An Access Control Model Based Testing Approach for Smart Card Applications: Results of the POSÉ Project

P.-A. Masson[1] and M.-L. Potet[2] and J. Julliand[1] and R. Tissot[1] and G. Debois[3] and B. Legeard[4] and B. Chetali[3] and F. Bouquet[1] and E. Jaffuel[4] and L. Van Aertrick[5] and J. Andronick[3] and A. Haddad[2]

1 - LIFC, Université de Franche-Comté, 2 - LIG, INP Grenoble
3 - Gemalto, 4 - Smartesting, 5 - Silicomp/AQL

Abstract: This paper is about generating security tests from the Common Criteria expression of a security policy, in addition to functional tests previously generated by a model-based testing approach. The method that we present re-uses the functional model and the concretization layer developed for the functional testing, and relies on an additional security policy model. We discuss how to produce the security policy model from a Common Criteria security target. We propose to compute the tests by using some test purposes as guides for the tests to be extracted from the models. We see a test purpose as the combination of a security property and a test need issued from the know-how of a security engineer. We propose a language based on regular expressions for the expression of such test purposes. We illustrate our approach by means of the IAS[1] case study, a smart card application dedicated to the operations of Identification, Authentication and electronic Signature.

Keywords: Security Policy, Model Based Testing, Common Criteria, Security Testing, Smartcard applications.

1 Introduction

Generating tests for security policies is still a challenge: it is not fully addressed by nowadays test generation techniques. We consider in this paper access control policies for smart card applications. Our intent is to ensure that security properties are specifically tested, completing in that the functional tests. This work has been performed in the context of the French RNTL POSÉ[2] project (ANR-05-RNTL-01001) that aimed at proposing a methodology for model based security testing, compatible with the Common Criteria methodology.

Common Criteria (CC)[3] internationally define common requirements for the security evaluation of Information Technology products. They classify security requirements into families, and define several certification levels (from EAL1 to EAL7). A high certification level requires the use of formal models for verifying that the system implements its security policies. The ambition of the POSÉ project was to help automating the generation and execution of tests dedicated to the validation of these security policies. Security requirements are initially described as a CC document named *Security Target*. The objective of our approach is to formalize the security target as a *Security Policy Model* (SPM) and to automatically compute tests from this model, following a model-based testing (MBT) approach. The generated tests are afterwards executed on the system. Due to the context that we consider (smart card applications), the project focused on policies relative to the control of commands execution.

MBT (3, 42) proceeds by computing tests from a formal model (FM) of the system to be tested, according to selection criteria. An example of a test selection criterion is, for instance, to exercise any operation of the system on the boundary values of its parameters. The formal model does not deal with implementation details, and is supposed to provide a reliable functional view of the implementation under test (IUT). As the tests have the same abstraction level as the FM, they have to be concretized before they can be executed on the IUT. This is obtained by writing a concretization layer. The verdict of the tests is obtained by comparing the results given by the IUT with the ones predicted by the FM, with respect to a given conformance relationship. Industrial studies have proven the efficiency of the method to detect faults in an implementation (see for example (16, 6)).

In our framework, a functional MBT campaign has already been performed, and so a functional model and a concretization layer are available. Nevertheless, functional tests appear to be insufficient to exercise the IUT through elaborated scenarios of attack, attempting to violate a security property. As aforementioned, we write an additional model, the SPM, to formalize the security target and we use this model to compute some additional tests using scenarios as selection criteria. The tests are then animated on the FM in order to bring them to the same abstraction level as the functional tests. This allows re-using the existing concretization layer in order to play security tests on the IUT and ensures the traceability of the tests generated by our approach with the original Common Criteria expression of the security requirements.

The original part of this paper is to present the full security model based testing process that as been adopted in the POSÉ project and how it has been successfully deployed on a real case study, the IAS platform. This work relies on previous works published by the partners. In (14, 13) a formal definition of the conformance of an application with respect to an access control policy has been proposed, taking into account a mapping relation allowing to relate models stated at different levels of abstraction. Hints on our MBT approach for security testing have been sketched in (33), with scenarios basically expressed as regular expressions. A language allowing to describe the scenarios in terms of actions to fire and states to reach has been defined in (26). In (27) the restriction of a B model to the executions satisfying a given scenario is presented, by means of a synchronous

[1] IAS is a *de-facto* standard issued by the GIXEL consortium
[2] see `http://www.rntl-pose.info`
[3] see `http://www.commoncriteriaportal.org/`

product of the B model with an automaton representing the scenario.

In Sec. 2 we describe the context (POSÉ project, B language) in which this work took place, as well as the case study IAS. We present in Sec. 3 the principle of the functional MBT campaign that was first performed. Our process for completing the functional tests is described in Sec. 4. Section 5 explains how to produce the SPM from the security target. The language that we have defined to describe the test patterns is presented in Sec. 6. The implementation of the test generation is discussed in Sec. 7, and our experimental results are given in Sec. 8. We finally compare our approach to related works and conclude in Sec. 9.

2 Context of the Work

This work has been performed in the context of the POSÉ project. The aim of this project was to propose a methodology for security testing, based on formal models, and compliant with Common Criteria methodology. The formal framework that has been retained is the B method for several reasons. First, previous experiments based on B models have already been led by the partners. Second, behind its modelling language, the B method supports a proof process for invariance properties and refinement. This aspect has been exploited in POSÉ in order to establish the theoretical framework of our approach (13). Finally a more anecdotal point is that the B method is one of the formal methods recommended by the CC evaluation methodology.

We first relate the project to the Common Criteria approach. Then we very succinctly present the B modelling language, that was used in this project. The IAS platform on which we have experimented our approach is also described in this section.

2.1 Common Criteria Approach of the POSÉ Project

The IAS based products are generally ordered by Public authorities (ID cards, e-passports or Health card) and then require to be Common Criteria certified. Therefore, the approach to be proposed by the project POSÉ should be as close as possible to the Common Criteria methodology.

Common Criteria (10) is an ISO standard (ISO 15408) for the security of Information Technology products that provides a set of assurances w.r.t. the evaluation of the security implemented by the product. Common Criteria provide confidence that the process of specification, development, implementation and evaluation has been conducted in a rigorous and standardized manner. The part of the system that has been identified to be evaluated and certified is called the target of evaluation (TOE). The Common Criteria approach is based on two kinds of assurances: in (9), part 2 is dedicated to security functional components, used to describe the security behavior of the system, and part 3 is dedicated to assurance components used to describe how the system implements this security behavior. The result is a level of confidence (called EAL for evaluation Assurance level) measuring the assurance that the product implements its security behavior.

The security functional components are relative to various aspects of security and various mechanisms enforcing security. For instance, the FDP class lists requirements relative to user data protection as access control policies, transfers between the TOE and the outside, protections against residual information, etc. The main assurance classes are relative to the design of the application to be evaluated (ADV class), how functional testing has to be conducted (ATE class) and vulnerability analysis (AVA class). For instance, aspects covered by the ATE class are how coverage analysis is conducted, the depth of the testing activities based on the knowledge of the conception (global interfaces, modular design, implementation level, etc.), the content of the documentation and, finally, tests developed by the evaluators themselves.

The POSÉ project focuses on access control policies for several reasons. First, in the domain of smart card applications, data protection is a central piece of security. Furthermore this aspect becomes more important when standardized platforms are concerned. For instance, the IAS standard which was the POSÉ case study, aims at receiving security data objects that carry out their own access control rules. Thus the correctness of this platform is crucial w.r.t. the security requirements of applications as electronic passports or health care cards.

The approach proposed in this paper can be seen as a contribution to the fulfillment of the ATE assurance requirements regarding the Common Criteria access control security components.

2.2 B Modelling Language

The B specification language was introduced by J.-R. Abrial in (1). It is defined as a notation based on first order logic and set theory. It allows the formal specification of open systems by means of state based models called *abstract machines*. More precisely, a B abstract machine defines an open specification of a system by an initialization state and a set of operations. The environment interacts with the system by invoking the operations. Intuitively, an operation has a precondition and modifies the internal state variables by a generalized substitution. Let S be a substitution. Let *out* be a list of output parameters and *in* be a list of input parameters. Let P be a precondition. An operation named o is defined in B as:

$$\mathit{out} \leftarrow o(\mathit{in}) \hat{=} \mathsf{PRE}\ P\ \mathsf{THEN}\ S\ \mathsf{END}.$$

Here are some generalized substitution examples: $x := \mathit{expr}$, $\mathsf{IF}\ Q\ \mathsf{THEN}\ S_1\ \mathsf{ELSE}\ S_2\ \mathsf{END}$, and $S_1 \parallel S_2$ where *expr* is an expression, Q a predicate, and S_1 and S_2 two generalized substitutions. Invariants relative to state variables can be stated and established, using proof obligations derived by the classical weakest precondition approach (15).

We give in this section the meaning of the B symbols and clauses that appear in the forthcoming examples of Fig. 5, Fig. 6 and Fig. 7. The clause SETS is used to declare some given sets or enumerated sets as in Z. Concrete constants and their properties are respectively declared under the clauses CONSTANTS and PROPERTIES of a B machine. The B notations appearing in the B expression examples have the following meaning:

- $r \in E \leftrightarrow F$ denotes the declaration of a relation between E and F; r^{-1} is its inverse and $r[d]$ is the relational image of a set d,
- $f \in E \rightarrow F$ denotes the declaration of a total function from the domain E to the range F; $f(x)$ denotes the image of x by f,
- $x \mapsto y$ denotes a pair of values of a function or a relation,
- $E \times F$ denotes the cartesian product of the sets E and F,
- $E - F$ denotes the subtraction of the set F to the set E.

Thanks to the proof capabilities of the B method, we have verified invariant properties on our formal models. We have not used the refinement capabilities of B.

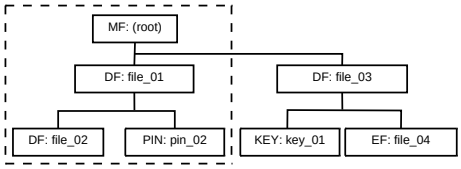


Figure 1: A sample IAS tree structure

2.3 IAS Premium Case Study

As stated before, the POSÉ project aims at producing conceptual, methodological and technical tools for the conformity validation of a system to its security policy, with smart card applications as a target domain. Experiments have been made with a real size industrial application, the IAS platform. We give all the technical details required to fully understand the examples that illustrate the following sections.

IAS stands for *Identification, Authentication and electronic Signature*. It is a standard for Smart Cards developed as a common platform for e-Administration in France, and specified (19) by GIXEL[4]. IAS provides identification, authentication and signature services to the other applications running on the card. Smart cards such as the french identity card, or the "Sesame Vitale 2" card[5] are expected to conform to IAS.

As a beginning, functional tests have been produced by a model based approach for a Gemalto implementation of IAS. For that, a functional model has been written in B by Smartesting and LIFC and an concretization layer have been written by Gemalto. We first describe below some aspects of the IAS case study and of the previous functional model. Then we focus on the access control security part of this platform, i.e. how APDU command executions are controlled. Finally, we describe how the SPM has been formalized, in order to be compatible with the Common Criteria and the POSÉ approach.

2.3.1 IAS File System Overview IAS conforms to the ISO 7816 standard, and can be implemented either as a JavaCard™ or as a standalone application. The file system of IAS is illustrated with an example in Fig. 1. Files in IAS are either *Elementary Files* (EF), or *Directory Files* (DF), such as `file_01` and `file_03` in Fig. 1. The file system is organized as a tree structure whose root is designed as MF (*Master File*). Directory files host *Security Data Objects* (SDO). SDO are objects of an application that contain highly sensible data such as PIN codes or cryptographic keys (see for example `pin_02` or `key_01` in Fig. 1), that can be used to restrict the access to some of the data of the application.

2.3.2 Some Data and Commands of IAS The services provided by the IAS platform can be invoked by means of various APDU[6] commands.

Some of these commands allow to create objects: for example `CREATE_FILE_DF` is for creating a directory file and `PUT_DATA_OBJ_PIN_CREATE` is for creating a PIN code, etc. Some other commands allow to navigate in the file hierarchy, such as `SELECT_FILE_DF_PARENT` or `SELECT_FILE_DF_CHILD`, or to change the life cycle state of files, such as `DEACTIVATE_FILE`, `ACTIVATE_FILE`, `TERMINATE_FILE`, or `DELETE_FILE`, ...

Finally, a group of commands allows to set attributes. For example `RESET_RETRY_COUNTER` is for resetting the try counter to its initial value, `CHANGE_REFERENCE_DATA` is for changing a PIN code value, `VERIFY` sets a validation flag to `true` or `false` depending on the success of an identification over a PIN code, etc.

As usual with APDU commands, the IAS platform responds by means of a *status word* (i.e. a codified number), which indicates whether the APDU command has been correctly executed or not. Otherwise, the status word returned by the APDU indicates the nature of the problem that prevented the command to end normally.

2.3.3 Functional Model of IAS The B model for IAS is 15,500 lines long. The complete IAS commands have been modelled as a set of 60 B *operations*. As the B model of IAS is intended to serve as an oracle for the tests, and for the operations to behave in an "APDU-like" manner, it has been written as a defensive formal specification. This means that invoking an operation with well-typed parameters is always allowed, as its pre-condition only checks the typing of the parameters. The operation responds by returning a value that models a status word, and that indicates if the operation should succeed or fail from a functional point of view. For example, trying to apply the operation `DEACTIVATE_FILE`[7] to a file that is already deactivated returns a status word value[8] of error meaning that the file is already deactivated.

2.3.4 Security Target for IAS Due to the complexity of the IAS platform, we have focused our security target on the control of the APDU commands execution, depending on the current files and security data objects. This target describes subjects, security attributes and rules, in conformance with the Common Criteria security functional components. Here is an example of an instance of the component ACF.1.2 (class FDP):

The operation `VERIFY(ref_sdo, PIN_code)` *can be executed by the subject* `TERMINAL` *if and only if* `ref_sdo` *currently denotes a well-defined PIN object, belonging to an activated file, and if the access conditions attached to the command VERIFY for this object are verified in the current state of the application.*

The access to an object by an operation (an APDU command) in IAS is protected by security rules based on the security attributes of the object. The access rules can possibly be expressed as a conjunction of elementary access conditions, such as *Never* (which is the rule by default, stating that the command can never access the object), *Always* (the command can always access the object), or *User* (user authentication: the user must be authenticated by means of a PIN code). Application of a given APDU command to an object can then depend on the state of some other SDOs: for instance the command `VERIFY` can successfully be applied on `pin_02` only if `pin_01` has been previously verified with success.

We give below some variables of the security model that we use in the remainder of this paper.

[4] `http://www.gixel.fr` - it is the trade association in France for electronic components industries

[5] A card with medical and personal data of the holder of the card

[6] *Application Protocol Data Unit* - it is the communication unit between a reader and a card; its structure conforms to the ISO 7816 standards

[7] More precisely, the operation in the model corresponding to the APDU command `DEACTIVATE_FILE`.

[8] More precisely, a number that corresponds to a status word of the functional specification.

Domains of values The set OBJ_ID denotes the set of references attached to objects manipulated by the IAS platform (PINs, files, SDOs, ...). The variables OBJ_list, DF_list, SDO_list, PIN_list respectively denote the subsets of objects, directory files, Security Data Objects and Personal Identification numbers of the current application (OBJ_list $\subseteq$ OBJ_ID, DF_list $\subseteq$ OBJ_list, SDO_list $\subseteq$ OBJ_list, PIN_list $\subseteq$ SDO_list).

The files and SDOs hierarchy The variable current_DF ($\in$ DF_list) stores the reference of the current selected DF. The variable PIN_2_dfParent ($\in$ PIN_list $\rightarrow$ DF_list) associates with a PIN reference the reference to the DF in which the PIN object is located. In the same way, the variable DF_2_dfParent ($\in$ DF_list $\rightarrow$ DF_list) associates with a DF reference df the reference to the DF where df is located[9]. These dependencies can be extended to the closure: for instance the variable DF_2_dfParent_closure ($\in$ DF_list $\leftrightarrow$ DF_list) associates a DF with all its antecedents, including itself.

Security dependencies The variable rule_2_obj ($\in$ OBJ_list $\leftrightarrow$ (SDO_list $\cup$ {always, never})) associates with an object reference the SDO that protects it. An object o that is always (resp. never) accessible is represented by ($o \mapsto$ always) (respectively ($o \mapsto$ never)). Notice that always and never are two particular SDO references that are not in the SDO list. The variable pin_authenticated_2_df ($\in$ PIN_list $\leftrightarrow$ DF_list) associates with a pin reference the DF references where the PIN object is authenticated.

Consider for example the data structure of Fig. 1. The pair pin_02 $\mapsto$ file_01 $\in$ PIN_2_dfParent means that the PIN object pin_02 is located in the DF file_01. The pair file_02 $\mapsto$ pin_02 $\in$ rule_2_obj means that the access to the DF file_02 is protected by a user authentication over the SDO pin_02. If pin_02 $\mapsto$ file_02 $\in$ pin_authenticated_2_df, then the access to the DF file_02 is authorized, otherwise it is forbidden.

3 Model-Based Testing using LTG

This section describes the principles of the Model Based Testing approach used to perform a functional test campaign on the IAS case study. This approach is implemented within the Leirios Test Generator (LTG) tool (23) from Smartesting[10], that takes a B model (1) as an input and automatically computes functional test cases based on a structural coverage of the operations of the model.

3.1 Model-Based Testing Process

The process for computing model-based functional tests is summarized by Fig. 2. The process is made of three steps.

- *Test Generation.* A set of *functional tests* is first statically computed from a functional model FM according to some selection criteria. In our case, the test generation is performed by LTG. The tool computes test targets from the model according to behavior, decision and data coverage criteria, as will be further detailed in Sec. 3.2 and Sec. 3.3.

[9] DF_2_dfParent is arbitrarily extended by the pair MF $\mapsto$ MF in the case of the master file.

[10] formerly Leirios Technologies; see http://www.smartesting.com

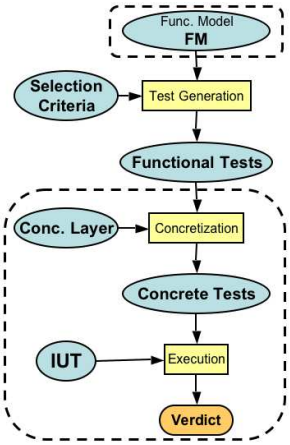


Figure 2: Functional Model-Based Test Generation Process

- *Concretization.* As the tests computed have the abstraction level of the functional model FM, they have to be transformed into *concrete tests*, at the level of the implementation. This step relies on the concretization layer which maps the operations and data of the model FM to the operations and data of the IUT, as further explained in Sec. 3.4.
- *Execution.* In this step the verdict of the execution of a concrete test is given by the comparison between the outputs predicted by the FM and the outputs given by the IUT (see Sec. 3.4).

The dashed circled parts in Fig. 2 shows what in the process will be reused to generate security tests, in addition to the functional ones. This will be performed by replacing the functional tests entering the lower dashed circled part by functional security tests (see Sec. 4.2).

The next three sections detail the composition of the test cases, the generation of test targets by application of coverage criteria and finally the concretization of test sequences into executable scripts.

3.2 Test Case Composition

The purpose of the model-based testing approach of LTG aims at activating the operations of the B model. More precisely, it focuses on a path-coverage of the control flow graph of the operations, in which each path is named *behavior*. Thus, each operation is covered according to its structure, by extracting its nested behaviors. Each behavior is composed of two elements: an activation condition and an effect that describes the evolution of the state variables if the activation condition is satisfied.

For each behavior, a test target is defined as its activation condition (decision). The tests covering the behavior will be constituted of a *preamble* that puts the system in a state that satisfies the activation condition of the behavior. To achieve that, a customized algorithm automatically explores the state space defined by the B model and finds one path from the initial state to a state verifying the target. LTG automatically selects the shortest preamble that reaches the test target. It is equipped with a constraint solver and proceeds by symbolic animation to valuate the parameters of a test sequence.

Apart from the preamble, a test is thus composed of the 4 elements, as shown in Fig. 3. The test *body* consists in the invocation of the tested operation with the adequate parameters so that the considered behavior is effectively activated. The *identification* phase is a set

of user-defined operation calls that are supposed to perform the observation of the system state. Their invocation when playing the test case on the IUT will make it possible to compare the concretely observed values w.r.t. their expected values computed from the model. Finally, a test case is ended by a *postamble* that is a (facultative) sequence of operations calls that resets the system to its initial state so as to chain the test cases.

3.3 Coverage Criteria for Test Target Generation

From the previous basic definition of a test target, based on the coverage of the structure of the operation, two other model coverage criteria can be applied, namely predicate and data coverage. These criteria are selected by the validation engineer.

Predicate coverage makes it possible to increase the test targets number, and possibly their error detection capabilities. This provides a mean for satisfying usual predicate coverage criteria such as: (i) *Decision Coverage* (DC) stating that the tests evaluate the decisions (each activation condition) at least once, (ii) *Condition/Decision Coverage* (C/DC) stating that each boolean atomic subexpression (called a condition) in a decision has been evaluated as true and as false, (iii) *Modified Decision/Condition Coverage* (MC/DC) stating that each condition can affect the result of its encompassing decision, or (iv) Multiple Condition Coverage (MCC) stating that the tests evaluate each possible combination of satisfying a predicate. In practice, different rewriting rules are applied on the disjunctive predicate form of the decisions, so as to refine the test targets in order to take this coverage criteria into account.

Data coverage makes it possible to indicate which of the test data have to be computed in order to instantiate the tests. The options, applied to operation parameters and/or state variables, propose a choice between: (i) all the possible values for a given variable/parameter that satisfy the test target, (ii) a smart instantiation that selects a single value for each test data, or (iii) a boundary values coverage, for numerical data, that will be instantiated to their extrema values (minimal and maximal values).

3.4 Executable Scripts and Verdicts

Once the abstract test cases have been computed, they have to be translated into the test bench syntax so as to be automatically executed on the IUT. This is the concretization step.

To achieve that, the validation engineer has to provide two correspondence tables. One of these tables maps the operation signatures of the B model to the control points of the test bench. The other one maps the abstract constant values of the B model to the internal data values of the IUT. By using an appropriate translator, a test script is automatically generated into the syntax of the test bench, ready to be run on the IUT. The correspondence tables and the translator implement the concretization layer.

Concretely, about 7000 tests were generated by LTG on the IAS case study. The average length of these tests in number of operation calls is approximately 5. Running these tests on the simulators of Gemalto took two days.

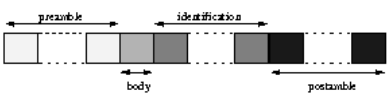


Figure 3: Composition of a LTG test case

For each test, the verdict is established by comparing the outputs of the system in response to inputs sent as successive operations. The concretization layer is in charge of delivering the verdict, by implementing functions that perform the comparison. In this context, the more observation operations (identification phase of Fig. 3) are available, the more accurate the verdict is.

Limitations This approach aims at ensuring that the behaviors described in the model also exist in the IUT, and their implementation conforms to the model. Nevertheless, this approach suffers from several limitations.

First, the preamble computed by LTG is systematically the shortest path from the initial state to the test target. As a consequence, possibly interesting scenarios for reaching this target may be missed. This implies a lack of variety in the composition of these preambles, that may possibly miss some errors. Second, the preamble computation is bounded in depth and/or time. This may prevent a test target to be automatically reached.

Third, the accuracy of the conformance verdict depends on the *testability* of the IUT, i.e., the number of observation points that are provided. When using LTG, one has to provide a systematic sequence of operations that can be used to observe the system state. Nevertheless, in smart card applets, the complexity of command calls (embedded within APDU buffers) prevents this solution to be easily set up, reducing the observation points to comparing the status words of the commands. Thus, the tests have to be built so as to, first, provoke an error, and, second, observe the resulting defect through an unexpected output status word.

Finally, the security requirements of the security target, for which the Common Criteria require testing evidences, may not easily be expressed in the model and related to the numerous functional tests.

To overcome these limitations, we develop, in the remainder of the paper, a security model-based testing approach that consists in using scenarios in order to ensure that security properties are correctly implemented. It is important to notice that a direct link can be established between a scenario and the security requirement it addresses.

4 Security Property Based Testing Process

We illustrate in this section the concepts of security property and test purpose and we detail the different steps of POSÉ security model based testing process.

4.1 Test Needs and Test Purposes

We see a *test purpose* as a mean to exercise the system in a particular situation w.r.t. a property. Based on its know-how, an experienced security engineer will imagine possible dangerous situations in which a property needs to be tested.

Consider for example an access control *security property* for IAS stating that to write inside a directory file, a given access condition has to be true, otherwise the writing is refused. Functional testing of this property with LTG activates two kinds of behaviors for the operation of writing: a success is reached by placing the system into a state where the access condition is true, whereas a failure is reached by placing the system into a state where it is false. Security engineers involved in the POSÉ project have expressed a need for testing such a

security property in other situations. For example, they have thought of the case when the access condition is true at an instant t and then becomes false at $t+\delta t$. The test need is to make sure that the previous true value for the access rule has no side effect at the time of writing.

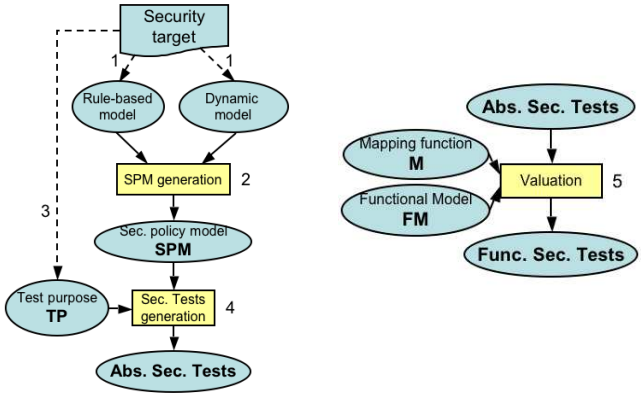
A test purpose corresponding to this test need is to: *reach a state where the access rule is true; perform the writing operation*[11]*; reach a state where the access rule is false; perform the writing operation.*

This example illustrates that one often wants to express a test purpose as both states to be reached and operations to perform. We have designed a language for expressing such test purposes by means of states and actions (see (26)). Once formalized, a test purpose is called a *test pattern*. In our process, we use test patterns as selection criteria to compute abstract security tests. An *abstract test* is a sequence of operation calls, with parameters computed according to a (functional or security) model. The abstract test also incorporates the expected result of each call and thus provides an oracle for the concrete test that will be executed on the IUT.

4.2 POSÉ Process for Generating Security Tests

Our process for generating security tests uses a security model as an oracle and test purposes as dynamic selection criteria to extract tests from this model. The idea is to reuse the dashed circled parts of the MBT process of Fig. 2, by replacing the functional tests with functional security tests. Figure 4 illustrates our approach: the abstract security test generation process is represented in Fig. 4(a), while the valuation of these tests into functional security tests is represented in Fig. 4(b). The process is made of five steps, numbered from 1 to 5 in Fig. 2.

1. *Formalization of the static and dynamic access control security rules*. In this step, the security engineer writes a semi-formal document from the security specifications, called the *security target*. It is written as a Common Criteria document. For access control requirements, he formalizes both the access control rules (the rule-based model) and how subjects, objects and security attributes can change (the dynamic model).
2. *Generation of a security policy model SPM*. This step takes as input the two previous models and automatically produces a behavioral SPM that abstracts the system in a level that only focuses on security aspects.
3. *Formalization of test patterns*. Based on their security expertise and their knowledge of the security policy model, the security engineers state some test patterns using a well-defined language, allowing to describe sequence of operations and conditions on variable values. The parameters are not instantiated in the operation calls.
4. *Generation of a set of abstract security tests*. This step takes as input a test pattern TP and an SPM to produce a set of *abstract security tests* in which security the parameters are instantiated.
5. *Valuation of the abstract security tests into functional security tests*. In this step, security tests are replayed on the FM in order to valuate the functional parameters and results. This provides *functional security tests*. During the valuation, the conformance between the functional and security models is checked w.r.t. a mapping function M that links the functional and security results. A non-conformance reveals an inconsistency between the SPM and the FM (see Sec. 7.2.3).



(a) Security Tests Generation (b) Tests Valuation

Figure 4: Security tests generation process

In this process the parameters are added to the operation calls at successive steps of the process. The parameters are completely abstracted at step 3 in the test patterns, then step 4 adds the security parameters to the calls and finally the functional parameters are computed at step 5. Some implementation parameters can also be added at concretization time (see Fig. 2), where the abstract operation calls are translated into concrete calls of the implementation (APDUs in the case of IAS).

Consider for example the operation `VERIFY` that performs an identification by means of a PIN code. A call to this operation would appear as `VERIFY` in a TP (step 3) and as `VERIFY(ref_sdo, PIN_code)` (see Sec. 2.3.4) in an abstract security test (step 4), with `ref_sdo` and `PIN_code` being some sdo and pin values. Then the call appears as `VERIFY(ref_sdo, PIN_code, IN_SM_Level, IN_Good_SM)` in a functional security test (step 5) because IAS operations have additional parameters in FM that indicate the level of secure messaging between the card and the terminal[12]. Finally, in the case of the `VERIFY` command, no implementation parameters are added at the concretization step, but the operation calls are translated into APDUs.

4.3 Discussion

This process completes the model-based generation of the former functional tests. We re-use the functional model FM, the concretization layer and the execution ground installation of the concrete tests. The security engineer has to design and to formalize the security policy model SPM and the test pattern TP. With this approach, the security engineer is only concerned by the security policy specification. He does not need to know the remainder of the functional specification. In addition, our process is consistent with the Common Criteria approach that explicitly distinguishes between the security and application models and imposes to relate these two models (our conformance relationship). From a practical point of view,

[11] this is for making sure that before the loss of the right to write, the writing operation was indeed possible, and not refused for any other reason.

[12] These parameters are not considered in our SPM that only focuses on the verification of access conditions.

the use of the security model for test generation allows us to master the combinatory: because the security model is more abstract than the functional model, the state space explored for the generation of security tests is smaller. For instance in the IAS case study the FM contains about 15,500 lines and 60 operations and the SPM only contains about 1,100 lines and 12 operations.

In the next sections we detail how the different steps of the POSÉ security model based testing process have been effectively implemented and applied on the IAS case study.

5 Security Policy Model Formalization

From this target of evaluation, a formal model of the access control part has been developed. This model can be assimilated to the assurance requirement SPM (for Security Policy Model) of the class ADV. A language based on the B method has been developed in order to formally specify and verify security access control models. This language is supported by a tool, named Meca (20, 14).

5.1 POSÉ Security Model of IAS

In the POSÉ approach, a security model contains the traditional rules part attached to access control policies but also a dynamic model describing how security attributes, subjects and objects can evolve. The *rule-based model* specifies subjects, objects, security attributes and operations whose execution is controlled.

We give in Fig. 5 a sub-part of this model allowing to describe conditions attached to the command `SELECT_FILE_DF_CHILD`. In this example the access control rule does not depend on a subject: `permission` is then a binary relationship between operations whose execution has to be controlled and the object on which the operation is applied. Here, the command `SELECT_FILE_DF_CHILD` can be invoked only if the current directory is activated and if the selected directory file `df_id` is effectively a sub-directory of the current directory file. Security attributes are here the life cycle of files (`DF_2_life_cycle_state`), the file and SDOs hierarchy (`PIN_2_dfParent` and `DF_2_dfParent`) and the state of the pin authentications (`pin_authenticated_2_df`). Variables `DF_2_life_cycle_state` is a total function from `DF_list` to the set {activated, deactivated, terminated}. Other variables have been already defined Sec. 2.3.

The *dynamic model* gives an abstract view of commands, focusing on the behavioral changes of security attributes. Figure 6 describes the part of the dynamic model relative to the command `SELECT_FILE_DF_CHILD`. This specification describes how the current directory file evolves as well as the set of authenticated pins. In particular, pins that are redefined in `df_id` lose their authenticated status.

5.2 Generation of the Security Model SPM

The inputs of the Meca tool are the rule-based and the dynamic models. Meca implements some verifications related to the consistency of these two models and produces a security model, obtained in weaving the two input models. The security model can be seen as a monitor that traps the execution requests and enforces the access control rules. In the context of the POSÉ project, this security model can be assimilated to the SPM assurance component of Common Criteria.

For each controlled operation, the security model contains a new operation corresponding either to the execution of the controlled operation, if security conditions are verified, or to a null execution. This operation returns a new result, `rs`, indicating if the execution has been authorized or not (`success`/`error`). For instance let $out \leftarrow op(i) \mathrel{\hat{=}} \text{PRE } P \text{ THEN } S \text{ END}$ be the definition of the operation op in the dynamic model. Let $C \Rightarrow (s \mapsto op \mapsto o) \in permission$ be the unique rule associated with the operation op (to simplify). The generated security model contains a new operation also named op (Fig. 7) describing how the execution of the operation op is controlled. Predicate `pre_typ` denotes the part of the precondition P relative to how input parameters i are typed. Variables `subject` and `object` contain the value of the current subject and object. These variables have to be defined in the dynamic model.

```
MACHINE IAS_RULE

SETS
   ...
   OPERATION = {SELECT_FILE_DF_CHILD, CREATE_FILE_DF, ... }

CONSTANTS
   ...

   /* Permission relationship */
   permission

PROPERTIES
   ...
   permission ∈ OPERATION ↔ DF_list ∧

   /* access rule relative to the command SELECT_FILE_DF_CHILD */

   ∀(df_id).(
     /* df_id denotes a child of the current directory file */
     DF_2_dfParent(df_id)=current_DF
     /* the current directory file is activated. */
     ∧ DF_2_life_cycle_state(current_DF) = {activated}
   ⟹ (SELECT_FILE_DF_CHILD ↦ df_id) ∈ permission)

   /* access rule relative to the command CREATE_FILE_DF */
   ∧ ...

END
```

Figure 5: Rule-based part of the security model of IAS

The security model can be seen as the specification of all implementations that conform to the rule-based and dynamic models. Intuitively an implementation for which all sequences of positive calls (associated to an effective execution of the operations) can also be played by the security model is conform. In particular the implementation can refuse more executions than the security model, for instance for functional reasons. A more formal definition of how functional models and security models can be linked will be given in Sec. 7.

Finally, the use of a formal method can be exploited to establish properties related to security aspects. As pointed out in Sec. 2.2, in the B method, invariant properties can be stated and proved. The first class of properties that has been proved on our security model is related to the file structure (a tree) and its consistency with the file life cycle states. A second class of properties is related to the consistency between authenticated pins and the current directory file: there cannot exist an authenticated pin that does not belong to a directory

```
SELECT_FILE_DF_CHILD(df_id) =
PRE    df_id ∈ DF_ID ∧ DF_2_dfParent(df_id)=current_DF
THEN
   current_DF := df_id /* update of the current df*/
   || LET pin_loosing_auth BE
     pin_loosing_auth = pin_authenticated_2_df⁻¹[{current_DF}]
     ∩ PIN_2_dfParent⁻¹[{df_id}]
   IN pin_authenticated_2_df := pin_authenticated_2_df ∪
     (pin_authenticated_2_df⁻¹[{current_DF}] - (pin_loosing_auth)×{df_id}
   END
END;
```

Figure 6: An operation of the dynamic part of security model of IAS

file between the root and the current directory file (see property (1)). Finally, another class of properties is related to the absence of cycle between security conditions attached to SDOs.

$$\forall(\texttt{pin\_id}, \texttt{df}).((\texttt{pin\_id} \mapsto \texttt{df} \in \texttt{pin\_authenticated\_2\_df}) \Rightarrow (\texttt{pin\_id} \in (\texttt{PIN\_2\_dfParent}^{-1}[\texttt{DF\_2\_dfParent\_closure}[\{\texttt{df}\}]))) \quad (1)$$

Establishing formal properties from the target of evaluation is one of the requirements in the higher level of assurance in the Common Criteria, used both to prove the consistency of the constructed formal models and to show the correspondence between the security target and the formal models. This allows giving further assurances on the security target. Furthermore, because the security model only focuses on some aspects of the system, security attributes, objects and subjects, it is generally small and abstract enough to support formal verifications.

6 Language for Test Patterns Description

In this section, we introduce the language that we have designed to formally express the tests purposes as test patterns (26).

We want the language to be as generic as possible w.r.t. the modelling language used to formalize the system. The language is structured as three different layers: *model*, *sequence*, and *test generation directive*.

The *model layer* is for describing the operation calls and the state properties in the terms of the SPM. This layer constitutes the interface between the SPM and the test patterns. The *sequence layer* is based on regular expressions and allows to describe the shape of test scenarios as sequences of operation calls leading to states that satisfy some state properties. The *test generation directive layer* is used to deal with combinatorial issues, by specifying some selection criteria intended to the test generation tool.

```
out, rs ← op (i) ≙
PRE pre_typ THEN /* typing of parameters */
   IF subject=s ∧ object=o ∧ C ∧ P
   THEN S || rs := success
   ELSE rs := error
   END
END
```

Figure 7: SPM general format of an operation

We give the syntax of each layer and then we give an example of a test pattern issued from the IAS study.

6.1 Syntax of the Model Layer

The syntax of the model layer is given in Fig. 8. The rule SP describes conditions as state predicates over the variables of the SPM. The rule OP allows to describe the operation calls, either by an operation name indicating which operation is called, or by the token $OP meaning that any operation is called or by $OP\{OPLIST} meaning that any operation is called but one from the list OPLIST.

```
OP      ::=   operation_name
          |   "$OP"
          |   "$OP \{"OPLIST"}"

OPLIST  ::=   operation_name
          |   operation_name","OPLIST
SP      ::=   state_predicate
```

Figure 8: Syntactic Rules for the Model Layer

6.2 Syntax of the Test Generation Directive Layer

This part of the language is given in Fig. 9. It allows to specify guidelines for the test generation step. We propose two kinds of directives aiming at reducing the search for instantiations of the test patterns.

```
CHOICE  ::=   "|" | "⊗"
OP1     ::=   OP | "["OP"]"
```

Figure 9: Syntactic Rules for the Test Generation Directive Layer

The rule CHOICE introduces two operators denoted as | and $\otimes$ for covering the branches of a choice. Let S_1 and S_2 be two test patterns. The pattern $\texttt{S}_1 \mid \texttt{S}_2$ specifies that the test generator must generate tests for both the pattern S_1 and the pattern S_2. $\texttt{S}_1 \otimes \texttt{S}_2$ specifies that the test generator must generate tests for either the pattern S_1 or the pattern S_2.

The rule OP1 tells the test generator to cover one of the behaviors of the operation OP. It is the default option. The test engineer can also ask for the coverage of all the behaviors of the operation by surrounding its call with brackets.

6.3 Syntax of the Sequence Layer

This part of the language is given in Fig. 10. The rule SEQ is for describing a sequence of operation calls as a regular expression.

```
SEQ     ::=   OP1 | "("SEQ")" | SEQ"⇝("SP")"
          |   SEQ "." SEQ
          |   SEQ REPEAT
          |   SEQ CHOICE SEQ

REPEAT  ::=   "*" | "+" | "?"
          |   "{"num"}" | "{"num",}" | "{,"num"}" | "{"num","num"}"
```

Figure 10: Syntactic Rules for the Sequence Layer

A step of a sequence is either an operation call as denoted by OP1 (see Fig. 9) or an operation call that leads to a state satisfying a state predicate, as denoted by SEQ ⇝(SP).

Sequences can be composed by the concatenation of two sequences, the repetition of a sequence or the choice between two sequences. We use the usual regular expression repetition operators (`*` for zero or many times, `+` for one or many times, `?` for zero or one time), augmented with bounded repetition operators ({`n`} means exactly n times, {`n,`} means at least n times, {`,m`} means at most m times, and {`n,m`} means between n and m times). Notice that using the operators `*` and `+` possibly define infinite sets of tests. To be of practical interest, they will have to be instantiated as explicit numbers some time in the process. Using these operators in a test pattern allows the engineer to postpone this question, as explained in Sec. 7.1.1.

6.4 Test Pattern Example

Here, we exhibit one of the test patterns (based on the language introduced above) written for the experimentation of our approach. The property to be tested is "*to access an object protected by a PIN code, the PIN must be authenticated*" and the test need is "*we want to test this property after all possible ways to lose an authentication over a PIN*".

The test pattern is given in two stages: the initialization stage and the core testing stage. Figure 11 presents the initialization stage of the test pattern in four steps, aiming at building the data structure required on the card to run the test (see Sec. 2.3.4 for the explanation of the variables used in this example). The purpose of the first step is to create a new DF (`file_01`). The second step aims at creating a PIN object (`pin_02`) into the DF `file_01` and to gain an authentication over it. The aim of the third step is to create the DF `file_02` into the DF `file_01`. Finally, the last step aims at setting the current DF to `file_01` in order to start the core of the test. The resulting data structure is the left part of the Fig. 1: the DF `file_02` is protected by the PIN `pin_02` for all commands.

We have given in Fig. 11 and Fig. 12 a label to each target state predicate expressed in the pattern, so we can refer to it afterwards. These labels appear as double slashed comments on the right hand of each predicate: // P1, // P2, etc.

```
CREATE_FILE_DF
    ⇝ (rule_2_obj[{file_01}] ={always} ∧ current_DF = file_01)          // P1
. PUT_DATA_OBJ_PIN_CREATE . VERIFY
    ⇝ (PIN_2_dfParent(pin_02) = file_01
        ∧ file_01 ∈ pin_authenticated_2_df[{pin_02}])                   // P2
. CREATE_FILE_DF
    ⇝ (rule_2_obj[{file_02}] = {pin_02} ∧ current_DF = file_02)         // P3
. SELECT_FILE_DF_PARENT
    ⇝ (current_DF = file_01)                                            // P4
```

Figure 11: Example of a test pattern — initialization step

Figure 12 shows the core testing stage, describing the test purpose of a successful authentication after all possible ways to lose an authentication. First, the pattern describes the five possible ways for losing the authentication over the PIN `pin_02` (for instance, a failure of the `VERIFY` command or a reset of the retry counter). The aim of the second step is to select the DF `file_02`, with the command `SELECT_FILE_DF_CHILD`. The final step of the test pattern describes the application of six commands, withe the current directory file being `file_02` in order to test the correctness of the access conditions.

```
. (VERIFY | CHANGE_REFERENCE_DATA
| (RESET . SELECT_FILE_DF_CHILD) | RESET_RETRY_COUNTER
| (SELECT_FILE_DF_PARENT . SELECT_FILE_DF_CHILD))
    ⇝(current_DF = file_01 ∧ file_01 ∉ pin_authenticated_2_df[{pin_02}])   // P5
. SELECT_FILE_DF_CHILD
    ⇝(current_DF = file_02)                                                 // P6
.[ CREATE_FILE_DF|DELETE_FILE | ACTIVATE_FILE | DEACTIVATE_FILE
| TERMINATE_FILE_DF | PUT_DATA_OBJ_PIN_CREATE ]
```

Figure 12: Example of a test pattern — execution step

7 From Security Test Patterns to Concrete Security Tests Execution

In this part we describe how concrete security tests are produced from test patterns, using the POSÉ tools suit. The process is in three steps. Section 7.1 presents the generation of the abstract security tests, by unfolding the test patterns and valuating the security parameters from the SPM. In Sec. 7.2, we describe the valuation of the functional parameters from the FM. We finally present the tests execution in Sec. 7.3.

We apply this test generation process to the test pattern example introduced in Fig. 11 and Fig. 12. We also present the practical and theoretical restrictions of the proposed approach.

7.1 Abstract Security Tests Generation

7.1.1 Unfolding of the Test Patterns Each test pattern has to be transformed into the set of test sequences it represents. To do so, we translate a test pattern into an automaton and then unfold it. This gives *test sequences* that are made of operation calls and states to reach. Notice that we bound the number of repetitions induced by the operators '*' and '+', in order to have a finite number of test sequences. The bounds can either be chosen by the validation engineer or set to a default value. Also notice that the "exclusive choice" operator $\otimes$, allowed by the language in the test generation directive layer, have not been implemented yet.

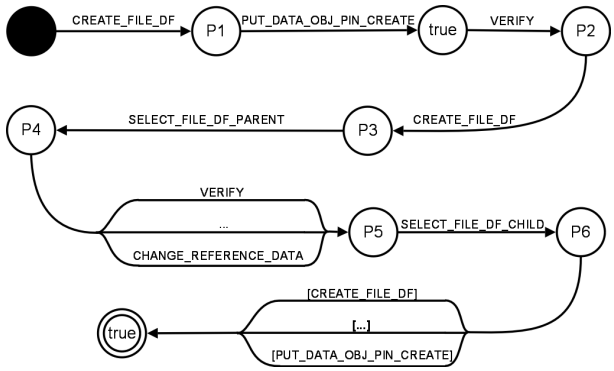


Figure 13: Automaton associated to the test pattern example

Figure 13 gives the automaton for the test pattern example given in Fig. 11 and Fig. 12 of Sec. 6.4. The edges are labelled by the operation names of the pattern and the labels in the vertices refer to the target state predicates Pi of Fig. 11 and Fig. 12. Predicate `true` denotes a state that is not constrained.

The unfolding of this pattern gives thirty test sequences, since five commands provoke the loss of authentication (transitions between

P4 and P5), and six different commands test the access control (transitions between P6 and the final state true).

7.1.2 Test Generation from the SPM In this step the SPM is used to compute parameter values for operations that match the constraints expressed in the test sequence. For example the call SELECT_FILE_DF_CHILD, between predicates P5 and P6, will be instantiated in SELECT_FILE_DF_CHILD(file_02) returning the value success.

We use LTG to compute abstract security tests. By default, LTG tries to cover every behavior of every operation of the model. By using a test pattern, we guide the test generation by forcing LTG to visit the successive target states and to call the successive operations given in the pattern. An extension of LTG has been developed for research purposes in POSÉ to take into account test selection guided by test pattern. This extension relies on the *preamble helper* mechanism of LTG, which allows to describe a desired test by the sequence of operations it activates. Technically, we have automatically added one "fictive" operation in the model per state to reach. Such an operation reaches the targeted state, provided it is possible to reach it from the current state.

Notice that the efficiency of the computation of the abstract security tests can be improved, by considering a restriction of the model to its executions matching the test pattern. We have shown in (27) how this can be obtained in B, by a synchronous product of the test pattern with the model. This synchronous product was not implemented in the POSÉ experimental prototype, as it was developed prior to (27).

The valuation of a test sequence may fail when the constraints are unsatisfiable due for example to an unreachable state. For instance the test pattern of Fig. 12 imposes that the execution of the command SELECT_FILE_DF_CHILD leads to the state current_DF = file_02 (P6) from the initial state current_DF = file_01 (P5). As specified in the dynamic model (Fig. 6), this command succeeds only if the following condition holds:

DF_2_dfParent(file_02) = file_01.

If the initial hierarchy does not fit this condition, LTG will fail and the test pattern will not produce any test. The valuation of a test sequence may also fail for a more pragmatic restriction, when the test generation tool fails at finding a valuation in some given time. Table 1 summarizes the possible results for the abstract security test generation step.

Result of the abstract security test generation step
OK : a set of abstract security tests is generated
KO : an unsatisfiable scenario is detected or some LTG limitations are encountered

Table 1: Abstract security test generation step

When the abstract security test generation fails, the current test sequence must be analyzed in order to detect the reason of this failure. In particular the test pattern associated to the faulty test sequence could be redefined.

7.2 Functional Security Tests Generation

In this section we explain how functional security tests are produced from abstract security tests.

7.2.1 Test Valuation from the FM Reusing the layer that concretizes the tests issued from the FM (see section 3.4) requires that the tests given by the SPM are brought to the same abstraction level as the FM. We obtain it by "replaying" these tests with the FM, using the LTG tool. For a given abstract security test, the input of LTG is the sequence of operation calls with their security parameter values and in omitting the output values. We expect that LTG produces some sequences with the same operation calls, enriched by values for functional parameters and output results. In the next sections we discuss how the functional security tests are shown to be in concordance with the SPM. Due to the fact that smart card applications are generally defensive, i.e. operations are always callable even if it terminates with an error status word, it is always theoretically possible to obtain a functional sequence replaying a security test. Table 2 summarizes the possible results for the functional security generation step.

Result of the functional security generation step
OK : a set of functional security tests is generated
KO : some LTG limitations are encountered

Table 2: Functional security test generation step

7.2.2 Mappings Between SPM and FM results By means of a conformance relation, we verify that the results returned by the SPM and the FM models are consistent. The conformance relation is based on a function, called a *mapping*, that associates to each status word returned by a given operation, an abstract security status belonging to the set {success, error}, as defined in section 5.2. Table 3 shows a part of the mapping function for the SELECT_FILE_DF_CHILD command.

	Status word	Security status
A success	6900	success
A security error: the current directory file is not activated	6985	error
A functional error: the secure messaging parameter is invalid	6982	error

Table 3: Mapping for the SELECT_FILE_DF_CHILD command

Status words mapped to success correspond to behaviors that are in conformance with the access control conditions and security attributes modifications described in the dynamic model. For instance for the VERIFY command the two behaviors corresponding to a right or erroneous pin value are both mapped with success, when the access control conditions hold. Status words corresponding to a violation of a part of the access control conditions are mandatory mapped to error and security attributes can not be modified, in any way. In (14, 13) a finer form of mapping has been proposed, allowing to distinguish authorized behaviors as in a VERIFY command that succeeds or fails. Nevertheless, such forms of powerful mappings are in general non-deterministic and have not been used in our case study, in order to master the complexity of mapping expression.

7.2.3 Functional Security Tests Conformance with respect to the SPM In this step we verify that the functional security tests, produced by LTG using the FM, conform to the SPM. A semantic conformance relationship between a functional and a security model

has been defined in (14). For a given mapping function M, all sequences of the FM, in which status word values sw_i are replaced by $M(sw_i)$, should be accepted by the SPM after elimination of functional parameters and calls that are mapped to `error`. By this definition, all sequence of successful calls accepted by the FM should also be accepted by the SPM. On the contrary, the FM should be more restrictive, for example for functional reasons.

Table 4 gives the conformance verdict, w.r.t. a given mapping M. In particular we exploit the fact that the SPM is a deterministic model, as imposed by LTG. Let $\sigma_s \hat{=} < r_1, \ldots, r_n >$ and $\sigma_f \hat{=} < sw_1, \ldots, sw_n >$ be the two sequences of output respectively produced by the SPM and the FM, for a given sequence of operation calls with the same security parameter values. We compute the greatest index k such that $r_i = M(sw_i)$ for $i \in 1..k$.

Condition	Conformance verdict
$k = n$	σ_f conforms to SPM
$k < n \wedge r_{k+1} = error$ $\wedge M(sw_{k+1}) = success$	σ_f does not conform to SPM
$k < n \wedge r_{k+1} = success$ $\wedge M(sw_{k+1}) = error$	inconclusive

Table 4: Conformance verdict

As summarized in table 4, if $k = n$, that means that any operation call returns the same status word. In other words, if σ_s detects a security violation then σ_f must also detect it. If it is not the case ($k < n$), and due to the fact that the SPM is a deterministic model, an inconsistency is detected between the two models. On the contrary, if σ_s succeeds while σ_f fails, then the FM could be more restrictive than the SPM. In this case we have to establish wether σ_f is in conformance with the SPM, by verifying if the subsequence of successful calls are accepted by this model, as defined in the conformance relationship (14). This verification can be made by playing this sequence on the SPM, with the help of LTG.

Then we have developed a script, written in Perl, that verifies the conformance of a functional security test produced by LTG according to table 4. This script is based on a small language dedicated to the definition of mappings.

Finally, an important question is the relevance of the functional security tests produced by LTG. For instance, a test that systematically chooses functional values producing an error is fully conform, but not necessary a good test. Then LTG must be guided in order to target tests as relevant as possible. The strategy that has been adopted is the following one: when a success is expected then the search is guided by any status words mapped to `success`. When this search fails, we are looking for an error. On the contrary, when an error is expected we search both a call producing a status word mapped to `success` and a call producing a status word mapped to `error`. This way, if there exists an inconsistency between the SPM and the FM, it will be detected.

To summarize, functional tests produced from abstract security tests are in conformance with the SPM through a relationship that admits more restrictive implementations. The correctness of the conformance relationship, and its application to our security model based testing approach, strongly depends on the relevance of the mapping function M. Due to the fact that the models that are considered are formal, the correctness of the mapping can be verified in terms of refinement (see (13) for a formal definition). Due to the proximity of the structure of the two models, the mapping of the IAS case study has been validated by a review process.

7.3 Tests Execution

The fully valuated test sequences are finally concretized by means of the concretization layer, and executed on the IUT.

Practically, this is performed at Gemalto through the EVA (Easy Validation Application) environment. EVA is the validation data base environment of Gemalto. It uses the Visual Basic 6 language. It is based on a proprietary tool, used to write validation script tests and to execute them on different targets: simulator, emulator or smartcards and with different smart card readers. This environment allows to use the same script on the different types of simulator and cards, thereby improving the validation in terms of time process and debug. Figure 14 shows a screenshot of EVA. The "TreeViewer" panel shows the card image, while the "EVA View" panel displays the result of the execution of the tests. The down part of the screenshot shows the test code, while a list of available sets can be seen on the left hand of the screenshot.

The (security or not) functional tests are run on the IUT by EVA, *via* a dedicated interface (the concretization layer) relying on the functional model. The concretization layer had been previously developed for the non security functional validation tests. For efficiency reasons, the constraint was to use the same concretization layer in order to avoid additional developments.

This concretization layer implements the definition and the translation of each operation call of the test by:

1. providing concrete values for the parameters of the commands and encapsulating them in specific formats,
2. initializing the secret data (PIN values, key values) and storing it in the concretization layer, to be used for comparison (because there is no mean to retrieve those values from the card),
3. translating the command into a format understood by the card (i.e. APDU format (22)),
4. sending the command to the card,
5. receiving the data response from the card,
6. verifying the results, i.e. verifying that the data received from the card equals the one expected by the test design. This includes side channels verification, e.g. no secret value is returned from the card.

The verdict is thus given facing the results of the IUT to the ones predicted by the oracle, namely the FM. The mapping between both results is a bijection as the functional model returns the same status words as the implementation (6900, 6985, etc.). If the results differ, this indicates that there is a problem, either in the IUT or in the model. The problem is reported to the validation engineer for analysis.

As the security functional tests are computed from the expressions of security requirements, the traceability of a test to an original requirement is easy to ensure. Every test can, for example, include a tag that refer to the requirement from which it is issued. Consequently, a bad verdict can easily be related to an original security requirement. This facilitates the human analysis of a problem discovered by a test.

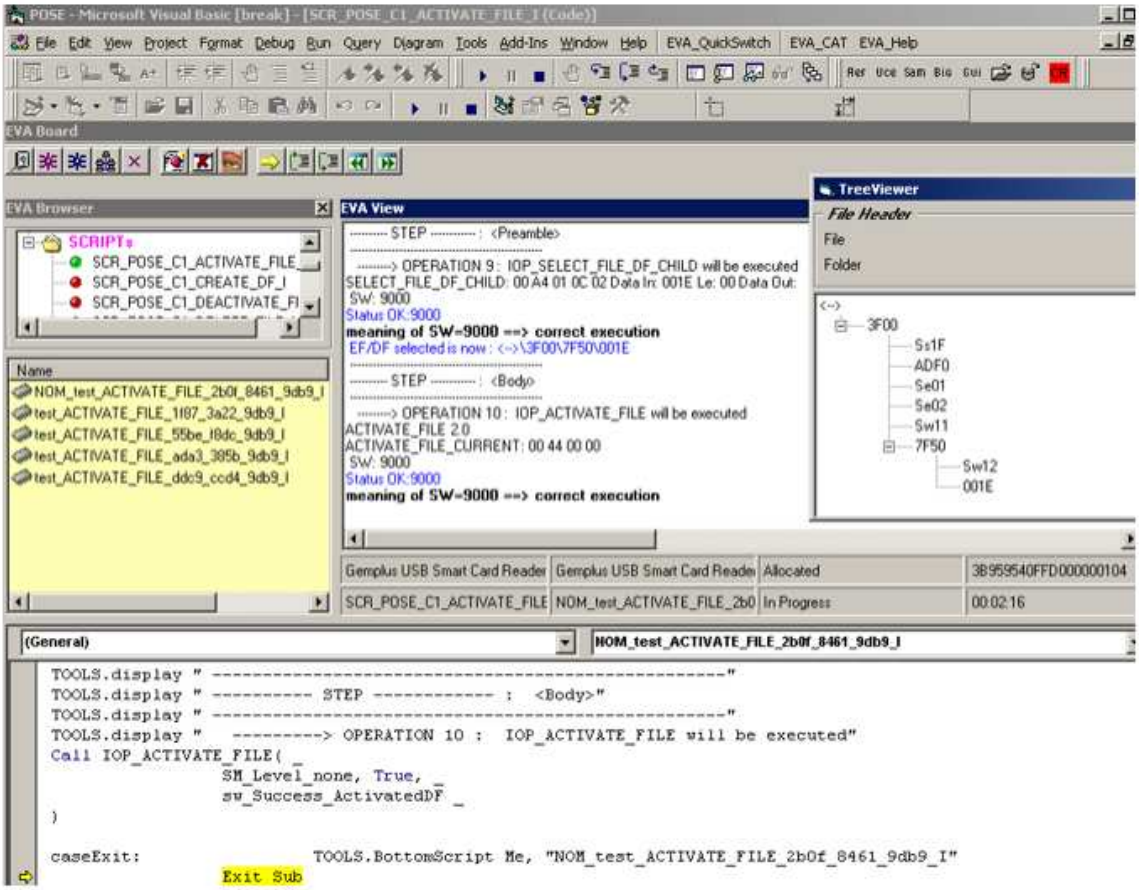


Figure 14: A screenshot of EVA

8 Experimental Results

We describe in this section the three test patterns that we have experimented on the IAS platform, and the test generation based on these test patterns with LTG. We also present the concretization and execution steps in an industrial process, and comment the results obtained on the IAS implementation.

8.1 Three Test Patterns

For each of the test pattern that have been experimented, we informally give the property from which it is issued, the test need associated to the property, and the shape of the test pattern itself.

The first test pattern that we have experimented is the one depicted in Fig. 11 and Fig. 12 (see Sec. 6.4). The property to be tested is that the access to an object protected by a PIN code requires to gain an authentication over the PIN code. Functional tests will exercise this property in a case where the authentication is gained, and in a case where it is not. But they don't take into account if a PIN was previously authenticated, and that the authentication has been lost. So the test need is to exercise the access control mechanism in the case of a loss of authentication, in all possible ways, following a previous gain. The pattern proceeds by targeting a state where the authentication is gained, accessing the object, targeting by all possible operations a state where the authentication is loss, and accessing again the object.

As already stated, the unfolding of this pattern gave 30 sequences. From these sequences, we have obtained 35 abstract security tests from the SPM. This is due to the fact that there were multiple possible valuations for the parameters of the last operation of some sequences. The functional valuation of these abstract security tests gave 35 functional security tests.

The second test pattern aims at testing the access control based on a PIN authentication for various locations of different PIN objects with the same name in the file structure. In IAS, each PIN is a file saved under a directory. The location of the PIN w.r.t. the current DF matters for an authentication gained over it. For example, accessing the DF parent of the current DF leads to a loss of the authentication. Thus, the property that we want to test is the same as before: the access to an object protected by a PIN code requires to be authenticated over this PIN code. But here, the test need is to exercise the property with several PIN objects saved under multiple directories (e.g. the current directory and his child) when these PIN objects are homonyms. Indeed, two distinct objects can share the same local name (they are homonyms) if they are located in two distinct DF.

Furthermore, the test need also aims at exploring the different combinations of the authentication states of these PINs. These test needs are addressed by a pattern targeting various situations to reach before applying the access commands. For example, it describes by state predicates the directory selected as the current directory, and

Model	Number of lines	Number of operations	Number of variables
FM	15,500	60	150
rule based model	200	11	—
dynamic model	1000	12	20
SPM	1100	12	20

Table 5: Size of the different models

Test pattern (see Sec. 8.1)	Number of seq. in TP	Number of func. sec. tests	Maximum number of op. calls per test	Minimum number of op. calls per test	Average number of op. calls per test
1	30	35	10	9	9.4
2	48	66	11	8	9.5
3	68	82	8	5	6.9

Table 6: Experimental results about test generation

which PIN is authenticated or not. Some constraints over the commands sequencing (expressed by concatenations and choices over command names), enable to reduce the possible paths to reach these state targets. From this test pattern, we have generated a total amount of 66 functional security tests.

The authentication gained over a PIN not only depends on the location of the PIN, but also on the life cycle state of the DF where a command protected by the PIN is applied. Thus, the third test pattern aims at testing some situations where the life cycle state of the directory is not always *activated*. In addition to the property already seen in the two previous test patterns, we exercise the property that when a command is executed in a directory, this one needs to be in an appropriate life cycle state. The functional test cases test such situations in a static way, with a life cycle state of the directory that does not change during the test sequence.

So, the test need used in this pattern is to change the life cycle state of the directory one or several time(s) during the test sequence (e.g. just before applying the command, or before gaining an authentication over the PIN or before a reset of the card, as if the card was removed from the terminal and inserted again). The pattern combines these life cycle state changes with the different authentication states of the PIN protecting the access to the directory. 82 functional security tests have been generated from this pattern.

8.2 Test Generation

Every test pattern gives several abstract security tests. For each abstract security test, we compute only one valuation of the functional parameters, so one functional security parameter is computed per abstract security test. In our experiment, the three test patterns gave a total amount of 183 tests. This number seems small in comparison to the 7000 tests generated for the non-security functional test campaign. But it is necessary to consider that these three test patterns did not intend to address the whole system. Instead, they focused on selected properties and test needs, regarding access control mechanisms. Furthermore, each of these 183 tests is complementary to the non-security functional tests previously generated. This can be seen from tables 5 and 6. In table 5, we give the size (number of lines, operations and variables) of each model that was used for the test generation. Table 6 presents the experimental results (number of tests generated, and length of these tests in number of operations) about the test generation using the three patterns presented in Sec. 8.1. In comparison, the average length of the non-security functional tests is 5, which is lower than the average length (8.26) of the security tests.

For each test pattern, the complete test generation (first from the SPM and then from the FM) took about two or three hours. It may seem a little bit long, but our main objective was the concrete use of the developed approach in the industrial environment to test real products. Nevertheless, our implementation is a research prototype whose efficiency could be improved in a second phase.

8.3 Discussion About the Experimentation

We propose in this part to give some experience returns with a point of view of industrial partners.

8.3.1 Functional and Security Validation For the **functional validation**, two ways have been deployed to validate the IAS implementation. For the first, we used the model-based approach, with automatic generation of tests and for the second, we used the traditional approach where the test scripts are developed manually. The first approach has generated more than 7000 tests. The execution time on the smart card was approximately 2 full days. The manual approach has delivered nearly 500 tests, which were mainly designed to complement the automatic generated tests. They were focused on parts that the modeling could not take into account, e.g. some limit cases, *stress* cases where the test stresses a specific feature (quality of the random value, memories cell values,...). The corresponding execution time was nearly the same than the automatic phase, due to the time allocated to the stress tests.

The **security validation** takes advantage of the functional validation as it was based on the same functional model describing the behavior and in particular the tests oracles. All the generated security tests were correctly executed on the target. As already stated, the three families have delivered 183 tests, executed in one hour on the target. Although no problems were detected in the IAS implementation, the approach has improved the confidence in that implementation. This is crucial for the certification of products embedding the IAS application because this step is part of the testing task that will be done by the evaluator. Indeed, although the approach has covered only a subset of the security properties of the IAS (only the access control on PIN objects), the global concept has been validated in the industrial framework.

Additionally, one test issued from the security model has raised a non conformance between the security model and the functional one. This was due to distinct interpretations in the two models of an imprecise point of the specification. The previous (non-security) functional test campaign alone would not have pointed out this specification ambiguity.

8.3.2 Coverage For the functional validation, the coverage of a specific behavior was done manually, using the parametrization features to force the tool to cover a specific path in the model. The approach used for the security validation, with test patterns description and their unfolding allows a systematic coverage, that is larger.

Let us take the example of homonymy (the second test pattern example in Sec. 8.1): one could have several SDOs with the same reference but at different levels within the file structure. But the ac-

cess conditions relying on a SDO PIN in a specific DF are different from another SDO PIN with same reference but inside a different DF. The non security functional validation campaign, though manually parameterized to cover this point, only generated five tests. In comparison, 66 tests were obtained to exercise this security point with the security validation approach. Indeed, the security tests were designed using the know-how of the security experts and the testing experience of the validation engineer. Having a systematic mean to design the security tests is the main advantage of this approach.

8.3.3 Conclusion From the industrial point of view, the main advantages of the POSE methodology are the following:

- cost reduction of the validation process: capitalizing on the developments required by the functional validation, i.e. functional model and interface of the concretization layer.
- time improvement of the validation process: the security validation step is no longer a "subtask" of the validation phase but an independent phase. This separation allows for a significant saving time in the validation of the product because the security properties are clearly identified and their test is reproducible.
- quality improvement of the validation process: complete chain that provides a traceability between the abstract property and the corresponding test suites. This traceability is critical first for the certification of the product and secondly for the security validation of several products based on the same specification.

9 Conclusion

We have presented in this paper a security model based testing approach, that has been successfully deployed on a real size industrial application, the IAS platform for smart cards. To conclude we discuss about the proposed security model based testing approach and the theoretical contributions of the POSÉ project .

9.1 The POSÉ security model based testing approach

The method makes use of already existing material, written for model based functional testing: the functional model and the concretization layer. An additional dedicated model is written for modelling the security rules. Abstract security tests are obtained by using test purposes as patterns for extracting relevant tests from the security model. These tests are then automatically replayed on the functional model in order to bring them to the abstraction level required to interact with the implementation, through the concretization layer. The method easily ensures the traceability of the tests generated to the original test patterns, since the tests are computed from these patterns. Also, with the mechanism for functional test generation offered by LTG, we exactly know which behaviors of the operations have been covered.

From a methodological point of view, the distinction between security models and functional models effectively corresponds to distinct stages in the life cycle of secured applications. A security model is written by security engineers and exploited by certification evaluators, independently of a given implementation. This model focuses on some particular aspects and is generally small enough to be successfully exploited for validation and verification. Furthermore several security models can be written, corresponding to several aspects of security, mastering in that the complexity of the validation and verification process. From a practical point of view, the proposed model based testing approach, and its tools suite, has been proved to be realistic even for a sizeable application. The difficulty of the test generation part is in finding, with the help of LTG, some suitable instantiations for parameters. Due to the fact that the security model is small and abstract enough, the use of LTG with the SPM generally succeeds. On the other hand, search for suitable instantiations for functional parameters is strongly guided, because we reuse sequences generated at the first level. Finally, we gain some confidences in our formal models because we test the FM against the SPM.

On the contrary, the proposed approach is time and cost consuming because two models have to be written. In the general case, this effort is disproportionate. But when Common Criteria certifications are targeted, like often for smart cards and especially for the IAS on which several kinds of products (ID card, e-passport or health card) are built, formal models are a central piece for reusable methodology. In particular a new certification must be conducted as soon as a new implementation or a new hardware support is used. In our approach, security and functional models, as well as the proposed methodology, can be reused to be adapted to new versions of the IAS standard or new implementations. Furthermore, the IAS case study is a generic platform dedicated to the development of proper applications, that also have to be certified. An application deployed on the IAS platform firstly consists in a personalization specifying a particular set of PINs, keys, SDOs and files and their security dependencies. A security model attached to such an application can be defined in terms of a specialization of the generic IAS security model or as an independent model that can be confronted to this generic model, instantiated by the given personalization. Finally, the proposed approach can be used in a light manner, only in using a security model. In this case the concretization layer is in charge of bringing the gap between the security abstraction level and the implementation.

Theoretical contributions of the project POSÉ are the proposition of the MECA form of access control security models in concordance with the Common Criteria requirements, a formal definition of a conformance relationship based on a notion of mapping relating models stated at different levels of abstraction and a language of test patterns allowing to express security tests requirements.

9.2 Security model and conformance relationship

There are several sorts of formalisms dedicated to access control specifications. Usual formalizations are based on rules (28, 4, 7, 37) and mainly focus on access control conditions. On the other hand, security automata (38) describe behaviors resulting both of access control conditions and some operational specification. In concordance with the Common Criteria approach, the Meca approach distinguishes these two parts, through the rule-based and the dynamic models. In this way a traceability is established between the informal security policies described in the security target and the SPM (the rule-based part corresponds to the User Data protection class of Common Criteria and the dynamic part to the Security Management class). Finally these two models are woven to produce a behavioral model that can be assimilated to a security model. Such automata can be obtained for instance with the help of tools (12, 31).

The B method has already been used as a support for access control policies (5, 39). In (5), the authors propose a form of modeling attached to Or-BAC access control, including permissions and prohibitions, and characterize behaviors which conform to a given access control policy. Our approach can be seen as an extension of (5, 39)

taking into account the conformance of an application with respect to a security model, with the help of a mapping correspondence between models stated at different levels of abstraction. In (32), the authors use Labeled Transition Systems (LTS) to describe test purposes from Or-BAC rules specifying access control. They act as an oracle for the test execution, based on the ioco conformance relation (41). Our approach is similar, since they both rely on trace inclusions, and our notion of stuttering is close to the notion of quiescence. Nevertheless, our relation is not exclusively destined to be used as a test oracle. Indeed, by giving a formal definition of our relation, as done in (13), it would be possible to prove properties on the implementation w.r.t. the abstract security model. In this way we are closer to the Common Criteria approach that requires to establish correspondences between the SPM and some application models, depending on the targeted assurance level.

9.3 Tests patterns and security tests

Many other works use temporal logic properties or test purposes as selection criteria to extract tests from a model. By exploiting its ability to produce counter-examples, a model-checker can be used to compute tests from temporal properties w.r.t. a formal model (18, 34, 2, 21, 40). Linear temporal logic model-checking uses cycles search algorithms to compute tests from explicit transition systems, while we use artificial intelligence constraint solving techniques to compute tests directly from B models. The TGV approach (17, 24), and works from the Vertecs project[13] (36, 35, 11) use explicit test purposes to extract tests from specifications, both given as Input/Output Symbolic Transition Systems (IOSTS). Our approach differs since our test purposes mix operation calls and target states description. In (2) and (40), the test purposes are linear temporal logic formulas describing state sequences. In (25), (36) and (30), the test purposes are sequences of operation calls expressed either by IOSTS or by regular expressions. Moreover in (30), the symbolic tests are generated independently from a behavioral model, which leads to a combinatorial explosion of the number of tests. Also, our approach is methodologically different because our intention is to use abstract models. Finally, the language we use to express the test purposes can be instantiated, thanks to the model level, with various modelling languages. We have performed experiments with formal specifications written in B and in UML/OCL. The language is intended to be easily manipulated by the security engineers.

In (29), the authors show how tests dedicated to exercise a given security policy can be obtained by reusing functional tests. In comparison, we do not reuse the existing functional tests, but we augment them with security tests, independent from the functional ones. What we reuse is the existing functional material (i.e. the functional model and the concretization layer). They mention two types of strategies for generating security tests w.r.t. functional tests. Our approach fits in their independent strategy. And as a difference with security policies specified through OrBAC-like models, our SPM is not a list of static rules, but models also dynamic operational modifications of the security attributes.

9.4 Further Works

In a previous work (33), we have foreseen the possibility for the test purposes to be automatically computed, by modelling the test needs as syntactic transformation rules that transform regular expressions. Integrating a test need to a security property could then be obtained by transforming the formalization of the security property. The tool Tobias (30), that unfolds in a combinatorial way tests expressed as regular expressions, could be used to unfold our test patterns.

We are currently working at identifying and writing such transformation rules, based on the IAS case study. This work needs to be developed by studying many other case studies, in order to produce rules sufficiently generic to be applicable to a variety of examples. Rules could also be automatically deduced from the syntactic expression of a property, as suggested by (8) for properties expressed in JTPL, a temporal logic for JML.

Also, rules could be expressed for transforming other formalisms than regular expressions. In particular, we think of rules that could transform automata. They could be applied to security properties expressed as temporal logic formulas, as well as regular expressions.

Another follow-up to this work would be to explore the possibility to use several smaller security models, instead of just one that contain all the security features. These models could be very easy to write as they would focus on a limited set of security features at a time, the ones concerned by some particular test purposes. The tests computed from any of these models could still be brought to the abstraction level of the functional model by replaying them on it.

[13] http://www.irisa.fr/vertecs/

Acknowledgments

This work is funded by the ANR (*Agence Nationale de la Recherche*), in France. The POSE project is a RNTL (*Réseau National de recherche et d'innovation en Technologies Logicielles*) project, recorded under the number ANR-05-RNTL-01001.

References

[1] Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] P. Amman, W. Ding, and D. Xu. Using a model checker to test safety properties. In *ICECCS'01, 7-th Int. Conf. on Engineering of Complex Computer Systems*, page 212, Washington, DC, USA, 2001. IEEE Computer Society.

[3] B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, New York, USA, 1995.

[4] D. Elliot Bell and Leonard J. LaPadula. Secure computer systems: A mathematical model. Technical report 2547, vol 1, MITRE, 1973.

[5] N. Benaissa, D. Cansell, and D. Mery. Integration of Security Policy into System Modeling. In J. Julliand and O. Kouchnarenko, editors, *B 2007: Formal Specification ans Development in B*, volume 4355 of *LNCS*. Springer, 2007.

[6] E. Bernard, B. Legeard, X. Luck, and F. Peureux. Generation of test sequences from formal specifications: GSM 11-11 standard case study. *Software: Practice and Experience*, 34(10):915–948, 2004.

[7] K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts, April 1977.

[8] F. Bouquet, F. Dadeau, J. Groslambert, and J. Julliand. Safety property driven test generation from JML specifications. In

FATES/RV'06, 1st Int. WS on Formal Approaches to Testing and Runtime Verification, volume 4262 of *LNCS*, pages 225–239, Seattle, WA, USA, August 2006. Springer.

[9] Common Criteria for Information Technology Security Evaluation, Part 2: Security functional components, and Part 3: Security assurance components. Technical Report CCMB-2006-09-002/CCMB-2006-09-003, version 3.1, September 2006.

[10] Common Criteria for Information Technology Security Evaluation, version 3.1. Technical Report CCMB-2006-09-001, sept 2006.

[11] C. Constant, T. Jéron, H. Marchand, and V. Rusu. Integrating formal verification and conformance testing for reactive systems. *IEEE Transactions on Software Engineering*, 33(8):558–574, August 2007.

[12] M-L. Potet D. Bert and N. Stouls. GeneSyst: a Tool to Reason about Behavioral Aspects of B Event Specifications. Application to Security Properties. In H. Treharne, S. King, M. C. Henson, and S. A. Schneider, editors, *ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users*, volume 3455 of *Lecture Notes in Computer Science*, pages 299–318. Springer-Verlag, 2005.

[13] F. Dadeau, J. Lamboley, T. Moutet, and M-L Potet. A Verifiable Conformance Relationship between Smart Card Applets and Security Models. In *ABZ'08*, volume 5115 of *LNCS*, London, UK, September 2008. Springer.

[14] F. Dadeau, M-L Potet, and R. Tissot. A B Formal Framework for Security Developments in the Domain of Smart Card Applications. In *SEC 2008: 23th International Information Security Conference*, IFIP proceedings. Springer, 2008.

[15] E.W. Dijkstra. *A discipline of Programming*. Prentice-Hall, 1976.

[16] E. Farchi, A. Hartman, and S. S. Pinter. Using a model-based test generator to test for standard conformance. *IBM Systems Journal*, 41(1):89–110, 2002.

[17] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on the fly verification techniques for the generation of test suites. In *CAV'96, Conference on Computer Aided Verification*, 1996.

[18] 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.

[19] GIXEL. *Common IAS Platform for eAdministration*, Technical Specifications, 1.01 Premium edition, 2004. http://www.gixel.fr.

[20] A. Haddad. Meca: a tool for access control models. In J. Julliand and O. Kouchnarenko, editors, *B 2007: Formal Specification ans Development in B*, volume 4355 of *LNCS*. Springer, 2007.

[21] H. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *TACAS'02, 8th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 327–341, London, UK, 2002. Springer-Verlag.

[22] European standard - identification cards - integrated circuit(s) card with contacts - electronic signal and transmission protocols. Technical Report ISO/CEI 7816-3, Comité européen de Normalisation. En27816-3, 1992.

[23] E. Jaffuel and B. Legeard. LEIRIOS Test Generator: Automated test generation from B models. In *B'2007, the 7th Int. B Conference - Industrial Tool Session*, volume 4355 of *LNCS*, pages 277–280, Besancon, France, January 2007. Springer.

[24] 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. *Software Tools for Technology Transfert*, 7(1), 2005.

[25] T. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva. Symbolic test selection based on approximate analysis. In *TACAS'05*, volume 3440 of *LNCS*, pages 349–364. Springer, 2005.

[26] J. Julliand, P.-A. Masson, and R. Tissot. Generating security tests in addition to functional tests. In *AST'08, 3rd Int. workshop on Automation of Software Test*, pages 41–44, Leipzig, Germany, May 2008. ACM Press.

[27] J. Julliand, P.-A. Masson, and R. Tissot. Generating tests from B specifications and test purposes. In *ABZ'2008, Int. Conf. on ASM, B and Z*, volume 5238 of *LNCS*, pages 139–152, London, UK, September 2008. Springer.

[28] Butler W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, 1974.

[29] Y. Le Traon, T. Mouelhi, and B. Baudry. Testing security policies: Going beyond functional testing. In *ISSRE'07, 18th IEEE Int. Symp. on Software Reliability*, pages 93–102, 2007.

[30] Y. Ledru, F. Dadeau, L. Du Bousquet, S. Ville, and E. Rose. Mastering combinatorial explosion with the TOBIAS-2 test generator. In *ASE'07: Procs of the 22nd IEEE/ACM int. conf. on Automated Software Engineering*, pages 535–536, New York, NY, USA, 2007. Acm.

[31] Michael Leuschel and Michael J. Butler. Prob: an automated analysis toolset for the b method. *STTT*, 10(2):185–203, 2008.

[32] K. Li, L. Mounier, and R. Groz. Test Generation from Security Policies Specified in Or-BAC. In *COMPSAC – IEEE International Workshop on Security in Software Engineering (IWSSE'07)*, Beijing, July 2007.

[33] P.-A. Masson, J. Julliand, J.-C. Plessis, E. Jaffuel, and G. Debois. Automatic generation of model based tests for a class of security properties. In *A-MOST'07, 3rd int. Workshop on Advances in Model Based Testing*, pages 12–22, London, UK, July 2007. ACM Press.

[34] S. Rayadurgam and M.P.E. Heimdahl. Coverage based test-case generation using model checkers. In *ECBS 2001, 8th Annual IEEE Int. Conf. and Workshop on the Engineering of Computer Based Systems*, pages 83–91. IEEE Computer Society, April 2001.

[35] V. Rusu, L. Du Bousquet, and T. Jéron. An approach to symbolic test generation. In *IFM'00, Int. Conf. on Integrating Formal Methods*, volume 1945 of *LNCS*, pages 338–357. Springer Verlag, November 2000.

[36] V. Rusu, H. Marchand, V. Tschaen, T. Jéron, and B. Jeannet. From safety verification to safety testing. In *TestCom'04, 16-th IFIP Int. Conf. on Testing of Communicating Systems*, volume 2978 of *LNCS*, Oxford, UK, March 2004. Springer-Verlag.

[37] R. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[38] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[39] N. Stouls and M-L. Potet. Security Policy Enforcement through Refinement Process. In J. Julliand and O. Kouchnarenko, editors, *B 2007: Formal Specification ans Development in B*, volume 4355 of *LNCS*. Springer, 2007.

[40] 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, November 2004.

[41] Jan Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29(1):49–79, 1996.

[42] M. Utting and B. Legeard. *Practical Model-Based Testing - A tools approach*. Elsevier Science, 2006.