A Process-Centric Approach to Insider Threats Identification in Information Systems

Akram Idani, Yves Ledru, German Vega

A Process-Centric Approach to Insider Threats Identification in Information Systems

Akram Idani[0000−0003−2267−3639], Yves Ledru, and German Vega

Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, F-38000 Grenoble France
firstname.lastname@univ-grenoble-alpes.fr

Abstract. The development of complex software systems as done today generates countless security vulnerabilities that are difficult to detect. In this context, several research works have adopted the Model Driven Security (MDS) approach, which investigates software models rather than implementations. However, although these works provide useful techniques for security modeling and validation, they do not address the impact of functional behavior on the security context of the system, which can be cause for several flaws, specially insider threats. In order to address this challenge, we propose a dynamic analysis based on the B method for both functional and security concerns. Our contribution extends the B4MSecure platform that we developed in our previous works, by introducing a workflow-centric layer to model expected business processes, as well as possible malicious activities. This new layer is built on CSP||B and brings new validation possibilities to B4MSecure.

Keywords: B Method, CSP, Verification, RBAC, Access Control.

1 Introduction

Computer security often refers to hackers or intruders, who are persons with high technical skills and whose intention is to exploit security breaches in order to get an illegal access to a system. However, in reality the greatest threats come from inside the system, *i.e.* from trusted users who are already granted a legal access. This kind of threat is called “insider attack” in cyber-security and it is known to be difficult to tackle [14]. Studies done by IBM X-Force Research in the cyber-security landscape state that: “*In 2015, 60 percent of all attacks were carried out by insiders* [...] *and they resulted in substantial financial and reputational losses*”. The problem is beyond the access control frontier since it includes unpredictable human behaviours. To deal with these threats, existing industrial, academic and government studies [6,7,10] elaborate human profiles and advocate for the use of surveillance systems. Without being exhaustive, some of these profiles are:

- Curious persons who, without a malicious intention but without self-control too, get access to sensitive data or do some actions that are in contradiction with the company rules.

- Super-heroes who, in order to fix a problem or help someone, bypass the company policies believing that it may be useful or simply be approved.

Other profiles are established in the literature, like audacious, greedy, disgruntled, opportunistic, etc. Unfortunately, the eventuality of a breach of trust is difficult to predict in advance based on human-centric factors. On the one hand there is no certainty about a possible acting out, and on the other hand people surveillance must comply with privacy legislation, which makes it almost ineffective. Nonetheless, Information Systems (IS) together with their business logic and processes, provide useful knowledge allowing one to deal with the insider threat problem. In fact, based on the aforementioned studies it can be observed that insiders often do not have high computer skills (contrary to intruders), but they have a fine-grained knowledge about the IS procedures. The latter are mostly well-established and already protected via access control mechanisms. Hence, by being able to answer the question "*who has access to sensitive data and what kind of access is given?*", one cannot claim that the system is secure enough. The good question should be: "*is the user able to run a sequence of actions that may bring him from a prohibition to an authorization?*".

The first question refers to static concerns, and it is widely addressed in Model-Driven Security (MDS) thanks to several access control models (*e.g.* SecureUML, UMLSec). However, the second question remains open in MDS because it refers to behavioural features and the reachability of unwanted situations granting to the user misappropriated privileges. In [11], we applied the B method and its composition mechanism, to provide a way to take into account the intertwining of security policies and functional concerns of the IS. The underlying analysis technique builds on a forward search approach that is intended to verify whether a given targeted functional state is reachable, and eliminate threats by proving that a given unwanted state is unreachable. Forward search is a classical approach that is often done thanks to model-checking techniques, but it is more efficient for relatively small applications. As this is not always possible in the case of IS, a model-checker may require some guidance. This paper contributes towards our previous works by taking into account business processes. We propose to extend our approach and its tool support with a process-based search applying CSP||B [4], which would exhibit malicious behaviours more efficiently.

Section 2 provides a formal framework to MDS, which allows us to apply automated reasoning tools in order to verify the correctness of the IS concerns. Section 3 gives the contribution of this paper and describes how business processes can be taken into account when dealing with insider attacks. Section 4 discusses related works and Section 5 draws the conclusions and the perspectives.

2 Separation of Concerns

This work applies B4MSecure [8], a tool that we developed in order to model the IS as a whole by covering its functional description, and its security policy. The tool generates from these models a formal B specification, allowing one to formally reason about their correctness: functional and security models can be first proved separately, and then integrated in order to verify their interactions.

B4MSecure is built on an MDE architecture in which the input models are UML class diagrams that are extended with the SecureUML profile. The extraction of B specifications from these input models applies transformation rules that are defined at a meta-level. The ideas behind the tool are inspired by existing software products, such as popular commercial database management systems (*e.g.* Oracle, Sybase) or webservers (*e.g.* JBoss, Tomcat). The available implementations of RBAC act like a filter which intercepts a user request to a resource in order to permit or deny the access to associated functional actions (*e.g.* transactions on databases, file operations, etc). The tool is based on the same principles, but at a modeling level. Each functional operation is encapsulated in a secure operation checking that the current user has the required authorizations.

2.1 Functional Modeling

To illustrate our approach, we consider the UML class diagram of Figure 1. This model is inspired by [1], and represents functional concerns of a banking IS: it deals with customers (class *Customer*) and their accounts (class *Account*).

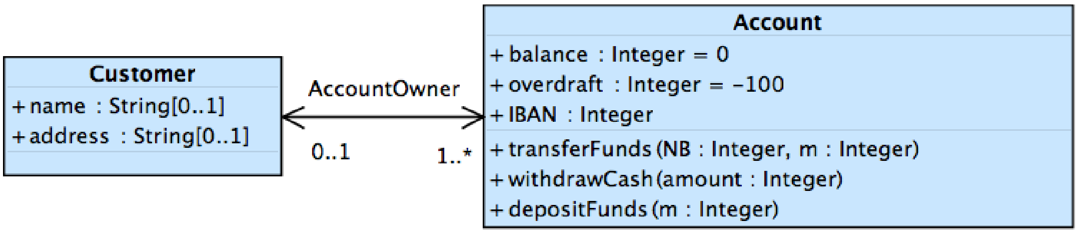

Customer
+name : String[0..1]
+address : String[0..1]
AccountOwner
0..1
1..*
Account
+balance : Integer = 0
+overdraft : Integer = -100
+IBAN : Integer
+transferFunds (NB : Integer, m : Integer)
+withdrawCash (amount : Integer)
+depositFunds (m : Integer)

Fig. 1. UML Class Diagram – Functional Model

A bank account is characterized by its balance (attribute *balance*), the authorized overdraft (attribute *overdraft*) and a unique identifier (attribute *IBAN*). Operation *transferFunds* allows one to transfer an amount of money (parameter *m*) from the current account to any account defined with an *IBAN* number (parameter *NB*). Operations *withdrawCash* and *depositFunds* allow respectively to withdraw or to deposit money.

Translation into B. The translation of this diagram into B follows well-established UML-to-B rules. First, a UML class (*e.g.Customer*) produces: (*i*) an abstract set[1] (*CUSTOMER*) defining the set of possible instances; (*ii*) a variable[2] (*Customer*) defining existing instances; and (*iii*) an invariant[3] meaning that the set of existing instances is a subset of the set of possible instances ($Customer \subseteq CUSTOMER$). Regarding class attributes, they are translated into

[1] clause SETS.

[2] clause VARIABLES.

[3] clause INVARIANT.

B functions relating the set of existing instances to the type of the attribute. The resulting functions depend on the attribute character: mandatory or optional, unique or not unique, single or multi-valuated. For example, attribute *IBAN* of class *Account* is single-valued, mandatory and unique; it is translated as a total injection function. The translation of associations follows the same principle. Indeed, each association leads to a functional relation that depends on the multiplicities of the two ends of the association. For example, association *AccountOwner* is translated into a partial surjective function since its multiplicities are `0..1` and `1..*`. Figure 2 presents the typing invariants that are automatically produced by B4MSecure from Figure 1.

INVARIANT

$$\begin{array}{l}
Account \subseteq ACCOUNT \\
\wedge\ Customer \subseteq CUSTOMER \\
\wedge\ AccountOwner \in Account \mathrel{+\!\!\!\twoheadrightarrow} Customer \\
\wedge\ Account\_\_balance \in Account \rightarrow \mathbb{Z} \\
\wedge\ Account\_\_overdraft \in Account \rightarrow \mathbb{Z} \\
\wedge\ Customer\_\_name \in Customer \mathrel{+\!\!\!\rightarrow} STRING \\
\wedge\ Customer\_\_address \in Customer \mathrel{+\!\!\!\rightarrow} STRING \\
\wedge\ Account\_\_IBAN \in Account \rightarrowtail \mathbb{N}
\end{array}$$

Fig. 2. Structural invariants produced by B4MSecure

Basic operations. The B specifications produced by B4MSecure from a given class diagram are intended to be animated using an animation tool such as ProB [12]. This allows one to see the evolution of the IS and observe the impact that an execution scenario could have on the functional state. Thus, B4MSecure generates all basic operations such as creation/deletion of class instances, creation/deletion of links between these instances, getters/setters of attributes and links, etc. In general, these operations are correct by construction, meaning that they do not violate the generated typing invariants. In fact, the proof of correctness of the functional model means that basic operations preserve the multiplicities of the associations as well as the character of attributes. Figure 3 gives an example of a basic operation that is generated by B4MSecure. It is a creation operation of class *Account*. This operation preserves, on the one hand, the mandatory character of attribute *IBAN* because a value is assigned to the attribute when the object is created, and on the other hand, the uniqueness of this attribute. The operation also takes into account the default values of attributes *balance* and *overdraft*; they are respectively initialized to 0 and -100.

User-defined concerns. The user may introduce within the resulting formal specification invariant properties and the underlying preconditions in order to keep correct the basic operations. The user-defined operations, such as operations *transferFunds* and *withdrawCash* of class *Account*, must also be defined. Figure 4 presents the B specification of operation *transferFunds*. It takes an account number (parameter N) and a positive amount (parameter m) and performs the

```
**Account_NEW**$(Instance,\ Account\_\_IBANValue) ==$
**PRE**
    $Instance \in ACCOUNT \land Instance \notin Account$
     $\land \quad Account\_\_IBANValue \in \mathbb{N}$
     $\land \quad Account\_\_IBANValue \notin \mathbf{ran}(Account\_\_IBAN)$
**THEN**
    $Account := Account \cup \{Instance\}$
    $||\ Account\_\_balance := Account\_\_balance \cup \{(Instance \mapsto 0)\}$
    $||\ Account\_\_overdraft := Account\_\_overdraft \cup \{(Instance \mapsto \text{-100})\}$
    $||\ Account\_\_IBAN := Account\_\_IBAN \cup \{(Instance \mapsto Account\_\_IBANValue)\}$
**END**;
```

Fig. 3. Basic creator generated by B4MSecure

transfer of funds if the following conditions are met: the current account and the beneficiary account are held by customers, N corresponds to an existing account other than the current account, and the authorized overdraft will not be exceeded by this transfer.

```
**Account__transferFunds**$(Instance,\ N,\ m) ==$
**PRE**
    $Instance \in Account \land N \in \mathbb{N} \land m \in \mathbb{N}_1$
     $\land \quad AccountOwner[\{Instance\}] \neq \emptyset$
     $\land \quad N \in \mathbf{ran}(\{Instance\} \lhd Account\_\_IBAN)$
     $\land \quad AccountOwner[\{Account\_\_IBAN^{-1}\ (N)\}] \neq \emptyset$
     $\land \quad Account\_\_balance(Instance) - m \geq Account\_\_overdraft(Instance)$
**THEN**
    $Account\_\_balance :=$
        $\{(Instance \mapsto (Account\_\_balance(Instance) - m))\}$
        $\cup \{(Account\_\_IBAN^{-1}(N) \mapsto (Account\_\_balance(Account\_\_IBAN^{-1}(N)) + m))\}$
        $\cup (\{Instance,\ Account\_\_IBAN^{-1}\ (N)\} \lhd Account\_\_balance)$
**END** ;
```

Fig. 4. Operation transferFunds of class *Account*

2.2 Security Modeling

SecureUML is an extension to UML Class Diagrams whose concrete syntax is based on UML stereotypes. In the version we consider, the main ones used are Role and Permission, where Permission is an association class between roles and functional classes, and can be further annotated with Authorization Constraints. The latter are logical predicates denoting the conditions under which a given permission holds. Figure 5 is a SecureUML model associated to the class diagram of Figure 1. This model defines two roles: *CustomerUser* and *AccountManager*. They respectively represent the customer of the system and the financial manager in charge of the bank's customers. Customers can read their personal data (permission *CustomerUserPerm1*), transfer money, deposit and withdraw cash (permission *CustomerUserPerm2*). The account manager has a full access (read and write) on class *Customer* (permission *AccountManagerPerm1*). He/She can thus create customers, read or modify their data. However, his/her rights on class *Account* are limited to the creation of new accounts (permission *AccountManagerPerm2*). Furthermore, an authorization constraint is associated to permissions *CustomerUserPerm1* and *CustomerUserPerm2* in order to grant the corresponding actions to the sole holder of the account on which they are invoked. In this security policy, the account manager has no access, neither read nor write, to the attributes of class *Account*.

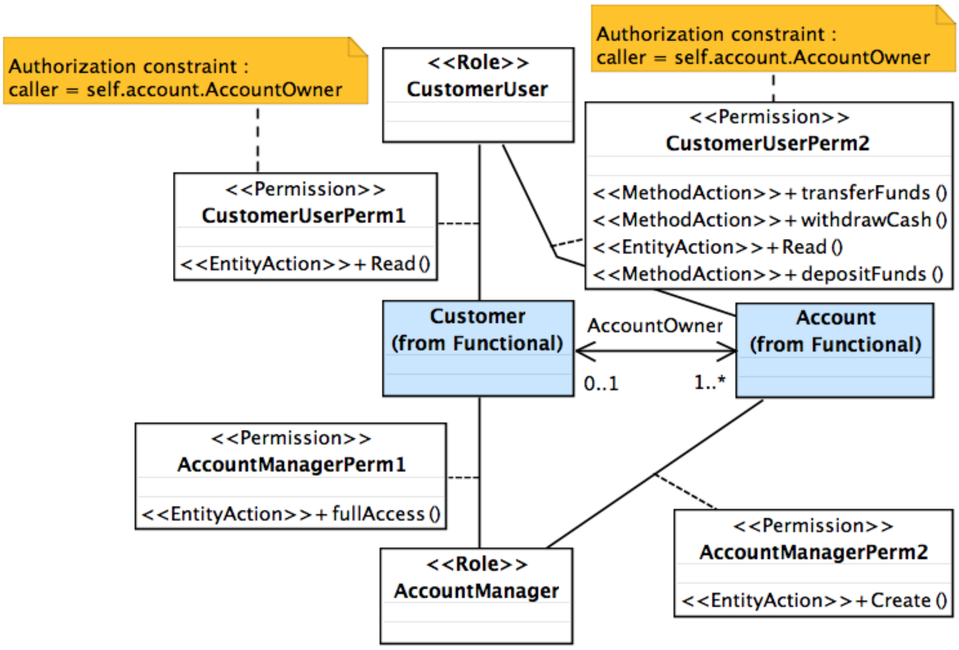


Fig. 5. Security modeling with SecureUML

Translation into B. Given a SecureUML model, B4MSecure produces a B machine that grants permission or forbids functional operations based on the set of roles that are activated by a user. For example, if *Paul* is a *CustomerUser*, he can only read his personal data by calling getters of class *Customer*. The other operations (modification, creation, etc) are forbidden to him. In order to translate the security model, our approach follows two steps: (*i*) propose a "stable" formalization of the SecureUML meta-model, then (*ii*) translate a given security model and inject it into the formalization of the meta-model. Each operation of the functional model is encapsulated in a secure operation checking that the current user is allowed (or not) to call this operation. Figure 6 presents an excerpt of the resulting B machine that is dedicated to user assignments. It refers to: *ROLES*, *USERS* and *SESSIONS*. The assignment of roles to users is defined with relation *roleOf*. Contrary to users and roles, which are explicitly represented by sets, sessions are defined by means of a relation between users and roles. We consider that a user cannot open several sessions in the system. When a user *u* belongs to the domain of relation *Session*, thus a session is created for him and *Session*$[\{u\}]$ gives the set of roles that are activated by u. Variable *currentUser* is useful during the animation of the model because it allows us to identify the user who is running a given operation. Machine *SecureUML* provides also several utility operations that are useful during the animation. For example, Figure 7 gives the B specifications of operations *Connect* and *setCurrentUser*. Operation *Connect* creates a session to a given user in which a set of roles is activated. This operation is done under two conditions: (*i*) the user is not concerned by any existing session, (*ii*) if a role r_1 is a super-role of a role r_2, therefore the user can activate r_1 or r_2 but not both of them $\{r_1, r_2\}$. Operation *setCurrentUser* selects the user who is currently concerned with the animation.

```
MACHINE SecureUML INCLUDES Functional_Model
SETS ROLES ; USERS
VARIABLES roleOf, Roles_Hierarchy, currentUser, Session
INVARIANT
/* Typing invariants */
    currentUser ∈ USERS
    ∧ roleOf ∈ USERS → ℙ (ROLES)
    ∧ Roles_Hierarchy ∈ ROLES ↔ ROLES
    ∧ Session ∈ USERS ↔ ROLES
/* No cycles in role hierarchy */
    ∧ (Roles_Hierarchy)⁺ ∩ id(ROLES) = ∅
/* Conformance of role assignments and role activation */
    ∧ ∀ (uu).(uu ∈ USERS ∧ uu ∈ dom(Session) ⇒ Session[{uu}] ⊆ roleOf(uu))
```

Fig. 6. Excerpt of the SecureUML meta-model dedicated to user assignments

```
Connect(user, roleSet) =
PRE
  user ∈ USERS ∧ user ∉ dom(Session)
  ∧ roleSet ∈ ℙ₁(ROLES) ∧ roleSet ⊆ roleOf(user)
  /* avoid hierarchical redundancy in the roleSet */
  ∧ ∀ (r1,r2).(r1 ∈ roleSet ∧ r2 ∈ roleSet ∧ r1 ≠ r2 ⇒ r2 ∉ (Roles_Hierarchy)⁺[{r1}])
THEN
  Session := Session ∪ ({user} × roleSet)
END;
```

```
setCurrentUser(user) =
    PRE
        user ∈ USERS ∧ user ≠ currentUser ∧ user ∈ dom(Session)
    THEN
        currentUser := user
    END ;
```

Fig. 7. Utility operations

B4MSecure produces for every functional operation, a secured operation that verifies (using a security guard) whether the current user is allowed to call the functional operation. The secure operation also verifies the authorization constraints, if they are defined in the underlying permissions, and updates the assignment of roles when required. Figure 8 shows the secure operations associated to *Account_transferFunds* . The security guard is defined in clause *SELECT*. It verifies that the functional operation belongs to set $isPermitted[currentRoles]$, where definition *currentRoles* refers to the roles activated by *currentUser* (in a session) as well as their super-roles: $currentRoles == Session[\{currentUser\}] \cup \mathbf{ran}(Session[\{currentUser\}] \lhd (Roles\_Hierarchy)^{+})$.

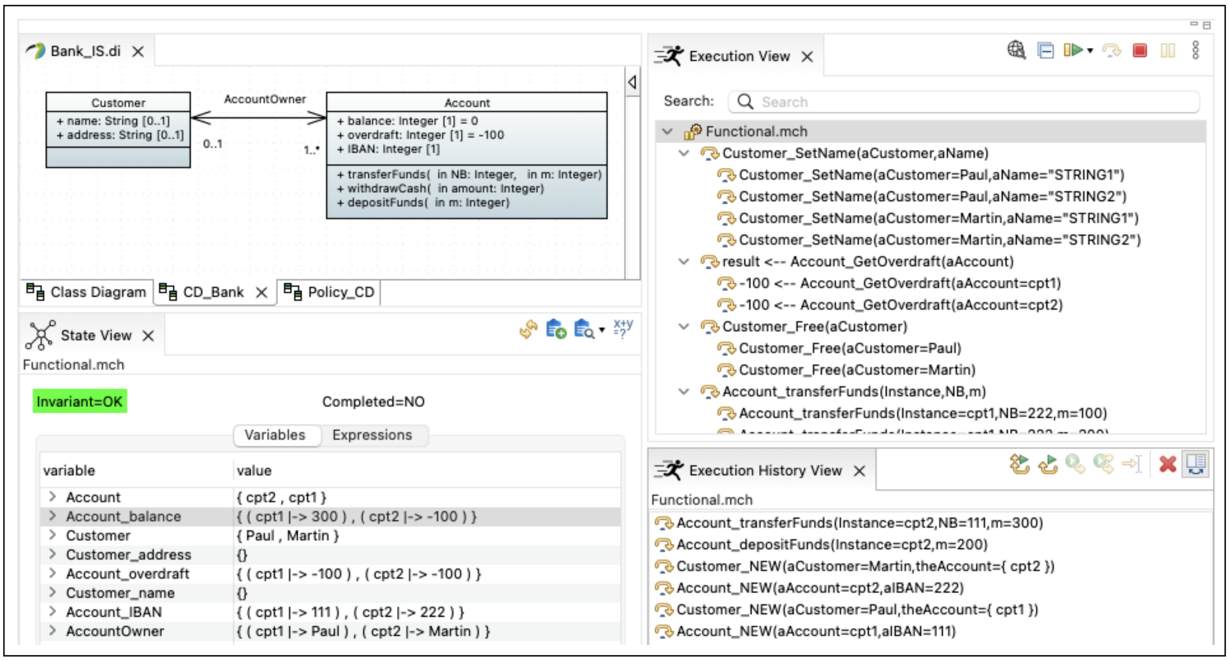
```
$\mathbf{secure\_Account\_transferFunds}(aAccount, NB, m) =$
    $\mathbf{SELECT}$
        $Account\_transferFunds\_ \in isPermitted[currentRoles]$
        $\wedge\ (CustomerUser \in currentRoles \Rightarrow AccountOwner(aAccount) = currentUser)$
    $\mathbf{THEN}$
        $\mathbf{Account\_transferFunds}(aAccount, NB, m)$
    $\mathbf{END};$
```

Fig. 8. Operation *secure_Account_transferFunds*

3 Dealing with Business Processes

3.1 Animation in B4MSecure

In order to validate the model, B4MSecure uses ProB Java API as presented in Figure 9.



Functional.mch

Invariant=OK Completed=NO

variable	value
Account	{ cpt2 , cpt1 }
Account_balance	{ (cpt1 |-> 300) , (cpt2 |-> -100) }
Customer	{ Paul , Martin }
Customer_address	{}
Account_overdraft	{ (cpt1 |-> -100) , (cpt2 |-> -100) }
Customer_name	{}
Account_IBAN	{ (cpt1 |-> 111) , (cpt2 |-> 222) }
AccountOwner	{ (cpt1 |-> Paul) , (cpt2 |-> Martin) }

```
Functional.mch
  Customer_SetName(aCustomer,aName)
    Customer_SetName(aCustomer=Paul,aName="STRING1")
    Customer_SetName(aCustomer=Paul,aName="STRING2")
    Customer_SetName(aCustomer=Martin,aName="STRING1")
    Customer_SetName(aCustomer=Martin,aName="STRING2")
  result <-- Account_GetOverdraft(aAccount)
    -100 <-- Account_GetOverdraft(aAccount=cpt1)
    -100 <-- Account_GetOverdraft(aAccount=cpt2)
  Customer_Free(aCustomer)
    Customer_Free(aCustomer=Paul)
    Customer_Free(aCustomer=Martin)
  Account_transferFunds(Instance,NB,m)
    Account_transferFunds(Instance=cpt1,NB=222,m=100)
```

```
Functional.mch
Account_transferFunds(Instance=cpt2,NB=111,m=300)
Account_depositFunds(Instance=cpt2,m=200)
Customer_NEW(aCustomer=Martin,theAccount={ cpt2 })
Account_NEW(aAccount=cpt2,aIBAN=222)
Customer_NEW(aCustomer=Paul,theAccount={ cpt1 })
Account_NEW(aAccount=cpt1,aIBAN=111)
```

Fig. 9. Animation in B4MSecure

The State View (bottom-left) gives the values of the B variables in the current state, *i.e.* after animating the sequence of the History View (bottom-right).

The Execution View (top-right) shows the B operations that can be triggered in the current state. The content of these views is computed by the Java API of ProB; B4MSecure just provides some convenient actions to ensure animation and/or model-checking via the API. Figure 9 shows a sequence of functional operations that creates customers Paul and Martin, as well as their respective accounts cpt_1 and cpt_2. An amount of 200€ is added to Martin's account and then 300€ are transferred from this account to Paul's account. This scenario corresponds to a normal use case of the IS without considering security concerns. Playing with functional scenarios shows that use cases are feasible with the current specification, and helps identifying missing steps in the use cases or the specification. A similar animation can be performed by calling the secured version of the use case. This eases the understanding and validation of the security policy and shows that the security policy does not prevent the execution of functional use cases.

3.2 A CSP||B Approach

One facility supported by ProB is the use of CSP||B, that is, a CSP layer that guides the animation of the B machine. This guidance restricts the execution space to relevant traces with respect to pre-established processes, which would make verification potentially faster. In CSP, a process refers to a sequence of events and the communication between processes is ensured via channels. A channel `ch` may transmit data `d`, which is denoted as `ch?d` for inputs, and `ch!d` for outputs. Note that by convention, processes are named in uppercase and channels in lowercase. Some of the used CSP constructs are:

```
PROCESS ::= SKIP                /* terminating process */
          | ch -> PROCESS       /* simple action prefix where ch is a channel */
          | PROCESS ; PROCESS   /* sequential composition */
          | PROCESS [] PROCESS  /* external choice */
```

In the CSP||B approach [4] processes are used as controllers for a B machine where channels correspond to B operations, and events to a call to the operation, with channel inputs and outputs being the operation's parameters. Our formal framework, ensured with B4MSecure, favours the integration of a process-centric approach for animation. Let's consider for example the business process of Figure 10 written in CSP. Process `UI` is a loop where first a user is connected and next, depending on the activated role, he/she executes process `MANAGER_FUNC` or process `CLIENT_FUNC`. For space reason we only show the former. The manager has the choice between creating a new account or a new customer, or updating an existing customer record. The `SKIP` process terminates process `MANAGER_FUNC` and hence operation `disconnecUser` of process `UI` is executed, which disconnects the user.

Having this CSP model, ProB can be used to animate the operational formal model of the security policy, by following traces allowed by the CSP processes. The sequence of operations given in Figure 11 corresponds to a normal use case

```
MAIN = UI

UI = (Connect?user!{AccountManager} -> setCurrentUser(user) -> MANAGER_FUNC
     [] Connect?user!{CustomerUser} -> setCurrentUser(user) -> CLIENT_FUNC)
     ;  disconnectUser -> UI

MANAGER_FUNC =
        CREATE_ACCOUNT [] CREATE_CUSTOMER [] UPDATE_CUSTOMER [] SKIP

CREATE_ACCOUNT =
        secure_Account_NEW -> (CREATE_ACCOUNT [] MANAGER_FUNC)

CREATE_CUSTOMER =
        secure_Customer_NEW?customer -> secure_Customer__SetName!customer
        -> (ADD_CUSTOMER_ACCOUNT(customer) [] CREATE_CUSTOMER [] MANAGER_FUNC)

ADD_CUSTOMER_ACCOUNT(customer) =
        secure_Customer__AddAccountOwner!customer
        -> (ADD_CUSTOMER_ACCOUNT(customer) [] MANAGER_FUNC)

UPDATE_CUSTOMER =
        secure_Customer__GetName?customer -> UPDATE(customer)

UPDATE(customer) =
        secure_Customer__SetName?customer -> UPDATE(customer)
        [] secure_Customer__SetAddress?customer -> UPDATE(customer)
        [] secure_Customer__RemoveAccountOwner?customer -> UPDATE(customer)
        [] ADD_CUSTOMER_ACCOUNT(customer)
        [] MANAGER_FUNC
```

Fig. 10. Business process in CSP

where users execute permitted operations after they activate the right role and such that they satisfy functional preconditions as well as authorization constraints. In this trace, we show for every step the CSP process that is considered by the animator. The resulting state is represented with the object diagram of Figure 12. In this sequence, the account manager Bob creates two customers (Paul and Martin) and two accounts (cpt_1 and cpt_2) and then Paul deposits money into his own account.

3.3 Insider threats identification

One major advantage of B is theorem proving, which refers to the demonstration of logical formulas (called proof obligations, POs) to ensure a correctness claim for a given property (such as an invariant property). For example, the correctness of an operation guarantees that the invariant is true before and after the execution of the operation. For our example, we proved the correctness of our

```
-> Process: MAIN -> UI
       Connect(Bob, {AccountManager}) ;
       setCurrentUser(Bob) ;
-> Process: MANAGER_FUNCTIONS -> CREATE_ACCOUNT
       secure_Account_NEW(cpt1, 111) ;
       secure_Account_NEW(cpt2, 222) ;
-> Process: MANAGER_FUNCTIONS -> CREATE_CUSTOMER
       secure_Customer_NEW(Paul,{cpt1}) ;
       secure_Customer_SetName(Paul,"Paul Durand") ;
       secure_Customer_NEW(Martin,{cpt2})
       secure_Customer_SetName(Martin,"Martin Favier") ;
-> Process: MANAGER_FUNCTIONS -> SKIP
       disconnectUser
-> Process: UI
       Connect(Paul, {CustomerUser}) ;
       setCurrentUser(Paul) ;
-> Process: CLIENT_FUNCTIONS -> DEPOSIT
       secure_Account__depositFunds(cpt1,500)
-> Process: CLIENT_FUNCTIONS -> SKIP
       disconnectUser
```

Fig. 11. Execution trace

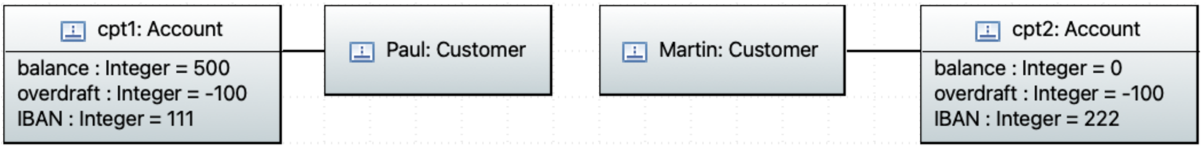


Fig. 12. Resulting state represented with an object diagram

B specifications. The advantage is that when looking for threats, the security analyst has the guarantee that flaws are not issued from invariant violations, but rather from the functional or the security logic. In this sense, the identification of attack scenarios is mainly a validation task, which can be done by animation and/or model-checking. Indeed, an exhaustive model exploration may exhibit a malicious sequence of operations leading to a state (where a property holds) that represents an unwanted situation.

To exhibit a malicious scenario based on our simple example, we start the exploration from a normal state, that of Figure 12. In this state *Paul* is a customer and owns account cpt_1 whose balance is equal to 500. *Bob* as *AccountManager* cannot execute operations *transferFunds* or *withdrawCash* on cpt_1. The answer to a static query such as "Is Bob able to transfer funds from Paul's account?" would be NO, since the permission given to a manager on class *Account* only allows instance creation. In fact, the good question should be "Is there a sequence of operations that can be executed by Bob in order to become able to transfer funds from Paul's account?". To answer the question, one naive solution is to use the model-checking feature of ProB to exhaustively explore the state space and find states that satisfy property: $AccountOwner(\mathrm{cpt}_1) = \mathrm{Bob}$. We are therefore looking for a sequence of operations executed by Bob allowing him to become the owner of cpt_1 meaning that he may reach a state granting him the permission to execute an action that initially he cannot do. Without considering the CSP guidance, ProB reached a time out after exploring millions of transitions, meaning that the state space is too big to be explored efficiently.

To solve this issue we propose to describe insider threats using a CSP model and take benefit of the CSP||B approach of ProB to check if the business process contains traces that are conformant to this attack model. Figure 13 shows the proposed insider threat model for the example discussed above. Statement `[]x:Set(S)@P` used in process `ATTACKER` is a replicated external choice. This statement evaluates process `P` for each value of set `S` and composes the resulting processes together using external choice. Hence, process `ATTACKER` means that Bob is trying to connect to the system by varying his roles. Statement `P|||Q` used in process `ATTACK` is an interleaving, which runs `P` and `Q` in parallel without any synchronisation. In fact, the goal of the attack is to run operation *secure_transferFunds* on account cpt_1 by the attacker who abuses his/her roles.

```
MAIN = UI [|{| Connect, secure_Account_transferFunds |}|] ATTACK

ATTACK = ATTACKER ||| secure_Account_transferFunds!cpt1 -> goal -> SKIP

ATTACKER = []role:Set(ROLES) @ Connect!Bob!role -> ATTACKER
```

Fig. 13. Insider threat model

The synchronisation of process `UI` with the attack model is done in the `MAIN` process. The latter applies a generalized parallel composition with synchronisation on critical actions. In fact, statement `P[|A|]Q` runs processes `P` and `Q` in parallel forcing them to synchronise on events in `A`; any event not in `A` may be performed by either process. In other words, if event `goal` is produced by `ATTACK` therefore the critical operation has been also executed by process `UI`; which means that a flaw conformant to the attack model is detected.

Based on this model, ProB explored about 70000 states and 200000 transitions and was able to exhibit sequence of Figure 14. We structure it in three steps. In `step 1` *Bob* adds himself to the system as a customer. As the creation of a customer requires at least one account, *Bob* creates a fictive account cpt_3 and then he calls operation *secure_Customer_NEW*. In `step 2`, the attacker becomes the owner of cpt_1. To this purpose he must first remove the link between *Paul* and cpt_1. But, in the functional model a customer must have at least one account, consequently operation $secure\_Customer\_RemoveAccount(Paul, \{cpt_1\})$ is possible only if *Paul* has another account. For this reason, *Bob* creates another fictive account cpt_4 and adds it to *Paul*'s accounts. The last action of `step 2` reaches the malicious state where *Bob* is the owner of cpt_1. Finally, `step 3` realizes the attack.

```
/* step 1: create customer Bob */
    Connect(Bob, {AccountManager}) ;
    setCurrentUser(Bob) ;
    secure_Account_NEW(cpt_3, 333) ;
    secure_Customer_NEW(Bob,{cpt_3}) ;
    secure_Customer_SetName(Bob,"...") ;

/* step 2: get the ownership of Paul's Account */
    secure_Account_NEW(cpt_4, 444) ;
    secure_Customer_AddAccount(Paul,{cpt_4}) ;
    secure_Customer_RemoveAccount(Paul,{cpt_1}) ;
    secure_Customer_AddAccount(Bob,{cpt_1}) ;

/* step 3: attack */
    disConnect(Bob) ;
    Connect(Bob, {CustomerUser}) ;
    secure_Account_transferFunds(cpt_1, 333, 500) ;
```

Fig. 14. Malicious scenario

This malicious scenario can be countered by enhancing the functional model and/or the security model. If the analyst assumes that the flaw is favored by the functional logic, one possible solution would be to introduce the following invariant: $Account\_\_balanceValue \neq 0 \Rightarrow AccountOwner[\{Instance\}] \neq \emptyset$. In fact, operation $secure\_Customer\_RemoveAccount(Paul, \{cpt_1\})$ is the dangerous operation. This invariant means that accounts whose balance is not equal to zero must be owned by a customer. By introducing this invariant, several functional operations must be corrected and proved, such as: *Customer_RemoveAccount* and *Account_SetBalance*. In other words, to remove the ownership relation between cpt_1 and *Paul*, the account of *Paul* must be empty. If the analyst assumes that the flow is favored by the security logic, a possible solution would be to limit the scope of permission *AccountManagerPerm1* because it currently grants a full access to role *AccountManager* on customer's data, including the deletion of his accounts.

4 Related works

Model-Driven Security [3] advocates for the separation of concerns principle and suggests the validation of functional and security models in isolation. Hence, most existing works [5] in MDS are stateless and they mostly validate security policies statically without taking into account the dynamic evolution of the IS. A major contribution of our proposal in MDS is that it favours dynamic analyses, using animation and model-checking, of the interactions between the IS concerns.

As far as we know, works that addressed access control together with a formal method, did not deal with the insider threat problem, such as discussed in this paper. However, we can assume that this kind of threat is a typical reachability problem. In [16], the authors proposed a plain model-checking approach, built on security strategies, in order to check specifications written in the RW (Read-Write) language. However, the proposed algorithm faces scalability issues because the RW language is poor compared to B. A similar approach is proposed in [9] in order to validate access control in web-based collaborative systems. Even though their experiments show that they achieve better results compared to [16], the approach still has a partial coverage of realistic policies.

In [13], the authors proposed two approaches to prove reachability properties in a B formal information system modelling. In the first one, they used substitution refinement techniques based on Morgan's specification statement, and in the second one, they proposed an algorithm that produces a proof obligation in order to prove whether a given sequence of operations reaches (or not) a defined state. However, unlike our approach, they don't search sequences leading to a goal state from an initial one. Their approach starts from a given sequence of operations, and tries to prove its reachability.

Existing works, including our previous work [11,15] in the field, do not deal with business processes, which is indeed a limitation because, in IS, the three concerns (functional models, security policies and business processes) are important. This work introduced the business process dimension via CSP||B, which brings the ability to limit the state space exploration during model-checking.

This work led to the development of an extension of B4MSecure that is used to exhibit execution paths from a B modelling of an IS. The approach has been experimented with several case studies such as the meeting scheduler example discussed in [2], the medical IS studied in [11] and the conference review IS inspired by[16]. For each example, the tool aimed to reach the same malicious goal as handled in the article which addressed the same example, and it was able to extract all reported attacks. Some metrics about these experiments are given in Table 1.

Case study	Operations	Variables	Permissions	Roles	Users	scenarios
Library	13	4	3	2	3	8
Medical IS [11]	15	9	3	4	3	10
Meeting scheduler [2]	23	7	5	3	3	8
Bank IS [1]	31	11	4	2	3	9
Conference Review [16]	48	24	8	3	4	14

Table 1. Summary table of experiments

Several research works have been devoted to the validation of access control policies. They are mainly focused on detecting external intrusion. Recently, the interest to insider attacks grew leading to two categories of validation: stateless and dynamic access control validation. Stateless access control validation is dedicated to validate security policies in a given state without taking into account the dynamic evolution of the IS states. Among these works we can cite the SecureMova tool [2] which models security policies using SecureUML and OCL expressions. In this paper we proposed to take into account business processes in order to identify these attacks. A business process model represents a set of steps in which intrinsically operations concerning the IS data (like reading, modification, etc.) and responsibilities for performing tasks are defined.

5 Conclusion

Authorized actions often lead to evolutions of the functional state, which may favour insider threats. A well known attack that was possible due to evolutions of the functional state is that of 'Société Générale'. This attack resulted in a net loss of $7.2 billion to the bank[4]. The insider circumvented internal security mechanisms to place more than $70 billion in secret, unauthorized derivatives trades. Through authorized actions, he was able to cover up operations he has made on the market by introducing into the functional system fictive offsetting inverse operations, so that the unauthorized trades were not detected. Dynamic analysis is therefore crucial because it would establish that a system evolves as expected and that unwanted situations are not possible.

Perspectives. One major perspective of this work is to identify inconsistencies between functional models, security policies and business processes. Indeed, insider threats may also come from a bad alignment of these models, such as when the access control policy gives more permissions than the actions required by the business process. In this case following the process may hide several authorizations giving the impression that some bad actions are not possible while they can still be done from outside the process, which is a typical example of insider attacks. We also plan to extend the notion of attack models. A security expert, when faced with the challenge of finding a way of performing a malicious operation, will often try to break down the requirements for performing this operation and try to find how to achieve them, one by one. We may translate this problem solving technique with "checkpoints", that is, intermediate steps necessary for the attack to take place, similarly to how a privilege escalation attack is composed of various steps that must be climbed.

B4MSecure addresses the modeling activities and is based on Platform Independent Models (PIM), described using UML models and their associated formal B specifications. However, in addition to modeling notions, MDS also promotes the transformation of the PIM into a PSM. One interesting perspective is to translate the models into concrete security mechanisms of a target infrastructure. In practice, this transformation usually includes manual coding activities. The challenge is therefore to guarantee that security models, graphically designed and formally validated, correspond to a deployed security policy.

[4] The New York Times. French Bank Says Rogue Trader Lost $7 Billion. January 2008.

References

1. Bandara, A., Shinpei, H., Jurjens, J., Kaiya, H., Kubo, A., Laney, R., Mouratidis, H., Nhlabatsi, A., Nuseibeh, B., Tahara, Y., Tun, T., Washizaki, H., Yoshioka, N., Yu, Y.: Security Patterns: Comparing Modeling Approaches. IGI Global (2010)
2. Basin, D., Clavel, M., Doser, J., Egea, M.: Automated analysis of security-design models. Information & Software Technology **51** (2009), http://dblp.uni-trier.de/db/journals/infsof/infsof51.html#BasinCDE09
3. Basin, D., Doser, J., Lodderstedt, T.: Model driven security: From UML models to access control infrastructures. ACM Trans. Softw. Eng. Methodol. **15**(1) (2006). https://doi.org/10.1145/1125808.1125810
4. Butler, M.J., Leuschel, M.: Combining CSP and B for specification and property verification. In: International Symposium of Formal Methods - FM 2005. Lecture Notes in Computer Science, vol. 3582, pp. 221–236. Springer (2005)
5. Geismann, J., Bodden, E.: A systematic literature review of model-driven security engineering for cyber–physical systems. Journal of Systems and Software **169**, 110697 (2020). https://doi.org/https://doi.org/10.1016/j.jss.2020.110697
6. Greitzer, F.L.: Insider Threats: It's the HUMAN, Stupid! In: Proceedings of the Northwest Cybersecurity Symposium. NCS '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3332448.3332458
7. Homoliak, I., Toffalini, F., Guarnizo, J., Elovici, Y., Ochoa, M.: Insight Into Insiders and IT: A Survey of Insider Threat Taxonomies, Analysis, Modeling, and Countermeasures. ACM Computing Surveys **52**(2) (2019)
8. Idani, A., Ledru, Y.: B for Modeling Secure Information Systems - The B4MSecure Platform. In: 17th International Conference on Formal Engineering Methods and Software Engineering - ICFEM. LNCS, vol. 9407, pp. 312–318. Springer (2015)
9. Koleini, M., Ryan, M.: A knowledge-based verification method for dynamic access control policies. In: 13th International Conference on Formal Engineering Methods, ICFEM. LNCS, vol. 6991, pp. 243–258. Springer (2011)
10. Kont, M., Pihelgas, M., Wojtkowiak, J., Trinberg, L., Osula, A.M.: Insider Threat Detection Study. The NATO Cooperative Cyber Defence Centre of Excellence (2018)
11. Ledru, Y., Idani, A., Milhau, J., Qamar, N., Laleau, R., Richier, J.L., Labiadh, M.A.: Validation of IS security policies featuring authorisation constraints. International Journal of Information System Modeling and Design (IJISMD) (2014)
12. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: FME 2003: Formal Methods Europe. LNCS, vol. 2805. Springer-Verlag (2003)
13. Mammar, A., Frappier, M.: Proof-based verification approaches for dynamic properties: application to the information system domain. Formal Asp. Comput. **27**(2), 335–374 (2015). https://doi.org/10.1007/s00165-014-0323-x
14. Probst, C.W., Hunker, J., Gollmann, D., Bishop, M. (eds.): Insider Threats in Cyber Security, Advances in Information Security, vol. 49. Springer (2010). https://doi.org/10.1007/978-1-4419-7133-3
15. Radhouani, A., Idani, A., Ledru, Y., Rajeb, N.B.: Symbolic Search of Insider Attack Scenarios from a Formal Information System Modeling. LNCS Transactions on Petri Nets and Other Models of Concurrency **10**, 131–152 (2015)
16. Zhang, N., Ryan, M., Guelev, D.P.: Synthesising verified access control systems through model checking. Journal of Computer Security **16**(1), 1–61 (2008)