HAL
open science

Extraction of Insider Attack Scenarios from a Formal Information System Modeling

Amira Radhouani, Akram Idani, Yves Ledru, Narjes Ben Rajeb

To cite this version:

Amira Radhouani, Akram Idani, Yves Ledru, Narjes Ben Rajeb. Extraction of Insider Attack Scenarios from a Formal Information System Modeling. Formal Methods for Security, Jun 2014, Tunis, Tunisia. pp.5-19. hal-01062812

HAL Id: hal-01062812

https://hal.science/hal-01062812

Submitted on 10 Sep 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Extraction of insider attack scenarios from a formal Information System modeling

Amira Radhouani[1,2,3,5], Akram Idani[1,2], Yves Ledru[1,2], Narjes Ben Rajeb[3,4]

[1] Univ. of Grenoble Alpes, LIG, F-38000 Grenoble, France
[2] CNRS, LIG, F-38000 Grenoble, France
[3] LIP2-LR99ES18, 2092, Tunis, Tunisia
[4] INSAT - Carthage University, Tunisia
[5] FST - Tunis-El Manar University, Tunisia

Abstract. The early detection of potential threats during the modelling phase of a Secure Information System is required because it favours the design of a robust access control policy and the prevention of malicious behaviours during the system execution. This paper deals with internal attacks which can be made by people inside the organization. Such attacks are difficult to find because insiders have authorized system access and also may be familiar with system policies and procedures. We are interested in finding attacks which conform to the access control policy, but lead to unwanted states. These attacks are favoured by policies involving authorization constraints, which grant or deny access depending on the evolution of the functional Information System state. In this context, we propose to model functional requirements and their Role Based Access Control (RBAC) policies using B machines and then to formally reason on both models. In order to extract insider attack scenarios from these B specifications our approach first investigates symbolic behaviours. The use of a model-checking tool allows to exhibit, from a symbolic behaviour, an observable concrete sequence of operations that can be followed by an attacker. In this paper, we show how this combination of symbolic execution and model-checking allows to find out such insider attack scenarios.
keywords: Information System, B-Method, RBAC, attack scenario, Model Checking, Symbolic Search.

1 Introduction

Developing secure Information Systems remains an active research area addressing a wide range of challenges mostly interested in how to prevent from external attacks such as intrusion, code injection, denial of service, identity fraud, etc. Insider attacks are less addressed despite they may cause much more damage because an insider is over all a trusted entity. Intrinsically it is given means to violate a security policy, either by using legitimate access, or by obtaining unauthorized access. This paper deals especially with Role Based Access Control (RBAC) concerns with the aim to exhibit potential insider threats from a formal modelling of secure Information Systems. We are interested in finding

attacks which conform to the access control policy, but lead to unwanted states. These attacks are favoured by policies involving authorization constraints, which grant or deny access depending on the evolution of the functional Information System state. This reveals, on the one hand, the need to link the security model to the functional model of the information system, and on the other hand, to build tools taking into account the dynamic evolution of the IS state.

Tools such as SecureMova [3] and USE [6] are dedicated to validate security policies related to a functional model. But these tools don't take into account dynamic evolution of the functional state. In [10], we discussed shortcomings of existing approaches in this context, and showed the advantages of using a formal specification assisted by animation tools. This paper goes a step further than our previous works by taking advantage of model-checking and proof tools in order to automatically find insider attack scenarios composed of a sequence of actions modifying the functional state and breaking the authorization constraint.

This paper is organized as follows: section 2 gives an overview of our approach and its underlying methodology. In section 3 we present a simple example that illustrates our contribution. Section 4 defines semantics and technical aspects. In section 5 we propose a symbolic search that automates generation of attack scenarios and we discuss results of its application on the given example. Finally, we draw conclusions and perspectives.

2 Overall approach

Bridging the gap between formal (*e.g.* Z, B, VDM ...) techniques and graphical languages such as UML has been a challenge since several years. On the one hand, formal techniques allow automatic reasoning assisted by proof and model-checking tools, and on the other hand, graphical techniques allow visualization and better understanding of the system structure. These complementary aspects are useful to ensure a software development process based on notations with precise syntax and semantics and which allows to structure a system graphically. Most existing research works [1, 5, 7, 13] in this context have been focused only on modelling and validation of functional aspects which are initially described by various kinds of UML diagrams (class, state/transition, sequence, ...) and then translated into a formal specification. These works have shown the interest of linking formal and graphical paradigms and also the feasibility of such translations.

In our work, we adopt a similar approach in order to graphically model and formally reason on both functional and security models. We developed the B4MSecure[6] platform [9] in order to translate a UML class diagram associated to a SecureUML model into B specifications. The resulting B specifications illustrated in figure 1 follow the separation of concerns principles in order to be able to validate both models separately and then validate their interactions.

The functional B model on the left hand side of figure 1 is issued from a conceptual class diagram. It integrates all basic operations generated automatically

[6] http://b4msecure.forge.imag.fr/

(constructors, destructors, setters, getters, ...) and also additional user-defined operations which are integrated into the graphical model and specified using the B syntax. This functional specification can be further improved by adding invariants and carrying out proof of correction with the help of AtelierB prover.

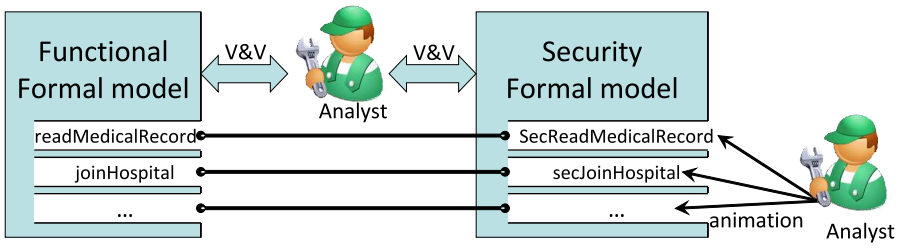

Fig. 1. Validation of functional and security models

The security model, on the right hand side of figure 1 is dedicated to control the access to functional operations with respect to access control rules defined in the SecureUML model. In our approach, we don't deal with administration operations because we make the simplifying assumption that access control rules don't evolve during the system execution. The security formal model allows to validate RBAC well-formedness rules such as no role hierarchy cycles, and separation of duty properties (SoD) such as assignment of conflicting roles to users...

This paper assumes that validation of both models in isolation is done: operations of functional model don't violate invariant properties, and the security model is robust. Such validation activities are widely discussed in the literature [12]. However, currently available validation approaches do not take sufficiently into account interactions between both models which result from the fact that constraints expressed in the security model also refer to information of the functional model. In fact, security policies often depend on dynamic properties based on the functional system state. For example, a bank customer may transfer funds from his account, but if the amount is greater than some limit the transfer must be approved by his account manager. Access control decisions depend then on the satisfaction of authorization constraints in the current system state. Dynamic evolution of the functional state impacts these constraints and may lead to a security vulnerability if it opens an unexpected access. In this paper we use validation tools (prover and model-checker) in order to search for malicious sequences of operations by analysing authorization constraints.

3 A simple example

In this section we use a running example issued from [3] and which deals with a SecureUML model associated to a functional UML class diagram.

3.1 Functional model

The functional UML class diagram (presented in figure 2) describes a meeting scheduler dedicated to manage data about two entities: Persons and Meetings.

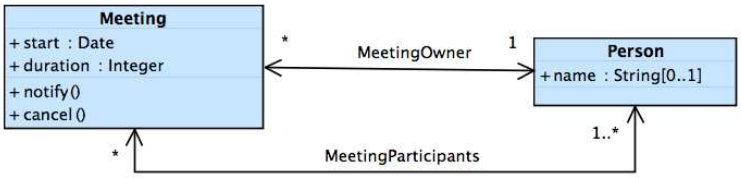

Fig. 2. Functional model of meeting scheduler system

A meeting has one and only one owner (association *MeetingOwner*), a list of participants (association *MeetingParticipants*), a duration, and a starting date. A person can be the owner of several meetings and may participate to several meetings. Operations *notify* and *cancel* are user-defined, and allow respectively to send messages to participants and to delete a meeting after notifying their participants by e-mail. Constructors, setters and getters are implicitly defined for both classes and both associations.

3.2 Access control rules

The access control model is given in Figure 3. It features three different roles:

- **SystemUser**: defines persons who are registered on the system and then have permission **UserMeetingPerm** which allow them to create and read meetings. Deletion and modification of meetings (including operation *cancel*) are granted to system users by means of permission **OwnerMeetingPerm**, featuring an authorization constraint checking that the user who tries to run these actions is the meeting owner.
- **Supervisor**: defines system users with more privileges because they can run actions notify and cancel on any meeting even if they are not owners.
- **SystemAdministrator**: having a full access on entity Person, an administrator manages system users. Full access grants him the right to create a new person, remove or modify an existing one. Furthermore, a system administrator has only a read access on meetings.

3.3 Validation

This example is intended to be validated in [3] based on a set of static queries that query a given system state in order to grasp some useful information like "*which user can perform an action on a concrete resource in a given state*".

Authorization constraint associated to **OwnerMeetingPerm** requires information from the functional model because it deals with the *MeetingOwner*

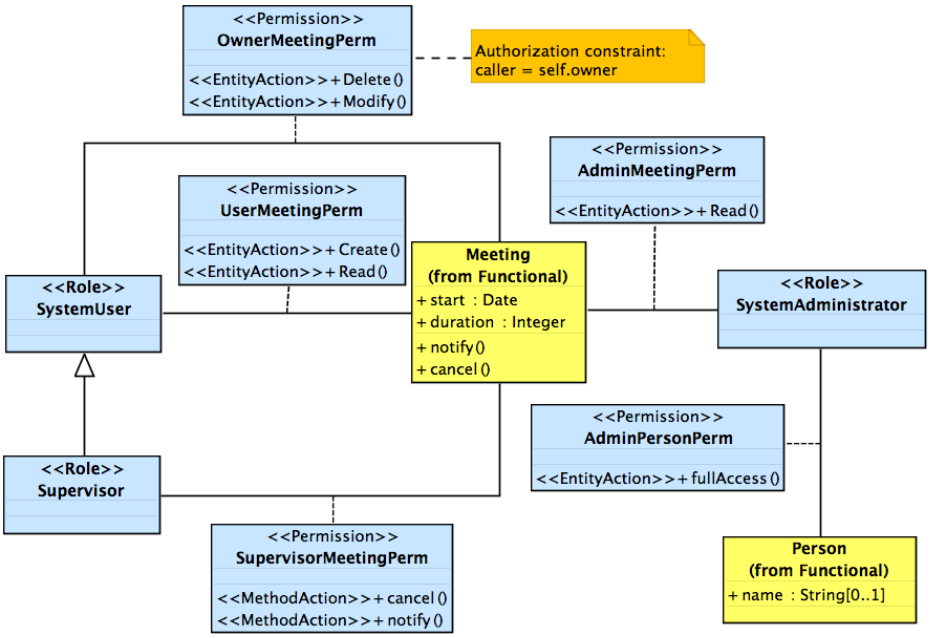

Fig. 3. Security model of meeting scheduler system

association. In the rest of this article, we consider three users John, Alice and Bob such that user assignments are as defined by figure 4 and a given initial state in which Alice is owner of meeting m_1, Bob is a participant of m_1. In such a state, the above static query establishes that only Alice is allowed to modify or delete m_1 because she is the owner of m_1.

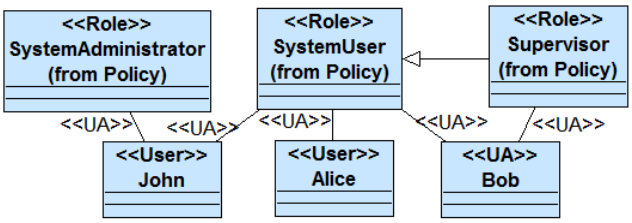

Fig. 4. Users assignement

In [10, 9] a dynamic analysis approach based on animation of a formal specification showed that validation should not only be based on a given static state, but should search for sequences of actions modifying this state and breaking the authorization constraint. For example, starting from the above state, a static query would only report that John, and also Bob, can't modify m_1 because none

of them satisfies the authorization constraint. A dynamic analysis would ask if there exists a sequence of operations enabled by John, or Bob, that allows them to modify m_1. This paper contributes towards automatically finding these malicious sequences. To perform these analysis, we applied the B4MSecure tool to the UML and SecureUML diagrams and generated a B specification counting 946 lines. This tool generates automatically a specification for all basic functional operations, which is enriched manually by some user-defined operations (*i.e.* cancel, notify).

4 Proposed approach

4.1 Trace semantics for B specifications

In order to find malicious behaviours of an operational secure IS modelling, we rely on the set of finite observable traces of our B specifications. Indeed, B specifications can be approached by means of a trace semantics composed of an initialization substitution *init*, a set of operations $\mathcal{O}$ and a set of state variables $\mathcal{V}$. We note *val* a possible state predicate allowed by the invariant and *op* an operation from $\mathcal{O}$. A functional behaviour is an observable sequence $\mathcal{Q}$

$$\mathcal{Q} \mathrel{\hat{=}} init \ ; \ op_1 \ ; \ op_2 \ ; \ \ldots \ ; \ op_m$$

such that $\forall i.(i \in 1..m \Rightarrow op_i \in \mathcal{O})$ and there exists a sequence $\mathcal{S}$ of state predicates which does not violate invariant properties:

$$\mathcal{S} \mathrel{\hat{=}} val_0 \ ; \ val_1 \ ; \ \ldots \ ; \ val_m$$

in which val_0 is an initial state, and op_i is enabled from state val_{i-1} and state val_i is reached by op_i, starting from state val_{i-1}.

The security model filters functional behaviours by analysing access control premises which are triplets (u, R, c) where u is a user, R is a set of possible roles assigned to u, and c is an authorization constraint. An observable secure behaviour is a sequence $\mathcal{Q}$, where for every step i, premise (u_i, R_i, c_i) is valid (expressed as $(u_i, R_i, c_i) \models true$). This means that roles R_i activated by user u_i grant him the right of running operation op_i and if a constraint c_i exists, then it must be satisfied. The following premises sequence $\mathcal{P}$ must be valid for $\mathcal{Q}$:

$$\mathcal{P} \mathrel{\hat{=}} (u_1, R_1, c_1) \ ; \ (u_2, R_2, c_2) \ ; \ \ldots \ ; \ (u_m, R_m, c_m)$$

4.2 Tools to exhibit behaviours from B specifications

Model-checking and symbolic proof techniques are of interest in order to exhibit a relevant behaviour from an operational B specification. Proof techniques deal with infinite systems and can prove constraint satisfiability, or establish that some operation can be enabled from an abstract state predicate. Model-checking is based on model exploration of finite systems, and can be used to find a sequence of actions leading to a given state or property. In our approach, we combine both techniques in order to overcome their shortcomings: complexity of proofs for the first one, and state explosion for the second one. In this sub-section, we illustrate both tools.

Model checking and animation (the ProB tool). ProB [11] is an animation and a model-checker of B specifications that explores the concrete state space of the specification and generates accessibility graphs. Then, every predicate val_i (where $i \in 0, 1, \ldots, m\}$) of sequence $\mathcal{S}$ is a valuation of variables issued from $\mathcal{V}$. For example, starting from an initial state val_0 where:
$\mathcal{V} = \{person, meeting, meetingOwner, meetingParticipants\}$ and such that:

$$\begin{aligned} val_0 \hat{=}\ & person = \emptyset \\ & \wedge\, meeting = \emptyset \\ & \wedge\, meetingOwner = \emptyset \\ & \wedge\, meetingParticipant = \emptyset \end{aligned}$$

and having $\mathcal{O} = \{personNew, meetingNew, meetingAddParticipants, \ldots\}$, the scenario of table 1 is successfully animated using ProB tool. Column "reached states" gives only modified B variables from the previous step.

step	Sequence $\mathcal{Q}$	Reached states $\mathcal{S}$	RBAC premises $\mathcal{P}$
1	personNew	person={Alice}	John SystemAdministrator no constraint
2	personNew	person={Alice, Bob}	John SystemAdministrator no constraint
3	meetingNew	meeting={m_1} meetingOwner={(Alice, m_1)}	Alice SystemUser no constraint
4	meetingAddParticipants	meetingParticipants={(m_1, Bob)}	Alice SystemUser Constraint: Alice is the owner of m_1

Table 1. animation of a normal scenario with ProB

In step 1, the tool animates operation $personNew$ which modifies variable person (initially equal to emptyset) and this action was performed by user John using role SystemAdministrator without need of authorization constraint. In step 4, the tool adds participant Bob to the meeting m_1 by animating operation meetingAddParticipants, after validating that authorization constraint is valid for Alice using role SystemUser. Indeed, Alice is the owner of m_1.

Symbolic proof (the GeneSyst tool). ProB is useful to animate scenarios identified during requirements analysis, or to exhaustively explore a finite subset of state space. As we are interested in finding malicious scenarios that exhibit a potential internal attack, the ProB technique may be useful only if it explores the right subset of state space in the right direction, which is not obvious for infinite systems. Symbolic proof techniques, such as that of GeneSyst [4], are more interesting because they allow to produce symbolic transition systems that represent a potentially infinite set of values. Such tools reason on the reachability

properties of a symbolic state F by some operation op from a symbolic state E. In [4], three reachability properties are defined in terms of the following proof obligations, where E and F are two disjoint state predicates:

(1) possibly reached: $E \wedge Pre(op) \Rightarrow \neg[Action(op)]\neg F$
(2) not reachable: $E \wedge Pre(op) \Rightarrow [Action(op)]\neg F$
(3) always reached: $E \wedge Pre(op) \Rightarrow [Action(op)]F$

In the generalized substitution theory, formula $[S]R$ means that substitution S always establishes predicate R, and $\neg[S]\neg R$ means that substitution S may establish predicate R. Hence, proof (1) means that state F can be reached by actions of operation op, when operation precondition is true in state E. Proof (2) means that F is never reached by actions of operation op from state E. Finally, proof (3) means that F is always reached by op from E. Let us consider, for example, the functional operation $meetingNew$:

$$
\begin{array}{l}
meetingNew(m,p) \hat{=} \\
\qquad PRE\ m \notin meeting \wedge p \in person\ THEN \\
\qquad\quad meeting := meeting \cup \{m\} \\
\qquad\quad || \ meetingOwner := meetingOwner \cup \{(m \mapsto p)\} \\
\qquad END
\end{array}
$$

This operation adds a new meeting m and links it to an owner p. If we define states E and F such that:

$$
\begin{array}{l}
E \hat{=} meetingOwner[\{m_1\}] = \emptyset \\
F \hat{=} meetingOwner[\{m_1\}] \neq \emptyset
\end{array}
$$

then proof obligation produced by GeneSyst for property (1) was successfully proved showing that operation $meetingNew$ when enabled from a state where m_1 does not exist and there exists at least one person in the system, may lead to a state where m_1 is created and has an owner.

Our work will be focused on proof (1) which states the reachability of a target state from an initial one, illustrated above, because it is sufficient to decide wether an operation is potentially useful for a malicious behaviour. Proofs (2) and (3) can be used if one would like to assume that a state can never be reached, or it is always reached, by an operation.

4.3 Malicious behaviour

Based on the security requirements, several operations are identified as critical. For example, security requirements have identified the integrity of meeting information as critical. Therefore, operations which perform unauthorized modifications are identified as critical.

A malicious behaviour executed by a user u, regarding authorization constraints, is an observable secure behaviour $\mathcal{Q}$ with m steps such that:

- op_m is a critical operation to which an authorization constraint c_m is associated.
- user u is malicious and would like to run op_m by misusing his roles R_u.
- val_0 : is an initial state where $(u, R_u, c_m) \models false$
- for every step i ($i \in 1..m$) premise $(u, R_u, c_i) \models true$

In other words, malicious user u is not initially allowed to execute a critical operation, but he is able to run a sequence of operations leading to a state from which he can execute this operation. In our investigation we suppose that user u executes this malicious sequence without collusion. This problem will be tackled in a further work.

Section 3.3 gave an example where neither Bob nor John are allowed to run a modification operation, such as $meetingSetStart$ which modifies attributes of class Meeting, from the initial state due to the authorization constraint. This initial state is:

$$\begin{aligned} val_0 \mathrel{\hat{=}} & \; person = \{Alice, Bob\} \\ & \wedge \; meeting = \{m_1\} \\ & \wedge \; meetingOwner = \{(Alice \mapsto m_1)\} \\ & \wedge \; meetingParticipant = \{(m_1 \mapsto Bob)\} \end{aligned}$$

In the following, we denote as $init_0$ the sequence of operations leading to val_0 such as that presented in table 1. We used the model-checking facility of ProB in order to explore exhaustively the state space and automatically find a path starting from val_0 and leading to a state where operation $meetingSetStart$ becomes permitted to John. We asked ProB to find a sequence where John becomes the owner of m_1:

$$meetingOwner(m_1) = John$$

After exploring more than 1000 states, ProB found a scenario in which John executes sequentially operations personNew, personAddMeetingOwner and meetingSetStart. Indeed, this dynamic analysis showed that John, as a system administrator, has a full access to entity Person. This permission allows him to create, modify, read and delete any instance of class Person. First, he creates an instance John of class Person that corresponds to him by running operation $personNew(John)$. Then he adds meeting m_1 to the set of meetings owned by John, by running operation $personAddMeetingOwner(John, m_1)$ which is a basic modification operation of class Person. These two actions allowed him to become the owner of m_1 and then he was able to modify the meeting of Alice.
Like all model-checking techniques, when ProB explores exhaustively the state space, it faces the combinatorial explosion problem which depends on the number of operations provided to the tool and the state space size. In order to address this problem, our approach proposes a symbolic search which finds a sequence of potentially useful operations on which the model-checker should be focused.

5 Symbolic search

The proposed symbolic search is performed by an algorithm that looks for an observable sequence $\mathcal{Q} \hat{=} init_0 \; ; \; op_1 \; ; \; \ldots \; ; \; op_m$ executed by a user u, and where (u, R_u, c_m) is not valid for a critical operation op_m in the initial state val_0 but becomes valid for state val_{m-1} where op_m can be enabled. It is a backward search algorithm, starting from the goal state val_{m-1} from which the critical operation op_m can be enabled: $val_{m-1} \hat{=} c_m \wedge Pre(op_m)$; and working backwards until the initial state val_0 is encountered. The algorithm ends when sequence $\mathcal{Q}$ is found or when all operations are verified without encountering the initial state. We consider that val_0 is a completely valuated state such as that where Alice is the owner of m_1, and Bob is a participant to m_1. This prevents the initial state from being included in both states val_{m-1} and val_{m-2}, which would never verify the condition of the while loop. Note that each operation occurs at most once in a computed sequence, which ensures the termination of our algorithm.

```
1. Q ≙ op_m;
2. val_{m-1} ≙ c_m ∧ Pre(op_m);
3. val_{m-2} ≙ ¬val_{m-1};
4. while val_0 ⇏ val_{m-1} do
5.         choose any o_i ∈ O where
6.                 (u, R_u, c_i) ⊨ true ∧
7.                 val_{m-2} ∧ Pre(o_i) ⇒ ¬[Action(o_i)]¬val_{m-1}
8.         do
9.                 Q ≙ o_i ; Q ;
10.                val_{m-1} ≙ val_{m-2} ∧ Pre(o_i);
11.                val_{m-2} ≙ val_{m-2} ∧ ¬Pre(o_i);
12.        else
13.                raise exception: No sequence found
14.        enddo
15. endwhile
16. Q ≙ init ; Q ;
```

5.1 Step by step illustration

We take advantage of abstraction and step by step we refine the val_{m-2} symbolic state:

1. At the first step of the algorithm, the state space is represented by two symbolic states: the first one val_{m-1} includes all states where the authorization constraint c_m is true and which are enabling op_m, and the second one val_{m-2} is the negation of val_{m-1} which is then $\neg c_m \vee \neg Pre(op_m)$. As they are two disjoint state predicates, we conduct proof (1) in order to find an operation o_i that belongs to $\mathcal{O}$ and which possibly reaches the first state val_{m-1} from the second one val_{m-2} and such that premise (u, R_u, c_i) is valid. If o_i does not exist, then no sequence could be found for the expected attack and we

can try proof (2) for each operation attesting that all operations never reach val_{m-1} from val_{m-2}.

2. At the second step of the algorithm, if the proof (1) succeeds for some operation op_{m-1}, then it may exist an observable sequence leading to the critical operation where access control premise (u, R_u, c_m) is valid, and hence a potential symbolic attack scenario can be found. The algorithm looks inside state val_{m-2} in order to find out the previous operations that can be invoked in the attack scenario. State val_{m-2} is partitioned into two sub-states which are:

$$val_{m-2} \wedge Pre(op_{m-1}) \equiv \neg(c_m \wedge Pre(op_m)) \wedge Pre(op_{m-1})$$
$$val_{m-2} \wedge \neg Pre(op_{m-1}) \equiv \neg(c_m \wedge Pre(op_m)) \wedge \neg Pre(op_{m-1})$$

Then, we look for operations that reach the first sub-state from the second one.

3. The algorithm proceeds iteratively by partitioning the second state into two sub-states until it finds a state that includes the initial state. In the best case, our algorithm gives some symbolic attack scenario, which consists of sequence $(init_0\ ;\ op_n\ ;\ op_{n+1}\ ;\ \ldots\ ;\ op_m)$ invoked by the same user u and where:

$$val_{n-1} \hat{=} \neg(c_m \wedge Pre(op_m)) \wedge \neg Pre(op_{m-1}) \wedge \neg Pre(op_{m-2}) \wedge \ldots \wedge Pre(op_n)$$

and such that $val_0 \Rightarrow val_{n-1} \wedge \forall i.(i \in (n..m) \Rightarrow (u, R_u, c_i) \models true)$

5.2 Application

We apply our algorithm to the meeting scheduler example starting from the following initial state val_0:

$$\begin{aligned} val_0 \hat{=}\ & person = \{Alice, Bob\} \\ & \wedge\ meeting = \{m_1\} \\ & \wedge\ meetingOwner = \{(Alice \mapsto m_1)\} \\ & \wedge\ meetingParticipant = \{(m_1 \mapsto Bob)\} \end{aligned}$$

In this state user John is not allowed to modify meeting m_1 because the authorization constraint allows modification only for the owner of m_1. A malicious scenario would lead to a state where John becomes able to execute a modification operation such as operation $meetingSetStart$ on meeting m_1. In this state we have to verify:

$$\begin{aligned} & Pre(meetingSetStart(m_1, start)) \hat{=}\ m_1 \in meeting \wedge start \in NAT \\ & \text{and } (John, SystemUser, MeetingOwner(m_1) = John) \models true \end{aligned}$$

1. First iteration: considering the following symbolic states

$$val_{m-1} = (MeetingOwner(m_1) = John) \wedge m_1 \in Meeting \wedge start \in NAT$$
$$val_{m-2} = \neg val_{m-1}$$

we have:
– $val_0 \not\Rightarrow val_{m-1}$ because, in state val_0, $MeetingOwner(m_1) = Alice$, and
– proof (1) succeeds for the operations meetingNew and personAddMeetingOwner.
Then, we may go on the second iteration of the algorithm for each of these operations.

2. Second iteration: we partition state val_{m-2} into two sub-states:

$$val_{m-2} = \neg val_{m-1} \wedge Pre(op_{m-1})$$
$$val_{m-3} = \neg val_{m-1} \wedge \neg Pre(op_{m-1})$$

– **case 1:** we choose $op_{m-1} = meetingNew$, and then we have:
– $Pre(meetingNew) \hat{=} m_1 \in meeting \wedge John \in person$, and
– $val_0 \not\Rightarrow val_{m-2}$ because $John \notin person$
In this case, the algorithm does not find an operation leading to a state where operation $meetingNew$ becomes enabled. Indeed, no operation satisfied proof obligation (1). Our algorithm concludes that it does not exist an attack scenario invoking $meetingNew$ at this step.

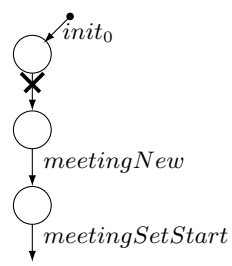

Fig. 5. No state enables operation $meetingNew$ is found

– **case 2:** we choose $op_{m-1} = personAddMeetingOwner$ and then we have:
– $Pre(personAddMeetingOwner) =$
$$m_1 \in meeting \wedge John \in person \wedge (John, m_1) \notin MeetingOwner$$
– $val_0 \not\Rightarrow val_{m-2}$ because $John \notin person$
In this case, proof (1) succeeds for operation $personNew$ which means that if $personNew$ is executed, it may lead to a state where $meetingNew$ can be enabled.

3. Third iteration: we partition state val_{m-3} into two sub-states:

$$val_{m-3} = \neg val_{m-1} \wedge \neg Pre(personAddMeetingOwner) \wedge Pre(personNew)$$
$$val_{m-4} = \neg val_{m-1} \wedge \neg Pre(personAddMeetingOwner) \wedge \neg Pre(personNew)$$

This stops normally the algorithm because in this case $val_0 \Rightarrow val_{m-3}$. Indeed, $Pre(personNew) \hat{=} John \notin person$ and in the initial state John does not belong to set $person$. Figure 6 presents the full symbolic scenario that allows John to modify Alice's meeting.

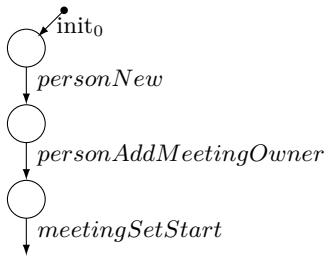

Fig. 6. Symbolic malicious behaviour for user John.

5.3 Discussion

Technically, our approach applies the GeneSyst tool in order to produce proof obligations and then asks the AtelierB prover to discharge them automatically. As the resulting scenarios are symbolic and based on "possibly reached proofs", the analyst can conclude that attacks may exist but he can not attest their feasibility for the concrete system. An interesting contribution of our proof-based symbolic sequences, besides the fact that they draw the analyst's attention to potential flaws, is that they give useful inputs to the model-checker. Indeed, a model-checking tool can be used to exhibit, from a symbolic behaviour, an observable concrete sequence of operations that can be followed by an attacker. In order to reduce significantly the state space, we can ask ProB to explore only operations found in the symbolic malicious scenarios. For our example, when trying only operations personNew, personAddMeetingOwner and meetingSetStart, ProB exhibits a concrete attack scenario after visiting a dozen of states which shows a significant speed up with respect to our initial ProB attempts (involving more than 1000 states).

Our technique was able to extract another scenario (figure 7) which can be executed by user Bob from the same initial state, in order to steal the ownership of m_1. In this scenario, Bob first cancels the meeting and then he recreates it before applying the critical operation.

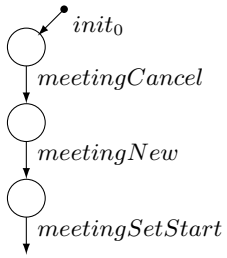

Fig. 7. Symbolic malicious behaviour for user Bob.

The first scenario, done by user John, is made possible by the full access permission to class Person, associated to role SystemAdministrator, which includes the right to modify association ends. This attack affects meeting integrity. One

solution can be to add a SSD constraint between roles SystemAdministrator and SystemUser. John will then still be able to become owner of the meeting, but will not be able to log in as SystemUser in order to modify it.
The second scenario done by Bob was possible due to role Supervisor which gives him the right to cancel a meeting, and then, as a SystemUser he can recreate it in order to become its owner. This scenario does not point out a flaw since whenever a meeting is cancelled it should be legitimate that a user can start a new meeting with the same identifier as the cancelled one.

6 Conclusion

We described in this paper a symbolic search approach that can extract insider malicious behaviours from a formal Information System modelling. The meeting scheduler example was discussed in several articles [2, 3]. However, they do not report the attack scenarios presented in this paper. This is due to the fact that dynamic evolution of the functional state is not taken into account. Contributions of this paper showed how dynamic analysis, assisted by proofs and model-checking, is useful to find out potential threats. In addition, thanks to our algorithm, proofs and model checking tools, our method can be fully automated in order to extract attack scenarios breaking authorization constraint. In [10], a dynamic analysis is done interactively with the help of a Z animator, but it is tedious and may miss many possible flaws. We also applied our approach on the case study that has been treated in [8] and we were able to find, automatically, the discussed threat. Currently we are looking for application on a real case study, issued from the ANR-Selkis project[7], and which deals with a medical information system involving various authorization constraints.

Our approach is automated by exploiting tools B4MSecure[8], GeneSyst[9], AtelierB[10] and ProB[11]. B4MSecure translates functional and security graphical models into B specification, from which we automatically produce proof obligations on reachability properties by taking advantage of the GeneSyst tool. Then, these proof obligations are discharged automatically using the AtelierB prover. When a symbolic scenario is found, ProB is used to explore concrete state space focusing on operations issued from the symbolic scenario. The main limitation of our work is that sometimes, when proof obligations are complex, AtelierB fails to prove them automatically. Interactive proofs are then required, but they may be pretty difficult for the analyst. One naive solution is to keep operations for which proofs don't succeed automatically in order to be exploited further using the model-checker. A more interesting solution is to focus on other kinds of proof obligations. For example, one can try to prove that an operation o is never enabled from a state E and/or o never reaches a state F. Applying

[7] http://lacl.univ-paris12.fr/selkis
[8] http://b4msecure.forge.imag.fr
[9] http://perso.citi.insa-lyon.fr/nstouls/?ZoomSur=GeneSyst
[10] http://www.atelierb.eu/
[11] http://www.stups.uni-duesseldorf.de/ProB

these proofs to the meeting scheduler example we were able to eliminate half of the operations after proving automatically that they cannot be involved in the attack scenario.

We believe that reachability properties can be expressed by means of LTL formula. We started exploring this direction basing on the LTL formula checking facilities of ProB and the first results are promising.

References

1. K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. Uml2alloy: A challenging model transformation. *Model Driven Engineering Languages and Systems*, pages 436–450, 2007.
2. David Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security: From uml models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, 15(1):39–91, January 2006.
3. David A. Basin, Manuel Clavel, Jürgen Doser, and Marina Egea. Automated analysis of security-design models. *Information Software Technology*, 51:815–831, 2009.
4. M-L. Potet D. Bert and N. Stouls. GeneSyst: a Tool to Reason about Behavioral Aspects of B Event Specifications. Application to Security Properties. In H. Treharne, S. King, M. C. Henson, and S. A. Schneider, editors, *ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users*, volume 3455 of *Lecture Notes in Computer Science*, pages 299–318. Springer-Verlag, 2005.
5. Akram Idani, Jean-Louis Boulanger, and Laurent Philippe. Linking paradigms in safety critical systems. *International Journal of Computers and their Applications (IJCA), Special Issue on the Application of Computer Technology to Public Safety and Law Enforcement*, 16(2):111–120, 2009.
6. Mirco Kuhlmann, Karsten Sohr, and Martin Gogolla. Employing uml and ocl for designing and analysing role-based access control. *Mathematical Structures in Computer Science*, 23(4):796–833, 2013.
7. Kevin Lano, David Clark, and Kelly Androutsopoulos. UML to B: Formal Verification of Object-Oriented Models. In *Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, pages 187–206. Springer, 2004.
8. Yves Ledru, Akram Idani, Jérémy Milhau, Nafees Qamar, Régine Laleau, Jean-Luc Richier, and Mohamed-Amine Labiadh. Taking into account functional models in the validation of is security policies. In Camille Salinesi and Oscar Pastor, editors, *CAiSE Workshops*, volume 83 of *Lecture Notes in Business Information Processing*, pages 592–606. Springer, 2011.
9. Yves Ledru, Akram Idani, Jérémy Milhau, Nafees Qamar, Régine Laleau, Jean-Luc Richier, and Mohamed-Amine Labiadh. Validation of IS security policies featuring authorisation constraints. *International Journal of Information System Modeling and Design (IJISMD)*, 2014.
10. Yves Ledru, Nafees Qamar, Akram Idani, Jean-Luc Richier, and Mohamed-Amine Labiadh. Validation of security policies by the animation of z specifications. In *Proceedings of the 16th ACM Symposium on Access Control Models and Technologies*, SACMAT '11, pages 155–164, New York, NY, USA, 2011. ACM.
11. M. Leuschel and M. Butler. ProB: A Model Checker for B. In *FME 2003: Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874. Springer-Verlag, 2003.

12. Muhammad Nafees Qamar, Yves Ledru, and Akram Idani. Evaluating RBAC Supported Techniques and their Validation and Verification. In *6th International Conference on Availability, Reliability and Security (ARES 2011)*, pages 734–739, Vienna, Autriche, August 2011. IEEE Computer Society.
13. C. Snook and M. Butler. UML-B: Formal modeling and design aided by UML. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1):92–122, 2006.