Automatic Generation of Model Based Tests for a Class of Security Properties*

Pierre-Alain Masson
LIFC, Universite de Franche-Comte
16, route de Gray
25 030 Besancon Cedex - FRANCE
masson@lifc.univ-fcomte.fr

Jacques Julliand
LIFC, Universite de Franche-Comte
16, route de Gray
25 030 Besancon Cedex - FRANCE
julliand@lifc.univ-fcomte.fr

Jean-Chritophe Plessis
LIFC, Universite de Franche-Comte
16, route de Gray
25 030 Besancon Cedex - FRANCE

Eddie Jaffuel
LEIRIOS Technologies
TEMIS Innovation
18, Rue Alain Savary
25000 Besancon - FRANCE
25 000 Besancon
eddie.jaffuel@leirios.com

Georges Debois
GEMALTO
6, rue de la Verrerie
92 190 Meudon - FRANCE
georges.debois@gemalto.com

ABSTRACT

This paper is a contribution to the problem of getting confident in the fact that an implementation correctly meets a security policy assigned to it. To do so, we compute tests that exercise security properties issued from the security policy. We proceed by model based testing. Classically, we use a functional model that formalizes the functional specification. But we also use a second model, in the shape of security properties, that formalize a part of the security policy. Tests are computed from the security properties, with the formal functional model as an oracle.

We first formalize the informal security requirements as regular expressions. Then we introduce mutations in the regular expressions as to reflect the specific situations in which we intend to test the security properties. These mutated regular expression are unfolded into abstract test sequences.

We present a set of four mutation rules that apply to a class of properties that we call *sequencing properties*, and we experiment our method on a standard in the smart card domain named IAS, for Identification, Authentication and electronic Signature.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

*This research is supported by the French National Research Agency (ANR) as the POSE project, which is a RNTL (*Réseau National en Technologies Logicielles*) project.

General Terms

Reliability, Security

Keywords

Security policy, security properties, automatic test generation

1. CONTEXT, MOTIVATIONS AND METHOD

Some systems require strong security guarantees. This is the case for example for embedded software in the transport domain, or for applications in the banking domain. This is also the case when it comes to put personal or medical data on smart cards.

A security policy is usually assigned to such systems, describing functional aspects related to the security of the system itself, as well as constraints on the members of organizations involved in using the system, possible adversaries, etc.

We are only interested in this paper in the functional aspects of the security policy assigned to a system. A challenging question is "how to increase the confidence one can have in the fact that an implementation correctly meets the security requirements expressed in its security policy?"

We propose to compute tests that concentrate on exercising in a variety of manners the security features of a system. In this paper, we consider the testing of *sequencing properties*, which express how the commands of a system are allowed to succeed to each other in an execution. These properties relate to dynamic aspects of the system under test. For example, a requirement for a system might be that an *access* command must always be preceded by an *authentication* command, which verifies whether the access is authorized or not.

These tests can be obtained by means of a model based test-

ing approach [11, 26]. A formal functional model is written according to the specification of the system to test. The model is an abstraction of any real implementation of the system. Hence it is simpler, and is supposed to provide a trusty representation of the expected behavior of the system under test[1].

The tests are computed from the functional model according to one or other test selection criterion [20]. The model also predicts the expected output of the test (the oracle). The system under test is the implementation, and the results of the tests on it are compared with the ones predicted by the model.

For a system where security is crucial, we expect tests to exercise the security requirements in a variety of manners, not just in a "functional way". With the access example, simple functional tests would probably exercise the *access* command in a case where it is authorized (for example with a sequence looking like `authentication` · `access`), and in one case where it is refused (for example with a sequence looking like[2] $\overline{\texttt{access}}$ where no authentication occurred).

We would like to have more tests for such security requirements. For example, we would like to have the test `authentication` ·$\overline{\texttt{authentication}}$ · $\overline{\texttt{access}}$ for testing the absence of side effects due to a previous authentication, or the test `authentication` · ...· `access` where the access is processed some time after the authentication.

To achieve this, we propose to express the security requirements apart from the functional model, as formal security properties, and to compute tests from these properties. The functional model still serves as an oracle for these tests. The security tests obtained this way comes in complement of the already obtained functional tests.

We report in this paper a proposition of such a *property based testing* approach. This work has been experimented in the framework of the french RNTL POSE project. The aim of the project is to produce conceptual, methodological and technical tools for the conformity validation of a system to its security policy. The project brings together industrial (LEIRIOS, GEMALTO, SILICOMP/AQL) and academic (LSR, LIFC/LORIA INRIA Cassis project) partners. The target application domain is the smart card applications. In particular, the results of the project are applied to the IAS, a standard for the operations of Identification, Authenticated and electronic Signature on a smart card.

LEIRIOS has produced a functional model in B of the IAS system, and created and continue to develop the tool LTG[3] that produces functional tests from functional models. The possible behaviors appearing in the functional specification are modelled as operations in the B model. LIFC has formally expressed sequencing properties for IAS, and have computed tests from these properties, with the B model as an oracle. GEMALTO has produced an implementation of IAS, on which the tests have been executed. As neither the IAS specification, nor its B model and implementation are public domain, we won't go too much into the details of these.

Notice that security requirements may appear in the specification as authorized or forbidden cases. We call *nominal* the behaviors described as they appear in the specification. A nominal behavior thus can be positive (describing what is allowed) or negative (describing what is forbidden). The nominal behavior for the *access* example can be expressed (in a positive way) as `authentication`·`access`.

Besides, we consider some *test needs* for the security properties. A test need is issued from the experience of a validation engineer, and can be thought of as a scenario to exercise the property in non-nominal situations. A test need is for example to first gain an authentication and then to loose it, and to try the access, to see if the first authentication has no unwanted side effect on the access.

Our approach is as follows: first we express the nominal behavior as a regular expression: we call it the *original property*. Then we inject a test need inside the property by producing some *mutation* on the regular expression: we syntactically transform the original property, in order to modelize either erroneous or non-trivial cases of execution. The mutated regular expression is unfolded as abstract test patterns, which are sequences of calls to the operations of the functional model. The abstract test patterns are then transformed into abstract tests, by finding from the functional model the correct values for the parameters of the operations that make it possible to execute this sequence of operations. The abstract tests are then transformed into concrete tests and executed on the implementation by translating the operations into commands of the implemented system.

In Section 2, we present some elements about the IAS standard, and also a fragment of its security policy. We study various formalisms for the security properties in Section 3, and explain why we have chosen to express the properties as regular expressions. Our overall approach for the generation of tests based on mutated regular expressions is presented in Section 4, and the results of the experiments on the IAS implementation are given in Section 5.

2. THE IAS APPLICATION AND ITS SECURITY POLICY

IAS is the application on which we have performed our experimentations. It is an industrial standard for smart cards. The specification of IAS [17] is not public domain, and so we will only give a few hints on what it contains in this section. Whereas the IAS 1.01 version of the specification was intended to meet the expectations of the french market, the IAS 2.0 version converges towards the European standard CENTS 15480-2 [1].

2.1 IAS

2.1.1 Presentation of IAS

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. It specifies a set of functions devoted to the above cited opera-

[1]This question is left to the ability of the "modelization engineer".
[2]the line over `access` means *access refused*
[3]LEIRIOS Test Generator

tions of identification, authentication and electronic signature, needed for a smart card application to be declared in conformance with the IAS standard. An application executing on the card will rely on the IAS platform for all such operations.

Smart cards applications such as the french identity card, or the "Sesame Vitale 2" card[4] should conform to IAS.

The IAS standard is proposed and specified by GIXEL[5], which is an association of industries whose intention is to promote and develop new technologies, especially in the smart card domain.

2.1.2 File System Overview

IAS conforms to the ISO 7816 standard. Depending on an industrial choice, it can be based on a JavaCard or not.

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). The file system is organized as a tree structure whose root is designed as MF (*Master File*). An application is assigned a specific DF on the card, which is called an ADF (*Application Directory File*).

A file must first be created. Then it can be activated and then deactivated. May it be activated or not, it can always be terminated or deleted. A file can also be *selected*, to become the *current* file.

The *Security Data Objects* (SDO) are objects of an application that contain highly sensible data such as PIN codes (see for example `PIN1`, `PIN2` and `PIN3` in Fig. 1) or cryptographic keys (see for example `KEY1`, `KEY2` and `KEY3` in Fig. 1), that can be used to restrict the access to some of the data of the application.

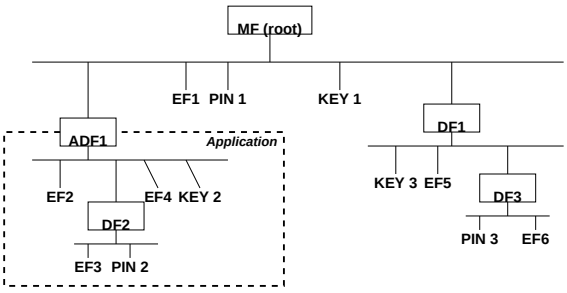

Figure 1: A sample IAS tree structure

2.1.3 A few Commands of IAS

The services implemented by the IAS module in the card can be addressed by the application by means of various APDU[6] commands. We only cite here a few of these commands, as an illustration of what kind of services are proposed by IAS to the applications on the card.

- `CREATE_FILE`: creation of a file (EF or DF),
- `ACTIVATE_FILE`: setting the life-cycle state of a file (EF or DF) to *activated*,
- `DEACTIVATE_FILE`: setting the life-cycle state of a file (EF or DF) to *deactivated*,
- `READ_BINARY`: reading of the content of a file,
- `VERIFY`: authentication by means of a PIN code,
- `GET_CHALLENGE`: getting of an authentication challenge,
- `INTERNAL_AUTH`: authentication of the card *wrt* the terminal,
- `EXTERNAL_AUTH`: authentication of the terminal *wrt* the card,
- `MUTUAL_AUTH`: mutual authentication of the card and the terminal,
- `GENERATE_ASYMETRIC_KEY_PAIR`: generation of the private and public parts of an asymmetric key,
- ...

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

2.2 B Modelization of IAS

Leirios Technologies have developed a B model of IAS, in order to generate model based functional tests of it. The B language [2] is a specification language that allows an abstract description of the functional behavior of the system under test.

The B model for IAS is 15500 lines long. The commands of IAS have been modelized by a set of B *operations*. 60 operations were necessary to fully modelize the commands of IAS.

As the B model of IAS is intended to serve as an oracle for the tests, it has been written as a defensive formal specification. This means that there is no restriction on the invocation of the operations: their pre-condition only checks the typing of the parameters. Thus every input is accepted by the model, provided the parameters in the operation calls fit the signatures of the operations.

The operation responds to an invocation by returning an appropriate status word[7], meaning either *success* or *error*, depending on the context in which the operation is called. For

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

[5] *Groupement des industries de l'interconnexion, des composants et des sous-ensembles électroniques*, it is the trade association in France for electronic components industries

[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, a number that corresponds to a status word of the functional specification

example, trying to apply the operation DEACTIVATE_FILE[8] to an EF that is already deactivated returns a status word of error meaning that the file is already deactivated.

As a consequence, any sequence of operation calls is accepted by the model, which responds by means of status words indicating whether the calls are supposed to be correct or not with respect to the functional specification.

2.3 Security Policy of IAS

The security policy of IAS is not written as a separate document from [17]. Thus we have extracted the security properties directly from the technical specification of IAS.

The main part of the security policy of IAS is devoted to the access control mechanisms, which we describe below. But there are also security requirements regarding the authentication process, and in particular the sequencing of the commands involved in the process. The other features of the security policy of IAS are not given in this paper.

Our approach of producing tests from security properties has been applied to the sequencing properties for the authentication process. The approach also applies to the access control properties, as is discussed as a future direction in Section 6.

2.3.1 Access Control Mechanisms

The access to an object by a command in IAS is protected by means of security rules. Such rules are called the *access rules* to an object in IAS, and they are attached to the object itself. Thus, every object in IAS can define its own access rules for a particular command.

The access rules can possibly be expressed as a conjunction of five elementary access conditions, which are as follows:

- *User* (user authentication) - the user must be authenticated (by means of a PIN code);
- *Ext* (external authentication) - the terminal must be authenticated (by means of an exchange of keys);
- *SM* (secure messaging) - the messages are exchanged in a communication process according to one of the 3 following modes:
  - the message is clear,
  - the message is clear but hashed, and the hash is encrypted,
  - the message is encrypted and hashed, and the hash is encrypted;
- *Always* - the command can always access the object;
- *Never* - the command can never access the object.

The rule *Never* is the default rule.

Consider for example the object EF3 in Fig. 1. The access rule *User* with PIN2 for the command READ_BINARY means that the user must be authenticated by means of the PIN code PIN2 to read the content of the file EF3.

[8]more precisely, the operation in the model corresponding to the APDU command DEACTIVATE_FILE

2.3.2 Sequencing of Commands for the Authentication

The security policy regarding the correct sequencing of the commands in IAS for the authentication process is illustrated by the two following properties:

- any call to the INTERNAL_AUTH command must be preceded (not necessarily immediately) by a successful call to the EXTERNAL_AUTH command,
- any call to the EXTERNAL_AUTH command must be *immediately* preceded by a successful call to the GET_CHALLENGE command.

In the rest of this paper, we will focus on these two properties, and more generally on the properties about the correct sequencing of actions, to illustrate our method of automatic test generation for security properties.

3. FORMALIZATION OF THE SECURITY POLICY

A security policy is in general given as a separate document from the technical specification of a product. This is not the case with IAS where we have extracted from the technical specification [17] the security properties, relevant to security policy of IAS.

In any case, the security policy is not given as a formal document. Thus, the first step towards an automatic exploitation of the security policy is to formalize it as a set of security properties. Our intention is to automatically generate test cases that cover the security properties that we have identified.

Our experiment in the formalization of the security policy has been driven by the security properties encountered in the IAS specification. We have formalized these properties as regular expressions, after having investigated other formalisms such as temporal logics (see section 3.3).

3.1 Regular Expressions

Regular expressions are often well known and used in many tools such as text editors, lexical analyzers, etc. They allow the description of sequences of *patterns* by means of operators such as $\cdot$ (sequence of patterns), * (repetition of a pattern a finite – possibly null – number of times) or + (repetition of a pattern a finite non-null number of times). Also, we sometimes use an interval instead of the operators * or + to bound the number of possible repetitions of a pattern. For example, the replacement of a^* by $a^{0..3}$ indicates that the pattern a can be repeated between 0 and 3 times. We also use the symbol $\times$ to denote a number of repetitions ranging into $\{1, 0..n, *, +\}$. Thus, $a^\times$ can be interpreted either as a, as $a^{0..n}$, as a^* or as a^+.

3.2 Formalization of Some Security Properties of IAS

We will focus in this section on some of the security properties of IAS and formalize them as regular expressions. We recall that each IAS command is modelized by means of B operations in the functional model.

3.2.1 Notations Used in the Formalization

We denote by Ops the set of all the names of the operations of the model. The notation op denotes any operation in Ops. The notation $\mathtt{op}_{\overline{a_1,\ldots,a_n}}$ denotes any operation of Ops distinct from $a_1, \ldots, a_n$. When no suffix is given, the notation op or $\mathtt{op}_{\overline{a_1,\ldots,a_n}}$ simply indicates that the operation has been called, whatever the status word returned. These are predicates of actions. In these predicates, the status word returned by an operation op can be indicated as:

- op_success for a successful execution of op,
- op_error if the status word returned by op indicates an error.

Notice that it is easy to distinguish between these two cases from a B formal model (see Section 2.2).

3.2.2 Properties as Regular Expressions

Sequencing of GET_CHALLENGE and EXTERNAL_AUTH.

We recall that this property specifies that any call to EXTERNAL_AUTH must be immediately preceded by a successful call to GET_CHALLENGE. Operation EXTERNAL_AUTH is denoted as EA and operation GET_CHALLENGE is denoted as GC.

This property can be expressed as a regular expression in the following way:

$$(\mathtt{op}_{\overline{\mathtt{GC,EA}}}{}^{*} \cdot \mathtt{GC\_success} \cdot \mathtt{EA\_success})^{*}$$

Sequencing of EXTERNAL_AUTH and INTERNAL_AUTH.

We recall that this property states that when INTERNAL_AUTH is called, the last call to EXTERNAL_AUTH must have been successful. The operation INTERNAL_AUTH is abbreviated as IA.

The property is expressed with a regular expression as

$$((\mathtt{op}_{\overline{\mathtt{EA,IA}}})^{*} \cdot \mathtt{EA\_success} \cdot (\mathtt{op}_{\overline{\mathtt{EA\_error}}})^{*} \cdot \mathtt{IA\_success})^{*}$$

3.3 Regular Expressions: Limited but Well Suited

It is clear that not all the security properties can be expressed by means of a regular expression. Let us think of a property about the active parents of a DF in IAS: *for a DF to be selected and become the current DF, then its parents must be active.* This property can not be expressed by means of a regular expression. It can be expressed as an invariant state predicate in the shape of

$$(\mathtt{DF\_state}(\mathtt{DF\_parent}(\mathtt{currentDF})) = \mathtt{active})$$

where the meaning of DF_state, DF_parent and currentDF is straightforward.

This "active parents" property had already been formalized as an invariant in the B model, and thus correctly exercised by any functional test. Sequencing properties can not be expressed as such directly in the B model.

Regular expressions are well suited to the expression of properties about the dynamic of the operation calls in a system, which was our main preoccupation in this study. In particular, properties about the correct sequencing of commands, such as GET_CHALLENGE / EXTERNAL_AUTH or EXTERNAL_AUTH / INTERNAL_AUTH exist not only in IAS, but also as equivalents in all smart cards systems.

Other formalisms such as temporal logics are well suited to the expression of dynamic properties. Indeed, we have also tried to express these properties as Linear Temporal Logic (LTL) properties [21, 22], and as Temporal Pattern Language (TPL)[9] properties [25, 16].

The reasons why we have finally chosen regular expressions as the appropriate formalism are double.

First, regular expressions is a language well known and often practiced by the validation engineers to whom the method is intended. This makes it a ready-to-use formalism for the expression of security properties. In contrast, temporal logics such as LTL or TPL are seldom used in an industrial context. As such, a specific formation is needed in order to use them easily as a descriptive language.

Second, temporal logics usually have their semantics defined over infinite executions. A test case has to be a finite execution. The potential length of the test cases described as a regular expression can easily be controlled by bounding the number of possible repetitions of the operation calls inside the expression. This corresponds to the replacement of the $*$ and $+$ operators by intervals (see Section 3.1).

3.4 Link Between the Properties and the Functional Model

We intend to compute tests from the security properties, and to use the formal functional model as an oracle for these tests. To do so, it is necessary that the security properties are "glued" to the functional model. This section explains what knowledge of the functional model is required for the expression of the security properties.

The idea is that a regular expression expresses some particular sequencings of the commands that we want to observe. That is to say we expect the formal functional model to predict the output values of a particular sequencing of commands, so that the values returned by the implementation under test can be compared with those.

3.4.1 Operations and Parameters

We use a tool that performs a symbolic animation of the execution modelled by the regular expression on the functional model. That is to say, the commands in the sequence are executed one by one as operations of the model, with the result of every call predicted by the model for some parame-

[9] TPL is the result of a recent collaboration between LIFC and the INRIA/EVEREST project. It is a specification language close to Linear Temporal Logic, but which allows to mix states and actions aspects in the same formula.

ter values chosen by the model amongst the possible values. The parameter values are chosen as to try to produce a success or an error for the last operation in the sequence (see Section 4.3).

To do so, we need to know the names of the operations of the model, in order to relate them to the names of the commands used in the regular expression.

The operations in the model are parameterized, whereas the commands in the regular expressions need not be so. Indeed, the symbolic animation will guess which values of the parameters make it possible to sequence the operation calls in the same order as they are sequenced in the regular expression. Remember that the model is defensive, so that any sequence of operation calls is feasible (see Section 2.2). The model may respond to some sequence of calls by a status word indicating that, though the sequence is feasible, it will end in an error state. For example, the sequence `GET_CHALLENGE` · `CREATE_FILE` · `EXTERNAL_AUTH` is feasible, though it necessarily leads `EXTERNAL_AUTH` to end in an error state.

Let us precise that it is allowed to give some parameters for the commands inside the regular expressions. If this is to be done, a complete knowledge of the signatures of the operations of the model is needed to relate the parameters of the commands to the ones of the operations. Moreover, paths specified in this way might not be feasible if wrong values are given for the parameters.

3.4.2 Return Values

As indicated in Section 3.2.1, it is possible inside the regular expressions to precise if we want a given operation to end in success state or in an error state.

The B functional model responds to an operation call by returning a status word. So we have to distinguish between all the possible status words returned by an operation which ones correspond to `success` and which ones correspond to `error`.

It is also possible not to precise the expected result of a command, and to guess the possible values from the model by symbolic animation. Here again, if the expected results are given *a priori* and not guessed from the model, there is a risk that the sequence might not be feasible.

4. THE TEST GENERATION PROCESS

We have developed a method to automatically generate abstract tests for the fragment of the security policy of a system that can be expressed as regular expressions. This method is illustrated in Fig. 2.

We take as input the security requirements, the test needs and a functional model. In our experiment with IAS, the security requirements are a part of the functional specification. The test needs for the various security requirements have been expressed by validation engineers from both GEMALTO and LEIRIOS. The functional model is a B model.

The security requirements are formally expressed as security properties in the shape of regular expressions. The test needs are formalized as mutation rules, that is to say as syntactic replacements that apply to regular expressions.

The output of the process is a set of abstract tests, that is to say some sequences of parameterized calls to the operations of the model, with the expected results of such calls (the oracle).

The method proceeds in four steps: *formalization*, *mutation*, *unfolding* and *instantiation*.

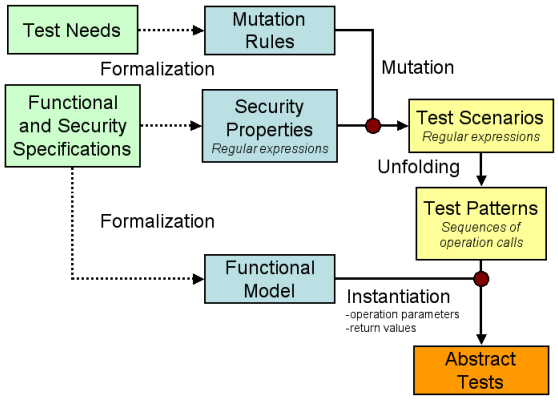

Figure 2: The test generation process

Step 1 - Formalization. The first step in the process is the formalization of the security requirements and of the test needs. The security requirements are formalized as security properties in the shape of regular expressions. The test needs are formalized as mutation rules, which perform syntactic transformations on regular expressions.

The formalization of the security requirements is performed by the validation engineer for each security requirement as expressed in the specification. Thus the set of security properties is specific to every system studied. On the contrary, the formalization of test needs as mutation rules can be performed once and for all. Test needs are generic, and when a new test need appears, it can be added as a new mutation rule to the existing base of mutation rules.

We give in Section 4.1 some generic rules that express generic test needs.

Step 2 - Mutation. At the second step, the security properties issued from the previous step are mutated as to incorporate the test needs. This is performed by applying the mutation rules to the regular expressions. The aim is to exercise the property in non-nominal situations. The result is a new set of regular expressions that we call the *test scenarios*.

Step 3 - Unfolding. The result of the third step is a set of *test patterns*, in the shape of sequences of operation calls. This is obtained from the previous step by unfolding the regular expressions (the test scenarios).

Notice that the number of test patterns issued from a single

regular expression is potentially infinite, due to the operators $*$ and $+$. In practice, every number of repetitions should be bounded before unfolding.

Also notice that the non-named operations (denoted by `op`) have to be replaced by all the possible operation names.

Step 4 - Instantiation. The fourth step computes for each test pattern a set of *abstract tests*, in the shape of a sequence of parameterized operation calls, with the expected results of each operation call. This is performed by the tool LTG[10][10] by means of a constraint solving technique from the functional model, as explained in Section 4.3.

Notice that the abstract tests generated this way have the same abstraction level as the functional model. So a concretization phase is needed before the tests can be executed on the implementation under test. In our case, the concretization is performed by LTG by means of an adaptation layer, which is written in coordination by both the model and the implementation providers. We do not discuss this technical point here.

4.1 Mutation of Properties

The mutation of a property syntactically changes the expression of the property, in order to express a test need related to the property. Test needs are supposed to be generic and should apply to various case studies. They are first expressed by experienced validation engineers and formalized as mutation rules, that perform syntactic transformations.

We give in this section four mutation rules that were inspired by the IAS study. The first one is completely generic, and the others may apply to any other system with security properties stating how operations are allowed to be sequenced.

The first of these four rules omits the kind of termination of an operation (`_success` or `_error`), so that all the possible behaviors of the operation can be guessed from the functional model. This is expressed as rule 1:

$$\texttt{op}\{\texttt{\_success}, \texttt{\_error}\} \rightsquigarrow \texttt{op} \tag{1}$$

The second and third rules break the sequencing of two operations. When a given operation a must be immediately followed by a given operation b:

- the rule 2 introduces the possibility for a either not to occur at all, or to occur several times[11],
- the rule 3 introduces the possibility for any operation (denoted as `op`) distinct from a and b to occur between a and b.

The second rule is expressed as

$$a \cdot b^{\times} \rightsquigarrow a^{*} \cdot b^{\times} \tag{2}$$

The third rule is expressed as

$$a^{\times} \cdot b^{\times} \rightsquigarrow a^{\times} \cdot (\texttt{op}_{\overline{a,b}})^{0..1} \cdot b^{\times} \tag{3}$$

Notice that breaking the sequencing of any operation `op` and a given operation a is not necessary.

The fourth rule is for the case when a given operation b must have been preceded by a given operation a with any number of other operations between a and b. The mutation rule 4 introduces the possibility for a not to occur. This rule is expressed as

$$a \cdot (\texttt{op}_{\overline{a,b}})^{*} \cdot b^{\times} \rightsquigarrow a^{*} \cdot (\texttt{op}_{\overline{a,b}})^{*} \cdot b^{\times} \tag{4}$$

4.2 Generation of the Test Patterns

The test patterns are obtained by unfolding the mutated regular expressions. As noticed above, the possible repetitions of the sub-patterns inside the regular expression have to be bounded, so that the expression unfolds as a finite number of test patterns.

In our implementation of the method, the regular expression is first transformed into a minimal deterministic automaton, which is then unfolded. We have used for this purpose a standard tool, the Java package `dk.brics.automaton`[12].

The coverage criterion "all the paths of the property" leads to a complete unfolding of the automaton. There is a risk of a combinatorial explosion of the number of test patterns obtained this way. If so, we suggest to offer to the validation engineer the ability to choose a different coverage criterion, such as *all the states*, *all the transitions*, *all the cycles*.

Besides, the advantage of using automata is that this intermediate form is common to other property describing languages such as temporal logics or sequence diagrams. Thus the method could easily be generalized to these formalisms, provided that the mutation are applied directly to the automata and not to the regular expressions.

4.3 Test Generation

A test pattern obtained at the previous step is a sequence of operation calls in the shape of $\texttt{op}_1 \cdot \texttt{op}_2 \cdots \texttt{op}_n$. Some of the operation calls may be parameterized, if they were so in the original regular expression.

The aim of the step of instantiation is to compute a set of test cases that conform to the above cited pattern. For this, all of the parameters of the operations have to be instantiated, and the expected results of such successive calls are computed from the functional model. These two tasks are performed by the tool LTG as explained below.

In the sequence $\texttt{op}_1 \cdot \texttt{op}_2 \cdots \texttt{op}_n$, we distinguish between the last operation call $\texttt{op}_n$, which we design as the *targeted* operation, and the preceding sequence $\texttt{op}_1 \cdot \texttt{op}_2 \cdots \texttt{op}_{n-1}$. LTG tries to generate one test for each possible behavior of the targeted operation. In our case where the operations return status words, this means that there will be one test computed for each possible status word returned by $\texttt{op}_n$.

[10] LTG is an automatic test generator commercialized by LEIRIOS Technologies (see `http://www/leirios.com`)

[11] several occurrings of an action allows testing the possible side effects of the first occurring on the following ones

[12] This package has been developed by A. MØLLER, and can be found at `http://www.brics.dk/automaton/`

A test is computed by finding a suitable *preamble* for each behavior of the targeted operation. A preamble is one sequence $\texttt{op}_1 \cdot \texttt{op}_2 \cdots \texttt{op}_{n-1}$ of operation calls in which the values of the parameters of the operations are chosen as to produce the desired behavior when $\texttt{op}_n$ is called. If no instantiation of the parameters make it possible to produce one particular behavior, then no test is computed for this case.

For example, if the operation `EA` is supposed to return only two possible status words (one for `success` and one for `error`), then two tests are computed from the sequence $\texttt{GC} \cdot \texttt{EA}$. In the first one the parameters of the call to `GC` are chosen so that `EA` ends successfully, and in the second one the parameters of `GC` make that `EA` fails.

The values of the parameters are computed from the functional model by means of a constraint solving technique with a limit strategy [9, 12].

5. EXPERIMENTATION, CHECKUP AND STATE OF THE ART

5.1 Implementation

We have implemented the method as part of the LTG platform. The steps of mutation and of sequence generation have been implemented by means of the Java package `dk.brics.automaton`. The tool produces all the sequences that can possibly be obtained by unfolding the automaton. For the number of sequences to be finite, the tool takes a regular expression as input in which the number of possible occurrences of each pattern is bounded (i.e. every sign $*$ and $+$ in the regular expression is replaced by a constant interval).

The tests generation has been implemented in LTG by means of the *preamble helper* technique [12], which valuates by constraint solving a preamble $\texttt{op}_1 \cdot \texttt{op}_2 \cdots \texttt{op}_{n-1}$ for a target behavior of an operation $\texttt{op}_n$ (see Section 4.3).

5.2 Experiments and Results on IAS

We have experimented our method on the property stating that any external authentication must be immediately preceded by a successful command `GET_CHALLENGE` (see Section 2.3.2). This property has been formalized as the the following regular expression (see Section 3.2.2):

$$(\texttt{op}_{\overline{\texttt{GC,EA}}}{}^{*} \cdot \texttt{GC\_success} \cdot \texttt{EA\_success})^{*}$$

The application of mutation rule 1 about the abstraction of the type of termination of the operations gives the following regular expression:

$$(\texttt{op}_{\overline{\texttt{GC,EA}}}{}^{*} \cdot \texttt{GC} \cdot \texttt{EA})^{*}$$

The application of mutation rule 2, allowing none or several occurrences of `GC` gives:

$$(\texttt{op}_{\overline{\texttt{GC,EA}}}{}^{*} \cdot \texttt{GC}^{*} \cdot \texttt{EA})^{*}$$

The application of mutation rule 3 about the sequence breaking gives:

$$(\texttt{op}_{\overline{\texttt{GC,EA}}}{}^{*} \cdot \texttt{GC}^{*} \cdot (\texttt{op}_{\overline{\texttt{GC,EA}}})^{0..1} \cdot \texttt{EA})^{*}$$

Then we have limited the length of the test sequences to be produced from this expression, in the following way:

- we want to observe this particular sequence of operations only once or twice, and we have replaced the final * by the bounded repetition 1..2. Notice that playing the sequence of operations twice allows the observation that playing it successfully once have no side effect on replying it in a different context (where it could possibly fail);
- we have bounded $\texttt{op}_{\overline{\texttt{GC,EA}}}{}^{*}$ as $\texttt{op}_{\overline{\texttt{GC,EA}}}{}^{0..1}$;
- we have bounded $\texttt{GC}^{*}$ as $\texttt{GC}^{0..2}$.

Thus, the final expression from which we have generated the abstract test sequences is:

$$(\texttt{op}_{\overline{\texttt{GC,EA}}}{}^{0..1} \cdot \texttt{GC}^{0..2} \cdot (\texttt{op}_{\overline{\texttt{GC,EA}}})^{0..1} \cdot \texttt{EA})^{1..2}$$

The complete unfolding of this test scenario has produced 24 different test patterns.

LTG has then computed 36 different test cases from these 24 patterns. Indeed, when no intermediate operation have been added between `GC` and `EA` (12 patterns), LTG has computed 2 tests: one for `EA_success` and one for `EA_error`. With an operation between `GC` and `EA` (12 patterns), only the behavior `EA_error` was reachable, which gave 12 tests.

All of these 36 tests have been executed by the GEMALTO team on the IAS implementation, and the results were all OK. The fact that the results were all OK is not surprising since the GEMALTO team had correctly taken care of this sequencing property during the development phase of IAS.

Amongst these 36 tests, only 2 had already been obtained by the standard functional tests generation process of LTG [6, 8]. These are the tests `EA_error` and $\texttt{GC\_success} \cdot \texttt{EA\_success}$. This result comes from the functional test strategy, which efficiently faces the combinatorial explosion problem by limiting the number of tests issued from one abstract sequence. Thus, the functional test strategy produces only one test by target behavior of `EA`, i.e. `EA_success` and `EA_error`, and this test is as short as possible. Then, our test generation strategy completes the set of functional tests by a set of sequencing tests in various contexts, chosen by the validation engineer.

5.3 Related Works

TOBIAS[13] [19, 13] is a tool for generating test cases in a combinatorial way. Its principle is very close to our method since it works by unfolding test patterns described as regular expressions. Nevertheless, our work differs from the approach of test generation in TOBIAS in two points. One the first hand, by using the LTG capacity of performing a constraint evaluation of the functional model, we only produce executable tests provided with the operation parameters and the oracle. As TOBIAS generates a set of sequences in a combinatorial way, and independently from any predicting

[13]Test OBjective desIgn ASsistant

model, many of the sequences generated may correspond to non executable tests. On the other hand, we have defined a step of mutation of the property, in order to generate robustness tests of the properties formalized as regular expressions. This allows an automatic computation of many interesting test cases from a single nominal property. Let us notice that once the properties have been mutated, it is possible to use TOBIAS to unfold them as test patterns. LTG can still be used after that to generate the executable tests from these patterns.

The mutation technique is also used by P. AMMANN and P. BLACK [3, 7]. They propose a different approach of mutation of the operands and of the relational and logical operators in the expressions of the functional model. What we do is a mutation of the operations sequencing.

Our approach is to be compared with the work on test generation by model-checking, as in [24, 3, 15, 23], or to similar approaches like TGV [14, 18]. TGV is a tool allowing automatic synthesis of conformance tests of an implementation to a formal specification. The functionalities to be tested are described as *test purposes*. The concept of test purpose is close to our concept of test need. Our approach is different in the sense that our "test purposes" need not be given explicitly by the user. They are computed automatically from the nominal cases, where the test need is captured by applying appropriate mutation rules.

In [24], O. SOKOLSKY proposes a method that generates a set of tests from a LTL property that is quantified existentially. These tests are intended to be *non trivial* tests of the LTL property. They relate the non triviality concept to the non vacuity satisfaction concept [5, 4] of a LTL formula. We also produce non trivial tests, but in a different way, by applying some selected mutation rules to a property describing the nominal behavior. In [24], no human interaction is needed to produce the abstract test sequences since the non trivial properties can be automatically computed from the original property, and that the corresponding abstract test sequences are computed by means of a model-checker. These test sequences are witness traces satisfying the non trivial properties. The length of these sequences is bounded by the number of states (possibly very high) of the implementation, which may be an information hard to know *a priori*. On the contrary, our method requires that a validation engineer pilots the production of the abstract test sequences. This allows shorter tests sequences to be generated, as no information about the possible number of states of an implementation is required, and that the piloting allows the validation engineer to bound the length of the abstract test sequences. Moreover, our method allows to produce the tests directly from a property specified as a regular expression and a functional model in B, with no need to translate it into an explicit state transition system intended to a model-checker.

6. CONCLUSION AND PERSPECTIVES

6.1 Summary of the Approach

We have presented in this paper some experiments relative to a property based testing approach, which is a model based testing approach from both a formal functional model of a system, and a formal expression of the security requirements as security properties. We have formalized a class of security properties (the sequencing properties) as regular expressions. We have produced some mutations to these regular expressions in order to modelize erroneous or non-trivial executions with respect to the security property. The mutated regular expressions have then been transformed into tests by unfolding the regular expressions, and by computing the corresponding sequence of operation calls from the functional model.

Our experiments have used the IAS standard for smart cards as a case study, and we have produced a set of security tests that have been exercised on an implementation of IAS.

6.2 Open Issues

6.2.1 Generalization of the Method

We are now aiming at generalizing this approach to other security properties than sequencing properties, and to other formalisms than regular expressions.

In particular, we have mentioned in Section 3.3 the possibility to express the security properties as temporal logic formulas. Regular expressions, LTL and TPL share a common underlying formalism: the one of automata. Our idea is that instead of defining mutation rules that only apply to regular expressions, it would be interesting to define mutation rules that apply directly to the automata. This way we could generalize our mutation method to all of the above cited formalisms, as tools exist that transform each of these formalisms to (regular or Büchi) automata.

Then the problem is to compute the abstract test sequences from the automaton. If the property is an *action* property, as it is the case with our regular expressions, then the unfolding of the automaton directly produces sequences of operation calls as before. But what if the input property is a *state* property, as it is in general the case with LTL properties? The unfolding of the automaton produces a sequence of states of the system, and not a sequence of operation calls. It is thus necessary to find from the functional model which sequences of operation calls make it possible to reach successively the states in the sequence. This can be obtained by means of the constraint solving techniques used in LTG, and we are working at it at the present time.

Also, the method is not limited to sequencing properties, and should apply to any property that can be formalized as an automaton. We think in particular to some access control properties, where we could successively exercise the acquisition and loss of the rights to access an asset. It is necessary, for such a generalization, that new mutation rules are proposed, according to the "meaning" of the property that is formalized as an automaton.

6.2.2 Control of the Combinatorial Explosion of the Number of Tests

There is a risk that the systematic application of the method leads to a combinatorial explosion of the number of tests produced, all along the test generation process. The introduction of piloting at various steps of the process shall make it possible to control the combinatorial explosion.

Currently, the abstract test sequences are obtained by a complete unfolding of the automaton. Some coverage criteria (such as *all of the states*, or *only some of the transitions*, ...) could be chosen by the validation engineer to restrict the number of abstract test sequences obtained by the traversal of the automaton.

Also, when a mutation introduces the possibility for any operation `op` to occur in a sequence, it can potentially be instantiated as *any operation* of the model. As there might be a great number of these, it multiplies the number of possible abstract tests sequences issued from the same mutated property. The validation engineer could allow only a subset of the operations to instantiate `op`.

The same kind of problem occurs with the parameters of the operations. As they are not specified in the original property, there can be a great number of instantiations of these parameters that fit the pattern defined by the mutated property. The validation engineer should then also have the possibility to choose amongst the various possible instantiations of the parameters.

We are currently pursuing our experiments on the IAS standard, in the framework of the POSE project.

7. REFERENCES

[1] CEN/TS 15480 - Standard European citizen card progress, 2006. www.unilink.it/Portals/57ad7180-c5e7-49f5-b282-c6475cdb7ee7/08_CEN-TS%2015480%20standard.pdf.

[2] J.-R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[3] P. Ammann, P. Black, and W. Majurski. Using model checking to generate tests from specifications. In *ICFEM'98, 2nd IEEE Int. Conf. on Formal Engineering Methods*, pages 46–54. IEEE Computer Society Press, Dec. 1998.

[4] R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M. Vardi. Enhanced vacuity detection in linear temporal logic. In *CAV'03, Computer Aided Verification*, volume 2725 of *LNCS*, pages 368–380, 2003.

[5] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in temporal model checking. *Formal Method in System Design*, 18(2):141–163, Mar. 2001.

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

[7] P. Black, V. Okun, and Y. Yesha. Mutation operators for specifications. In *ASE'2000, 15th Automated Software Engineering Conference*, pages 81–88, Grenoble, France, Sept. 2000. IEEE Computer Society Press.

[8] F. Bouquet, F. Lebeau, and B. Legeard. Test case and test driver generation for automotive embedded systems. In *5th Int. Conf. on Software Testing, ICS-Test 2004*, pages 37–53, Düsseldorf, Germany, Apr. 2004.

[9] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B: A constraint solver to animate a B specification. *International Journal on Software Tools for Technology Transfer, STTT*, 6(2):143–157, Aug. 2004.

[10] F. Bouquet, B. Legeard, F. Peureux, and E. Torreborre. Mastering Test Generation from Smart Card Software Formal Models. In *CASSIS'04, Int. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *LNCS*, pages 70–85, Marseille, France, Mar. 2004. Springer.

[11] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*. Springer, 2005.

[12] S. Colin, B. Legeard, and F. Peureux. Preamble computation in automated test case generation using Constraint Logic Programming. *The Journal of Software Testing, Verification and Reliability*, 14(3):213–235, 2004. Selected papers from the 2003 UK-Test Workshop.

[13] L. du Bousquet, Y. Ledru, O. Maury, C. Oriat, and J.-L. Lanet. Case study in JML-based software validation. In *ASE'04, 19th IEEE Int. Conf. on Automated Software Engineering*, pages 294–297, Linz, Austria, Sept. 2004. IEEE Computer Society.

[14] 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.

[15] 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.

[16] A. Giorgetti and J. Groslambert. JAG: JML Annotation Generation for verifying temporal properties. In *FASE'2006, Fundamental Approaches to Software Engineering*, volume 3922 of *LNCS*, pages 373–376, Vienna, Austria, Mar. 2006. Springer.

[17] GIXEL. *Plateforme commune pour l'eAdministration*, spécification technique, IAS 2.0 edition, 2004.

[18] 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.

[19] Y. Ledru, L. du Bousquet, O. Maury, and P. Bontron. Filtering TOBIAS combinatorial test suites. In *FASE'04, Fundamental Approaches to Software Engineering*, volume 2984 of *LNCS*, pages 281–294, Barcelona, Spain, Mar. 2004. Springer.

[20] L. Lúcio and M. Samer. Technology of test-case generation. In *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*, pages 323–354. Springer, 2005.

[21] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symp. on Foundations of Computer Science (FOCS'77)*, pages 46–57, 1977.

[22] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[23] S. Rayadurgam and M. 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, Apr. 2001.

[24] 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*, Nov. 2004.

[25] K. Trentelman and M. Huisman. Extending JML with temporal logic. In *9th Int. Conf. on Algebraic Methodology And Software Technology (AMAST'02)*, volume 2422 of *LNCS*, pages 334–348, St-Gilles-Les-Bains, Ile De La Réunion, France, Sept. 2002. Springer-Verlag.

[26] M. Utting and B. Legeard. *Practical Model-Based Testing - A tools approach*. Elsevier Science, 2006. ISBN 0-12-372501-1.