Model-based Testing of Cryptographic Components Lessons Learned from Experience

Julien Botella[‡], Fabrice Bouquet[*†], Jean-François Capuron[§], Franck Lebeau[*], Bruno Legeard[†‡], Florence Schadle[§]
[*]INRIA, Equipe Cassis – Nancy, France
[†]Université de Franche-Comté, FEMTO-ST DISC – Besançon, France
[‡]Smartesting – Besançon, France
[§]DGA Maîtrise de l'Information – Bruz, France

***Abstract*—We present an approach to use techniques of model-based testing (MBT) applied on security cryptographic components. This application of MBT is done in the context of a *qualification testing phase* made by an entity independent from designers, developers and sponsors of the cryptographic components under test. This qualification phase targets both hardware and software cryptographic components and the testing activities cover functional and security testing objectives. In this context, we present the application of MBT for two cryptographic components (one hardware and one software) and show the complementary of test selection criteria based on one side on a structural coverage of the behavioral model used for test generation, and on the other side on a test purpose approach to meet some security test objectives. The test purpose language used in this project is novel and has been designed to complete behavioral model coverage criteria in the MBT process.**

***Index Terms*—Model-Based Testing; Cryptographic components; UML4MBT; Test purpose**

I. Introduction

A. Context, Objectives and results of the study

This paper presents an experience report on deploying model-based testing (MBT) on cryptographic components. Such components (hardware or software) implement cryptographic algorithms, including symmetric ciphers or public-key ciphers. They provide encryption and signing keys, and form a set of primitives that can be used as building blocks to construct security mechanisms. In the defense industry, these components are used in a large variety of systems, for example for network security protocols in radio communication and system command and control.

An acceptance testing phase, named qualification testing phase, is done by the relevant French authority (DGA), for cryptographic components delivered by industrials and to be used by the French Army. This qualification testing phase uses several complementary techniques, including black-box and white-box testing, in order to validate the conformance of the tested component to its requirements and to verify security properties such as confidentiality, integrity and availability. Our work targets the black-box testing part, addressing the components through their external APIs, and aiming to ensure some security test objectives using a MBT approach.

In this context, MBT has been deployed to test two cryptographic components since 2010: one hardware component (based on FPGA - field-programmable gate array) and a large software cryptographic library. The goal was to evaluate, on full size components, the adequacy of MBT with respect with security testing objectives in the qualification testing phase achieved at the DGA. The security testing objectives define a set of testing needs, based on the experience of the DGA, and systematically applied to the components in the qualification phase. For this kind of (security) components, functional and security testing objectives are mixed up because the functionalities of these components are merely dedicated to security.

To achieve this technological assessment, an UML-based MBT approach have been chosen, supported by a commercial MBT tool [1]. A specific adaptor of generated tests to executable scripts has been developed. It targets the format supported by the pre-existing DGA internal test bench for cryptographic components. Therefore, a full tool chain supporting the entire model-based testing process, from test generation models to executable test scripts was used, allowing a precise assessment of the MBT approach in this context. The main results from this work are twofold:

- The subset of the UML/OCL notation used to formalize the expected behavior of the cryptographic components for test generation purpose was adequate, allowing the right abstraction level, and providing enough expression power to capture the specification details for automated test generation;
- Test selection criteria based on behavioral coverage of the model, which is the standard way to generate functional tests with the MBT tool used for the project, was not sufficient to several security testing objectives; Therefore, a novel test purpose language has been set-up, implemented, and used during the project, to drive test generation from the models. It has allowed to better cover the security testing objectives defined for the cryptographic components under test. This test purpose mechanism makes it possible to reason in terms of sequence of states and actions, allowing to produce negative test cases (tests that make trial of what should not happen, by contrast with positive tests that make trial of what the system should do).

B. Originality of the work

Model-based testing is an active research area since mid 90 (e.g.[2] [3]), and commercial MBT tools are available since mid 2000 (see [4] for an updated summary on this subject). A MBT User Conference has been set-up in 2011 with industrial experience reports in several application domains (see conference web site - [5]) such as embedded systems, telecom middleware and enterprise application software. These MBT experience reports mainly focus on functional test generation, which is the current mainstream use of MBT tools.

This paper addresses both functional testing and security testing from defined test objectives. Functional test generation is mainly related to covering positive cases and a reasonable number of negative cases. Security testing is more related to robustness testing: regarding a defined security test objective, the testing goal is to apply as much as possible non nominal use of the component's API to achieve the testing objectives.

Then, model-based testing for cryptographic components relates to two main challenges: 1. Testing security mechanisms to ensure that their functionality is properly implemented; 2. Performing risk-based security testing motivated by understanding and simulating the attacker's approach and detecting possible vulnerabilities.

In our context, the security testing objectives defined for the qualification phase of cryptographic components by the security experts at DGA cover both aspects. Each security test objective is dedicated to a set of security mechanisms and possible associated vulnerabilities.

C. Contributions

The contributions of this paper are twofold. On one hand, it gives the results of an industrial assessment of MBT for cryptographic components, providing data on the various parts of the MBT process: modeling for test generation using a subpart of the UML/OCL notation, test generation from security test objectives and test execution. This includes the various aspects of knowledge transfer from MBT experts to DGA Test Engineers, and the setting-up of the full process.

On the other hand, this paper describes a new test purpose language that has been developed to capture security test objectives and to drive the test generation process from the model. The test purposes act as test selection criteria, and complete the structural coverage of the model that is supported by the test generation tool. A test purpose is a high level expression that formalizes a test intention linked to a testing objective to drive the automated test generation on the behavioral model. Each generated test is therefore a sequence of operation calls with parameter values, which yields a distinguished execution of the model. Their results (the test oracle) are predicted by the model. The generated test cases are automatically translated into test scripts for automated execution on the cryptographic components.

D. Structure of the paper

The paper is structured as follow: Chapter 2 (Presentation of the experimentation) gives a short introduction on the context of cryptographic component qualification testing at the DGA, presents the MBT process in used and describes the two tested cryptographic components. Chapter 3 (Model-based Testing from UML/OCL Behavioral Models for Cryptographic Components) gives the detail of the experimentation including the modeling part and the test selection part, provides data on generated tests and results, and summarizes the technical results. Chapter 4 (Lessons Learned from Experience) provides the lessons learned from experience and summarizes the industrial feedback from demonstrating the experiment. Chapter 5 (Related Works) gives a short review to other research in the area. Finally, chapter 6 gives the conclusion of the paper and provides perspectives for this work.

II. Presentation of the Experimentation

A. Context

The experimentation was performed in the context of the evaluation process of cryptographic products. The regulation requires that these products have a qualification issued by a national authority, the French Network and Information Security Agency (*ANSSI*). This qualification ensures the robustness of the security product against attackers of a defined skill: it indicates that the product can protect information of a given sensibility level (potentially classified information), under specified conditions of use. In this context, the evaluation of cryptographic software supplies to the authority in charge of the qualification all the technical elements needed for this assurance. This evaluation focuses in particular on the ability of the product to ensure information availability, confidentiality and integrity. The evaluation is conducted by an entity independent from designers, developers and sponsors. It requires the use of advanced techniques of security requirement testing.

B. Presentation of the MBT Process

Figure 1 depicts the process for model-based testing from a behavioral test generation model and test purposes that have been used.

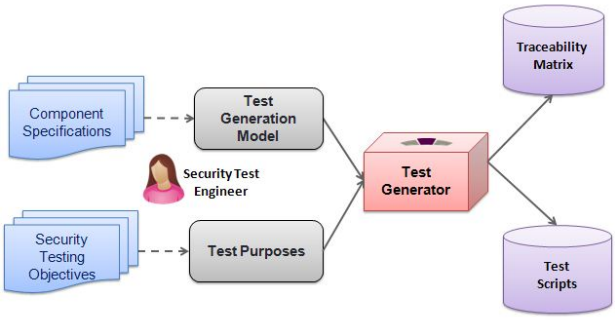

Figure 1. MBT process used for the experimentation

This MBT process is highly incremental: from the specifications of the component and on the basis of security test objectives defined for the project, the security test engineer creates incrementally the test generation model and

test purposes to drive automated test generation. Security testing objectives expose the testing goals for a particular cryptographic components in a detailed but informal form, and a possible way of testing them (how to test). These security testing objectives define the testing strategy and impact the modeling activities and the definition of the test selection criteria for test generation.

In this process, the following artifacts are developed and maintained by the Test Engineer:

- The *Test Generation Model* is a formal artifact developed in a subset of UML/OCL. This is a behavioral modeling fully dedicated for automated test generation.
- The *Test Purposes* are formal artifacts that define the test selection criteria to be applied on the model to generate the required tests. The test purposes formalize the security testing objectives in an adequate test selection language.

This process is supported by a tool-set based on Smartesting MBT tool, that have been extended with a test purpose language (presented in detail in section 3) to drive test generation from security testing objectives and by a publisher in order to translate generated test cases into executable test scripts on the pre-existing test bench[1]. This tool-set consists of the following features:

- Support for behavioral modeling activities;
- Support for test purposes design activities;
- Automated test generation;
- Traceability support between test cases and security test objectives.

C. Presentation of Security components

During the experimentation phase, two components have been targeted: a software cryptographic library and a hardware cryptographic component.

The first experimentation was achieved on a cryptographic library that we will call *SCM* for *Software Cryptographic Module*. This library offers classical cryptographic services like symmetrical and asymmetrical encryption, digital signature, hash computing and random generation. This module embeds an internal sequencing controller which maintains a coherent state of the module in any condition. An objective of our work was to address the underlying state-machine which can't be manually validated due to its complexity (more than a thousand states and sixteen thousands transitions). A second challenge for this experiment was to test combination among error cases. Sometimes, when two or more errors occur simultaneously, a software can have an unwanted behavior. For example, consider the basic case where the smooth-running of a function is represented by a boolean, and where each time an error occurs in the process of the function, the boolean is switched. As a result, for every even number of errors, this status will be positive. It's a trivial example, but side effects of this kind sometimes occurs. Thus, we wanted to be able to check the robustness of our case studies when more than one error is present, and, if possible, for every combination of error types.

The second experimentation was achieved using a hardware cryptographic component specification that we will call *HCM* for *Hardware Cryptographic Module*. This component must allow to encrypt and decrypt messages: it is especially in charge of cryptographic calculus and key storage. It must maintain an isolation between a *red* connection port and a *black* connection port: sensitive information should only circulate on the red port, and has to be encrypted to circulate on the black one. Sensitive information includes plain text messages (before encryption or after decryption), cryptographic keys and parameters dedicated to encryption algorithms.

III. Model-based Testing from UML/OCL Behavioral Models for cryptographic components

A. Behavioral modeling for automated test generation using UML/OCL

In this section, we remind the UML subset for model-based testing proposed in [6] and describe how it has been used to create models for test generation applied on cryptographic components. The goal of the UML subset is to offer precise, necessary and sufficient modeling features to design behavior model for test generation purpose. In sequel, we call this subset UML4MBT.

1) UML diagrams: The UML subset we are using for model-based testing is based on UML class diagrams (for the specification of the SUT's points of control and observation - further referred to PCOs), UML object diagrams (for the definition of the test datum) and OCL (for the specification of the dynamic behavior). The two diagrams make it possible to design comprehensive, precise and interpretable models for automated test generation on finite systems with our MBT tool, and the OCL notation allows to define the expected dynamic behavior of the system under test. We briefly present the different UML diagrams and their own elements used to design models in the context of the project. Note that two other diagrams (UML State-machine Diagrams and Business Process Diagrams) can also be used to define the behavior of the SUT, but as they have not been used in the experimentation, this paper do not present them.

a) Class diagram: The UML class diagram is the static view of the model. It describes the abstract objects of the system and their dependencies. The UML elements available to model the class diagrams are the following:

- Classes, without inheritance, define the types of the objects of the system,
- Reflexive and binary associations, represent the dependencies between classes. The set of available multiplicities are $\{0..1, n, n..*, m..n\}$ (with m and n integers such as 0<m<n). Class associations are not available. A 1-1 association can substitute an attribute with class type,
- Enumeration classes, only composed of literals, are used to model static types and values,

[1]Due to space limit of the paper, the description of the test bench and the translation from generated test cases to executable test scripts is deliberately considered out of scope.

- Class attributes of type defined below, compose the state variables of the system,
- Class operations encapsulate the actions owned by an object. An operation can optionally define inputs/outputs/return parameters (typed as Boolean, integer, enumeration or class), preconditions, and postconditions.

A class attribute is defined with a type among the following:

- Native OCL Boolean,
- Native OCL integer. However a restricted domain is recommended (for example the integer interval [-32668, 36767]),
- Enumeration literal, defined in an enumeration class.

In the two case studies, these UML elements are used the same way:

- classes model business objects; for example a class models the SUT, another class models a Session (which is a global context containing cryptographic contexts and a scheduling context which controls the APIs flow), and another class models a Buffer (which is a physical memory area which may be encrypted);
- associations model relations between business objects; for example, an association links the SUT to all its active Sessions;
- enumerations model a set of abstract values; for example, an enumeration lists all the possible states of a Session (eg. STATE_INIT, STATE_CONNECTE, STATE_HASHINIT, ...)
- class attributes model evolving characteristics of business objects; for example, an attribute models the active state of a Session (which may be one of STATE_INIT, STATE_CONNECTE, STATE_HASHINIT, ...), which serves to model the scheduling control system of a Session (ie. depending on the active state, API calls are accepted/rejected);
- class operations model SUT operations or PCOs; operations can be partitioned in three groups:
  (i) some operations model the *APIs* of the SUT (for example, an operation models the Hash API),
  (ii) others operations model external events which alter datum (for example, a modification of memory) are *alterations*
  (iii) other operations are *observations* which probe model's characteristics and serve as oracles (for example, an observation returns the current state of a Session).

The figure 2 gives some metrics related to class diagrams based on the two experimentations.

b) Object diagram: The UML object diagram lists the objects used to compute test cases. Notice the following restriction: objects cann't be created or deleted dynamically by the actions designed in the model. So all objects used to describe the life cycle of the system must be created in this object diagram. The dynamic creation (resp. deletion) of entities in the concrete system is simulated by creation (resp. deletion) of links between objects in the UML model.

	HCM	**SCM**
# classes	28	12
# associations	33	28
# enumerations	20	13
# enumeration literals	92	89
# literals per enumeration	min: 2 max: 25 avg: 4.6	min: 2 max: 24 avg: 6.8
# class attributes	75	76
# operations	24 API 7 alterations 14 observations	38 API 4 alterations 2 observations

Figure 2. Experimentations metrics related to Class Diagrams

Furthermore, the object diagram is needed to design the initial state of the model. The UML elements available to construct the initial model state are the followings:

- Objects, or class instances, defines datum and entities of the system. All object slots - or attribute instances - must have a value,
- Links, or association instances, define the dependencies between the objects in the initial state of the system.

The figure 3 gives the metrics related to object diagrams based on the two experimentations.

	HCM	**SCM**
# instances	44	54
# links	27	54

Figure 3. Experimentations metrics related to Object Diagrams

2) OCL subset: OCL is used in models for test generation to express the expected dynamic behavior of the system under test. To be able to execute operation postconditions, UML4MBT proposes an operational interpretation of OCL expressions used in such context. For example the OCL expression `self.attribute=true` can be used in two different contexts: a passive and an active context. The first one permits to express constraints on the system under test, while the second one permits to express model state changes. So the expression `self.attribute=true` is interpreted and evaluated as a standard Boolean expression in a passive context. In an active context it is interpreted as an assignment of the Boolean state variable attribute to the value true. Notice that OCL easily lends itself to this particular second interpretation; the context of an expression is deterministic and is comprehensive in the two cases. This non ambiguous interpretation of OCL makes it possible to use OCL as an action language for UML test generation models. The two own interpretations (passive and active) are exhaustively defined in [6].

To be clear, before presenting the OCL passive and active contexts, we first introduce an example. This example deals with the HASH API of the SCM case study. The figure 4 presents the original specification, and the figure 5 presents the modeling of the expected behavior using OCL.

a) Passive OCL context: An OCL expression is used in a passive context to test the model state variables. Its interpretation does not modify the model state. We name here a passive expression such expression. This can be an operation

ErrCode Hash(Session session, UI8 *buffer, UI8 len)
This API hashes a memory area pointed by *buffer*.
It returns OK when everything goes right; BAD_SESSION if the session is corrupted; BAD_ARGS when data are invalids; a negative integer in case of incorrect scheduling. As input, session must have the property HASHINIT. A call to this API adds the property HASH.

Figure 4. Specification of HASH

```
context: SUT::Hash(session:Session, src:Buffer)
postcondition:
0     /**@REQ: Hash*/
1     if(
2           session.isInState(HASHINIT)
3           and not(session.isCorrupted())
4           and not(src.isInvalid())
5           )
6     then
7           /**@AIM: OK*/
8           session.addState(HASH)
9           and self.returnCodes.add(OK)
10          and self.internal.setCurrentSession(session)
11    else
12          if( not(session.isInState(HASHINIT)) )
13          then
14                /**@AIM: NEGATIVE, HASHINIT*/
15                self.returnCodes.add(NEGATIVE)
16          else
17                true
18          endif
19          and if( session.isCorrupted() )
20          then
21                /**@AIM: BAD_SESSION*/
23                self.returnCodes.add(BAD_SESSION)
24          else
25                true
26          endif
27          and if( src.isInvalid() )
28          then
29                /**@AIM: BAD_ARGS*/
31                self.returnCodes.add(BAD_ARGS)
32          else
33                true
34          endif
35    endif
```

Figure 5. OCL representation of SCM's Hash operation

precondition, a decision in a conditional structure or a sub-expression of a passive expression.

The subset of all interpretable OCL passive expressions is presented in the figure 6. In this table, p1 and p2 are passive Boolean expressions; i1 and i2 are Integer expression; e1 and e2 are enumeration literals, o1 and o2 are objects; c1 is a class; s1 is a set of objects.

b) Active OCL context: An OCL expression is used in an active context when its interpretation can change the value of the model state variables or define values for the return parameter of operations. We call such expression an active expression. This can be an operation postcondition, an action in the branches of a conditional structure or a sub-expression of an active expression.

The subset of all interpretable OCL active expressions is presented in figure 7.

c) Behavioral modeling: The UML4MBT set allows designing behavioral models. These behaviors are designed in the operation postconditions (in the class diagram). A set of consecutive actions (ie. active expressions) constitutes the model behaviors. A conditional structure makes it possible to model several and complex behaviors in a single postcondition or action.

Operator group	**Passive OCL Operators**
Boolean operators	equal (p1 = p2), not equal (p1 <> p2), disjunction (p1 or p2), excl. disjunction (p1 xor p2), conjunction (p1 and p2), negation (not p1)
Integer operators	equal (i1 = i2), not equal (i1 <> i2), lesser (i1 < i2), greater (i1 > i2), lesser or equal (i1 <= i2), greater or equal (i1 >= i2), plus (i1 + i2), minus (i1 - i2), unary minus (- i1), multiplication (i1 * i2), division (i1.div(i2)), absolute value (i1.abs()), modulo (i1.mod(i2)), maximum (i1.max(i2)), minimum (i1.min(i2))
Enumeration operators	equal (e1 = e2), not equal (e1 <> e2)
Object/class operators	equal (o1 = o2), not equal (o1 <> o2), is undefined (o1.oclIsUndefined()), get all instances (c1.allInstances())
Collection iterative operators	collect (s1->collect(o1)), select (s1->select(p1)), exists (s1->exists(p1)), for all (s1->forAll(p1)), any (s1->any(p1))
Conditional assignment	if p1 then ... else ... endif

Figure 6. Passive OCL operators

Active OCL Operator	**Effect**
e1=expr	assigns the evaluation of the right hand expression to the left hand element
coll->includes(o) coll->excludes(o)	adds/removes an element to/from a collection
o.oclIsUndefined() coll->isEmpty()	deletes a link/collection of links
coll.forAll(exp)	iterates on a collection, and applies the active expression on each element
and	acts as a separator between two active expressions

Figure 7. Active OCL operators

For example, the postcondition of `Hash` (see figure 5) defines nine behaviors. These behaviors correspond to the nine full paths of the postcondition, considering the alternative branches of the *if ... then ... else* structures and their combination *via* the *and* keyword. The table 8 presents the three first behaviors.

Id	**Target context**	**Target effect**
1	session.isInState(HASHINIT) and not(session.isCorrupted()) and not(src.isInvalid())	session.addState(HASH) and returnCodes.add(OK) and self.internal.setCurSession(session)
2	not(session.isInState(HASHINIT) and not(session.isCorrupted()) and not(src.isInvalid())) and not(session.isInState(HASHINIT)) and session.isCorrupted() and src.isInvalid()	returnCodes.add(NEGATIVE) and returnCodes.add(BAD_SESSION) and returnCodes.add(BAD_ARGS)
3	not(session.isInState(HASHINIT) and not(session.isCorrupted()) and not(src.isInvalid())) and not(session.isInState(HASHINIT)) and session.isCorrupted() and not(src.isInvalid())	returnCodes.add(NEGATIVE) and returnCodes.add(BAD_SESSION)
...	...	...

Figure 8. Subset of behaviors of the *Hash* API

The first behavior deals with the success case of the operation. Its context is the verification of the condition of the first *if ... then ... else* structure. Its effects are (i) updating the current state of the session,(ii) saving the return code OK, and (iii) updating the current active session. The eight others behaviors are related to error cases. Each behavior deals with a particular combination of the three kinds of errors (if the session's state is valid or not, if the session is corrupted or not, and if the buffer is valid or not). Effects of these behaviors basically list

the expected return codes.

Table 9 gives some metrics related to behaviors based on the two experimentations.

	HCM	**SCM**
# lines of OCL	1216	3771
# lines of OCL per operation	min: 26 max: 135 avg: 48.6	min: 9 max: 205 avg: 99.2
# behaviors	358	8356
# behaviors per operation	min: 7 max: 30 avg: 15	min: 2 max: 1729 avg: 220

Figure 9. Metrics related to modelled behaviors

B. Test generation

1) Generating tests based on a behavioral model coverage of the OCL expressions (Structural approach): Models for test generation make it possible to generate tests by covering paths of the OCL postconditions of operations. The test generation method uses symbolic state exploration of the model (see [7] for more technical details) to cover the various expected behaviors (called test targets) formalized by OCL postconditions. More precisely, within an operation, a test target is a pair defined by an effect of the OCL expression and a context in this OCL expression to produce the effect. For each (reachable) test target, the generator produces a test case that is a sequence of model operation calls with parameters aiming to activate the test target from the initial state defined in the model.

Figures 10 and 11 presents two test cases obtained respectively for the first and second behavior of the Hash API. The aim is to illustrate how the automated test generation mechanism, targeting two distinct behaviors, produces two distinct test cases.

The figure 10 presents the test case obtained for the first behavior of the Hash API, which is the behavior related to a correct and successful call of the API. The context of the behavior is: (i) calling the API in the adequate state of a session, using a valid session, and using a valid buffer.

```
0   sutInstance.Initialise(session_Alice, ...)
1       returnCodesInstance.check_code(OK)
2       session_Alice.check_state(ALLOC)
3   sutInstance.InitialiseHash(session_Alice)
4       returnCodesInstance.check_code(OK)
5       session_Alice.check_state(ALLOC)
6       session_Alice.check_state(HASHINIT)
7   sutInstance.Hash(session_Alice, aBufferInstance)
8       returnCodesInstance.check_code(OK)
9       session_Alice.check_state(ALLOC)
10      session_Alice.check_state(HASHINIT)
11      session_Alice.check_state(HASH)
```

Figure 10. Test case related to the successful call of the Hash API

Eleven steps compose the computed test case. Three steps are API calls which stimulate the SUT (0, 3, and 7), whereas the others are observations which serve as oracles. The last stimulation (ie. step 7) activates the targeted behavior. Other stimulations are automatically computed by the underlying MBT technology, because these steps are required to make the SUT being in a configuration suiting the context of the targeted behavior:

- step 3 makes the used Session being in the adequate HASHINIT state
- in the same way, step 0 is required to make the call of the step 3 a success

After each stimulation, some observations are automatically fired. They allow to check the return codes of each API call, and allow to check the internal state of the used Session.

As a second test case sample, the figure 11 presents the test case obtained for the second behavior of the Hash API, which is an unsuccessful call of the API, where (i) the session is not in the adequate HASHINIT state, (ii) the session is altered, and (iii) the buffer is altered.

```
0   sutInstance.Initialise(session_Alice, ...)
1       returnCodesInstance.check_code(OK)
2       session_Alice.check_state(ALLOC)
3   sutInstance.corruptSession(session_Alice)
4   sutInstance.Hash(session_Alice, InvalidBufferInstance)
5       returnCodesInstance.check_code(NEGATIVE)
6       returnCodesInstance.check_code(INVALID)
7       returnCodesInstance.check_code(BAD_SESSION)
8       returnCodesInstance.check_code(BAD_ARGS)
9       session_Alice.check_state(ALLOC)
```

Figure 11. Test case related to an unsuccessful call of the Hash API

For this test case, the underlying MBT technology computes a test case quite different from the first one:

- the InitialiseHash API is no longer used, so that the call of the Hash API in step 4 is made in a non suitable state;
- the step 3 corrupt the used session *via* a specific control point;
- the call of Hash uses an invalid buffer.

2) Covering test objectives with test purposes: A test purpose is a high level expression that formalizes the test intention linked to a test objective to drive the automated test generation. In this project, a test purpose language has been developed in order to capture the security testing objectives defined to test the cryptographic components. Security testing objectives are expressed informally, and defined on the basis of the experience of the Security Test Engineer at the DGA. We propose test purposes as a mean to exercise the system to validate that it behaves as predicted by the model w.r.t. these security testing objectives. Based on his/her know-how, an experienced Security Test Engineer will imagine possible scenarios in which (s)he thinks some security property might be violated by an erroneous implementation, and then on the basis of this test intention, (s)he will formalize test purposes to drive the automated test generation.

The test purpose language we are presenting is called *Smartesting test purpose language*. This is a textual language dedicated to test engineer. Its semantics is based on regular expressions (see [8] and [9] for more details) and allows the Test Engineer to formalize his/her test intention in terms of states to be reached and operations to be called.

a) Test purpose sample: This example deals with a security testing objective called "Authorization loss during a

stop & start". The figure 12 presents the original specification, and the figure 13 presents its modeling for the SCM case study, using a test purpose.

> When the equipment suffers a stop & start, authentication of users must be considered lost, and users willing to administrate the equipment must log-in first.

Figure 12. Specification of the security testing objective 'Authorization loss during a stop & start'

```
0   for_each_operation $op from any_operation_but Connect
        or Disconnect,
1   use Connect to_activate behavior_with_tag @AIM:OK
2   then use Disconnect to_activate behavior_with_tag @AIM:OK
3   then use $op
```

Figure 13. Expression of the security testing objective 'Authorization loss during a stop & start'

The language relies on combining keywords, to produce expressions that are both powerful and easy to read by a Test Engineer.

Basically, a test purpose is a pattern of a test sequence, ie. a sequence of important stages to reach. A stage is a set of operations or behaviors to use, or/and a state to reach. Transforming the pattern into a complete test case, based on the model's behaviors and constraints, is left to the MBT technology. To illustrate this statement, in the first stage of the test purpose of the figure 13 (line 1), the Test Engineer only defines to use the Connect API that leads to a correct connection (OK). Finding the relevant inputs of the API is left to the underlying MBT technology.

Moreover, at the beginning of the test purpose, the test engineer can define quantified variables on sets of operations/behaviors/literals/... These quantifiers are reused in the pattern part of the test purpose. Each combination of quantifiers produces a specific test objective. For example, the test purpose of the figure 13 defines the $op quantifier on a list of 36 operations. Hence, this test purpose will lead to 36 specific test objectives, each dedicated to a specific operation.

b) Syntax: The syntax of the language is defined by means of the grammar given in Figure 14. In this figure, underlined text represents final keywords of the grammar.

The language makes it possible to design test purposes as a sequence of quantifiers or blocks, each block being composed of a set of operations (possibly iterated at least once, or many times) and aiming at reaching a given target (a specific state, the activation of a given operation, etc.).

A dedicated test purpose language editor has been implemented by Smartesting as Eclipse plug-in. Its aim is to provide a means to express testing objectives at a high level, close to a textual representation or by using OCL expressions in the same way that in the behavioral model.

Figure 15 shows the Smartesting test purpose editor. It supports keywords to factorize some parts of the test purposes. It makes it possible to re-use parts of the test purposes, and makes it easier to maintain them.

This test purpose mechanism acts as a test selection criteria in the model-based testing process. It completes the structural

```
test_purpose      ::= ( quantifier_list , )? seq
quantifier_list   ::= quantifier (, quantifier )*
quantifier        ::= for_each_behavior var from behavior_choice
                    | for_each_operation var from operation_choice
                    | for_each_literal var from literal_choice
                    | for_each_instance var from instance_choice
                    | for_each_integer var from integer_choice
                    | for_each_call var from call_choice
operation_choice  ::= any_operation
                    | operation_list
                    | any_operation_but operation_list
call_choice       ::= call_list
behavior_choice   ::= any_behavior_to_cover
                    | behavior_list
                    | any_behavior_but behavior_list
literal_choice    ::= <identifier> (or <identifier>)*
instance_choice   ::= instance (or instance)*
integer_choice    ::= { <number> (, <number>)* }
var               ::= $<identifier>
state             ::= ocl_constraint on_instance instance
ocl_constraint    ::= <string>
instance          ::= <identifier>
seq               ::= bloc (then bloc)*
bloc              ::= use control restriction? target?
restriction       ::= at_least_once
                    | any_number_of_time
                    | <number> times
                    | var times
target            ::= to_reach state
                    | to_activate behavior
                    | to_activate var
control           ::= operation_choice
                    | behavior_choice
                    | var
                    | call_choice
call_list         ::= call (or call)*
call              ::= instance.operation(parameter_list)
operation_list    ::= operation (or operation)*
operation         ::= <identifier>
parameter_list    ::= (parameter (, parameter)*)?
parameter         ::= free_value
                    | <identifier>
                    | <number>
                    | var
behavior_list     ::= behavior (or behavior)*
behavior          ::= behavior_with_tag tag_list
                    | behavior_without_tag tag_list
tag_list          ::= { tag (, tag)* }
tag               ::= @REQ: <identifier>
                    | @AIM: <identifier>
```

Figure 14. Grammar of the Smartesting Test Purpose Language

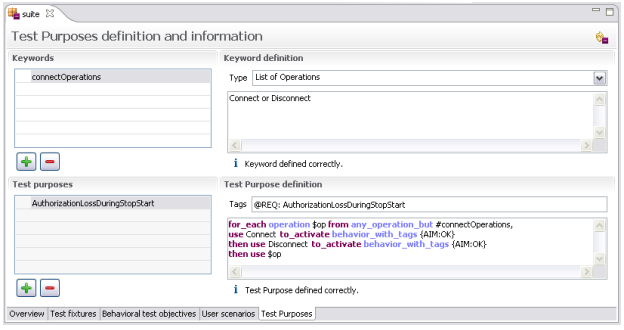

Figure 15. Smartesting test purpose editor

model coverage and makes it possible for the Test Engineer to express directly some security testing objectives, by combining sequences of actions of the system to be called, along with the description by means of predicates of the states to be reached by these sequences of calls. The language is designed to be easy to use by a Test Engineer. (S)he writes test purposes

using model's artifacts, with constructions close to usual computer programming paradigms. That frees him/her from manipulating mathematical notations such as in the temporal logics.

The generation mechanism from test purposes proceeds as follow. Firstly, a processing extracts generalized test targets from the test purposes. One test purpose relates to several (potentially many) generalized test targets. A generalized test target is a sequence of intermediate goals for the test generator. Secondly, the symbolic test generation engine tries to cover each generalized test target to produce one test case. In the sequel of this paper, we call test targets generated from the structural analysis of the model and generalized test targets obtained from test purposes by the unique wording 'test targets'. Indeed, test targets are an intermediate artifacts in our MBT process between the test selection criteria associated to a model and the automatically generated test cases.

C. Results summary

In this section, we give the results of the application of this combined approach (structural model coverage and test purposes) for the two cryptographic components (HCM and SCM). The starting points were:

- the specifications of the components, explaining how they respectively behave; these documents were the references for the creation of the test generation model;
- a high level document listing security requirements, that analysts, developers, and testers should take care of.

Based on the later document, security testing objectives have been defined by Test Engineers. Tests related to some security testing objectives were produced using the structural model coverage approach, and others were produced using test purposes. The figures 16 and 17 present the list of security testing objectives considered during our experimentations. For each security testing objective, the figure details which approach (structural model coverage or test purposes) has been chosen, and how many test targets are related to it. Due to the confidential nature of these cryptographic components, we cannot give more details on each individual security testing objective. We focus more on the process itself and associated results (i.e. the test selection criteria that have been used).

Security Testing Objective	**Approach**	**# Test targets**
Reachability and completeness of transitions	Structural Model Coverage	161
Management of return codes	Structural Model Coverage	358
Input correctness	Structural Model Coverage	296
Deletion of related datum	Structural Model Coverage	69
Re-initialization of memory areas	Structural Model Coverage	40
Flow of red elements	Test purposes	41
behavior of the component in the 'alarm' state	Test purposes	25
Manual firing of the 'auto-test'	Test purposes	7

Figure 16. Test generation methods for HCM

These tables show that testing objectives may be covered either by the structural coverage of behaviors in the model, of by using specific test purposes. We firstly start be achieving

Security Testing Objectives	**Approach**	**# Test targets**
Reachability and completeness of transitions	Structural Model Coverage	855
Management of return codes	Structural Model Coverage	2812
Input correctness	Structural Model Coverage	2058
Authorization loss during a stop & start	Test purposes	36

Figure 17. Test generation methods for SCM

testing objectives using structural model coverage, and then, for the remaining testing objectives not yet covered, *ad hoc* test purposes have been developed. We can see from the numbers that most of the testing objectives are managed by the structural coverage of the model (5 on 8 for HCM, and 4 on 5 for SCM). But, without the test purpose mechanism, we would not be able to complete the coverage of the testing objectives. This is true both for HCM and for SCM.

From the generated test targets, test cases are then computed using symbolic state exploration of the model. Please notice that on one hand a test target may be unreachable (the test generator is able to demonstrate it), and on the other hand one unique test case may cover several test targets.

Test cases related to the HCM case study were produced using a standard desktop PC (6 GB of RAM, 2 CPUs at 3 GHz), whereas those related to the SCM case study were produced using a high performance computer (66 GB of RAM, 24 CPUs at 2.67 GHz).

Figures 18, 19 and 20 give some metrics on the computed test cases:

- figure 18 gives general metrics on computed test cases (total number of computed test cases, total number of computed model operation calls, min./max./avg. number of operation calls per test case)
- figures 19 and 20 detail the number of test cases related to each security testing objectives, for each cryptographic component

	HCM	**SCM**
Total number of test cases	385	2127
Operation calls:		
- Total number of operation calls	1285	17419
- Min. number of operations calls per test case	1	1
- Max. number of operations calls per test case	9	11
- Average number of operations calls per test case	3.3	8.1

Figure 18. Test cases metrics

Security Testing Objectives	**# Test cases**
Structural model coverage for	
- Reachability and completeness of transitions	117
- Management of return codes	312
- Input correctness	259
- Deletion of related datum	51
- Re-initialization of memory areas	30
Test purposes for	
- Flow of the red secured elements	41
- behavior of the component in the 'alarm' state	25
- Manual firing of the 'auto-test'	7

Figure 19. Distribution of test cases per security requirements on HCM

As we have seen, a test case may cover several test targets, so the number of produced test cases is at most equal to the

Security Testing Objectives	# Test cases
Structural model coverage for	
- Reachability and completeness of transitions	750
- Management of return codes	2127
- Input correctness	1578
Test purposes for	
- Authorization loss during a stop & start	36

Figure 20. Distribution of test cases per security requirements on SCM

number of test targets. This explains why the 358 test targets of the security testing objective called *management of return codes* of the HCM case study are covered by only 312 test cases.

IV. Lessons learned from experience

A. UML4MBT as convenient notation used in an incremental MBT process

During the project, the know-how for developing and maintaining test generation models using UML4MBT has been transferred from the MBT experts to the Test Engineers in charge of the cryptographic component qualification phase. This transfer was easy (few weeks) and the Test Engineers found the notation adequate and powerful enough to translate the specifications of HCM and SCM into test generation models.

The modeling approach is incremental: modeling and test generation are done following short iterations in a process guided by the testing objectives. Generated test cases can profitably be used as non regression scenarios to validate model changes.

B. MBT is an efficient method to test these kind of applications

We have identified three main benefits of using MBT for such cryptographic components:

- A better coverage of the security requirements and testing objectives in the qualification phase being based on the systematic and powerful character of automatic tests generation;
- An acceleration of the qualification phase based on capitalization on model artifacts associated to the security component, the test purposes and the fact that some parts of a model can be re-use on various projects;
- An early modeling phase improves the development processes by deleting specification ambiguities and validate end-user needs very upstream.

C. Structural model coverage is not enough to ensure security testing objectives

MBT makes use of test selection criteria that define how to select the tests to be computed from the model.

Structural behavioral coverage of the model exercises the functionalities of the system by *directly* activating and covering the corresponding operations. The test purposes act as dynamic selection criteria in the sense that they orchestrate the successive calls of the operations of the model. On the two cryptographic components, HCM and SCM, this allowed a complete coverage of the testing objectives defined for each component.

D. Threats to validity

The two cryptographic components, HCM and SCM, are fully representative of the family of components that are commonly tested by the Test Engineer team. The complexity of SCM is important with more than 8000 atomic behaviors (a pair condition / effect - see section III.2.c) formalized in the OCL postconditions of the test generation model. On another hand, the (security) testing objectives that have been managed during the project was defined by Security Engineers on the basis of security requirements for these products and they are also fully representative of the test objectives commonly managed by the Test Engineers.

At the beginning of the project, Test Engineers involved in the experimentation were new in MBT, with very few information on the concepts and on the tooling. Similarly, the MBT experts involved in the project didn't have any knowledge on the cryptographic components and the manner to test them. Therefore, after a three-days training on the MBT technology for the Test Engineers, we organized an iterative process, involving a pair Test Engineer / MBT expert to design the test generation model, to generate the test cases and to adapt generated test cases into executable test scripts for the pre-existing test bench. After practicing MBT on the two cryptographic components, the Test Engineers are now able to manage in autonomy new MBT projects.

To summarize, this experimentation was achieved in realistic conditions, as well for the cryptographic components that have been managed, the testing objectives that have been satisfied and the team involved in the project. The results obtained are related to the adequacy of the MBT4UML modeling style for this kind of applications and to the need to complete behavioral model coverage by test purposes as test selection criteria in order to satisfy the testing objectives.

V. Related works

Our test purpose language supports a *"Scenario-Based MBT"* approach as proposed in the classification of [10]. This scenario-based approach allows to extend MBT based on structural coverage criteria of the model (see [3] for a detailed presentation of various coverage criteria used in MBT).

There are many approaches in the literature that introduce test purposes in MBT. This concept has been particularly studied in the MBT approaches based on Input/Output Labeled Transition Systems or Symbolic Transition System such as TGV [11], STG [12], TorX [13] or Agatha [14]. One particularity of our test purpose language, behind the link with UML4MBT modeling concepts, is the capability to define expressions and constraints mixing states and actions. Another characteristics is the textual language format and the capability to reuse keywords, to facilitate the use by the Test Engineer, who is already in charge of the creation and the maintenance of the test generation model using UML4MBT modeling style.

Regarding MBT on cryptographic software, the focus of previous publications was mainly on cryptographic protocol. Rosenzweig and al. [15] proposes a Dolev-Yao intruder model to perform attacks with Spec explorer. Dadeau and al. [16]

proposes 7 mutation operators to simulate implementation leaks into 50 HLPSL models of protocol. Our work is more related to the test of the whole cryptographic component through its APIs, than on a single protocol.

VI. Conclusion

Cryptographic components are part of a class of applications where the functional behavior is fully or mainly dedicated to ensure security-related functions. For such security components (e.g. cryptographic components, smart card software, ...), functional and security testing are mixed-up. The testing objectives relate to the verification of conformance of the (security) functions with the component's specifications (a kind of positive testing) but also aim to ensure that counter-measure for potential attacks are operational (a kind of negative testing).

For such security components, the application of MBT techniques leads to two main differences with respects to the usual MBT for pure functional testing:

- the test generation model, which represents the expected behavior of the component under test with respect to the testing objectives, may be extended with some operations describing potential stimuli from an attacker ; For example, in the test generation models of HCM and SCM, the alteration operations represent such potential events (see section III.A.a).
- the test selection criteria supported by the MBT technology should allow to generate positive and negative test cases. In our context, positive test cases are generated using behavioral coverage of the OCL post-conditions. The test purpose language helps to formalize some security testing objectives in a way that supports negative testing. For example, the test purpose language can be used do put the security component into a critical state (for example a blocked state) and then trying to apply any possible stimuli to change that state in an unauthorized manner.

Therefore, the approach we have introduced in this paper may be used for security component testing, by constructing test generation model extended with stimuli that may used by attackers, and by mixing structural coverage and scenario-based approach to satisfy the testing objectives.

Acknowledgment

This work is sponsored by the ANR ASTRID OSeP project (On-line and Off-line Model-based Testing of Security Properties, ANR 11 ASTR 002) http://osep.univ-fcomte.fr.

It has been also partially supported by FP7 SecureChange project (Security Engineering for Lifelong Evolvable Systems) http:www.securechange.eu/ and the ITEA2 Diamonds project (Development and Industrial Application of Multi-Domain Security Testing Technologies) http://www.itea2-diamonds.org.

References

[1] "Smartesting web site," http://www.smartesting.com, September 2012.

[2] I. El-Far and J. Whittaker, "Model-based software testing," in *Encyclopedia of Software Eng.*, J. Marciniak, Ed. Wiley, 200, pp. 825–837.

[3] M. Utting and B. Legeard, *Practical Model-Based Testing - A Tools Approach*. Morgan & Kauffmann, 2006.

[4] I. Schieferdecker, "Model-based testing," *IEEE Software*, vol. 29, no. 1, pp. 14–18, 2012.

[5] "MBT UC web site," http://www.model-based-testing.de/mbtuc11/, September 2012.

[6] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting, "A subset of precise UML for model-based testing," in *3rd int. Workshop on Advances in Model Based Testing*, 2007, pp. 95–104.

[7] F. Dadeau, F. Peureux, B. Legeard, R. Tissot, J. Julliand, P.-A. Masson, and F. Bouquet, "Test generation using symbolic animation of models," in *Model-Based Testing for Embedded Systems*, ser. Series on Computational Analysis, Synthesis, and Design of Dynamic Systems. CRC Press, 2011, pp. 195–218.

[8] J. Julliand, P.-A. Masson, and R. Tissot, "Generating security tests in addition to functional tests," in *AST'08, 3rd Int. workshop on Automation of Software Test*, Leipzig, Germany, May 2008, pp. 41–44. [Online]. Available: http://doi.acm.org/10.1145/1370042.1370051

[9] E. Fourneret, M. Ochoa, F. Bouquet, J. Botella, J. Jürjens, and P. Yousefi, "Model-based security verification and testing for smart-cards," in *ARES 2011, 6-th Int. Conf. on Availability, Reliability and Security*, Vienna, Austria, Aug. 2011.

[10] M. Felderer, B. Agreiter, P. Zech, and R. Breu, "A classification for model-based security testing," in *VALID 2011 : The Third International Conference on Advances in System Testing and Validation Lifecycle*. IARIA, 2011, pp. 109–114.

[11] C. Jard and T. Jéron, "Tgv: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems," *Int. J. Softw. Tools Technol. Transf.*, vol. 7, no. 4, pp. 297–315, 2005.

[12] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva, "STG: A symbolic test generation tool," in *TACAS'02, Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 2280. Springer, 2002, pp. 151–173.

[13] G. J. Tretmans and H. Brinksma, "TorX: Automated model-based testing," in *First European Conference on Model-Driven Software Engineering*, Nuremberg, Germany, Dec. 2003, pp. 31–43.

[14] C. Bigot, A. Faivre, J.-P. Gallois, A. Lapitre, D. Lugato, J.-Y. Pierron, and N. Rapin, "Automatic test generation with AGATHA," in *TACAS 2003, Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference*, ser. LNCS, H. Garavel and J. Hatcliff, Eds., vol. 2619. Springer, 2003, pp. 591–596.

[15] D. Rosenzweig, D. Runje, and W. Schulte, "Model based testing of cryptographic protocols," in *Trustworthy Global Computing*, ser. Lecture Notes in Computer Science, R. De Nicola and D. Sangiorgi, Eds. Springer Berlin / Heidelberg, 2005, vol. 3705, pp. 33–60, 10.1007/11580850_4. [Online]. Available: http://dx.doi.org/10.1007/11580850_4

[16] F. Dadeau, P.-C. Héam, and R. Kheddam, "Mutation-based test generation from security protocols in HLPSL," in *ICST 2011, 4th Int. Conf. on Software Testing, Verification and Validation*, M. Harman and B. Korel, Eds. Berlin, Germany: IEEE Computer Society Press, Mar. 2011, pp. 240–248. [Online]. Available: http://dx.doi.org/10.1109/ICST.2011.42