Towards Antipatterns-Based Model Checking

Hassan Loulou, Sebastien Saudrais, Hassan Soubra, Cherif Larouci

Towards Antipatterns-Based Model Checking

Hassan Loulou
University of Paris-Sud
Paris, France
email: hassan.loulou@u-psud.fr

Sebastien Saudrais
ESTACA'LAB
Laval, France
email: sebastien.saudrais@estaca.fr

Hassan Soubra
ESTACA'LAB
Saint-Quentin-en-Yvelines, France
email: hassan.soubra@estaca.fr

Cherif Larouci
ESTACA'LAB
Saint-Quentin-en-Yvelines, France
email: cherif.larouci@estaca.fr

***Abstract*—Discovering bugs in the early stages of the development life cycle is an important issue. However, software model checking realized by transforming design models into formal methods cannot test all the possible execution scenarios. Thus, we developed an approach to guide the model checker and the security engineer to the most suspicious parts of these models firstly. The objective is to build and analyze antipatterns to notify the security engineer to concentrate on specific parts of their models during the model checking. Our first contribution is dedicated to exploring ProB model checker features which help the translated model to find attack scenarios automatically. The second one is the definition and the analysis of 10 antipatterns as a step towards their automatic detection.**

Keywords–Antipatterns; Model Checking; Formal Methods.

I. INTRODUCTION

Hidden errors in software design phase lead in later software development stages into complex bugs which need a lot of time to be solved. Thus, discovering these errors at this phase is of high importance.

Few works have examined the impacts of models' functional and non-functional artifacts co-evolution on design constraints. This type of co-evolution exists in UML profiles such as SecureUML [1]. Our approach validates SecureUML models' dynamic aspects by applying model checking operations after transforming them into a formal representation called B-method [2]. Meanwhile, the validation of design models cannot explore all the possible software executions and requires certain level of experience with formal methods. For realizing a systematic validation of security policies, the model checker needs to be guided to discover design constraints by applying appropriate enhancements on the models' resulted formal representations. Furthermore, system designers would prefer intuitive solutions depending on antipatterns representations for those design structures which must be avoided. Also, a solution for highlighting suspicious parts of the models which are likely to introduce violations during the system evolution may make their work easier.

We study the existing facilities for guiding the model checker to detect security constraints' violations in SecureUML. Thereafter, we define the design artifacts which were the source of violations in form of antipatterns. These antipatterns are substructures suspected to be the reason of design constraints' violations during software evolution. They are discovered after the transformation of models into a formal representation and after launching their formal verification using ProB [3] model checker's facilities. We aim at paving the way towards an automatic detection of the root cause of security bugs by introducing design artifacts which are responsible for these bugs to the software security engineer. We applied our approach on a case study related to the verification of access control constraints.

This paper is organized as follows: In Section II, we explain the structure of SecureUML [4] profile and the limitations of its validation works. In Section III, we show how we exploited ProB model checker to find the violations automatically. Thereafter, in Section IV, we introduce an example of the extracted antipatterns. Finally, in Section V, we make a conclusion on our contribution and perspectives.

II. SECUREUML VALIDATION

In this section, we show the basic idea of SecureUML, the works related to validating it and our choice to represent SecureUML models and to exploit this representation.

A. SecureUML

Role-based access control (RBAC) has been standardized by the National Institute of Standards and Technology (NIST) [5]. It defines a role as a set of permissions to access resources. Users get their permissions by being assigned to one or more roles. SecureUML [1], is an extension of UML for specifying RBAC access control policies. In SecureUML, a permission is a relation connecting a role to actions on resources. The permission semantics are defined by the *action* elements used to classify the permissions [6]. Every *action* represents a class of security relevant operations on a protected resource. Access control can be expressed as an assignment of users to permissions by using their roles. Otherwise, authorization constraints, or more commonly the Object Constraint Language (OCL) [7] constraints, are checked on snapshots of the meta-model. OCL can specify constraints on users permissions in an expressive-contextual manner.

B. Related Works

Yu *et al* [8][9] exploited USE tool to support lightweight analysis of RBAC security policies expressed by UML and OCL to find violations. In their approach, an application design model was translated into a model containing predefined valid sequences of object diagrams. A snapshot is an object configuration that describes a system state. A sequence of object interactions, called a scenario, is then checked against the invariants defined in the snapshot model. After the analysis, if a good scenario is described as invalid, or a bad scenario is described as valid, then there is a problem in the security policy which either prevents valid scenarios or permits invalid ones. Basin *et al* [6][10] introduced SecureMOVA. This tool is

 ISBN: 978-1-61208-465-7

TABLE I. THE LACK OF AUTOMATED DISCOVERY FOR ATTACK SCENARIOS

	Supporting tool	Model expressed as	Coverage of static aspects	Animation of models	Generating attack scenarios	Funcutional evolution empacts on the security	Define attack patterns in UML	Systematic discovery of attack scenarios
Yu	USE	SecureUML diagrams	Yes	No	No	No	No	No
Basin	SecureMOVA	Component diagram *OCL + RBAC*	Yes	No	No	No	No	No
Wuliang Sun	Alloy	Translation from Class diagrams and OCL	Yes	Yes	Only security parts evolution	No	No	No
Nafees Qamar	Z	Translation from SecureUML	Yes	Yes	No	Yes	No	No

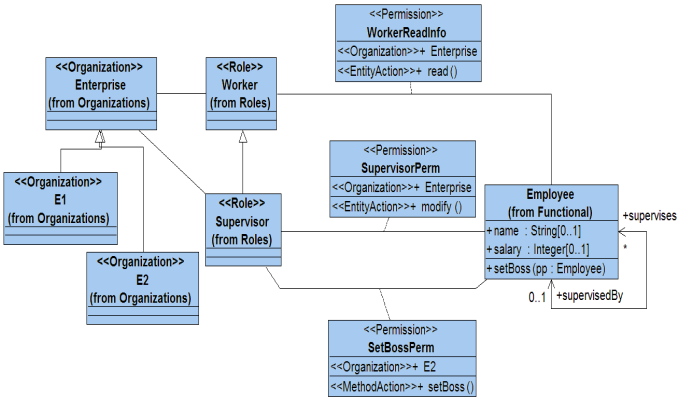

Figure 1. SecureUML case study

used to ask questions about a current state, i.e., a given object diagram. Such queries return the permissions authorized for a given role, or a given user. However, these two approaches do not consider the real execution sequence of operations to reach a specified state. They just give a possible object diagram representing a required constraint and conforming to the class diagram according to some specified constraints. Nevertheless, they do not ensure the reachability of this state. Another approach tried to simulate the real execution of software systems by introducing the notion of execution scenarios. It depends on transforming UML diagrams augmented with OCL constraints into Alloy [11][12][13]. However, these works do not transform the security constraint to validate them by Alloy model checker and they do not consider the impacts of the functional model evolution on the security policy. Moreover, these works suffer from the false negative alerts (Alerts that should have happened but did not because the model checker did not reach certain states). Additionally, they do not define a systematic approach for guiding the model checker. An interesting approach exploited Z formal method to animate SecureUML models [14]. However, false negatives still exist and antipatterns are not defined and thus not exploited to automate the search for attacks. We compare these works in Table I, showing the lack of a validation approach for the functional model evolution impacts on the security parts of secureUML. Also, no antipatterns are defined to guide the security engineer or the model checker into the suspicious parts of the models under test. To solve this problem, we translated both functional and security parts into B-method. Then, we proposed solutions to guide the model checker to find the co-evolution of SecureUML parts and we exploited these solutions to resume our findings in the form of antipatterns.

C. B-method for SecureUML validation

We decided to transform SecureUML models into B-method [15]. The reason is that B-method allows to simulate the co-evolution of functional and non-functional parts of models by its animator called ProB. This simulation mimics the real execution of software system. Moreover, a B-method tool called B4Msecure [16] is developed by Ledru *et al.* [17][14] to transform SecureUML models into B-method. B-method is a formal method for specifying, refining and implementing software systems. It is based on *Set Theory* and *Predicate Calculus*. Each B model is called an abstract machine. These machines are animated using ProB. Yet, the model checker needs to be guided by the user in order to find the potential attacks. Thus, the tool still needs an automation in order to search for models' most suspicious artifacts and to examine the impacts of these artifacts' possible compositions.

III. OUR PROPOSITIONS FOR VALIDATING SECUREUML

In this stage, we aim at exploring ProB features which help to find an attack scenario depending on the capabilities of ProB model checker.

A. Illustrative example

We built the security model shown in Figure 1. It is supplied with a stereotype representing the notion of *organization*. In this way, a *user*, called Jack, who has the role *supervisor*, is able to access data related to *employees* in *organization* E2, but he is prevented from doing so in E1. In this model, we notice that a new function *setBoss()* is added to express the uncontrolled reflexive association presented in [10] and to examine the potential risks resulting from the solution introduced there.

B. Validating the resulting model by ProB

We explored many mechanisms to find attack scenarios. The validation process considers an initialization state as shown in Figure 2, where there are two *employees* and one *supervisor* in the same *organization* E2.

1) Finding flaws by querying the functional model: When an *employee* in the functional model gets the role *supervises* of the reflexive association, he gets simultaneously a new role in the security model and new permissions according to this new role. In our example, when the *employee* Martin becomes a *supervisor*, an interaction between the functional and the security models occurs by invoking the operation

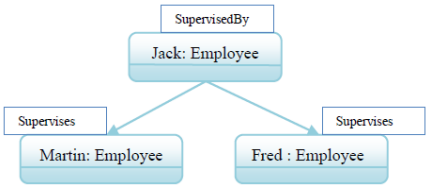

Figure 2. The initial state

add_userAssign. This operation makes him a *supervisor* in his *organization*. As a result, the following assignment, which adds new role to the user in his *organization*, takes place: *(user ↦ (org ↦ role))*. This relation has the following values in our illustrative example: *(Martin ↦ (E2 ↦ Supervisor))*.

We can search for this scenario in ProB by searching a state satisfying specific predicate in the checked model as follows: find a state in which *Boss(Martin) = Fred*. As a result, the model checker finds a scenario which makes an *employee* as a *supervisor*. Consequently, this new *supervisor* has new permissions and this case is suspicious. We found that as the reflexive association is not controlled by constraints, it does not have a clear interpretation. Moreover, as Basin *et al* [10] used an *actional* tool, which cannot simulate the real reachable methods in a specific system state, they could not consider the operations' real effects on the security policy and they could not find this potential attack.

2) Querying security aspects of the model: The proposition here is to make queries about all the roles which someone must not be authorized to have. Thus, we ask the model checker queries about the roles of users. If the translated model contains a scenario which can give a user, Martin, the role *supervisor*, considering that he was an *employee* with the role *worker*, then this scenario is suspicious and it can be a flaw in the security policy. The query about the system states takes the following form: *User_assign (Martin) = (E2 ↦ Supervisor)*. This query means: is there a scenario that leads the system to a state where Martin is a *supervisor*? Considering that the latter state is assumed as an invalid state. We search for this state, because the existence of such sequence of operations may make Martin able to change the salary of other users in the system, which is considered illegal in that *organization*.

The result given by the model checker is the following susceptible scenario: after giving the values of the system constants concerning the system users, the system's functional and security models are initialized. After that, the users are assigned to their permissions. Jack, who is a *supervisor*, connects to the system as a *supervisor* in the *organization* E2. This assignment of Jack to his roles occurs in the session S1. While, in another session, a user whose name is Fred and who has the role *worker* takes the session S2 and connects to the system as a *worker*. When Fred gives Martin a position *supervisor* by applying the function *secure_Employee_setBoss*, the state of the system evolves and Martin gets the role *supervisor*. As a result, with the existence of another *employee* like Bob, Martin who has got temporarily the role *supervisor* becomes able to make Bob as a *supervisor*. Subsequently, even if Martin loses his new role, Bob holds this role and can change the salary of his colleagues Martin and Fred.

3) Exploiting the attainability of an operation: An operation is enabled if its precondition and its guard's sections are true. These two sections are computed taking into account the system state. We proposed a new solution benefiting from this property. The mechanism of this scenario detection solution is explained by the following example. A system variable *currentUser* is related to an active session user. Therefore, we can add a constraint in the guard's section of an operation, where this constraint says that the operation will not be enabled if the current user is not Fred, for example. We do that taking into account that this user does not have an authorization to execute this operation. Next, we search for a scenario in which this operation could be attainable and executable. If such scenario exists, we conclude that there is a suspicious scenario which may cause a serious attack. As will be shown next, the constraint *(currentUser = Fred)* is added in the guards section of the operation *secure_Employee_SetSalary*:

```
secure_Employee_SetSalary(Instance, Employee_
    ↪ salaryValue) = SELECT currentUser = Fred
```

The disadvantage of this solution is: to reach the state we are searching for (where currentUser = Fred), we may need the same operation without the added pre-condition. Thus, the solution is to add another operation with the same name but without that obstructive constraint. In this way, this additional operation will be executed first, if needed. Then, ProB continue searching for the target suspicious state.

4) Searching flaws by asking ProB about permissions: In this way, we are interested in finding a scenario where a user is capable of reaching a permission he did not have before. For example, is there a scenario in which the user Martin can get the access to execute the operation *employee_SetSalary*? The execution of this operation was granted, at the beginning of the system execution, only to Jack who is a *supervisor*. This question can be formulated using B as follows:

```
employee_SetSalary ϵ isPermitted[currentOrgRole_
    ↪ Session] ∧ CurrentUser=Martin
```

Consequently, if the model checker reaches a state where the previous constraint holds, we conclude that the user Martin will be able to get an unauthorized permission. The resulted scenario given by the model checker is the same as the one mentioned in the previous solutions.

5) Taking the initial state into account: We consider different possible object diagrams resulted from the reflexive association and its multiplicities in the illustrative example.

a) Studying initialization object diagrams: The first object diagram, Figure 3, is composed of the following structure: an *employee* with a role *worker* who is not associated to a *supervisor* and another *employee* with a role *worker* associated to a *supervisor*. In this state, we cannot find an attack scenario. No scenario can lead Martin to change Freds salary because Jack is not the *supervisor* of Fred. Thus, he is not able to execute the *setBoss()* operation. Likewise, in the following state, Figure 4, there is no attack scenario. The only case where there is an attack scenario is, as shown in Figure 2, when there is an object diagram containing more than one *employee* managed by a common *supervisor*. This is because the existence of a *supervisor* for an *employee* is essential to change the value of the reflexive association represented as a relation called boss in the functional model. This relation issues an association between two objects in the Employees functional class diagram.

Moreover, we found another kind of attack happening when we have a hierarchy of *employees* in the same *organization*, as shown in Figure 5. It is impossible to change the salary of the *employee* E7, as he is not associated to a *supervisor*. An *employee* E1, E3 has found a scenario to change the salary of their *supervisors* E3, E5 respectively. As a result, we found out that the initial state which is constructed using the *supervisor* hierarchy two times at least is essential to have such risk.

 ISBN: 978-1-61208-465-7

Figure 3. Initialization state1.

Figure 4. Initialization state2

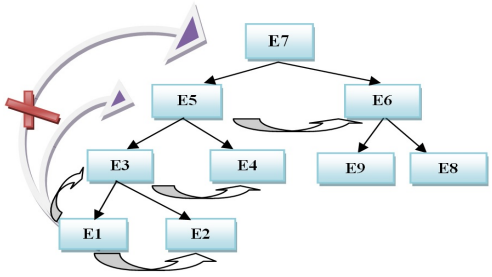

Figure 5. Initialization state3

b) An approach to construct an important object diagram (tree): A solution is needed to construct either all the possible object diagrams of the class diagram or all those states which are considered more suspicious to produce the attack scenario. The produced object diagram differs according to the structure it tries to instantiate. For example, the different states we must consider in the previous reflexive association are: (i) initializing the B machine with two *employees*, one of them has a different *manager*, (ii) two *employees*, each of them has one different *manager*, (iii) a tree structure representing the structure of the hierarchy. (iv) considering another *organization* with some users.

The initial state must be considered separately in each new diagram. However, in addition to the previous recurring structure, we try to give solutions to produce initial states for the main possible structures of SecureUML as follows:
(A) When there is an association affecting an OCL constraint, the multiplicity of the two ends of the association has to be considered in the initialization. For the *many* multiplicity, two objects at least are instantiated.
(B) Each of the entities mentioned in a constraint must be instantiated in order to construct a structure capable of tricking the OCL constraint. As shown in Figure 6, a malicious user connects with a *manager* and *staff* roles at the same time. When he is a *manager*, he delegates the user with the role *staff* (himself) in order to make him able to approve the refund, as stated by the OCL constraint associated with the permissions on the class refund. Thus, this user will be able to give the value True to the variable *IsApproaved* in the class *refund*. Hence, the constraint *Approved by 2 managers* became true, as the staff become also a *manager*. Then, this *manager* approves the refund and *refund* procedure starts. Thus, the *manager* prepares and issues the payments after approving them. This separation of duties (SoD) problem comes from the structure in

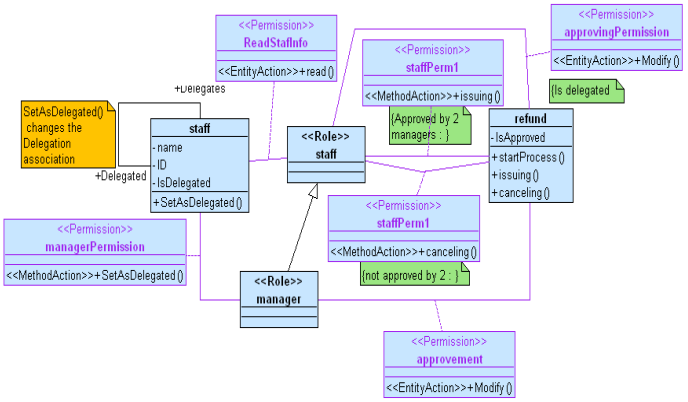

Figure 6. Example showing the importance of instantiating OCL entities

which the *manager* inherits the permissions of *staff* role. The solution to this problem is to add an OCL constraint saying that the same person can connect as a *manager* or as a staff in the same session, but not as the both.
The instantiation of this diagram could take the following way: an instance of the class refund and two users had to be instantiated as mentioned in the OCL constraint *Approved by 2 managers*. Additionally, according to the constraint *is delegated*, the delegation instance must be presented. That means, we must produce an object *delegates* and an object *delegated*. As a result, this initialization diagram is capable of starting the game of finding the attack.
(C) For a unidirectional association, the accessed object must be instantiated and linked to the instance accessing it.
(D) When there is an OCL constraint controlling permissions and depending on an attribute value, we ought to instantiate the class where this variable exists, then we assign a user who has a *modify* permission.
(E) When there is a role inheritance, we assign users to the inheriting roles to check all the prevented separation of duties conditions as happened in Figure 6. Thus, an invariant has been added to prevent this user getting two conflicting roles.

The result of Figure 1 translation contains a reflexive association: $bb \in Employee \nrightarrow Employee \wedge bb$. Before setting the permissions of users, we applied the previous approach for constructing an efficient object diagram. The approach contains the following steps: (i) give the possible values for associations' ends and specify their multiplicities, (ii) avoid constructing circular associations to avoid a state where a person is *supervisor* of himself, (iii) add properties for associations members' multiplicities, (iv) initialize the functional model, (v) produce the possible values of the relation: $user \rightarrow (\{E2\} \times \text{ROLES})$, (vi) initialize the user assignment model. To make the possible suspicious initial state (the tree shown before), the following constraint is defined on the association multiplicity:

$$(card(ran(bb))=1 \wedge card(dom(bb))=2) \bigvee (card(ran(bb)) \hookrightarrow =2 \wedge card(dom(bb))=2) \bigvee (card(ran(bb))=4$$

As a result, the way we ask our queries to the model checker must change. This is because the model checker possibly will take a shorter path while searching and this path is not normally an attack scenario. For example, if we ask ProB about a state in which Martin can change the salary of Fred, the

model checker finds rapidly a scenario satisfying this query. However, in this scenario, Martin is assigned directly to the role *supervisor* and he changes the salary of Fred, which is not an interesting scenario for us. As a solution, we propose to make the queries separated from users names and related to a general description of the attack meaning, as shown in the next section.

6) Generalizing the way we search the suspicious scenarios: To avoid the problems of making automatic initialization with specified properties, we propose to avoid asking about a user who has a specific permission. The question becomes more general as follows: is there a scenario which allows a user who have a *worker role* to have a *supervisor role*?

$$\forall(u).(u \in USERS \wedge u \in dom(user\_assign\ (ran(user\_assign)\ \{Worker\})) \wedge u \notin ran(B) \implies u \notin dom(user\_assign\ (ran(user\_assign)\ \{Supervisor\})))\wedge..$$

Where, the variable B contains the *boss* relations values calculated during the initialization of the security machine. In another way, the set *wasWorker* contains all the users assigned to the role *worker* at the initialization step of the security machine. Consequently, we add this formula as an invariant of the machine. Thus, when this invariant is violated, this means one of the users has obtained the role *supervisor*. This formula is constructed as follows:

$$\forall\ (u).(u \in USERS \wedge u \wedge wasWorker \implies ran(\{user\_assign(u)\}) \neq \{Supervisor\})$$

Where:

$$wasWorker := dom(user\_assign \rhd (ran(user\_assign) \rhd \{Worker\})) \cap dom(boss)$$

The other invariant we added to capture the attack risks concerns preserving the permissions during the execution of the model checker. It can answer the following question: is there a scenario that leads to an increase in any users permissions during the system execution? The formula is constructed to ask always about the user connected to the role which has a lower number of permission such as the role *worker* in the illustrative example. The following formula shows an example of an added invariant in the security machine. This formula searches if the number of a user's permissions with the role *worker* may increase during the execution of the system.

$$\forall\ (u,org,role).(u \in USERS \wedge u{=}currentUser \wedge u \in dom(boss) \wedge org \in ORG \wedge org{=}E2 \wedge role \in ROLES \in role{=}Worker \implies isPermitted[\{user\_assign(u)\}] - isPermitted[\{(org \mapsto role)\}] \neq \varnothing)$$

Similarly, we can add the names of other roles if they exist in the system to be contained in the last formula. By doing so, the query covers all the users in the system. As a result, searching those scenarios, which may affect the security policy, has become easier.

7) Detecting possible vulnerabilities in OCL constraints: The goal in this stage is to violate the OCL constraints which describe security aspects because the ability to violate them is considered as a flaw. To succeed in doing that, we express this constraint but with a change in their parts expressing the name of the roles. Then, these names are replaced by other

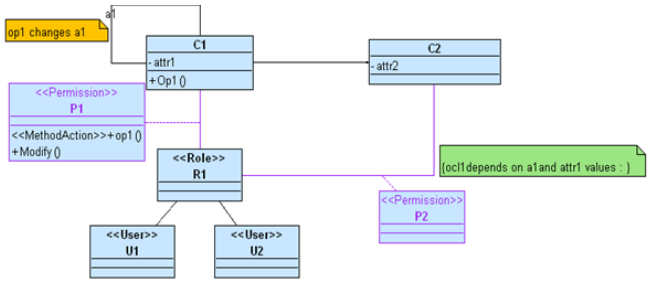

Figure 7. An example of the defined patterns

unauthorized ones. In other words, we reached this state by adding a constraint as a pre-condition for an operation like the operation *secure_setSalary()*. This constraint restricts the operation execution permission to Fred who has the role *worker*, such access must be prohibited. This constraint specifies Fred as the *employee* whose salary will change. Checking the model under this condition shows an important evolution of the functional model. This evolution violates the access permission which limits the execution of setSalary to a *supervisor* in the same *organization* of the *employee*.

Furthermore, when an employee is delegated to do the *supervisor* tasks, he keeps his previous role. That means a person who executes the operation *secure_setSalary()* can be the same one for whom this operation is executed. As a result, an *employee* is able to change his own salary. Thus, a user keeps owning his previous roles when his position changes. The structure of SecureUML model, which contains the previous artifacts, could be considered as a recurring antipattern that needs to be checked in new models under test.

IV. Extracting The Antipatterns

To reduce the impacts of the state explosion, we are going to define scenario attack patterns according to our experimentations using the previous translation and validation techniques. Thereafter, we will try to find out the suspicious recurring structures, as well as the operations affecting these structures to help the model checker to find its way to the suspicious attack states.

A. Defining SecureUML antipatterns structures

To help the model checker estimating if a translated SecureUML model may have an access attack, we constructed a table containing characteristics of previous suspicious SecureUML diagrams. For the time being, we are going to define some patterns found in the examined diagrams.

In the following, we introduce an example of the defined antipatterns structure, *suspicious recurring structure*, shown in Figure 7. In this antipattern, there is a read permission on an entity. But, after initializing the security machine, the user finds himself able to modify parameters in the entity C2.

The detailed parts of this *suspicious structure* are as follows: Users u1, u2 assigned to role r1. Class C1 has a reflexive association a1. R1 has a *modify* permission p1 on class c1. R1 has *methodAction* permission on the operation op1 which changes a1. A1 and attr1 decide the value of ocl1. R1 has permission p2 on class c2 controlled by ocl1.
This structure existed in different examples of SecureUML.

 ISBN: 978-1-61208-465-7

TABLE II. REASONS OF VIOLATIONS IN THE ANTIPATTERNS

SubStructure	Description
Sub1	There is a unidirectional association.
Sub2	The accessing class is itself accessed directly by a fullaccess permission which affects the accessed class operations and attributes.
Sub3	The role owning a direct permission on the accessing class has only Read permission on an accessed class.
Sub4	An OCL constraint depends on an association value (grant permission).
Sub5	An operation modifies an association's values.
Sub6	Methodaction permission on an operation which modifies an important association.
Sub7	An OCL constraint depends on an attribute value.
Sub8	Role inheritance without adding SoD constraints.
Sub9	More than one user are assigned to the same role, but they have some hierarchy defined by a reflexive association. Their permissions could differ according to a constraint.
Sub10	Assigning roles after changes in a hierarchy association.
Sub11	User assigned to two roles at the same time. These two roles did not come from the inheritance. They have permissions on the same entity.
Sub12	An operation changes an attributes value which participates in calculating the value of an OCL constraint.

We use it with the other supposed 9 antipatterns to analyze and predict the existence of flaws in the new models.

B. Extracting the reasons of attack in the previous structures

In order to avoid the arbitrary search of the model checker algorithms, we have extracted the most important factors of the security policy vulnerabilities depending on some characteristics, shown in Table II. These characteristics describe the most suspicious recurring structures used in SecureUML diagrams.

For each attack pattern, we search for the existence of each of the sub-structures (sub-structures 1 to 12). When this structure exists, we modify the scores of this pattern parts in a probabilistic suspicious-table. Thereafter, the suspicious-table is used for guiding the model checker. Due to the limited space, we show later how to extract the exhaustive search plan which concentrates on suspicious execution scenarios. The optimization process of the suspicious-table is done according to the incremental feedback loop process, shown in Figure 8.

V. CONCLUSION AND FUTURE WORK

According to our knowledge, bugs resulted from the co-evolution of the functional and non-functional parts of SecureUML models are not sufficiently studied. Moreover, no antipatterns have been defined to help the system designers to avoid suspicious compositions of design artifacts.

This paper introduced an approach based on the transformation of SecureUML diagrams into formal models to simulate their execution. Moreover, we exploited the possible offered validation techniques in the model checker ProB. By applying them on many different systems, we could define 10 kinds of suspicious rudimentary antipatterns. Then, we analyzed them to notify the security engineer to investigate much more validation efforts on the suspicious parts when they are introduced in their models. However, this work must be extended to detect the antipatterns automatically. Moreover, the transformation of the security models into B-method still needs user interventions to make the resulted model executable. This limitation should be avoided to make the approach presented in Figure 8 achievable.

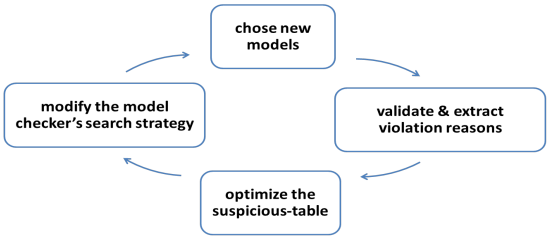

Figure 8. Feedback loop for guiding the model checker

Next, we are going to use a graph query language to efficiently discover antipatterns substructures in large models. The objective of this task is to generate a controller to guide the model checker automatically to design flaws.

REFERENCES

[1] D. Basin, M. Clavel, J. Doser, and M. Egea, "A metamodel-based approach for analyzing security-design models," in Model Driven Engineering Languages and Systems. Springer, 2007, pp. 420–435.

[2] J.-R. Abrial, M. K. Lee, D. Neilson, P. Scharbach, and I. H. Sørensen, "The b-method," in VDM'91 Formal Software Development Methods. Springer, 1991, pp. 398–405.

[3] "Prob," https://www3.hhu.de/stups/prob/index.php/Main_Page, accessed: 2016-01-21.

[4] T. Lodderstedt, D. Basin, and J. Doser, "Secureuml: A uml-based modeling language for model-driven security," in ł UML 2002The Unified Modeling Language. Springer, 2002, pp. 426–441.

[5] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli, "Proposed nist standard for role-based access control," ACM Transactions on Information and System Security (TISSEC), vol. 4, no. 3, 2001, pp. 224–274.

[6] D. Basin, M. Clavel, J. Doser, and M. Egea, "Automated analysis of security-design models," Information and Software Technology, vol. 51, no. 5, 2009, pp. 815–831.

[7] "Ocl," http://www.omg.org/spec/OCL/, accessed: 2016-01-25.

[8] L. Yu, R. B. France, and I. Ray, "Scenario-based static analysis of uml class models," in Model Driven Engineering Languages and Systems. Springer, 2008, pp. 234–248.

[9] L. Yu, R. France, I. Ray, and S. Ghosh, "A rigorous approach to uncovering security policy violations in uml designs," in Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on. IEEE, 2009, pp. 126–135.

[10] D. Basin, M. Clavel, and M. Egea, "A decade of model-driven security," in Proceedings of the 16th ACM symposium on Access control models and technologies. ACM, 2011, pp. 1–10.

[11] "Alloy," https://www3.hhu.de/stups/prob/index.php/Main_Page, accessed: 2016-01-22.

[12] W. Sun, R. France, and I. Ray, "Rigorous analysis of uml access control policy models," in Policies for Distributed Systems and Networks (POLICY). IEEE, 2011, pp. 9–16.

[13] M. Toahchoodee, I. Ray, K. Anastasakis, G. Georg, and B. Bordbar, "Ensuring spatio-temporal access control for real-world applications," in Proceedings of the 14th ACM symposium on Access control models and technologies. ACM, 2009, pp. 13–22.

[14] Y. Ledru, A. Idani, J. Milhau, N. Qamar, R. Laleau, J.-L. Richier, and M.-A. Labiadh, "Taking into account functional models in the validation of is security policies," in Advanced Information Systems Engineering Workshops. Springer, 2011, pp. 592–606.

[15] J.-R. Abrial, J.-R. Abrial, and A. Hoare, The B-book: assigning programs to meanings. Cambridge University Press, 2005.

[16] "B4msecure," http://b4msecure.forge.imag.fr/, accessed: 2016-01-15.

[17] Y. Ledru, A. Idani, and J.-L. Richier, "Validation of a security policy by the test of its formal b specification: a case study," in Proceedings of the Third FME Workshop on Formal Methods in Software Engineering. IEEE Press, 2015, pp. 6–12.

 ISBN: 978-1-61208-465-7