Managing Evolution by Orchestrating Requirements and Testing Engineering Processes

Federica Paci
and Fabio Massacci
DISI
University of Trento
Email:{federica.paci, fabio.massacci}@unitn.it

Fabrice Bouquet
and Stephane Debricon
Laboratoire d'Informatique, Université de France-Comté
{fabrice.bouquet,stephane.debricon}@lifc.univ-fcomte.fr

***Abstract*—Change management and change propagation across the various models of the system (such as requirements, design and testing models) are well-known problems in software engineering. For such problems a number of solutions have been proposed that are usually based on the integration of model repositories and on the maintenance of traceability links between the models.**

We propose to manage the mutual evolution of requirements models and tests models by orchestrating processes based on a minimal shared interface. Thus, requirement and test engineers must only have a basic knowledge about the "other" domain, share a minimal set of concepts and can follow their "own" respective processes. The processes are orchestrated in the sense that when a change affects a concept of the interface, the change is propagated to the other domain. We illustrate the approach using the evolution of the GlobalPlatform® standard.

I. Introduction

Change management is a well known problem in software engineering and in particular the change propagation across the various models of the system (such as requirements, design and testing models). For such problem a number of solutions have been proposed that are usually based on the idea of integrating model repositories and on the maintenance of traceability links.

However, such solutions are not always feasible when the development of various design artifacts is outsourced to subcontractors. Here is a concrete example drawn from multi-application smart cards domain: test engineer in charge of certifying that the card is secure might not have access to system code (and the relative design models) simply because he is part of a third-party company to which the certification has been outsourced. Still, the test engineer must coordinate with the requirement engineer to show that requirements are achieved. In any security engineering process, the cooperation between test engineer and requirement engineer is of primary importance. For this process to work smoothly in presence of changes we need to orchestrate the work of the requirement engineer with the work of the test engineer. In many cases, requirements evolution can have impact on a confined part of the system. In such cases it would be beneficial to clearly identify only those parts of the system that have been affected by the evolution and that need to be re-tested for compliance with requirements. In this way re-running all test cases is avoided because it is possible to identify which new test case need to be added to the test suite and which test cases can be discarded as obsolete.

To address these issues, we propose a framework for managing the impact of changes happening at requirement level on testing generation process and vice versa. The key features of the framework are *model-based traceability* by *orchestration* and *separation of concerns* between the requirement and the testing domain. Separation of concern allows the requirement engineer to have very little knowledge about the test (process, modeling or generation) domain, and similarly for the test engineer. They only share a minimal set of concepts which is the *interface* between the requirement and the testing frameworks. Moreover, both engineers simply need to follow their respective processes (i.e., requirement engineering and testing generation process) separately. The processes are *orchestrated* in the sense that when a change affects a concept of the interface, the change is propagated to the other domain. The interface also supports traceability between the requirement and test models through mapping of concepts in the two domains.

For sake of concreteness, we instantiate the requirement framework to SI* [13] and the test framework to SeTGaM [11]. However, our approach is independent from the specific requirement and testing frameworks that are adopted, and can thus be applied to other competing instantiations if these have mapping concepts similar to the ones we propose in Section V.

In the next section we introduce the evolution of the GlobalPlatform standard for multi-application smard cards that will be our running example. Then we describe how changes are managed at the requirements level (§III) and at the level of model-based testing (§IV). Sec.V illustrates the conceptual interface. Sec. (§VI) presents the overall orchestrated process while Sec.VII illustrates the application of the process to the case study. Finally we discuss related works in Sec. VIII and conclude the paper in Sec IX.

II. GlobalPlatform Evolution

The most popular solution for smart cards now is *GlobalPlatform* (GP) [1] on top of Java Card [20]. Loosely speaking GP is a set of card management services such as loading,

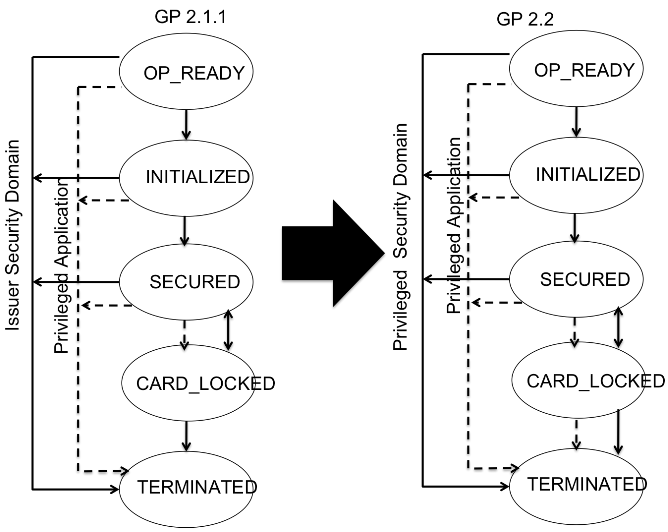


Fig. 1. Card Life Cycle in GP-2.1.1 and GP-2.2

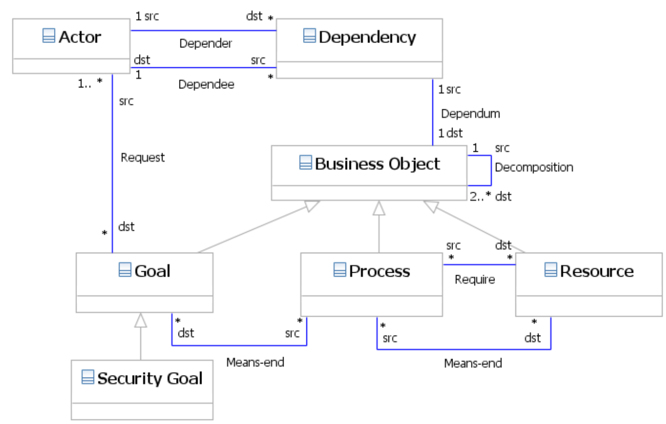


Fig. 2. SI* conceptual model

enabling, or removing applications. The GP specification describes the card life cycle and the GP components that are authorized to perform a transition in the life cycle: Security Domains and Applications. Security Domains act as the on-card representatives of off-card authorities such as the Card Issuer or Application Providers.

The card life cycle begins with the state OP_READY. Then the card can be set to the states INITIALIZED, SECURED, CARD_LOCKED and TERMINATED that is the state where the card cannot longer be used. During the evolution of card life cycle between versions 2.1.1 and 2.2 of GP specification as illustrated in Figure 1 a number of changes took place giving authority to perform transitions to different stakeholders. In GP-2.1.1 a privileged application can terminate the card from any state, except CARD_LOCKED. Additionally, a privileged application can lock the card by changing card state from SECURED to CARD_LOCKED. However, only the issuer of the card can move it to TERMINATED state. A security domain is a special kind of privileged application, and therefore, has exactly the same behavior of privileged application in terms of card lifecycle management. In GP-2.2 two main changes with respect to GP-2.1.1 are introduced: a) a privileged application can terminate the card from any state if the application has appropriate privileges; b) any privileged security domain can trigger all card life cycle transitions while in GP-2.1.1 only the issuer security domain can do that.

III. Change Management for Evolving Requirements

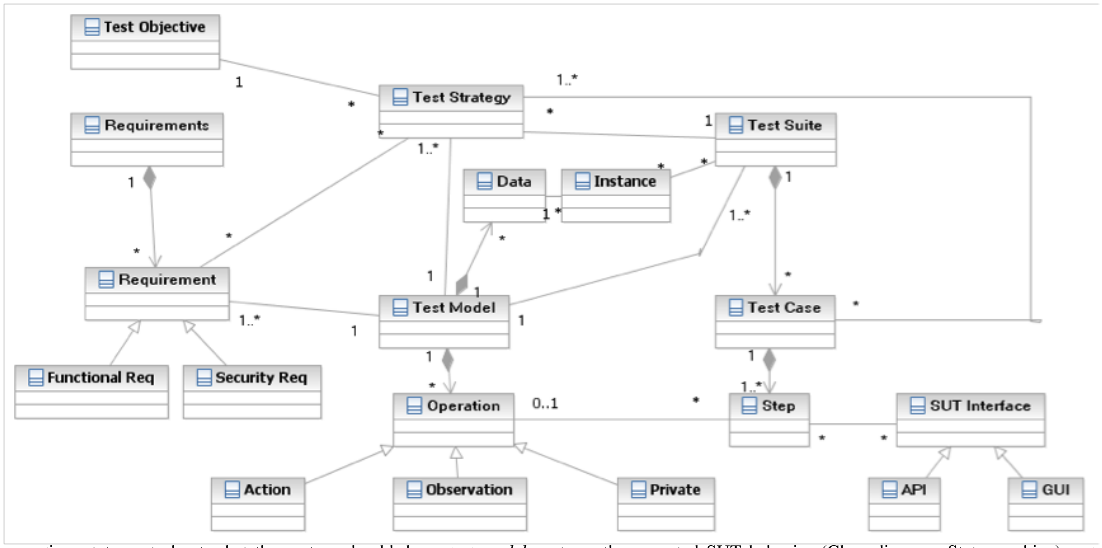
SI* [13] is a modeling framework which supports security requirement analysis. SI* is part of a complete security methodology, which aims at analyzing and modeling organizational settings and its security and dependability requirements. As illustrated in Fig. 2, the SI* language [1] is founded on the concepts of *actor*, *goal*, *resource* and relations such as *AND/OR decomposition* and *means-end*. An actor is an entity which has intentions, capabilities, and entitlements; a *goal* captures a strategic interest that is intended to be fulfilled; a *resource* is an artifact produced/consumed by a goal; *AND/OR decomposition* is used to refine a goal; *means-end* identifies goals that provide means for achieving another goal or resources produced or consumed by a goal/task. The requirement analysis is an iterative process that aims at refining the stakeholders' goals until all high-level goals are achieved. We use the analysis proposed by Asnar et al. [2] where the keyword SAT denotes that the evidence is in favor of the achievement of the the goal and DEN denotes that the evidence is against it.

The evolution of a requirements model can be triggered by a *change request* that can be placed by stakeholders, or it can be a reaction to a previous change, or caused by external circumstances and merely observed. A change in a SI* model can be represented as a transition of the model from a *pre-state* model, to a *post-state* model. A change in a pre-state model can be detected and its impact assessed by means of an automated analysis. The analysis evaluates the impact of the change on the security principles that should be satisfied by the system. These security principles are declaratively specified by an extensible set of *security patterns*. A security pattern expresses a situation (a graph-like configuration of model elements) that leads to the violation of a security property. Whenever a new match of the security pattern (i.e. a new violation of the security property) emerges in the model, it can be automatically detected and reported. The specification of security patterns may also be augmented by automatic remedies (i.e. templates of corrective actions) that can be applied in case of a violation to fix the model and satisfy the security property once again.

IV. Change Management for Evolving Tests

As testing generation process we consider SeTGaM [12]. The model-based testing generation process starts by the design of the test model by the test architect: the model

[1] We only mention concepts that are relevant to this work



Legend: A *requirement*is a statement about what the system should do; a *test model* captures the expected SUT behavior (Class diagram, State machine); a *test case* is a finite sequence of test steps; a *test intention* is a user's view of testing needs; a *test suite* is a finite set of test cases; a *test script* is executable version of a test case; a*test step*is an operation's call or verdict computation; a *test objective*is an high level test intention.

Fig. 3. Basic testing concepts

should describe the expected behaviour of the system under test (SUT). Then, the test model is used to generate the test cases and the coverage matrix, which relates the tests with the covered model elements. The tests are then exported, or published, in a test repository and then executed. After the test execution, test results and metrics are provided.

A test model consists of three different types of UML diagrams (Fig. 3). First, a class diagram describes the data model, namely the set of classes that represent the entities of the system, with their attributes and operations. Second, an object diagram provides a given instantiation of the class diagram together with the test data (i.e. the objects) that will be used as parameters for the operations composing the tests. Finally, the behavior of the system is described by two (complementary) means: a state chart diagram, and/or OCL constraints associated with the operations of the class diagram. The test coverage of system requirements and test objectives is achieved by using the tags @REM and @AIM to annotate the OCL code.

When evolution occurs, the status of the test changes depending on the impact of the evolution on the model elements covered by the test case. Evolution of status is defined by considering two versions of the test model, M and M', in which addition, modification or deletion of model elements (operations, behaviors, transitions, etc.) have been performed. Test may have a status *new* in case of a newly generated test for a newly introduced target.

If none of the model elements covered by the test is impacted, the test is run as is on the new version of the model M', without modifying the test sequence. The test is thus said to be *reusable*. More precisely, there are two cases: *unimpacted* and *re-executed*. A test is unimpacted if the test sequence is identical to its previous version, and the covered requirements still exist. The test is re-executed if it covers impacted model elements, but it can still be animated on the new version of the model without any modification.

If a test covers model elements impacted by the evolution of M to M', and if the test cannot be animated on M' the test becomes *obsolete*. There are two cases: either the target represents deleted model elements, and thus the test does not make any sense on M' and it is said to be *outdated*; or, the test fails when animated on model M' (e.g. due to a modification of the system behaviour), and it is then *failed*. When the test case operations can be animated but produce different outputs, a new version of the test is created in which the expected outputs (i.e. the oracle) are updated w.r.t. M'. In this case the tests have the status *updated*. When the test case operations can not be animated as is in the first version of the test, a new operation sequence has to be computed to cover the test target. In the latter case, tests have status *adapted*.

To determine the status of a test when evolution takes place, the SeTGaM approach relies on dependency analysis that is performed to compute the differences between the models, and their impacts on test cases. We have four different classification suites.

- *evolution test suite* contains tests classified as new and adapted;
- *regression test suite* contains tests classified as unimpacted and re-executed;
- *stagnation test suite* contains tests classified as outdated and failed.
- *deletion test suite* contains tests, that come from the stagnation test suite from the previous version of the model.

V. Conceptual Interface

The orchestration of the requirements engineering process and the test generation process is based on the identification

TABLE I
REQUIREMENTS COVERAGE

Test Classification	Test Status	Test Result	Achievement Level
Evolution	New, Adapted,Updated	Pass	Fulfill
Regression	Unimpacted, Re-Executable	Pass	Fulfill
Evolution	New, Adapted,Updated	Fail	Deny
Regression	Unimpacted, Re-Executable	Fail	Deny
Stagnation	Outdated, Failed	Pass	Deny
Stagnation	Outdated, Failed	Fail	Fulfill

of a set of concepts that are shared or mappable in the two domains: a *shared concept* is a concept that has the same semantics in both domains while a *mappable concept* is a concept that is related to one in the other domain. We identify one shared concepts that is *Requirement*. A Requirement in both domains represents a statement by a stakeholder about what the system should do. The concepts of *Actor*, *Goal*, *Process* are mapped on the Test Model. In particular, the concept of Actor is used to identify the system under test (SUT). The concepts of Goal and Process are used by the test engineer to build the different types of Test Models. The goals and processes in the Requirement Model are identified by a unique name that is used to annotate the State Machine of the Test Model and the OCL code in order to achieve traceability between the Requirement Model and the Test Model.

Mapping of a test case's result and status to a requirement achievement level allows the requirement engineer to quantify the requirement *coverage* after evolution. This correspondence is reported in Table I: if the status of a test case after evolution is *new*, *adapted* or *updated*, and the test result is *pass* the requirement covered by the test case is fulfilled while it is denied (i.e. we have evidence that has not been achieved) if the test result is *fail*. A subtle case is present when a test case is part of the stagnation suite (i.e. *obsolete*) and the test result is *fail*. Here the test covers requirements that have been deleted from the model and thus the corresponding behavior should no longer be present (for example a withdrawn authorization) so failing the test shows that the unwanted behavior is no longer present.

We also consider *completion* indicators for the change propagation process which indicates whether all changes in the requirement model have been propagated to the test model. Table II summarizes the mapping between Goal and Process in the requirement model and the Test Model element. In a nutshell we say that the change propagation process has been completed if:

- for each new or modified model element in the ReM model a new test case and an adapted are added to the evolution test suite,
- for each model element not impacted by evolution there is a re-executable test case in the regression test suite,
- for each model element deleted form the model there is an obsolete test case in the stagnation test suite.

TABLE II
COMPLETION OF CHANGE PROPAGATION

Change in ReM Model	Test Status	Test Suite
New Element (Goal, Process)	New	Evolution
Modified Element (Goal, Process)	Adapted	Evolution
Model Element Not Impacted	Re-Executable	Regression
Deleted Element	Obsolete	Stagnation

VI. ORCHESTRATED CHANGE MANAGEMENT PROCESS

The orchestration of the test and the requirements engineering processes is based on the principle of *separation of concern*: the two processes should be understood as separate processes with their own iterations, activities and techniques for managing change.

The UML activity diagram of Fig. 4 gives a high-level overview of the orchestrated process.

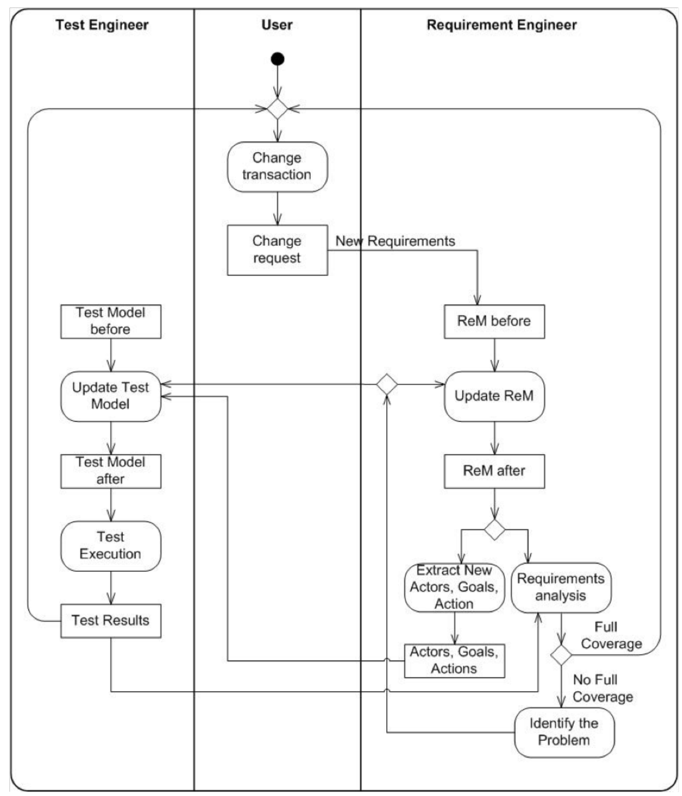


Fig. 4. Change Management Process
Legend: The diagram is divided into three partitions to distinguish between the activities and objects under the control of users, requirement engineers, and test engineers. A user is typically the client commissioning the testing and may be the owner of the SUT. In the diagram, the diamonds specifies branching of the sequence of activities.

Interactions are triggered by the change request from a stakeholder of the system. To illustrate the process, we start from changes in the requirement domain:

1) ***Update ReM***. The requirement engineer uses the previous requirement model (ReM-before) and the change request to update the model, producing ReM-after.
2) ***Extract New Actors, Goals, Actions***. Based on the ReM-after, new actors, goals and processes are extracted if relevant and provided to the test engineer.

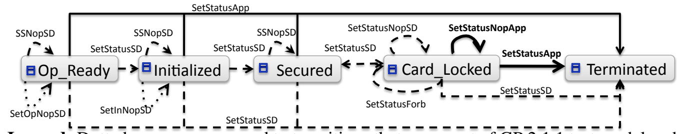
3) ***Update Test Model***. Receiving the extracted actors, goals and processes the test engineer based on the traceability links between the ReM and the test model (TeM), identify the part of the TeM that are affected by the changes in the ReM. The test engineer thus updates the TeM and the test suite for the updated TeM (See Sec.IV for the test suite generation).
4) ***Test Execution***.Then, the test engineer executes the new test suite. The test engineer returns the test results to the requirement engineer in a suitable table. The table shows for each test case in the test suite the number of times the test has been executed, the status of the test after evolution, the TeM element and the requirements/goals covered by the test case, and the test result.
5) ***Requirement Analysis***. The requirement engineer evaluates the matrix for each requirement covered by the test and translates the test results into a level of achievement (partial satisfaction/denial or full satisfaction/denial) for the low level requirements. Once the requirement engineer gets the achievement levels for low level requirements, he can run the requirement analysis to determine the level of achievement also for top-level requirements.
6) ***Identify the problem***. If some of the requirements are not fulfilled, the requirement engineer must identify the problem.
   a If there is a problem with the ReM, the requirement engineer must backtrack and search for an alternative way of updating the ReM when considering the change request that was initially passed from the user.
   b If there is a problem with testing, the test engineer must determine whether there is the need to generate new test cases or not.

VII. Application to Case Study

We first illustrate how a change from the GP-2.1.1 to the GP-2.2 requirement model is propagated to the test model and thus how the test suite for the GP-2.2 test model is generated. Then, we show how the test classification for the test model of GP-2.2 can be used to evaluate the *completion* of the change propagation process.

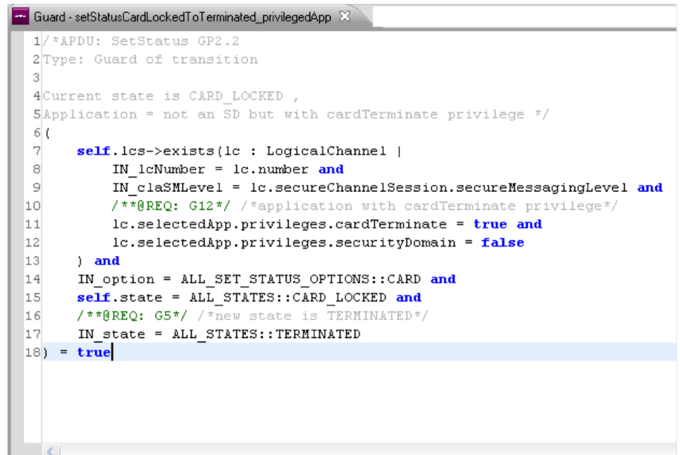
Fig. 5 shows the SI* model for GP-2.1.1 and 2.2. The main actors are Global Platform Environment (OPEN), Privileged application (App), Privileged Security Domain (SD), and Issuer Security Domain (ISD). We only focus on the card lifecycle transition to TERMINATED state that is the one impacted by the evolution. This transition is represented by the goal G5: in the GP-2.1.1 model the goal G5 is AND decomposed in two subgoals G13 and G11' the latter further decomposed into subgoals in goals G8' and G12'; in the GP-2.2 model the goal G5 has only G12 as subgoal (labeled in grey).

Fig. 6 represents the test model for GP-2.1.1 and GP-2.2: transitions *SetOpNopSD* and *SetInNopSD* (dotted arrows) are removed in GP-2.2 because they are associated with the deleted goal G9'. The transitions *SetStatusNoApp* and *SetStatusNoApp* (bolded arrows) between the states *Card_Locked* to



Legend: Dotted arrows correspond to transitions that were part of GP 2.1.1 test model and has been removed in the model of GP 2.2, bolded arrows represent new transitions, dashed arrows correspond to modified transitions, while full arrows correspond to transitions not impacted by evolution of requirements.

Fig. 6. Test Model for GP specifications 2.1.1 and 2.2



Guard - setStatusCardLockedToTerminated_privilegedApp

```
/*APDU: SetStatus GP2.2
Type: Guard of transition

Current state is CARD_LOCKED ,
Application = not an SD but with cardTerminate privilege */
(
    self.lcs->exists(lc : LogicalChannel |
        IN_lcNumber = lc.number and
        IN_claSMLevel = lc.secureChannelSession.secureMessagingLevel and
        /**@REQ: G12*/ /*application with cardTerminate privilege*/
        lc.selectedApp.privileges.cardTerminate = true and
        lc.selectedApp.privileges.securityDomain = false
    ) and
    IN_option = ALL_SET_STATUS_OPTIONS::CARD and
    self.state = ALL_STATES::CARD_LOCKED and
    /**@REQ: G5*/ /*new state is TERMINATED*/
    IN_state = ALL_STATES::TERMINATED
) = true
```

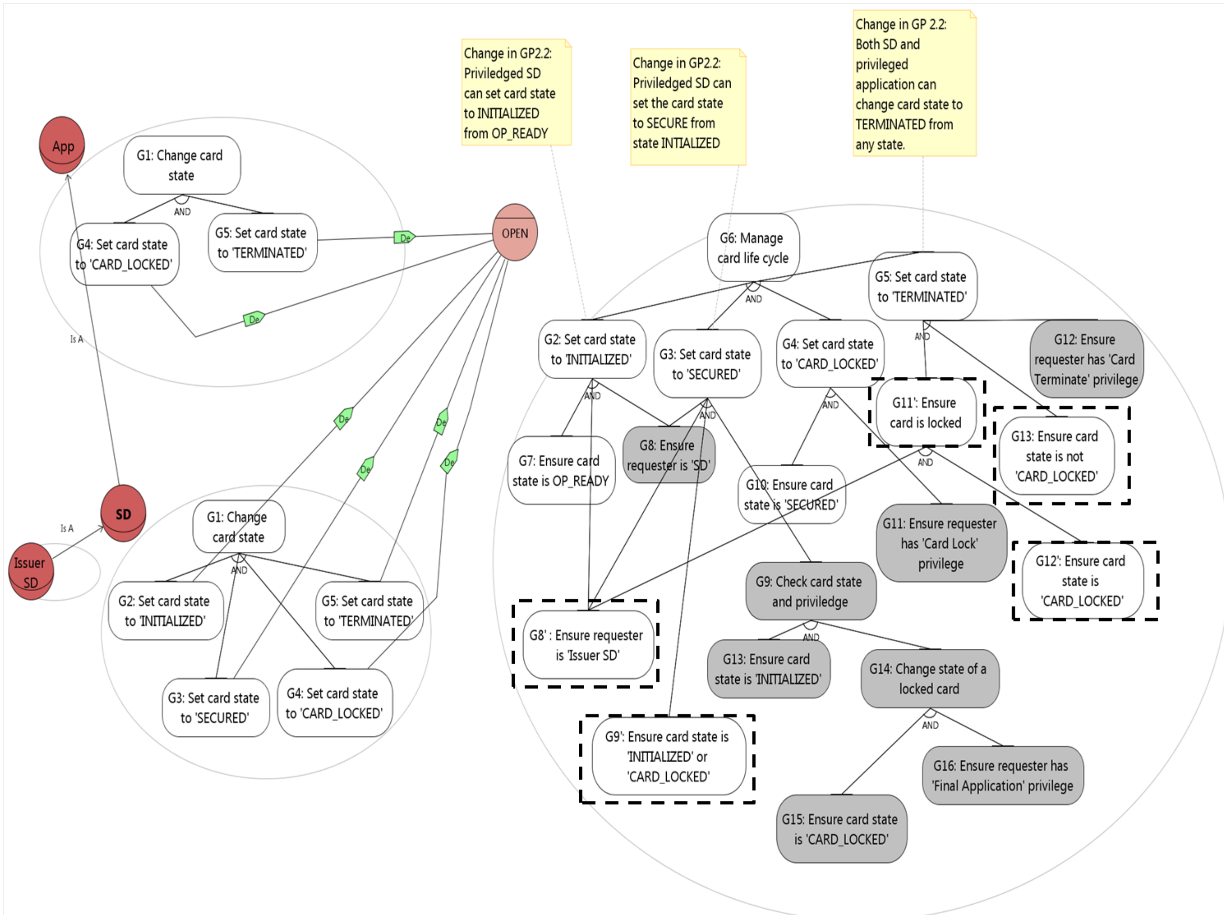
Legend: OCL code for the transition *setStatusApp* from CARD-LOCKED to TERMINATED requested by an Application with cardTerminate privilege. The code is annotated with the identifiers of the goals G5 (@REM G5) and G12 (@REM G12) covered by the test cases *Test 3* and *Test 5*.

Fig. 7. OCL code for SetStatus APDU command setting card state to TERMINATED

Terminated are added to the test model because in GP-2.2 requirement model the decomposition of G5 goal is changed.

The traceability link between the goals in Fig. 5 and the transitions in the test model of Fig. 6 is illustrated in Fig. 7 representing the dynamic behavior of the transition *setStatusApp*. In order to trace the transition to goals G5 and G12 , the OCL code is annotated with the tags @REM G5 and @REM G12 referring the goals G5 and G12. Based on the traceability link between goals and test model transitions we can generate the test suites for GP-2.1.1 and GP-2.2 that are illustrated respectively in Tab. III and Tab. IV. The tables only focus on the test cases for transitions from *Card_Locked* to *Terminated* states: *SetStatusNopSD* and *SetStatusNopApp* that correspond to *setStatus* command performed by a Security Domain and Application with no Terminate privilege, *SetStatusSD* and *SetStatusApp* correspond to *setStatus* command performed by a Security Domain and Application with Terminate privilege and *SetStatusForb* corresponding to a Security Domain and Application with no Card Locked privilege. For example, since a new goal G12 has been added to the SI* model for GP-2.2 two new test cases *Test 3* and *Test 5* covering G12 and its top goal G5 have been added to the test suite.

With respect to the completion of the change propagation process, we can see that the changes in the card life cycle related to the state TERMINATED has been propagated from the requirement model to the test model: two new test cases



Legend: Goals surrounded by dashed rectangles correspond to requirements that belong only to G-2.1.1 model, goals in grey are new requirements related to the card life cycle introduced in version 2.2, and goals in white are goals corresponding to requirements that are present in both versions.

Fig. 5. Requirement Model for GP specs 2.1.1 and 2.2

TABLE III
TEST SUITE FOR GP-2.1.1

Transition Covered	Test	Requirement
SetStatusForb	$Test_1$	$G_6, G_{11'}$
SetStatusNopSD	$Test_2$	$G_5, G_{8'}, G_{11'}$
SetStatusSD	$Test_3$	$G_5, G_{8'}, G_{11'}, G_{12'}$

TABLE IV
TEST SUITE FOR GP-2.2

Transition Covered	Test	Requirement	Status
SetStatusForb	$Test_1$	G_6, G_{11}	Re-executed
SetStatusNopSD	$Test_2$	G_5, G_8, G_{11}	Updated
SetStatusSD	$Test_3$	$G_5, G_8, G_{12}, G_{15}, G_{16}$	Updated
SetStatusNopApp	$Test_4$	G_5, G_{11}, G_{15}	New
SetStatusApp	$Test_5$	$G_5, G_{12}, G_{15}, G_{16}$	New

Test 4 and *Test 5* and two updated test cases *Test 2* and *Test 3* corresponding to G5 and its subgoals has been included in the evolution test suite. *Test 4* and *Test 5* correspond to an Application executing *setStatus* command without and with Terminate privilege respectively, while *Test 2* and *Test 3* are related to the execution of the *setStatus* command performed by a Security Domain without and with Terminate privilege.

VIII. Related Works

Change management is well known for being a difficult and costly process. However, only some requirement engineering proposals provide support for handling change propagation and for change impact analysis. Goal-oriented approaches such as KAOS, Secure Tropos, and Secure i* [27], [13], [18] provide good support for change propagation because they are based on goal models which explicitly show relationships and dependencies between goals, and also support the modeling and the analysis of dependencies between agents. Tanabe et al. [25] propose an approach to requirements change management that supports version control for goal graphs and impact analysis of adding and deleting goals. Chechik et al. [6] propose a model-based approach to propagate changes between requirements and design models that utilize the relationship between the models to automatically propagate changes. Hassine et al. [14] present an approach to change impact analysis that applies both slicing and dependency analysis at the Use Case Map specification level to identify the potential impact of requirement changes on the overall system. Lin et al. [17] propose capturing requirement changes as a series of atomic changes in specifications and using algorithms to relate changes in requirements to corresponding changes in specifications.

Other works relevant to change propagation are the one about the *generation and maintenance of traceability links*, and *model-to-model transformations*. Most of the works on the maintenance of traceability matrix focus on the recovery of traceability links between requirements and artifacts of different types [19], [22], [29], and on methods and CASE tools for the representation and management [16], [29], [15] of traceability links.

Model-to-model transformation techniques such as VIATRA2 [28], QVT [24], and ATLAS [5] support change propagation by means of bidirectional incremental model synchronization. With respect to change management in test engineering, several works about regression testing have been proposed.There are two kinds of regression testing: *code-based* regression testing and *specification-based* regression testing.

Code-based testing is limited to unit testing, and is mainly applied to concurrent programs ([9]). At program level, in [8], the authors describe how to select a tests' subset to be used for regression testing. This subset is defined by using data coverage of the test w.r.t. the changes that occurred in a program.

In the specification-based regression testing field, a variety of techniques can be found, based on various selection criteria, such as requirement coverage [7]. In [26] the authors use EFSM models for safe regression technique based on dependence analysis. They select test cases and compute the regression test suite by identifying three types of elementary modifications applicable to a machine (addition, deletion, modification of a transition). Our approach is grounded on these principles, but improves them by keeping the test history. In addition, we consider three test suites fulfilling different purposes. In [10] the authors propose a methodology to identify impacted part of the model. A list of all depending operations is created for each operation modification. They identify all parts of dynamic UML diagrams in which the behaviour of this operation can be found. This approach can be seen as a variation of the approach proposed here, that does necessarily consider statecharts diagrams. The authors present in [23] a regression testing approach based on Object Method Directed Acyclic Graph (OMDAG) using class diagrams, sequence diagrams and OCL code. They consider that a change in a path of the OMDAG affects one or more test cases associated to the path. They classify changes as NEWSET, MODSET and DELSET, which can be identified as the elementary modifications we consider. In [21] a model-based selective regression technique is described, based on UML sequence diagrams and OCL code used to describe the system's behavior. In [4] the author describe a regression testing method using UML class, sequence diagrams and use case diagrams. Changes in actions are collected by observing sequence diagrams, while changes in the variables, operations (OCL), relationships and classes are collected by comparing class diagrams.

IX. Conclusion and Future Works

In this paper we have proposed a novel framework for propagating changes between requirement and testing models. The framework supports *model-based traceability* by means of *orchestration* and *separation of concerns* between the requirement and the testing domain.

We are planning to implement the approach into the SeCMER tool for requirements evolution management developed in the context of SecureChange European project [2]. The core of the tool is the EMF-IncQuery [3], an incremental EMF model query engine for change-driven transformations. Thus a first step for tool support is to specify the mappings between requirements and test conceptual models in the declarative model query language of EMF-IncQuery. We would like also to investigate the applicability of our approach to security testing. Finally, we want to run a user study with practitioners from industry to assess the perceived usefulness of the framework in an industrial security engineering process.

Acknowledgment

This work has been partially funded by the EU-FP7-ICT-IP-SecureChange (Grant No.231101), and EU-FP7-ICT-NoE-NESSoS project (Grant No.256980).

References

[1] Global platform specification. http://www.globalplatform.org, May, 2011. v.2.1.1 available in March'03 and v.2.2 available in March'06.

[2] Y. Asnar, P. Giorgini, and J. Mylopoulos. Goal-driven risk assessment in requirements engineering. *REJ*, pages 1–16, 2011.

[3] G. Bergmann et al. Incremental evaluation of model queries over EMF models. In *Model Driven Engineering Languages and Systems, MODELS'10*. Springer, 2010.

[4] L. Briand, Y. Labiche, and G.Soccar. Automating impact analysis and regression test selection based on uml designs. In *Proc. of ICSM '02*, page 252, 2002.

[5] J. Bzivin, G. Dup, F. Jouault, G. Pitette, and J. E. Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, 2003.

[6] M. Chechik, W. Lai, S. Nejati, J. Cabot, Z. Diskin, S. Easterbrook, M. Sabetzadeh, and R. Salay. Relationship-based change propagation: A case study. In *Proc. of MISE'09*, pages 7–12. IEEE Press, 2009.

[7] P. K. Chittimalli and M. J. Harrold. Regression test selection on system requirements. In *Proc. of the 1st India Soft. Eng. Conf. (ISEC'08)*, pages 87–96. ACM, 2008.

[8] P. K. Chittimalli and M. J. Harrold. Recomputing coverage information to assist regression testing. *TSE*, 35(4):452–469, 2009.

[9] I. S. Chung, H. S. Kim, H. S. Bae, Y. R. Kwon, and D. G. Lee. Testing of concurrent programs after specification changes. In *Proc. of ICSM '99*, page 199, 1999.

[10] D. Deng, P. C. Y. Sheu, T. Wang, and A. K. Onoma. Model-based testing and maintenance. In *Proc. of ISMSE'04*, pages 278–285. IEEE Press, 2004.

[11] E. Fourneret, F. Bouquet, F. Dadeau, and S. Debricon. Selective test generation method for evolving critical systems. In *REGRESSION'11*. IEEE Press, 2011.

[12] E. Fourneret, F. Bouquet, F. Dadeau, and S. Debricon. Selective test generation method for evolving critical systems. In *Proc. of ICST'11*, 2011.

[13] P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone. Requirements engineering for trust management: model, methodology, and reasoning. *IJIS*, 5(4):257–274, 2006.

[2] www.securechange.eu

[14] J. Hassine, J. Rilling, and J. Hewitt. Change impact analysis for requirement evolution using use case maps. In *Proc. of the 8th Int. Workshop on Principles of Soft. Evolution*, pages 81–90. IEEE Press, 2005.

[15] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Trans. Softw. Eng.*, 32:4–19, January 2006.

[16] IBM Rational DOORS. http://www-01.ibm.com/software/awdtools/doors/features/.

[17] L. Lin, S. J. Prowell, and J. H. Poore. The impact of requirements changes on specifications and state machines. *Softw. Pract. Exper.*, 39:573–610, April 2009.

[18] L. Liu, E. Yu, and J. Mylopoulos. Security and privacy requirements analysis within a social setting. In *Proc. of RE'03*, pages 151–161, 2003.

[19] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Trans. Softw. Eng. Methodol.*, 16, 2007.

[20] S. Microsystems. Runtime environment specification. Java Card™ platform, Connected edition. Specification 3.0, 2008.

[21] L. Naslavsky, H. Ziv, and D. J. Richardson. Mbsrt2: Model-based selective regression testing with traceability. In *Proc. of ICST'10*, pages 89–98. IEEE Press, 2010.

[22] J. Natt och Dag, B. Regnell, P. Carlshamre, M. Andersson, and J. Karlsson. A feasibility study of automated natural language requirements analysis in market-driven development. *Requir. Eng.*, 7(1):20–33, 2002.

[23] O. Pilskalns, G. Uyan, and A. Andrews. Regression testing uml designs. In *Proc. of ICSM'06*, pages 254–264, 2006.

[24] P. Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *MoDELS*, pages 1–15, 2007.

[25] D. Tanabe, K. Uno, K. Akemine, T. Yoshikawa, H. Kaiya, and M. Saeki. Supporting requirements change management in goal oriented analysis. In *Proc. of RE'08*, pages 3–12, 2008.

[26] H. Ural, R. L. Probert, and Y. Chen. Model based regression test suite generation using dependence analysis. In *Proc. of the 3rd Int. Workshop on Advances in Model-based testing*, pages 54–62, 2007.

[27] A. van Lamsweerde. Elaborating security requirements by construction of intentional anti-models. In *Proc. of ICSE'2004*, pages 148–157, 2004.

[28] D. Varró and A. Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):214–234, 2007.

[29] A. von Knethen and M. Grund. Quatrace: A tool environment for (semi-) automatic impact analysis based on traces. In *ICSM*, pages 246–255, 2003.