Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Complementary Test Selection Criteria for Model-Based Testing of Security Components*

Julien Botella[1], Jean-François Capuron[2], Frédéric Dadeau[3], Elizabeta Fourneret[1], Bruno Legeard[1,3], Florence Schadle[2]

[1]Smartesting Solutions and Services, 18 rue Alain Savary, 25000 Besançon, France
[2]DGA Information Superiority, France
[3]FEMTO-ST Institute (UMR CNRS 6174) - University of Bourgogne Franche-Comté, Besançon, France

Received: date / Revised version: date

Abstract. This article presents a successful industrial application of a model-based testing approach to the validation of security components. We present a smart combination of three test selection criteria applied to testing security requirements of components such as Hardware Security Modules (HSM). This combination relies on the use of static test selection criteria, namely structural model coverage, complemented by dynamic test selection criteria, based on abstract test scenarios or temporal properties, designed to target corner cases of security functional requirements. Our approach is implemented in an industrial and scalable MBT tool. We evaluated and successfully applied it on three real-world security components. The outcome of these experiences showed that the three test selection criteria target distinct kinds of errors in the software and are able to reveal inconsistencies in the specification. Moreover, a 5-year experience of working with both manufacturers and evaluators of security components, along with other industial collaborations, showed that the approach is easy to adopt in the industry and the time spent to learn the methodology is negligible with respect to its benefits. Finally, the approach can be completely applied in a more general context on systems that underlay thorough validation of compliance to specifications or audits.
Keywords: Model-Based Testing; structural coverage; test scenarios; temporal properties; security components

1 Introduction

Testing security components should be a rigorous and, thus, expensive task due to their specific nature, which requires increased efforts to be validated against security standards such as Common Criteria [16]. Security components, such as Hardware Security Modules (HSM), store cryptographic keys and perform cryptographic operations, for example signing data (messages, authentication information, documents, etc.). Security components validation has to address two features of these modules. On the one hand, a security component implements security functions, such as cryptographic algorithms, including symmetric and public-key cyphers. On the other hand, the interaction with the component is allowed through an Application Program Interface (API). This kind of APIs allow the exchange of information and access to critical data in a secure way. Such APIs are commonly defined by specifications, e.g. PKCS#11 [37] or GlobalPlatform [19]. Due to this specific and critical nature of the security components, they may contain exploitable weaknesses and vulnerabilities that can appear at various levels (cryptographic algorithms, protocol or API level).

Even if cryptography can be considered as perfect, it has been shown that security flaws can arise from an unexpected usage of the API provided by the component [20]. Though many techniques are proposed for testing cryptographic protocols [21,56], very few techniques exist in the literature for testing the applicative parts of the components, which are the main source of errors in these systems (around 83% [38]). More generally, most errors in an implementation are related to an incorrect implementation of specifications [31]. Moreover, the existing approaches dedicated to API analysis and testing are manual or specific to key and key-sensitive information extraction from cryptographic devices [11].

Model-Based Testing (MBT) approaches have shown their usefulness for systematic compliance testing of systems that undergo specific standards [6,52,7] describing the *functional requirements (FR)* of the system. Indeed, MBT relies on the use of a formal model to describe

* This research was supported by the French ANR ASTRID Maturation MBT_Sec project (ANR-13-ASMA-0003)

the functional behavior of the system under test (SUT). The model is then exploited to produce abstract test cases that are concretized and executed on the SUT. Moreover, as the model predicts the expected behavior of the SUT, it provides a test oracle, based on which a test verdict is established. MBT is thus well-suited to conformance testing, when one wants to ensure the compliance of the SUT to a given standard.

In addition to functional aspects, standards of security component define also *security functional requirements (SFR)*. The system's security depends on the usage of the security-related functions (such as hashing, signing, signature verification etc), which create strong interconnection between the security and the functionality. Consequently, specific sequences of commands have to be respected to successfully perform security functions. Thus, in this context, *security functional testing* aims to ensure that the system correctly implements these sequencing aspects. From our experience, static test selection criteria (for example behavioral coverage) are insufficient to properly address this issue of validating specific sequences of commands [41,12]. This represents a challenging task in the validation of security components, that we propose to address by using complementary model-based test selection criteria.

In this article, we propose a tailored MBT approach, depicted in Figure 1, that combines and proposes a guiding methodology for the use of existing static and dynamic test selection criteria to perform functional security testing of cryptographic components [12,15]. This approach relies on a model made of UML diagrams augmented with Object Constraint Language (OCL) pre- and postconditions to formalize the SUT and its behavior. Functional tests are obtained by applying a structural coverage of the OCL code describing the operations of the SUT (functional requirements). This approach is complemented by two dynamic test selection criteria that make it possible to generate additional tests that would not be produced by a structural test selection criterion. The first dynamic selection criterion is based on temporal logic expressed in a dedicated temporal language called *TOCL* (for Temporal OCL) [15]. Such properties make it possible to cover the security functional requirements (SFR) by formalizing the sequencing properties of commands (for instance *"a money transfer operation should never be authorized unless a user is successfully logged in"*), to systematize the generation of test cases that illustrate or exercise such sequencing properties or to simply validate an existing test suite by evaluating its coverage. Although TOCL properties can easily and naturally express such temporal aspects, this formalism is not suitable for expressing specific and complex chains of command calls with meaningful combinations of input parameters that a security expert may want to exercise on the system to cover an SFR, due to his or her knowledge on the system and background experience. For this reason, we propose a second and complementary dynamic test selection criterion, called *Test Purposes (TP)* [12]. A TP is an abstract test scenario close to textual representation, that allows the test engineer to describe specific sequences of operations that exercise the system's behavior (for instance *"before performing a money transfer, log in and log out several times several users"*). TP can capture the expressiveness of TOCL properties, however their expressivity allows to write test scenarios that are more generic than TOCL properties. As illustrated in Figure 1 TOCL properties and Test Purposes can be used to capture the security functional requirements. However, these artifacts relate to the MBT model, as they rely on the model operations and data to respectively express the properties and the test scenarios. While the test model itself describes the behavior of the system, these additional artifacts provide a means to drive the test generation without adding test directives in the behavioral model.

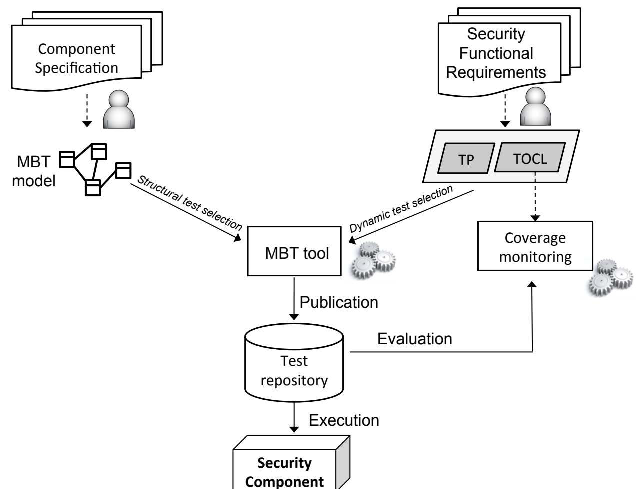

Fig. 1: The MBT Process for Security Components

For clarity reasons, we define the terminology used in this paper as defined by ISTQB (International Software Testing Qualifications Board) [10] and the MBT Taxonomy [53]:

- **test selection criteria**: the criteria used to guide the generation of test cases or to select test cases in order to limit the size of a test suite.
- **static test selection**: these criteria determine the tests to be generated based on the structural coverage of the OCL code of an operation. These criteria aim to exercise a given behavior of an operation which represents a test target that only focuses on this operation.
- **dynamic test selection**: these criteria determine the tests to be generated based on test case specifications in some formal notation, and provided in addition to the model. Contrary to static test selection criteria, they aim to exercise the dynamics of the system by considering several operations in a test. Thus,

for simplicity reasons, in this article, we refer to these criteria as *dynamic test selection criteria*.

The difference between static and dynamic test selection criteria resides in the underlying test intention: static test selection criteria aims to validate a given operation of the system, by focusing the test on this operation that represents the test objective (i.e., the coverage of a given behavior). On the contrary, dynamic test selection criteria aim to exercise the dynamics of the system, by considering the specific use of several operations in a test.

This approach has been experimented on 3 real-life security components: PKCS#11, a platform-independent API for cryptographic tokens such as hardware security modules or smart cards, SCM, a software cryptographic module, and GlobalPlatform, a last-generation smart card operating system. Our experiments have shown the usefulness, accuracy, and relevance of this combination of test selection criteria for functional security testing, in terms of: test generation, error detection capabilities, test suite evaluation and reporting for security audits or Common Criteria evaluations. This approach has notably been experienced during a 5-year partnership with the DGA (Direction Générale de l'Armement) Information Superiority, part of the French Ministry of Defence, which showed that the Test Purposes and TOCL properties could be adopted with relatively low effort, and resulted in a benefit for the validation team. Notice that, in addition, other industrial usages of the approach have been made with various national or international industrials which are the clients of the Smartesting company.

The contributions of this article address the following three research questions.

RQ1 *To what extent do dynamic test selection criteria complement static test selection criteria to cover security functional requirements of security components?*
We propose a combination of the three complementary test selection criteria for functional security testing, that are well-suited to validate the sequencing of API command calls, which is the main challenge in the validation of security components.

RQ2 *What are the benefits of introducing dynamic test selection criteria to the model-based testing of security components?*
We report three real-world experiments of this approach that illustrate the efficiency of each test selection criteria to target specific errors, and the benefits that can be provided to the targeted users (both manufacturers and evaluators of security components).

RQ3 *What is the additional cost of handling dynamic test selection criteria to the MBT process of security components?*
During several research projects, we had the opportunity to experiment the use of these complementary test selection criteria by our industrial partners (both product manufacturers, such as Gemalto for smart cards, and security evaluators such as evaluators from the DGA). Their feedback indicated that the Test Purpose language and the TOCL properties formalisms were easy to learn and put into practice. As a consequence, these two extensions have been transferred and integrated into an industrial and commercial tool, Smartesting CertifyIt [1], and is now available to the users of this technology.

To summarize, the contributions of the article are the following:

- Methodology for complementing a set of model-based test selection criteria to validate security functional requirements (applied to security components) ;
- Full integration of the complementary set of test selection criteria into an industrial MBT tool suite. The test selection criteria of TP and TOCL within this work have been pushed at a software technology readiness level TRL 6 [2] [30].

We have experimented and evaluated our contributions on 3 real life industrial case studies during 5-year industrial experience.

The rest of the article is organized as follows. Section 2 presents the context of this work, namely the security components and the existing static test selection criterion, historically implemented in the Smartesting CertifyIt test generator. Section 3 introduces the two additional dynamic test selection criteria. Then, Section 4 reports and discusses three experiments that we ran during various research projects, in collaboration with industrial partners. Section 5 presents and compares the related works. Finally, Section 6 concludes and presents the future directions that arise from this 5-year industry experience.

2 Context

To ensure safe and secure communication with the security components and their API, their interfaces undergo well-defined standards. Example of such standard is GlobalPlatform (GP) [19]. Figure 2 illustrates on a high-level the link between the security component offering cryptographic services, the defined API and the third-party applications. In addition, other security layers might regulate the communication between the API and the third-party applications. Although it is often assumed that the cryptographic algorithms are correctly

[1] http://www.smartesting.com/en/certifyit

[2] TRL levels are commonly used for evaluation of software in many research projects. TRL 6: Representative model or prototype system, which is well beyond that of TRL 5, is tested in a relevant environment. Represents a major step up in software demonstrated readiness. Examples include testing a prototype in a live/virtual experiment or in a simulated operational environment. Algorithms run on processor of the operational environment are integrated with actual external entities.

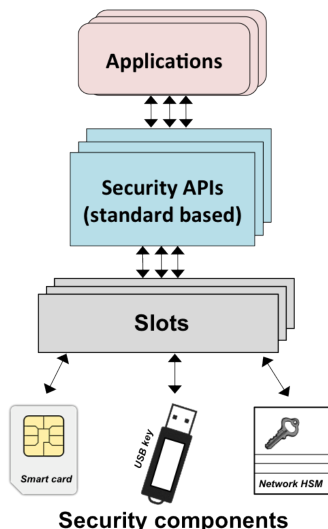

Fig. 2: General high-level environment of security components

implemented, misuses of the cryptographic algorithms in the API and non-conformances to the standard may lead to a number of vulnerability issues. To ensure security and safety, the cryptographic components should be thoroughly tested to assess their conformance to these standards.

Our work was conducted in the context of the evaluation process of cryptographic products performed by the DGA. This process, among others, includes black-box testing techniques in order to validate the conformance of the tested component to its requirements and to verify security properties such as confidentiality, integrity and availability.

The communication with these tokens is commonly regulated by specifications, for example PKCS#11 [37]. Our approach ensures the API's compliance to such specifications.

Although the application domain of our work is more general and concerns any standard, PKCS#11 fits perfectly to illustrate our approach. For this reason, the remaining part of the paper presents the approach using examples based on PKCS#11, which is one RSA Public Key Cryptography Standard.

2.1 Cryptoki API

RSA Public Key Cryptography Standards (PKCS) propose various standards to promote interoperability and security. Our study focuses on the PKCS#11 V2.20 specification (the official version published in 2004), which defines the interface *Cryptoki*, an API for cryptographic hardware, such as HSM or smartcards. The adoption of this standard for communicating with cryptographic tokens in the industry is nearly omnipresent, even though other complementary interfaces are offered by the security tokens.

Shortly, an API based on the PKCS#11 specification *initiates the communication* with the token before any other function call. Then, in order to perform cryptographic functions, such as signing a message, it *opens a session* and *logs* the user. When a function is called in the token's API with a reference to a specific *object* (for instance a key used for signing a message), the token first checks the permissions of the object in order to allow the usage of the function. Permissions are *attributes* that might be represented as boolean flags representing the properties of an object (for example CKA_SIGN flag of a cryptographic key indicates whether a key can be used for signing a message). Further, accesses to operations and objects are controlled through the interface. In general, to perform cryptographic operations, the user must *log in* to the application. To guarantee security, *Cryptoki* implicitly or explicitly defines security requirements that must hold. Most of these requirements can be assimilated to sequencing properties (e.g. *"a signature verification operation must have been initialized"*).

2.2 An MBT Model for Cryptoki

Our approach relies on the use of an MBT model, written using a subset of UML/OCL, called UML4MBT [12] that is considered by the CertifyIt tool on which our approach relies. More precisely, class diagrams describe the points of control and observation and the elements that model the system under test (SUT), and an object diagram represents its instantiation at the initial state. OCL constraints, applied on the operations of the class diagram express the dynamics of the system.

We designed a full MBT model for PKCS#11, covering a set of 24 functions. Figure 3 depicts a simplified class diagram of the PKCS#11 model, which contains six classes: *Cryptoki, User, Token, Slot, Session, Mechanism*. We represent the API *Cryptoki* that offers to a *User* an interface for communicating with cryptographic tokens, modeled by the class *Token*. Each token is connected to the system through a *Slot*. Finally, once the user has been connected to a *Session*, *Cryptoki* offers cryptographic operations, such as signing a message (e.g. function "C_Sign") or verifying a message signature (e.g. function "C_Verify"), with different cryptographic algorithms, represented by the class *Mechanism*.

The behavioral view of the Cryptoki API is modeled using OCL. To be able to execute operation postconditions, UML4MBT uses OCL as an action language. For example, the OCL expression `self.attribute=value` can be used in two different contexts: a passive and an active context. The passive context (defined by preconditions, and conditions in if structures) is used to express constraints on the model state. Thus, in a passive context the expression `self.attribute=value` is interpreted and evaluated as a boolean expression (equality

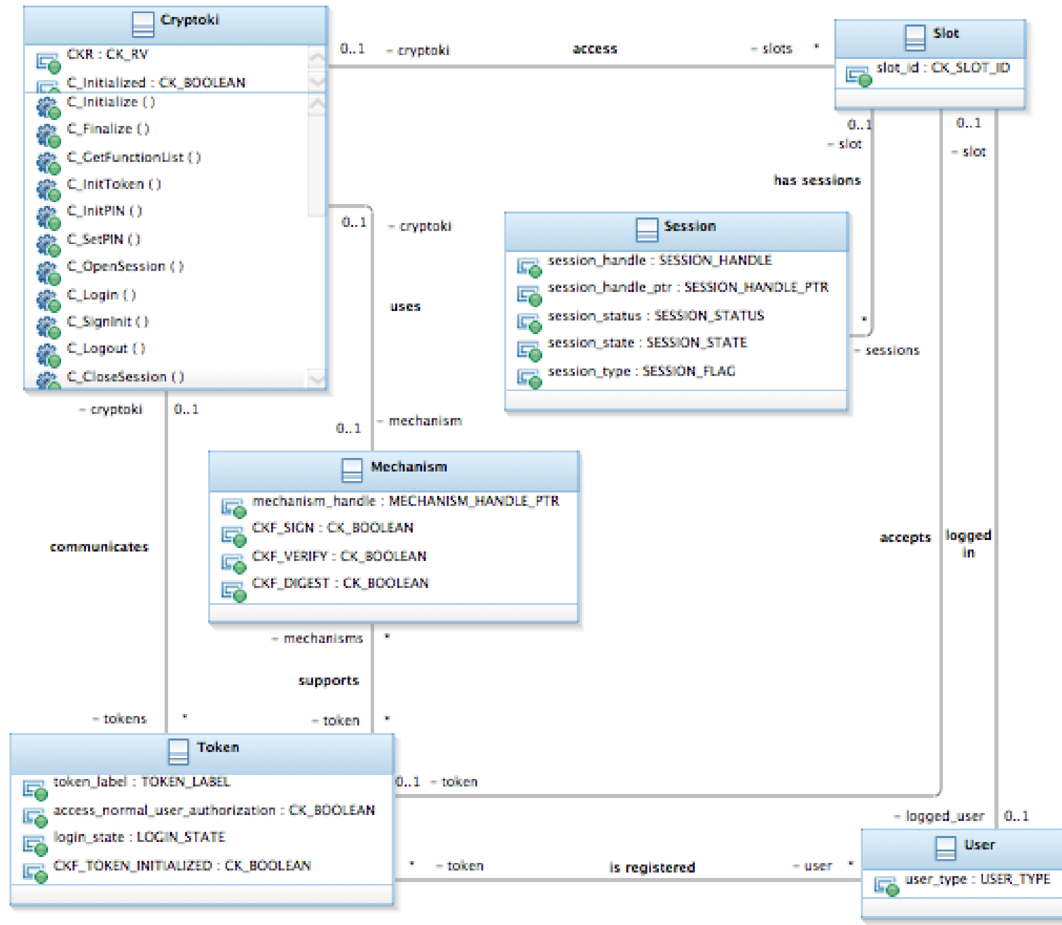

Fig. 3: PKCS#11 MBT model

check). The active context (by opposition to the passive context) is used to express model state changes. Thus, the expression is considered as an assignment of *value* to state variable *attribute*. Notice that the context of an expression is deterministic and 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 [14].

We illustrate the modeling of the expected function behavior in Figure 4, which shows an OCL sample of the “C_SignInit” function, which initializes the data signature function. The OCL postcondition captures the model behaviors for the function. Further, using tags @REQ and @AIM, we annotate each behavior in this postcondition. We consider two types of tags: @REQ, a high-level requirement, and @AIM, the behavior itself, seen as a refinement of the high-level requirement. Both tags are followed by an identifier. For instance, function ”C_SignInit” has one high-level requirement identifier @REQ, which is the name of the function, and three model behaviors, identified by the tag @AIM: (1) the Cryptoki API has not been initialized – *CRYPTOKI_NOT_INITIALIZED*, (2) the normal user has not been

—*@REQ:C_SignInit*
if *self.initialized* = *false* **then**
  —*@AIM:CRYPTOKI_NOT_INITIALIZED*
  result = *CKR_CRYPTOKI_NOT_INITIALIZED*
else
  if *session.loggedUser.oclIsUndefined()* **then**
    —*@AIM:USER_NOT_LOGGED_IN*
    result = *CKR_USER_NOT_LOGGED_IN*
  else
    —*@AIM:OK*
    result = *CKR_OK*
  end
end
return result

Fig. 4: OCL postcondition sample of C_SignInit

logged in – *USER_NOT_LOGGED_IN* and (3) successful execution of the signing initialization function – *OK*.

The PKCS#11 model is directly designed from the specification. It shares a common API with an actual implementation of the standard (operation names and parameters, return values, etc.). This model has been validated using the Smartesting Certify simulator that

```
input: Test Model M
output: Test Cases TC
begin
  TT ← compute_test_targets(M)
  TC ← ∅
  for each tt ∈ TT do
    tc ← compute_test_case(M,tt)
    merge(tc, TC)
  done
end
```

Fig. 5: Algorithm for the test generator

Guard:
self.initialized=true **and**
session.loggedUser.oclIsUndefined()
Assignment:
result = CKR_USER_NOT_LOGGED_IN
Tags:
—@REQ:C_SignInit
—@AIM:USER_NOT_LOGGED_IN

Fig. 6: Test target of the C_SignInit function

makes it possible manually test the model by invoking the different operations, and checking the current model state at each step.

2.3 Structural test selection criteria with Smartesting CertifyIt

We present in this section the static/structural test selection criterion based on behavioral coverage that is implemented within the Smartesting CertifyIt test generator.

The basic algorithm of the test generator is described in Figure 5. It takes as an input a test model M and computes a set of test cases. The following paragraphs describe the differents steps that are thus performed.

Test Target Computation. The test generator starts by computing, from the MBT model, the test targets to cover, by considering structural decision coverage criterion. Thus, each control path of the control flow graph of the operation represents a behavior of the operation. As OCL does not contain any iterative structure, the number of paths (only introduced by if-then-else structures) is finite. A control-flow graph is built based on the conjunction of the pre- and postcondition described by the OCL code. Finally, each test target is identified by a set of tags (labelling the considered control path), which refer to a requirement covered by the behavior.

Figure 6 describes a behavior of the “C_SignInit” function, identifying its guard, the assignments (both extracted from the OCL constraints) and the tags identifying the behavior. From there, the test generation engine will compute a test case, as a sequence of operations that, from the initial state, make it possible to reach a state satisfying the guard of the behavior. The test verdict can be established by using the return values of the operations. In addition, the model may contain a specific kind of operations, called observations, that can be used to observe internal model state variables, to be compared to an actual value of the SUT, in order to refine the verdict.

The precondition of the behavior (also called *guard*) in Figure 6) describes the test target, namely, a predicate that characterizes a state from which the operation can be invoked to result in the activation of the considered behavior.

Test Case Computation The test generator applies the structural criterion to decompose the potential disjunctions in the decisions, and, thus, to produce test targets. Then, the tool generates a test case by computing a sequence of steps, from the initial state, that reaches a model state in which the test target is satisfiable. To achieve that, CertifyIt uses a custom solver with symbolic animation. This technique consists in simulating the execution of the model using symbolic parameters. Each operation activation gathers constraints (the path conditions in the operation) that are evaluated by the solver to check if there exists an instantiation of the symbolic variables that satisfies these constraints. If a solution is found, the considered sequence of operations can be kept to reach the target state. The invocation of the operation from which the test target originates is then concatened to the sequence, so as to create the test case. This process makes it possible to compute test cases, as sequences of operation calls that reach a specific test target.

A test case, such as given in Figure 7, is defined as a sequence of steps, each step being defined as a tuple $(op, parameters, tags)$ in which *op* designates the operation that is invoked, *param* is the instantiation of the parameters, and *tags* is the set of tags that are covered by this invocation. The set of tests based on a given test selection criteria and a model is called a *test suite*.

On the PKCS#11 example, given in Figure 4, the tool generates three tests to cover each test target and thus requirement of the “C_SignInit” function. The test cases presented in Figure 7 cover the behaviors *USER_-NOT_LOGGED_IN* and *OK*.

Test *t1* initializes the communication with the token (function “C_Initialize”), opens a session (function “C_OpenSession”), and calls the signature initialization function “C_SignInit” expecting the error code *CKR_-USER_NOT_LOGGED_IN*.

Test *t2* initializes the communication with the token (function “C_Initialize”), opens a session (function “C_OpenSession”), logs in (function “C_Login”), and calls successfully the signature initialization function “C_SignInit” with an existing key for signing messages.

Test	Operation(Params)	Tags (REQ/AIM)
t1	C_Initialize()	C_Initialize / OK
	C_OpenSession(s_rw)	C_OpenSession / OK
	C_SignInit(s_rw, mechanism, key)	C_SignInit / USER_NOT_LOGGED_IN
t2	C_Initialize()	C_Initialize / OK
	C_OpenSession(s_rw)	C_OpenSession / OK
	C_Login(s_rw , user, user_pwd)	C_Login / OK
	C_SignInit(s_rw, mechanism, key)	C_SignInit / OK

Fig. 7: Functional test cases for "C_SignInit" (a subset)

Test Cases Merging. As a final step of the test generation algorithm, the computed test case is added to the set of computed test cases, in a merging step. During this step, the set of test cases is updated to add the new test case, and, if possible, remove existing test cases whose covered behaviors are also covered by the new test. This step aims to reduce the final number of test cases. Thus, a test case may cover several test targets at once: the considered test target, covered by the last operation of the test case, but also additional test targets, that are covered by the previous operations of the test case. As a consequence, the number of test cases is generally smaller than the number of test targets.

2.4 Limitations of Structural Coverage Criteria

During our various collaborations on different case studies, security expert's from the industry analysed the tests generated using structural criteria (behavioral or based on state machines). The analysis showed that although these tests cover entirely the functional requirements, they barely cover half of the security functional requirements.

Indeed, the test generator based on structural criteria computes the shortest path to activate a given behavior, which is not enough for covering the specificities of security functional requirements. For instance, as shown in Figure 7, when testing the log in for a signing operation, one might be more interested in creating longer sequence of steps crossing various states in the system; for instance, initialize the communication with the token, open a valid session, login, then logout and finally try to activate the signing operation, expecting this function to return again the *CKR_USER_NOT_LOGGED_IN* error.

This kind of test cannot be automatically generated using static coverage criteria, unless additional information is added to the model to drive the test generator (e.g. by encoding an automaton with additional model/ghost variables as done in [34]). As a consequence, the useful information is drown in additional noise that pollutes the MBT model, making it really harder to maintain, and restricting the model usage to the test generation process.

To overcome this issue, we propose to use two dynamic test selection criteria: TOCL properties and Test Purposes, which add a certain degree of dynamics to guide the test generator and produce tests covering the security functional requirements (discussed in the next section). Further, our work resulted in fruitful real-life experiments that made evolving the approach towards a mature technology.

3 Complementary Test Selection Criteria for Testing Security Components

In this section, to tackle security components testing, we present the two dynamic test selection criteria that complement the structural coverage criterion presented previously:

- TOCL: to express security functional requirements using temporal properties formalized in a language called *Temporal OCL* (TOCL).
- Test Purposes: abstract test scenarios that capture the procedural aspects and the combination of various function parameters in a test scenario in order to cover security functional requirements.

Finally, we present their integration into the MBT industrial tool CertifyIt.

3.1 Temporal OCL coverage criterion

In a previous work [15], we have introduced the TOCL language that allows one to express security functional requirements by means of property patterns. These patterns are based on Dwyer's seminal paper [23] that have been adapted to our model-based testing approach. In his paper Dwyer showed that a large majority (92%) of common temporal properties on a system could be expressed using a simple set of patterns. We decided to extend these patterns as they represent an easy means for a test engineer to express a requirement, as it is a textual representation with predefined constructs. It replaces a more complex language (such as LTL) or a formalism (such as an automaton) the test engineer may not be familiar with. TOCL properties can be used for two purposes. First, it is possible to evaluate the relevance of an existing test suite w.r.t. a given property. Second, it is possible to generate test cases to cover the uncovered parts of the property.

3.1.1 TOCL

The property description language is a temporal extension of OCL. It relies on the idea that a temporal property is composed by a *temporal pattern* that is applied in a *scope*. Thus, the user can define a temporal property choosing a pattern and a scope among a list of predefined schemas. The scopes are defined from *events* and delimit the impact of the pattern. The patterns are defined from events and *state properties* to characterize execution sequences that are correct. The state properties and the event are described from OCL expressions.

An event is denoted by `isCalled(op, pre, post, {tags})` and represents operation calls. In this expression, `op` designates an operation; `pre` and `post` are OCL predicates respectively representing a precondition and a postcondition. Finally, `tags` represents a set of tags that can be activated by the operation call. Such an event is satisfied on a transition when the operation `op` is called from a source state satisfying the precondition `pre` and leading to a target state satisfying the postcondition `post` and executing a path of the control flow graph of the operation `op` which is marked by at least one tag of the set of tags denoted `{tags}`.
There are 5 *temporal patterns*: (*i*) `always` *oclExpr* means that a state property *oclExpr* is satisfied by any state. (*ii*) `never` E means that event E never occurs. (*iii*) `eventually` E means that event E eventually occurs in a state in the future. This pattern can be suffixed by a bound which specifies how many occurrences are expected `(at least/at most/exactly` k `times)`. (*iv*) E_1 (`directly`) `precedes` E_2 means that event E_1 (directly) precedes event E_2. (*v*) E_1 (`directly`) `follows` E_2 means that event E_2 is (directly) followed by event E_1.

There are 5 *scopes* that can apply to a temporal pattern P: (*a*) P `globally` means that P must be satisfied on any state. (*b*) P `before` E means that P must be satisfied before the first occurrence of E. (*c*) P `after` E means that P must be satisfied after the first occurrence of E. (*d*) P `between` E_1 `and` E_2 means that P must be satisfied between any occurrence of E_1 followed by an occurrence of E_2. (*e*) P `after` E_1 `unless` E_2 means that P must be satisfied between any occurrence of E_1 followed by an occurrence of E_2 and even after an occurrence of E_1 that is not followed by an occurrence of E_2.

To illustrate the test selection using TOCL and the tool, let us consider the specification of PKCS#11 defining the API Cryptoki offering an interface for managing the security and inter-operability of security components. The specification defines various security requirements for which we are able to generate test cases, for example:

> **Security Requirement:**
> *"a user cannot sign a message using the C_Sign operation without login to Cryptoki (using the operation C_Login)".*

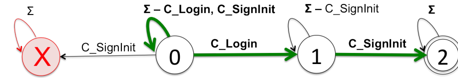

Fig. 8: Automaton of the TOCL property 1

This requirement is interpreted as the user must call successfully a "C_Login" operation before calling successfully "C_SignInit", which initiates the signature operation. This requirement is expressed by the following two TOCL properties.

The first property defines the nominal case, whenever a signature operation is successfully done (model behavior @AIM:OK), it must be preceded by a login operation, performed also with a success. We can distinguish the temporal pattern (*before* the first occurrence of an event) and the scope (*eventually* an event that precedes the previous one). The *events* in this context are calls of functions for which a specific behavior is required (denoted by the tag @AIM:OK).

> **TOCL Property 1:**
> **eventually** isCalled(C_Login, @AIM:OK)
> **before** isCalled(C_SignInit, @AIM:OK)

The second property complements the previous one. This property expresses the mandatory event of logging in a (previously logged-out) user before performing successfully the message signing function.

> **TOCL Property 2:**
> **eventually** isCalled(C_Login,@AIM:OK)
> **between** isCalled(C_Logout, @AIM:OK)
> **and** isCalled(C_SignInit, @AIM:OK)

Each TOCL property is transformed into an automaton. Based on these automata, it is possible, first, to measure the coverage of the property by the existing tests, and, second, to generate tests to complete the coverage of the TOCL properties.

3.1.2 Monitoring the Coverage of TOCL Properties

The first possibility that the TOCL test selection criterion offers is to monitor the coverage of the TOCL property by measuring the coverage of the automata transitions by each existing test, as usual in certification for audits for instance [3]. Consider the two TOCL properties we described and the structural tests generated, given in Fig. 7.

Figure 8 illustrates the automaton produced for the first property. The automaton displays an error state, which is represented by a crossed state. When measuring the coverage of the automaton, we evaluate which transition is covered, and which state is reached, at each step of the test, by matching the event on the transition and the operation call that is performed during the step. If the error state is reached by any set of the test suite, it

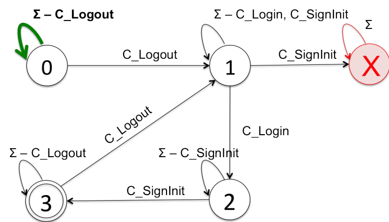

Fig. 9: Automaton of the TOCL property 2

means that the property is violated. In this case, either the property is written in a too restrictive manner, or the model operations are too lax.

The static/structural tests created previously cover entirely this property automaton (as shown in the figure by a thick line). The first test does not reach any terminal state of the automata (state with a double circle). However, the second test reaches the terminal state, and does cover the transitions of the automaton that are labelled by the events in the property (the reflexive transitions, labelled by Σ events do not need to be covered [15]).

Figure 9 depicts the automaton of the second TOCL property, which expresses another point of view of the security requirement. Contrary to the previous TOCL property, the second one is not covered. As shown in the figure, the structural tests cover only one transition of the automaton and do not reach the terminal state. In the following, we discuss the test generation based on TOCL.

3.1.3 Test Generation from TOCL Properties

Once the coverage is evaluated, if any transition of a property automaton is not covered, CertifyIt produces test targets based on the coverage of these transitions and then generates additional abstract test cases.

We have seen that the second property is not fully covered by the existing test cases. Based on TOCL criteria coverage, the generator creates 4 new test targets, which will guide the test generator to create new test cases. For example, the test given in Figure 10 will be generated.

This test case, in complement to the previous tests, logs in, then logs out and logs in again the user before creating the key used for signature and initializing the signature operation. It covers all valid states (from 0 to 3) and 5 transitions of the TOCL automaton, illustrated by thick line in Figure 11.

TOCL coverage is interesting as it systematizes the coverage of test properties that can be expressed easily with predefined patterns. However, it is sometimes necessary provide a means to drive the test generator by a direct user input. This technique is now described.

3.2 Test Purpose coverage criterion

The Test Purpose criterion relies on defining additional dynamic selection criteria in the shape of test scenarios close to textual representations in order to express particular sequences of steps in a test case.

The key idea is to allow the test engineer to drive the test generation engine without having to introduce any additional information in the model, notably to restrict its execution. Thus, the Test Purpose language makes it possible to combine sequences of operations and values for the input parameters for the operations, along with the description by means of state predicates to be reached by these sequences of calls. The language is designed in a textual manner, with constructions close to usual programming languages [41,34,12]. Finally, it is possible to relate each test purpose to a test requirement, providing a traceability means.

For better comprehension of the subsequent parts of the paper, we describe shortly the syntax of the Test Purposes language and associated test generation process.

3.2.1 Test Purpose Language

A Test Purpose is based on regular expressions and allows the test engineer to conceive its scenario in terms of states to be reached and operations to be called, by making possible to select meaningful parameters for the operations.

Figure 12 shows an example of a Test Purpose addressing the following security requirement :
"C_SignInit cannot be correctly executed unless a user has been logged into the token"
with the following (informal) test scenario: first, the communication is initiated with the token, a valid session is opened, the agent logs in, then logs out and then tries to activate the signing function, with return code *CKR_USER_NOT_LOGGED_IN*. Then it log in again and successfully initializes he signing function. This test sequence can be performed for several combinations of sessions and keys.

As shown in the example, the syntax describes *quantifiers*, followed by *blocks*, each block being composed of a set of operations (possibly iterated at least once, or many times) and aiming at activating a given objective (a specific state, the activation of a behavior in an operation, etc.). The quantifiers section defines the quantified variables (preceded by keyword *for_each*) on sets of literals (such as *$SESSION* shown on line 1 in Figure 12), but it can also be defined on a set of operations or model behaviors. The quantifiers specify various contexts in which an objective must be activated within a block (identified by the key word *use* e.g., line 4 in Figure 12), thus, making possible to formalize several test scenarios from one test purpose. The test targets are result of the quantifiers combination.

Operation	Tags (REQ/AIM)	Transition
C_Initialize()	C_Initialize / OK	$0 \to 0$
C_OpenSession(s_rw)	C_OpenSession / OK	$0 \to 0$
C_Login (s_rw, user, user_pwd)	C_Login / OK	$0 \to 0$
C_Logout(s_rw)	C_Logout / OK	$0 \to 1$
C_Login(s_rw, user, user_pwd)	C_Login / OK	$1 \to 2$
C_CreateObject(key1)	C_Login / OK	$2 \to 2$
C_SignInit(s_rw, mechanism, key1)	C_SignInit / OK	$2 \to 3$

Fig. 10: Generated test case covering the TOCL property 2

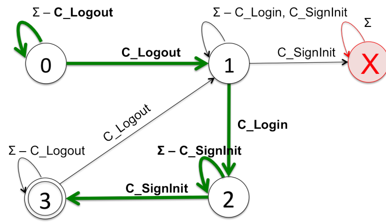

Fig. 11: Coverage of Property 2 by the test of Fig. 10

```
1   for_each litteral $SESSION from s_rw or s_other or null
2   for_each litteral $KEY from key1 or key2 or key3
3   use any_operation any_number_of_times then
4   use sut.C_Login($SESSION,_,_)
5       to_activate behavior_with_tags {AIM:OK} then
6   use sut.C_Logout($SESSION) then
7   use sut.C_SignInit($SESSION,_,$KEY) then
8   use any_operation any_number_of_times then
9   use sut.C_SignInit($SESSION,_,$KEY)
10      to_activate behavior_with_tags {AIM:OK}
```

Fig. 12: Test Purpose for the "C_SignInit" security functional requirement

In more details, the quantified variables in the Test Purpose, given in Figure 12, are: *$SESSION* and *$KEY*, on a list of 3 literals representing abstract values for sessions and cryptographic keys, respectively shown in lines 1 and 2. The next block, calls successfully the "C_Login" function (line 4). In addition, by using the special character _ (underscore), the test generator automatically chooses the parameters that will activate the desired behavior, in this case *successful log in*. The procedure needed to reach the successful log in state is also calculated by the generator. The following blocks (lines 6 and 7) activate the states of successful log out and then successful initialization of the signing function. The last block at line 9, activates a successful initialization of the signing function, while the intermediate objectives to reach are abstracted away using the block with the key words *any_operation any_number_of_times* at line 8.

3.2.2 Test Generation from Test Purposes

The generation process makes usage of the behavioral model written in UML4MBT and the written Test Purposes. Each Test Purpose produces a sequence of intermediate goals given to the test generation engine. We can informally define the test generation process as follows:

1. pick an assignment to the quantified variables,
2. extract the test targets from the test purpose, by means of test case specifications (TCS), which represent a sequence of intermediate goals for the test generation engine.
3. symbolically animate the model to cover the TCS in order to produce a test,
4. return to step 1, and pick the next combination of values for the set of assigned variables

Consider for instance the previous Test Purpose, in Figure 12, it iterates over several sessions and key handles to activate successfully log out and initialization of the signature, immediately one after another. The unfolding of this Test Purpose, by combining the 3 sessions with 3 different keys, produces 9 test targets. Then, for each test target the generator creates one test case, if the test case specification is reachable [3].

The test in Figure 13 illustrates one of the 9 generated tests, it combines a read/write session and one of the listed keys. The comparison of this test case to the test case in Figure 10, generated by the TOCL property 2, shows that their sequence is the same and the test generated from the Test Purposet does not increase the coverage of the property. Nonetheless, it exercises the same sequence for different types of sessions and keys and observes the security component's expected behavior. This kind of combinations of different values for function parameters is important because one specific combination may reveal errors that another one can miss. Thus, in a complement to the TOCL criterion, which in a systematic way and automatically produces meaningful tests, the Test Purpose criterion will allow the creation of tests that will exercise a given procedure with meaningful combinations of sets of input parameters.

[3] A test target is said to be reachable if the test generator is able to compute a sequence of operation calls in the model state space to reach it.

Operation	Tags (REQ/AIM)
C_Initialize	C_Initialize / OK
C_OpenSession(s_rw)	C_OpenSession / OK
C_Login(s_rw, user, user_pwd)	C_Login / OK
C_Logout(s_rw)	C_Logout / OK
C_Login(s_rw, user, user_pwd)	C_Login / OK
C_CreateObject(key2)	C_Login / OK
C_SignInit(s_rw, mechanism, key2)	C_SignInit / OK

Fig. 13: Test case extracted from the Test Purpose

Fig. 14: TOCL Plugin

3.3 Tool suite

We implemented the three test selection criteria: structural, TOCL and Test Purpose in the core of the Smartesting CertifyIt tool suite. It integrates the test selection criteria in the form of plugins for IBM Rational Software Architect (RSA). Each plugin contains a panel for editing and managing the corresponding test selection criterion.

The CertifyIt plugin exports the test targets based on the structural coverage of the model in a format comprehensible by the CertifyIt test generator, as detailed in Section 2.3. Next, the panel dedicated for TOCL allows the user to formalize the security functional requirements in the form of TOCL properties using syntax highlighting and auto-completion features. If any TOCL properties are written, it calculates and exports the test targets based on the defined TOCL properties. If any tests are available, for instance manually written as test scenarios, it further offers property coverage monitoring (see Section 3.1). Figure 14 depicts the TOCL plugin integrated with CertifyIt. The plugin allows to generate a web report providing evidence for traceability between the tests and the (functional) security requirements (an HTML export button on the right up part at the figure). If the coverage is not satisfying, the tool generates the missing tests to fulfil the coverage criteria.

Finally, the Test Purpose panel allows writing the test requirements also using syntax highlighting and auto-completion features. If any Test Purposes are written, the tool calculates and exports the test targets, as de-

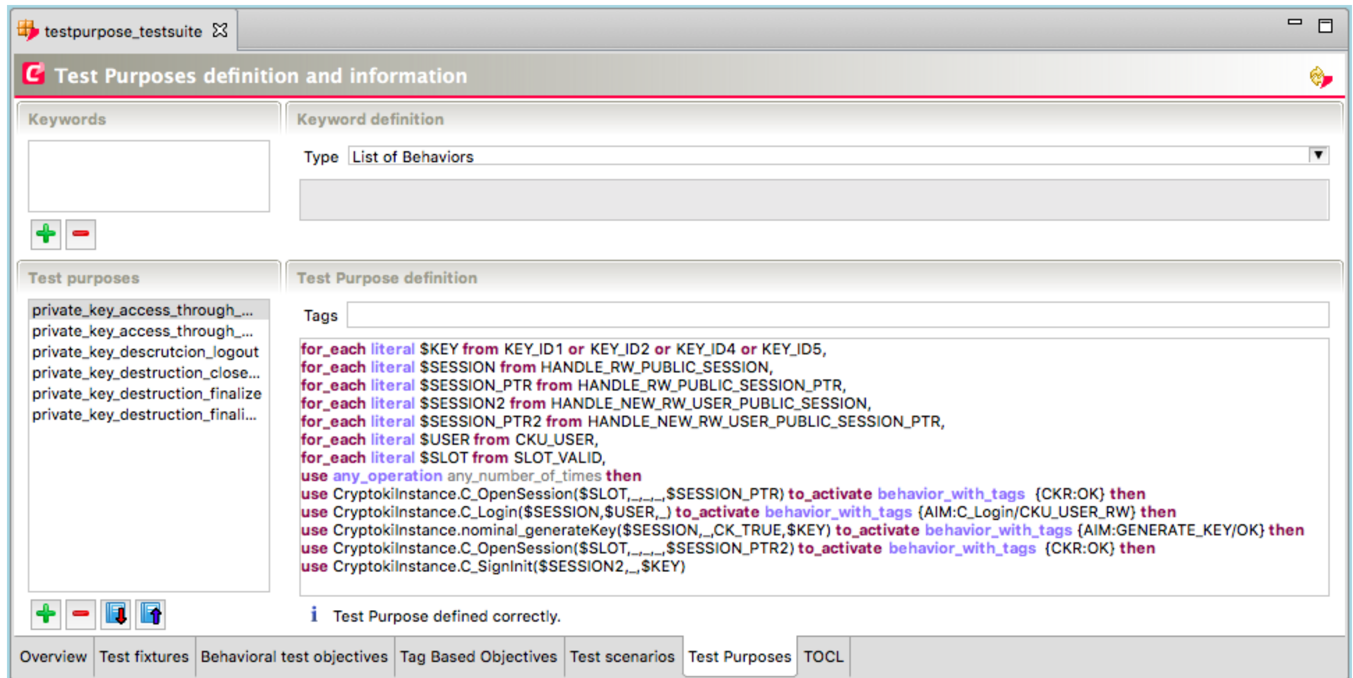

Fig. 15: Test Purpose Plugin

tailed in Section 3.2. The plugin is illustrated in Figure 15.

This work along with the industral experiments, on virtual and on simulated operational systems, discussed in the next section, allowed to improve the efficiency and scalability of the integrated tool chain by reaching high maturity level estimated to TRL 6.

4 Experimentation

In this section we summarize the research questions we addressed by our approach and the results on three studies: the PKCS#11 case study (Section 4.2), a cryptographic component at the DGA (Section 4.3) and a GlobalPlatform smart-card (Section 4.4).

4.1 Research questions

We chose three case studies to answer the following research questions:

RQ1 *To what extent do dynamic test selection criteria complement static test selection criteria to cover security functional requirements of security components?*
We assess this question in terms of the ability to cover security functional requirements that cannot be covered by structural coverage criteria, without supplementary information added to the model. One of the main challenges on the testing security components is to validate especially security functional requirements, such as the sequencing of API command calls. We expect that the combination of test selection criteria will cover these test requirements.

RQ2 *What are the benefits of introducing dynamic test selection criteria to the model-based testing of security components?*
The dynamic test selection criteria allow generation of tests dedicated to the security functional requirements and maintain their traceability. We expect that the systematic use of dynamic criteria will increase the *distinct fault detection*. In addition, the model, as the code, may contain errors, we expect that the introduction of dynamic criteria helps in *debugging the model*. We assess it further in terms of the ability to *simplify the MBT model* by removing the additional information added to the model that pollutes the functional aspect of the model and makes its maintenance harder.

RQ3 *What is the additional cost of handling dynamic test selection criteria to the MBT process of security components?*
This question investigates the cost-benefit relationship of applying our MBT approach for testing security components. We assess this research question in the light of our large experience in MBT with industry partners. During funded projects, we had the opportunity to experiment the use of these complementary test selection criteria by our industrial partners (both product manufacturers, such as Gemalto for smart cards, and security evaluators such as evaluators from the DGA). Furthermore, if *RQ1* and *RQ2*

Table 1: PKCS#11 case study perimeter

Test Requirement category	#FR	#SFR
general purpose	7	4
slot and token management	22	5
session management	32	9
object management	6	2
digesting	28	9
signing	32	10
verifying signatures	31	10
total	158	49

demonstrate that the test suite quality is increased by using additional criteria, we expect this cost effort to be largely acceptable when using dynamic criteria additionally to the structural criterion.

4.2 PKCS#11 experiment

The PKCS#11 case study responds to RQ1, RQ2 and RQ3. We draw our conclusions on RQ1 based on the model-coverage and test generation results of applying the three test selection criteria (structural, TOCL and Test Purpose) on the MBT model of PKCS#11, and compare their coverage further to the model-coverage of the manual tests (delivered with the PKCS#11-based token under test). The analysis of the test execution on the token draws the conclusion on RQ2. Finally, we analyze metrics on time spent in MBT activities to draw conclusions on RQ3.

4.2.1 PKCS#11 Case Study

In an MBT approach, test requirements (functional and security functional ones) are commonly identified from a defined testing perimeter, on the basis of the specifications and available documents. Thus, our case study relies on a subset of the PKCS#11 specification, which, based on industry experts opinion, was qualified as self-contained, realistic and sufficient to illustrate the main aspects of the specification and as well to illustrate the use of model-based testing for such security components. Classically, in the industry, security tokens support only sub-parts of PKCS#11. Thus, the considered perimeter of the study are 24 functions most commonly present in the tokens: general purpose, session, token, key and user management functions, as well as cryptographic functions for digesting, signing messages and verifying signatures.

Table 1 summarizes the functional and security functional requirements of the PKCS#11 case study, according to the groups of functions defined by the PKCS#11 specification. The total number of functional requirements (FR) for the considered subset of PKCS#11 is 158. In addition to these requirements we have identified 49 security functional requirements (SFR).

Table 2: Metrics related to PKCS#11 model

PKCS#11 model elements	
#classes	9
#enumerations	20
#enum. literals	123
#associations	17
#class attributes	34
#operations	24
#observations	1
#behaviors	206
#tocl properties	50
#test purposes	5
#LOC	1308

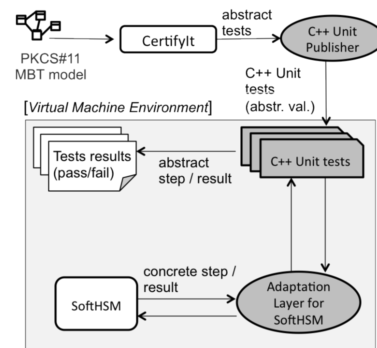

Fig. 16: SoftHSM 2.0.0b2 Test Environment

The MBT model for the PKCS#11 perimeter contains a class diagram, to represent the API and the token environment and the signature of functions (name, input parameters, error codes) and it contains OCL expressions, added as postconditions to the functions to model their behavior. The MBT model covers one part of the test requirements, thus, their full coverage is ensured by the TOCL and Test Purpose coverage criteria. Table 2 provides some metrics about the model elements (for instance classes, attributes, enumerations, operations) and the behaviors, TOCL properties and Test Purposes expressing the requirements. It represents a total of 1308 lines of OCL code (LOC).

The PKCS#11 specification is widely used by the security component providers. In our case, we have applied our approach on an open-source virtual component - SoftHSM (our SUT) - a software implementation of a cryptographic store accessible through the PKCS#11 interface. We chose this token because of its representativeness of an HSM and implementation of most PKCS#11 functions. We also selected this open source virtual security component, because it avoids the use of a physical device, which simplifies the test environment.

*Test environment*The test environment is a Virtual Machine (VM), which contains: the generated tests in C++, the test adaptation layer and test execution scripts. Af-

ter test execution the pass/fail test results are collected in an XML format. We depict the SoftHSM test environment in Fig. 16. More specifically, we created a Virtual Machine (VM) with Ubuntu 32bits and 2GB of RAM on which we installed the SoftHSM v2.

4.2.2 Evaluation

For each test selection criteria, we report the number of covered FR and SFR. The FR are completely covered by the behavioral model, thus we use the static/structural test selection criteria to generate test cases. The SFR, depending on the requirements, are either already covered by the behaviors, or by creating a TOCL property or a Test Purpose. Table 3 summerizes the information about requirements coverage.

Further during the experiment, for each test selection criterion (structural, TOCL properties and Test Purposes) we created a separate test suite, for which we generated tests using the chosen criterion and executed them on the SUT. We evaluated the tests cases generation by the 3 combined test selection criteria. In addition, we evaluated the existing manually-designed test suite of SoftHSM. We selected tests that are part of the PKCS#11 perimeter, and imported them as test scenarios in CertifyIt. Each imported test scenario produced one test target. However, we discarded 3 manual tests, because they were non-conform to the specification i.e. the test verdict of the test case is different from the one computed by the specification. In this table, for each test selection criterion (structural, TOCL and Test Purpose), we report the test targets and the corresponding generated tests. Finally, we report the test requirements coverage by the test suites. The model is a representation of the specification defining the testing perimeter, thus we can measure the coverage of the functional and security functional requirements (FR and SFR) by each test suite.

Table 4 summarizes the results of this evaluation. As given in the table, the manual tests cover barely 45% of the functional requirements, showing thus the incompleteness of the manual test suite, compared to the automatically-generated tests, with respect to the specification.

In addition, the dynamic test selection criteria (TOCL and Test Purpose) on their own are not sufficient to cover the FR, as the structural criteria are not sufficient to cover the security functional requirements (SFR). The coverage measure notified that the manual test suite covers 14 TOCL properties, which represent about 16% of the SFR. The analysis of Tables 3 and 4 shows that the structural coverage criterion covers only part of the SFR, and they are complementary covered by all 3 test selection criteria. Each test suite (structural, TOCL, Test Purpose and manual) on SoftHSM was executed on the system under test and the results were collected in log files. Based on these results, we evaluated the number of *distinct faults* revealed by the failed tests. As several tests can reveal one same fault, we were specifically interested in the diversity of the detected faults by each test selection criterion, that we refer to as *distinct faults*. Note, that the failed tests report discrepancies with respect to the specification, but we did not evaluated whether they represent exploitable vulnerabilities. We report these results in Table 5, showing that all three test selection criteria detected a panel of faults, while the manual test suite did not reveal any fault.

In addition, Table 6 details the SoftHSM discrepancies with respect to the specification detected by the failed tests. Each discrepancy is identified by the function, the expected and returned error code by the function. Results from Tables 5 and 6 show that each test suite reveals different discrepancies complementary to the other test suites, thus increasing the detection of distinct faults. We see that there is no intersection between the different faults detected by each test suite.

An MBT model may also contain errors. This process can thus also be used to detect such errors and correct them at the earliest. Indeed, the TOCL coverage measure may possibly result in reaching an error state, which indicates that the property has been violated (as shown in Fig. 8). If the property is correctly written, this identifies an error in the model behaviors that can to be corrected immediately. On the case study, this helped us to correctly implement the sequences of cryptographic operations (digest, sign and verify).

Answer to RQ1: Based on the experimental results in tables 3 and 4, we see that structural test selection criteria are not enough to ensure the security functional requirements. The TOCL and the Test Purpose test selection criteria act as dynamic test selection criteria by orchestrating the calls of functions in the model, and thus increase the coverage of the test requirements, by increasing the number of test steps in the test cases and diversifying them.
Results in Tables 5 and 6 show that each test selection criterion detects a panel of distinct faults, missed by the other criteria. Thus, it confirms their complementarity in detecting additional *distinct faults*.

Answer to RQ2: As for RQ1, results in Tables 5 and 6 show that the three selection criteria augment the fault detection capabilities and shows the *accuracy* of our dynamic test selection criteria w.r.t. structural coverage. In addition, this process can be used to validate the model, and increase the confidence in its correctness, w.r.t. the (security) functional requirements that it is supposed to ensure.

The last part of the experience on the case study concerned the effort involved for creating the MBT model (static and dynamic view separately) and formalizing the security functional requirements in the form of TOCL and Test Purpose, debugging the model by using the

Table 3: Coverage of PKCS#11 test requirements by each test selection criterion

Test Requirement Category	**#Test requirements covered by:**		
	Structural	TOCL	Test Purpose
FR	158		
SFR			
- general purpose		4	
- slot and token management	4		1
- session management	7	1	1
- object management	1	1	
- digesting	2	7	
- signing	2	8	
- verifying signatures	3	7	
- total of covered SFR	19	28	2

Table 4: PKCS#11 test suites metrics

Test Selection Criterion	**#Test targets**	**#Test cases**	**Cov. in %**	
			FR	**SFR**
Structural	206	184	100	40
TOCL	311	90	31	58
Test Purpose	24	24	9	2
Manual	24	24	45	16

Table 5: PKCS#11 test suites execution

Test Selection Criterion	**#Test cases**	**Test execution**	
		#Failures	**#Distinct Faults**
Structural	184	6	5
TOCL	90	12	3
Test Purpose	24	6	3
Manual	24	0	0

test selection criteria and creating the test environment. The time spent in creating the MBT model includes also the time spent to understand the specification. The time spent on the test environment includes the development of the publisher of unit tests in C++, the adaptation layer and understanding the existing SoftHSM test environment.

Table 7 summarizes these results. It took approximately 16 person-days to create the MBT model within the scope defined earlier in this section. It is important to note that for the test selection criteria based on TOCL and Test Purpose, all artefacts were re-used, since they were created when applying the structural criterion. The considered time is evaluated to less than one and half a day for understanding the SFRs and creating the TOCL properties and the Test Purposes, respectively.

Answer to RQ3: If the model artefacts already exist, in terms of cost-effectiveness, results show that *creating the artefacts for the TOCL and Test Purpose criteria* for the PKCS#11 is *largely acceptable* with respect to the benefices. As given in Table 7, the cost of applying TOCL and Test Purpose coverage criteria is about 2 person-days.

4.3 Experiment at the DGA

The DGA experiment responds to RQ1, RQ2 and RQ3. We draw our conclusions on RQ1 based on the security functional requirements coverage and test generation results of applying the three test selection criteria (structural, TOCL and Test Purpose) on the MBT model. This draws the conclusion on RQ2 in terms of benefits of applying the three test selection criteria, for instance scalability and model simplification. Finally, based on experts opinion we draw conclusions on RQ3.

4.3.1 SCM case study

The experimentation was performed in the context of an evaluation process of cryptographic products. The DGA 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.

Our experimentation focused on a cryptographic library we call SCM for *Software Cryptographic Module*. This library offers classical cryptographic services like

Table 6: SoftHSM discrepancies with PKCS#11 specification

Function	Expected output	Actual output	Test suite		
			Structural	TOCL	TP
C_Logout	CKR_USER_NOT_LOGGED_IN	CKR_OK	X		
C_DigestInit	CKR_USER_NOT_LOGGED_IN	CKR_OK	X		
C_DigestInit	CKR_OK	CKR_OPERATION_ACTIVE		X	
C_SignInit	CKR_OK	CKR_OPERATION_ACTIVE		X	
C_SignInit	CKR_OBJECT_HANDLE_INVALID	CKR_OK			X
C_SignInit	CKR_OBJECT_HANDLE_INVALID	CKR_OPERATION_ACTIVE			X
C_SignUpdate	CKR_USER_NOT_LOGGED_IN	CKR_OK	X		
C_Sign	CKR_USER_NOT_LOGGED_IN	CKR_OK	X		
C_SignFinal	CKR_USER_NOT_LOGGED_IN	CKR_OK	X		
C_VerifyInit	CKR_OK	CKR_OPERATION_ACTIVE		X	
C_VerifyInit	CKR_OBJECT_HANDLE_INVALID	CKR_OK			X

Table 7: Effort involved for each MBT activity in person-days

MBT Activity	**#Involved effort**		
	Structural	TOCL	Test Purpose
MBT model creating			
- static view	4		
- dynamic view (FR)	12		
- SFR		1	0.5
MBT model debugging	10	2	0
Test Environment	20	0	0

Table 8: Experimentations metrics related to Class Diagrams

SCM model elements	
# classes	12
# enumerations	13
# enumeration literals	89
# associations	28
# class attributes	76
# operations	38 API
# observations	2
# behaviors	8073
# tocl properties	935
# test purposes	1
# LOC	3771

symmetrical and asymmetrical encryption, digital signature, hash computing and random generation. It embeds an internal sequencing controller which maintains a coherent state of the module in any state of the system. An objective of our work was to address the underlying state-machine which can not be manually validated due to its complexity (more than a thousand states and sixteen thousands transitions).

Based on the available documents (specifications, tables etc), during the project Security Engineers and Test Experts collaborated to define the security functional requirements, which are fully representative of the testing requirements commonly defined for such components.

For the SCM case study, we group the security functional requirements in three categories:

- management of return codes: sometimes, when two or more errors occur simultaneously, a software can have an unwanted behavior. For instance, consider the case of a function, having its status of execution represented as a boolean attribute; the function switches the boolean to its opposite every time an error occurs. A side-effect of this function is that the attribute has a positive status every even number of errors, which can be misused by third-party users or applications. Thus, in the case of a simultaneous errors, our goal is to validate every combination of the possible returned errors.
- command sequencing: this table describes, for each boolean *flag*, composing the internal system state, and for each command of the SCM, if the flag is *required* (R) or *forbidden* (F) in order to execute the command, and, after the command's execution, if the flag is *set* (S) or *erased* (E). An example of such a table is given in Fig. 17.
- authorization loss during a stop & start : verify the authorisation parameters during a starting and shutting down a connection with the token.

The MBT model of the SCM contains a class diagram, to represent the API and the cryptographic component environment and the signature of functions (name, input parameters, error codes) and it contains OCL postconditions, to model the function's behavior. The MBT model covered one part of the test requirements, thus, their full coverage was ensured by the TOCL and Test Purpose coverage criteria, which we discuss more completely in the next section. Table 8 summarizes the metrics related to the UML/OCL class diagram of the SCM case study.

4.3.2 Evaluation

To cover the three security functional requirement of the SCM component we combined the three test selection criteria. The *management of multiple return codes* was completely covered using the structural criterion.

Next, to test a *command sequencing* (for example as the one given in Fig. 17) it is possible to compute a

		flag1	flag2	flag3
Command 1	Before	R	F	
	After	E		S
Command 2	Before	R	F	F
	Before	R	R	F
	After	E	E	S

Fig. 17: Sample of a table of command sequencing

sequencing automaton of the commands in the application.

Initially we represented this automaton using 11 additional classes and 38 operations per class. This resulted in 418 operations and additional 2269 lines of OCL to guide the test generation engine. This resulted in very time consuming maintenance of the model. Rather than exploiting this hardly maintainable model structure, the TOCL mechanism allowed to systematize the approach by defining 7 templates of TOCL properties that check the implementation of the sequencing, which are the following:

- there is no erasure of a flag between its last setting and its subsequent usage (as required).
- once a flag is set, and until it is erased, a command that requires this flag can be invoked.
- in order to execute a command that requires the flag to be set, it has to be set first.
- once a flag has been erased, it has to be set in order to execute a command that requires it.
- there is no setting of a flag between the last erasure of the flag and its use (as forbidden) to execute a command.
- once a flag is erased, and until it is re-set, a command that requests the absence of this flag can be invoked.
- once a flag has been set, it has to be erased in order to execute a command that forbids it.

By considering the SCM table of 18 flags and 37 commands (describing 44 combinations of before/after flags), we were able to easily generate an exhaustive set of 935 TOCL properties. We evaluated the existing test suite on these properties to check if they were covered by the tests. Notice that each property could be documented by its informal expression, instantiated for the considered flag and appropriate commands, providing an interesting and useful feedback for the analysis of the coverage measure.

In Table 9 we present the chosen coverage criterion for each category of SFR, the number of created elements (behaviors, TOCL properties and Test Purposes), the produced test targets and generated test cases, respectively for each SFR category.

The coverage of the defined test requirements by the generated tests was evaluated further by the DGA Test Engineers experts in the domain. Due to confidentiality reasons we cannot give details about the test execution nor evaluation of the TOCL property templates. The Test Engineers confirmed a complete coverage of the testing requirements defined for the component. In addition, they found the TOCL and Test Purpose notations adequate and powerful enough to translate the requirements into tests. As the complexity of the SCM is important with more than 8000 atomic behaviors and the sequencing module has more than thousand states and sixteen thousands transitions. Initially, the test engineers tested the command sequencing, by instrumenting the model. Our process has simplified the model by removing more than 2000 lines of OCL, which represents about 40% of the total OCL. We have generated 935 TOCL properties to validate automatically the module, which includes and completes the initial test suite produced with an instrumented model.

Answer to RQ1: The evaluation of our approach using this case study confirmed that dynamic test selection criteria in addition to structural criteria contribute to the coverage of the defined test requirements. The structural criterion covered the management of multiple return codes. Moreover, the TOCL test selection criterion is well-suited to validate the sequencing of API command calls, which is one of the main challenges in the validation of security components. The Test Purposes were suitable for validating the authorisation parameters.

Answer to RQ2: This analysis showed that our approach: (*i*) has dramatically simplified the model, since all the control of the test generation was removed, to focus only on the behavioral aspects, (*ii*) has systematized the generation of test properties, and thus, test cases, (*iii*) sharpened the validation of commands sequencing by considering not only the nominal sequences, but also corner cases, that were not addressed before, and (*iv*) scaled for large industrial security components.

Furthermore, at the beginning of the project, Test Engineers involved in the experimentation were new to MBT, with very few information on the concepts and on the tools. Similarly, the MBT experts involved in the project did not 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 practising MBT, the Test Engineers are now able to manage in autonomy new MBT projects.

Table 9: Test Selection Criteria Distribution of test cases per SFR on SCM

Security Functional Requirements	**# Elts**	**# Test targets**	**# Test cases**
Structural coverage for - Management of return codes	8333	3203	2398
TOCL coverage for - Command sequencing	935	6251	5620
Test Purpose coverage for - Authorization loss during a stop & start	1	36	36

Answer to RQ3: This case study showed that the dynamic test selection criteria are: (*i*) relatively easily adopted by the test engineers and (*ii*) scalable for handling test generation for large models- more than 8000 atomic behaviors and 935 TOCL properties. It proves thus its cost-effectiveness:
- capitalization of involved effort in creating the MBT model,
- improvement of requirements analysis phase,
- acceleration of the entire qualification phase,
- finally, the trained Test Experts in MBT can reproduce the approach on new projects.

4.4 GlobalPlatform experiment

In this section, we summarize the results of combining the structural and the TOCL test selection criteria on the GlobalPlatform case study, in the context of the TASCCC project[4].

4.4.1 GP UICC case study

GlobalPlatform is an industrial standard for managing resources for multi-application smartcards. It describes all the functionalities and interfaces for managing the administrative aspects of a card all along its life cycle. An important fact related to the GlobalPlatform standard is that it is designed to allow different actors (phone companies, banks, transportation operators, etc.) to coexist on the same card. Such a possibility is offered by the notion of a Security Domain (SD) that represents an application through which all interactions with the operating system are performed.

During the TASCCC project, we focused on GP UICC profile, and specifically on the life cycle of the card, which is expected to comply with a simple state machine displaying 5 states. The OP_READY and INITIALIZED states both indicate that the card is ready to receive commands from the issuer, but not from the card holder. State SECURED means that the card is ready to receive commands from the card holder. If a security violation happens, the card goes to the CARD_LOCKED state.

[4] 2009–2012, funded by the French National Research Agency (ANR)

Finally, when the card is TERMINATED, no command can be successfully invoked. The life cycle of the card is controlled by the applications, which use the *"setStatus"* operation to set the life cycle state, accordingly to the state machine.

We designed three security functional requirements that express the dynamics of the card life cycle:

1. before going to the SECURED state, the card was in the INITIALIZED state.
2. the card can not escape the TERMINATED state.
3. to reach the CARD_LOCKED state from the INITIALIZED state, it is mandatory to go through the SECURED state.

The objective was to check if the configurations that are described in these security functional requirements were covered by the test suite, generated using the structural coverage criterion. This test suite contained 60 test cases aimed at checking the possible changes in the card life cycle state by the *"setStatus"* operation.

4.4.2 Evaluation

Each one of the previous security functional requirements was expressed by a TOCL property [15]. To evaluate the relevance of the test suite w.r.t these requirements we ran the structural test cases on the model and their associated automaton. Thus, based on this evaluation, we monitored the satisfaction of these requirements on the MBT model.

While these tests were not generated from the properties, these latter made it possible to evaluate the quality of the test suite, by measuring the coverage of the corresponding automaton. Our industrial partners in the project (namely the Gemalto company and Serma Technologies, a Common Criteria evaluator) gave us a positive feedback on the use of properties for testing. We further created a publisher that generates a report highlighting the traceability between the test cases and the security requirements.

Property (1) was covered by 13 tests, which performed various actions consisting in reaching the INITIALIZED state and then the SECURED state. Property (2) was covered by 4 tests. However, we noticed

that these tests only focused on reaching the TERMINATED state, without performing any additional action. We thus completed the test suite with additional actions, performed in the TERMINATED state, to check that it was not possible to escape this trap state. Finally, Property (3) was covered by 8 tests that reproduced the chain of successive states:

INITIALIZED → SECURED → CARD_LOCKED.

Validation engineers from Gemalto reported that (extract from evaluation report [8]):

> *The proposed approach* [...] *makes it possible, using a high level of abstraction, to get full test suites that target at best a given perimeter of larger expected results.*

The Common Criteria evaluator of the TASCCC project reported that (extract from evaluation report [45]):

> *It has been validated that the produced tests fully satisfy the usual evaluation criteria applied for this kind of product [i.e. smart cards]. One of the most important criterion is the relevance of the test cases, especially when automatic tools are used. The study shows that the test cases of the TASCCC campaign have the same level of relevance as test cases that would have been manually produced by a validation engineer. The advantage of this approach is to produce more tests and thus exercise the product in additional various contexts.*

Answer to RQ1: Results on this case study showed the complementarity of the structural and the TOCL test selection criteria. Domain experts confirmed that dynamic test selection criteria, such as TOCL, fit perfectly to cover their test requirements.

Answer to RQ2: This case study showed the benefits of using the TOCL coverage in terms of easy reporting for Common Criteria evaluation, by documenting the traceability between the generated tests and the security functional requirements. Experts confirmed that the produced test cases exercise various contexts, thus addressing corner cases.

4.5 Summary

The first case study was a proof of concept that our approach is applicable on a real-life components, increasing the fault-detection capability. The two case studies on SCM component at the DGA and GlobalPlatform were done in an industrial context and they foreground the scalability of our approach for industry components.

4.6 Threats to validity

The experiments were conducted on three real-life security components. Thus, we can conclude that the approach is generalizable to security components underlying specific standards. We discuss below some threats to validity specific for each experiment:

- *PKCS#11:* Threats to validity for this case study concern the subject's skills. Actual effort for all elements is largely dependent on skills and experience and therefore hardly generalisable measure. In an MBT approach when test fails, one may ask the question whether the fault is due to an error in the model, the code or the test adaptation layer. All failed tests were examined manually and to the best of our knowledge of the PKCS#11 specification and SoftHSM implementation, failed tests correspond to faults in SoftHSM v2, revealing non-conformance to the specification.
  Another threat to validity on the PKCS#11 study concerns the domain knowledge on cryptography and the PKCS#11 specification itself, because we only had basic knowledge on cryptography. However, if the MBT approach is integrated within the project, the testing teams have already this knowledge. In addition, we do not had the knowledge of the SoftHSM v2 code, thus the time spent to create the adaptation layer is not measurable. This is linked to the developers/testers experience and we consider it as negligible.
  Finally, a threat to validity is the selection of manual test cases for the considered PKCS#11 perimeter. It is possible that some tests are omitted, due to the use of function not being part of the perimeter. The omitted tests may augment the coverage of the specification, however, it will not impact the drawn conclusions.
- *GlobalPlatform:* A threat to validity on this case study is the choice of security functional requirements. Due to confidentiality reasons we could not work on the existing security functional requirements at Gemalto, thus in collaboration with its security experts we need to define adhoc requirements. This threats to validity concerns RQ1 and RQ2. However, requirements were considered as representative to the type of security functional requirements they work with, and the produced reports were seen as a great benefit for the certification process.

5 Related work

The MBT approach we propose relates to existing work in the domain of Model-Based Security Testing. In addition, the dynamic test selection criteria (TOCL and Test Purpose), relate more specifically to work in the

domains of property-based and scenario-based testing. Our approach is further related to research addressing the combination of MBT approaches.

5.1 Model-Based Security Testing

Model-Based Security Testing (MBST) approaches on the one hand focus on access control policies, security properties, such as privacy or protocols [50,26]. For instance, Le Traon et al. worked on proposing structural test selection criteria of access control policies [39] and test generation based on access control models [42]. Further, they have proposed an approach for test generation using combinatorial testing technique by combining roles, permissions and contexts [43]. Recently, they proposed a tool-supported process for building access-control test models from contracts and access-rules [58]. On the other hand, Anisetti et al. express the privacy properties of a service (P-ASSERT) and propose to generate tests cases based on service models, which further is used in their certification scheme for digital privacy of services [2,3]. Mallouli et al. provided a formal approach to integrate timed security rules, expressed according to Nomad language into a TEFSM functional specification of a system. Then they use the TestGen-IF tool to generate test cases, which are later executed on the system using tclwebtest scripts [40].

On the other hand, MBST approaches aim at discovering vulnerabilities in the systems.

Regarding MBT on cryptographic software, the focus of previous publications was mainly on cryptographic protocol. Rosenzweig and al. [44] proposes a Dolev-Yao intruder model to perform attacks with Spec explorer. Dadeau and al. [21] 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. Fuzzing, applied to data, but also behavioral fuzzing [47], have also been used for security testing with success [33]. Our approach differs in the sense that we rely on external artifacts to drive the test generation. As a consequence, we especially target errors in the security functional requirements implementation. Furthermore, using MBT models for fuzzing can be seen more as robustness testing, for verifying residual discrepencies in the system, it is not possible to formalize security properties and provide evidences for the test strategy (as needed for example in CC evaluations) Smiliarly, on electronic purse protocols has been reported by Jürjens and Wimmel [36], using the AutoFocus tool [48]. This approach relies on the use of various diagrams to model the SUT, and possibly the attacks that can be performed. This approach focuses on testing vulnerabilities using attack scenarios. However, our approach goes a bit further, as it can model attack scenarios to discover security flaws, but it also allows to write test scenarios that target functional security requirements.

In addition, Bortolozzo et al. [11] focus on online generating attacks on PKCS#11 tokens. Hence, in their approach they focus on exploiting vulnerabilities related to extracting cryptographic sensitive information. To this goal, they reverse-engineer a security token to deduce the functions its supports and constructs a model that is sent to a model checker, which produces traces that are directly executed on the token. Contrary to them, our work is offline (tests are generated and stored before executing them on the system) and does not focus explicitly on attacks and extraction of cryptographic information from PKCS#11 based tokens, we used it to evaluate the effectiveness and efficiency of our approach. Our work can be completely applied in a more general context on systems that underlay thorough validation of compliance to specifications or for instance audits.

In addition, although our work can be adapted for vulnerability testing [13], it does not aim to identify and discover potential vulnerabilities, based on risk and threat analysis or based on the information given by databases such as National Vulnerability Database (NVD) or the Common Vulnerabilities Exposure (CVE) database.

5.2 Property-based testing

The notion of property-based testing is often employed in the test generation context. Several approaches [29,49,1] deal with LTL formulae, that are negated and then given to a model-checker that produces traces leading to a counter-example of this property, and thus defining the test sequences. Our work aims at illustrating the property and checking the system's robustness with respect to it. Fraser et al. [28] defines the notion of property relevant test cases, introducing new coverage criteria that can be used to determine positive and negative test cases. Nevertheless, our approach does not rely on LTL, but on a dedicated language easier to manipulate than LTL by non-specialists.

Based on Dwyer's work, jPost [24] uses a property expressed in a trace logic for monitoring an implementation. Similarly, in [46] the authors introduce the notion of observers, as ioSTS, that decide the satisfaction of the property and guide the test generation within the STG tool. Our work differs in the sense that the coverage criteria are not only used as monitors for passive testing, but they can also be employed for active testing.

5.3 Scenario-based testing

The test purpose language we presented in this paper supports a *"Scenario-Based MBT"* approach as proposed in the classification of [25]. This scenario-based approach allows to extend MBT based on structural coverage criteria of the model (see [53] 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 [32], STG [17], TorX [51] or Agatha [9]. Julliand et al. in [41] propose to generate test cases based on B-models and dynamic test selection criteria (also called test purposes) for producing test objectives, represented as regular expressions. They have applied their approach on smart-cards. 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.

5.4 Combining MBT approaches

Finally, the necessity to select MBT techniques according to project's specific needs has already been tackled early in 1991 by Basili and Rombach [5]. In 2005 Basili and Vegas [54] created a tool *Characterization Schema* to suggest a set of technologies that can be used for testing a specific project. Hence, the combination of the suggested techniques and guidance on how to combine them for project remains manual task, which has been studied by Dias-Neto and Travassos [22]. They proposed the successor of the Basili's tool, *Porantim*, to accompany the user towards selecting one or several suitable MBT techniques for testing his system based on the user interaction. *Porantim* is a decision making tool for choosing one or more already existing approaches suitable for testing a given system. They construct a knowledge database of a set of MBT techniques being selected through surveys and systematic reviews not compatible between each other. Wojcicki and Strooper in [57] propose systematic classification for combining verification and validation techniques. However, they do not consider providing information based on adequacy criteria between the project under test and software testing technology. Other work deal with multi-criteria approach, for instance based on resources and schedules, for selection of testing technologies [55]. To the difference of the previous work they do not offer analysis nor support for the combination of technologies. However, combining different MBT technologies requires skills in several modelling notations and learning several tools, which poses a challenge when introducing them into industry. In addition, most probably each MBT tool requires new conception of the model, making extremely hard the reuse of model elements. Contrary to these works we propose a guiding methodology to combine three test selection criteria in an integrated tooled environment to cover on a best possible way (security) functional requirements and avoid cost on multiple models creation.

Our previous work [27] has focused on the combination of model-based testing and verification applied in the context of smart cards. This approach consisted in using UMLsec stereotypes [35] to annotate the model for, first, checking the consistency w.r.t. the security annotations, and, second, driving the test generation by producing dedicated test scenarios originating from the considered stereotypes. This previous work also used the test generation engine of Smartesting CertifyIt, and was a proof-of-concept of this kind of combination for validating smart card security properties. Our work is fruit of multiple research projects that permitted to evolve the current MBT methodology and technology. Contrary to what has been done in [27], in this submitted paper we do not address the combination of model-based testing with model-based verification. We further present novelties and evolutions in the formalization language and methodology. The Test Purpose Langage has evolved within the past years to capture experts experience and we combined it in a new methodology including complementary generation techniques to improve: the coverage of security requirement, the quality and thus fault detection capabilities of a test suite.

Finally, our approach relates to Model-Driven Testing using UML Testing Profile [4]. This latter presents dedicated concepts that make it possible to describe testing artifacts using UML. Such an approach shares the idea of keeping the test generation directives excluded from the behavioral test model, as for the TOCL properties and Test Purposes. However, our approach is different in the sense that UML has been chosen as a notation to describe the system, using a model. Our industrial experience has shown that test engineers are very reluctant to use various formalisms. Thus, we decided to use a textual notation (close from a code in the case of the Test Purposes, and pattern-based for TOCL) which simplifies the definition of the test directives, and does not require the user to design additional models (sequence diagrams or statecharts). The feedback from our various industrial partners showed that it eased the adoption of the approach.

6 Conclusion and future work

In this article, we have presented the integration of three test selection criteria (structural, TOCL an Test Purpose) in one industrial tool suite. We have assessed the process and tool's effectiveness and scalability through a real world case study PKCS#11 and two industrial components. The TOCL and Test Purpose results have proven their accuracy on covering security functional requirements, test generation and monitoring. This led to maturity of the tool to TRL 6 and an official transfer of TOCL in the industry, as a part of the industrial MBT tool - CertifyIt. The results show further that the three test selection criteria are complementary and increase

the effectiveness of the generated tests. In terms of scalability, the effort spent in the MBT activities is largely acceptable with respect to the benefits from applying systematically our approach. In addition, in order to establish the methodology in the industry, results on the effort and time spent to learn the methodology and the tools is very small.

Within the PKCS#11 context, the created MBT model is compliant for PKCS#11 v2.20, 2.30 and 2.40 specifications and can be reused for testing PKCS#11 based security tokens.

However, introducing MBT in the industry remains challenging in terms of simplifying the MBT model constructions, creating valuable reports for certifications, such as for Common Criteria certification [15]. This approach can be generally applied on specifications defining APIs for security components. We foreground it by our current activities of applying the approach for testing two new components: a GlobalPlatform smartcard underlying the TEE (Trusted Execution Environment) security profile [18] and a security component for digital rights management.

References

1. P. Amman, W. Ding, and D. Xu. Using a model checker to test safety properties. In *7th Int. Conf. on Engineering of Complex Computer Systems (ICECCS'01)*, page 212. IEEE, 2001.
2. M. Anisetti, C. A. Ardagna, M. Bezzi, E. Damiani, and A. Sabetta. Machine-readable privacy certificates for services. In R. Meersman, H. Panetto, T. Dillon, J. Eder, Z. Bellahsene, N. Ritter, P. De Leenheer, and D. Dou, editors, *On the Move to Meaningful Internet Systems: OTM 2013 Conferences*, volume 8185 of *Lecture Notes in Computer Science*, pages 434–450. Springer Berlin Heidelberg, 2013.
3. M. Anisetti, C. A. Ardagna, E. Damiani, and F. Saonara. A test-based security certification scheme for web services. *ACM Trans. Web*, 7(2):5:1–5:41, May 2013.
4. P. Baker, Z. R. Dai, J. Grabowski, O. Haugen, I. Schieferdecker, and C. Williams. *Model-Driven Testing: Using the UML Testing Profile*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.
5. V. R. Basili and H. D. Rombach. Support for comprehensive reuse. *Softw. Eng. J.*, 6(5):303–316, September 1991.
6. B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, New York, USA, 1995.
7. G. Bernabeu, E. Jaffuel, B. Legeard, and F. Peureux. MBT for global platform compliance testing: Experience report and lessons learned. In *25th IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Naples, Italy, November 3-6, 2014*, pages 66–70, 2014.
8. J. Bernet. Tasccc project - deliverable 5.5 - report on the industrial use of the tasccc process. Technical report, Gemalto, 2012.
9. C. Bigot, A. Faivre, J.-P. Gallois, A. Lapitre, D. Lugato, J.-Y. Pierron, and N. Rapin. Automatic test generation with AGATHA. In H. Garavel and J. Hatcliff, editors, *TACAS 2003, Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference*, volume 2619 of *LNCS*, pages 591–596. Springer, 2003.
10. International Software Testing Qualifications Board. Standard glossary of terms used in software testing, March 2015.
11. M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel. Attacking and fixing pkcs#11 security tokens. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 260–269, New York, NY, USA, 2010. ACM.
12. J. Botella, F. Bouquet, J.-F. Capuron, F. Lebeau, B. Legeard, and F. Schadle. Model-based testing of cryptographic components - lessons learned from experience. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, Luxembourg, Luxembourg, March 18-22, 2013*, pages 192–201, 2013.
13. J. Botella, B. Legeard, F. Peureux, and A. Vernotte. Risk-based vulnerability testing using security test patterns. In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II*, pages 337–352, 2014.
14. 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*, pages 95–104, 2007.
15. K. Cabrera Castillos, F. Dadeau, and J. Julliand. Coverage criteria for model-based testing using property patterns. In *Proceedings Ninth Workshop on Model-Based Testing, MBT 2014, Grenoble, France, 6 April 2014.*, pages 29–43, 2014.
16. Common criteria for information technology security evaluation, version 3.1, July 2009.
17. D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG: a tool for generating symbolic test programs and oracles from operational specifications. In *ESEC/FSE-9: Proc. of the 8th European Software Engineering Conference*, pages 301–302, New York, NY, USA, 2001. ACM.
18. Global Platform Device Commitee. Global platform tee protection profile version 1.0, August 2013.
19. Global Platform Consortium. Global platform uicc configuration version 1.0, October 2008.
20. V. Cortier, S. Delaune, and P. Lafourcade. A survey of algebraic properties used in cryptographic protocols. *Journal of Computer Security*, 14(1):1–43, 2006.
21. F. Dadeau, P.-C. Héam, R. Kheddam, G. Maatoug, and M. Rusinowitch. Model-based mutation testing from security protocols in HLPSL. *Software Testing, Verification and Reliability*, 2014.
22. A.C. Dias-Neto and G. Horta Travassos. Supporting the combined selection of model-based testing techniques. *Software Engineering, IEEE Transactions on*, 40(10):1025–1041, Oct 2014.
23. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.

24. Y. Falcone, L. Mounier, J.-C. Fernandez, and J.-L. Richier. j-POST: a Java Toolchain for Property-Oriented Software Testing. *Electr. Notes Theor. Comput. Sci.*, 220(1):29–41, 2008.
25. M. Felderer, B. Agreiter, P. Zech, and R. Breu. A classification for model-based security testing. In *Proceedings of the 3rd International Conference on Advances in System Testing and Validation Lifecycle(VALID 2011)*, pages 109–114, 2011.
26. M. Felderer, P. Zech, R. Breu, M. Bchler, and A. Pretschner. Model-based security testing: a taxonomy and systematic classification. *Software Testing, Verification and Reliability*, pages 119–148, 2015.
27. E. Fourneret, M. Ochoa, F. Bouquet, J. Botella, J. Jürjens, and P. Yousefi. Model-based security verification and testing for smart-cards. In *Sixth International Conference on Availability, Reliability and Security, ARES 2011, Vienna, Austria, August 22-26, 2011*, pages 272–279, 2011.
28. G. Fraser and F. Wotawa. Using Model-Checkers to Generate and Analyze Property Relevant Test-Cases. *Software Quality Journal*, 16:161–183, 2008.
29. A. Gargantini and C Heitmeyer. Using model checking to generate tests from requirements specifications. In *Procs of the Joint 7th Eur. Software Engineering Conference and 7th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, 1999.
30. C.P. Graettinger, S. Garcia, J. Siviy, R. J. Schenk, and P.J. VanSyckle. Using the Technology Readiness Levels Scale to Support Technology Management in the DoDs ATD/STO Environments. Technical Report CMU/SEI-2002-SR-027, Carnegie Mellon University and Software Engineering Institute, September 2002.
31. Hadi Hemmati. How Effective Are Code Coverage Criteria? An Empirical Analysis of 274 Faults. In *Proceedings of the IEEE International Conference on Software Quality Reliability and Security*, 2015.
32. C. Jard and T. Jéron. Tgv: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 7(4):297–315, 2005.
33. W. Johansson, M. Svensson, U. E. Larson, M. Almgren, and V. Gulisano. T-fuzz: Model-based fuzzing for robustness testing of telecommunication protocols. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 323–332, March 2014.
34. J. Julliand, P.-A. Masson, R. Tissot, and P.-C. Bué. Generating tests from B specifications and dynamic selection criteria. *FAC, Formal Aspects of Computing*, 23:3–19, 2011.
35. J. Jürjens. *Secure Systems Development with UML.* Springer-Verlag, Berlin, Heidelberg, 2010.
36. J. Jürjens and G. Wimmel. Formally testing fail-safety of electronic purse protocols. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 408–411, Nov 2001.
37. RSA Laboratories. Pkcs#11 specification, 2004.
38. D. Lazar, H. Chen, X. Wang, and N. Zeldovich. Why does cryptographic software fail?: A case study and open problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, APSys '14, pages 7:1–7:7, New York, NY, USA, 2014. ACM.
39. Y. Le Traon, T. Mouelhi, A. Pretschner, and B. Baudry. Test-driven assessment of access control in legacy applications. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 238–247, April 2008.
40. W. Mallouli, M. Lallali, A. Mammar, G. Morales, and A. Cavalli. Modeling and testing secureweb applications. In *Web-Based Information Technologies and Distributed Systems*, volume 2 of *Atlantis Ambient and Pervasive Intelligence*, pages 207–255. Atlantis Press, 2010.
41. P.-A. Masson, M.-L. Potet, J. Julliand, R. Tissot, G. Debois, B. Legeard, B. Chetali, F. Bouquet, E. Jaffuel, L. Van Aertrick, J. Andronick, and A. Haddad. An access control model based testing approach for smart card applications: Results of the POSÉ project. *JIAS, Journal of Information Assurance and Security*, 5(1):335–351, 2010.
42. T. Mouelhi, F. Fleurey, B. Baudry, and Y. Traon. A model-based framework for security policy specification, deployment and testing. In *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems*, MoDELS '08, pages 537–552, Berlin, Heidelberg, 2008. Springer-Verlag.
43. A. Pretschner, T. Mouelhi, and Y. le Traon. Model-based tests for access control policies. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 338–347, April 2008.
44. D. Rosenzweig, D. Runje, and W. Schulte. Model based testing of cryptographic protocols. In R. De Nicola and D. Sangiorgi, editors, *Trustworthy Global Computing*, volume 3705 of *Lecture Notes in Computer Science*, pages 33–60. Springer Berlin / Heidelberg, 2005.
45. D. Rouillard. Tasccc project - deliverable 5.4 - report on the integration of the ate requirements. Technical report, Serma Technologies, 2012.
46. V. Rusu, H. Marchand, and T. Jéron. Automatic verification and conformance testing for validating safety properties of reactive systems. In J. Fitzgerald, A. Tarlecki, and I. Hayes, editors, *Formal Methods 2005 (FM05)*, LNCS. Springer, July 2005.
47. M. Schneider, J. Grossmann, I. Schieferdecker, and A. Pietschker. Online model-based behavioral fuzzing. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 469–475, March 2013.
48. O. Slotosch, S. Molterer, M. Sihling, A. Rausch, B. Schtz, and F. Huber. Tool supported specification and simulation of distributed systems. *Software Engineering for Parallel and Distributed Systems, International Symposium on*, 00:155, 1998.
49. L. Tan, O. Sokolsky, and I. Lee. Specification-based testing with linear temporal logic. In *IRI'2004, IEEE Int. Conf. on Information Reuse and Integration*, pages 413–498, nov 2004.
50. G. Tian-yang, S. Yin-sheng, and F. You-yuan. Research on software security testing. *World Academy of Science, Engineering and Technology*, 70, 2010.
51. G. J. Tretmans and H. Brinksma. TorX: Automated model-based testing. In *First European Conference on Model-Driven Software Engineering*, pages 31–43, Nuremberg, Germany, December 2003.
52. M. Utting and B. Legeard. *Practical Model-Based Testing - A tools approach.* Elsevier Science, 2006. 550 pages.

53. M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, August 2012.
54. S. Vegas and V. Basili. A characterisation schema for software testing techniques. *Empirical Software Engineering*, 10(4):437–466, October 2005.
55. M. Victor and N. Upadhyay. Selection of software testing technique: A multi criteria decision making approach. In D. Nagamalai, E. Renault, and M. Dhanuskodi, editors, *Trends in Computer Science, Engineering and Information Technology: First International Conference on Computer Science, Engineering and Information Technology, CCSEIT 2011, Tirunelveli, Tamil Nadu, India, September 23-25, 2011. Proceedings*, pages 453–462. Springer, 2011.
56. G. Wimmel and J. Jürjens. Specification-Based Test Generation for Security-Critical Systems Using Mutations. In *ICFEM '02: Proceedings of the 4th International Conference on Formal Engineering Methods*, pages 471–482, London, UK, 2002. Springer-Verlag.
57. M. A. Wojcicki and P. Strooper. An iterative empirical strategy for the systematic selection of a combination of verification and validation technologies. In *Software Quality, 2007. WoSQ'07: ICSE Workshops 2007. Fifth International Workshop on*, pages 9–9, May 2007.
58. D. Xu, L. Thomas, M. Kent, T. Mouelhi, and Y. Le Traon. A model-based approach to automated testing of access control policies. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, SACMAT '12, pages 209–218, New York, NY, USA, 2012. ACM.