned by these rules are considered as *Unimpacted*. We can conclude about the final test suite's elements only after the *Re-testable* tests execution. Figure 6.3 shows clearly the two steps to respect in this approach with respect to the life cycle of test sequences:

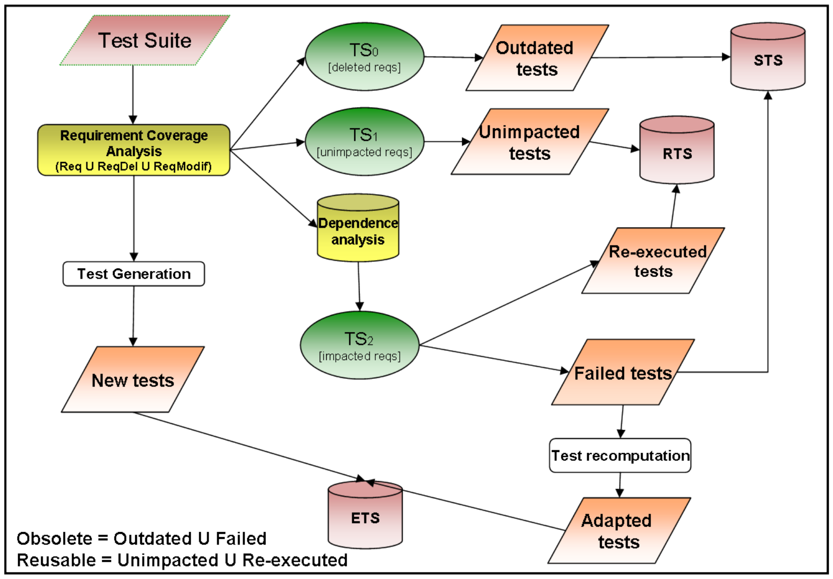


Figure 6.3: Test sequence's life cycle.

By running the set of tests *Re-testable* we are creating the sets *Re-executed*, *Failed* and *Adapted*. Thus, we are completing test suites for regression, evolution and stagnation testing (see Section 5.3). Bearing in mind the covering strategy on the state chart diagram as well as the dependence results, when tests are put in the stagnation test suite, we may need to generate new test sequences. So, we can consider two kinds of obsolete tests:

- outdated tests as result of the rules implementation,
- failed tests as result of the Re-testable test sequence set run.

Thus the stagnation test suite (STS):

$$\text{STS} = \text{Outdated} + \text{Failed}$$

We can consider two types of tests sequences that are to be considered in the regression test suite or as reusable:

- unimpacted tests, directly selected by the rules from the test suite before evolution - *TS*,
- re-executed tests as result of the Re-testable test sequence set run.

So, we can consider the following rule that tests from the regression tests suite (RTS) are those which are not Outdated nor Failed and those which are part of the Re-executed set:

$$\text{RTS} = \text{Unimpacted} + \text{Re-executed}$$

We can also conclude about the evolution test suite, that we have:

- new tests as result of the rules application,
- adapted tests as result of the need to compute new tests in order to cover completely the behavior of transitions in the state chart diagram.

Thus:

$$\text{ETS} = \text{Adapted} + \text{New}$$

Finally, the new test suite after the evolution results from the set of tests to be reused, the adapted and the new generated ones:

$$\text{NTS} = \text{Unimpacted} + \text{Re-executed} + \text{Adapted} + \text{New}$$
moreover
$$\text{NTS} = \text{RTS} + \text{ETS}$$

6.3.2 Synthesis

The approach that we have proposed is using the behavioral elementary environment. The specification's modeling was done using *RSA* and the test sequences were previously generated by *TD*. We have proposed to use a new method for each elementary modification called *Selective test sequence generation method.*

This study allows us to define rules by using the same vocabulary (cf. Section 6.3.1) for added (cf. Table 6.1), deleted (cf. Table 6.2) and modified (cf. Table 6.3) transitions. As you can see rules are defined for each elimination or creation of data or control dependence.

Table 6.1 gathers all rules for an added transition (see Section 6.3.1) in a state chart diagram.

In Table 6.2, you can find a short recall of the rules applied when a transition is deleted from a state chart diagram.

As we have noticed previously the case of transition's modification includes modifications in the OCL code of one operation, which simplifies rules for this case. In Table 6.3, we recall the rules with reference to the test sequences' life cycle.

With this method we are validating the system step by step, and it allows us to save time and memory resources. Using the selective test sequences generation, we have introduced a new testing methodology that we have called evolution testing.

Dependence	Outdated	Re-testable	New
$\forall$t' $\in$ New-Transition			$\cup_=$ T'
cd(t',t) - **create**	$\cup_=$ T		$\cup_=$ T1
dd(t',t) - **create**		$\cup_=$ T	
cd(t1,t2) - **create**/ **elimin** or dd(t1,t2) - **elimin**	$\cup_=$ T2		$\cup_=$ T2'

Table 6.1: Rules for added transition.

Dependence	Outdated	Re-testable	New
$\forall$ t' $\in$ Del-Transition	$\cup_=$ T'		
cd(t',t) - **elimin**	$\cup_=$ T		$\cup_=$ T1
dd(t',t) - **elimin**	$\cup_=$ T		$\cup_=$ T1
cd(t1,t2) - **create**/ **elimin** or dd(t1,t2) - **elimin**	$\cup_=$ T2		$\cup_=$ T2'
dd(t1,t2) - **create**		$\cup_=$ T2	

Table 6.2: Rules for deleted transition.

Dependence	Outdated	Re-testable	New
dd(t',t) - **create**		$\cup_=$ T	
dd(t',t) - **elimin**	$\cup_=$ T		$\cup_=$ T1
dd(t,t') - **create**		$\cup_=$ T'	
dd(t,t') - **elimin**	$\cup_=$ T'		$\cup_=$ T1'
dd(t1,t2) - **create**		$\cup_=$ T2	
dd(t1,t2) - **elimin**	$\cup_=$ T2		$\cup_=$ T2'

Table 6.3: Rules for modified transition.

6.3.3 Bob's adventure test case

Initial behavior

In order to apply our method on Bob's example we have created a state chart UML diagram (see Figure 6.4). In this state chart we have created elementary transitions, i.e. one transition per requirement (see Table 3.1). By applying the data dependence algorithm we can extract the following data dependencies:

- The transition *access_ called_ block* is data dependent from the transition *block* about the variable *applicationState*. This can be noted as:

  DD(block, access_ called_ block, applicationState)

- The transition *access_ caller_ block* is data dependent from the transition block about the variable *applicationState*. This can be noted as:

  DD(block, access_ caller_ block, applicationState)

- The transition *block_ already_ blocked* is data dependent from the transition block about the variable *applicationState*. This can be noted as:

  DD(block, block_ already_ blocked, applicationState)

Using the control dependence algorithm we have obtained that the transition *end_ error* is control dependent on *access_ called_ blocked*, *access_ caller_ blocked*, *access_ same_ sd_ same_ app*, *access_ diff_ sd_ caller_ sd_ not_ contain_ called*, *block_ sd_ not_ owner* and *block_ already_ block*:

- *CD(access_ called_ blocked, end_ error)*

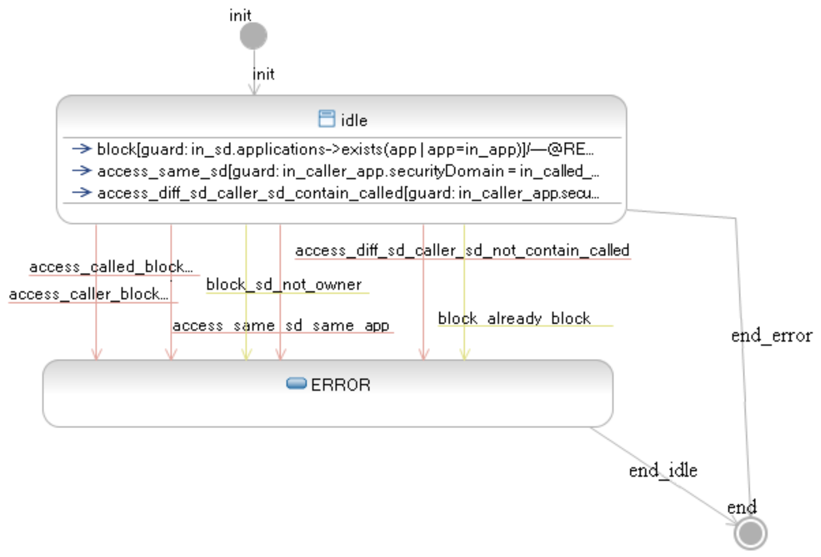
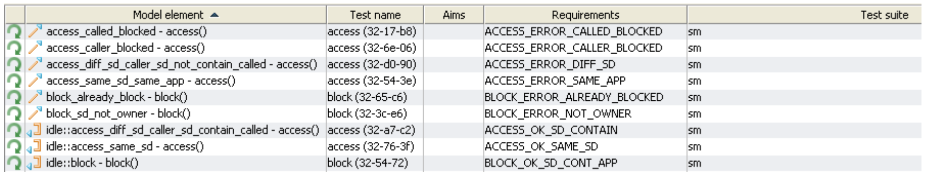

init
init
idle
block[guard: in_sd.applications->exists(app | app=in_app)]/—@RE...
access_same_sd[guard: in_caller_app.securityDomain = in_called_...
access_diff_sd_caller_sd_contain_called[guard: in_caller_app.secu...
access_called_block...
access_caller_block...
block_sd_not_owner
access_diff_sd_caller_sd_not_contain_called
access_same_sd_same_app
block_already_block
end_error
ERROR
end_idle
end

Figure 6.4: State chart of the smart card's initial behavior

- *CD(access_ caller_ blocked, end_ error)*
- *CD(access_ same_ sd_ same_ app, end_ error)*
- *CD(access_ diff_ sd_ caller_ sd_ not_ contain_ called, end_ error)*
- *CD(block_ sd_ not_ owner, end_ error)*
- *CD(block_ already_ block, end_ error)*

Using the TestDesigner tool we have created the reference test suite. In Figure 6.5, you can map requirements and transitions represented by their behavior. This test suite will be next used as reference for the test selection method.

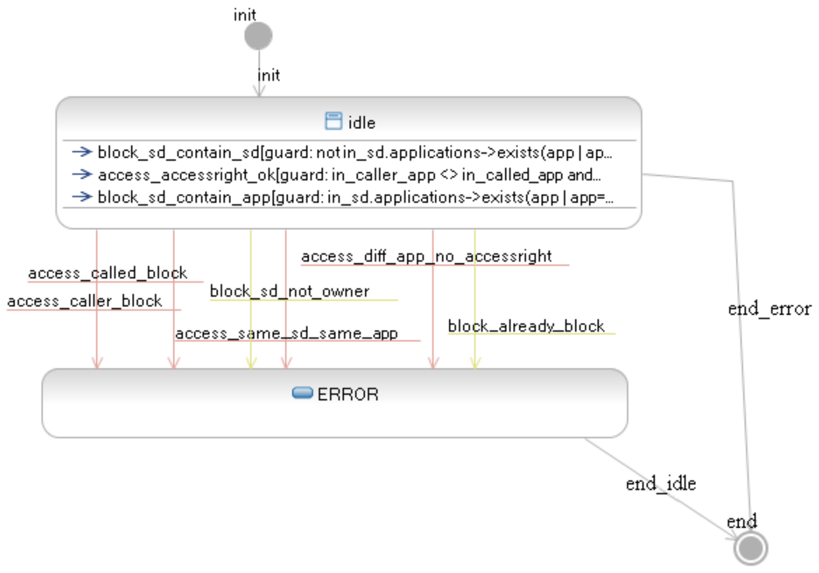


Model element ▲	Test name	Aims	Requirements	Test suite
access_called_blocked - access()	access (32-17-b8)		ACCESS_ERROR_CALLED_BLOCKED	sm
access_caller_blocked - access()	access (32-6e-06)		ACCESS_ERROR_CALLER_BLOCKED	sm
access_diff_sd_caller_sd_not_contain_called - access()	access (32-d0-90)		ACCESS_ERROR_DIFF_SD	sm
access_same_sd_same_app - access()	access (32-54-3e)		ACCESS_ERROR_SAME_APP	sm
block_already_block - block()	block (32-65-c6)		BLOCK_ERROR_ALREADY_BLOCKED	sm
block_sd_not_owner - block()	block (32-3c-e6)		BLOCK_ERROR_NOT_OWNER	sm
idle::access_diff_sd_caller_sd_contain_called - access()	access (32-a7-c2)		ACCESS_OK_SD_CONTAIN	sm
idle::access_same_sd - access()	access (32-76-3f)		ACCESS_OK_SAME_SD	sm
idle::block - block()	block (32-54-72)		BLOCK_OK_SD_CONT_APP	sm

Figure 6.5: Initial test suite

Evolution

As you can see in Table 3.2, the Bob's model evolution implied some changes into the UML model. We can clearly separate new requirements, deleted requirements as well as requirements that changed the model dynamic level (the OCL code). In Figure 6.6, we have represented the state chart diagram of this evolution. There you can distinguish the different transitions and observe their behavior.



init
init
idle
block_sd_contain_sd[guard: not in_sd.applications->exists(app | ap...
access_accessright_ok[guard: in_caller_app <> in_called_app and...
block_sd_contain_app[guard: in_sd.applications->exists(app | app=...
access_called_block
access_caller_block
block_sd_not_owner
access_same_sd_same_app
access_diff_app_no_accessright
block_already_block
end_error
ERROR
end_idle
end

Figure 6.6: State chart of Bob's adventure evolution

With reference to the Bob's UML model presented previously we have the following new requirements, and so new transitions:

- *access_ accessright_ ok*
- *block_ sd_ contain_ sd*
- *access_ diff_ app_ no_ accessright*

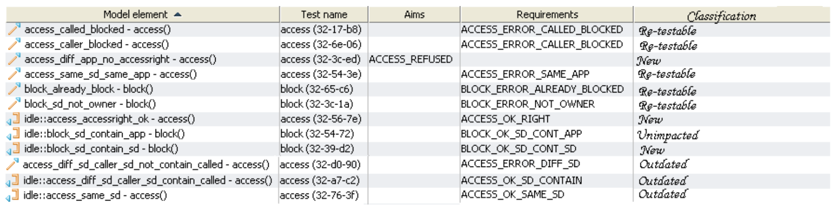
As we have introduced new requirements, some changes were introduced in some transitions at dynamic level, and some transitions, as well as the requirements, disappeared completely.
In this case we have only one transition that has been modified at dynamic level, which is *block_ sd_ not_ owner*. We have also three deleted transitions for this small example that are *access_ same_ sd*, *access_ diff_ sd_ caller_ sd_ contain_ called* and *access_ diff_ sd_ caller_ sd_ not_ contain_ called*.
When the model changes, then data and control dependence diagrams can change, too. Thus, we can see transitions which are affected by the change. Bearing in mind the evolution, we have obtained the following data dependencies *(the data dependencies in italic are the new ones)*:

- DD(block_sd_contain_app, access_called_block, applicationState)

- DD(block_sd_contain_app, access_caller_block, applicationState)
- DD(block_sd_contain_app, block_already_blocked, applicationState)
- *DD(block_sd_contain_sd, access_called_block, applicationState)*
- *DD(block_sd_contain_sd, access_caller_block, applicationState)*
- *DD(block_sd_contain_sd, block_already_blocked, applicationState)*

Furthermore, we have some changes in the control dependence diagram. On one hand one control dependence has been removed - *CD(access_diff_sd_caller_sd_not_contain_called, end_error)*. On the other hand there is a new one - *CD(access_diff_app_no_accessright, end_error)*.

At this stage, we have all the elements required to apply our rules in case of detected evolution (see Section 6.3.1). We have used rules for adding, deleting and modifying transition. Finally, the result is presented in Figure 6.7 and you can see the test cases classification from the test suite as New, Unimpacted, Re-testable or Outdated.



Model element	Test name	Aims	Requirements	Classification
access_called_blocked - access()	access (32-17-b8)		ACCESS_ERROR_CALLED_BLOCKED	Re-testable
access_caller_blocked - access()	access (32-6e-06)		ACCESS_ERROR_CALLER_BLOCKED	Re-testable
access_diff_app_no_accessright - access()	access (32-3c-ed)	ACCESS_REFUSED		New
access_same_sd_same_app - access()	access (32-54-3e)		ACCESS_ERROR_SAME_APP	Re-testable
block_already_block - block()	block (32-65-c6)		BLOCK_ERROR_ALREADY_BLOCKED	Re-testable
block_sd_not_owner - block()	block (32-3c-1a)		BLOCK_ERROR_NOT_OWNER	Re-testable
idle::access_accessright_ok - access()	access (32-56-7e)		ACCESS_OK_RIGHT	New
idle::block_sd_contain_app - block()	block (32-54-72)		BLOCK_OK_SD_CONT_APP	Unimpacted
idle::block_sd_contain_sd - block()	block (32-39-d2)		BLOCK_OK_SD_CONT_SD	New
access_diff_sd_caller_sd_not_contain_called - access()	access (32-d0-90)		ACCESS_ERROR_DIFF_SD	Outdated
idle::access_diff_sd_caller_sd_contain_called - access()	access (32-a7-c2)		ACCESS_OK_SD_CONTAIN	Outdated
idle::access_same_sd - access()	access (32-76-3f)		ACCESS_OK_SAME_SD	Outdated

Figure 6.7: Test case classification using the selective test case generation method

Finally the tests *block_already_block* and *block_not_owner* are to be reused, while the other tests have failed. The *access()* operation has changed the number of parameters, so for these transitions we have adapted the old test case.

To conclude we can say that with the new method we have minimized test generation time. Instead of recomputing all the test cases in order to cover all requirements, we have generated only three test cases. Three other tests were put into the Outdated set. After the execution of the Re-testable set two test cases were classified as Unimpacted, because there was no need to change them and all others needed to be adapted. The next future work will be to optimize the test suite by test sequence refactoring.

6.4 Telling TestStories

Telling TestStories (TTS) provides a methodology and a tool implementation for model–driven system testing of service oriented systems. In this section, we give an overview of the underlying artifacts and the methodology. A more detailed overview is given in [FBCO+09]. An application of this framework is shown in Section 7.1.

6.4.1 Security Requirements

IT systems store, process and share information. To protect this information from unauthorized modification, harm or disclosure, different techniques of information security are used. Information security is concerned with different security goals of protecting information, e.g. confidentiality, integrity and availability.

Security is a quality factor of software systems and can be decomposed into a hierarchy of underlying quality subfactors [Fir04]. These quality subfactors are subject for testing to assure proper compliance. This means that if the quality requirements on the subfactors are met, a system is considered *secure* with respect to the tested factors. Nevertheless, it should be noted that the absence of problems can not be demonstrated by testing, only their presence [Dij69].

Several classifications of security requirements can be found in the literature, e.g. [Fir03]. Here we focus on one of the most prominent lists of requirements, but further subclassification is still possible. In the following we list the security requirements and give their definitions according to [NST06]:

- *Confidentiality*: Assurance that information is not disclosed to unauthorized individuals, processes, or devices.
- *Integrity*: Condition existing when data is unchanged from its source and has not been accidentally or maliciously modified, altered, or destroyed.
- *Authentication*: Security measure designed to establish the validity of a transmission, message, or originator, or a means of verifying an individual's authorization to receive specific categories of information.
- *Authorization*: Access privileges granted to a user, program, or process.
- *Availability*: Timely, reliable access to data and information services for authorized users.
- *Non-repudiation*: Assurance that none of the partners taking part in a transaction can later deny of having participated.

Testing security requirements is different from testing functional requirements. Security testing is mostly about negative requirements, so called *abuse cases*, where an attacker tries to do something he is not permitted to do. Traditional software testing instead deals with the detection of software failures [PM04].

Defects or deviating behavior is normally detected by checking specific properties of a system during execution. These checks are also called *assertions*. Security requirements are an adequate source for the definition of such assertions.

However, the same security requirement can be violated in different parts of the system. Consider the following security requirement restricting access to a home automation environment: *"Unregistered users are not allowed to access the domotics system."*. For testing whether this authorization requirement is not violated it has to be checked at every point where a service of the domotics system is accessed, e.g. illumination, air conditioning etc. Resulting from the nature of negative requirements, a one-to-one mapping from the security requirement to code artifacts is not easily found in most cases [MR05]. Nevertheless, to know which parts of the system could be affected by deviating (security) behavior such information is neccessary. This motivates the need for traceability from assertions to security requirements and possible consequences.

6.4.2 System and Testing Artifacts

Figure 6.8 shows the artifacts of the TTS framework. Informal artifacts are depicted by clouds, formal models by graphs, code by transparent blocks and running systems by filled blocks.

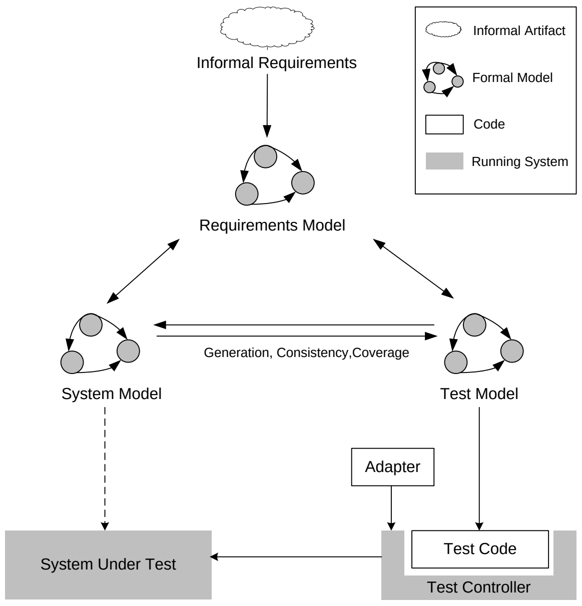


Figure 6.8: Artifacts overview

In the following paragraphs we explain the formal models and the code fragments of Figure 6.8 in more details. The **Informal Requirements**, i.e. written or non–written capabilities and properties of the system, and the **System** providing and requiring services callable by the test controller, are not discussed in details because they are not the main focus of our testing methodology.

Requirements Model. The requirements model describes requirements for system development and testing in a formal way. It consists of actors, use cases, domain types, and requirements hierarchies denoted in use case diagrams, class diagrams, and requirements diagrams. The formal requirements are based on written or non–written informal requirements depicted as cloud.

System Model. The system model describes the system structure and system behavior in a platform independent way. Its static structure is based on the notions of services, components and types. Each service operation call is assigned to use cases, actors correspond to components providing and requiring services, and domain types correspond to types. We assume that each service in the system model corresponds to an executable service in the running system to guarantee traceability. Therefore the use cases, the service operations and the executable services are traceable.

Test Model. The test model defines the test configuration, the test data and the test scenarios as so called test stories. *Test stories* are controlled sequences of service operation invocations exemplifying the actors' interaction. Test stories may be generic in the sense that they do not contain concrete objects but variables which refer to test objects provided in tables. Test stories can also contain setup resp. tear down procedures and contain assertions for test result evaluation. The notion of a test story is principally independent of its representation [FCOB09]. We have used UML activity diagrams in previous papers [FZF+09] and use sequence diagrams in the present work.

If the system model and the test model are created manually, it has to be guaranteed that they are consistent with each other and that the test model fulfills some coverage criteria with respect to the system model (see [FBCO+09] for consistency and coverage examples). Alternatively, if the system model is complete then behavioral parts of the test model can be generated, or otherwise if the test model is complete, behavioral fragments of the system model can be generated.

Each test story is linked to a use case and can be considered as part of the requirements. The approach is suitable for test–driven modeling resp. development because the test stories can be defined before the behavioral artifacts of the system model or the system implementation are available. Test–driven development is possible because from test models and adapters it is possible to derive executable tests even before the implementation has been finished. Test–driven modeling can be applied because the test model can be defined before the behavioral system models whose design can be supported by checking consistency and coverage between the system and the test model [FBCO+09]. In a system–driven development approach behavioral artifacts can be used to derive test models and test data.

Test Code. The test code is generated by a model–to–text transformation from the test model as explained in [FFZ+09]. It generates test code that can be executed by a test controller.

Adapters. The adapters are needed to access service operations provided and required by components of the system under test. For a service implemented as web service, an adapter can be generated from its WSDL description. Adapters for each service guarantee traceability.

Testing Methodology

Figure 6.9 shows the workflow of our testing methodology. The methodology of our framework supports test–driven development of systems on the model level.

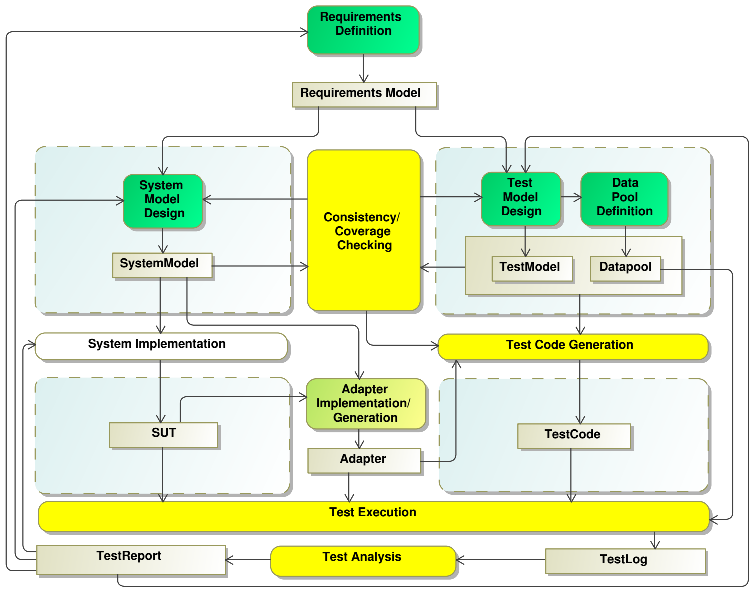


Figure 6.9: Testing methodology

The first step is the definition of requirements. Based on the requirements, the system model and the test model are designed in parallel. The test design includes the data pool definition, i.e. the definition of test data, and the test sequence definition, i.e. the sequence of test stories together with states and data to be tested. The system model and the test model, including the test stories, the data and the test sequences, can be checked for consistency and coverage. This allows for an iterative improvement of their quality and supports model–driven system and test development. The methodology does not consider the system development itself but is based on traceable services offered by the system under test. As soon as adapters which may be – depending on the technology – generated automatically or implemented manually are available for the system services, the process of test code generation can take place. The generated test code is then automatically compiled and executed by a test execution engine which logs all occurring events into a test log. The test evaluation is done offline by a test analysis tool which generates test reports and annotations to those elements of the system and test model influencing the test result.

TTS is appropriate for security testing on the system level based on security requirements. Our approach guarantees traceability between security requirements, the system and test model and the executable service oriented system. Security tests can be modeled in the same way as functional tests. The approach therefore provides information which security requirement is fulfilled and not just negative information claiming which requirement is not fulfilled.

6.5 Conclusion

We have presented in this chapter two complementary methods. The first allows for managing evolution of the system and the second for managing security. Future works will be a hybrid approach to take the better of each one to improve validation of security for evolving system.

In the next chapter, we present the first results of Work Package 7 and the application of these two frameworks on the case studies provided by Work Package 1.

7. Case Study

This chapter presents two of the three case studies provided by Work Package 1. In fact, we will only present these two case studies because Work Package 7 works only on these two ones. Each case study is presented by means of an overview, with a focus on the part that is being studied, followed by the application of our methodology (see Chapter 6) on it.

7.1 Home Gateway

With this case study, we show how *security tests* can be deduced from *security requirements* to assure the reliability of a system. While we describe the evolution of tests and test suites with the previous example, we focus on testing security requirements by this example. More precisely, we will pursue the idea of *functional security tests* here, i.e. we test the security functionalities of a system.

We provide an industrial case study to show that model–driven testing based on the methodology of Telling Test Stories (TTS) which is explained in Section 6.4 is appropriate for security testing on the system level. Based on the definition of security requirements we define a system model and test model containing test stories that are traceable to security requirements. Test stories are then transformed into executable test code and therefore make security requirements executable. We also show how tests can be executed by integrating the test component as passive participant into the process under test.

The aim of the case study[1] is to control the network access of clients in a home network depending on different client attributes. Typical attributes are for example the age of a user or a check whether the anti-malware application has been updated during the last 24 hours. An example for a resulting effect after checking the client attributes is that an underage user may only be allowed to access a restricted set of resources of the network and is only allowed to access the internet and not the private home network (e.g. if access to music collection should be denied).

The scenario consists of different peers distributed among the home network and the operator network. These peers are the *Access Requestor* (AR), *Home Gateway* (HG), *PolicyEnforcement Point* (PEP) and the *Policy Decision Point*

[1] The case study was kindly provided by Telefónica.

(PDP). Following service oriented principles [Erl05], each peer shares interfaces defining the terms of information exchange. The AR is the client application to establish and use the connection to the home network. An AR always connects to a HG. The HG is a device installed at the home of customers controlling access to different networks and services (e.g. domotics, multimedia, data services). The enforcement of who is allowed to access which resources on the network is made by an internal component of the HG called PEP. The PEP gets the policy it has to enforce for a specific AR by the PDP which is the only component run by the operator and not the end user herself. Because we have four independent components only interacting via well-defined interfaces to execute a process, the example adheres to our definition of a service oriented system. Furthermore, we are focusing on testing dedicated example sequences (i.e. the test stories) of the system and verify whether certain security requirements hold under such conditions. The components are normally already tested in an isolated way before they are deployed in a service oriented setting. The TTS framework adheres to a test-driven development approach, thus it allows the execution of test stories in early stages of system development and supports the evolution of the underlying system. Thus, we think that TTS is an ideal choice for a testing framework of the presented system.

The remaining subsections discuss the elicitation of some sample security requirements and their integration into the requirements model in a structured and traceable way. After that, we present the system model and, finally, we present the test model for testing the security requirements and discuss the technical realization for executing tests and asserting security requirements.

7.1.1 Requirements Model

We represent requirements in two different ways, by requirements hierarchies and by use cases. Both representations allow the annotation of security requirements as shown in this section.

Requirements hierarchies define a refinement from abstract requirements resp. goals to more detailed requirements. Security requirements and any other type of non–functional requirements can be integrated into this hierarchy in a natural way. For this purpose, we use SysML requirements diagrams [OMG07] and mark security requirements with the stereotype `securityRequirement`. In Figure 7.1 the security requirements are represented in a requirements hierarchy.

In this representation to each security requirement as to all other requirements, model elements of the test model (test stories, assertions, test sequence elements) verifying the requirement can be assigned. In our basic requirements hierarchy security requirements are formulated as positive statements, defining how a vulnerability can be avoided. Attached test stories may then define possible vulnerabilities that make the requirement failing. In Figure 7.1 we have defined an example for all types of security requirements listed in Section 6.4.1. Requirement 1.2.1 is an example for authentication, 1.2.2 for confidentiality, 1.3.1 for availability, 1.4.1 for authorization, 1.4.2 for integrity and 2.1 for non–repudiation.

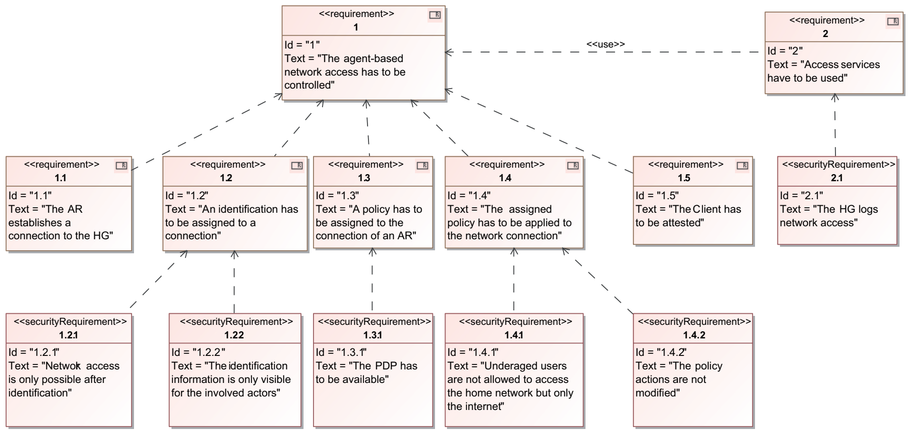

<<requirement>> 1
Id = "1"
Text = "The agent-based network access has to be controlled"
<<use>>
<<requirement>> 2
Id = "2"
Text = "Access services have to be used"
<<requirement>> 1.1
Id = "1.1"
Text = "The AR establishes a connection to the HG"
<<requirement>> 1.2
Id = "1.2"
Text = "An identification has to be assigned to a connection"
<<requirement>> 1.3
Id = "1.3"
Text = "A policy has to be assigned to the connection of an AR"
<<requirement>> 1.4
Id = "1.4"
Text = "The assigned policy has to be applied to the network connection"
<<requirement>> 1.5
Id = "1.5"
Text = "The Client has to be attested"
<<securityRequirement>> 2.1
Id = "2.1"
Text = "The HG logs network access"
<<securityRequirement>> 1.2.1
Id = "1.2.1"
Text = "Netwok access is only possible after identification"
<<securityRequirement>> 1.2.2
Id = "1.2.2"
Text = "The identification information is only visible for the involved actors"
<<securityRequirement>> 1.3.1
Id = "1.3.1"
Text = "The PDP has to be available"
<<securityRequirement>> 1.4.1
Id = "1.4.1"
Text = "Underaged users are not allowed to access the home network but only the internet"
<<securityRequirement>> 1.4.2
Id = "1.4.2"
Text = "The policy actions are not modified"

Figure 7.1: Requirements

Use case diagrams depict application scenarios and the involved actors. In our approach, use cases and actors can be mapped to components and behaviors. If a security requirement can be transformed to a functional requirement then it can be represented as use case. If a security requirement is formulated as constraint on a use case, we attach it as comment to a use case. In some cases it is also possible to transform a non–functional security constraint into a functional use case. In Figure 7.2 the security requirements of Figure 7.1 are represented as comments on use cases.

The explicit modeling of actors allows the definition of relationships between roles and access permissions of roles.

Requirements traceability refers to the ability to describe and follow the requirement's life, in both a forwards and backwards direction [GF94]. Traceability has to be guaranteed by a system testing approach to report the status of system's requirements. Our representation of requirements allows the definition of traceability by links between model elements, i.e. by assigning test stories to requirements. We have established traceability between the requirements model, the system model, the test model and the executable system, because service operation calls in test stories are linked to service operation calls in the system model which are linked to executable service operations in the system implementation.

7.1.2 System Model

As already mentioned, the AR is the client application to establish and use the connection to the home network. We model this with an `AccessRequest` interface required by the AR. This interface is provided by the HG because an AR always connects to a HG. The data used to decide to which networks a client is granted access is retrieved via the `Identification` interface which is

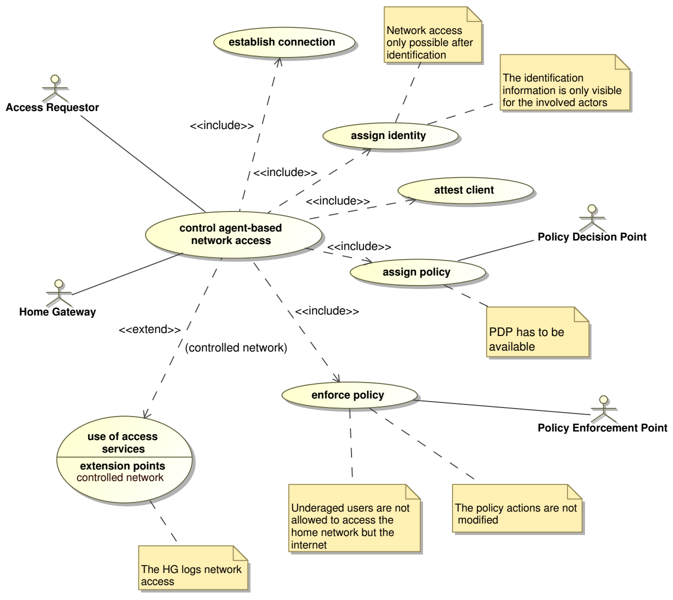


Figure 7.2: Requirements expressed on use cases

provided by the AR. The HG uses an internal component - PEP to enforce these restrictions. The PEP receives the policy it has to enforce for a specific AR by the PDP via the `PolicyDecision` interface.

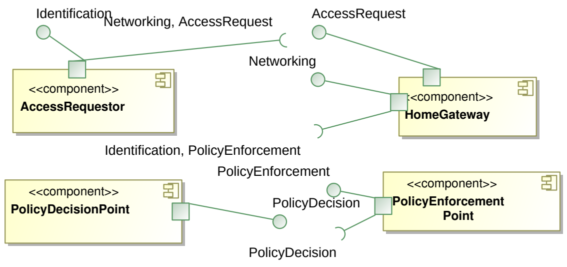


Figure 7.3: Actors modelled as components with provided and required interfaces.

All components in this scenario are connected with each other and the interfaces between them are well-defined (see Figure 7.3). A request by the AR will trigger the input of user credentials via the identification interface. The data returned by the AR is of type `IdentificationData` (see Figure 7.4). With this information the PDP is able to look up the appropriate policy for the request and send the corresponding list of `PolicyAction`s to the PEP which enforces them. `PolicyAction` and `IdentificationData` are data types defined internally in the system model. The type `IdentificationData` describes a username, a password and an optionally attestation data of an AR; the type `PolicyAction` is currently only used to describe to which VLAN the PEP should allow access by a specific AR.

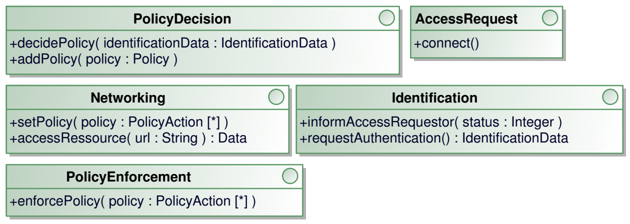

PolicyDecision
+decidePolicy(identificationData : IdentificationData)
+addPolicy(policy : Policy)
AccessRequest
+connect()
Networking
+setPolicy(policy : PolicyAction [*])
+accessRessource(url : String) : Data
Identification
+informAccessRequestor(status : Integer)
+requestAuthentication() : IdentificationData
PolicyEnforcement
+enforcePolicy(policy : PolicyAction [*])

Figure 7.4: Interface definitions of services.

The communication among the peers is based on different protocols and standards. Authentication follows IEEE 802.1X[2] which defines a supplicant, an authenticator and an authentication server. In our case the supplicant is the AR, the role of the authenticator is taken over by the HG and the authentication server (a RADIUS[3] database) is represented by the PDP. Note that the system model describes all components, their interfaces and optionally also behavioural parts of the system. For the present contribution we only show components and interface definitions as it suffices to describe the present scenario.

7.1.3 Test Model

The TTS test model contains a set of test stories whose execution order is defined in a test sequence. To make all requirements executable, we assign an assertion, a test story or a whole sequence element to it. Here we present just one complex and representative test story in Figure 7.5 and show how the requirements can be mapped to it.

The test story in Figure 7.5 defines a basic network access scenario containing two assertions. First the `AccessRequestor` connects to the `HomeGateway` (step 1 in Figure 7.5), which then requests the authentication data containing a username, a password and an assessment data from the `AccessRequestor` (steps 2 and 3). This information is forwarded to the `PolicyDecisionPoint` (step 4), which sends a sequence of policy actions to the `HomeGateway` (step 5) based on the identity information of the `AccessRequestor`. We then assert that there has to be a policy action that contains the expected VLAN (`$vlan`) to check the policy actions for integrity. The `HomeGateway` sends the policy actions to the `PolicyEnforcementPoint` (step 6), and then informs the `AccessRequestor` (step 7), which then accesses a specific URL (steps 8 and 9). Finally, we check

[2]Available at http://www.ieee802.org/1/pages/802.1x-rev.html.
[3]<u>R</u>emote <u>A</u>uthentication <u>D</u>ial <u>I</u>n <u>U</u>ser <u>S</u>ervice, as specified in RFC 2865.

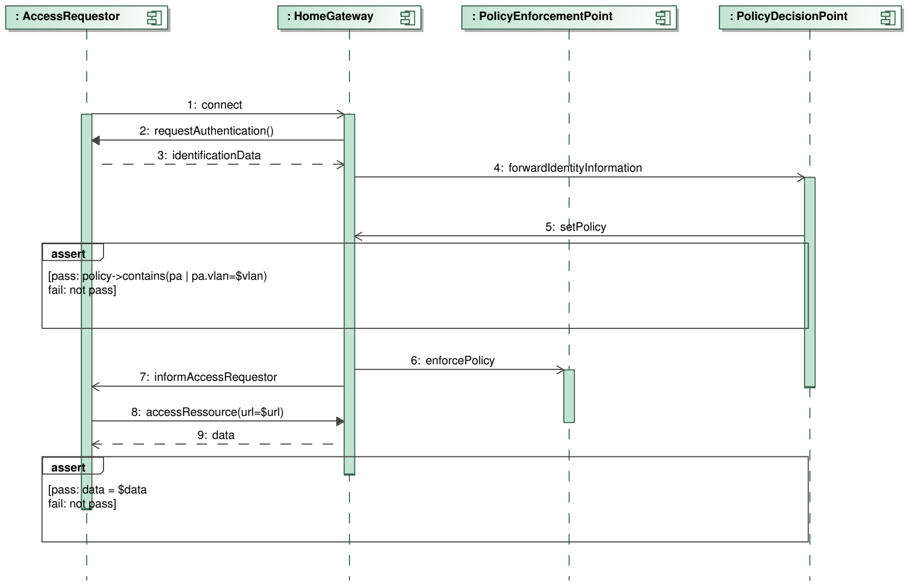


Figure 7.5: Teststory `TestPolicy`

whether the `accessRessource()` call returns the expected data. Test cases for this test scenario are defined in Table 7.1.

`#TC`	`$username`	`$password`	`$vlan`	`$url`	`$data`
1	'michael'	'0815'	'HomeNetwork'	'http://74.125.43.99'	webpage_1
2	'michael'	'0815'	'HomeNetwork'	'http://192.168.1.1'	webpage_2
3	'philipp'	'0000'	'Internet'	'http://74.125.43.99'	webpage_1
4	'philipp'	'0000'	'Internet'	'http://192.168.1.1'	null
4	'guest'	'0000'	'Internet'	'http://192.168.1.1'	null

Table 7.1: Test Data Table

The test story is completed by adding some policies to the `PolicyDecisionPoint` in an initial setup. In Figure 7.6(a) three policies are added to the `PolicyDecisionPoint`. Each policy assigns a sequence of policy actions, in our basic example just a sequence set of accessible VLANs, to a username/password combination. The identification data objects are stored on our data pool and are as follows in our example:

```
policy1:('michael','0815',(['Internet','HomeNetwork'])
policy2:('philipp','0000',(['Internet'])
policy3:('*','*',(['GuestNetwork'])
```

The test sequence element in Figure 7.6(b) defines that the setup state `initializePDP` has to be executed before the test story `TestPolicy` can be

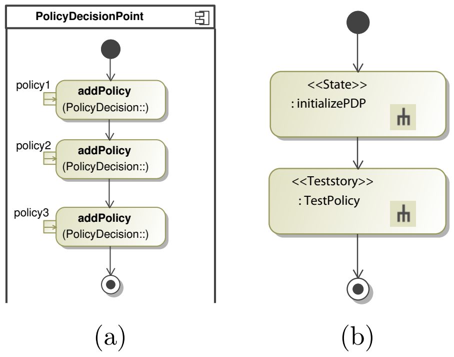


Figure 7.6: Story for system setup (a) and global test story invoking setup and another test story (b)

executed for every test case of Table 7.1. Test sequence elements can contain additional arbitrations that aggregate the verdicts of the stories' test cases, e.g. such an arbitration could be `pass%=100%`, i.e. all test cases of a test story have to pass. The two assertions in our test story are traceable to requirements. In the requirements model of Figure 7.1, the first assertion can be assigned to Requirement 1.4.2 testing integrity, and Requirement 1.4.1 testing authorization. Additionally the overall test story can be mapped to Requirement 1.4 which is done implicitly in this case because the test story covers all sub requirements. In the use case representation of Figure 7.2 the test story is mapped to the use case 'enforce policy'. Test stories, their states, test sequence elements and traceability for testing other requirements are similar to the one presented in Figure 7.5 but differ at least in the assertions.

7.1.4 Test Execution

The case study consists in four different actors communicating with each other. A crucial point of our testing strategy is that the different services are not tested individually and in an isolated way. Instead we define test stories which describe possible sequences of service invocations on the SUT. Testing each service separately is out of scope for this testing strategy because they may consist of third-party implementations (i.e. authentication server and access requestor) and have normally already run through extensive test suites. What we are interested in is the behaviour and the value of certain parameters at specific points in a test story, i.e. the *assertions*.

Another important point of the test execution technique is that the test engine is primarily a passive participant in this process. However, this is not a limitation of the Telling TestStories framework itself, see [FBCO+09]. The reason for a passive execution engine lies in the scenario itself: all actors except the AR are hard-wired to each other. For instance, when the AR sends the `EAP-Response/Identity` message (i.e. the return value of the `requestAuthentication()`-call) to the HG this will trigger a message exchange between HG and PDP. The AR is not aware of this communication as it is solely controlled by the HG. Thus, a central execution engine acting as an orchestration unit is not reasonable in this scenario because it would simply "miss" certain messages. The test execution technique for this scenario starts a test story and only interacts with the `AccessRequestor`. The parameters for this interaction are given in the data table. The remaining communication will only be observed by the execution engine. For monitoring this communication we use packet sniffers (TShark[4]) at various points in the environment. This allows us to track the full communication in a non-intrusive way, i.e. the architecture remains unchanged and no central orchestration point controlling the communication between components is to be integrated.

Before the test story is started, the system is first set to a specific state. This is to set up the environment so that the SUT can be tested. In our case the setup consists of a number of `addPolicy()`-calls to the PDP for installing the policies, see Figure 7.6(a). After the system is initialized, the execution engine triggers the `connect()`-call by the AR. The HG then requests the user credentials from the AR in the `requestAuthentication()`-call. These are provided by the execution engine by consulting the data table, i.e. `$username` and `$password`. The next step, where the AR is involved, is the `informAccessRequestor()`-call where the AR is notified about the decision by the PDP. Immediately after this notification the AR can try to access a specific network resource via the `accessRessource()`-call. Again, the parameter for the requested URL is fetched from the data table, i.e. `$url`. The rest of the communication, where the AR is not involved, is only observed.

By monitoring all messages, the execution engine is able to keep track of the current value of variables defined in the interfaces among services, e.g. which `PolicyActions` are returned by the PDP. This information and the content of the data table are sufficient to compose assertions and to check the behaviour of the system. For example, the assertion `[pass: data = $data]` in the test story checks whether accessing a specific URL is allowed/denied as specified in the policy. This assertion can be evaluated by getting the value for `data` out of the monitored return value of `accessRessource()` and the value for `$data` out of the data table.

Technically, in the present setting there are two points where information has to be sniffed, i.e. IEEE 802.1X traffic between the AR and HG, and RADIUS traffic between the HG and PDP. For each captured message of a running test story the sniffer matches it to an interaction step of the test model and assigns the values according the defined interface. If an assertion is encountered during this traversal then a verdict can be computed based on the current content of the variables. After the test execution the results can be evaluated as described in Section 6.4.2.

[4]Available at http://www.wireshark.org.

7.2 GlobalPlatform

The goal of this part is to determine the domain of our work on GlobalPlatform. The GlobalPlatform specification is wide. For now, we are interested in differences between **GP 2.1.1** and **2.2** specification versions. The SeTGaM method (see Section 6.3) can be also applied to differences between configurations of **GP**, such as **UICC**[5] for GP 2.2 specification.

In the GlobalPlatform documents you can find different life cycle models in order to control the working and the security of GlobalPlatform components :

- card
- executable load files
- executable modules
- applications

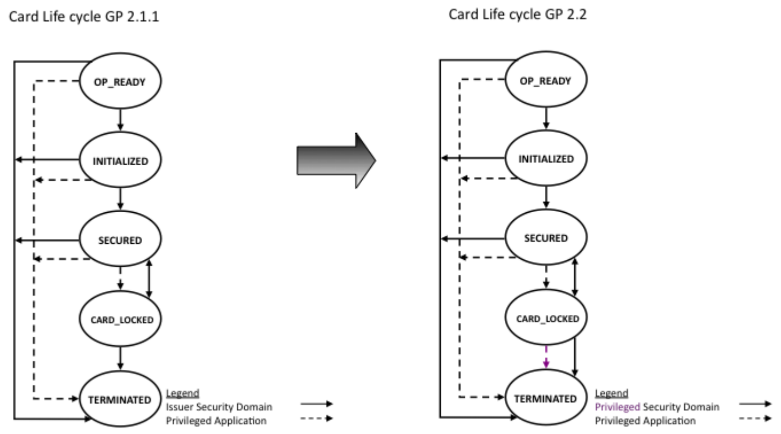


Figure 7.7: Life cycle in each version of GlobalPlatform

For our test model we have taken into account the card life cycle. The card life cycle begins with the state OP_READY. The card's life can pass through

[5]The GP UICC Configuration is a configuration of v2.2, standardising the minimum interoperability for (U)SIM cards for supporting remote application management.

different states: INITIALIZED, SECURED, CARD_LOCKED. Finally, the card life cycle ends with the state TERMINATED and then the card is no more operational. In the card life cycle we can see two important periods: the pre-issuance and the post-issuance. The OP_READY and INITALIZED states are part of the pre-issuance card's period. The SECURED, CARD_LOCKED and TERMINATED are part of the post-issuance card's period. In Figure 7.7, we have the two life cycles of GlobalPlateform gard in version 2.1.1 and version 2.2. To go from one state to another and execute different operations, the system should use APDU's commands. In our model we are going to test the following commands :

- `setStatus`: used to modify card or application life cycle state.
- `getStatus`: used to retrieve ISD, executable load file or module, application or SD life cycle state.
- `storeData` (for card life cycle state): generic command used to transfer data to a SD or application.

ISD is a specific application. In fact, the ISD's life cycle state is the card's life cycle. So the status of the card is returned by a call of `getStatus` with the option "CARD". These APDUs cannot be called in the model if some constraints are not satisfied, according to rules in GlobalPlatform specification. That is why we are using other commands to create the context that allows to call these APDUs :

- ManageChannel
- Select
- The macro openSecureSession :
  - initialize update
  - external authenticate
- PutKey [DES]
- The macro INSTALL :
  - Install [for load]
  - Load
  - Install [for install]
  - Install [for make selectable]

7.2.1 The GlobalPlatform test model

Once we have defined that the card life cycle is our scope, we have created the GlobalPlatform test model for the card life cycle. It is to be noted, that this scope will be extended to ***card content management*** once the method SeTGaM (see Section 6.3) is validated on the card life cycle.

The Class Diagram

You can find here the class diagram (see Figure 7.8) representing the GP 2.2 objects. Associations between classes illustrated the *Card* interactions with *applications, executable load files and executable modules* as well as with *logical channels*. We represent the card life cycle *state* as a card attribute.

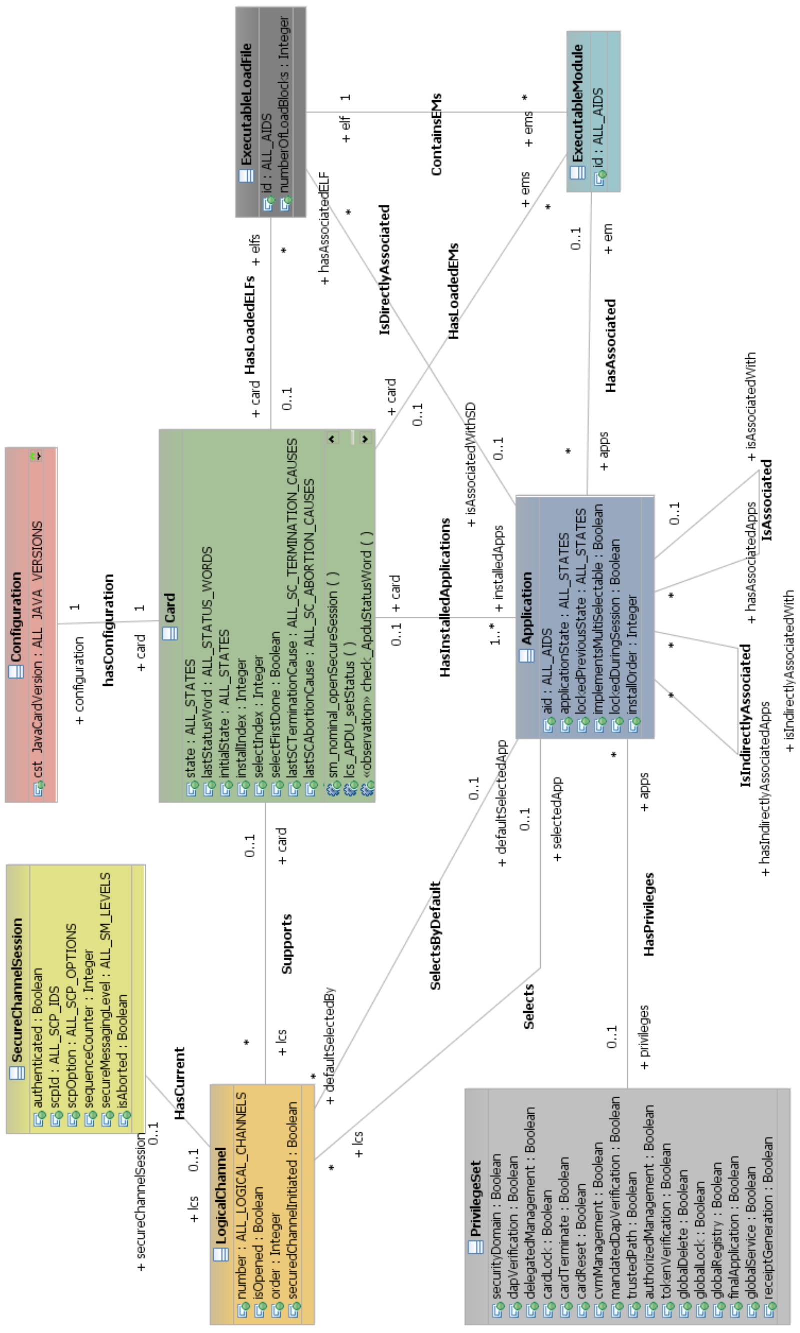


Figure 7.8: GlobalPlatform class diagram

Example

The following example is briefly representing the card state change from **INITIALIZED** to **SECURED**. Figure 7.9 illustrates that only *Privileged Security Domain* can change the card's state.

```
sm_nominal_openSecureSession(lc_00, sm_no_sm, KVN_00h)
lcs_APDU_setStatus(lc_00, CARD, INITIALIZED, aid_ISD)
lcs_APDU_setStatus(lc_00, CARD, SECURED, aid_ISD)
```

Figure 7.9: *Example*

7.2.2 Model's dynamic level

To express the model's behavior we used the Object Constraint Language (OCL) that is part of Unified Modeling Language (UML). For each command listed above we have defined its behavior using OCL. In this part, our goal is not to show how we represented each command. As in Figure 7.10, we only focused on the *setStatus()* command passing the card state to *TERMINATED*.

```
if (IN_state = ALL_STATES::TERMINATED) then
        if (l_lc.selectedApp.privileges.cardTerminate = true) then
                self.state = IN_state and
                /*...*/
                self.lastStatusWord = ALL_STATUS_WORDS::SUCCESS
                /**@REQ: APDU_SETSTATUS_SUCCESS_CARD_
                     LOCKED_TO_TERMINATED */
        else
                self.lastStatusWord  = ALL_STATUS_WORDS::ERROR_
                SETSTATUS_mustHaveTerminatePriv
                /**@AIM: FROM_LOCKED */
                /**@REQ: APDU_SETSTATUS_ERROR_CARD_
                      MUST_HAVE_TERMINATE_PRIVILEGE */
        endif
```

Figure 7.10: *setStatus()* operation

Above you can see two different behaviors. The first one is when command's success in card's state TERMINATED. The second one is an error case. These two cases can be covered by two test sequences. One covering the success state (see Figure 7.11). The second covers the error state (see Figure 7.12).

```
sm_nominal_openSecureSession(lc_00, sm_CMAC, KVN_FFh)
lcs_APDU_setStatus(lc_00, CARD, INITIALIZED, aid_ISD)
lcs_APDU_setStatus(lc_00, CARD, SECURED, aid_ISD)
lcs_APDU_setStatus(lc_00, CARD, CARD_LOCKED, aid_ISD)
lcs_APDU_setStatus(lc_00, CARD, TERMINATED, aid_ISD)
-> SUCCESS
```

Figure 7.11: Test sequence (success): APDU_SETSTATUS_ SUCCESS_CARD_-LOCKED_TO_TERMINATED

```
sm_nominal_openSecureSession(lc_00, sm_CMAC, KVN_FFh)
nominal_APDU_installAndMakeSelectable(lc_00, aid_01,
    priv_SSD18_AuthorizedManagement_FinalApp_CLock,
    aid_ISD,instance_ExecutableModuleForSd)
cm_APDU_select(lc_00, BY_NAME, FIRST_OR_ONLY_OCCURRENCE,
    aid_01, sm_no_sm, true)
sm_nominal_openSecureSession(lc_00, sm_CMAC, KVN_FFh)
lcs_APDU_setStatus(lc_00, CARD, INITIALIZED, aid_ISD)
lcs_APDU_setStatus(lc_00, CARD, SECURED, aid_ISD)
lcs_APDU_setStatus(lc_00, CARD, CARD_LOCKED, aid_ISD)
lcs_APDU_setStatus(lc_00, CARD, TERMINATED, aid_ISD)
-> ERROR
```

Figure 7.12: Test sequence (error): FROM LOCKED APDU_SETSTATUS_ERROR_-CARD_MUST_HAVE_TERMINATE_PRIVILEGE

7.2.3 Model's evolution management

We are considering one model for each **GP** version or configuration. Our goal is to compare them and then to reduce the number of tests to re-generate. For instance we present you a brief example of change from *GP 2.1.1* to *GP 2.2*. As you can see in Figure 7.13, on one hand GP 2.1.1 does not allow the transition from *CARD_LOCKED* to *TERMINATED* performed by an application that is not a **Security Domain**. On the other hand GP 2.2 allows that behavior if the application has a specific privilege, i.e. card_lock_privilege.

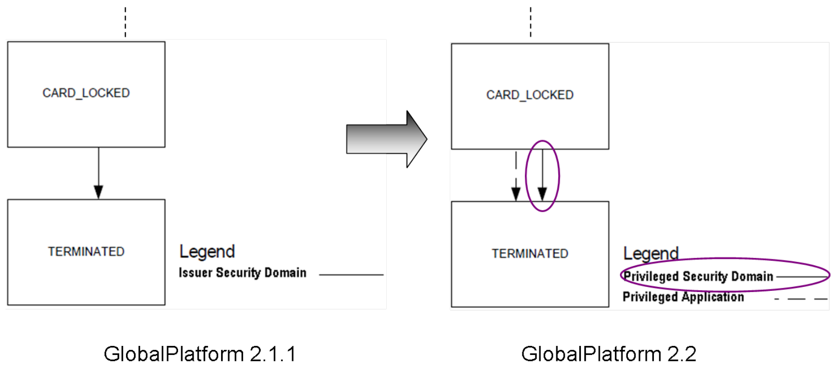


Figure 7.13: Evolution from GP 2.1.1 to GP 2.2

As previously described according to *GP 2.1.1* specification it is not possible to pass from state *CARD_LOCKED* to *TERMINATED*. In Figure 7.14, we present a test sequence proving that this behavior is taken into account in the model. As a result we have an error message *ERROR_SETSTATUS_onlyISDCanTerminateFromLocked*.

In Figure 7.15, we presented a test sequence for *GP 2.2*. For this test sequence, the behavior is activate and the test sequence results with a *SUCCESS* message.

To conclude, we have created a complete model for the card life cycle for GP 2.2. In future we are going to create GP 2.1.1 model and state chart diagrams as well in order to apply the method mentioned above.

```
sm_nominal_openSecureSession(lc_00, sm_CMAC, KVN_FFh)
                                                  ->SUCCESS
nominal_APDU_installAndMakeSelectable(lc_00, aid_01,
priv_SSD12_AuthorizedManagement_FinalApp_CLock_CTerm,
aid_ISD, instance_ExecutableModuleForSd)
                                                 -> SUCCESS
cm_APDU_select(lc_00, BY_NAME, FIRST_OR_ONLY_OCCURRENCE,
                         aid_01, sm_no_sm, true)->SUCCESS
sm_nominal_openSecureSession(lc_00, sm_CMAC, KVN_FFh)
                                                  ->SUCCESS
lcs_APDU_setStatus(lc_00, CARD, INITIALIZED, aid_ISD)
                                                  ->SUCCESS
lcs_APDU_setStatus(lc_00, CARD, SECURED, aid_ISD)
                                                  ->SUCCESS
lcs_APDU_setStatus(lc_00, CARD, CARD_LOCKED, aid_ISD)
                                                  ->SUCCESS
lcs_APDU_setStatus(lc_00, CARD, TERMINATED, aid_ISD)
     -> ERROR_SETSTATUS_onlyISDCanTerminateFromLocked
```

Figure 7.14: Test sequence (error) : ERROR_SETSTATUS _onlyISDCanTerminateFromLocked

```
sm_nominal_openSecureSession(lc_00, sm_CMAC, KVN_FFh)
                                                    ->SUCCESS
nominal_APDU_installAndMakeSelectable(lc_00, aid_01,
priv_SSD12_AuthorizedManagement_FinalApp_CLock_CTerm,
        aid_ISD, instance_ExecutableModuleForSd)->SUCCESS
cm_APDU_select(lc_00, BY_NAME, FIRST_OR_ONLY_OCCURRENCE,
                           aid_01, sm_no_sm, true)->SUCCESS
sm_nominal_openSecureSession(lc_00, sm_CMAC, KVN_FFh)
                                                    ->SUCCESS
lcs_APDU_setStatus(lc_00, CARD, INITIALIZED, aid_ISD)
                                                    ->SUCCESS
lcs_APDU_setStatus(lc_00, CARD, SECURED, aid_ISD)
                                                    ->SUCCESS
lcs_APDU_setStatus(lc_00, CARD, CARD_LOCKED, aid_ISD)
                                                    ->SUCCESS
lcs_APDU_setStatus(lc_00, CARD, TERMINATED, aid_ISD)
                                                    ->SUCCESS
```

Figure 7.15: Test sequence: SUCCESS

8. Conclusion

This deliverable proposes two points of view for evolution integration into a model-based testing approach identified in Deliverable 7.1. The first one is based on evolution and the second one is based on security.

At first, we are comparing two model versions in order to extract information about evolution. So, we can classify computed test suites from the first version into three categories (obsolete, adapted, reusable) to be used on the new one. By this classification, we reduce the time to generate test suites because we only compute new and adapted test suite. Another advantage is that we can use the obsolete test suite to define stagnation test suite. The stagnation test suite is used to validate that the SUT has correctly taken evolution into account.

Second, we create a system's test model based on security requirements. We can manage several kinds of requirements as confidentiality, integrity, authentication, authorization, availability and non repudiation. We can generate dedicated test suites with respect to the system model in order to ensure coverage.

Each approach is based on requirements management, proposed by Work Package 3 and the UML/OCL model by the Work Package 4. We begin to validate our approaches with a sub-part of case studies provided by Work Package 1 (see Chapter 7).

In the future, we will implement the proposals and algorithms to evaluate the approach with a bigger example or a part of the case studies. So, the integration into a full process from design to execution of test scripts will be done during the next year.

Bibliography

[Dij69] E.W. Dijkstra. Notes on Structured Programming. 1969.

[Erl05] T. Erl. *Service-oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2005.

[FBCO+09] M. Felderer, R. Breu, J. Chimiak-Opoka, M. Breu, and F. Schupp. Concepts for Model-based Requirements Testing of Service Oriented Systems. In *Proceedings of the IASTED International Conference*, volume 642, page 018, 2009.

[FCOB09] M. Felderer, J. Chimiak-Opoka, and R. Breu. A Standard–Aligned Approach to Model–Driven System Testing of Service Oriented Systems. submitted to SAC 2009, 2009.

[FFZ+09] M. Felderer, F. Fiedler, P. Zech, , and R. Breu. Flexible Test Code Generation for Service Oriented Systems. 2009. QSIC'2009.

[Fir03] D.G. Firesmith. Engineering Security Requirements. *Journal of Object Technology*, 2(1):53–68, 2003.

[Fir04] D.G. Firesmith. Specifying Reusable Security Requirements. *Journal of Object Technology*, 3(1):61–75, 2004.

[FZF+09] Michael Felderer, Philipp Zech, Frank Fiedler, Joanna Chimiak-Opoka, and Ruth Breu. Model-driven System Testing of a Telephony Connector with Telling Test Stories. In *Software Quality Engineering. Proceedings of the CONQUEST 2009*, pages 247–260, 2009.

[GF94] O. C. Z. Gotel and C. W. Finkelstein. An analysis of the requirements traceability problem. pages 94–101, 1994.

[IST10] ISTQB. *Manual in Download Section*, 2010. http://www.istqb.org/.

[KHV06] Bogdan Korel, Luay H.Tahat, and Boris Vaysburg. Model based regression test reduction using dependence analysis. In *IEEE ICSM '06*, 2006.

[MR05] C. Michael and W. Radosevich. Risk-based and functional security testing. Technical report, Technical report, US Department of Homeland Security and Cigital Inc, 2005.

[NST06] CNSS Instruction Formerly NSTISSI. 4009, "National Information Assurance Glossary", Committee on National Security Systems, June 2006. 4009, 2006.

[OMG07] OMG. *OMG Systems Modeling Language*, 2007. http://www.omg.org/docs/formal/2008-11-01.pdf.

[PM04] B. Potter and G. McGraw. Software Security Testing. *IEEE Security & Privacy*, pages 81–85, 2004.

[TMa08] *TMap Test Topics*. Tutein Nolthenius Nederland, 2008.

[UPC07] Hasan Ural, Robert L. Probert, and Yanping Chen. Model based regression test suite generation using dependence analysis. In *Proceedings of the third internationnal workshop on Advances in model-based testing*, pages 54–62, 2007.

[UPL06] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing. Technical Report 04/2006, Computer Science Department, The University of Waikato, April 2006. Available from http://www.cs.waikato.ac.nz/pubs/wp.