Model-Based Security Verification and Testing for Smart-cards

Elizabeta Fourneret*, Martín Ochoa†,
Fabrice Bouquet*, Julien Botella‡, Jan Jürjens†, Parvaneh Yousefi†

*LIFC, Université de Franche-Comté, Besanon, France
elizabeta.fourneret, fabrice.bouquet@lifc.univ-fcomte.fr

†Software Engineering, Department of Computer Science, TU Dortmund, Germany
martin.ochoa, jan.jurjens, parvaneh.yousefi@cs.tu-dortmund.de

‡Smartesting, TEMIS Innovation, Besanon, France
botella@smartesting.com

***Abstract*—Model-Based Testing (MBT) is a widely used methodology for generating tests aiming to ensure that the system behaviour conforms to its specification. Recently, it has been successfully applied for testing certain security properties. However, for the success of this approach, it is an important prerequisite to consider the correctness of test models with respect to the given security property. In this paper we present an approach for smart-card specific security properties that permits to validate the system with MBT from test schemas. We combine this MBT approach with UMLsec security verification technique, by using UMLsec stereotypes to verify the model w.r.t. given security properties and gain more confidence in the model. We then define an automatic procedure to generate security test from the UMLsec model via so-called "test schemas". We validate this approach on a fragment of the Global Platform specification and report on available tool support.**

***Keywords*-Verification; Model-Based Testing; Model-Based Testing from schemas; UML/OCL statechart; smart-cards; Global Platform**

I. Introduction

Typically, UML models verified against security properties are explicit models of the *system* design, whereas in Model-Based Testing (MBT) we describe the expected behaviour of an application, seen thus as a *blackbox*. With the current state of the art, on one hand it is possible for a system engineer to design a conception model annotated with security properties that can be verified using automated theorem provers and model-checking, for example using the UMLsec approach [1]. On the other hand, the validation engineer designs a UML test model, and writes test scenarios that are used to produce test cases exercising security properties. This situation is depicted in Figure 1. The security properties considered for testing are typically expressed at different abstraction levels with respect to the used properties, because they will be executed on the implementation of the system (the System Under Test or SUT).

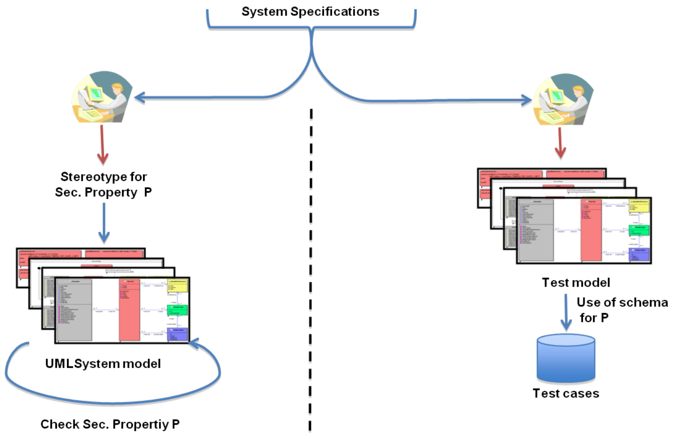

Figure 1. Initial approach to security verification and model-based testing

However, the test engineer has no formal guarantee that the test model under consideration is trivially violating the security property, that (s)he would like to test. In other words, little attention is paid to the fact that the *expected behaviour* expressed in the model could be contradicting the property under test. In this article, we focus only security property but it is same in general purpose. Thus, our goal is to generate tests guided by the security properties and the testing model, but we want to start with a correct model in the first place. Moreover, it is desirable that the security property formalized to check the model for correctness can be further used to generate test sequences following the model based testing paradigm.

In this paper we consider two generic security properties that are relevant for smart-cards and we show how can one benefit from the UMLsec verification approach for security to ensure the correctness of the UML test model and from the MBT approach in order to generate tests for complex situations issued from the security properties. We also show

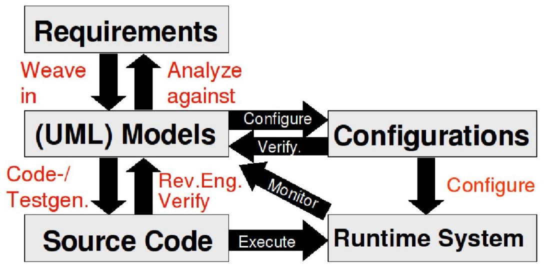

Figure 2. Model-based Security Engineering

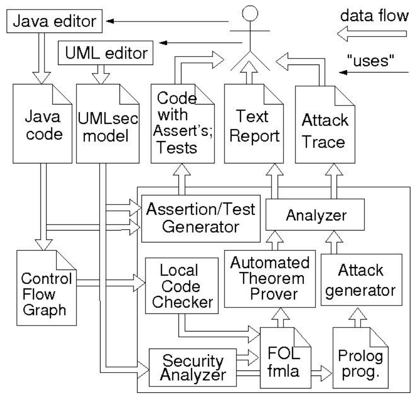

Figure 3. UMLsec Tool support

how the two approaches are linked by means of a testing Schema language, that can be used to automatically generate testing sequences.

To validate this approach, we demonstrate our results on the Global Platform Specification for smart cards [2]. We also report on available tool support for our approach.

The paper is organized as follows: Section II contains a summary of the core concepts of the UMLsec approach while Section III introduces Model-Based Testing for security, that we consider. Section IV explains how to verify consistency of the UML testing model with UMLsec with respect to the chosen security properties. Sect. V contains the validation case study and summarizes the available tool support. Relevant related work is discussed in Section VI.

II. Background: UMLsec

Generally, when using model-based development (Fig. 2), the idea is that one first constructs a model of the system. Then, the implementation is derived from the model: either automatically using code generation, or manually, in which case one can generate test sequences from the model to establish conformance of the code regarding the model. In the model-based security engineering (MBSE) approach based on the UML extension: UMLsec [1], recurring security requirements (such as secrecy, integrity, authenticity, and others) and security assumptions on the system environment, can be specified either within UML specifications, or within the source code (Java or C) as annotations (Fig. 3). This way we encapsulate knowledge on prudent security engineering as annotations in models or code and make it available to developers who may not be security experts. The UMLsec is given in form of a UML profile using the standard UML extension mechanisms. *Stereotypes* are used together with *tags* to formulate the security requirements and assumptions. *Constraints* give criteria that determine whether the requirements are met by the system design, by referring to a precise semantics of the used fragment of UML. The added security-relevant information using stereotypes includes security assumptions on the physical level of the system, security requirements related to the secure handling and communication of data, and security policies that system parts are supposed to obey. The semantics for the fragment of UML used for UMLsec is defined in [1] using so-called *UML statecharts*. On this basis, important security requirements such as secrecy, integrity, authenticity, and secure information flow are defined.

The UMLsec tool can be used to check the constraints associated to UMLsec stereotypes mechanically, based on XMI output of the diagrams from the UML drawing tool in use [3], [4]. They generate logical formulas formalizing the execution semantics and the annotated security requirements. Automated theorem provers and model checkers automatically establish whether the security requirements hold. If not, we can use Prolog to automatically generate an attack sequence violating the security requirement, which can be examined to determine and remove the weakness. Since the analysis that is performed is too sophisticated to be done manually, it is also valuable to security experts. There is also a framework for implementing verification routines for the constraints associated with the UMLsec stereotypes. Thus, advanced users of the UMLsec approach can use this framework to implement verification routines for the constraints of self-defined stereotypes. There are several existing UMLsec stereotypes supporting security related information including security assumptions (such as the «internet» stereotype considering security properties of the physical layer of the system), security requirements (such as the «authenticity» stereotype considering authenticity of any entity), and security policies (such as the «secure links» stereotype supposed to be obeyed on different parts of the system).

The tags defined in UMLsec represent a set of desired properties. For instance, ”freshness” of a value means that an attacker cannot guess what its value was. Moreover, to represent a profile of rules that formalize the security requirements, the following are some of the stereotypes that are used: «critical», «high», «integrity», «internet», «encrypted», «LAN», «secrecy», and «secure links».

In this paper, we present an extension of UMlsec stereotypes for security relevant properties for smart-cards, which are used to drive the test generation dedicated to security. We use then the UMLsec tool to check the property on the model and extract a logical formula, that are going to be used to create test schemas, detailed in the next section.

III. Test Process for Security Properties

Model-Based Testing makes use of selection criteria that indicates how to *select* the tests to be extracted from the model. These criteria usually ensure a given structural coverage of the model, such as *all the states*, *all the transitions*, etc.

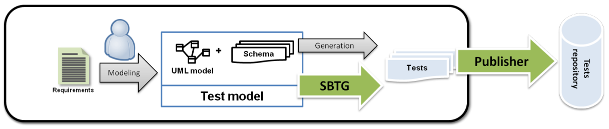

Figure 4. The Model-Based Process for testing Security Properties

Each test is a sequence of operation calls with parameter values, which yields a distinguished execution of the model. Their results are predicted by the model. Our approach (depicted on the Figure 4) for testing security properties relies on defining additional selection criteria in the shape of *test schemas*. We use the Schema Based Test Generator (SBTG) to unfold the schema and then we use the *Smartesting CertifyIt* tool to generate tests dedicated to the security property. Once the tests are generated it is possible to publish them into a test repository, used for test management.

A. Presentation

A test schema is a high level expression that formalizes the test intention linked to a security property to drive the automated test generation on the behavioral model. In this approach, the security requirements that a system must fulfil are expressed as a set of *security properties*. We propose test schemas as a means to exercise the system to validate that it behaves as predicted by the model w.r.t. these security properties. Based on his know-how, an experienced security engineer will imagine possible scenarios in which (s)he thinks the property might be violated by an erroneous implementation, and then on the basis of this test intention, (s)he will formalize test schemas to drive the automated test generation.

Test schema is based on regular expressions and allows the security testing engineer to conceive its test schemas in terms of states to be reached and operations to be called. It is based on the work done in [5] and its formal semantics has been defined in [6].

Based on this conceptual language, an operational "user friendly" language has been defined within the SecureChange project and implemented by Smartesting as Eclipse plug-in to IBM Rational Software Architect.

The syntax of the language is defined by means of the grammar given in Figure 6. The language makes it possible to design test schemas as a sequence of quantifiers or blocks, each block being composed of a set of operations (possibly iterated at least once, or many times) and aiming at reaching

for_each	quantifier for an operation or a behaviour
from	to introduce a list of operations or behaviours
then	a separator for sequencing the targets to be reached
use	to introduce an operation a behaviour or a variable to use
to_reach	to introduce a state to be reached
to_activate	to introduce a behaviour to be activated
state_respecting	to introduce a constraint that characterize a set of states
on_instance	to introduce an instance on which a constraint holds
any_operation	the set of all the operations of the model
any_operation_but	the set of all the operations of the model minus the ones whose list follows
or	for a disjunction of operations or of behaviours
any_behaviour_to_cover	the set of all the behaviours of the model
any_behaviour_to_ cover_but	the set of all the behaviours of the model minus the ones whose list follows
behaviour_activating	to introduce a list to be covered of behaviours tagged in the model
behaviour_not_activating	to introduce a list whose complementary must be covered of behaviours tagged in the model
at_least_once	repetition operator indicating to apply at least once the operation or behaviour previously specified
any_number_of_times	repetition operator indicating to apply any number of times the operation or behaviour previously specified
$	variable prefix
REQ	to introduce a tag that corresponds to a requirement
AIM	to introduce a tag that corresponds to an aim

Figure 5. Keywords of Schema Language

a given target (a specific state, the activation of a given operation, etc.).

```
         SCHEME  ::=  (QUANTIFIER_LIST ,)? SEQ
QUANTIFIER_LIST  ::=  QUANTIFIER (, QUANTIFIER)*
     QUANTIFIER  ::=  for_each VAR from ( BEHAVIOR_CHOICE
                                          | OP_CHOICE )
BEHAVIOR_CHOICE  ::=  any_behaviour_to_cover
                  |   any_behavior_to_cover_but
                                          BEHAVIOR_LIST
  BEHAVIOR_LIST  ::=  BEHAVIOR (or BEHAVIOR)*
       BEHAVIOR  ::=  behavior_activating TAG_LIST
                  |   behavior_not_activating TAG_LIST
       TAG_LIST  ::=  { TAG (, TAG)* }
            TAG  ::=  REQ: tag name | AIM: tag name
      OP_CHOICE  ::=  any_operation | OP_LIST
                  |   any_operation_but OP_LIST
        OP_LIST  ::=  OPERATION (or OPERATION)*
      OPERATION  ::=  operation name
            SEQ  ::=  BLOC (then BLOC)*
           BLOC  ::=  use CONTROL (RESTRICTION)? (TARGET)?
        CONTROL  ::=  OP_CHOICE | BEHAVIOR_CHOICE | VAR
            VAR  ::=  $variable name
    RESTRICTION  ::=  at_least_once | any_number_of_times
         TARGET  ::=  to_reach STATE
                  |   to_activate BEHAVIOR
                  |   to_activate VAR
          STATE  ::=  state_representing ocl constraint
                      on_instance instance name
```

Figure 6. Syntax of the Smartesting Schema Language

A dedicated schema language editor has been implemented as a plug-in of Smartesting CertifyIt. Its aim is to provide a means to express security properties at a high level, close to a textual representation or by using usual computer programming paradigms. The expression of these properties allows the test specifications generating, called *Test Case Specification - TCS*, that are high level scenarios from which tests will be generated by CertifyIt.

The language relies on combining keywords, to produce expressions that are both powerful and easy to read by a validation engineer.
In Fig. 5 we define the language keywords. For each keyword, we give its intuitive meaning. A couple of illustrative examples are in section V.

B. *Interest*

We find, that there are several benefits from the schema language. Firstly, from a scientific point of view, the language that is described previously makes it possible for the validation engineer to express his or her test schemas by combining sequences of actions of the system to be called, along with the description by means of predicates of the states to be reached by these sequences of calls.

Secondly, from a technological point of view, the language is designed to be easy to use by a validation engineer. (S)he writes test schemas using model's artefacts, with constructions close to usual computer programming paradigms. That frees him or her from manipulating mathematical notations such as in the temporal logics.

Thirdly, by its expressivity, the language is designed as a mean for the validation engineer to describe the test intentions w.r.t. a property that has to be tested. This feature strongly helps to monitor the coverage of tested properties.

IV. Integrated approach

In this section, first we describe how to improve the quality of the test models by using UMLsec. And secondly, how to obtain schemas used for test generation with respect to a given security property from a UMLsec stereotype. The model that is used for test generation has to be verified for consistency with respect to the considered security properties. If not, the model may authorize an incorrect behaviour and the produced tests will expect from the System Under Test (SUT) to present the same behaviour as the model.

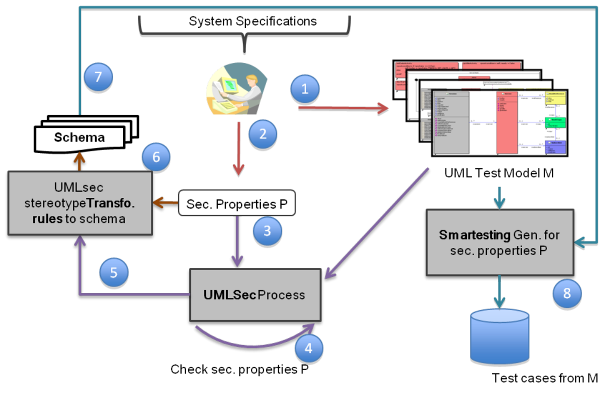

Figure 7. Integration of the two approaches

The process is summarized in Fig. 7. First, a validation engineer designs a test model (Step 1). (S)He then extracts the security properties from the specification (Step 2). Afterwards in Step 3 (s)he writes the corresponding UMLSec stereotypes. (S)He uses the UMLsec approach to validate the model against the security properties (Step 4), to make sure that the model respects them. Once the model is declared correct, a Hoare triple for each property is exported by Step 5. Then, we use transformation rule to automatize the schema writing with respect to the property (Step 6). The created schema (Step 7), can be used to produce test cases exercising the property (Step 8).

A. *Security properties*

We consider the following two properties from Global Platform [2] that are critical for a card issuer in order to have control over compromised running smart-cards.

Security Property 1: *For any execution, whenever the card is set to the state TERMINATED by means of a operation performed by a privileged application, then it should not be possible to revert to another state.*

This property ensures that whenever an application with enough privileges terminates a card, the card cannot be put back in operation. This is an important feature to control smart-cards running malicious applications or that have been compromised in some way.

Security Property 2: *It should not be possible for an application that does not have the given privilege to set the card into a given state TERMINATED.*

Conversely, to avoid the Denial of Service (D.O.S) attacks on the card, only applications with sufficient privileges should be able to terminate the card.

B. *Extending UMLsec for these security properties*

Assuming the SUT has a variable state representing the card status, we choose to add to SUT model a statechart where its states represent the status of the card's life-cycle. We further assume there is a command set_status, only executable by privileged applications to change the card's status from one to another, and this is the event triggering all transitions in the model. To model failed attempts i.e change the card's status by an non privileged application, we allow internal transitions in a given state to represent them, for which the consequence is that a variable statusWord is affected with an error message.

Under these assumptions, a statechart in which from one state having the value `{status}` there are not only incoming but also *outcoming* transitions (with satisfiable guards, otherwise they would be superfluous) would be trivially violating the Security Property 1. This would contradict the property to test and could be the source of misinterpretations of the testing results. Potentially, it could also mean that the system specification is contradictory with respect to the wished security properties. To avoid this, we can extend UMLsec with a stereotype «locked-status» together with a tag {status} where a specific status can be defined. Semantically, a statechart annotated with this stereotype would require that there are not outgoing transitions from the state specified in {status}.

Similarly, we can define a stereotype «authorized» with two tags {status} and {permission}. This stereotype enforces that there exists no incoming transition to the status specified in {status} with a guard NOT containing {permission}. Under the assumption that in each transition from state to state we check for given application privileges, this stereotype would avoid having a model trivially violating Security Property 2.

These properties can be checked statically on UML statecharts since we are not aiming at verifying behavioural properties, but at ensuring a structural property as a precondition to the testing process. For example, the check performed by «authorized» on a statechart could be summarized by the Algorithm 1, where Status is the state corresponding to the value of {status} and the auxiliary function IncomingTransitions() returns all the incoming transitions relatively to that status.

Algorithm 1: Algorithm for stereotype 'authorized'

```
Transitions := Status.IncomingTransitions();
for $T$ in $Transitions$ do
    if $Permission \notin T.Guard$ then
        return false ;
    end
end
```

In a similar way define the algorithm for «locked-status», where we check whether Status.OutgoingTransistions() is empty.

C. Transformation Rules of UMLsec stereotypes to Schemas

At the end of the verification process we export the security property using Hoare triples, encapsulating the expected behaviour of the system after executing particular instructions that could potentially violate the property and make the system not to behave as expected. These is the base for generating Test Schemas (from which we generate automatically test sequences), thus they represent the link from UMLsec to testing.

Assume S is a set of instructions performed by an application in the system and T and Q are FOL formulas quantifying over system variables. Thus, let $\{T\}\ S\ \{Q\}$ be generalized exported formula by the UMLsec tool.

When taking into account the locked-status property, the formula can be exported as:

$$\{\mathsf{state} = \{\mathtt{status}\}\}\ \mathsf{set\_status}\ \{\mathsf{state} = \{\mathtt{status}\}\}$$

Let A an application of the system and the set of associated permissions is given by A.permissions. We assume that the set of instructions S does not include an operation that allows to select another application with different privileges:

$$T := \mathsf{state} \neq \{\mathtt{status}\} \wedge \{\mathtt{permission}\} \notin A.\mathsf{permissions}$$

$$Q := \mathsf{statusWord} = \mathsf{Error\_not\_Privilege\_}\{\mathtt{permission}\}.$$

Then, the authorized-status property can be exported as:

$$\{T\}\ \mathsf{set\_status}\ \{Q\}$$

Intuitively we can define a generic rule ϕ to transform the exported formula by UMLsec into a Schema. We define as follows:

```
$\phi(\{T\}\ S\ \{Q\})$:
    for_each $X from S,
    use any_operation any_number_of_times
    to_reach state_respecting $T$
    on_instance 'chosen_instance' then
    use $X at_least_once to_reach state_respecting $Q$
    on_instance 'chosen_instance'
```

We will consider two instantiations of this transformation for the given rule in the case of the Global Platform. In the next section, more details are given about the test sequences generated from schemas.

V. Validation

We have applied our methodology to a real case study: the Global Platform [2] in the context of the SecureChange project. The Global Platform is a non-profit organization involving over 60 industry members (including American Express, MasterCard, Visa, Nokia, Sun and Gemalto) that defines a publicly available smart card application management specification. The goal of this specification is to be hardware and operating neutral system and to cover a wide range of security critical industrial applications and therefore focuses on many security aspects. For example precise protocols for the communication of the card with an application provider or central server are defined aiming at guaranteeing confidentiality, integrity and authenticity aspects of both over-the-air and terminal connections. Moreover, Global Platform supports external software updates. Implementations of the specification with tailored applications include Financial, Mobile telecommunications, Government initiatives, Healthcare, Retail merchants and Transit domains.

The scope of our work is the management of the card life cycle, from the card's production until its destruction. We have created test models for the version on the Card Life Cycle Scope of Global Platform 2.1.1 respecting the assumptions mentioned in the previous section: each status of the statechart correspond to a state of the card and each transition's guard from state to state checks for certain application privileges.

A. Correctness Verification with UMLsec

We have verified a life-cycle testing model representing the expected behaviour of the card according to the Global Platform 2.1.1 Specification with respect to the stereotypes «locked-status» and «authorized» using the UMLsec verification tool, which we have extended for these new stereotypes. The UMLsec plug-in takes an ArgoUML statechart diagram in XMI format as an input and runs the proposed verification process on it. For illustrative purposes a fragment of a violating statechart w.r.t «locked-status» for the Global Platform 2.1.1 life cycle is shown on Fig. 8. In this statechart, there is a transition coming out from TERMINATED to SECURED, which is a contradiction to the desired property. This is reported to the user, who can correct the model accordingly and re-verify it.

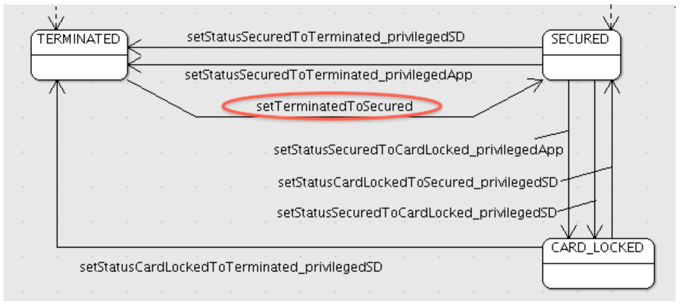
TERMINATED
SECURED
CARD_LOCKED
setStatusSecuredToTerminated_privilegedSD
setStatusSecuredToTerminated_privilegedApp
setTerminatedToSecured
setStatusSecuredToCardLocked_privilegedApp
setStatusCardLockedToSecured_privilegedSD
setStatusSecuredToCardLocked_privilegedSD
setStatusCardLockedToTerminated_privilegedSD

Figure 8. Example of a violating fragment of the GP 2.1.1 Life-Cycle modelled with ArgoUML

B. Schemas for the Security Properties

Using the transformation rules defined in Sec. IV-C, we have obtained the following test schema, that completely reflects the security property w.r.t. to the test intention we have defined. We needed only to define manually the existing model instance, for which we want to generate tests:

```
for_each $X from APDU_Set_status
use any_operation any_number_of_times to_reach
state_respecting (self.state = TERMINATED)
on_instance ”card” then
use $X at_least_once to_reach state_respecting
(self.state = TERMINATED) on_instance ”card”
```

Informally the *test intention* associated to this schema is:

- set the status of the card to TERMINATED;
- try all operations (to see if they behave as predicted by the model, i.e. by returning a status word of error).

The test intention for the authorized-status security property that we exhibit, is defined informally as a scenario to test the nominal case of failure of this security property:

- select any application without the Card Terminate Privilege
- set the card to a state different than terminated
- try to set the status of the card to TERMINATED, which results with an error code.

Then, we can create the corresponding *test schema* from the verified stereotype. We give here one possible transformed schema to cover it. However, sometimes is impossible to express one property with only one schema and that there is only one manner to express it.

```
 for_each $X from APDU_Set_status,
 use any_operation any_number_of_times to_reach
state_respecting (self.lcs->exists(lc : LogicalChannel|
lc.selectedApp.privileges.cardTerminate=false))
on_instance ”card” then
use any_operation any_number_of_times
to_reach state_respecting (self.state!=TERMINATED)
on_instance ”card” then
 use $X at_least_once to_reach
state_respecting
(self.StatusWord =
 APDU_SETSTATUS_ERROR_MustHaveTerminatePriv)
 on_instance ”card”
```

C. Discussion

Using the UMLsec approach we have verified our test model and permitted to the user to increase the confidence in it w.r.t the given properties in a realistic industrial scenario. When generating the tests we can be sure that they are consistent w.r.t. the property. The schema we have created for the *locked-status* property sets the card into the state *TERMINATED*. Then finds different manners to stay in the same state using the *APDU_Set_Status* command. The exit code of this command results with an error code. The error code corresponds to the one that the card is already *terminated*. Thus, we obtain 13 different tests for this property.

For the *authorized-status* property, using the schema we have generated 13 tests, also. Each test selects first an application without *terminated privilege* and then, the schema allows to select states different to *TERMINATED*. Afterwards, the generator tries to reach the *TERMINATE* state. It results with an error code, that the application has not the privilege to terminate the card. Then, these tests are ready to be exported and used for testing the program w.r.t the security property. But, here the created schema does not select each state different to *TERMINATED*. It selects only one among the possible list of states. To include this possibility, we need to define another variable, for example *$Y*, that will iterate the wanted states. Or we can iterate a set of functional behaviors (expressed in the model by using the keyword REQ and AIM) that we are interested in, and create tests that reflect the security property. But currently, with UMLsec we cannot identify the special tags used for testing **REQ** and **AIM**. Our goal is thus, to adapt the exported FOL formulas and add rules that will enable the transformation to benefit fully from the schema language expression power.

VI. Related work

Tests can be obtained by means of a model-checker in the shape of traces of a model that contradict the properties (see [7], [8] for example). M. Dwyer, to facilitate the use of temporal properties by validation engineers, has identified in [9] a set of design patterns that allow for expressing as temporal properties a set of temporal requirements frequently met in industrial studies.

Input/Output Symbolic or Labelled Transition Systems have frequently been used to specify test purposes [10][11]. These formalisms specify sequencing of actions by using the same set of actions as the model, and possess two trap states named *Accept* and *Refuse*. The *Accept* states are used as end states for the test generation while the *Refuse* states allow for cutting the traces not wanted in the generated tests. These formalisms are for example used in tools such as TGV [10], STG [12], TorX [13], Agatha [14].

Some approaches are based on the definition of scenarios for the test, e.g. in [15][16], where test cases are issued from UML diagrams as a set of trees. The scenarios are extracted by a breadth-first search on the trees. A similar approach is that implemented in the tool *Telling TestStories* [17], based on defining a test model from elementary test sequences made of an initial state, a *test story* and test data.

Work close to ours is done for Tobias tool[18], which provides a combinatorial unfolding of some test schemas. The schemas are sequences of patterns made of operation calls and parameter constraints. They are unfolded independently from any model, thus the tests obtained have to be instantiated from a model. In [19], a connection between Tobias and the UCASTING tool is studied to produce instantiated tests. UCASTING [20] allows for valuating sequences of operations that are not or only partially instantiated from an UML model. In [21], close to the previous work, authors use scenarios based on regular expressions, to enrich the test generation test suite produced by the Smartesting Test Designer Tool, which cannot generate tests for dynamic system properties. Thus, they propose scenario language that can be used by the validation engineer, who basing on his experience can produce interesting scenarios to generate tests involving complex situations. Their work, is an adaptation for UMl base on the work done in [5]. This language was designed during a project (RNTL POSE) dedicated to testing the conformance of a system to a security policy. Its conception has been guided by the experience of security practitioners, resulting in a language that well serves the aim of testing security properties. Indeed, considering both actions to perform and states to reach is the way a security engineer thinks of testing a security issue.

[22] proposes an approach for systematically generating test sequences for security properties in a model-based way that can be used to test the implementation for vulnerabilities.

The originality of our Schema Language with respect to these related approaches can be summarized in three points as discussed in Section III: scientific point of view, technological point of view and its expressivity. This language allows a validation engineer to benefit plainly from its good knowledge of the model and to explicitly use all artifacts of the model (such as objects names).

Chetali in [23] has pointed out the need to have an automated approach allowing to prove security properties on a system and to write scenarios to produce functional tests as well in the smart-card context. To the extent of our knowledge there is however so far no published work on consistent model-based testing for security properties, neither for smart-cards or in general.

VII. Conclusion and Future works

This paper presents a model-based technique for test generation from schemas for UML/OCL models and its integration with the UMLsec verification approach, in order to gain more confidence in models for testing with respect to security properties and to facilitate the property expression from stereotypes into test schemas.

We enhance the verification activities to test models, and the kind of properties that are verified on the model. In addition, we propose that both approaches are used by the same actor (the validation engineer).

We have illustrated our approach by applying it to the Global Platform 2.1.1 life-cycle and by considering two critical security properties that an actual implementation should meet. We reported also on the existing tool support of the proposed methodology.

As underlined in the discussion part our integrated approach for now is limited only to two security properties, thus our goal is to generalize it. It is in our perspective to adapt more the exported formulas and enrich the transformation rules, thus to be able to benefit as much as possible of the schema language expression power. We are limited by the schema language also, for example we are not able to use explicitly the operation parameters. We focus also on its improvement.

Another objective in the context of the EU project SecureChange is to extend this approach to smart-card specifications under evolution and to deal with regression testing. On the one hand, it is challenging to guarantee correctness of an evolving UMLsec without having to re-verify it from scratch, but re-using as much as possible the unchanged fragment. On the other hand, it is important to manage the testing of the evolving life cycle thus creating dedicated test suites: regression, evolution, stagnation and deletion to test as given in [24] by taking into account evolution and management of security properties w.r.t. to the specification.

Acknowledgment

This research is supported by the EU project Security Engineering for Lifelong Evolvable Systems (Secure Change, ICT-FET-231101) http://www.securechange.eu/.

References

[1] J. Jürjens, *Secure Systems Development with UML*. Springer-Verlag, 2005.

[2] "Global platform specification," http://www.globalplatform.org/specificationscard.asp, May 2011.

[3] "UMLsec tool," http://umlsec.de, May 2011.

[4] J. Jürjens and P. Shabalin, "Tools for secure systems development with UML," *Intern. Journal on Software Tools for Technology Transfer*, vol. 9, no. 5–6, pp. 527–544, Oct. 2007, invited submission to the special issue for FASE 2004/05.

[5] P.-A. Masson, M.-L. Potet, J. Julliand, R. Tissot, G. Debois, B. Legeard, B. Chetali, F. Bouquet, E. Jaffuel, L. Van Aertrick, J. Andronick, and A. Haddad, "An access control model based testing approach for smart card applications: Results of the POSÉ project," *JIAS, Journal of Information Assurance and Security*, vol. 5, no. 1, pp. 335–351, 2010.

[6] J. Julliand, P.-A. Masson, R. Tissot, and P.-C. Bué, "Generating tests from B specifications and dynamic selection criteria," *FAC, Formal Aspects of Computing*, vol. 23, pp. 3–19, 2011.

[7] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," *SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, pp. 146–162, 1999.

[8] P. E. Ammann, P. E. Black, and W. Majurski, "Using model checking to generate tests from specifications," *Formal Engineering Methods, International Conference on*, p. 46, 1998.

[9] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *ICSE'99, 21st international conference on Software engineering*, LA, California, United States, 1999, pp. 411–420.

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

[11] L. Frantzen, J. Tretmans, and T. Willemse, "Test generation based on symbolic specifications," in *FATES 2004, Formal Approaches to Software Testing*, ser. LNCS, J. Grabowski and B. Nielsen, Eds., vol. 3395. Springer, 2005, pp. 1–15.

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

[13] G. J. Tretmans and H. Brinksma, "TorX: Automated model-based testing," in *1st Europ. Conf. on Model-Driven Software Engineering*, Nuremberg, Germany, Dec. 2003, pp. 31–43.

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

[15] A. Bertolino, E. Marchetti, and H. Muccini, "Introducing a reasonably complete and coherent approach for model-based testing," *Electron. Notes Theor. Comput. Sci.*, vol. 116, pp. 85–97, Jan. 2005.

[16] F. Basanieri, A. Bertolino, and E. Marchetti, "The Cow_Suite approach to planning and deriving test suites in UML projects," in *UML'02, 5-th int. conf. on the UML language*, ser. LNCS, vol. 2460, London, UK, 2002, pp. 383–397.

[17] M. Felderer, R. Breu, J. Chimiak-Opoka, M. Breu, and F. Schupp, "Concepts for Model-based Requirements Testing of Service Oriented Systems," in *Proceedings of the IASTED International Conference*, vol. 642, 2009, p. 018.

[18] 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*, 2007, pp. 535–536.

[19] O. Maury, Y. Ledru, and L. du Bousquet, "Intégration de TOBIAS et UCASTING pour la génération des tests," in *ICSSEA'03, 16th Int. Conf. on Software and Systems Engineering and their Applications*, Paris, France, 2003.

[20] L. Van Aertryck and T. Jensen, "UML-CASTING: Test synthesis from UML models using constraint resolution," in *AFADL'03*, 2003.

[21] K. Cabrera Castillos and J. Botella, "Scenario based test generation using test designer," in *SCENARIOS'11, 1st Int. Workshop on Scenario Based Testing – co-located with ICST'2011*. Berlin, Germany: IEEE Computer Society Press, Mar. 2011, to appear.

[22] J. Jürjens and G. Wimmel, "Formally testing fail-safety of electronic purse protocols," in *16th International Conference on Automated Software Engineering (ASE 2001)*, 2001, pp. 408–411.

[23] B. Chetali, "Security testing and formal methods for high levels certification of smart cards," in *Proceedings of the 3rd International Conference on Tests and Proofs*, ser. TAP '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 1–5. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02949-3_1

[24] E. Fourneret, F. Bouquet, F. Dadeau, and S. Debricon, "Selective test generation method for evolving critical systems," in *REGRESSION'11, 1st Int. Workshop on Regression Testing - co-located with ICST'2011*. Berlin, Germany: IEEE Computer Society Press, Mar. 2011, to appear.