ANR programme ARPEGE 2008

Systèmes Embarqués et Grandes Infrastructures

Projet SELKIS : Une méthode de développement de systèmes d'information médicaux sécurisés : de l'analyse des besoins à l'implémentation.

ANR-08-SEGI-018

Février 2009 - Décembre 2011

Principles of the coupling between UML and formal notations

Livrable numero 3.3
update of work package 3.2

Akram Idani, Yves Ledru, Jean-Luc Richier, Abderrahmen Mokni
Laboratoire d'Informatique de Grenoble

Dimitris Vekris, Catalin Dima
LACL, Université Paris-Est

Aout 2012

Table des matières

Introduction

Ce livrable est une mise à jour du livrable 3.2 dédié aux principes de la formalisation en B (et en Z) des politiques de contrôle d'accès de type RBAC. Dans cette mise à jour nous abordons l'intérêt d'une validation formelle de politiques RBAC et nous étendons notre démarche de formalisation par l'introduction du concept d'organisation au niveau du méta-modèle de sécurité. Ce concept impacte plusieurs aspects de notre formalisation :

- la relation "user assignment" permettant l'affectation des utilisateurs aux rôles : cette relation devient une affectation d'utilisateurs aux rôles qu'ils peuvent jouer dans des organisations
- la relation "permission assignment" permettant l'association des rôles aux permissions : cette relation devient un lien entre rôles, organisations et permissions.
- la notion de session : la connexion d'utilisateurs au système est de ce fait réalisée dans le contexte des organisations dans lesquelles ils peuvent jouer des rôles
- les opérations sécurisées dans le filtre de sécurité
- le calcul pour chaque utilisateur les opérations fonctionnelles qu'il est autorisé d'invoquer pour une session donnée.

Ce livrable est organisé en cinq chapitres :

- Chapitre 1 : rappels des principes de formalisation du noyau RBAC en B développés dans le livrable 3.2
- Chapitre 2 : principes de notre démarche de test d'une politique de sécurité RBAC spécifiée en B
- Chapitre 3 : introduction du concept d'organisation et spécifications B associées
- Chapitre 4 : formalisation du concept de session et validation par l'animation de politiques de contrôles d'accès basé sur les organisations.
- Chapitre 5 : intitulé "Operational Semantics for EB3 and translation to Nu-SMV", illustre une approche de vérification de propriétés à partir de spécifications EB3.

Chapitre 1

Rappels issus du livrable 3.2

1.1 Principes de l'approche

Les principes de l'approche de formalisation du filtre de sécurité en B ont été décrits dans le livrable 3.2 et publiés dans [LIM+11]. L'approche consiste à transformer en B un modèle fonctionnel représenté par un diagramme de classes UML et complété par des politiques de contrôle d'accès basé sur les rôles. Comme illustré par la figure 1.1, l'objectif de notre démarche est d'obtenir un filtre de sécurité, formalisé en B, permettant de contrôler l'accès aux opérations du modèle fonctionnel.

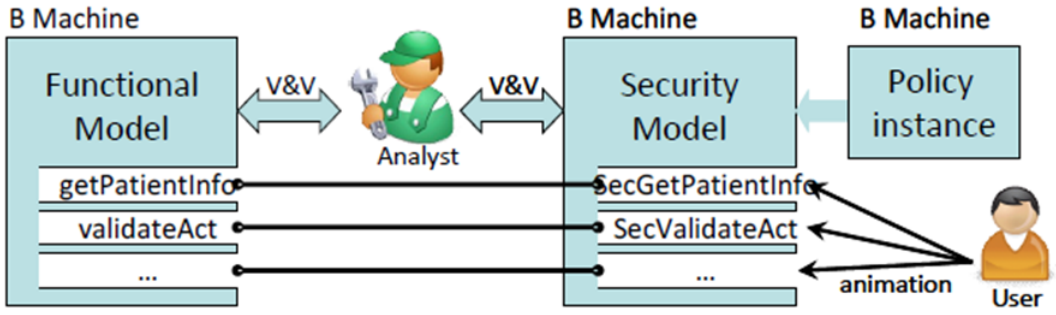


Fig. 1.1 – Principe de la traduction [LIM+11]

Dans l'intérêt de séparer les préoccupations, l'approche vise à dissocier la formalisation en B du modèle fonctionnel de celle du modèle de sécurité. Le processus de dérivation de spécifications formelles produit ainsi deux modèles B :

- Un premier modèle B (Functional_Model) issu du modèle fonctionnel suivant l'approche compilée. Les spécifications résultantes peuvent être enrichies par des contraintes fonctionnelles spécifiques et servent à vérifier la correction du modèle fonctionnel.
- Un deuxième modèle B (Security_Model) traduisant les politiques de contrôle d'accès aux diverses entités fonctionnelles. Ce modèle est généré suivant l'approche interprétée. D'abord, le méta-modèle de sécurité est formalisé en B. Ensuite, les instances du méta-modèle (Policy Instance) sont traduites en B et injectées dans la formalisation statique du méta-modèle de sécurité.

Les spécifications du modèle de sécurité (Security_Model) peuvent être animées en utilisant un animateur tel que ProB [LB08] et permettent à l'utilisateur courant d'exécuter, suivant les droits qu'il possède, certaines opérations du modèle fonctionnel "Functional_Model" faisant ainsi évoluer l'état du système.

1.2 Traduction du modèle fonctionnel

Dans le but d'illustrer les différentes règles de traduction de UML vers B, nous considérons un diagramme (figure 1.2) composé de deux classes *Patient* et *MedicalRecord* liées avec une association indiquant l'enregistrement médical associé à chaque patient.

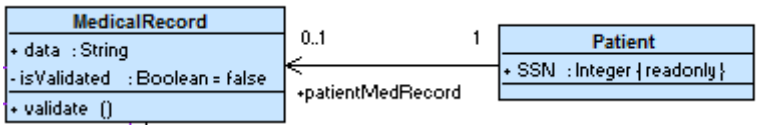


FIG. 1.2 – Exemple d'un modèle fonctionnel

1.2.1 Traduction des classes

Chaque classe du modèle fonctionnel est traduite en deux ensembles. Un ensemble énuméré contenant toutes les instances possibles de la classe et une variable abstraite qui contient les instances effectives (ou créées). Cette dernière est typée par l'ensemble des instances possibles grâce à un prédicat d'inclusion. A titre d'exemple, la classe *MedicalRecord* est traduite comme suit :

MACHINE
Functional_Model
SETS
/*Ensemble des instances possibles*/
MEDICALRECORD;
...
ABSTRACT_VARIABLES
/*Ensemble des instances effectives*/
MedicalRecord ,
...
INVARIANT
$MedicalRecord \subseteq MEDICALRECORD$

1.2.2 Traduction de constructeurs et destructeurs de classes

Un constructeur et un destructeur sont générés pour chaque classe du modèle fonctionnel. Ils sont traduits par des opérations permettant respectivement l'ajout ou la suppression d'un élément de l'ensemble des instances effectives de la classe en question. A titre d'exemple, le constructeur et destructeur de la classe *MedicalRecord* sont générés comme suit :

MedicalRecord_NEW(*Instance*,*patientValue*)=
PRE
Instance ∈ *MEDICALRECORD* ∧ *Instance* ∉ *MedicalRecord*
∧ *patientValue* ∈ *Patient*
/*préconditions relatives aux associations*/
∧ ...
THEN
/*mise à jour de l'ensemble des instances effectives*/
MedicalRecord := *MedicalRecord* ∪ {*Instance*}
/*substitutions relatives aux associations*/
...
END;

MedicalRecord_Free(*Instance*)=
PRE
Instance ∈ *MEDICALRECORD* ∧ *Instance* ∈ *MedicalRecord*
THEN
/*mise à jour de l'ensemble des instances effectives*/
MedicalRecord := *MedicalRecord* - {*Instance*}
/*mise à jour des attributs et associations*/
...
END;

Le constructeur prend en paramètre une instance de la classe *MedicalRecord* (*Instance*) et une instance de la classe *Patient* (*patientValue*). Le deuxième argument découle de l'association entre les classes *Patient* et *MedicalRecord*. Le destructeur prend en paramètre uniquement l'instance à éliminer. Il permet de mettre à jour l'ensemble des instances effectives, les attributs et les associations liées à l'instance. Les préconditions et les substitutions liées aux attributs et aux associations seront discutées dans les sections suivantes.

1.2.3 Traduction des attributs de classe

Les attributs sont traduits par des relations fonctionnelles associant l'ensemble des instances effectives de la classe, au type de l'attribut. Le type est traduit par un ensemble abstrait s'il n'est pas une classe ou il n'est pas pris en compte par le langage B (comme les types NAT, BOOL ou Integer). A titre d'exemple, le type *String* de l'attribut data est traduit par l'ensemble abstrait STR.
Par application de la règle sur la classe *MedicalRecord*, nous obtenons le résultat suivant :

ABSTRACT_VARIABLES
...
, *MedicalRecord_Data*
, *MedicalRecord_IsValidated*
, ...
INVARIANT
...
∧ *MedicalRecord_Data* ∈ *MedicalRecord* ⇸ *STR*
∧ *MedicalRecord_IsValidated* ∈ *MedicalRecord* → **BOOL**
...

La spécialisation de la relation traduisant un attribut dépend de son unicité (unique ou non) et de sa nature (optionnel ou obligatoire). Le tableau 1.1 résume le type de la fonction associée à chaque combinaison des paramètres unicité et nature de l'attribut.

	Optionnel	**Obligatoire**
Unique	Injection partielle	Injection totale
Non unique	Fonction partielle	Fonction totale

TAB. 1.1 – Relations fonctionnelles issues des attributs de classes

L'attribut *data* est non unique et optionnel, c'est pourquoi il est traduit par la fonction partielle *MedicalRecord_Data*. Son type *String* n'est ni une classe ni un type supporté par B. Par conséquent, il est traduit par l'ensemble abstrait STR. L'attribut *isValidated* est traduit par la fonction totale *MedicalRecord_IsValidated* car il est obligatoire et non unique.

1.2.4 Prise en compte des attributs dans les constructeurs et destructeurs

L'attribut *isValidated* admet une valeur par défaut évaluée à FALSE. Pour cela, nous introduisons la substitution suivante dans le constructeur de la classe *MedicalRecord* :

MedicalRecord_NEW(*Instance*, *patientValue*)=
PRE
/*préconditions de l'opération*/
...
THEN
...
/*Valeur par défaut de l'attribut *isValidated/**
|| MedicalRecord_IsValidated (*Instance*) := FALSE
END;

Les relations relatives aux attributs sont également mises à jour lors de la suppression de l'instance d'une classe du modèle. Dans le cas de la classe *MedicalRecord*, les substitutions suivantes sont introduites dans le destructeur *MedicalRecord_Free* :

MedicalRecord_Free(*Instance*)=
PRE
/*préconditions du destructeur*/
...
THEN
...
/*mise à jour des attributs*/
|| *MedicalRecord_Data* := {*Instance*} ⩤ *MedicalRecord_Data*
|| *MedicalRecord_IsValidated* := {*Instance*} ⩤ *MedicalRecord__IsValidated*
END;

1.2.5 Traduction des setter et getter d'attribut

En général, un getter et un setter sont générés pour chaque attribut de classe. A titre d'exemple, l'attribut SSN de la classe *Patient* est en lecture seule et ne possède donc pas de setter. Son getter est généré comme suit :

result ← **Patient__GetSSN**(*Instance*)=
PRE
Instance ∈ *Patient*
THEN
result := *Patient__SSN*(*Instance*)
END ;

Les setters sont traduits par des opérations mettant à jour la valeur retournée par la relation fonctionnelle correspondante. A titre d'exemple le setter de l'attribut *data* de la classe *MedicalRecord* est généré comme suit :

MedicalRecord_SetMedicalRecordData(*Instance*, *data*) =
PRE
Instance ∈ *MedicalRecord* ∧ *data* ∈ *STR*
THEN
MedicalRecord_Data := ({*Instance*} ⩤ *MedicalRecord_Data*) ∪ {*Instance* ↦ *data*}
END ;

1.2.6 Traduction des associations

Les associations sont traduites par des relations entre les instances effectives des classes. La spécialisation de ces relations dépend surtout des cardinalités de chaque extrémité.

A titre d'exemple la règle de traduction de l'association entre la classe *Patient* et *MedicalRecord* est donnée par le tableau 1.2.

ClasseA	ClasseB	Nom de la relation	Nature de la relation
Patient(1..1)	MedicalRecord(0..1)	patientMedicalRecord	Bijection partielle

TAB. 1.2 – Traduction des associations de l'exemple

En B, l'association entre la classe *Patient* et *MedicalRecord* est traduite comme suit :

ABSTRACT_VARIABLES
patientMedicalRecord ,
...
INVARIANT
patientMedicalRecord ∈ *Patient* ⤖ *MedicalRecord*
...

Un invariant supplémentaire garantissant qu'une instance de *MedicalRecord* ne peut pas être unique sera ajouté par la suite.

1.2.7 Génération de setters et getters de liens

Un setter et un getter sont en général, générés pour chaque lien. Si le lien est marqué par une contrainte *ReadOnly* alors seulement un getter est généré. Comme illustration, le getter de l'association *PatientMedicalRecord* est généré comme suit :

result ← **Patient_GetPatientDepartment**(*Instance*)=
PRE
Instance ∈ *Patient*
THEN
result := *patientDepartment*(*Instance*)
END ;

1.2.8 Prise en compte des associations dans les constructeurs et destructeurs de classes

Un lien entre deux classes est établi, en général, dans le constructeur de la classe de départ et détruit dans le destructeur de cette dernière. A titre d'exemple, nous revenons au constructeur et destructeur de la classe *MedicalRecord* pour le compléter par les préconditions et substitutions relatives à l'association *patientMedicalRecord* :

/*Mise à jour du constructeur*/
MedicalRecord_NEW(*Instance*, *patientValue*)=
PRE
$Instance \in MEDICALRECORD \wedge \quad Instance \notin MedicalRecord$
$\wedge\ patientValue \in Patient$
/*Préconditions relatives aux associations*/
$patientValue \notin \mathbf{dom}(patientMedicalRecord)$
THEN
$MedicalRecord := MedicalRecord \cup \{Instance\}$
/*mise à jour des associations*/
$patientMedicalRecord := patientMedicalRecord \cup$
$\{(patientValue \mapsto Instance)\}$
END;

/*Mise à jour du destructeur*/
MedicalRecord_Free(*Instance*)=
PRE
/*préconditions*/
THEN
...
/*mise à jour des asscoiations*/
$patientMedicalRecord := patientMedicalRecord \rhd \{Instance\}$
END;

1.3 Formalisation en B du modèle de sécurité

1.3.1 Méta-modèle de sécurité

Conformément à l'approche interprétée, la formalisation du modèle de sécurité repose, en premier lieu, sur la traduction du méta-modèle de sécurité permettant de modéliser des politiques de contrôle d'accès. En s'inspirant de SecureUML [LBD02], nous avons proposé un méta-modèle de sécurité, adapté au besoin de la traduction vers B et permettant d'exprimer des politques de sécurité basées sur les rôles (figure 1.3).

Les relations *UserAssignment* et *PermissionAssignment* du modèle RBAC [FK] sont respectivement modélisées par les associations entre les méta-classes *User* et *Role* et entre les méta-classes *Role* et *Permission*. Un rôle peut être affecté à plusieurs utilisateurs et un utilisateur peut jouer plusieurs rôles. De même une permission peut être affectée à plusieurs rôles et chaque rôle peut avoir plusieurs permissions. Une permission est constituée de plusieurs actions (méta-classe *Action*) et concerne une seule entité du modèle fonctionnel (méta-classe *Class*). Deux types d'actions sont proposés par le méta-modèle de sécurité. Le type *EntityAction* qui permet

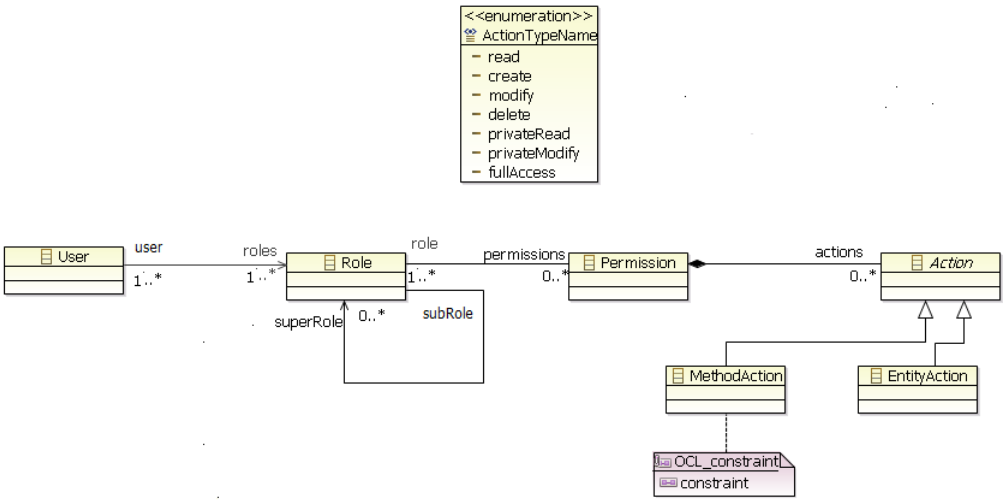


FIG. 1.3 – Méta-modèle de sécurité

de rassembler toutes les actions de base (*read*, *create*, *modify*, *delete*, *privateRead* et *privateModify*) relatives aux constructeurs, destructeurs, getter et setter de classes et le type *MethodAction* qui rassemble les opérations spécifiques du modèle fonctionnel(méta-classe *Operation*).

1.3.2 Affectation des utilisateurs aux rôles (*UserAssignment*)

La relation *UserAssignment* est traduite dans une machine B nommée "UserAssignments". Celle-ci contient la formalisation de la portion du méta-modèle (figure 1.4) mettant en jeu les méta-classes *Role* et *User*.

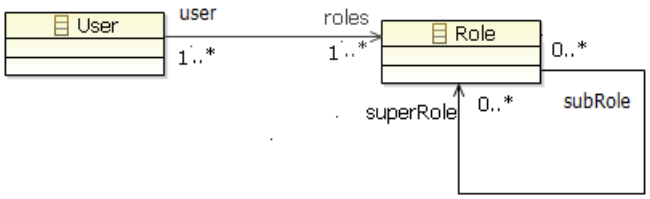


FIG. 1.4 – L'association UserAssignment

Le résultat de la formalisation de *UserAssignement* est donné par les spécifications B suivantes :

MACHINE
UserAssignments
SETS
ROLES ; *USERS*
VARIABLES
roleOf,
Roles_Hierarchy
INVARIANT
$roleOf \in USERS \rightarrow \mathcal{P}\ (ROLES) \wedge$
$Roles\_Hierarchy \in ROLES \leftrightarrow ROLES \wedge$
$\mathbf{closure1}(Roles\_Hierarchy) \cap \mathbf{id}(ROLES) = \emptyset$

Les méta-classes *Role* et *User* sont respectivement traduites par les ensembles abstraits ROLES et USERS. La fonction *roleOf* permet d'associer un ensemble de rôles à chaque utilisateur. La hiérarchie entre les rôles est traduite par la relation *Roles_Hierarchy*.

L'invariant closure1($Roles\_Hierarchy$) $\cap$ $\mathbf{id}(ROLES) = \emptyset$ indique que la hiérarchie de rôles ne doit pas contenir de cycle.

L'étape suivante de l'approche interprétée consiste à traduire une instance du méta-modèle de sécurité et l'injecter dans les spécifications B du méta-modèle. A titre d'exemple, prenons l'instance illustrée par la figure 1.5 correspondant à la portion du méta-modèle de la figure 1.4.

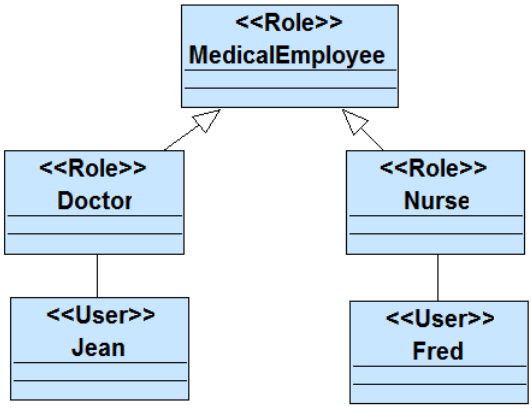


FIG. 1.5 – Instance de la figure 1.4

La formalisation des instances des méta-classes *Role* et *User* se traduit par la valuation des ensembles ROLES et USERS et par l'initilisation adéquate de la machine *UserAssignments* :

SETS
$ROLES = \{MedicalEmployee, Doctor, Nurse\}$;
$USERS = \{Fred, Jean\}$;
...
INITIALISATION
$Roles\_Hierarchy := \{(Doctor \mapsto MedicalEmployee), (Nurse \mapsto MedicalEmployee)\}$ ||
$roleOf := \{(Fred \mapsto \{Nurse\}), (Jean \mapsto \{Doctor\})\}$
...

1.3.3 Affectation des permissions aux rôles (*PermissionAssignment*)

La formalisation de la relation *PermissionAssignement* suit une logique semblable à celle de la relation *UserAssignment*. Elle commence par la traduction de la méta-classe *Permission* ainsi que tous les concepts du modèle fonctionnel associés aux permissions (Classe, attributs, opération, ...). Les spécifications B qui en résultent sont intégrées dans la machine de sécurité "RBAC_Model" :

MACHINE
RBAC_Model
INCLUDES
Functional_Model,
UserAssignments
SETS
$ENTITIES$;
$Attributes$;
$Operations$;
$KindsOfAtt = \{public, private\}$;
$PERMISSIONS$;
$ActionsType = \{read, create, modify, delete, privateRead, privateModify\}$;
$Stereotypes = \{readOp, modifyOp\}$
VARIABLES
$AttributeKind$, $AttributeOf$, $OperationOf$,
$constructorOf$, $destructorOf$, $setterOf$, $getterOf$, $EntityActions$,
$MethodActions$, $StereotypeOps$, $PermissionAssignement$,
$isPermitted$
INVARIANT
$AttributeKind \in Attributes \rightarrow KindsOfAtt \wedge$
$AttributeOf \in Attributes \rightarrow ENTITIES \wedge$
$OperationOf \in Operations \rightarrow ENTITIES \wedge$
$constructorOf \in Operations \rightarrowtail ENTITIES \wedge$
$destructorOf \in Operations \rightarrowtail ENTITIES \wedge$
$setterOf \in Operations \rightarrowtail Attributes \wedge$
$getterOf \in Operations \rightarrowtail Attributes \wedge$
$StereotypeOps \in Stereotypes \leftrightarrow Operations \wedge$
$setterOf \cap getterOf = \emptyset \wedge$
$PermissionAssignement \in PERMISSIONS \rightarrow (ROLES \times ENTITIES) \wedge$
$EntityActions \in PERMISSIONS \nrightarrow \mathcal{P}(ActionsType) \wedge$
$MethodActions \in PERMISSIONS \nrightarrow \mathcal{P}(Operations) \wedge$
$isPermitted : (ORG \times ROLES) \leftrightarrow Operations$

Les ensembles *ENTITIES*, *Attributes*, *Operations* et *KindsOfAtt* contiennent les éléments du modèle fonctionnel indispensables pour formaliser les permissions telles que définies dans le méta-modèle de sécurite. Les liens entre ces ensembles permettant de reconstruire la partie fonctionnelle sont réalisés grâce aux relations suivantes :

- *AttributeKind* : permet de retrouver le type de l'attribut (public ou privé).
- *AttributeOf* : permet de retrouver l'entité fonctionnelle encapsulant l'attribut.
- *OperationOf* : permet de retrouver l'entité fonctionnelle encapsulant l'opération.
- *constructorOf* et *destructorOf* : servent de classificateurs d'opérations (respectivement les constructeurs et desructeurs de chaque entité fonctionnelle).
- *setterOf* et *getterOf* : permettent aussi de classifier les setters et getters et les rattacher à leurs attributs.

Outre les setters et les getters qui permettent d'exercer respectivement des opérations de lecture et de modification, les spécifications B offrent la possibilité à l'analyste de stéréotyper les opérations de manière spécifique en indiquant s'il s'agit d'opérations de lecture ou de modification. Ceci est réalisé grâce à l'ensemble énuméré *Stereotypes* et la relation *StereotypeOps*.

Les autres éléments de la spécification représentent les concepts relatifs à la partie sécurité nécessaire à l'expression de permissions. Chaque permission de l'ensemble PERMISSIONS est associée à un couple (*role*, *entity*) où *role* ∈ ROLES et *entity* ∈ ENTITIES. Ceci est réalisé grâce à la relation *PermissionAssignment*. La relation *EntityActions* permet d'identifier les permissions contenant des actions globales (lecture, écriture, modification, ...). Quant à la relation *MethodActions*, elle permet d'indiquer les permissions contenant des opérations spécifiques.
La classification des opérations selon le type d'action (OperationOf, construtorOf, destructorOf, ...) est nécessaire pour le calcul des permissions en fonction des rôles associés à l'utilisateur courant. A titre d'exemple, une permission associée à un rôle R et une entité E, et contenant une action de type EntityAction *Create*, indique que si l'utilisateur courant est affecté au rôle R alors il a le droit d'exécuter l'opération issue de $constructorOf^{-1}$(E).

1.4 Conclusion du livrable 3.2

Bien qu'elle permette d'exprimer des politiques de sécurité basées sur les rôles, la formalisation du modèle de sécurité présentée dans le livrable 3.2 manque de certains aspects intéressants comme le concept d'organisation, les sessions et la gestion des conflits entre les rôles. De plus, les liens entre le modèle fonctionnel et le modèle de sécurité ne sont pas traités et les modèles des SI couverts par la formalisation actuelle sont limités à des environnements sécuritaires et fonctionnels totalement dissociés (aucune entité partagée entre le modèle de sécurité et le modèle fonctionnel). Le but de présent livrable est donc de pallier à cela en abordant les améliorations apportées à cette formalisation.

Chapitre 2

Generation of Security Tests from SecureUML diagrams

This chapter explores the generation of security tests in the context of SecureUML models translated in B.

2.1 Principles of test generation

The basic idea is to exercise each permission rule of the security policy both in order to make it (a) grant a permission (positive test case) and (b) deny a permission (negative test case). Granting the permission shows that the rule does not block all activity and ensures some level of availability of the protected resource. Denying the permission shows that the rule has some effect in protecting against some interactions.

Similarity of positive and negative test cases Positive and negative test cases should be similar, and only differ by one or two instructions, so that the difference in behaviour can be easily attributed to the permission rule. For example, the same test can be played with the same user playing two different roles.

B animation The test activity is performed here at the PIM level, i.e. at an abstract level which allows to concentrate on the "logic" of the security policy. The goal is here to protect the information system against internal threats which exploit the logic of the security policy to access unauthorized resources. The SecureUML diagram is translated into B using the B4MSecure tool. We use the ProB animator in order to animate the specification.

Because we are working in the context of B/ProB, operations whose precondition is not verified will not be enabled in the animator. This means that the tests that deny a permission should prevent using a given operation.

2.2 A first example : Patient and Medical Record

Our first example is taken from the B4MSecure distribution.

2.2.1 Functional model

Its functional model (Fig. 2.1) features two classes : *Patient* and *MedicalRecord*, linked by a 1 to many association. Medical records have two attributes : *Data* which stores the contents

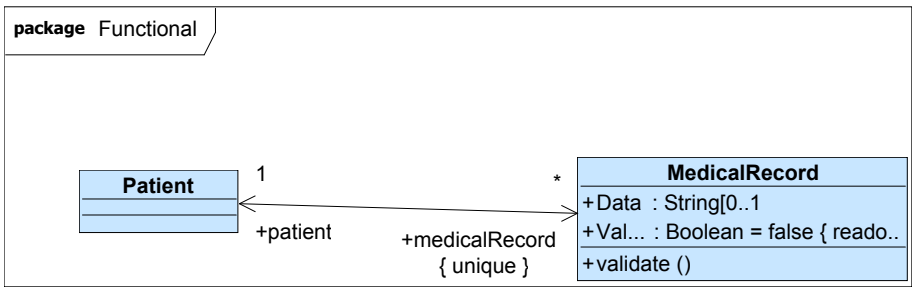


FIG. 2.1 – The functional model of the simple Medical Record example

of the medical record and $Valid$ which is true once the record has been validated by a doctor. $Valid$ is marked as "readonly", which means that the B4MSecure tool will not generate a setter for this attribute. Actually, modifications of this attribute may only occur through the *validate* operation, which is user-defined and is aimed to change the $false$ status of $Valid$ into *true*.

2.2.2 Roles

Five roles have been identified in this example (Fig. 2.2) : $PatientRole$, $Secretary$, $Nurse$, $Doctor$ and $MedicalStaff$. $Doctor$ and $Nurse$ inherit the permissions of $MedicalStaff$.

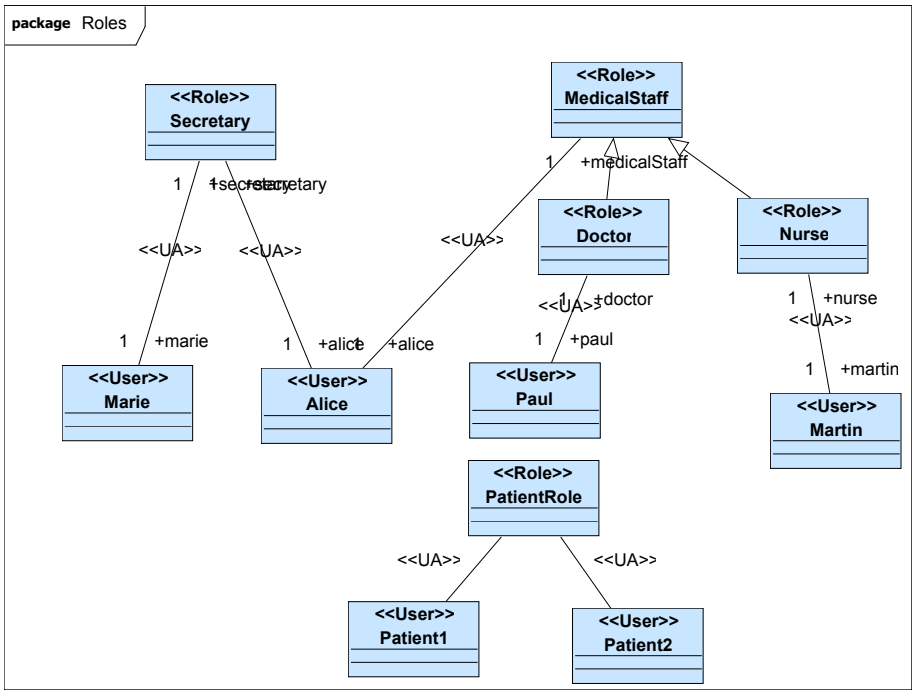


FIG. 2.2 – The roles of the simple Medical Record example

Six users have been defined at this point, and assigned various roles.

2.2.3 Security rules

Five permissions are expressed in the security models (Fig. 2.3 and Fig. 2.4).

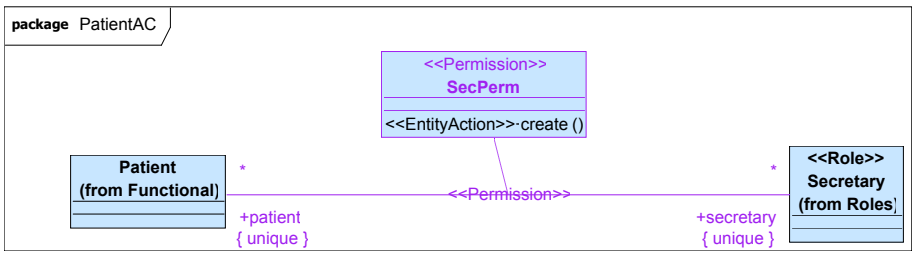


FIG. 2.3 – Permissions associated to patients data

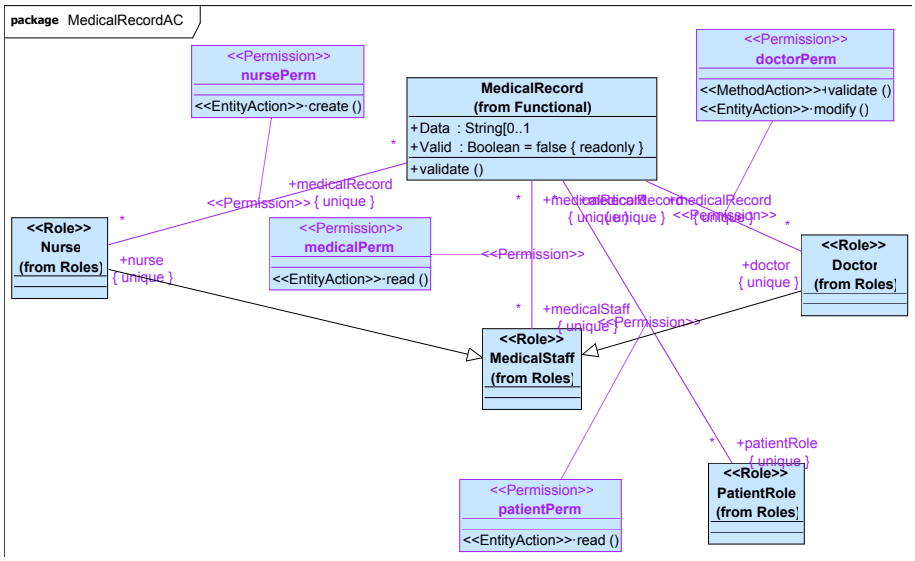


FIG. 2.4 – Permissions associated to medical records

2.3 Testing permissions

The permissions are tested, starting with the ones that must be exercised first in a normal scenario. Here, the patient should be created before the medical record, and the medical record should be created before being modified or read.

2.3.1 Testing SecPerm

SecPerm grants to role *Secretary* the permission to create a patient. It appears that it is the only permission allowing to create a patient.

Two tests should be produced. The first one (positive test) will succeed in applying the permission, and the second one will fail.

Positive test of SecPerm

The following test connects Alice with role *Secretary*, creates a session and exercises the permission by creating a patient.

```
setPermissions
Connect(Alice,{Secretary})
changeCurrentUser(Alice)
secure_Patient_NEW(Patient1)
```

Negative test of SecPerm

The goal of the negative test is to show that `secure_Patient_NEW(PATIENT1)` is not enabled at some point, due to the *SecPerm* permission. One must take care that other reasons, linked to functional preconditions, could disable the operation. For example, once created, the patient may not be created twice because a precondition prevents the creation of an existing patient.

In order to make sure that it is the security policy which disables the operation, we will start from the positive test, and make a small variation, which is clearly relevant to the security model only. Here, Alice performs the same operations, but using role $MedicalStaff$. The negative test is the following one.

```
setPermissions
Connect(Alice,{MedicalStaff})
changeCurrentUser(Alice)
secure_Patient_NEW(Patient1)
```

Actually, when we play these steps in the proB animator, the first three steps succeed, but the fourth one is not enabled, which shows the failure of the negative test and hence that the security policy is correct.

2.3.2 Testing nursePerm

nursePerm is used to create a medical record. It is the only way to access the create operation. There is a functional precondition associated to the create operation : the associated patient must have been created before.

Positive test of nursePerm

This positive test first connects a user (Marie) as *Secretary*, then creates a patient. Then it connects Martin as a nurse and creates the medical record.

```
setPermissions
Connect(Marie,{Secretary})
changeCurrentUser(Marie)
secure_Patient_NEW(Patient1)
Connect(Martin,{Nurse})
changeCurrentUser(Martin)
secure_MedicalRecord_NEW(MEDICALRECORD1,Patient1)
```

This test succeeds using proB.

Negative test of nursePerm

As mentioned before, it is necessary to perform a small modification of the previous scenario and check that the call to `secure_MedicalRecord_NEW(MEDICALRECORD1,Patient1,FALSE)` no longer
succeeds. The smallest modification would be to connect Martin in another role than $Nurse$. But the model does not provide another role for Martin. We may either modify the model to add this user assignment link, or even take benefit of the fact that $Nurse$ is a subrole of $MedicalStaff$

and thus log Martin as $MedicalStaff$ [1]. We chose to try to connect Paul as a doctor and then perform the operation.

```
setPermissions
Connect(Marie,{Secretary})
changeCurrentUser(Marie)
secure_Patient_NEW(Patient1)
Connect(Paul,{Doctor})
changeCurrentUser(Paul)
secure_MedicalRecord_NEW(MEDICALRECORD1,Patient1)
```

As expected, the tool allows to perform all steps but not the last one. Notice that we could replace the connection of Paul, by the connection of Alice as $MedicalStaff$, or the connection of Mary as *Secretary*. These variants would lead to the same result.

2.3.3 Testing doctorPerm

doctorPerm grants the permission to modify the patient record, or to validate a medical record. A functional precondition is that a medical record is available and that it has not been validated yet.

Positive tests of doctorPerm

If we trust the implementation of permissions, only one of the two following tests suffices to exercise the condition.

In the first test, we create a patient and a medical record. Then Paul, acting as a *Doctor* will validate the record.

```
setPermissions
Connect(Marie,{Secretary})
changeCurrentUser(Marie)
secure_Patient_NEW(Patient1)
Connect(Martin,{Nurse})
changeCurrentUser(Martin)
secure_MedicalRecord_NEW(MEDICALRECORD1,Patient1)
Connect(Paul,{Doctor})
changeCurrentUser(Paul)
secure_MedicalRecord__validate(MEDICALRECORD1)
```

A second test performs the same actions but ends with a call to the setter of field *Data*.

```
setPermissions
Connect(Marie,{Secretary})
changeCurrentUser(Marie)
secure_Patient_NEW(Patient1)
Connect(Martin,{Nurse})
changeCurrentUser(Martin)
secure_MedicalRecord_NEW(MEDICALRECORD1,Patient1)
```

[1] The current version of the B4MSecure tool does not allow to connect using a super-role unless there is an explicit user assignment link.

```
Connect(Paul,{Doctor})
changeCurrentUser(Paul)
secure_MedicalRecord__SetData(MEDICALRECORD1,STR1)
```

One could also group modification and validation in a single test case.

Negative tests of doctorPerm

These are the same tests than the positive ones, but the last user to connect should not be a doctor. We can try connecting as a secretary or as a medical staff.

```
setPermissions
Connect(Marie,{Secretary})
changeCurrentUser(Marie)
secure_Patient_NEW(Patient1)
Connect(Martin,{Nurse})
changeCurrentUser(Martin)
secure_MedicalRecord_NEW(MEDICALRECORD1,Patient1)
Connect(Alice,{MedicalStaff})
changeCurrentUser(Alice)
secure_MedicalRecord__validate(MEDICALRECORD1)
```

```
setPermissions
Connect(Marie,{Secretary})
changeCurrentUser(Marie)
secure_Patient_NEW(Patient1)
Connect(Martin,{Nurse})
changeCurrentUser(Martin)
secure_MedicalRecord_NEW(MEDICALRECORD1,Patient1)
Connect(Alice,{Secretary})
changeCurrentUser(Alice)
secure_MedicalRecord__SetData(MEDICALRECORD1,STR1)
```

2.3.4 Testing medicalPerm

medicalPerm grants the permission to read the medical record to all medical personal. This includes access to both fields *Data* and $Valid$.

Positive test of medicalPerm

Here a medical staff (Alice) tries to read the $Valid$ attribute of a medical record, just after Martin has created it.

```
setPermissions
Connect(Marie,{Secretary})
changeCurrentUser(Marie)
secure_Patient_NEW(Patient1)
Connect(Martin,{Nurse})
changeCurrentUser(Martin)
secure_MedicalRecord_NEW(MEDICALRECORD1,Patient1)
Connect(Alice,{MedicalStaff})
```

```
changeCurrentUser(Alice)
secure_MedicalRecord_GetValid(MEDICALRECORD1)-->FALSE
```

We could write a longer test were a doctor modifies the *Data* field, and then some other medical personal reads this field.

Negative test of medicalPerm

The negative test is a variant of the previous one where user Alice connects as *Secretary* instead of $MedicalStaff$.

```
setPermissions
Connect(Marie,{Secretary})
changeCurrentUser(Marie)
secure_Patient_NEW(Patient1)
Connect(Martin,{Nurse})
changeCurrentUser(Martin)
secure_MedicalRecord_NEW(MEDICALRECORD1,Patient1)
Connect(Alice,{Secretary})
changeCurrentUser(Alice)
secure_MedicalRecord_GetValid(MEDICALRECORD1)-->FALSE
```

2.3.5 Testing patientPerm

patientPerm grants users in the *PatientRole* the permission to read the patient record. It is associated to an authorization constraint which requires that the user be the same as the patient. In other words, a patient may only read his/her own record.

Positive test of patientPerm

In the positive test, Marie, acting as *Secretary*, creates two patients (actually only the first one is useful for the positive test). Then Martin, acting as a $Nurse$, creates a medical record for Patient1. Finally, Patient1 connects in the *PatientRole* and reads his medical record.

```
setPermissions ;
Connect(Marie,{Secretary}) ;
changeCurrentUser(Marie) ;
secure_Patient_NEW(Patient1) ;
secure_Patient_NEW(Patient2) ;
Connect(Martin,{Nurse}) ;
changeCurrentUser(Martin) ;
secure_MedicalRecord_NEW(MEDICALRECORD1,Patient1) ;
Connect(Patient1,{PatientRole}) ;
changeCurrentUser(Patient1) ;
secure_MedicalRecord__GetValid(MEDICALRECORD1)-->FALSE
```

Negative tests of patientPerm

Two negative tests are designed, as a variant of the positive test. The first negative test checks that the read access is denied to some user using the role *Secretary*, which is the only role that does not have read access to medical records.

```
setPermissions ;
Connect(Marie,{Secretary}) ;
changeCurrentUser(Marie) ;
secure_Patient_NEW(Patient1) ;
secure_Patient_NEW(Patient2) ;
Connect(Martin,{Nurse}) ;
changeCurrentUser(Martin) ;
secure_MedicalRecord_NEW(MEDICALRECORD1,Patient1) ;
Connect(Alice,{Secretary}) ;
changeCurrentUser(Alice) ;
secure_MedicalRecord__GetValid(MEDICALRECORD1)-->FALSE
```

The second negative test checks the authorisation constraint. Here a second patient, Patient2, tries to access the medical record of Patient1.

```
setPermissions ;
Connect(Marie,{Secretary}) ;
changeCurrentUser(Marie) ;
secure_Patient_NEW(Patient1) ;
secure_Patient_NEW(Patient2) ;
Connect(Martin,{Nurse}) ;
changeCurrentUser(Martin) ;
secure_MedicalRecord_NEW(MEDICALRECORD1,Patient1) ;
Connect(Patient2,{PatientRole}) ;
changeCurrentUser(Patient2) ;
secure_MedicalRecord__GetValid(MEDICALRECORD1)-->FALSE
```

2.4 First conclusions

The permissions used in this example have some simplified characteristics :
– each operation (except Read) is associated to one permission. There is thus only one way to perform the secured operation.
– There is no organisation associated to the security policy.
In these first tests, we have mostly focused on producing a single positive test and a single negative one. Several evolutions may be considered :
– We could systematically disconnect a user (not the case now).
– We could rework the user assignments so that the same user may take several roles and that depending on this role only we have positive and negative tests (as was done in *SecPerm*).
– There may exist shorter tests where the same user, maybe using several roles, performs actions in sequence. But such tests may be more confusing, and more difficult to tranform into negative tests. For example, a nurse may create a medical record and then read it, but the negative test will require to change the user before trying to read.
– Several positive tests are the prefix of another. The most elaborated tests actually exercise several permissions to reach a state where the next permission will be reachable.
– One may be interested in producing several tests, which may take advantage of a combinatorial generator such as Tobias.

2.5 Structuring tests in dedicated B machines

The above tests have been successfully experimented using ProB. The animator displays the tests which will succeed and hides the other ones.

We have already pointed out that positive and negative tests should be similar, in order to make clear which statements lead to the success or failure of the test. The following table gives an example of such similarities, corresponding to the test of *nursePerm*. It suggests that tests should be structured in three parts : preamble, several nominal or robustness choices, and the operation call exercising the permission. The preamble and the operation call are the same for both positive and negative tests. The nominal and robustness steps should be compared to identify which elements (user, role,...) are involved in the success or failure of the test. Note that the call itself might be expressed in two forms : nominal and robustness.

	Positive test (nominal)	Negative test (robustness)

preamble	`setPermissions`
	`Connect(Marie,Secretary)`
	`changeCurrentUser(Marie)`
	`secure_Patient_NEW(Patient1)`	

variants	`Connect(Martin,{Nurse})` `changeCurrentUser(Martin)`	`Connect(Paul,{Doctor})` `changeCurrentUser(Paul)`
main call	`secure_MedicalRecord_NEW(MEDICALRECORD1,Patient1)`	

We have adopted this structure in the B machines. A first machine `TestSteps_Abstract` defines four abstract steps corresponding to the preamble, the nominal and robustness parts, and the operation call. This machine is generic and can be reused for the test of several permissions.

```
MACHINE
    TestSteps_Abstract
OPERATIONS
  preamble = BEGIN skip END
  ;
  nominal = BEGIN skip END
  ;
  robustness = BEGIN skip END
  ;
  call = BEGIN skip END
END
```

A second machine refines this abstract machine and provides the definition of each test step for a given permission. For example, the test steps for *nursePerm* are defined as follows :

```
REFINEMENT
   TestSteps_nursePerm_Concrete
REFINES
   TestSteps_Abstract
INCLUDES
    RBAC_Model, ContextMachine
```

```
OPERATIONS
  preamble =
    BEGIN
      setPermissions ;
      Connect(Marie,{Secretary}) ;
      changeCurrentUser(Marie) ;
      secure_Patient_NEW(Patient1)
    END
    ;
  nominal =
    BEGIN
      Connect(Martin,{Nurse}) ;
      changeCurrentUser(Martin)
    END
    ;
  robustness =
    BEGIN
      Connect(Paul,{Doctor}) ;
      changeCurrentUser(Paul)
    END
    ;
  call =
    BEGIN
      secure_MedicalRecord_NEW(MEDICALRECORD1,Patient1)
    END
END
```

This presentation of the test cases allows to easily compare the nominal and robustness steps for a given permission.

The remaining machines will sequence these test steps into a positive test case and a negative one. First these tests are introduced in an abstract machine, which is also reusable to test other permissions.

```
MACHINE
    Tests_Abstract
OPERATIONS
  TestPos = BEGIN skip END /* should succeed */
  ;
  TestNeg = BEGIN skip END /* should fail */
  ;
  preambleAndRobustness = BEGIN skip END /* should succeed */
END
```

Please take note that a third operation, `preambleAndRobustness` is also provided. It corresponds to a prefix of the negative test, which should succeed. This will show that the negative test fails while playing its last step, i.e. the operation call.

Finally, we introduce machine `Tests_nursePerm_Concrete` which combines the test steps in the three test cases. It is the machine that should be loaded in ProB.

```
REFINEMENT
   Tests_nursePerm_Concrete
```

```
REFINES
   Tests_Abstract
INCLUDES
    TestSteps_nursePerm_Concrete
OPERATIONS
  TestPos =
    BEGIN
      preamble ;
      nominal ;
      call
    END
  ;
  TestNeg =
    BEGIN
      preamble ;
      robustness ;
      call
    END
  ;
  preambleAndRobustness =
    BEGIN
      preamble ;
      robustness
    END
END
```

Fig. 2.5 shows a screen copy of the ProB animator. After loading `Tests_nursePerm_Concrete` and performing the initialisation step, only two operations are enabled (*TestPos* and *preambleAndRobustness*), which show that the positive test succeeds and that the negative one will only fail because of the last call.

This structuration of the test suites in B machine has been performed successfully for each of the five permissions. When the number of positive or negative tests increases, new versions of the abstract machines must be written. In the common case where only one positive and one negative tests are needed, the abstract machines can be reused, and the analyst only needs to write the two concrete machines.

2.6 Final conclusion

We have shown how a security policy, expressed in SecureUML and translated in B using B4MSecure, can be animated using positive and negative tests. The systematic definition of positive and negative tests for each permission ensures that these permissions are able to grant or deny access to the protected resource. This also applies to permissions associated to an authorization constraint, and could be extended to permissions associated to organisations.

We have advocated that positive and negative tests corresponding to the same rule should be similar. This gives additional confidence that the failure of the negative test is actually the result of the rule under test and is not influenced by other rules or by elements of the functional specification.

In order to produce similar positive and negative test cases, more effort might be needed in the definition of the user assignments. For example, this would enable the same user to try the same operation call using two different roles. In the case study discussed in this report, we relied on

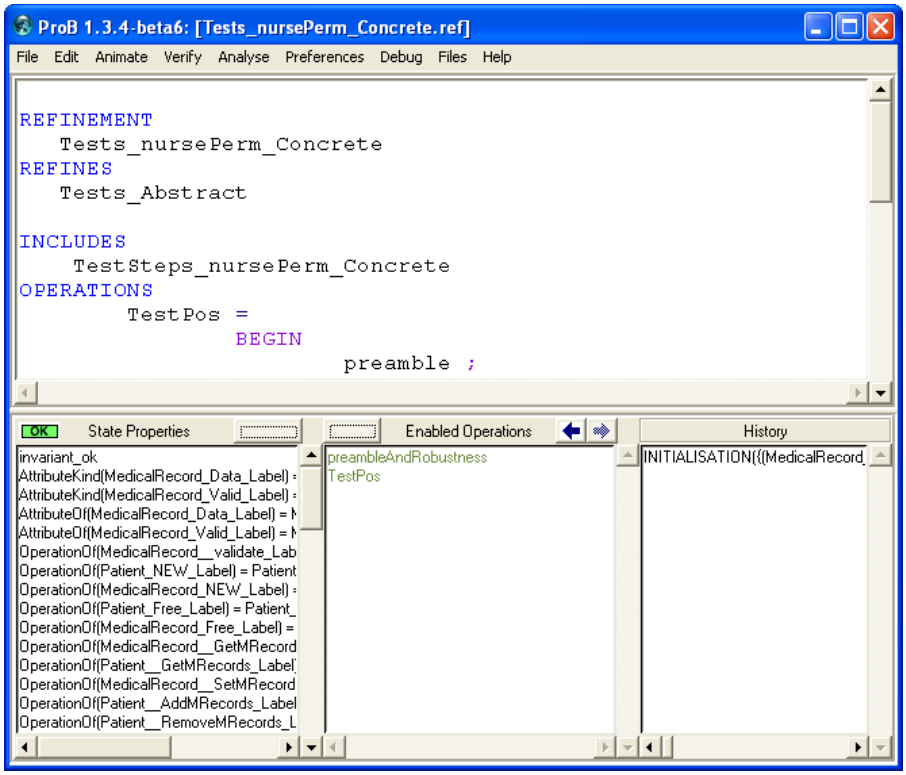


FIG. 2.5 – The ProB animator running Tests_nursePerm_Concrete

existing user assignments, which should ideally be reworked.

We have also proposed a structure of B machine which identifies common parts in the positive and negative tests (i.e. preamble and operation call) and also identifies the differences (nominal and robustness steps). We believe that this structure favours the production of similar test cases and allows to easily compare the nominal and robustness parts.

Further work may evaluate the benefit of producing several positive and negative test cases for each permission rule. A larger number of test cases could allow a more exhaustive coverage of the role hierarchy for a given rule. It could benefit from a combinatorial test generator such as Tobias.

Chapitre 3

Formalisation en B de politiques de contrôle d'accès basées sur les organisations

3.1 Introduction

Le concept d'organisation peut être un élément principal dans un Système d'Information. A titre d'exemple, les SI hospitaliers ont une architecture basée sur les organisations (hôpitaux, départements, unités de soin, ...). De plus, les règles de gestion peuvent différer d'une organisation à une autre au sein d'un même SI. Par conséquent, chaque organisation peut avoir des politiques de sécurité spécifiques offrant à une fonction (un rôle) des privilèges qui ne sont pas accordés dans une autre organisation. Le méta-modèle de sécurité, du livrable 3.2 (section 1.3.1), ne permet pas d'exprimer des politiques de sécurité basées sur les organisations. De plus, les liens entre le modèle fonctionnel et le modèle de sécurité n'ont pas été traités auparavant ce qui rend impossible l'expression de contraintes de sécurité impliquant des éléments du modèle fonctionnel. En première partie de ce chapitre, nous proposons une extension du méta-modèle de sécurité qui couvre le concept d'organisation ainsi que sa formalisation en B. L'approche de formalisation sera illustrée par un exemple d'un SI hospitalier où la confidentialité et l'integrité des enregistrements médicaux sont deux enjeux majeurs. En deuxième partie, nous étudions les liens entre le modèle fonctionnel et le modèle de sécurité afin de permettre l'expression de contraintes de sécurité liées aux entités fonctionnelles.

3.2 Les organisations

3.2.1 Extension du méta-modèle de sécurité

La figure 3.1 montre une extension du méta-modèle de sécurité présenté dans la section 1.3.1. L'extension consiste en l'ajout de la méta-classe *Organization* ainsi que ses associations avec les méta-classes *Role*, *Permission* et *Assignment*.

Selon le méta-modèle, une permission est associée à un rôle et un ensemble d'organisations. Dans chaque organisation, plusieurs rôles peuvent être joués. Grâce à l'héritage multiple entre les organisations, le méta-modèle offre à l'analyste la possibilité d'accorder des permissions qui peuvent être héritées et d'autres qui sont spécifiques à certaines organisations. L'analyste peut aussi ajouter des contraintes OCL sur certaines permissions. L'ajout de la méta-classe *Organization* introduit également, une nouvelle dimension dans l'affectation des utilisateurs. Un utilisateur

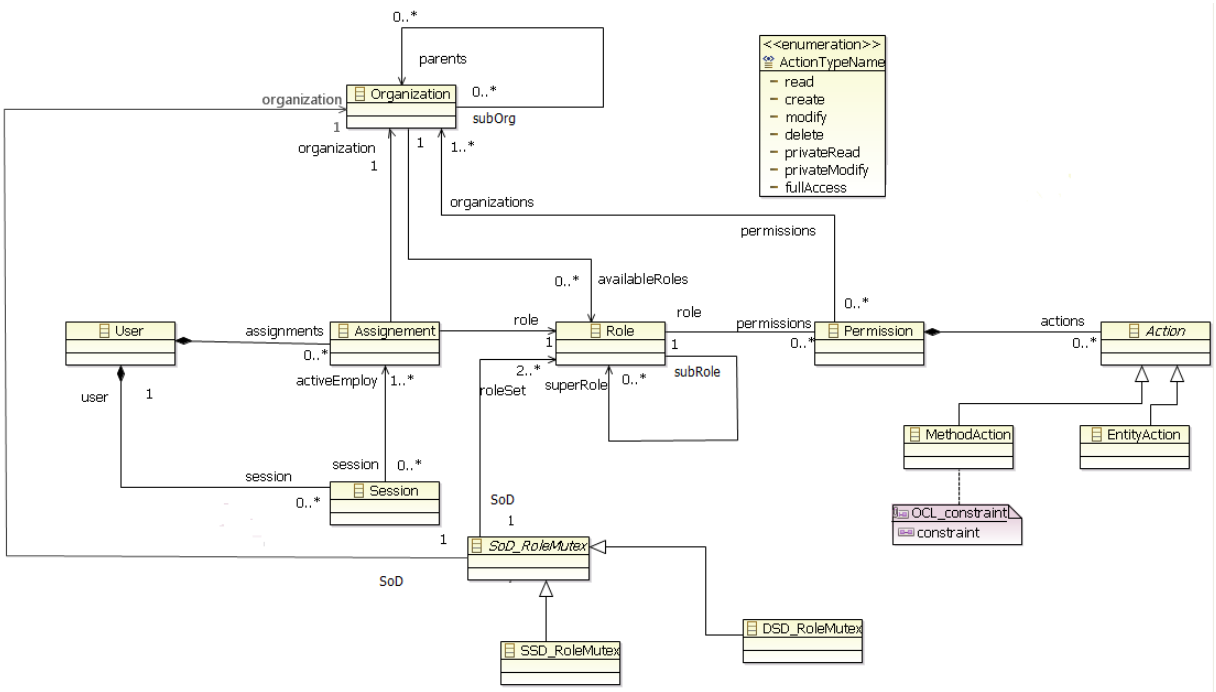


FIG. 3.1 – Extension du métamodèle de sécurité

est désormais affecté à un rôle et une organisation. Cette affectation est représentée par la méta-classe *Assignment*. Le méta-modèle offre également la possibilité de définir des conflits entre les rôles selon l'organisation. Ceci se traduit par l'association entre la méta-classe *Organization* et la méta-classe *SoD_RoleMutex*.

Dans la suite nous proposons, un exemple illustratif d'un modèle fonctionnel protégé par un modèle de contrôle d'accès conforme à ce méta-modèle.

3.2.2 Exemple illustratif

La figure 3.2 montre un modèle fonctionnel d'un SI hospitalier augmenté par des politiques de contrôle d'accès .

Le modèle fonctionnel de l'exemple (classes en bleu) est composé de :
– La classe *Patient* qui représente tous les patients de l'hôpital.
– La classe *MedicalRecord* qui représente les enregistrements médicaux des patients.
– La classe *MedicalEmployee* qui représente les membres du personnel médical (docteurs et infirmiers).
– La classe *Department* qui représente les départements de l'hôpital.
Chaque patient est hospitalisé dans un département et lui est associé un enregistrement médical. Les membres du personnel médical peuvent travailler dans plusieurs départements et se chargent de créer, modifier ou valider les enregistrements médicaux des patients. Un enregistrement médical ne peut être modifié après avoir été validé (fermé).

Le modèle de sécurité (classes en jaune) est composé des éléments suivants :
– Les rôles *MedicalEmployee*, *Doctor*, *Nurse* et *DepartmentDirector*.
– Les organisations *Hospital*, *Radiology* et *Cardiology*.
– Les permissions *ReadMedicalRecord*, *NurseMedicalRecordPerm* et *DoctorMedicalRecordPerm*

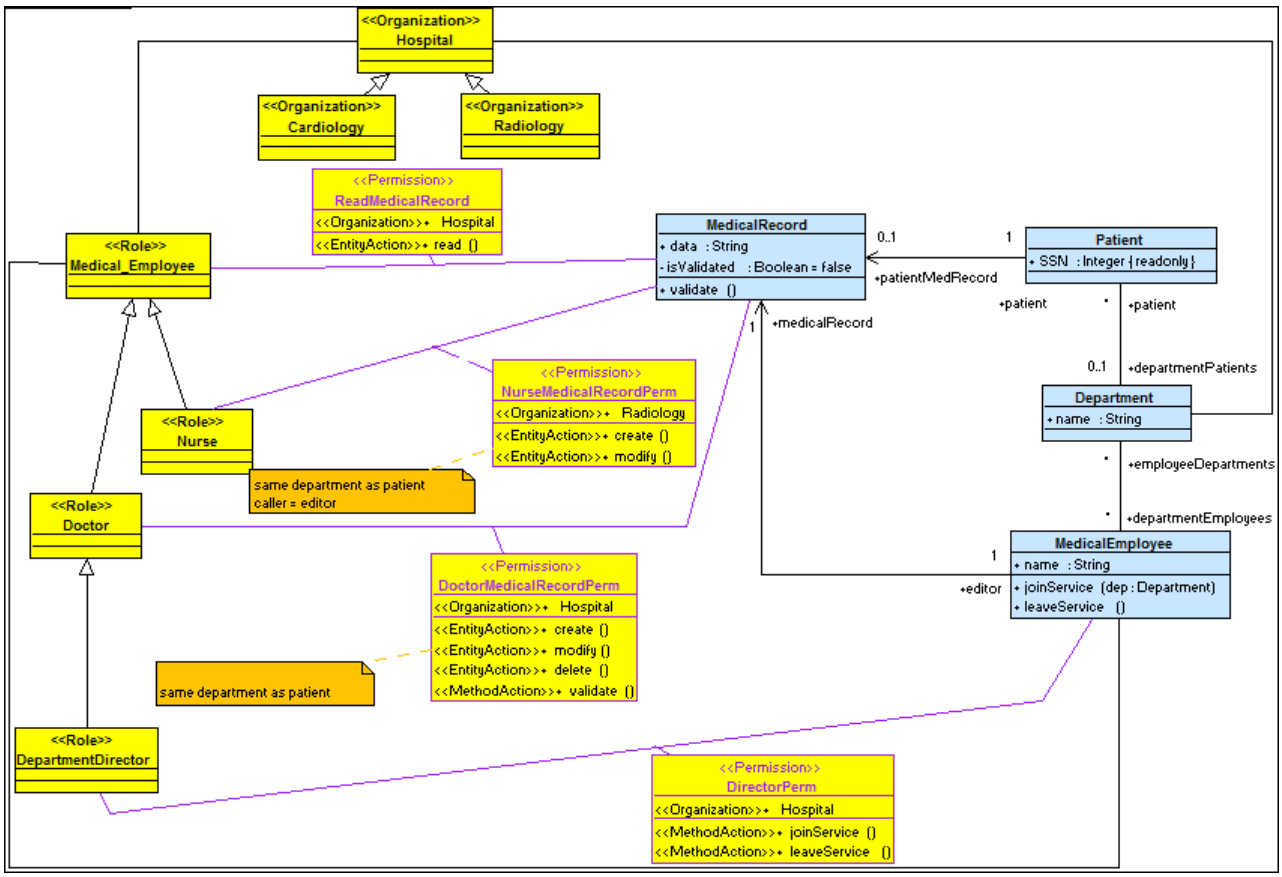


FIG. 3.2 – Modèle d'un SI hospitalier

définies sur l'entité fonctionnelle *MedicalRecord* et la permission *DirectorPerm* définie sur l'entité *MedicalEmployee*.

Les politiques de sécurité sont définies de la façon suivante :

- Chaque employé médical (rôle *MedicalEmployee*) a le droit de lire les enregistrements médicaux.
- Les docteurs peuvent créer, modifier, supprimer et valider les enregistrements médicaux des patients à condition qu'ils soient dans le même département que le patient.
- Seuls les infirmiers (rôle *Nurse*) du département *Radiology* peuvent créer les enregistrements médicaux des patients du même département et les modifier s'ils sont ses créateurs.
- Un chef de service (rôle *DepartmentDirector*) peut associer des employés à son département comme il peut en dissocier d'autres.

3.2.3 Traduction de la méta-classe *Organization*

Nous traduisons la méta-classe *Organization* par l'ensemble abstrait ORG. La hiérarchie entre les organisations est traduite par la relation suivante :

$$Org\_Hierarchy \in ORG \leftrightarrow ORG$$

Chaque lien d'héritage entre deux organisations fait partie de la relation *Org_Hierarchy* dont le domaine représente l'ensemble des organisations descendantes et le codomaine représente celui des organisations parentes. Afin d'éviter les cycles dans la relation *Org_Hierarchy*, nous introduisons l'invariant suivant :

$$\mathbf{closure1}(Org\_Hierarchy) \cap \mathbf{id}(ORG) = \emptyset$$

Comme exemple d'instance de la méta-classe *Organization*, nous reprenons, par la figure 3.3, les classes *Hospital*, *Radiology* et *Cardiology* figurant dans l'exemple illustratif 3.2.2.

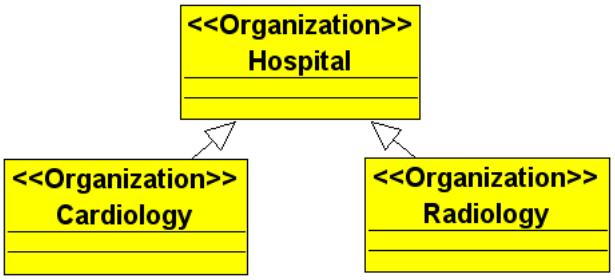


FIG. 3.3 – Instance de la méta-classe *Organization*

La formalisation de ces instances introduit les valeurs *Hospital*, *Radiology* et *Cardiology* dans l'ensemble ORG et les relations d'héritage correspondantes dans l'initialisation de l'ensemble *Org_Hierarchy* :

SETS
$ORG = \{Hospital, Radiology, Cardiology\}$;
...
INITIALISATION
...
$Org\_Hierarchy := \{(Cardiology \mapsto Hospital), (Radiology \mapsto Hospital)\}$
...

3.2.4 Traduction de l'association entre les méta-classes *Role* et *Organization*

L'association entre la méta-classe *Role* et *Organization* (figure 3.4) permet à l'analyste de définir le jeu de rôles existant dans chaque organisation du modèle.

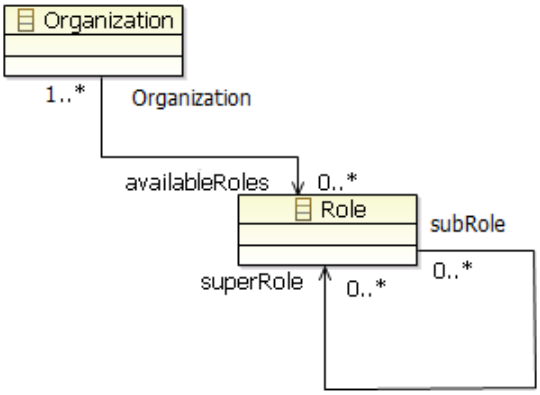


FIG. 3.4 – Association entre les méta-classes *Organization* et *Role*

Cette association est traduite en B par la fonction partielle suivante :

$$roleOfOrg \in ORG \nrightarrow \mathcal{P}\ (ROLES)$$

Rappelons que l'ensemble ROLES est la traduction de la méta-classe *Role* suivant l'ancienne formalisation (décrite dans la section 1.3). Nous avons choisi de garder cet ensemble ainsi que la relation *Roles_Hierarchy* traduisant l'héritage entre les rôles. Suivant la figure 3.5 issue de notre exemple illustratif, l'ensemble ROLES, la relation *Roles_Hierarchy* et la relation *roleOfOrg* sont valués comme suit :

SETS
$ROLES = \{Medical\_Employee, Doctor, Nurse, DepartmentDirector\}$
...
INITIALISATION
$Roles\_Hierarchy := \{(Doctor \mapsto Medical\_Employee),$
$(Nurse \mapsto Medical\_Employee),$
$(DepartmentDirector \mapsto Doctor)$
$\} \parallel$
$roleOfOrg := \{(Hospital \mapsto \{Medical\_Employee\})\}$
...

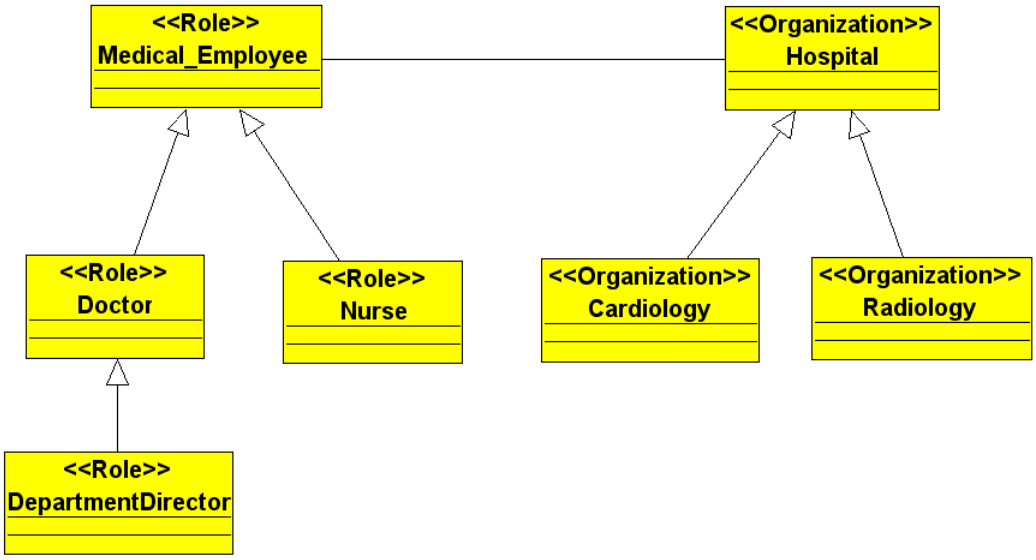


FIG. 3.5 – Instance de l'association entre les méta-classes *Organization* et *Role*

Les rôles associés aux organisations parentes sont systématiquement hérités par les organisations descendantes. Ce lien d'héritage offre à l'analyste la possibilité de généraliser et/ou spécialiser la relation d'affectation des rôles aux organisations. Afin de retrouver les rôles associés à chaque organisation du système, nous introduisons les définitions suivantes :

/*Trasnformation de la fonction *roleOfOrg* en une relation*/

$orgRole == \{org, ro \mid org \in ORG \wedge org \in \mathbf{dom}(roleOfOrg) \wedge ro \in roleOfOrg(org)\}$;

/*Définition permettant de déduire la hiérarchie de rôles associée à chaque organisation du domaine de roleOfOrg*/

$orgRoles == (orgRole; \mathbf{closure1}(Roles\_Hierarchy^{-1})) \cup orgRole$;

/*Définition permettant de retrouver les rôles associés à toutes les organisations du système*/

$allOrgRoles == (\mathbf{closure1}(Org\_Hierarchy); orgRoles) \cup orgRoles$;

Le résultat de ces définitions sur la base de l'exemple de la figure 3.5 est donné comme suit :

$orgRole == \{(Hospital \mapsto Medical\_Employee)\}$;

$orgRoles == \{(Hospital \mapsto Medical\_Employee), (Hospital \mapsto Doctor),$
$(Hospital \mapsto Nurse), (Hospital \mapsto DepartmentDirector)\}$;

$allOrgRoles == \{(Hospital \mapsto Medical\_Employee), (Hospital \mapsto Doctor),$
$(Hospital \mapsto Nurse), (Hospital \mapsto DepartmentDirector),$
$(Cardiology \mapsto Medical\_Employee), (Cardiology \mapsto Doctor),$
$(Cardiology \mapsto Nurse), (Cardiology \mapsto DepartmentDirector),$
$(Radiology \mapsto Medical\_Employee), (Radiology \mapsto Doctor),$
$(Radiology \mapsto Nurse), (Radiology \mapsto DepartmentDirector)\}$

3.2.5 Affectation des utilisateurs aux rôles et aux organisations

Selon notre méta-modèle, les rôles sont affectés aux utilisateurs dans le contexte d'une organisation. Ceci introduit une nouvelle dimension dans la relation *user_assign* du modèle RBAC. En effet, *user_assign* devient un triplet (user, role, org) qui signifie que l'utilisateur *user* peut jouer le rôle *role* au sein de l'organisation *org*. Nous traduisons ce triplet en B par la relation suivante :

$$user\_assign \in USERS \leftrightarrow (ORG \times ROLES)$$

A titre d'exemple, nous considérons l'affectation donnée par la figure 3.6. L'instanciation de l'ensemble USER ainsi que la relation *user_assign* correspondant à cette figure sont introduites dans la formalisation B comme suit :

$user\_assign := \{(Martin \mapsto (Cardiology, DepartmentDirector)),$
$(Fred \mapsto (Radiology, Doctor)),$
$(Paul \mapsto (Cardiology, Nurse))\}$

Notons que les couples (org, role) assignés à un utilisateur doivent être issus de la relation *roleOfOrg* qui permet de déterminer tous les couples (org, role) du modèle. Afin de vérifier cette condition, nous introduisons l'invariant suivant :

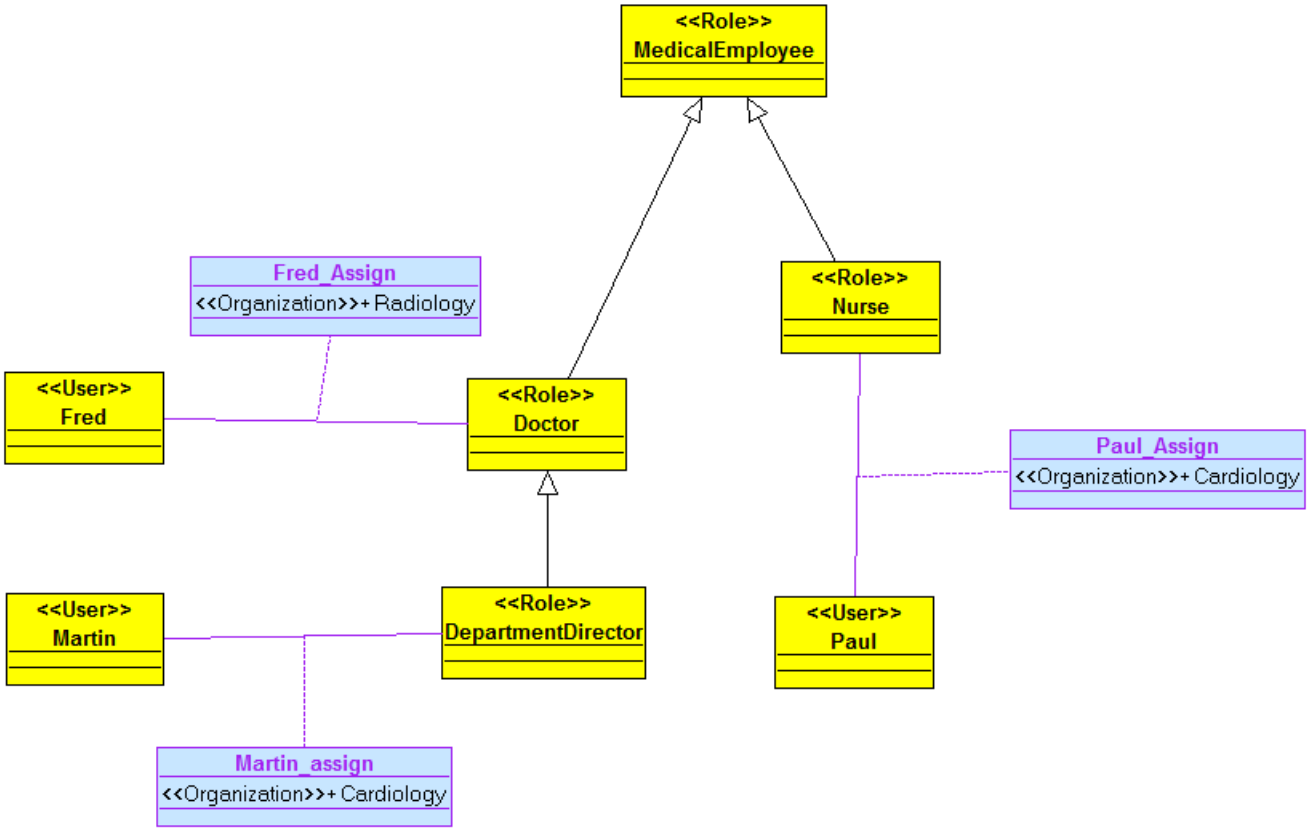


FIG. 3.6 – Instanciation de user_assign

INVARIANT

$$\ldots$$
$$\forall\, (uu,\ org,\ ro).(uu \in USERS \wedge uu \in \mathbf{dom}(user\_assign) \wedge org \in ORG \wedge ro \in ROLES \wedge (org,\ ro) \in user\_assign[\{uu\}] \Rightarrow (org,\ ro) \in allOrgRoles)$$

Le prédicat permet de vérifier que tout couple (org, ro) assigné à un utilisateur existe dans le modèle et ce en parcourant l'ensemble renvoyé par la définition *allOrgRoles* qui permet de retrouver tous les couples (org, ro) possibles.

3.2.6 La séparation des droits dans le contexte d'une organisation

Rappelons que RBAC définit une SoD (Separation of Duty) par une fonction affectant un entier n tel que $n \geq 2$ à un ensemble de rôles *rs*. L'entier affecté spécifie le nombre de rôles engendrant un conflit dans l'ensemble *rs*. Par conséquent, la cardinalité de l'ensemble *rs* doit être supérieure à l'entier affecté. Dans notre méta-modèle 3.1, une SoD est définie par rapport à une organisation ce qui permet à l'analyste de spécifier des conflits dans une organisation particulière. Ceci rajoute une nouvelle dimension dans le domaine de la fonction SoD. Ainsi, nous traduisons en B, la relation SoD (statique dans ce cas) de la façon suivante :

INVARIANT

...

/*Définition de la fonction SSD*/

$SSD\_mutex : (\ \mathcal{P}_1\ (ROLES) \times ORG) \nrightarrow \mathbf{NAT1}$

/*Prédicat imposant que le nombre de conflits (nn) est $\geq$ 2 */

$\wedge\ \forall\ nn.(nn \in \mathbf{NAT1} \wedge nn \in \mathbf{ran}(SSD\_mutex) \Rightarrow nn \geq 2)$

/*Prédicat imposant que card(rs) $\geq$ nn*/

$\wedge\ \forall\ (rs,org).(rs \in \mathcal{P}_1\ (ROLES) \wedge org \in ORG$
$\wedge\ (rs,org) \in \mathbf{dom}(SSD\_mutex) \Rightarrow \mathbf{card}(rs) \geq SSD\_mutex((rs,org)))$

Notons que nous utilisons la même définition pour la variable *DSD_mutex* correspondant à la séparation dynamique des droits. La différence entre *SSD_mutex* et *DSD_mutex* est définie au niveau des variables du noyau de sécurité. En effet, *SSD_mutex* permet de vérifier les conflits dans la relation *user_assign* tant dis que *DSD_mutex* permet de vérifier des conflits sur l'ensemble de rôles actifs dans une session. Notre formalisation de SSD et DSD prend en compte l'héritage des rôles et des organisations. En effet, si un rôle *r1* est en conflit avec un rôle *r2* alors tous les sous-rôles de *r1* et *r2* sont aussi en conflit. De plus, si dans une organisation *org*, certains rôles sont en conflit alors ils le sont dans toutes les organisations héritant de *org*. La formalisation de *SSD_mutex* et *DSD_mutex* est décrite dans les spécifications complètes du modèle de sécurité en Annexe B. Nous prenons à titre d'exemple, la figure 3.7 qui illustre une séparation statique entre les rôles *Nurse* et *Doctor*.

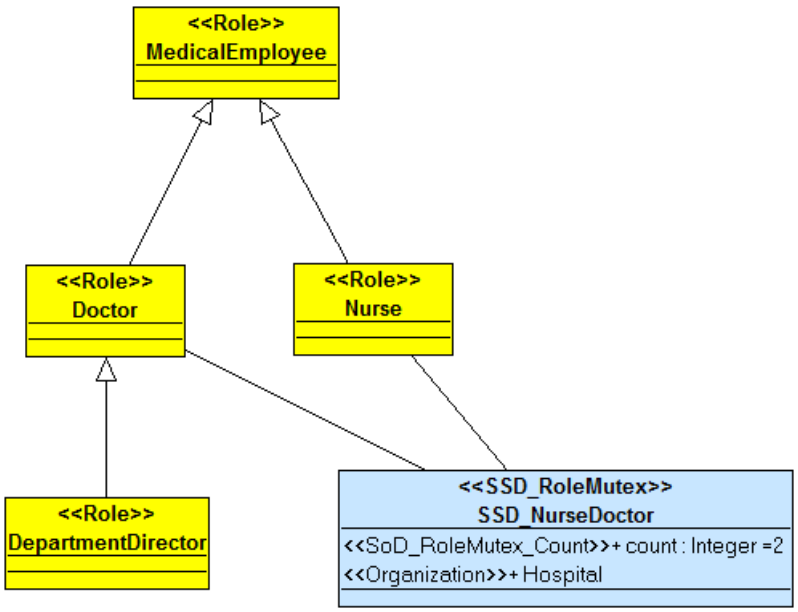


FIG. 3.7 – Spération statique entre les rôles Nurse et Doctor

La classe *SSD_NurseDoctor* indique qu'un utilisateur ne peut pas être affecté à la fois au rôle *Nurse* et au rôle *Doctor*. L'attribut *count*, de valeur égale à 2 dans cet exemple, indique que tous les rôles de la relation SSD sont deux à deux incompatibles.

L'initialisation de *SSD_mutex* associée à la figure 3.7 est donnée comme suit :

INITIALISATION
...
SSD_NurseDoctor := {(({Doctor, Nurse}, *Hospital)* ↦ *2*)}

3.2.7 Affectation des permissions aux rôles et aux organisations

La formalisation des concepts liés aux permissions (entités du modèle fonctionnel, types d'action, ...), évoquée dans la section 1.3.3 reste valable dans le modèle de contrôle d'accès basé sur les organisations. La différence réside dans la relation d'affectation des permissions (figure 3.8) à laquelle une nouvelle dimension s'ajoute.

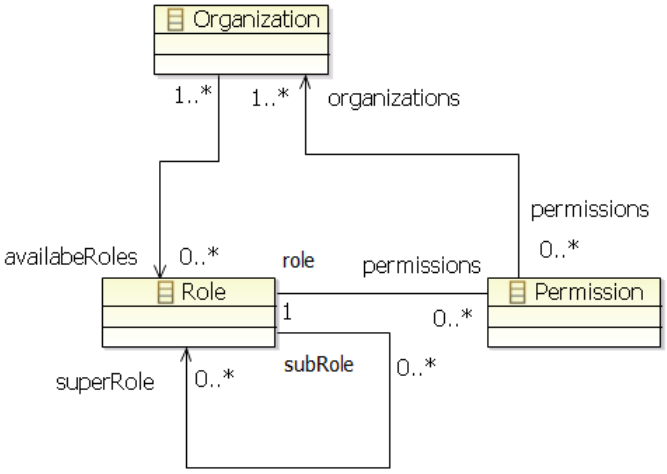


FIG. 3.8 – Affectation des permissions aux rôles et aux organisations

En effet, les permissions sont assignées non plus aux rôles seulement mais aussi à un ensemble d'organisations. Cette affectation est traduite par la relation *PermissionAssignment* suivante :

$$PermissionAssignment \in PERMISSIONS \rightarrow ((\ \mathcal{P}_1\ (ORG) \times ROLES) \times ENTITIES)$$

Conformément au méta-modèle, *PermissionAssignment* est une fonction totale permettant d'affecter chaque permission (PERMISSIONS) à un ensemble d'organisations $\mathcal{P}_1$ (*ORG*), un rôle (*ROLES*) et une entité du modèle fonctionnel (*ENTITIES*).

Notons que la relation *PermissionAssignment*, comme la relation *user_assign*, doit respecter l'instance du modèle graphique associée à la relation *roleOfOrg*. Pour ce faire, nous introduisons l'invariant suivant qui permet de vérifier que tous les couples (org, role) de la relation *PermissionAssignement* font bien partie de l'ensemble des couples (org, role) du modèle (*allOrgRoles*) :

INVARIANT
...
$\forall\ (pp, org, ro).(pp \in PERMISSIONS \wedge pp \in \mathbf{dom}(PermissionAssignement) \wedge org \in ORG \wedge$
$org \in \mathbf{union}(\mathbf{dom}(\mathbf{dom}(PermissionAssignement[\{pp\}]))) \wedge ro \in ROLES \wedge$
$ro \in \mathbf{ran}(\mathbf{dom}(PermissionAssignement[\{pp\}]))$
$\Rightarrow (org, ro) \in allOrgRoles)$

A titre d'exemple, nous illustrons par la figure 3.9, deux cas d'affectation de permissions.

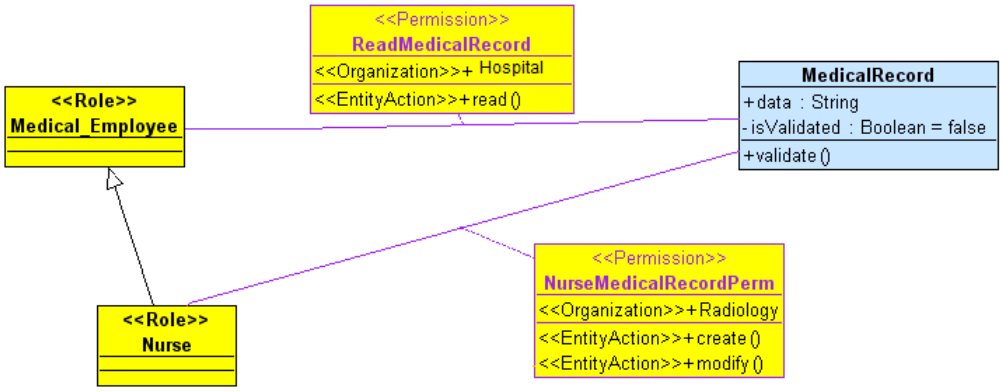


FIG. 3.9 – Spécialisation de permissions

– Un cas général (*ReadMedicalRecordPerm*) : Chaque employé médical (infirmier et docteur) peut lire les enregistrements médicaux des patients dans tout département de l'hôpital.
– Un cas spécifique (*NurseMedicalRecordPerm*) : seulement les infimiers du département radiologie peuvent créer et modifier les enregistrements médicaux de leurs patients.

La formalisation de ces instances est introduite dans l'initialisation de la relation *PermissionAssignement* comme suit :

INITIALISATION

$$
\begin{array}{l}
\ldots \\
PermissionAssignement := \{ \\
\quad (ReadMedicalRecord \mapsto (((\{Hospital\}, Medical\_Employee) \mapsto medicalRecord)), \\
\quad (NurseMedicalRecordPerm \mapsto (((\{Radiology\}, Nurse) \mapsto medicalRecord))\}
\end{array}
$$

Calcul des permissions pour les rôles et les organisations

Rappelons que le contrôle d'accès à une opération du modèle fonctionnel s'effectue à travers une opération sécurisée de la machine B "Security_Model". Le contrôle consiste à vérifier que l'utilisateur courant dispose d'une permission l'autorisant à appeler l'opération fonctionnelle et ce en parcourant l'ensemble *isPermitted* des permissions affectées aux organisations et aux rôles du système.

A titre d'exemple, l'appel à l'opération de modification d'un enregistrement médical *SetMedicalRecordData* passe par l'opération sécurisée *secure_SetMedicalRecordData* suivante :

secure_SetMedicalRecordData(*Instance*, *data*) =
PRE
/* Préconditions de base */
...
THEN
/* Vérification des droits d'accès */
SELECT
setMedicalRecordData ∈ *isPermitted*[*currentOrgRole_Session*]
THEN
/* Appel à l'opération fonctionnelle */
SetMedicalRecordData(*Instance*, *data*)
END
END;

Si l'utilisateur courant est affecté à une organisation et un rôle lui pemettant d'exercer le droit de modification sur l'entité *MedicalRecord* alors il pourra exécuter l'opération *SetMedicalRecordData* du modèle fonctionnel et faire par conséquent, évoluer l'état du système.

L'objectif est de calculer l'ensemble *isPermitted* qui sert à retrouver toutes les opérations permises pour un couple (*org*, *role*) donné. Ceci est réalisé à travers des définitions en B et de l'opération de calcul de permissions suivante :

setPermissions = **PRE***isPermitted* = ∅ **THEN***isPermitted* := *allPermissions* **END**;

Les définitions B permettent de calculer progressivement, à travers la relation *PermissionAssignement* et les classificateurs d'opérations (*operationOf*, *constructorOf*, *destructorOf*, *setterOf* et *getterOf*) , tous les triplets ((*org*, *role*), *op*) possibles où *op* est une opération permise pour un couple (*org*, *role*) du système et ce en respectant l'ordre suivant :

- Calcul des triplets ((*org*, *role*), *op*) par type d'opération (constructeur, destructeur, setter, ...) où (*org*, *role*) est issu directement de la relation *PermissionAssignement* sans prise en compte de l'héritage entre les rôles et les organisations. A titre d'exemple, une permission associée à un rôle *role*, à un ensemble d'organisations *orgSet* et à une entité *entity*, et contenant une action de type "EntityAction" Create, donne le droit à tout utilisateur affecté au couple (*org*, *role*) où *org* ∈ *orgSet*, d'exécuter les opérations issues de $constructorOf^{-1}(entity)$.
- Calcul de l'ensemble union de tous les triplets ((*org*, *role*), *op*) issus de la première étape via la définition *orgPermission*.
- Déduction des triplets ((*org*, *role*), *op*) issus de la relation d'héritage entre les organisations via la relation *allOrgPermissions* suivante :

$$
\begin{aligned}
&allOrgPermissions == \{org, ro, op \mid org \in ORG \wedge org \in \mathbf{dom}(\mathbf{dom}(orgPermissions)) \cup \\
&subOrg(\mathbf{dom}(\mathbf{dom}(orgPermissions))) \\
&\wedge ro \in ROLES \wedge ro \in \mathbf{ran}((\{org\} \cup \\
&superOrg(\{org\})) \lhd \mathbf{dom}(orgPermissions)) \\
&\wedge op : (\mathbf{ran}(\{pOrg, role \mid pOrg \in ORG \\
&\wedge role \in ROLES \wedge pOrg \in superOrg(\{org\}) \wedge role = ro\} \lhd orgPermissions) \cup \\
&\quad \mathbf{ran}(\{(org \mapsto ro)\} \lhd orgPermissions)) \}\,;
\end{aligned}
$$

- Déduction de tous les triplets ((*org*, *role*), *op*) en prenant en compte l'héritage des permissions via les rôles à travers la définition *allPermissions* suivante :

$$
\begin{array}{l}
allPermissions == \{org,\ ro,\ op \mid org \in ORG \land ro \in ROLES \land op \in Operations \\
\land\ op \in \mathbf{ran}(allOrgPermissions) \land ro : (\mathbf{ran}(\{org\} \lhd \mathbf{dom}(allOrgPermissions \rhd \{op\})) \cup \\
subRoles(\mathbf{ran}(\{org\} \lhd \mathbf{dom}(allOrgPermissions \rhd \{op\}))))\} \\
\land\ org \in \mathbf{dom}(\mathbf{dom}(allOrgPermissions \rhd \{op\}))
\end{array}
$$

Afin d'illustrer cette procédure de calcul, nous reprenons l'exemple de la figure 3.9. Initialement, nous avons :

$$
\begin{array}{l}
\textbf{/* Classification des opérations */} \\
\\
\quad constructorOf := \{(medicalRecord\_NEW \mapsto medicalRecord)\} \quad || \\
\quad setterOf := \{(medicalRecord\_SetMedicalRecordData \mapsto medicalRecord\_Data)\} \\
\quad || \\
\quad getterOf := \{(medicalRecord\_GetMedicalRecordData \mapsto medicalRecord\_Data)\} \\
\textbf{/* Affectation des permissions aux rôles et aux organisations */} \\
\\
\quad PermissionAssignement := \{ \\
\quad\quad (ReadMedicalRecord \mapsto ((\{Hospital\},\ Medical\_Employee) \mapsto medicalRecord)), \\
\quad\quad (NurseMedicalRecordPerm \mapsto ((\{Radiology\},\ Nurse) \mapsto medicalRecord))\} \\
\textbf{/* Types d'action associés aux permissions */} \\
\\
\quad EntityActions := \{(ReadMedicalRecord \mapsto \{read,\ privateRead\}), \\
\quad\quad (NurseMedicalRecordPerm \mapsto \{create,\ modify,\ privateModify\})\}
\end{array}
$$

Le résultat retourné par les définitions est le suivant :

/* Les triplets ((*org*, *role*), *op*) issus directement de *PermissionAssignment/**

orgPermissions == {((Radiology, Nurse),medicalRecord_NEW),
((Radiology, Nurse), medicalRecord_SetMedicalRecordData),
((Hospital, Medical_Employee), medicalRecord_GetMedicalRecordData)}

/* Prise en compte des permissions issues de l'héritage des organisations */

allOrgPermissions == {((Hospital, Medical_Employee), medicalRecord_GetMedicalRecordData),
((Cardiology, Medical_Employee), medicalRecord_GetMedicalRecordData),
((Radiology, Medical_Employee), medicalRecord_GetMedicalRecordData),
((Radiology, Nurse),medicalRecord_NEW),
((Radiology, Nurse), medicalRecord_SetMedicalRecordData)}

/* Prise en compte des permissions issues de l'héritage des rôles */

allPermissions == {((Hospital, Medical_Employee), medicalRecord_GetMedicalRecordData),
((Cardiology, Medical_Employee), medicalRecord_GetMedicalRecordData),
((Radiology, Medical_Employee), medicalRecord_GetMedicalRecordData),
((Hospital, Nurse), medicalRecord_GetMedicalRecordData),
((Cardiology, Nurse), medicalRecord_GetMedicalRecordData),
((Radiology, Nurse), medicalRecord_GetMedicalRecordData),
((Radiology, Nurse),medicalRecord_NEW),
((Radiology, Nurse), medicalRecord_SetMedicalRecordData)}

3.3 Etude de liens entre le modèle fonctionnel et le modèle de sécurité

La séparation entre la spécification du modèle fonctionnel et la spécification du modèle de sécurité rentre dans l'intérêt de séparer les préoccupations. Ceci permet la vérification et la validation des besoins fonctionnels et des besoins de sécurité séparément et faciliter ainsi l'identification du type d'erreurs trouvées (d'origine fonctionnelle ou sécuritaire). Le découplage entre le modèle fonctionnel et le modèle de sécurité offre aussi l'avantage de réutilisation. En effet, le même modèle fonctionnel peut être protégé par des politiques de sécurité différentes. Et inversement, le même noyau de sécurité peut être utilisé pour plusieurs modèles fonctionnels.
En contre partie, notre étude de diverses exemples de modèles graphiques a dégagé l'existence, parfois, de liens forts entre le modèle fonctionnel et le modèle de sécurité. A titre d'exemple, le modèle graphique du SI hospitalier présenté dans la section 3.2.2 (figure 3.2) montre l'existence des liens suivants :
– Un lien entre l'entité fonctionnelle *Department* et les organisations *Radiology* et *Cardiology*.
– Un lien entre l'entité fonctionnelle *MedicalEmployee* et tout utilisateur possédant le rôle *Medical_Employee*.
L'importance de ces liens est plus explicite dans les contraintes OCL attachées aux permissions. Par exemple, la contrainte "*Same department as patient*" signifie que l'utilisateur appelant l'opération doit travailler dans le même département que celui du patient. Pour vérifier ce type de contraintes, le système doit parcourir le modèle fonctionnel et identifier le département du patient (en tant qu'entité fonctionnelle) pour ensuite le comparer avec le département de l'utilisateur courant.
Le deuxième point dégagé par l'étude de liens entre le modèle fonctionnel et le modèle de sécurité, est que l'évolution du modèle fonctionnel peut parfois avoir un impact sur la politique de sécurité. A titre d'exemple, si un chef de service (rôle *DepartmentDirector*) associe un docteur à son département, il met ainsi à jour le lien fonctionnel entre les classes *Department* et *MedicalEmployee*. Le docteur qui est une entité du modèle fonctionnel est en même temps un utilisateur du système et le fait d'être associé à une nouvelle organisation lui accorde éventuellement de nouveaux privilèges d'où la nécessité de mettre à jour le modèle de sécurité.
En se basant sur l'exemple 3.2.2 comme illustration, nous proposons dans la suite, une formalisation de liens entre le modèle fonctionnel et le modèle de sécurité permettant de prendre en compte les contraintes liées aux permissions et l'impact de l'évolution du modèle fonctionnel sur le modèle de sécurité.

3.3.1 Formalisation des entités partagées entre le modèle fonctionnel et le modèle de sécurité

En général, les entités susceptibles d'être partagées entre le modèle fonctionnel et le modèle de sécurité sont les organisations et les utilisateurs affectés à des rôles ayant un lien sémantique avec certaines entités fonctionnelles (exemple, le rôle *Medical_Employee* et l'entité fonctionnelle *MedicalEmployee*). Afin d'expliciter le lien entre ces entités, nous proposons de définir les ensembles des organisations ORG et des utilisateurs USERS dans une machine B "Context" qui est accessible par la machine du modèle fonctionnel et la machine du modèle de sécurité :

MACHINE
Context
SETS
USERS ;
ORG
END

Les entités fonctionnelles concernées par le partage sont incluses dans les ensembles correspondant dans la machine "Context". Dans le cas de notre exemple, nous avons les entités *MedicalEMployee* et *Department* qui sont incluses respectivement dans l'ensemble USERS et l'ensemble ORG :

MACHINE
Functional_Model
SEES
Context
SETS
/*Les entités non partagées*/
$PATIENT$;
$MEDICALRECORD$;
ABSTRACT_CONSTANTS /*Les entités partagées*/
$MEDICALEMPLOYEE$, $DEPARTMENT$
PROPERTIES
/*prédicat d'inclusion faisant le lien entre les modèles*/
$MEDICALEMPLOYEE \subseteq USERS \wedge$
$DEPARTMENT \subseteq ORG$
...

Notons, que dans cette solution, nous considérons que l'ensemble des instances possibles de l'entité partagée, est inclus dans l'ensemble adéquat (ORG ou USERS) de la machine "Context" et fait donc partie des éventuels utilisateurs ou organisations du système.

3.3.2 Exemple de formalisation de contraintes liées aux permissions

Mis à part le contrôle d'accès qui consiste à vérifier si l'utilisateur courant possède la permission nécessaire pour exercer l'opération demandée, il peut exister d'autres contraintes spécifiques sur certaines permissions faisant impliquer des éléments du modèle fonctionnel. Nous reprenons à titre d'exemple, la contrainte "same department as patient" liée aux permissions *DoctorMedicalRecordPerm* et *NurseMedicalRecordPerm* (exemple illustratif 3.2.2). Cette contrainte impose qu'un infirmier ou un docteur soit dans le même département que le patient pour pouvoir exercer les opérations permises. La prise en compte de ce type de contraintes est réalisée en ajoutant le prédicat adéquat dans le filtre de toutes les opérations concernées.
Dans le cas de l'opération *secure_MedicalRecord_SetMedicalRecordData*, cette contrainte est traduite par le prédicat suivant :

$$patientDepartment(patientMedicalRecord^{-1}(Instance)) = currentOrg$$

Remarquons que le prédicat parcourt les liens fonctionnels (*patientDepartment*et *patientMedicalRecord*) afin d'identifier l'organisation du patient et la compare ensuite à la variable *currentOrg*(qui désigne l'organisation de l'utilisateur courant) pour vérifier que l'organisation de l'utilisateur courant correspond bien au département du patient propriétaire de l'enregistrement médical traité.
Les autres contraintes de l'exemple illustratif sont formalisées en suivant la même logique. Les détails de cette formalisation figurent dans les spécifications complètes du modèle de sécurité en annexe B.

3.3.3 Impact de l'évolution du modèle fonctionnel sur la politique de sécurité

Afin d'illustrer le problème de l'impact de l'évolution du modèle fonctionnel sur la politique de sécurité, nous reprenons avec la figure 3.10, extraite de l'exemple illustratif 3.2.2, la permission *DirectorPerm*.

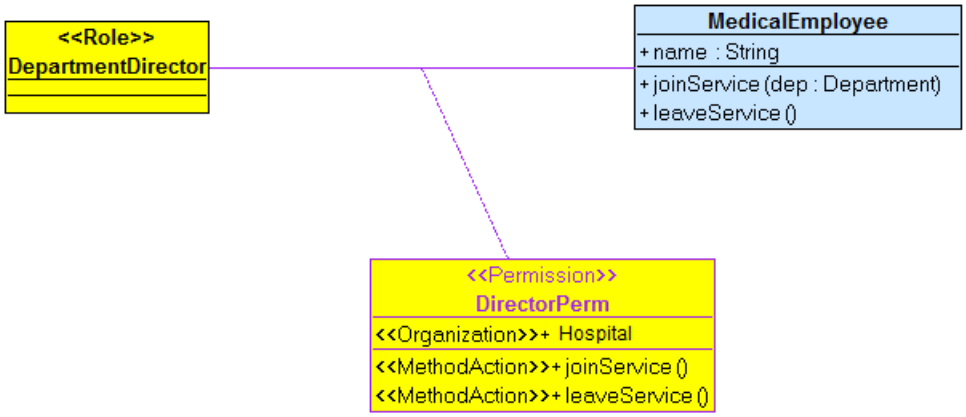


FIG. 3.10 – La permission *DirectorPerm*

La permission *DirectorPerm* accorde à un chef de service le droit d'associer un docteur à son département en lui donnant l'accès à l'opération *joinDepartment*. Cette dernière permet d'associer une instance de l'entité fonctionnelle *MedicalEmployee* à une instance de l'entité *Department*. Cette affectation dans le modèle fonctionnel implique une évolution dans le modèle de sécurité qui consiste à affecter un utilisateur à une nouvelle organisation, lui donnant ainsi d'éventuels nouveaux droits d'accès. Afin de prendre en compte cette mise à jour dans la politique de sécurité, nous introduisons l'opération *add_userAssign* suivante :

add_userAssign$(user, role, org) =$
PRE
$user \in USERS \wedge role \in ROLES \wedge org \in ORG \wedge$
$(org \mapsto role) \in allOrgRoles$
THEN
$user\_assign := user\_assign \cup \{(user \mapsto (org \mapsto role))\}$
END ;

L'opération permet d'ajouter une affectation dans la relation *user_assign* qui traduit l'assignement des utilisateurs aux rôles et aux organisations (section 3.2.5). L'appel à cette opération s'effectue dans le corps de l'opération sécuritaire chargée de filtrer l'accès à l'opération fonctionnelle correpondante.

A titre d'exemple, nous citons l'opération sécurisée *secure_MedicalEmployee_joinDepartment* qui permet au chef de service d'associer un employé médical à son département :

secure_MedicalEmployee_joinDepartment(*Instance*,*dep*)=
PRE
/*Préconditions de contrôle de d'accès*/
dep = *currentOrg* ...
THEN
/*Accès à l'opération fonctionnelle*/
MedicalEmployee_joinDepartment(*Instance*, *dep*) ||
/*Mise à jour de la politique de sécurité*/
add_userAssign(*Instance*, *Medical_Employee*, *currentOrg*)
END
END;

3.4 Conclusion

Durant ce chapitre, nous avons proposé une démarche illustrée pour la formalisation en B de politiques de sécurité basées sur le concept de l'organisation. La formalisation a permis, en premier lieu, l'évolution du noyau de sécurité formalisé dans le cadre du livrable 3.2 et ce en intégrant les nouveaux aspects suivants :

– Le concept d'organisation.
– Les droits accordés par héritage entre les rôles et les organisations.
– La séparation des droits (SoD) au sein d'une organisation.

En deuxième lieu, une étude de liens entre le modèle fonctionnel et le modèle de sécurité a permis d'une part, de renforcer la politique de contrôle d'accès par l'expression de contraintes spécifiques impliquant des éléments partagés entre les deux modèles. D'autre part, l'étude a permis de dégager une nouvelle problématique qui est l'impact de l'evolution du modèle fonctionnel sur la politique de sécurité.

Chapitre 4

Formalisation du concept de session et validation de politiques de contrôle d'accès basé sur les organisations

4.1 Introduction

Dans les méthodes classiques de développement, la validation s'effectue en général après la phase d'implémentation. Ceci offre l'avantage de faire des tests de validation directement sur un modèle exécutable et d'observer des résultats concrets. Par contre, la correction peut parfois être coûteuse en cas de non conformité des résultats de tests de validation avec les attentes et les exigences du client. Des techniques de validation basées sur des modèles formels et semi-formels ont été proposées comme alternative pour les méthodes classiques. Le principe est de rendre les modèles exécutables aux niveaux les plus abstraits du système en question. Les outils USE [GBR07] et SecureMova [BCDE07] permettent seulement de savoir si une opération peut être exécutée à partir d'un état modélisé par un diagramme UML. Leur limite est qu'ils ne permettent pas de faire évoluer l'état courant du modèle ni d'exécuter une séquence d'opérations. Pour les méthodes formelles, comme Z ou B, il existe des outils permettant d'animer les spécifications et jouer des scénarios d'utilisation du système.
Dans ce chapitre, nous complétons, en premier lieu, la formalisation des politiques de sécurité par l'introduction du concept de session. Les sessions traduisent l'aspect dynamique du modèle de sécurité et seront fortement utiles pour l'animation du modèle.

En deuxième lieu, nous utilisons l'animateur ProB [LB08] dans notre approche de validation de politiques de sécurité basées sur les organisations. Pour ce faire, nous reprenons les modèles B issus de l'exemple illustratif 3.2.2 et nous proposons une analyse de deux types de scénarios : des scénarios révélant un comportement normal d'utilisation du SI et des scénarios révélant un comportement malicieux dits scénarios d'attaque.

4.2 Formalisation des sessions

Le concept de session est modélisé par le noyau RBAC comme étant un mapping entre un utilisateur et un sous-ensemble de rôles parmi ceux qui lui sont affectés via la relation *userAssignment*. Dans le méta-modèle de sécurité que nous proposons dans la section 3.2.1, la définition du concept de session prend en compte la dimension "Organisation". En effet, une session de-

vient un mapping entre un utilisateur et un couple (organisation, rôle). L'intégration du concept de session dans le méta-modèle de sécurité est traduite par la portion illustrée par la figure 4.1.

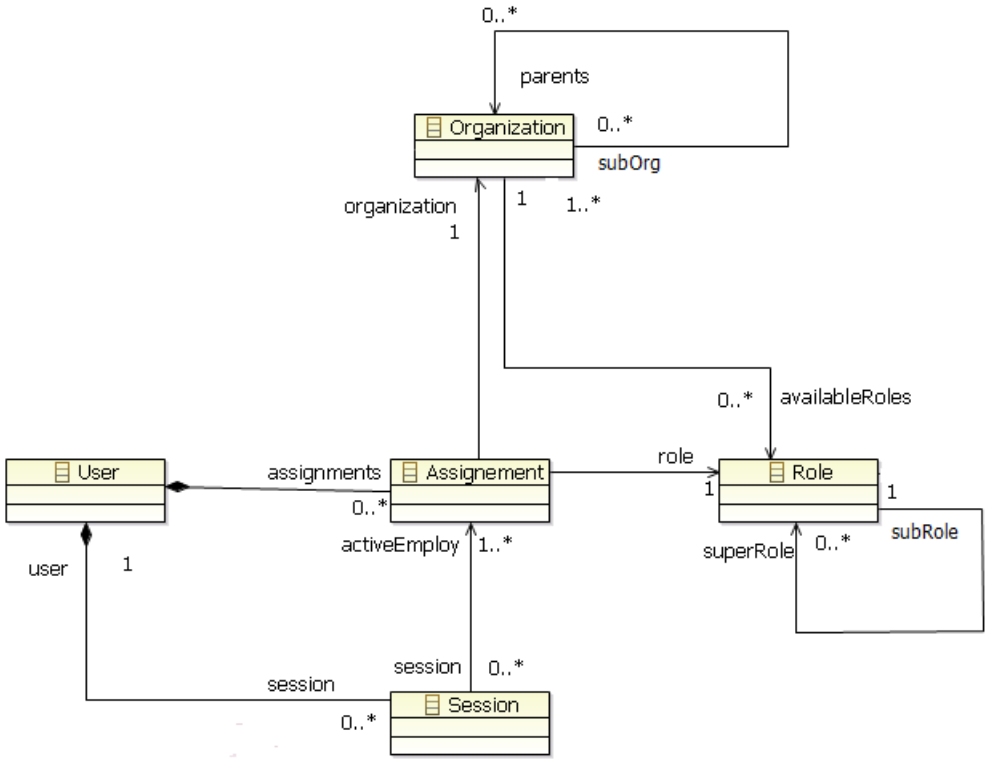


FIG. 4.1 – Portion du méta-modèle de sécurité traduisant le concept de Session

Un utilisateur (méta-classe *User*) peut se connecter à plusieurs sessions (méta-classe *Session*). Chaque session est assignée à une seule organisation (méta-classe *Organization*) et un ou plusieurs rôles (méta-classe *Role*).

Le concept de session représente également, l'aspect dynamique du modèle de sécurité. La formalisation des sessions nous permet de simuler par l'animation l'activité des utilisateurs connectés au système et par la suite de tester et valider leurs droits d'accès suivant les rôles et les organisations qui leur sont affectés. Nous traduisons le concept de session par les spécifications B suivantes :

SETS
$\ldots ;$
$SESSIONS ;$
VARIABLES
$\ldots ,$
$sessions,$
$session\_user,$
$session\_orgRole,$
$\ldots ,$
INVARIANT
$\ldots$
$sessions \subseteq SESSIONS \wedge$
$session\_user \in sessions \rightarrow USERS \wedge$
$session\_orgRole \in sessions \leftrightarrow (ORG \times ROLES) \wedge$
$\ldots$

La formalisation de la méta-classe *Session* suit une logique semblable à la traduction d'une classe dans le modèle fonctionnel. Nous avons l'ensemble SESSIONS contenant les sessions possibles et l'ensemble *sessions* de sessions créées. Notons que SESSIONS n'est pas extrait d'un modèle graphique comme les ensembles ORG et ROLES et est défini dynamiquement par l'analyste.
Une session est, d'une part, liée à un unique utilisateur via la fonction *session_user* et d'autre part liée à des couples (org, role) via la relation *session_orgRole*. Un utilisateur peut créer plusieurs sessions et activer l'ensemble des rôles qui lui sont affectés au sein de l'organisation à laquelle il est connecté. Notons qu'une session concerne une seule organisation et que les couples (organisation, role) appartenant à une session doivent exister dans le modèle. Ces deux conditions sont garanties par les invariants suivants :

INVARIANT
...
/*le codomaine de *session_orgRole* est inclu dans l'ensemble des couples (org, role) du modèle*/
$\forall$ (*ss*, *org*, *ro*).(*ss* $\in$ *SESSIONS* $\wedge$ *ss* $\in$ **dom**(*session_orgRole*) $\wedge$
org $\in$ *ORG* $\wedge$ *ro* $\in$ *ROLES* $\wedge$ (*org*, *ro*) $\in$ *session_orgRole*[{*ss*}]
$\Rightarrow$ (*org*, *ro*) $\in$ *allOrgRoles*) $\wedge$
/*Une session est propre à une seule organisation*/
$\forall$ *ss*.(*ss* $\in$ *sessions* $\wedge$ *ss* $\in$ **dom**(*session_orgRole*)
$\Rightarrow$ **card**(**dom**(**ran**({*ss*} $\lhd$ *session_orgRole*))) = 1)

A titre d'exemple, considérons les valuations suivantes des ensembles *session_orgRole* et *allOrgRoles* :

session_orgRole == { (*s1* $\mapsto$ (*Cardiology* , *Doctor*)),
(*s1* $\mapsto$ (*Radiology* , *Doctor*))}
allOrgRoles == {(Cardiology $\mapsto$ Medical_Employee), (Cardiology $\mapsto$ Doctor),
(Cardiology $\mapsto$ *Nurse)*}

Cette valuation cause la viloation du premier et deuxième invariant en même temps car d'une part le couple (*Radiology* , *Doctor*) affecté à la session *s1* ne figure pas dans l'ensemble *allOrgRoles* du modèle et d'autre part, la session *s1* est liée à plus d'une organisation du système (*Radiology* et *Cardiology* en même temps).

4.2.1 Opération de changement de la session courante

Lors de l'animation du modèle, l'analyste aura besoin de sélectionner une session parmi celles qui sont créées et voir les permissions qui sont autorisées durant cette session. Pour ce faire, nous introduisons, tout d'abord, les variables *currentSession*, *currentUser* et *currentOrg* permettant d'identifier respectivement la session courante, l'utilisateur courant et l'organisation courante :

VARIABLES
..., currentSession, currentUser, currentOrg, ...,
INVARIANT
... currentSession $\in$ *SESSIONS* $\wedge$
currentUser $\in$ *USERS* $\wedge$
currentOrg $\in$ *ORG* $\wedge$

Afin de positionner le filtre de sécurité par rapport à un état courant, nous introduisons l'opération *changeSession* suivante :

changeSession(*ss*) =
PRE
ss : (*sessions* $\cup$ {*noSession*})
THEN
currentSession := *ss* ||
currentUser := *session_user*(*ss*) ||
currentOrg := *currentOrg_session*(*ss*)
END

L'opération permet de mettre à jour les variables *currentSession*, *currentUser* et *currentOrg*. Notons qu'à l'état initial, aucune session n'est séléctionnée et par conséquent aucun utilisateur et aucune organisation courants n'existent. Nous introduisons pour cela les valeurs fictives *noSession* dans l'ensemble SESSIONS, *noUser* dans l'ensemble USER et *noOrg* dans l'ensemble ORG pour pouvoir initialiser respectivement *currentSession*, *currentUser* et *currentOrg*.

4.2.2 Opérations de connexion et déconnexion

L'opération *connect* permet à un utilisateur du système (argument *user*) de créer une session (argument *ss*) dans une organisation (argument *org*) à laquelle il appartient et activer un ensemble de rôles (argument *roleset*) qui lui sont assignés via la relation *user_assign* :

connect(*user*, *ss*, *org*, *roleSet*) =
PRE
/* Contraintes sur les arguments */
user ∈ *USERS* ∧ *org* ∈ *ORG* ∧
roleSet ∈ $\mathcal{P}_1$ (*ROLES*) ∧ *org* ∈ **dom**(**ran**({*user*} ◁ *user_assign*)) ∧
ss ∈ *SESSIONS*-{*noSession*} ∧ *ss* ∉ *sessions* ∧
roleSet ∈ $\mathcal{P}_1$ (*ROLES*) ∧ *roleSet* ⊆ (**ran**({*org*} ◁ **ran**({*user*} ◁ *user_assign*)) ∪
closure1(*Roles_Hierarchy*)[**ran**({*org*} ◁ **ran**({*user*} ◁ *user_assign*))]) ∧
roleSet ⊆ **ran**({*org*} ◁ *allOrgRoles*)
. . .
THEN
sessions := *sessions* ∪ {*ss*} ||
session_user := *session_user* ∪ {(*ss* ↦ *user*)} ||
session_orgRole := *session_orgRole* ∪
⋃ (*ro*).(*ro* ∈ *roleSet* | {*ss* ↦ (*org*, *ro*)})
. . .
END ;

Notons que les contraintes imposées sur les arguments de l'opération *connect*, servent à limiter la connexion uniquement aux utilisateurs autorisés à le faire. Cette autorisation est conditionnée d'une part, par la relation *user_assign* qui indique quel utilisateur est affecté à quelles organisations et quels rôles et d'autre part par la hiérarchie de rôles (**closure1**(*Roles_Hierarchy*)) qui permet de déduire l'ensemble de rôles que l'utiliateur pourra jouer en se connectant. A titre d'exemple, si nous considérons un ensemble SESSIONS valué par *s1* et *s2* et l'initialisation suivante des relations *user_assign* et *Roles_Hierarchy* :

INITIALISATION
Roles_Hierarchy := {(*Doctor* ↦ *Medical_Employee*),
(*Nurse* ↦ *Medical_Employee*),
(*DepartmentDirector* ↦ *Doctor*)
user_assign := {(*Fred* ↦ (*Cardiology*, *DepartmentDirector*)),
(*Jean* ↦ (*Radiology*, *Nurse*)) }

Nous aurons les interfaces de connexion suivantes :

connect(*Fred*, *s1*, *Cardiology*, *DepartmentDirector*, *Doctor*, *Medical_Employee*)
connect(*Fred*, *s2*, *Cardiology*, *DepartmentDirector*, *Doctor*, *Medical_Employee*)
connect(*Jean*, *s1*, *Radiology*, *Nurse*, *Medical_Employee*)
connect(*Jean*, *s2*, *Radiology*, *Nurse*, *Medical_Employee*)

L'opération de connexion prend en compte également les conflits entre les rôles en empêchant l'utilisateur de se connecter avec des rôles conflictuels simultanément. Cette condition est réalisée via le prédicat suivant :

/* Vérification des conflits entre les rôles (DSD) */
∀ (*rs*, *org_dsd*).(*rs* ∈ $\mathcal{P}_1$ (*ROLES*) ∧ *org_dsd* ∈ *ORG* ∧
(*org_dsd* = *org* ∨ *org_dsd* ∈ **closure1**(*Org_Hierarchy*)[{*org*}]) ∧
org_dsd ∈ **ran**(**dom**(*DSD_mutex*)) ∧ *rs* ∈ **dom**(**dom**(*DSD_mutex*) ⩤ {*org*}) ⇒
card((**closure1**(*Roles_Hierarchy*)[*roleSet*] ∪ *roleSet*) ∩ *rs*) < *DSD_mutex*(*rs*, *org*))

Le prédicat permet de vérifier par la relation des conflits dynamiques *DSD_mutex* que l'argument *roleSet* ne contient pas des rôles conflictuels dans l'organisation *org*. A titre d'exemple, si nous considérons l'initialisation suivante des relations *DSD_mutex* et *user_assign* :

INITIALISATION
$DSD\_mutex := \{((\{Doctor,\ Nurse\},\ Cardiology) \mapsto 2)\ \}$
$user\_assign := \{(Fred \mapsto (Cardiology,\ DepartmentDirector)),$
$(Fred \mapsto (Cardiology,\ Nurse))\ \}$

Nous aurons les interfaces de connexion suivantes :

$connect(Fred,\ s1,\ Cardiology,\ DepartmentDirector,\ Doctor,\ Medical\_Employee)$
$connect(Fred,\ s1,\ Cardiology,\ Nurse,\ Medical\_Employee)$

Remarquons que l'utilisateur *Fred* ne peut pas se connecter en même temps avec les rôles *Nurse* et *DepartmentDirector* car *DepartmentDirector* hérite de *Doctor* qui est en conflit avec *Nurse*.

L'opération de déconnexion *disconnect* permet de supprimer une session active en la retirant de l'ensemble des sessions créées. Cette opération met à jour également les relations *session_user* et *session_orgRole*. Les détails de l'opération *disconnect* sont décrits dans les spécifications B de la machine "userAssignement" en Annexe B.

4.2.3 Opérations d'ajout et retrait de rôles

Toujours par souci de validation de politiques de sécurité, il est intéressant de voir ce qu'un utilisateur peut activer ou désactiver comme rôles pendant une session. Pour celà, nous introduisons l'opération d'ajout de rôle dans une session active *add_role_session* et l'opération de retrait de rôle d'une session active *drop_role_session*. L'opération *add_role_session* permet d'ajouter un rôle à l'ensemble de rôles actifs dans une session en respectant les contraintes de conflits entre les rôles (*DSD_mutex*) de la même façon que décrite dans l'opération *connect*. L'opération retrait d'un rôle *drop_role_session* permet de désactiver un rôle dans une session et restreindre éventuellement les droits accordés à l'utilisateur propriétaire de la session. Les opérations *add_role_session* et *drop_role_session* sont détaillées dans la machine "userAssignement" en annexe A.

4.3 Validation de politiques de sécurité basées sur les organisations

4.3.1 Principe de la validation par l'animation

L'animation consiste à dérouler une séquence d'opérations faisant passer le système à valider par différents états (les valeurs des variables) possibles. Cette activité permet à l'analyste de jouer des scénarios d'utilisation du système à partir des modèles abstraits et indépendants de toute architecture. L'animation permet également de s'assurer que les propriétés invariantes (l'invariant du modèle) sont respectées après l'exécution de chaque opération. Dans notre approche, l'animation nous permet de :

– S'assurer que l'invariant du système est respecté quelque soit l'état ou détecter les cas de violations.

– Simuler l'activité des utilisateurs connectés au SI et valider ainsi les politiques de contrôle d'accès modélisées initialement en UML et traduites ensuite en B.
– Valider les propriétés confidentialité et integrité par des scénarios dits normaux.
– Trouver éventuellement des scénarios suspects qui permettent de contourner la sécurité du SI dits scénarios d'attaque

Nous utilisons dans notre activité de validation l'outil ProB dont l'interface est illustrée par la figure 4.2.

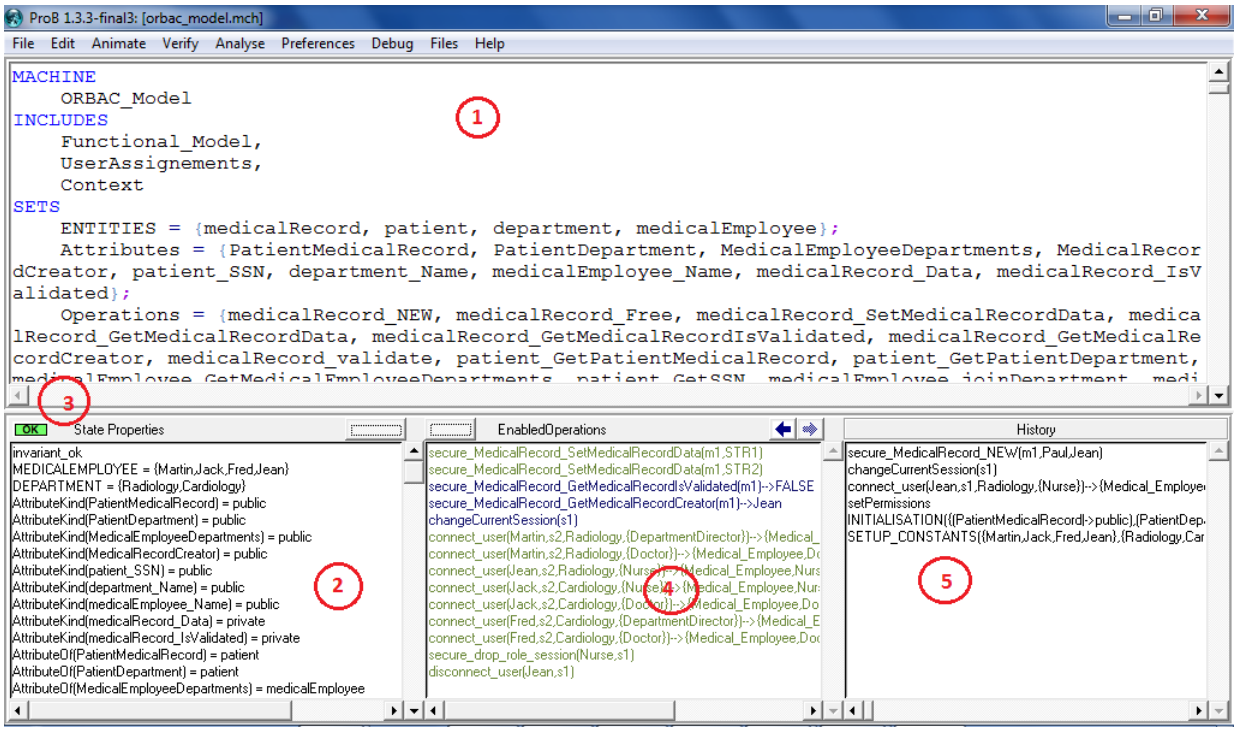


Fig. 4.2 – Interface de ProB

L'interface de l'animateur est composée de l'éditeur de spécifications B (1 dans la figure 4.2), l'affichage de l'état courant du système (2 dans la figure 4.2), l'indicateur de violation de l'invariant (3 dans la figure 4.2), l'ensemble des opérations pouvant être exécutées à partir de l'état courant (4 dans la figure 4.2) et la trace de l'animation (5 dans la figure 4.2). En plus du model-checking, ProB permet du vérifier si un état ou une opération du système, spécifié par l'analyste, est atteignable à partir d'un état donné. Si c'est le cas, l'animateur fournit en plus le chemin (la séquence d'opérations) le plus court trouvé pour atteindre l'état en question. Cette particularité de ProB s'avère très utile dans notre approche de validation car elle nous permettra d'une part de vérifier que pour chaque opération du modèle, il existe au moins un chemin permettant de l'atteindre et d'autre part de chercher d'éventuels scénarios d'attaque.

4.3.2 Exemples de scénarios normaux

Dans cette section, nous analysons trois exemples de scénarios normaux permettant chacun de valider une partie des politiques de sécurité définies dans l'exemple du SI médical 3.2.2 présenté dans le chapitre précédent. Nous présentons l'état initial ainsi que les résultats retournés par l'animateur ProB après l'exécution de chaque scénario.

L'état initial

On considère l'initialisation suivante de notre modèle fonctionnel :

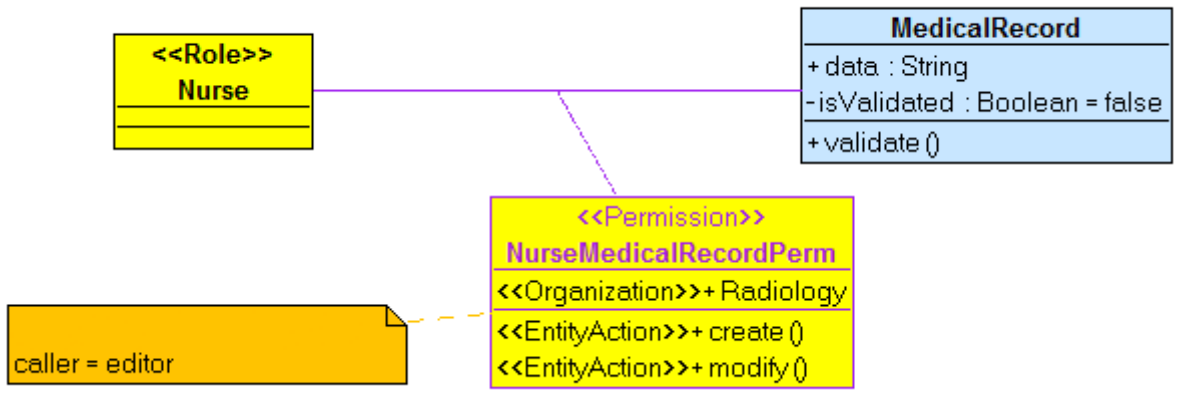
INITIALISATION
$Patient := \{Bob, Paul\}$
$||\ MedicalRecord := \emptyset$
$||\ Department := \{Radiology, Cardiology\}$
$||\ MedicalEmployee := \{Martin, Fred, Jack, Jean\}$
$||\ patientMedicalRecord := \emptyset$
$||\ patientDepartment := \{(Bob \mapsto Cardiology), (Paul \mapsto Radiology)\}$
$||\ MedicalEmployeeDepartment := \{(Fred \mapsto Cardiology),$
$(Martin \mapsto Radiology), (Jean \mapsto Radiology),$
$Jack \mapsto Cardiology)\}$
$||\ medicalRecordCreator := \emptyset$
$\ldots$

L'instance du modèle fonctionnel est composée initialement de deux patients *Paul* et *Bob*. *Paul* est hospitalisé dans le département *Radiology* et *Bob* dans le département *Cardiology*. Aucun enregistrement médical n'est créé auparavant. Les employés médicaux sont initialement répartis comme suit :
– *Jack* et *Fred* travaillent dans le département *Cardiology*.
– *Martin* et *Jean* sont associés au département *Radiology*.
Conformément à cette répartition, ces employés médicaux qui sont aussi des utilisateurs dans le modèle de sécurité auront initialement les affectations des rôles et des organisations suivantes :

$user\_assign := \{(Fred \mapsto (Cardiology, DepartmentDirector)),$
$(Fred \mapsto (Cardiology, Doctor)),$
$(Jack \mapsto (Cardiology, Nurse)),$
$(Martin \mapsto (Radiology, DepartmentDirector)),$
$(Martin \mapsto (Radiology, Doctor)),$
$(Jean \mapsto (Radiology, Nurse))$

Scénario 1

Dans ce scénario, nous visons à valider la politique de sécurité liée à la permission *NurseMedicalRecordPerm* que nous reprenons par la figure 4.3.2.



Cette permission autorise les infirmiers du département *Radiology* à créer et modifier des enregistrements médicaux pour les patients. La contrainte OCL attachée à la permission impose que seul l'infirmier qui a créé l'enregistrement médical a le droit de le modifier. La contrainte ne s'applique donc qu'à l'action de modification. Afin de valider la permission *NurseMedicalRecordPerm*, nous considérons le scénario suivant :

- L'utilisateur *Jean* se connecte au départment *Radiology* avec le rôle *Nurse* en créant une session *s1*
- *Jean* crée un MedicalRecord *m1* pour le patient *Paul*
- *Jean* modifie les données de *m1*
- *Jean* se déconnecte de la session *s1*

L'exécution de ce scénario avec ProB se déroule normalement sans violation de l'invariant ni de préconditions. La trace de l'exécution du scénario 1 avec ProB est donnée comme suit :

```
setPermissions
connect_user(Jean, s1, Radiology, {Nurse})
changeCurrentSession(s1)
secure_MedicalRecord_NEW(m1, Paul, Jean)
Secure_MedicalRecord_SetMedicalRecordData(m1, STR1)
dicsonnect_user(Jean, s1)
```

Par rapport à l'état initial 4.3.2, l'exécution aboutit aux mises à jour suivantes :

$$
\begin{array}{|l|}
\hline
\quad \mathit{MedicalRecord} = \{\mathrm{m1}\} \\
\quad \mathit{patientMedicalRecord} = \{(\mathrm{Paul} \mapsto \mathrm{m1})\} \\
\quad \mathit{MedicalRecord\_IsValidated} = \{(\mathrm{m1} \mapsto \mathrm{FALSE}\ )\} \\
\quad \mathit{medicalRecordCreator} = \{(\mathrm{m1} \mapsto \mathrm{Jean})\} \\
\hline
\end{array}
$$

Nous considérons cet état comme état initial pour le scénario 2.

Scénario 2

Dans ce scénario, nous visons, en premier temps, à valider la politique de sécurité liée à la permission *ReadMedicalRecord* qui autorise les docteurs et les infirmiers de tout l'hôpital à lire les enregistrements médicaux des patients.

En deuxième temps, nous validons l'opération *joinDepartment* de la permission DirectorPerm pemettant à un chef de service d'associer un docteur à son département :

- *Jack* se connecte au département *Cardiology* avec le rôle *Nurse* en créant une session *s1*.
- *Jack* a accès en lecture à tous les enregistrements médicaux de l'hôpital, il lit l'enregistrement *m1* du patient *Paul* localisé dans le département *Radiology*.
- *Jack* se déconnecte de la session *s1*
- *Martin* se connecte au département *Radiology* avec le rôle *DepartmentDirector* en créant une session *s2*.
- *Martin* associe *Jack* au département *Radiology*.
- *Martin* se déconnecte de la session *s2*

La trace de l'exécution du scénario 2 avec ProB est la suivante :

```
connect_user(Jack, s1, Cardiology, {Nurse})
changeCurrentSession(s1)
```

```
secure_MedicalRecord_GetMedicalRecordData(m1)
dicsonnect_user(Jack, s1)
connect_user(Martin, s2, Cardiology, {departmentDirector})
changeCurrentSession(s2)
secure_MedicalEmployee_joinDepartment(Jack, Radiology)
dicsonnect_user(Martin, s2)
```

L'éxécution du scénario 2 avec ProB aboutit à l'état suivant :

/*Mises à jour relatives au modèle fonctionnel*/
MedicalEmployeeDepartment={(*Fred* ↦ *Cardiology*),
(*Martin* ↦ *Radiology*), (*Jean* ↦ *Radiology*),
(*Jack* ↦ *Radiology*) }
/*Mises à jour relatives à la politique de sécurité*/
user_assign == {...
(*Jack* ↦ (*Radiology*, *Nurse*))}

Scénario 3

Dans ce scénario, nous visons à valider les opérations de modification et validation d'un enregistrement médical par les docteurs, autorisées via la permission *DoctorMedicalRecordPerm* :
– *Jack* se connecte au département *Radiology* avec le rôle *doctor* en créant une session *s1*.
– *Jack* modifie l'enregistrement médical *m1*.
– *Jack* se déconnecte de la session *s1*.
– *Martin* se connecte au département *Radiology* avec le rôle *doctor* en créant une session *s2*.
– *Martin* lit l'enregistrement Médical *m1*.
– *Martin* valide l'enregistrement médical *m1*.
– *Martin* se déconnecte de la session *s2*.
La trace de l'exécution du scénario 3 avec ProB est comme suit :

```
connect_user(Jack, s1, Radiology, {Doctor})-->{Medical_Employee, Doctor}
changeCurrentSession(s1)
secure_MedicalRecord_SetMedicalRecordData(m1, STR2)
dicsonnect_user(Jack, s1)
connect_user(Martin, s2, Radiology, {Doctor})-->{Medical_Employee, Doctor}
changeCurrentSession(s2)
secure_MedicalRecord_GetMedicalRecordData(m1)--> STR2
secure_MedicalRecord_validate(m1)
dicsonnect_user(Martin, s2)
```

L'exécution du scénario 3 introduit les mises à jour suivantes :

MedicalRecord_Data = {(m1 ↦ STR1)}
MedicalRecord_IsValidated = {(m1 ↦ TRUE)}

4.3.3 Exemple d'un scénario d'attaque

Les scénarios d'attaque sont des scénarios qui permettent de contourner la sécurité des SI sans violer aucune contrainte de sécurité. L'étude de ces scénarios permet de détecter les vulnérabilités des SI dès les phases conceptuelles et vise à prévenir les risques d'intrusion en vue de renforcer d'avantage la politique de sécurité.

A la base de l'exemple illustratif 3.2.2, nous proposons un scénario qui révèle un comportement abusif et qui peut être jugé comme étant un scénario d'attaque.

Etat initial

Nous supposons qu'initialement, un enregistrement médical *m1* a été créé par le docteur *Jack* pour le patient *Bob* hospitalisé dans le département *Cardiology*. L'enregistrement n'est pas encore validé est peut donc être modifié ou supprimé par un docteur faisant partie du même département.
Initialement, *Martin* qui est chef de service du département *Radiology* ne fait pas partie du département *Cardiology*.

Scénario

- *Martin* se connecte au département *Radiology* avec le rôle *DepartmentDirector* en créant une session *s1*.
- *Martin* rejoint le département *Cardiology* en exécutant l'opération *secure_joinDepartment* sur lui-même.
- *Martin* se déconnecte de la session *s1*.
- *Martin* se connecte au département *Cardiology* avec le rôle *doctor* en créant une session *s2*.
- *Martin* lit l'enregistrement Médical *m1*.
- *Martin* détruit l'enregistrement médical *m1*.
- *Martin* se déconnecte de la session *s2*.
- *Martin* se reconnecte au département *Radiology* avec le rôle *DepartmentDirector* en créant une session *s3*.
- *Martin* quitte le département *Cardiology* en exécutant l'opération *secure_leaveDepartment* sur lui-même.

En analysant ce scénario, nous pouvons remarquer que *Martin* abuse de ses droits d'accès lui permettant de rejoindre ou quitter un département pour accéder à un dossier médical d'un patient d'un autre département. Ses droits lui ont permis de détruire le dossier médical créé par un autre docteur d'un autre département.

L'étude du scénario 3 nous a permis de renforcer la politique de sécurité en ajoutant une contrainte sur la permission *DirectorPerm*. La contrainte empêche désormais un chef de service d'exercer les opérations *joinDepartment* et *leaveDepartment* sur lui-même :

$$\boxed{\text{org} \neq \text{currentUser}} \text{ /* ou org} = \text{CurrentOrg */}$$

Instance étant l'instance du l'utilisateur qui va rejoindre ou quitter le département et *currentUser* est l'utilisateur faisant appel à l'opération et qui possède forcément le rôle *departmentDirector*.

4.4 Conclusion

Dans ce chapitre, nous avons étendu la formalisation de politiques de sécurité par l'introduction du concept de session. Ce concept dynamique a permis d'assister la validation par l'animation

des modèles obtenus. L'activité de validation, a permis de s'assurer que les spécifications sont correctes et répondent aux besoins et aux exigences de sécurité. D'un autre côté, ce chapitre a permis d'ouvrir une nouvelle perspective qui est l'analyse des scénarios d'attaque dans les systèmes d'information.

Operational Semantics for EB^3 and translation to Nu-SMV

Dimitris Vekris and Catalin Dima

LACL, Université Paris-Est
61 av. du Général de Gaulle
94400 Créteil, France
{dimitrios.vekris,dima}@u-pec.fr

1 Introduction

EB^3 [11] is a specification language for information systems. The core of the EB^3 language consists of process algebraic specifications describing the behavior of the entity types in a system, and attribute function definitions describing the entity attribute types. In [8], an EB^3 interpreter under the name of EB^3 Process Algebra Interpreter EB^3PAI is implemented. It evaluates EB^3 specifications against an optimized set of reduction rules named PAI, which is proven equivalent to the standard set given in [11]. Though PAI is a considerable improvement to the standard rule system, evaluating the attribute functions still implicates traversing the whole system trace adding up to the major problem in model-checking, state-space explosion.

Specification errors in EB^3 can be detected with the aid of invariants also known as static properties or temporal properties known as dynamic properties. From a state-based point of view, an invariant describes a property on state variables that must be preserved by each transition or event. A dynamic property relates several events. Tools such as Atelier B [7] provide methodologies on how to define and prove invariants. In [13], an automatic translation of EB^3's attribute functions with B is attempted. Although the B Method [1] is suitable for specifying static properties, dynamic properties are very difficult to express and verify in B. A relevant comparison between state-based and event-based specifications can be found in [9].

The verification of temporal properties against EB^3 specifications has been the subject of some work in the recent years. [10] compares six model checkers for the verification of Information System case studies. The specifications used in [10] derive from industrial case studies, but the prospect of a uniform translation from EB^3 program specifications is not studied. [4] summarizes the fundamental difficulties for designing a compiler that would translate a given EB^3 specification to process algebra LOTOS-NT [3], an approach that would make use of the verification suite $CADP$ [12] to verify properties against the source system. In short, the majority of these works treat specific case studies drawn from the information systems domain leading to ad-hoc verification translations, but nonetheless lacking in generalization capability.

EB^3 is endowed with common process algebraic operations such as sequential and parallel composition. Several toolboxes have been implemented to support the design and verification of systems specified with process algebras. For example, $CADP$ [12] is a verification toolbox for asynchronous concurrent systems, which allows very large state spaces to be handled. We propose the translation of EB^3 into LOTOS-NT to enable EB^3 users to access the verification techniques available in the $CADP$ toolbox. In order to achieve this we need a formal semantics for EB^3.

However, the translation of EB^3 to LOTOS-NT specifications is not evident. The reason lies in the use of state variables. Global state variables are absent in process algebras such as CSP [14], CCS [16] and LOTOS-NT [3], but their effect can be simulated with synchronisation between process threads. For example in CSP, variables can be simulated in the following way:

$$Components = \{1..N\}\ DataType = \{0..M\}$$
$$channel\ read,\ write : Components.DataType$$
$$Var(x) = read?\_!x \leftarrow Var(x)[\,]write?\_?y \leftarrow Var(y)$$

Process Var is then placed in parallel to the other processes (Components) and is synchronized on $\{read.x.y,\ write.x.y \mid x \leftarrow Components,\ y \leftarrow DataType\}$. Although, EB^3 programmers cannot define global variables explicitly, EB^3 permits the use of a single state variable, the system trace, in predicates of guard statements. Attribute functions can express the evolution of entity attributes in time, option which introduces an indirect notion of state to the language. As a result, expressions of the form $p \rightarrow E$ can be written, where p refers to the system trace and E is a valid EB^3 expression.

We propose a formal semantics for EB^3 that treats attribute functions as state variables. This semantics will serve as the basis for applying a simulation strategy of state variables in LOTOS-NT similar to the one shown for CSP above. Intuitively, coding attribute functions as part of the system state is beneficial from a model-checking point of view as the new formalisation dispenses with the system trace. Our main contribution is an operational semantics in which attribute functions are computed during program evolution and stored into program memory. We show that this operational semantics is bisimilar with the original, trace-based operational semantics, in the sense that, for each EB^3 specification, the transition system corresponding with its memory-based semantics is bisimilar with the transition system corresponding with its trace-based semantics. We also come up with a number of restrictions on the syntax of EB^3 that result in sound EB^3 specifications. We view this unification of the EB^3's process algebra and attribute functions as a fundamental step towards the translation of EB^3 into LOTOS-NT and model-checking of EB^3 specifications.

We then develop an algorithm that translates EB^3 specifications to $Nu-SMV$ code. We use Petri Nets [17] as an intermediate model for EB^3's process algebra. In the case of attribute functions the translation is done automatically to equivalent $Nu-SMV$ code. We also present a case study of library management system, over which consistency property are verified. The result of our work is

a compiler that translates a system specification in EB^3 directly to Nu-SMV code.

2 Example

In this section, we present the user requirements for a basic library management system that manages book loans and reservations to members. Then we give a possible EB^3 specification that matches these requirements. This example is rather simplistic and in no way constitutes a case study. The reason it is given is to shed some light on the principal ideas developed throughout the paper.

The system's requirements can be summarized in the following sentences:

1. A book can be acquired by the library. It can be discarded, but only if it has not been lent.
2. An individual must join the library in order to borrow a book.
3. A member can relinquish library membership only when all his loans have been returned.
4. A member cannot borrow more than the loan limit defined at the system level for all users.

The EB^3 specification can be found in Figure 2. For conciseness, we removed

$$
\begin{aligned}
main &= (\ |||\ bId : \text{BID} : book(bId)\ )\ |||\ (\ |||\ mId : \text{MID} : member(mId)*\ ) \\
book(bId : \text{BID}) &= Acquire(bId).\ borrower(T,\ bId) = \bot \rightarrow Discard(bId) \\
member(mId : \text{MID}) &= Register(mId).\ (\ |||\ bId : \text{BID} : loan(mId, bId)*\ ).\ Unregister(mId) \\
loan(mId : \text{MID},\ bId : \text{BID}) &= borrower(T,\ bId) = \bot \wedge nbLoans(T, mId) < \text{NbLoans} \\
&\quad \rightarrow Lend(bId,\ mId).\ Return(bId)
\end{aligned}
$$

$nbLoans(T{:}\ tr,\ mId{:}\ \text{MID}){:}\ Nat_\bot =$	$borrower(T{:}\ tr,\ bId{:}\ \text{BID}){:}\ \text{MID} =$
match T **with**	**match** T **with**
$[\,] \rightarrow \bot$	$[\,] \rightarrow \bot$
$\mid T'.\ Lend(bId, mId) \rightarrow nbLoans(T', mId) + 1$	$\mid T'.\ Lend(bId, mId) \rightarrow mId$
$\mid T'.\ Register(mId) \rightarrow 0$	$\mid T'.\ Return(bId) \rightarrow \bot$
$\mid T'.\ \text{Unregister(mId)} \rightarrow \bot$	$\mid \_ \rightarrow borrower(T', bId)$
$\mid T'.\ Return(bId) \wedge mId = borrower(T, bId)$	
$\rightarrow nbLoans(T', mId) - 1$	
$\mid \_ \rightarrow nbLoans(T', mId)$	

Fig. 1. Specification

the signature of the atomic process expressions Lend, Register etc. and did not give explicitly the data sets BID and MID, which contain the instances of book and member IDs respectively. EB^3's expressivity is evident. The datasets can be modified on the signature with no effect on the rest of the code.

Function *main* in the script is a parallel interleaving operation between the various *book* and *member* processes. Process *book* expresses the need for a book to be acquired in order to be engaged in a *loan* process and eventually be discarded by the library. It relates to a simple ordering constraint between *Acquire*

and *Discard*. Note that condition $borrower(T,\ bId) = \perp$ is introduced to avoid discarding book bId if it is currently lent. Deciding who is the current borrower involves executing attribute function *borrower*. The use of attribute functions is not adherent to standard process algebra practices as it may naively trigger the complete traversal and inspection of the system trace. That is the case for attribute function *borrower*.

One may come up with simpler specifications based solely on process algebra operations when the constraints imposed by the user's requirements imply loose interdependence between entities and associations. With the extra hypothesis that all books are acquired by the library at the beginning and are eventually discarded if there are no demands for books, the attribute function *borrower* can be erased from *main*'s script. Note that the user's requirements are not contradicted, though the system's behaviour changes dramatically. Hence, we get the following specification:

$$main = (\ |||\ bId : \text{BID} : Acquire(bId)\ ).\ (\ |||\ mId : \text{MID} : member(mId)*\ ).\\ (\ |||\ bId : \text{BID} : Discard(bId)\ )$$

However, programming naturally in a purely process-algebraic style without attribute functions in EB^3 may not always be obvious. In fact, there are cases where ordering constraints involving several entities are quite difficult to express without guards and lead to less readable specifications than equivalent guard-oriented solution in EB^3 style. This is the case for process *loan*, whose guard illustrates the conditions under which a *Lend* can occur, notably when nobody else has borrowed the book and attribute function *nbLoans* is found less than a fixed upper bound *NbLoans*.

Furthermore, it would be of great interest from a model-checking point of view to develop techniques for discharging the system specification from attribute functions. To address the issue, we consider exploring the implicit notion of state inherent in attribute functions by redeveloping the system specification in LOTOS-NT. The translation resembles to the variable simulation presented above in the context of CSP. The primary obstacle we are opposed to is the limited scope of variables in LOTOS-NT notably between threads that execute in parallel. For example, in $process_1|||process_2$ changes induced to the state variables by $process_1$ are not communicated to $process_2$. In order to have a sound translation from EB^3 to LOTOS-NT, we need to reformulate EB^3, which is the topic of Section 3 and define a memory-based semantics that simulates the effect of attribute functions as is done in Section 4.

3 EB³'s Specification

We proceed with the formal definition of EB^3. The EB^3 method [11] is tailored for specifying the functional behaviour of information systems. A standard EB^3 specification comprises:

1. a class diagram representing entity types and associations for the information system being specified;

2. a process algebra describing the information system, i.e. the valid traces of execution describing its behaviour;
3. a set of entity attribute definitions, which are recursive functions on the system trace; and
4. input/output rules, to specify outputs for input traces, or SQL used to specify queries on the business model

Our approach deviates from the standard specification in that it considers the process algebra and the attribute functions as a whole. Our attention is drawn to bullets 2 and 3.

We make use of the following sets of symbols, domains, and variables to define the syntax of EB^3 process expressions:

1. A finite set of *entity types* T_{en}, corresponding with the entity types in [11]. In the example of the previous section, it is $T_{en} = \{\text{BID,MID}\}$.
2. A set of typed variables V_{en}, with a mapping $type : V_{en} \longrightarrow T_{en}$ giving the type of each variable. We use V instead of V_{en} for simplicity. It is $T_{en} = \{\text{bId,mId}\}$.
3. For each type $t \in T_{en}$, a *finite domain* D_t for the variables of type t. The finiteness of D_t is needed due to the universal quantification over variables that is employed in EB^3. We denote $D = \bigcup_{t \in T_{en}} D_t$. In the example, D refers to the instances of BID and MID.
4. A finite set of *typed action names* Act, endowed with a *typing* mapping $type : Act \longrightarrow T_{en}^*$ associating with each action the sequence of parameter types that the respective action takes. The length of $type(a)$ for each $a \in Act$ is the *arity* of a, and is denoted $ar(a)$. It is $type(Lend) = (\text{BID,MID})$ and $ar(Lend) = 2$.
5. The derived set of *atomic process expressions* $AtPEx$ defined as follows:

$$AtPEx = \left\{ a(x_1, \ldots, x_k) \mid a \in Act, type(a) = t_1 \ldots t_k, x_i \in V_{t_i} \cup D_{t_i} \right\}$$

6. A separate type tr denoting traces of process expressions (to be defined further), and two separate variables T, T' whose type is tr.
7. A set of *computational types* T_{comp}, corresponding with the results of attribute function evaluation or any other types like integers, booleans, lists or sets of values in D etc. On this extra set of types we consider usual operators like set or list operators, arithmetic or boolean operators etc. that we do not formally define here. In the example presented in the previous section, we do not have complex *computational types*.
8. A set of *simple function names* $Func$ defined:

$$Func = \left\{ f(x_1, \ldots, x_k) \mid type(f) = (t_1 \ldots t_k), x_i \in V_{t_i} \right\}$$

   Among these function names a special name *main* is used with $type(main) = Nil$, whose signification is similar to that of *main* in programming language C. Nil stands for the empty vector. We refer to function names *nbLoans* and *borrower* of the library management specification.

9. A set of *attribute function names* $AtFct$, endowed with a typing mapping $type : AtFct \longrightarrow tr \times (T_{en})^*$.
10. Two derived sets for *attribute function expressions* and one for *attribute function tail expressions*, defined as follows:

$$AtFEx_1 = \{a(T, x_1, \ldots, x_k) \mid a \in AtFct, type(a) = tr \times (t_1 \ldots t_k), x_i \in V_{t_i}\}$$
$$AtFEx_2 = \{a(T, x_1, \ldots, x_k) \mid a \in AtFct, type(a) = tr \times (t_1 \ldots t_k), x_i \in V_{t_i} \cup D_{t_i}\}$$
$$TlAtFEx = \{a(tl(T), x_1, \ldots, x_k) \mid a \in AtFct, type(a) = tr \times (t_1 \ldots t_k), x_i \in V_{t_i} \cup D_{t_i}\}$$

Here tl denotes the *tail* function which, given a trace, removes the first element of the trace and returns the rest of the trace. $AtFEx_1$ contains only formal parameters $x_i : t_i$, whereas $AtFEx_2$ may contain both formal and actual parameters.

11. The derived set of *attribute function definitions*, defined by the following grammar:

$$AtFDef ::= AtFEx_1 : ret\_type \stackrel{def}{=} \textbf{match } \mathcal{T} \textbf{ with } ListCases$$
$$ListCases ::= Cond\ ;\ CaseEx \mid Cond\ ;\ CaseEx\ ;\ ListCases$$
$$CaseEx ::= [\,] \longrightarrow Expr \mid (T' \cdot AtPEx) \longrightarrow Expr$$

where $Expr$ denotes the set of typed expressions constructed from objects in the set $AtFEx_2 \cup TlAtFEx \cup D \cup V_{en}$, integers, booleans and all the extra domains for the extra types T_{comp}, with the aid of operators like set or list operators, arithmetic or boolean operators etc, $Cond$ is obtained from the set of typed expressions $Expr$ using usual predicates: $=, \leq, \subseteq$, etc. $ret\_type : T_{en} \cup T_{comp}$ is the type of $AtFEx_1$ after $Expr$'s evaluation, $[\,]$ is the empty trace and $T' \cdot AtPEx$ is the right append of $AtPEx$ to T'. Note also that $Expr$ contains recursive calls to $AtFEx_2$ on trace T.
These are subject to the following restriction:

(*) The set of *attribute function* names is ordered, $AtFct = \{f_1, \ldots, f_n\}$, such that for each $i \leq n$ the attribute function expressions containing f_j with $j > i$ that occur in the definition of attribute function f_i should belong to $TlAtFEx$ – that is, the order of evaluation of the attribute function definitions is defined by the order on $AtFct$. The same restriction is applied to the corresponding $Cond$ expressions.

Hence, an *attribute function* declaration part for an EB^3 specification has the form:

$$AtFDecl ::= AtFDef_{i_1}\ ;\ \ldots\ ;\ AtFDef_{i_n}$$
$$AtFDef_{i_j} \stackrel{def}{=} f_{i_j} : ret\_type_{i_j} = \textbf{match } \mathcal{T} \textbf{ with } ListCases_{i_j}$$
$$\text{where for all } 1 \leq i_l \neq i_p \leq n,\ f_{i_l} \neq f_{i_p}$$
$$ListCases_i ::= Cond_{i,1}\ ;\ CaseEx_{i,1}\ ;\ \ldots\ Cond_{i,m_i}\ ;\ CaseEx_{i,m_i}$$

where the sequence $\{i_j,\ 1 \leq j \leq n\}$ respects restriction 11(*).

This restriction is satisfied by the library management system specification as both attribute functions *nbLoans* and *borrower* contain calls to *nbLoans* and *borrower* parameterized on the tail of the current trace. Besides, *nbLoans* makes call to *borrower* with parameter the current trace, fact which determines the priority of *borrower* over *nbLoans* on execution.

12. The set of *guard expressions* $GExp$, obtained from the set of typed expressions $Expr$ using usual predicates: $=, \leq, \subseteq$, etc like $Cond$ above.

With these sets defined, EB^3 *program specifications* are given by the following grammar:

$$
\begin{aligned}
EB^3 &::= AtFDecl \ ; \ ListFunc \\
ListFunc &::= Func \ = \ ExpDecl \mid Func \ = \ ExpDecl \ ; \ ListFunc \\
AtFDecl &::= AtFDef \mid AtFDef \ ; \ AtFDecl \\
ExpDecl &::= \surd \mid \lambda \mid a(\overline{v}) \mid ExpDecl.ExpDecl \mid ExpDecl|ExpDecl \mid ExpDecl^* \\
&\quad \mid ExpDecl|[\Delta]|ExpDecl \mid |\ \overline{x}{:}\overline{V}{:}ExpDecl \mid |[\Delta]|\overline{x}{:}\overline{V}{:}ExpDecl \\
&\quad \mid GExp \longrightarrow ExpDecl \mid Func
\end{aligned}
$$

where $\overline{x} \in 2^{V_{en}}$, $a \in Act$, $\overline{v} \in (V_{en} \cup D)^*$ is a tuple of variables or domain values having the same type as $type(a)$, and $\Delta \subseteq Act$.

The intended meaning of $ExpDecl$ is the following: $\surd$ is the termination process and λ is the idle process. $ExpDecl.ExpDecl$ denotes the Sequential Composition, $ExpDecl|ExpDecl$ is the Non-Deterministic Choice, $ExpDecl*$ is the Kleene Closure and $ExpDecl|[\Delta]|ExpDecl$ is Parallel Composition with Synchronisation on common $AtPEx$ whose function names belong to set Δ. The Quantified Choice operator $|\overline{x} : \overline{V} : ExpDecl$ extends $ExpDecl|ExpDecl$, and the operator $|[\Delta]|\overline{x} : \overline{V} : ExpDecl$ generalizes $ExpDecl|[\Delta]|ExpDecl$ respectively. Our definitions for the quantified versions of Choice and Parallel Composition operations stand for extended versions of the corresponding operations defined in [11]. It is $\overline{V} \subseteq \Pi_{x_i \in \overline{x}} V_{t_i}$ for $x_i : t_1, \ldots, x_k : t_k$ and $m \in \mathbb{N}$. A similar hypothesis to $11(*)$ can be made for $ListFunc$ to avoid circular dependencies on function calls in general.

EB^3 process expressions must also satisfy a number of restrictions. To this end, we say that a variable $x \in V$ is *bound* in an expression E if all its occurrences are within the scope of a quantifier, and a formula is called *closed* if all variables occurring in the formula are bound. The following restrictions apply to EB^3 expressions:

1. Each attribute function that is used in the guards of an EB^3 process expression must have exactly one definition and its type coïncides with the corresponding *ret_type* in $AtFDecl$.
2. Binary operators $|[\Delta]|$ can only be applied to EB^3 process expressions in which, for each action name $a \in \Delta$, if $a(\overline{x})$ occurs in one of the operands then all the variables in $\overline{x}$ must be bound in that operand.

This restriction forbids expressions of the type

$$a(x_1)[a]a(x_2)$$

which may be interpreted as either a single process being executed, when both variables have the same value, or two different processes being executed in an arbitrary order, when the two variables are instantiated with different values.

3. A similar constraint must be satisfied by expressions of the type $|[\Delta]|\overline{x}:\overline{V}:E$, namely for each action name $a \in Act$ for which $a(\overline{y})$ occurs in E for some tuple of variables and domain values $\overline{y} \in (V \cup D)^*$, the variables in $\overline{y}$ that do not occur in $\overline{x}$ must be bound in E.

It is obvious that all formulas used in the previous example are *closed*. Process expressions may also utilize derived operations: $E^+ = E.E*$, $E^{0..1} = E|\lambda$, $E_1|||E_2 = E_1|[\emptyset]|E_2$, $E_1 \parallel E_2 = E_1|[\alpha(E_1) \cap \alpha(E_2)]|E_2$, where E, E_1 and E_2 belong to $ExpDecl$ and $\alpha(E)$ denotes the set of atomic process names that appear in E.

4 Semantics

We give three Operational Semantics for EB^3. The first, called Trace Semantics Sem_T is the standard semantics defined in [11] adapted to the new EB^3 definitions. The second called Trace/Memory Semantics $Sem_{T/M}$ is the extension of the standard semantics, where attribute functions are computed during program evolution and stored into program memory. Trace is given explicitly as part of a given state. By removing the trace from each state in $Sem_{T/M}$ we obtain a third semantics for EB^3 specifications which we name Memory Semantics Sem_M.

4.1 Trace Semantics

The Trace semantics is given in Figure 2; it is similar to the classic semantics defined in [11]. The difference lies in the addition of attribute functions and their effect on the reduction rules. Note also that the trace T is given explicitly in the system's state. Thus, the state w.r.t Sem_T is the tuple (E, T), where E is the expression of EB^3 that remains to be consumed. We focus on the non-trivial rules.

In Rule $(R_T - 2)$, $GE \in GExp$ is a guarded expression. If $E \in ExpDecl$ is reduced to E' and $\|GE\|$ is true i.e. the evaluation of GE is true then $GE \Rightarrow E$ is reduced to E'. To compute $\|GE\|$, we note that GE is obtained from the set of typed $Expr$ using usual predicates such as $=$, $\wedge$ and $\subseteq$. We adopt the classic intepretation for these predicates as known from Peano arithmetics, Boolean logic and Set theory. The attribute functions that participate syntactically in GE are evaluated using the recursive definition of the attribute functions on the current trace T.

$$(R_T-1): \frac{\rho \in AtPEx}{(\rho, T) \rightarrow^{\rho} (\surd, T \cdot \rho)} \qquad (R_T-1'): \frac{}{(\lambda, T) \rightarrow^{\lambda} (\surd, T)}$$

$$(R_T-2): \frac{(E, T) \rightarrow^{\rho} (E', T \cdot \rho)}{(GE \Rightarrow E, T) \rightarrow^{\rho} (E', T \cdot \rho)} \|GE\|$$

$$(R_T-3): \frac{(E_1, T) \rightarrow^{\rho} (E_1', T \cdot \rho)}{(E_1.E_2, T) \rightarrow^{\rho} (E_1'.E_2, T \cdot \rho)} \quad (R_T-4): \frac{(E, T) \rightarrow^{\rho} (E', T \cdot \rho)}{(\surd.E, T) \rightarrow^{\rho} (E', T \cdot \rho)}$$

$$(R_T-5): \frac{(E_1, T) \rightarrow^{\rho} (E_1', T \cdot \rho)}{(E_1|E_2, T) \rightarrow^{\rho} (E_1', T \cdot \rho,)} \quad (R_T-6): \frac{(E_2, T) \rightarrow^{\rho} (E_2', T \cdot \rho,)}{(E_1|E_2, T) \rightarrow^{\rho} (E_2', T \cdot \rho)}$$

$$(R_T-7): \frac{}{(E*, T) \rightarrow^{\lambda} (\surd, T)} \qquad (R_T-8): \frac{(E, T) \rightarrow^{\rho} (E', T \cdot \rho)}{(E*, T, M) \rightarrow^{\rho} (E' \cdot E*, T \cdot \rho,)}$$

$$(R_T-9): \frac{}{(\surd|[\Delta]|\surd, T) \rightarrow^{\lambda} (\surd, T)}$$

$$(R_T-10): \frac{(E_1, T) \rightarrow^{\rho} (E_1', T \cdot \rho) \qquad (E_2, T) \rightarrow^{\rho} (E_2', T \cdot \rho)}{(E_1|[\Delta]|E_2, T) \rightarrow^{\rho} (E_1'|[\Delta]|E_2', T \cdot \rho)} in(\rho, \Delta)$$

$$(R_T-11): \frac{(E_1, T) \rightarrow^{\rho} (E_1', T \cdot \rho)}{(E_1|[\Delta]|E_2, T) \rightarrow^{\rho} (E_1'|[\Delta]|E_2, T \cdot \rho)} \neg in(\rho, \Delta)$$

$$(R_T-12): \frac{(E_2, T) \rightarrow^{\rho} (E_2', T \cdot \rho)}{(E_1|[\Delta]|E_2, T) \rightarrow^{\rho} (E_1|[\Delta]|E_2', T \cdot \rho)} \neg in(\rho, \Delta)$$

$$(R_T-13): \frac{(E[\overline{x} := \overline{t}], T, M) \rightarrow^{\rho} (E', T \cdot \rho, M')}{(|\overline{x} : \overline{V} : E, T) \rightarrow^{\rho} (E', T \cdot \rho)} \overline{t} \in \overline{V}$$

$$(R_T-14): \frac{(E[\overline{x} := \overline{t}], T) \rightarrow^{\rho} (E', T \cdot \rho)}{(P(\overline{t}), T) \rightarrow^{\rho} (E', T \cdot \rho)} P(\overline{t}) = E$$

$$(R_T-15): \frac{(E[\overline{x} := \overline{t}], T) \rightarrow^{\rho} (E', T \cdot \rho)}{(|[\Delta]|\overline{x} : \overline{V} : E, T) \rightarrow^{\rho} (E' \ |[\Delta]| \ (|[\Delta]|\overline{x} : \overline{V} \setminus \{\overline{t}\} : E), T \cdot \rho)}$$

$$(R_T-16): \frac{}{(|[\Delta]|\overline{x} : \emptyset : E, T) \rightarrow^{\rho} (\surd, T \cdot \rho)}$$

Fig. 2. Sem_T

Rule $(T_M - 10)$ treats the Interleave operation with Synchronisation. If expressions E_1, E_2 reduce to E_1', E_2' and the atomic process that is added to the system trace i.e. ρ is in set Δ then $E_1|[\Delta]|E_2$ reduces to $E_1'|[\Delta]|E_2'$. Note that Δ contains the set of action names that should be synchronised i.e. $\Delta \subseteq Act$ for operands E_1, E_2. Function α takes an object E in $ExpDecl$ and returns the action names of the atomic process expressions that constitute E.

$$in(\rho, \Delta) = \exists \alpha \in \Delta,\ \rho = \alpha(a_1, \ldots, a_k)\ \wedge\ type(\alpha) = (t_1, \ldots, t_k)\ \wedge\ a_i \in D_{t_i}$$

Rule $(T_T - 11)$ treats the Interleave operation without Synchronisation. If expression E_1 is reduced to E_1' and the atomic process to be added to the system trace is not in Δ then $E_1|[\Delta]|E_2$ reduces to $E_1'|[\Delta]|E_2$. $(T_T - 12)$ is symmetrical to $(T_T - 11)$.

Rule $(R_T - 13)$ is the quantified Choice. If $\overline{V} \subseteq \Pi_{x_i \in \overline{x}} V_{\alpha_i}$ for $\overline{x} = (x_1 : \alpha_1, \ldots, x_m : \alpha_m)$ and $E[\overline{x} := \overline{t}]$ reduces to E' then $|\overline{x} : \overline{V} : E$ reduces to E'. Notation $E[\overline{x} := \overline{t}]$ refers to a paramaterised expression E on $\overline{x}$ where $\overline{x}$ has been replaced by $\overline{t} \in \overline{V}$.

Rule $(R_T - 14)$ presupposes a condition of the form $P(\overline{t}) = E$. This equation must be found in $ListFunc$ of the EB^3 specification. In case of absence a compilation error should be invoked. Then if $E[\overline{x} := \overline{t}]$ is reduced to E', $P(\overline{t})$ should reduce to E'.

Rule $(R_T - 15)$ and $(R_T - 16)$ treat the quantified Parallel Composition operation with Synchronisation. If $\overline{t} \in \overline{V}$ and $E[\overline{x} := \overline{t}]$ reduces to E' then $|[\Delta]|\overline{x} : \overline{V} : E$ should reduce to $E' \ |[\Delta]| \ (|[\Delta]|\overline{x} : \overline{V} \setminus \{\overline{t}\} : E)$, where the second operand for $|[\Delta]|$ is a new quantified Parallel Composition and the set of possible values for $\overline{x}$ does not contain $\overline{t}$ any more. Rule $(R_T - 16)$ completes $(R_T - 15)$ as it treats $\overline{V} = \emptyset$. Since there are no possible values for vector $\overline{x}$, the result is $\surd$.

4.2 Trace/Memory Semantics

This second semantics is given in Figure.

$$(T_{T/M} - 1) : \frac{\rho \in AtPEx}{(\rho, T, M) \rightarrow^{\rho} (\surd, T \cdot \rho, next(M))} \qquad (T_{T/M} - 1') : \frac{}{(\lambda, T, M) \rightarrow^{\lambda} (\surd, T, next(M))}$$

$$(T_{T/M} - 2) : \frac{(E, T, M) \rightarrow^{\rho} (E', T \cdot \rho, M')}{(GE \Rightarrow E, T, M) \rightarrow^{\rho} (E', T \cdot \rho, M')} \ \|GE[f_i \leftarrow M_i]\|$$

$$(T_{T/M} - 3) : \frac{(E_1, T, M) \rightarrow^{\rho} (E_1', T \cdot \rho, M')}{(E_1.E_2, T, M) \rightarrow^{\rho} (E_1'.E_2, T \cdot \rho, M')} \qquad (T_{T/M} - 4) : \frac{(E, T, M) \rightarrow^{\rho} (E', T \cdot \rho, M')}{(\surd.E, T, M) \rightarrow^{\rho} (E', T \cdot \rho, M')}$$

$$(T_{T/M} - 5) : \frac{(E_1, T, M) \rightarrow^{\rho} (E_1', T \cdot \rho, M')}{(E_1|E_2, T, M) \rightarrow^{\rho} (E_1', T \cdot \rho, M')} \qquad (T_{T/M} - 6) : \frac{(E_2, T, M) \rightarrow^{\rho} (E_2', T \cdot \rho, M')}{(E_1|E_2, T, M) \rightarrow^{\rho} (E_2', T \cdot \rho, M')}$$

$$(T_{T/M} - 7) : \frac{}{(E*, T, M) \rightarrow^{\lambda} (\surd, T, M)} \qquad (T_{T/M} - 8) : \frac{(E, T, M) \rightarrow^{\rho} (E', T \cdot \rho, M')}{(E*, T, M) \rightarrow^{\rho} (E' \cdot E*, T \cdot \rho, M')}$$

$$(T_{T/M} - 9) : \frac{}{(\surd|[\Delta]|\surd, T, M) \rightarrow^{\lambda} (\surd, T, M)}$$

$$(T_{T/M} - 10) : \frac{(E_1, T, M) \rightarrow^{\rho} (E_1', T \cdot \rho, M') \qquad (E_2, T, M) \rightarrow^{\rho} (E_2', T \cdot \rho, M)}{(E_1|[\Delta]|E_2, T, M) \rightarrow^{\rho} (E_1'|[\Delta]|E_2', T \cdot \rho, next(M))} in(\rho, \Delta)$$

$$(T_{T/M} - 11) : \frac{(E_1, T, M) \rightarrow^{\rho} (E_1', T \cdot \rho, M')}{(E_1|[\Delta]|E_2, T, M) \rightarrow^{\rho} (E_1'|[\Delta]|E_2, T \cdot \rho, next(M))} \neg in(\rho, \Delta)$$

$$(T_{T/M} - 12) : \frac{(E_2, T, M) \rightarrow^{\rho} (E_2', T \cdot \rho, M')}{(E_1|[\Delta]|E_2, T, M) \rightarrow^{\rho} (E_1|[\Delta]|E_2', T \cdot \rho, next(M))} \neg in(\rho, \Delta)$$

$$(T_{T/M} - 13) : \frac{(E[\overline{x} := \overline{t}], T, M) \rightarrow^{\rho} (E', T \cdot \rho, M')}{(|\overline{x} : \overline{V} : E, T, M) \rightarrow^{\rho} (E', T \cdot \rho, M')} \overline{t} \in \overline{V}$$

$$(T_{T/M} - 14) : \frac{(E[\overline{x} := \overline{t}], T, M) \rightarrow^{\rho} (E', T \cdot \rho, M')}{(P(\overline{t}), T, M) \rightarrow^{\rho} (E', T \cdot \rho, M')} P(\overline{t}) = E$$

$$(T_{T/M} - 15) : \frac{(E[\overline{x} := \overline{t}], T, M) \rightarrow^{\rho} (E', T \cdot \rho, M')}{(|[\Delta]|\overline{x} : \overline{V} : E, T, M) \rightarrow^{\rho} (E' \ |[\Delta]| \ (|[\Delta]|\overline{x} : \overline{V} \setminus \{\overline{t}\} : E), T \cdot \rho, M')}$$

$$(T_{T/M} - 16) : \frac{}{(|[\Delta]|\overline{x} : \emptyset : E, T, M) \rightarrow^{\rho} (\surd, T \cdot \rho, M')}$$

Fig. 3. $Sem_{T/M}$

The system's state w.r.t. $Sem_{T/M}$ is the tuple (E, T, M), where E is the expression of EB^3 that remains to be consumed, T is the current trace and $M(i)(\overline{x})$ is the variable that keeps the current valuation for attribute function f_i with parameter vector $\overline{x}$. The mapping M, called in the sequel *memory mapping*, is computed as follows:

$$M(i) : D_{t_1} \times \ldots D_{t_{k_i}} \longrightarrow D_{ret\_type_i},\ type(f_i) = (tr, t_1, \ldots, t_{k_i})$$
$$M_0(i)(\overline{x}) = \|Expr_{i,k}(\overline{x})\|, \text{ if } \|Cond_{i,k}\| \wedge T = [\,]$$
$$next(M)(i)(\overline{x}) = \|Expr_{i,k}(\overline{x})[f_j \leftarrow \text{if } j < i \text{ then } next(M)(j) \text{ else } M(j)]\|,$$
$$\text{if } \|Cond_{i,k}[f_j \leftarrow \text{if } j < i \text{ then } next(M)(j) \text{ else } M(j)]\|$$
$$\text{where } 1 \leq i \leq n,\ 1 \leq k \leq m_i$$

$M(i)$ refers to attribute function f_i. $Cond_{i,k}$ is chosen non-deterministically. See $AtFDef$ in Section 3. M_0 applies to the initial value of M, thus when it is $T = [\,]$. Considering restriction $11(*)$ for expressions f_j within $Expr_{i,k}$, we substitute every f_j, if $j < i$ with $next(M)(j)$ that has been calculated previously and with $M(j)$, if $j \geq i$. It is guaranteed that there is at least one $Cond_{i,k}$ that is satisfied on every run. Otherwise, the EB^3 specification is considered incomplete. Note the classic intepretations for Peano arithmetics, Set theory and Boolean logic suffice to compute $\|\cdot\|$ above as the trace has been eliminated.

In the sequel we denote $\mathcal{M}$ the set of memory mappings.

Rule $(T_{T/M} - 1)$ shows that $\rho \in AtPEx \cup \{\lambda\}$ is consumed when atomic process expression ρ takes place i.e. process ρ is reduced to $\surd$ process and the symbol ρ is appended to the end of current trace T. M's new value is calculated according to function *next*. For all the rules in $Sem_{T/M}$, T is reduced to $T \cdot \rho$. In Rule $(T_{T/M} - 2)$, $GE \in GExp$ is a guarded expression. If $E \in ExpDecl$, M are reduced to E', M' respectively and $\|GE[f_i \leftarrow M_i]\|$ is true (again the trace is eliminated in GE), then $GE \Rightarrow E$ is reduced to E' and memory M to M'. The rest of $Sem_{T/M}$ looks similar to Sem_T.

4.3 Memory Semantics

Memory Semantics derives from Trace/Memory Semantics by simple elimination of trace T from each state (E, T, M) in $Sem_{T/M}$ i.e. (E, M). Intuitively, this means that the information on the history of executions is kept in M, thus rendering the presence of trace T redundant. We do not give the rules for Sem_M in a separate figure due to space limitations.

5 Example revisited

We show how the EB^3 specification describing the library management system is evaluated w.r.t. Sem_T and Sem_M. We consider books $bId1$, $bId2$ and members $mId1$, $mId2$. We set $NbLoans = 2$. Figure 4 shows how expression *main* is modified for trace: $T_D = Lend(bId1, mId1).Register(mId2).Register(mId1)$.

$main$ (A) $\xrightarrow{*}$

$borrower(T,\ bId1) = \bot \rightarrow Discard(bId1)\ |||$
$borrower(T,\ bId2) = \bot \rightarrow Discard(bId2)\ |||$
$(\ |||\ mId : \text{MID} : member(mId)*\ )$ (B) $\xrightarrow{*}$

$borrower(T,\ bId1) = \bot \rightarrow Discard(bId1)\ |||$
$borrower(T,\ bId2) = \bot \rightarrow Discard(bId2)\ |||$
$(\ |||\ bId : \text{BID} : loan(mId1, bId)*\ ).\ Unregister(mId1).\ member(mId1) *\ |||$
$(\ |||\ bId : \text{BID} : loan(mId2, bId)*\ ).\ Unregister(mId1).\ member(mId2) *$ (C) $\xrightarrow{*}$

$borrower(T,\ bId1) = \bot \rightarrow Discard(bId1)\ |||$
$borrower(T,\ bId2) = \bot \rightarrow Discard(bId2)\ |||$
$(Return(bId1).\ loan(mId1, bId1) *\ |||\ loan(mId1, bId2)*).Unregister(mId1).\ member(mId1) *\ |||$
$(\ |||\ bId : \text{BID} : loan(mId2, bId)*\ ).\ Unregister(mId1).\ member(mId2) *$ (D)

Fig. 4. Execution

	T	$M = (bor[bId1], bor[bId2], nbL[mId1], nbL[mId2])$
A	$[\,]$	$(\bot, \bot, \bot, \bot)$
B	$Acq(bId2).Acq(bId1)$	$(\bot, \bot, \bot, \bot)$
C	$Reg(mId2).Reg(mId1).T_B$	$(\bot, \bot, 0, 0)$
D	$Lend(bId1,\ mId1).T_D$	$(mId1, \bot, 1, 0)$

Fig. 5. States

$Acquire(bId2).Acquire(bId1)$. Figure 5 shows how the intermediate states A, B, C and D w.r.t. Sem_T i.e. (E, T) and Sem_M i.e. (E, M) are modified. For given state, expression E is common for the two interpretations. We notice that Sem_T keeps track of the trace, an approach which is highly inefficient causing state explosion issues, whereas Sem_M gives a finite state system if the domains of its attribute functions are finite and bounded. Let's concentrate on transition $C \rightarrow D$. In order to check $borrower(T, bId1) = \bot \wedge nbLoans(T, mId1) < 2$, Sem_T evaluates $borrower(T, bId1)$ and $nbLoans(T, mId1)$ by traversing the trace and applying their corresponding attribute function formulas. On the contrary, Sem_M evaluates M based solely on the current memory and the event to be absorbed i.e. $bor_D[bId1] = next(bor_C[bId1]) = mId1$, $bor_D[bId2] = bor_C[bId2] = \bot$, $nbL_D[mId1] = nbL_C[mId1] + 1 = 1$ and $nbL_D[mId2] = 0$[1], for event $Lend(bId1, mId1)$ to be absorbed.

6 LTS Construction

As suggested in the previous sections, we consider finite labeled transition systems as interpretation models, which are particularly suitable for action-based description formalisms such as EB^3.

[1] We wrote *bor* and *nbL* instead of *borrower* and *nbLoans* for simplicity.

Formally, a labeled transition system TS is a triple $(S, \{\rightarrow^a\}_{a \in Act}, I)$, where:

1. S is a set of states;
2. $\rightarrow^a \subseteq S \times S$, for all $a \in Act$;
3. $I \subseteq$ S is a set of initial states.

One can compute the LTSs that simulate a given EB^3 program w.r.t. to the three semantics Sem_T, $Sem_{T/M}$ and Sem_M namely TS_T, $TS_{T/M}$ and TS_M. We demonstrate how to construct $TS_{T/M}$. Following a similar approach we can construct TS_T and TS_M. For a given expression E in EB^3, we construct the tuple:

$$TS_E = (S_E, \delta_E, I_E)$$

The construction is given by structural induction on the process algebraic part of the EB^3 expression E. We consider the initial trace empty i.e. $T = [\,]$ and we refer to the initial memory as M_0 defined upon the fixed body of attribute function definitions. In all constructions below, we put $I_E = \{(E, [\,], M_0)\}$.

$TS_{\surd}$ consists of all nodes $(\surd, [\,], M)$ with no transitions as no execution steps are possible for the terminated process.

$$S_{\surd} = \{(\surd, [\,], M) \mid M \in \mathcal{M}\} \qquad \delta_{\surd} = \emptyset$$

TS_ρ denotes the LTS describing an atomic process expression ρ. δ_ρ contains transitions describing the consumption of ρ on any memory.

$$\delta\rho = \{(\rho, [\,], M) \rightarrow^\rho (\surd, [\rho], next(M)) \mid M \in \mathcal{M}\}$$
$$S_\rho = \{(\rho, [\,], M) \mid M \in \mathcal{M}\} \bigcup \{(\surd, [\rho], next(M_0)) \mid M \in \mathcal{M}\}$$

TS_λ differs from T_ρ in that no process ρ is added to the final trace as T_λ stands for the idle transition.

$$\delta_\lambda = \{(\lambda, [\,], M) \rightarrow^\lambda (\surd, [\,], next(M)) \mid M \in \mathcal{M}\}$$
$$S_\lambda = \{(\lambda, [\,], M) \mid M \in \mathcal{M}\} \bigcup \{(\surd, [\,], next(M)) \mid M \in \mathcal{M}\}$$

For the construction of $TS_{E_1 \cdot E_2}$, we apply a compositional approach based on TS_{E_1} and TS_{E_2}:

$$S_{E_1 \cdot E_2} = \Big\{(E_1' \cdot E_2, T, M) \mid (E_1', T, M) \in S_{E_1}\Big\} \bigcup \bigcup_{(\surd, T_1, M_1) \in S_{E_1}} \Big\{(E_2', T_1 \cdot T_2, M) \mid (E_2', T_2, M) \in S_{E_2}^{M_1}\Big\}$$

$$\delta_{E_1 \cdot E_2} = \Big\{(E_1' \cdot E_2, T, M) \rightarrow (E_1'' \cdot E_2, T', M') \mid (E_1', T, M) \rightarrow^\rho (E_1'', T', M') \in \delta_{E_1}\Big\} \bigcup \bigcup_{(\surd, T_1, M_1) \in S_{E_1}} \{(E_2', T_1 \cdot T_2, M) \rightarrow^\rho (E_2'', T_1 \cdot T_2', M') \mid (E_2', T_2, M) \rightarrow^\rho (E_2'', T_2', M') \in \delta_{E_2}^{M_1}\}$$

States of the form $(E_1' \cdot E_2, T, M)$ belong to $S_{E_1 \cdot E_2}$ if (E_1', T, M) belong to S_{E_1}, because the completion of E_1 is the precondition for executing E_2. In this

manner, considering $(\surd, T_1, M_1) \in S_{E_1}$ we construct the remaining states for $S_{E_1 \cdot E_2}$, whose form is $(E_2', T_1 \cdot T_2, M)$. It is $(E_2', T_2, M) \in S_{E_2(M_1)}$ as E_1's earlier completion imposes initial trace T_1 and initial memory M_1 to S_{E_2}. As the effect of trace T_1 is included in memory M_1, we avoid writing $S_{E_2}^{T_1,M_1}$ and use $S_{E_2}^{M_1}$ instead.

The construction for $TS_{E_1|E_2}$ is the following:

$$S_{E_1|E_2} = S_{E_1} \setminus \{(E_1, [\,], M) \mid M \in \mathcal{M}\} \bigcup S_{E_2} \setminus \{(E_2, [\,], M) \mid M \in \mathcal{M}\} \bigcup \{(E_1|E_2, [\,], M) \mid M \in \mathcal{M}\}$$

$$\delta_{E_1|E_2} = \delta_{E_1}[E_1 \leftarrow E_1|E_2] \bigcup \delta_{E_2}[E_2 \leftarrow E_1|E_2]$$

$S_{E_1|E_2}$ takes the union of sets S_{E_1} and S_{E_2} replacing initial states $(E_1, [\,], M_0)$ and $(E_2, [\,], M_0)$ by $(E_1|E_2, [\,], M_0)$ as $(E_1|E_2, [\,], M_0)$ is the new initial state. For $\delta_{E_1|E_2}$, we take the union of $\delta_{E_i}[E_i \leftarrow E_1|E_2]$, i=1,2, which stands for δ_{E_i} where the source state E_i in transitions δ_{E_i} is replaced by the new initial state $E_1|E_2$.

For TS_{E*} we have:

$$TS_{E*} = lfp_F, \text{ where } F(TS_{E_x}) = TS_{E \,.\, E_x} \cup TS_{\lambda}$$

In order to obtain TS_{E*} we need to compute the least fix point of function $F : TS \rightarrow TS$ w.r.t. the *lattice* $\mathcal{TS} = (TS, \subseteq)$, where TS is the possibly infinite set of LTS that simulate EB^3 specifications w.r.t. $Sem_{T/M}$ and $\subseteq$ is the predicate that expresses inclusion. F in particular is the union of $TS_{E \cdot E_x}$ and TS_{λ}. The union expresses the possibility of executing E followed by the eventual re-execution of TS_{E_x} or skipping E completely.

The construction of $TS_{E_1|[\Delta]|E_2}$ is tricky: For given states in S_{E_i}, i=1,2 we apply the *merge* operation in Fig.6 to construct $S_{E_1|[\Delta]|E_2}$. $merge : ExpDecl \times ExpDecl \times tr \times tr \times Act \rightarrow tr$ merges T_1, T_2 while considering synchronisation on common atomic processes that are present in Δ. We clarify that *merge* is not a function as branches (1) and (2) may be executed in any order i.e. non-deterministically leading possibly to multiple images. For the rest branches, the order must be respected. The first branch refers to the successful termination of the merging operation, since the traces T_1 and T_2 are consumed, while the produced expressions E_1^a and E_2^a are syntactically equal to E_1 and E_2. Note also that by writing $(E_i^a, T_i^a) \rightarrow^{hd(T_1)} (E_i^b, T_i^b)$, the existence of a state (E_i^b, T_i^b) is implied for which this transition is possible. *merge* is also *partial* as its last branch giving $\perp_T$ corresponds to *blockings*, i.e. T_1 and T_2 cannot be merged while respecting synchronisation on Δ. $\perp_{\mathcal{TR}}$ is the *bottom value* for the lattice $\mathcal{TR} = (\mathcal{T}, \leq)$, where $\mathcal{T}$ is the possibly infinite set of traces for given EB^3 specifications and $\leq$ is the $prefix$ operation for traces. Predicate $in : AtPEx \times \{Act\} \rightarrow Bool$:

$$in(\rho, \Delta) = \exists \alpha \in \Delta,\ \rho = \alpha(a_1, \ldots, a_k)\ \wedge\ type(\alpha) = (t_1, \ldots, t_k)\ \wedge\ a_i \in D_{t_i}$$

$$S_{E_1|[\Delta]|E_2} = \{(E_1'|[\Delta]|E_2', T, M) \mid (E_i', T_i, M_i) \in S_{E_i},\ \text{i=1,2} \wedge \exists T, M : update(T1, T2, T, M)\}$$
$$\delta_{E_1|[\Delta]|E_2} = \{(E_1'|[\Delta]|E_2', T, M) \rightarrow^{\rho} (E_1''|[\Delta]|E_2'', T \cdot \rho, next(M)) \mid \text{i, j} \in \{1,2\}$$
$$(E_i', T_i, M_i) \rightarrow^{\rho} (E_i'', T_i', M_i') \in \delta_{E_i} \wedge in(\rho, \Delta) \wedge \exists T, M : update(T1, T2, T, M) \bigvee$$
$$(E_i', T_i, M_i) \rightarrow^{\rho} (E_i'', T_i', M_i') \in \delta_{E_i} \wedge E_j' = E_j'' \wedge i \neq j \wedge \neg in(\rho, \Delta) \wedge \exists T, M : update(T1, T2, T, M)\},$$
where $update(T1, T2, T, M) :=\ T \neq \bot_{\mathcal{TR}} \wedge T = merge(E_1, E_2, [\,], [\,], T_1, T_2) \wedge M = next'(T, M_0)$
and $next'(T, M) = \underbrace{next(\ldots next(}_{|T|-times} M)\ldots)$

$$merge(E_1^a, E_2^a, T_1^a, T_2^a, T_1, T_2) =$$
$$\begin{cases} [\,], & \text{if } T_1 = T_2 = [\,] \\ & \wedge\ E_1 = E_1^a \wedge E_2 = E_2^a \\ hd(T_1) :: merge(E_1^b, E_2^b, T_1^b, T_2^b, tl(T_1), tl(T_2)), & \text{if } hd(T_1) = hd(T_2) \wedge in(hd(T_1), \Delta) \\ & \bigwedge_{i=1,2} (E_i^a, T_i^a) \rightarrow^{hd(T_1)} (E_i^b, T_i^b) \\ hd(T_1) :: merge(E_1^b, E_2^a, T_1^b, T_2^a, tl(T_1), T_2), & \text{if } \neg in(hd(T_1), \Delta) \quad (1) \\ & \wedge\ (E_1^a, T_1^a) \rightarrow^{hd(T_1)} (E_1^b, T_1^b) \\ hd(T_1) :: merge(E_1^a, E_2^b, T_1^a, T_2^b, T_1, tl(T_2)), & \text{if } \neg in(hd(T_2), \Delta) \quad (2) \\ & \wedge\ (E_2^a, T_2^a) \rightarrow^{hd(T_2)} (E_2^b, T_2^b) \\ \bot_{\mathcal{TR}}, & \text{otherwise} \end{cases}$$

Fig. 6. Merge

returns true if ρ is constructed by an action name in Δ that respects ρ's *type*. We set $in(\rho, \emptyset) = true$ to cover the case $hd(T) = \emptyset$.

The construction of $\delta_{E_1|[\Delta]|E_2}$ considers the transitions corresponding to the three types of reduction namely rules $(T_{T/M} - 10, 11, 12)$.

For $TS_{|\overline{x}:\overline{V}:E}$, we extend the construction for $TS_{E_1|E_2}$ by considering all possible instantiations for E i.e. $E[\overline{x} := \overline{t}]$, $\overline{t} \in \overline{V}$:

$$S_{|\overline{x} : \overline{V} : E} = \bigcup_{\overline{t} \in \overline{V}} S_{E[\overline{x} := \overline{t}]}[E[\overline{x} := \overline{t}] \leftarrow |\overline{x} : \overline{V} : E]$$
$$\delta_{|\overline{x} : \overline{V} : E} = \bigcup_{\overline{t} \in \overline{V}} \delta_{E[\overline{x} := \overline{t}]}[E[\overline{x} := \overline{t}] \leftarrow |\overline{x} : \overline{V} : E]$$

Similarly, based on $T_{E_1|[\Delta]|E_2}$ and with compositional reasoning we obtain: $T_{|[\Delta]|\overline{x}:\overline{V}:E}$:

$$S_{|[\Delta]|\overline{x} : \overline{V} : E} = \bigcup_{\overline{t} \in \overline{V}} S_{exp(\overline{t})}[exp(\overline{t}) \leftarrow |[\Delta]|\overline{x} : \overline{V} : E]$$
$$\delta_{|[\Delta]|\overline{x} : \overline{V} : E} = \bigcup_{\overline{t} \in \overline{V}} \delta_{exp(\overline{t})}[exp(\overline{t}) \leftarrow |[\Delta]|\overline{x} : \overline{V} : E],$$
$$\text{where } exp(\overline{t}) = E[\overline{x} := \overline{t}]|[\Delta]|\left(|[\Delta]|\overline{x} : \overline{V} \setminus \overline{t} : E\right)$$

For $\Delta = \emptyset$, it is $S_{|[\Delta]|\overline{x}:\overline{V}:E} = \emptyset$ and $\delta_{|[\Delta]|\overline{x}:\overline{V}:E} = \emptyset$.

To obtain $TS_{GE \to E}$, we consider the case in which GE evaluates to true i.e. $\|GE\|$. For the rest, we apply similar reasoning as in $TS_{E_1|E_2}$. The case $\|GE\| = false$ leads to trivial LTS with node $(GE \to E, [\,], M_0)$ and no transitions.

$$S_{GE \to E} = \begin{cases} \{(GE \to E, [\,], M_0)\} \bigcup S_E \setminus \{(E, [\,], M_0)\}, & \text{if } \|GE\| = true \\ \{(GE \to E, [\,], M_0)\}, & \text{otherwise} \end{cases}$$

$$\delta_{GE \to E} = \begin{cases} \delta_E[(E, [\,], M_0) \,/\, (GE \to E, [\,], M_0)], & \text{if } \|GE\| = true \\ \emptyset, & \text{otherwise} \end{cases}$$

7 Bisimulation Equivalence of Sem_T, $Sem_{T/M}$ and $Sem_{T/M}$

Bisimulation is a fundamental notion in the framework of concurrent processes and transition systems. According to the standard definition, a system is bisimilar to another system if the former can mimic the behaviour of the latter and vice-versa. In this sense, the associated systems are considered indistinguishable. Given two labeled transition systems $TS_i = (S_i, \{\to^a\}_{a \in Act}, I_i)$, where $i = 1, 2$ and a relation $R \subseteq S_1 \times S_2$, systems TS_i are said to be bisimilar or equivalent w.r.t. bisimulation if and only if

1. $\forall\, s_1 \in I_1\ \exists s_2 \in I_2$ such that $(s_1, s_2) \in R$
2. $\forall\, s_2 \in I_2\ \exists s_1 \in I_1$ such that $(s_1, s_2) \in R$
3. $\forall (s_1, s_2) \in R$:
   (a) if $s_1 \to^a s_1'$ then $\exists s_2' \in S_2$ such that $s_2 \to^a s_2'$ and $(s_1', s_2') \in R$
   (b) if $s_2 \to^a s_2'$ then $\exists s_1' \in S_1$ such that $s_1 \to^a s_1'$ and $(s_1', s_2') \in R$

Theorem 1. *TS_T and $TS_{T/M}$ are bisimilar.*

For the proof, we consider relation $R = \{< (E, T, M), (E, T) > \mid (E, T, M) \in S_{T/M} \wedge (E, T) \in S_T\}$, where $S_{T/M}$ and S_T are costructed as shown in the previous section. We deduce that conditions expressed in lines 1 and 2 of the previous definition are trivially satisfied, as $< (E_0, [\,], M_0), (E_0, [\,]) > \in R$. To complete the proof, we demonstrate that for any $< (E, T, M), (E, T) > \in R$ and transition in $TS_{T/M}$ $(E, T, M) \to^\rho (E', T \cdot \rho, M')$, we obtain another transition in TS_T of the form $(E, T) \to^\rho (E', T \cdot \rho)$ and vice-versa. We consider the possible forms of E in $(E, T, M) \to^\rho (E', T \cdot \rho, M')$. We show the proof for some cases.

For $(T_{T/M} - 1)$, we get $(\rho, T, M) \to^\rho (\surd, T \cdot \rho, next(M))$, where $E \equiv \rho$ and $E' \equiv \surd$. By eliminating M and $next(M)$, we notice that $(\rho, T) \to^\rho (\surd, T \cdot \rho)$ is a valid reduction rule for Sem_T and thus 3a) is demonstrated. For 3b), we imagine rule $(\rho, T) \to^\rho (\surd, T \cdot \rho)$. We can prove with induction on E that every state in TS_M is of the form:

$$(E, T, next'(T, M_0)), \text{ where } next'(T, M) = \underbrace{next(\ldots next(}_{|T| - times} M) \ldots)$$

where M_0 is the initial memory and $|T|$ is the length of T. As a result, there exists a transition

$$(\rho, T, next'(T, M_0)) \rightarrow^{\rho} (\surd, T \cdot \rho, next'(T \cdot \rho, M_0))$$

which proves $(T_{T/M} - 1)$ by replacing $M = next'(T, M_0)$ and $next'(T \cdot \rho, M_0) = next\,(next'(T, M_0))$.

For $(T_{T/M} - 3)$, we have $(E_1.E_2, T, M) \rightarrow^{\rho} (E_1'.E_2, T \cdot \rho, M')$, that assumes $(E_1, T, M) \rightarrow^{\rho} (E_1', T \cdot \rho, M')$. By induction hypothesis, $(E_1, T) \rightarrow^{\rho} (E_1', T \cdot \rho)$ in Sem_T and by $(R_T - 3)$, we get $(E_1.E_2, T) \rightarrow^{\rho} (E_1'.E_2, T \cdot \rho)$. Vice-versa, by virtue of $(R_T - 3)$ a transition $(E_1.E_2, T) \rightarrow^{\rho} (E_1'.E_2, T \cdot \rho)$ gives $(E_1, T) \rightarrow^{\rho} (E_1', T \cdot \rho)$. Using the induction hypothesis, $(E_1, T, M) \rightarrow^{\rho} (E_1', T \cdot \rho, M')$. Finally, by $(T_{T/M} - 3)$ we obtain $(E_1.E_2, T, M) \rightarrow^{\rho} (E_1'.E_2, T \cdot \rho, M')$.

For $(T_{T/M} - 5)$, we have $(E_1|E_2, T, M) \rightarrow^{\rho} (E_1'|E_2, T \cdot \rho, M')$, that assumes $(E_1, T, M) \rightarrow^{\rho} (E_1', T \cdot \rho, M')$. By induction hypothesis, we have $(E_1, T) \rightarrow^{\rho} (E_1', T \cdot \rho)$ in Sem_T and by $(R_T - 5)$, we get $(E_1|E_2, T) \rightarrow^{\rho} (E_1', T \cdot \rho)$. Vice-versa, a transition $(E_1|E_2, T) \rightarrow^{\rho} (E_1', T \cdot \rho)$ gives by virtue of $(R_T - 5)$ $(E_1, T) \rightarrow^{\rho} (E_1', T \cdot \rho)$. Using the induction hypothesis, $(E_1, T, M) \rightarrow^{\rho} (E_1', T \cdot \rho, M')$. Finally, by $(T_{T/M} - 5)$ we obtain $(E_1|E_2, T, M) \rightarrow^{\rho} (E_1', T \cdot \rho, M')$.

For $(T_{T/M} - 13)$, we assume $(|\overline{x} : \overline{V} : E, T, M) \rightarrow^{\rho} (E', T \cdot \rho, M')$, which due to $(T_{T/M} - 13)$, implies $(E[x := t], T, M) \rightarrow^{\rho} (E', T \cdot \rho, M')$. By induction, we get the transition $(E[\overline{x} := \overline{t}], T) \rightarrow^{\rho} (E', T \cdot \rho)$ in Sem_T, which with further assumption $\overline{t} \in \overline{V}$ gives $(|\overline{x} : \overline{V} : E, T) \rightarrow^{\rho} (E', T \cdot \rho)$. The inverse is founded as follows: we assume transition $(|\overline{x} : \overline{V} : E, T) \rightarrow^{\rho} (E', T \cdot \rho)$. $(R_T - 13)$ implies $(E[\overline{x} := \overline{t}], T) \rightarrow^{\rho} (E', T \cdot \rho)$. Induction principle then gives $(E[x := t], T, M) \rightarrow^{\rho} (E', T \cdot \rho, M')$ and rule $(T_{T/M} - 13)$ $(|\overline{x} : \overline{V} : E, T, M) \rightarrow^{\rho} (E', T \cdot \rho, M')$.

Theorem 2. *$TS_{T/M}$ and TS_M are bisimilar.*

Proof. The proof is straightforward, because the effect of the trace on the attribute functions and the program execution is coded in memory M. Hence, intuitively the trace is redundant.

Corollary 1. *TS_T and TS_M are bisimilar.*

Proof. Combining the two Theorems and with transitivity, we prove the lemma.

8 Transformation of EB^3 models to Petri Nets

In the following sections we present the technique to cast EB^3 expressions to Petri Nets:

8.1 Atomic expression: $e \equiv E$

In the simplest case, our expression is atomic. We therefore need an entry place (e), an exit place (x) and the transition t_E corresponding to event E. The relation function will associate e to t_E and t_E to x:

$$P_E \equiv (\{e, x\},\ \{t_E\},\ \{(e, t_E),\ (t_E, x)\},\ e,\ x\}$$

When running the Petri Net, we obtain $m_0 \equiv \{e\}$. As t_E is the only transition satisfying $\triangleleft t \subseteq m_0 \leftrightarrow \{e\} \subseteq \{e\}$, it is actually fired, leading to $m_F = \{x\}$.

8.2 Composition: $e \equiv \alpha.\beta$

In this case, we have a composition of two expressions α and β. The induction hypothesis guarantees the existence of their corresponding Petri Nets shown on the figure above. Note that the boxes in the graph do not refer to an atomic expression, as α and β can not only be atomic. In other words, they refer to complex Petri Nets. Entry point e_α and exit point x_α are indicated explicitely to distinguish them from other places within α. The arc associating e_α and α is basically a generalised connection between e_α and other transition nodes within α. That is not the case for the atomic expression seen in the previous section, where the box refers exactly to the transition. Same remarks can be made for expression β. Given the Petri Nets, $P_\alpha \equiv (S_\alpha, T_\alpha, R_\alpha, e_\alpha, x_\alpha)$ and $P_\beta \equiv (e_\beta, x_\beta, S_\beta, T_\beta, R_\beta, e_\beta, x_\beta)$, we construct the corresponding Petri Net for the composition:

$$P_{\alpha.\beta} \equiv (S_\alpha \bigcup S_\beta \setminus \{e_\beta\}, T_\alpha \bigcup T_\beta, \\ R_\alpha \bigcup R_\beta \setminus \{(e_\beta,\ p) \mid p \in e_\beta \triangleright\} \bigcup \{(x_\alpha,\ p) \mid p \in e_\beta \triangleright\},\ e_\alpha,\ x_\beta)$$

Entry point e_β has been merged with x_α. As a result, tuples containing e_β as their starting point, i.e. $\{(e_\beta,\ p) \mid p \in e_\beta \triangleright\}$, should be replaced by the tuples $\{(x_\alpha,\ p) \mid p \in e_\beta \triangleright\}$. e_α and x_β serve as the entry and exit point accordingly.

Proper execution is guaranteed and the proof follows:

$$\forall\, p \in S_\alpha \setminus \{x_\alpha\} \quad \neg\exists\, t \in T_\beta \quad \Rightarrow \quad (p,\ t) \in R_{\alpha.\beta}$$

$$\forall\, p \in S_\beta \bigcup \{x_\alpha\} \quad \neg\exists\, t \in T_\alpha \quad \Rightarrow \quad (p,\ t) \in R_{\alpha.\beta}$$

As there are no tuples connecting places of α with transitions within β, α will have to be executed first. As soon as x_α is activated, β will take its turn and finally terminate.

8.3 Kleene closure: $e \equiv \alpha*$

$$P_{a*} \equiv (S_\alpha,\ T_\alpha \bigcup \{\epsilon\},\ R_\alpha \bigcup \{(p, e_\alpha) \mid p \in x_\alpha \triangleright, e_\alpha, x_\alpha\}$$

Given the Petri Net corresponding to α, we construct the Petri Net for α*. Transition ϵ is taken when α is executed zero times. Box α refers to the whole Petri Net for expression α as seen in previous cases. Every time Petri Net α completes its task, e_α is reactivated, thus permitting a new choice between PN α and the exit transition ϵ.

8.4 Choice: $\epsilon \equiv \alpha \mid \beta$

$$P_{a \mid b} \equiv (e_\alpha,\ x_\alpha,\ S_\alpha \bigcup S_\beta \setminus \{e_\beta, x_\beta\},\ T_\alpha \bigcup T_\beta\ ,\ R_\alpha \bigcup R_\beta \setminus \{(p,\ x_\beta)\ \mid\ p\ \in\ \triangleleft x_\beta\} \setminus \{(e_\beta,\ p)\ \mid\ p\ \in\ e_\beta \triangleright\} \bigcup \{(e_\alpha,\ p)\ \mid\ p\ \in\ e_\beta \triangleright\})$$

Given the PNs corresponding to α and β, we construct the PN for the choice $\alpha \mid \beta$. Entry places e_α and e_β are unified, as is the case for exit places x_α and x_β. As a result, all connections from e_β to transitions within the PN of β should be modified by replacing e_β to e_α. Similar measures are taken for x_α.

8.5 Parallel Interleaving: $e \equiv \alpha \mid\mid\mid \beta$

The PN corresponding to the parallel composition operation considers an additional transition t_1, whose execution signals the beginning of the parallel composition. Transition t_2 assures the completion of both expressions α and β. The introduction of places e and x completes the definition. It is important to see that in case that expressions α and β contain common atomic subexpressions, they should be renamed to avoid undesirable connections between places of α and transitions of β and vice versa, which is the desirable behaviour for the parallel composition operation.

$$P_{a \parallel b} \equiv (e,\ x,\ S_\alpha \bigcup S_\beta \bigcup \{e,\ x\},\ T_\alpha \bigcup T_\beta \bigcup \{t_1,\ t_2\},\ R_\alpha \bigcup R_\beta \bigcup \{(e,\ t_1)\ ,\ (t_1,\ e_\alpha)\ ,\ (t_1,\ e_\beta)\ ,\ (x_\alpha,\ t_1),\ (x_\beta,\ t_1),\ (t_1,\ x))\ )$$

8.6 Parallel Composition: $e \equiv \alpha \parallel \beta$

The construction of the Petri Net corresponding to the parallel composition of α and β involves combining their syntactically identical atomic subexpressions. A formal presentation is rather tricky and we consider to add it to the next full paper.

8.7 Existential quantification: $\mid x : T : e$, $\mid\mid\mid x : T : e$ and $\parallel\ x : T : e$

These parts of EB^3's syntax are unfolded according to "x"'s possible values as a means to be transformed to the corresponding base cases seen above.

8.8 Conditional execution: $cond \implies e$

The treatment of this case does not affect directly the Petri Net we construct. Basically the expression corresponding to the condition is placed as an attribute next to the input place so as to be implanted later to the Nu-SMV code. Moreover, the way we model attribute functions does not affect the related Petri Net. The way they appear on the Nu-SMV code will be discussed in the following section.

9 Nu-SMV Model Checker

Nu-SMV is a symbolic model checker for the temporal logic Computational Tree Logic (CTL). The systems to be verified are finite state transition systems described in SMVs own input language. An input consists of two parts. One part is a state machine defined in terms of variables, initial values for the variables, and a description of the conditions under which variables may change value and what values they may take. The other part is a set of temporal logic clauses, in CTL, expressing invariant conditions and temporal logic constraints on possible execution paths. Conceptually, Nu-SMV visits all reachable states and verifies that the invariants and constraints are always satisfied.

10 Transformation of Petri-Nets to Nu-SMV code

For a given Petri Net $PN = (P,\ T,\ R,\ e,\ x)$ corresponding to a EB^3 model, we will show how our compiler produces the equivalent Nu-SMV code. For coding our system to Nu-SMV we will need as many boolean variables as are the places that we have created in our Petri Net. We will also need to define possibly multi-dimensional arrays corresponding to the attribute functions appearing in the initial EB^3 specification. Each dimension in the final code refers to each parameter in the initial code and the return type for the array coincides with the return type for the attribute function. For the example of the atttribute function *borrower* written above, we will need the definition VAR borrower: array[1..card(bookID)] of 1..card(memberID)

For the initialisation part, we will write a formula that sets the entry place to TRUE and the rest places to FALSE. The initial values for attribute functions are given by the trace's pattern matching and in particular are given by the command corresponding to an empty trace. The initial value for function *borrower* is therefore the bottom value.

The transition specification part is a disjunction between scenarios that enable transitions in the initial EB^3 specification. The number of operands to these disjunctions is equal to the number of possible transitions in our model. We therefore need to find in each such scenario the conjunction of places/predecessors to each transition, meaning the places that are fundamental to taking the transition. In the event that a transition's predecessor has an extra condition due to the existence of an attribute function in the initial model, this has to be added to the conjunction and we denote it by cond(p) in the code below. We then add to the same space the code that renders the successors of the transition in question true and its predecessors false.

What follows is a formal presentation of the translation:

$$
\begin{array}{l}
MODULE\ main \\
VAR \\
\quad \forall\, p\ \in\ P,\ p : boolean; \\
INIT \\
\quad (\bigwedge_{p\ \in\ P\ \setminus\ e} p\ =\ FALSE)\ \ \wedge\ \ (e\ =\ TRUE) \\
TRANS \\
(\ \bigvee_{t\ \in\ T} (\ (\wedge_{p\ \in\ {}^{\triangleleft}t} p\ \wedge\ cond(p))\ \bigwedge\ (\wedge_{p\ \in\ {}^{\triangleleft}t} next(p) = FALSE) \\
\qquad \bigwedge\ (\wedge_{p\ \in\ t^{\triangleright}} next(p) = TRUE)\ )\ )\ \ \bigwedge\ (\wedge_{\forall\ attr} code(attr))
\end{array}
$$

In the presence of attribute functions the code is augmented as shown above with the translation of the attribute function into Nu-SMV. This is denoted by $code(attr)$.

It remains to add to the so far constructed formula the code that computes attribute functions. As seen above the pattern matching on possible events of the trace is translated to finding the conjunction of predecessors for the given event/transition. The body that is executed whenever a particular pattern is matched is translated directly to Nu-SMV code. This is justified by the fact that we except to have relatively simple definitions as a attribute function command body or simply a recursive call to the same attribute function. We have restricted our research scope to attribute function with a similar form to the function *borrower* below in order to avoid potential state explosion problems.

For function *borrower*, the first line in the match pattern will be translated to the boolean formula that assures that no transition in the model can be activated. For the second line we will need the preconditions for all the possible transitions *Lend* that match the pattern. Note that the possible transitions *Lend* in our Petri Net matching line 2 may be more than one depending on the different parameters matching the pattern. All these preconditions are given in disjunction. We give the *borrower*'s initial and final code to clarify the translation process for attribute functions:

$$
\begin{array}{l}
borrower\ (t:\ trace,\ bId:\ bookID):\ memberID\ = \\
match(head(t)) \\
\quad []\ =>\ \bot \\
\quad Lend(bId : bookID, mId : memberID)\ =>\ mId \\
\quad Return(bId)\ =>\ \bot \\
\quad other\ =>\ borrower(tail(t),\ bId)
\end{array}
$$

$$
\begin{array}{l}
code(borrower)\ \equiv\ \bigwedge_{bID\ \in\ bookID} \\
\quad next(borrower[bID]) = \\
\quad case \\
\quad\quad !\ (\bigvee_{t\ \in\ T}\ \wedge_{p\ \in\ \triangleleft\ t} p)\ : \bot; \\
\quad\quad (\bigvee_{mId\ \in\ memberID}\ (\wedge_{p\ \in\ \triangleleft\ Lend(bID,\ mID)}\ p)) :\ mID \\
\quad\quad (\wedge_{p\ \in\ \triangleleft\ Return(bID)}\ p) : \bot \\
\quad\quad TRUE : borrower[bID]; \\
\quad esac
\end{array}
$$

11 Case study

To illustrate the transformation process, we will apply the developed ideas to a library management system. The motivation behind this transformation is to verify security and liveness properties on the initial model. This is carried out by casting directly these properties to the Nu-SMV model checking script language. The properties in question have been written in CTL and the results have been collected to evaluate the efficiency of our verification process.

The system has to manage book loans to members. A book is acquired by the library; it can be discarded, but only if it is not lent. A member must join the library in order to borrow a book and he can relinquish library membership only when all his loans are returned or transferred. A book can be lent by only one member at once.

The modeling of this system in EB^3 specification language is presented in the appendix.

The special type void is used to denote an action with no input-output rule; the output of such an action is always ok. Some input parameters can be instantiated by a default value, NULL, that denotes undefinedness.

A EB^3 specification usually contains the signature of the basic atomic events and the basic data set used throughout the script. In this case we have a $bookID$ data set that refers to a set of books managed by the library, a $memberID$ set for possible members of the library, a *btitle* set, which denotes a limited number of possible titles given to books throughout the system's execution and a *loantype* that signifies the type of membership to which somebody is engaged. In this case, we have considered, two kinds of membership: the classical and the permanent. One can easily notice the flexibility EB^3's syntax offers to the user in terms of expressivity. The datasets can be modified arbitrarily with no effect on the rest of the code. In our case study, we experimented on the number of books and members that can participate in the library.

In the rest of the script, we describe the possible scenarios for a system like this. The main function in the script is a synchronisation operation between the various *book* and *member* processes. Process *book* depicts the need for a book to be acquired in order to be engaged in a *loan* process and then possibly be discarded by the library. Process *loan* describes the fact that a *Lend* operation is always followed by a *Return* operation(process $loanNominal$) and illustrates the conditions under which a loan can occur i.e. the attribute function $nbLoans$, denoting the number of books borrowed by a member, is less that a fixed upper bound $NbLoans$. In the absence of this condition, a dummy event λ occurs. As for process *member*, its script gives the cycle for someone to become member and borrow some books.

We proceed with the verification of this library management system. We are interested in liveness properties i.e. the property that guarantees that a given event e.g. *Lend* can occur. This can be expressed intuitively in the form of a CTL property: $EF(Lend(b1,\ m1))$ i.e. there exists a path, where member m1 can borrow book b1. This is transcribed to:

$$EF(\wedge_{p \in \triangleleft\ Lend(b1,\ m1)} p)$$

In this case if the precondition for this transition is fulfilled, we can actually take the "Lend" operation for book b1 and member m1.

We will also verify the safety properties:

1. We cannot have a Lend operation for a book namely b1 that has been discarded, unless it has been reacquired.
   $AG(Discard(b1) \rightarrow A[!Lend(b1,\ m2,\ per) \vee Acquire(b1,\ \_)]$. _ denotes existential quantification. Or equivalently in Nu-SMV code it is written:

$$AG(\wedge_{p \in \triangleleft\ Discard(b1)} p \rightarrow AX(\wedge_{p \in\ Discard(b1) \triangleright} p \rightarrow AG[!\bigvee_{tID \in\ titleID} (\wedge_{p \in \triangleleft\ Acquire(b1,tID)} p)) \vee A(!\wedge_{p \in \triangleleft\ Lend(b1,\ m2,\ per)} \sqcup \bigvee_{tID \in\ titleID} (\wedge_{p \in \triangleleft\ Acquire(b1,tID)} p]))))$$

2. Dually, a member namely m2 that has been unregistered cannot borrow a book namely b1, unless he has been registered.
$AG(Unregister(m2) \rightarrow A[!Lend(b1, m2, per) \vee Register(m2)]$.

$$AG(\wedge_{p \in \triangleleft Unregister(m2)} p \rightarrow AX(\wedge_{p \in Unregister(m2) \triangleright} p \rightarrow AG[! \wedge_{p \in \triangleleft Register(m2)} p)) \bigvee A(! \wedge_{p \in \triangleleft Lend(b1, m2, per)} \bigsqcup \wedge_{p \in \triangleleft Register(m2)} p])))$$

And the results in seconds are presented in this table:

Books, Members	Load Time	Property1	Property2	Property3
1 x 1	0.0	0.0	0.0	0.0
2 x 2	0,2	0.1	0.0	0.1
3 x 3	1.3	1.4	0.5	1.1
4 x 4	6.2	273	170.2	385.4
5 x 5	23	4069	1478	2078

The machine used was an Intel(R) Core(TM) i7 CPU 880 @ 3.07GHz. It is obvious for members and books, whose number is greater than 5, verification of the system becomes a strenuous task.

12 Conclusion

In this paper we defined three operational semantics for EB^3: Sem_T, $Sem_{T/M}$ and Sem_M, which we proved strongly bisimilar to each other. The principal result is that Sem_M, in which attribute functions are computed during program evolution and stored into program memory can replace Sem_T, which keeps track of the current trace. For future work, we are interested in casting Sem_M and Sem_T to Coq [2] as a means to automate their proof of equivalence w.r.t. bisimulation. A mechanical proof of equivalence would validate the proof sketch presented in the previous section. Furthermore, we demontrated that Sem_M is a finitary operational semantics and as a result can be better handled by model-checking tools making the verification of EB^3 specifications more doable. We opt for formalizing the automatic translation of EB^3 specifications into LOTOS-NT and benefit from the $CADP$ toolbox for the verification of EB^3 specifications.

Until now, we have considered a number of possible representations for a given EB^3 system specification all leading to buffer overflow and memory starvation issues for really big inputs. All of these techniques required a direct translation to a Nu-SMV model. Although Nu-SMV benefits from numerous efficient symbolic model checking techniques to verify temporal logic formulas even on complex models, our model turned out to be hopelessly immense to be dealt with. Another possibility could have been a goal-oriented abstraction of the Nu-SMV model limiting our effort to the formulas needing to be checked every time. This approach is still under consideration for future research.

13 Appendix

```
Acquire(bId : bookID, bTitle : titleID) : void
Discard(bId : bookID) : void
Register(mID : memberID) : void
Unregister(mID : memberID) : void
Lend(bID : bookID, mID : memberID) : void
Return(bID : bookID) : void
bookID = (b1, b2), memberID = (m1, m2), btitle = (t1, t2), loantype = (per, clas)

loanNominal (mId, bId) = (| type : loantype : Lend(bId, mId, type)). Return(bId)
loanController(mId, bId) =
   nbLoans(trace, mId) < NbLoans → (| title : loantype : Lend(bId, mId, title)) | → λ :
loan(mId, bId) = loanNominal (mId, bId) ‖ (loanController (mId, bId)∗) :
book(bId) = (| bt : btitle : Acquire(bId, bt)).
   ((| mId : memberID : loan(mId, bId))∗). Discard(bId) :
member(mId) = Register(mId). (||| bId : bookID : loan(mId, bId)∗). Unregister(mId) :
main = (||| bId : bookID : book(bId)∗) ‖ (||| mId : memberID : member(mId)∗)

        nbLoans(t : trace, mId : memberID) : integer =
        match(head(t))
          Register(mId, _) : 0
          Lend(_, mId, _) : nbLoans(mID) + 1
          Return(bId) : mId ≡ borrower(t, bId) then nbLoans(tail(t), mId) − 1
          Unregister(mId) : ⊥
          other :=> nbLoans(tail(t), mId);
        borrower (t : trace, bId : bookID) : memberID =
        match(head(t))
          [] => ⊥
          Lend(bId : bookID, mId : memberID) => mId
          Return(bId) => ⊥
          other => borrower(tail(t), bId)
```

References

1. Jean-Raymond Abrial. *The B-book - assigning programs to meanings.* Cambridge University Press, 2005.
2. Yves Bertot and Pierre Castéran. Interactive theorem proving and program development. coq'art: The calculus of inductive constructions, 2004.
3. Tomasso Bolognesi and Ed Brinksma. Introduction to the iso specification language lotos. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.
4. Romain Chossart. Évaluation d'outils de vérification pour les spécifications de systèmes d'information. Master's thesis, Université de Sherbrooke, Avril 2010. Directeur: Marc Frappier, Codirecteur: Benoît Fraikin.
5. Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. pages 359–364. Springer, 2002.
6. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.
7. ClearSy. *Atelier B.* http://www.atelierb.societe.com/.
8. Benoît Fraikin. *Interprétation efficace d' Expression de Processus EB3.* PhD thesis, Janvier 2006.
9. Benoît Fraikin, Marc Frappier, and R'egine Laleau. State-based versus event-based specifications for information systems: a comparison of b and eb 3. *Software and Systems Modeling*, 4(3):236–257, July 2005.

10. Marc Frappier, Benoît Fraikin, Romain Chossart, Raphaël Chane-Yack-Fa, and Mohammed Ouenzar. Comparison of model checking tools for information systems. In *Proceedings of the 12th international conference on Formal engineering methods and software engineering*, ICFEM'10, pages 581–596, Berlin, Heidelberg, 2010. Springer-Verlag.
11. Marc Frappier and Richard St.-Denis. Eb 3: an entity-based black-box specification method for information systems. *Software and System Modeling*, pages 134–149, 2003.
12. Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. Cadp 2010: A toolbox for the construction and analysis of distributed processes. In *TACAS*, pages 372–387, 2011.
13. Frédéric Gervais. *Combinaison de spécifications formelles pour la modélisation des systèmes d'information.* PhD thesis, December 2006.
14. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
15. Gerard J. Holzmann. The model checker spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.
16. R. Milner. *Communication and Concurrency.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
17. C. A. Petri. Concurrency. In *Advanced Course: Net Theory and Applications'75*, pages 251–260, 1975.

Annexe A

Les spécifications B complètes issues du modèle fonctionnel de l'exemple du SI médical

MACHINE

Functional_Model

SETS

$PATIENT = \{Bob, Paul\}$;

$MEDICALRECORD = \{m1, m2\}$;

$DEPARTMENT = \{D1, D2\}$;

$MEDICALEMPLOYEE = \{Martin, Fred, Jack, Jean\}$;

$STR = \{RADIOLOGY, CARDIOLOGY, FRED, MARTIN, JACK, JEAN\}$

ABSTRACT_VARIABLES

$Patient$

, $MedicalRecord$

, $Department$

, $MedicalEmployee$

, $patientMedicalRecord$

, $patientDepartment$

, $MedicalEmployeeDepartment$

, $medicalRecordCreator$

, $Patient\_\_SSN$

, $Department\_Name$

, $MedicalEmployee\_Name$

, $MedicalRecord\_\_Data$

, $MedicalRecord\_\_IsValidated$

INVARIANT

$Patient \subseteq PATIENT$

$\wedge\ MedicalRecord \subseteq MEDICALRECORD$

$\wedge\ Department \subseteq DEPARTMENT$

$\wedge\ MedicalEmployee \subseteq MEDICALEMPLOYEE$

$\wedge\ patientMedicalRecord \in Patient \rightarrowtail\!\!\!\!\!\!\rightarrow MedicalRecord$

$\wedge\ patientDepartment \in Patient \rightarrow Department$

$\wedge$ $MedicalEmployeeDepartment \in MedicalEmployee \leftrightarrow Department$
$\wedge$ $medicalRecordCreator \in MedicalRecord \rightarrow MedicalEmployee$
$\wedge$ $Patient\_\_SSN \in Patient \rightarrowtail \mathcal{Z}$
$\wedge$ $MedicalRecord\_\_Data \in MedicalRecord \nrightarrow STR$
$\wedge$ $MedicalRecord\_\_IsValidated \in MedicalRecord \rightarrow \mathbf{BOOL}$
$\wedge$ $Department\_Name \in Department \rightarrow STR$
$\wedge$ $MedicalEmployee\_Name \in MedicalEmployee \rightarrow STR$
INITIALISATION
$Patient := \{Bob, Paul\}$
|| $MedicalRecord := \emptyset$
|| $Department := \{D1, D2\}$
|| $MedicalEmployee := \{Martin, Fred, Jack, Jean\}$
|| $patientMedicalRecord := \emptyset$
|| $patientDepartment := \{(Bob \mapsto D1), (Paul \mapsto D2)\}$
|| $MedicalEmployeeDepartment := \{(Fred \mapsto D1), (Jack \mapsto D2), (Martin \mapsto D1), (Jean \mapsto D2)\}$
|| $medicalRecordCreator := \emptyset$
|| $Patient\_\_SSN := \{(Bob \mapsto 1), (Paul \mapsto 2)\}$
|| $MedicalRecord\_\_Data := \emptyset$
|| $MedicalRecord\_\_IsValidated := \emptyset$
|| $Department\_Name := \{(D1 \mapsto RADIOLOGY), (D2 \mapsto CARDIOLOGY)\}$
|| $MedicalEmployee\_Name := \{(Fred \mapsto FRED), (Jack \mapsto JACK), (Martin \mapsto MARTIN), (Jean \mapsto JEAN)\}$
OPERATIONS

MedicalRecord_NEW($Instance$,$patientMedicalRecord\_\_patientValue$, $medicalRecordCreator\_creatorValue$)=
PRE

$Instance \in MEDICALRECORD \quad \wedge \quad Instance \notin MedicalRecord$
$\wedge$ $patientMedicalRecord\_\_patientValue \in Patient \wedge medicalRecordCreator\_creatorValue \in MedicalEmployee$

$\wedge$ $patientMedicalRecord\_\_patientValue \in \mathbf{dom}(patientDepartment)$
$\wedge$ $patientMedicalRecord\_\_patientValue \notin \mathbf{dom}(patientMedicalRecord)$
THEN

$MedicalRecord := MedicalRecord \cup \{Instance\}$

|| $medicalRecordCreator := medicalRecordCreator \cup \{(Instance \mapsto medicalRecordCreator\_creatorValue)\}$
|| $patientMedicalRecord := patientMedicalRecord \cup \{(patientMedicalRecord\_\_patientValue \mapsto Instance)\}$

|| **MedicalRecord__IsValidated**($Instance$) := **FALSE**
END;

MedicalRecord_Free($Instance$)=
PRE
$Instance \in MEDICALRECORD \quad \wedge \quad Instance \in MedicalRecord$

$\wedge$ *MedicalRecord__IsValidated*(*Instance*) = **FALSE**
THEN
MedicalRecord := *MedicalRecord* - {*Instance*}

|| *MedicalRecord__Data* := {*Instance*} $\lhd$ *MedicalRecord__Data*
|| *MedicalRecord__IsValidated* := {*Instance*} $\lhd$ *MedicalRecord__IsValidated*

|| *patientMedicalRecord* := *patientMedicalRecord* $\rhd$ {*Instance*}
|| *medicalRecordCreator* := {*Instance*} $\lhd$ *medicalRecordCreator*
END;

MedicalRecord_SetMedicalRecordData(*Instance*, *data*) =
PRE
Instance $\in$ *MedicalRecord* $\wedge$ *data* $\in$ *STR*

$\wedge$ *MedicalRecord__IsValidated*(*Instance*) = **FALSE**
THEN
MedicalRecord__Data := ({*Instance*} $\lhd$ *MedicalRecord__Data*) $\cup$ {*Instance* $\mapsto$ *data*}
END;

result $\leftarrow$ **MedicalRecord_GetMedicalRecordData**(*Instance*)=
PRE
Instance $\in$ *MedicalRecord*
THEN
result := *MedicalRecord__Data*(*Instance*)
END;

result $\leftarrow$ **MedicalRecord_GetMedicalRecordIsValidated**(*Instance*) =
PRE
Instance $\in$ *MedicalRecord*
THEN
result := *MedicalRecord__IsValidated*(*Instance*)
END;

MedicalRecord__validate(*Instance*)=
PRE
Instance $\in$ *MedicalRecord* $\wedge$ *MedicalRecord__IsValidated*(*Instance*) = **FALSE**
$\wedge$ *Instance* $\in$ **dom**(*MedicalRecord__Data*)
THEN
MedicalRecord__IsValidated(*Instance*) := **TRUE**
END;

result $\leftarrow$ **Patient__GetPatientMedicalRecord**(*Instance*)=
PRE

$Instance \in Patient$

THEN $result := patientMedicalRecord(Instance)$
END ;

$result \leftarrow$ **Patient__GetPatientDepartment**$(Instance)=$
PRE
$Instance \in Patient$

THEN $result := patientDepartment(Instance)$
END ;

$result \leftarrow$ **MedicalEmployee__GetMedicalEmployeeDepartment**$(Instance)=$
PRE
$Instance \in MedicalEmployee$

THEN $result := MedicalEmployeeDepartment(Instance)$
END ;

Patient__SetPatientDepartment$(Instance, patientDepartment\_\_DepartmentValue)=$
PRE
$Instance \in Patient \land patientDepartment\_\_DepartmentValue \in Department \land (Instance \mapsto patientDepartment\_\_DepartmentValue) \notin patientDepartment$
THEN
$patientDepartment := (\{Instance\} \lhd patientDepartment) \cup \{(Instance \mapsto patientDepartment\_\_DepartmentValue)\}$
END ;

$result \leftarrow$ **Patient__GetSSN**$(Instance)=$
PRE
$Instance \in Patient$

THEN $result := Patient\_\_SSN(Instance)$
END ;

MedicalEmployee__joinDepartment$(Instance, hh)=$
PRE
$Instance \in MedicalEmployee \land hh \in Department \land hh \notin MedicalEmployeeDepartment[\{Instance\}]$
THEN
$MedicalEmployeeDepartment := MedicalEmployeeDepartment \cup \{(Instance \mapsto hh)\}$
END ;

MedicalEmployee__leaveDepartment$(Instance)=$
PRE
$Instance \in MedicalEmployee \land Instance \in \mathbf{dom}(MedicalEmployeeDepartment)$
THEN
$MedicalEmployeeDepartment := \{Instance\} \lhd MedicalEmployeeDepartment$
END

END

Annexe B

Les spécifications B complètes issues du modèle de sécurité de l'exemple du SI médical

B.1 La machine *UserAssignment*

MACHINE
UserAssignments
SEES
Functional_Model,
Context
SETS
$ROLES = \{Medical\_Employee, Doctor, Nurse, DepartmentDirector\}$;
$SESSIONS = \{noSession, s1, s2\}$

VARIABLES
roleOfOrg,
Roles_Hierarchy,
Org_Hierarchy,
user_assign,
currentUser,
currentOrg,
sessions,
session_user,
session_orgRole,
currentOrg_session,
currentSession,
SSD_mutex, *DSD_mutex*
INVARIANT
$Roles\_Hierarchy \in ROLES \leftrightarrow ROLES \wedge$
$roleOfOrg \in ORG \nrightarrow \mathcal{P}\ (ROLES) \wedge$
$\mathbf{closure1}(Roles\_Hierarchy) \cap \mathbf{id}(ROLES) = \emptyset \wedge$
$Org\_Hierarchy \in ORG \leftrightarrow ORG \wedge$
$\mathbf{closure1}(Org\_Hierarchy) \cap \mathbf{id}(ORG) = \emptyset \wedge$

$user\_assign \in USERS \leftrightarrow (ORG \times ROLES) \wedge$

$\forall (uu, co, ro).(uu \in USERS \wedge uu \in \mathbf{dom}(user\_assign) \wedge$
$co \in ORG \wedge ro \in ROLES \wedge (co, ro) \in user\_assign[\{uu\}] \Rightarrow (co, ro) \in allOrgRoles) \wedge$
$currentUser \in USERS \wedge$
$currentOrg \in ORG \wedge$
$sessions \subseteq SESSIONS \wedge$
$session\_user \in sessions \rightarrow USERS \wedge$
$currentOrg\_session \in sessions \rightarrow ORG \wedge$
$currentSession \in SESSIONS \wedge$
$session\_orgRole \in sessions \leftrightarrow (ORG \times ROLES) \wedge$

$\forall (ss, co, ro).(ss \in SESSIONS \wedge ss \in \mathbf{dom}(session\_orgRole) \wedge$
$co \in ORG \wedge ro \in ROLES \wedge (co, ro) \in session\_orgRole[\{ss\}] \Rightarrow (co, ro) \in allOrgRoles) \wedge$

$\forall ss.(ss \in sessions \wedge ss \in \mathbf{dom}(session\_orgRole) \Rightarrow \mathbf{card}(\mathbf{dom}(\mathbf{ran}(\{ss\} \lhd session\_orgRole))) = 1)$
$\wedge$

$SSD\_mutex : (\mathcal{P}_1 (ROLES) \times ORG) \nrightarrow \mathbf{NAT1}$
$\wedge \forall nn.(nn \in \mathbf{NAT1} \wedge nn \in \mathbf{ran}(SSD\_mutex) \Rightarrow nn \geq 2)$
$\wedge \forall (rs,co).(rs \in \mathcal{P}_1 (ROLES) \wedge co \in ORG \wedge (rs,co) \in \mathbf{dom}(SSD\_mutex) \Rightarrow \mathbf{card}(rs) \geq SSD\_mutex((rs,co)))$

$\wedge \forall (uu, co).(uu \in USERS \wedge uu \in \mathbf{dom}(user\_assign) \wedge co \in ORG \wedge co \in \mathbf{dom}(\mathbf{ran}(\{uu\} \lhd user\_assign)) \wedge (co \in \mathbf{ran}(\mathbf{dom}(SSD\_mutex))$
$\vee \exists org.(org \in ORG \wedge org \in \mathbf{ran}(\mathbf{dom}(SSD\_mutex)) \wedge org \in \mathbf{closure1}(Org\_Hierarchy)[\{co\}]))$
$\Rightarrow$
$\forall (rs, org).(rs \in \mathcal{P}_1 (ROLES) \wedge org \in ORG \wedge (org = co \vee org \in \mathbf{closure1}(Org\_Hierarchy)[\{co\}])$
$\wedge org \in \mathbf{ran}(\mathbf{dom}(SSD\_mutex)) \wedge rs \in \mathbf{dom}(\mathbf{dom}(SSD\_mutex) \rhd \{org\}) \Rightarrow$
$\mathbf{card}((\mathbf{closure1}(Roles\_Hierarchy)[\mathbf{ran}(\{co\} \lhd \mathbf{ran}(\{uu\} \lhd user\_assign))] \cup \mathbf{ran}(\{co\} \lhd \mathbf{ran}(\{uu\} \lhd user\_assign))) \cap rs) < SSD\_mutex(rs,org)$
)
) $\wedge$

$DSD\_mutex : (\mathcal{P}_1 (ROLES) \times ORG) \nrightarrow \mathbf{NAT1}$
$\wedge \forall nn.(nn \in \mathbf{NAT1} \wedge nn \in \mathbf{ran}(DSD\_mutex) \Rightarrow nn \geq 2)$
$\wedge \forall (rs,co).(rs \in \mathcal{P}_1 (ROLES) \wedge co \in ORG \wedge (rs,co) \in \mathbf{dom}(DSD\_mutex) \Rightarrow \mathbf{card}(rs) \geq DSD\_mutex((rs,co)))$
$\wedge \forall (uu, co).(uu \in USERS \wedge uu \in \mathbf{dom}(actifUserOrgRole) \wedge co \in ORG \wedge co \in \mathbf{dom}(\mathbf{ran}(\{uu\} \lhd actifUserOrgRole)) \wedge (co \in \mathbf{ran}(\mathbf{dom}(DSD\_mutex)) \vee \exists org.(org \in ORG \wedge org \in \mathbf{ran}(\mathbf{dom}(DSD\_mutex)) \wedge org \in \mathbf{closure1}(Org\_Hierarchy)[\{co\}])) \Rightarrow$
$\forall (rs, org).(rs \in \mathcal{P}_1 (ROLES) \wedge org \in ORG \wedge (org = co \vee org \in \mathbf{closure1}(Org\_Hierarchy)[\{co\}])$
$\wedge org \in \mathbf{ran}(\mathbf{dom}(DSD\_mutex)) \wedge rs \in \mathbf{dom}(\mathbf{dom}(DSD\_mutex) \rhd \{org\}) \Rightarrow$
$\mathbf{card}((\mathbf{closure1}(Roles\_Hierarchy)[\mathbf{ran}(\{co\} \lhd \mathbf{ran}(\{uu\} \lhd actifUserOrgRole))] \cup \mathbf{ran}(\{co\} \lhd \mathbf{ran}(\{uu\} \lhd actifUserOrgRole))) \cap rs) < DSD\_mutex(rs,org)$
))

DEFINITIONS

$orgRole == \{org, ro \mid org \in ORG \wedge org \in \mathbf{dom}(roleOfOrg) \wedge ro \in roleOfOrg(org)\};$

$orgRoles == (orgRole ; \mathbf{closure1}(Roles\_Hierarchy^{-1})) \cup orgRole ;$

$allOrgRoles == (\mathbf{closure1}(Org\_Hierarchy) ; orgRoles) \cup orgRoles ;$

$actifUserOrgRole == (session\_user^{-1} \rhd \{currentSession\} ; \{currentSession\} \lhd session\_orgRole)$

INITIALISATION

$Roles\_Hierarchy := \{(Doctor \mapsto Medical\_Employee),$
$(Nurse \mapsto Medical\_Employee),$
$(DepartmentDirector \mapsto Doctor)$
$\} \parallel$
$roleOfOrg := \{(Hospital \mapsto \{Medical\_Employee\})\} \parallel$
$user\_assign := \{(Fred \mapsto (Cardiology, DepartmentDirector)),$
$(Jack \mapsto (Cardiology, Doctor)),$
$(Jack \mapsto (Cardiology, Nurse)),$
$(Martin \mapsto (Radiology, DepartmentDirector)),$
$(Jean \mapsto (Radiology, Nurse))$
$\}$
$\parallel$
$currentUser := none \parallel$
$Org\_Hierarchy := \{(Cardiology \mapsto Hospital),$
$(Radiology \mapsto Hospital)\} \parallel$
$currentOrg := noOrg \parallel$
$SSD\_mutex := \emptyset \parallel$
$DSD\_mutex := \emptyset \parallel$
$sessions := \emptyset \parallel$
$session\_user := \emptyset \parallel$
$session\_orgRole := \emptyset \parallel$
$currentOrg\_session := \emptyset \parallel$
$currentSession := noSession$

OPERATIONS

$\mathbf{connect}(user, ss, co, roleSet) =$
PRE
$user \in USERS \wedge co \in ORG \wedge roleSet \in \mathcal{P}_1(ROLES) \wedge co \in \mathbf{dom}(\mathbf{ran}(\{user\} \lhd user\_assign)) \wedge$
$ss \in SESSIONS\text{-}\{noSession\} \wedge ss \notin sessions \wedge$
$roleSet \in \mathcal{P}_1(ROLES) \wedge roleSet \subseteq (\mathbf{ran}(\{co\} \lhd \mathbf{ran}(\{user\} \lhd user\_assign)) \cup \mathbf{closure1}(Roles\_Hierarchy)[\mathbf{ran}(\{c$
$\lhd \mathbf{ran}(\{user\} \lhd user\_assign))]) \wedge$
$roleSet \subseteq \mathbf{ran}(\{co\} \lhd allOrgRoles) \wedge$

$\forall (r1,r2).(r1 \in roleSet \wedge r2 \in roleSet \wedge r1 \neq r2$
$\Rightarrow \quad r2 \notin \mathbf{closure1}(Roles\_Hierarchy)[\{r1\}]) \wedge$

$\forall (rs, org).(rs \in \mathcal{P}_1(ROLES) \wedge org \in ORG \wedge (org = co \vee org \in \mathbf{closure1}(Org\_Hierarchy)[\{co\}])$
$\wedge org \in \mathbf{ran}(\mathbf{dom}(DSD\_mutex)) \wedge rs \in \mathbf{dom}(\mathbf{dom}(DSD\_mutex) \rhd \{org\}) \Rightarrow$
$\mathbf{card}((\mathbf{closure1}(Roles\_Hierarchy)[roleSet] \cup roleSet) \cap rs) < DSD\_mutex(rs, org))$
THEN
$sessions := sessions \cup \{ss\} \parallel$

$session\_user := session\_user \cup \{(ss \mapsto user)\} \;||$

$currentOrg\_session := currentOrg\_session \cup \{(ss \mapsto co)\} \;||$

$session\_orgRole := session\_orgRole \cup$

$\bigcup (ro).(ro \in roleSet \mid \{ss \mapsto (co, ro)\})$

END ;

add_role_session$(ro, ss) =$

PRE

$ss \in SESSIONS \wedge ss \in sessions \wedge ro \in ROLES \wedge ro \notin \mathbf{ran}(\mathbf{ran}(\{ss\} \lhd session\_orgRole)) \wedge$

$ro \in \mathbf{ran}(currentOrg\_session[\{ss\}] \lhd \mathbf{ran}(\{session\_user(ss)\} \lhd user\_assign)) \wedge$

$\forall (rs, org).(rs \in \mathcal{P}_1\ (ROLES) \wedge org \in ORG \wedge (org \in \mathbf{dom}(\mathbf{ran}(\{ss\} \lhd session\_orgRole)) \vee org \in \mathbf{closure1}(Org\_Hierarchy)[\mathbf{dom}(\mathbf{ran}(\{ss\} \lhd session\_orgRole))]) \wedge org \in \mathbf{ran}(\mathbf{dom}(DSD\_mutex)) \wedge rs \in \mathbf{dom}(\mathbf{dom}(DSD\_mutex) \rhd \{org\}) \Rightarrow$

$\mathbf{card}((\mathbf{closure1}(Roles\_Hierarchy)[\mathbf{ran}(\mathbf{ran}(\{ss\} \lhd session\_orgRole)) \cup \{ro\}] \cup \mathbf{ran}(\mathbf{ran}(\{ss\} \lhd session\_orgRole)) \cup \{ro\}) \cap rs) < DSD\_mutex(rs, org))$

THEN

$session\_orgRole := session\_orgRole \cup \{sess, coRole \mid sess = ss \wedge coRole \in \{co, role \mid role = \mathbf{ro} \wedge co = \mathbf{currentOrg\_session}(ss)\}\}$

END ;

drop_role_session$(ro, ss) =$

PRE

$ss \in SESSIONS \wedge ss \in sessions \wedge ro \in ROLES \wedge ro \in \mathbf{ran}(\mathbf{ran}(\{ss\} \lhd session\_orgRole))$

THEN

$session\_orgRole := session\_orgRole \rhd \{co, role \mid co \in \mathbf{dom}(\mathbf{ran}(\{ss\} \lhd session\_orgRole) \rhd \{ro\}) \wedge role = \mathbf{ro}\}$

END ;

disconnect$(user, ss) =$

PRE

$user \in USERS \wedge ss \in sessions \wedge session\_user(ss) = user$

THEN

$session\_user := \{ss\} \lhd session\_user \;||$

$session\_orgRole := \{ss\} \lhd session\_orgRole \;||$

$currentOrg\_session := \{ss\} \lhd currentOrg\_session \;||$

$sessions := sessions - \{ss\}$

END ;

add_userAssign$(user, role, org) =$

PRE

$user \in USERS \wedge role \in ROLES \wedge org \in ORG \wedge$

$(org \mapsto role) \in allOrgRoles$

THEN

$user\_assign := user\_assign \cup \{(user \mapsto (org \mapsto role))\}$

END ;

remove_userAssign$(user, role, org) =$

PRE

$user \in USERS \wedge role \in ROLES \wedge org \in ORG \wedge$
$(user \mapsto (org \mapsto role)) \in user\_assign$

THEN

$user\_assign := user\_assign - \{(user \mapsto (org \mapsto role))\}$

END ;

changeSession$(ss) =$

PRE

$ss : (sessions \cup \{noSession\})$

THEN

$currentSession := ss \;||$
$currentUser := session\_user(ss) \;||$
$currentOrg := currentOrg\_session(ss)$

END

END

B.2 La machine *Security_Model*

MACHINE

ORBAC_Model

INCLUDES

Functional_Model,
UserAssignements,
Context

SETS

$ENTITIES = \{medicalRecord, patient, department, medicalEmployee\}$;

$Attributes = \{PatientMedicalRecord, PatientDepartment, MedicalEmployeeDepartments, MedicalRecordCreator, patient\_SSN, department\_Name, medicalEmployee\_Name, medicalRecord\_Data, medicalRecord\_IsValidated\}$

$Operations = \{medicalRecord\_NEW, medicalRecord\_Free, medicalRecord\_SetMedicalRecordData, medicalRecord\_GetMedicalRecordData, medicalRecord\_GetMedicalRecordIsValidated, medicalRecord\_GetMedicalRecordCreator, medicalRecord\_validate, patient\_GetPatientMedicalRecord, patient\_GetPatientDepartment, medicalEmployee\_GetMedicalEmployeeDepartments, patient\_GetSSN, medicalEmployee\_joinDepartment, medicalEmployee\_leaveDepartment\}$;

$KindsOfAtt = \{public, private\}$;

$PERMISSIONS = \{ReadMedicalRecord, NurseMedicalRecordPerm, DoctorMedicalRecordPerm, ValidateMedicalRecord, DirectorPerm\}$;

$ActionsType = \{read, create, modify, delete, privateRead, privateModify\}$;

$Stereotypes = \{readOp, modifyOp\}$

VARIABLES

AttributeKind, *AttributeOf*, *OperationOf*,
constructorOf, *destructorOf*, *setterOf*, *getterOf*, *EntityActions*,
MethodActions, *StereotypeOps*, *PermissionAssignement*,
isPermitted

INVARIANT

$AttributeKind \in Attributes \rightarrow KindsOfAtt \wedge$
$AttributeOf \in Attributes \rightarrow ENTITIES \wedge$

$OperationOf \in Operations \rightarrow ENTITIES \wedge$
$constructorOf \in Operations \rightarrowtail ENTITIES \wedge$
$destructorOf \in Operations \rightarrowtail ENTITIES \wedge$
$setterOf \in Operations \rightarrowtail Attributes \wedge$
$getterOf \in Operations \rightarrowtail Attributes \wedge$
$StereotypeOps \in \quad Stereotypes \leftrightarrow Operations \wedge$
$setterOf \cap getterOf = \emptyset \wedge$

$PermissionAssignement \in PERMISSIONS \rightarrow ((\ \mathcal{P}_1\ (ORG) \times ROLES) \times ENTITIES) \wedge$
$\forall\ (pp,\ org,\ ro).(pp \in PERMISSIONS \wedge pp \in \mathbf{dom}(PermissionAssignement) \wedge org \in ORG \wedge$
$org \in \mathbf{union}(\mathbf{dom}(\mathbf{dom}(PermissionAssignement[\{pp\}]))) \wedge ro \in ROLES \wedge ro \in \mathbf{ran}(\mathbf{dom}(PermissionAssigneme$
$\Rightarrow (org,\ ro) \in allOrgRoles) \wedge$
$EntityActions \in PERMISSIONS \nrightarrow \mathcal{P}\ (ActionsType) \wedge$
$MethodActions \in PERMISSIONS \nrightarrow \mathcal{P}\ (Operations) \wedge$
$isPermitted : (ORG \times ROLES) \leftrightarrow Operations$

DEFINITIONS

$orgRole == \{org,\ ro \mid org \in ORG \wedge org \in \mathbf{dom}(roleOfOrg) \wedge ro \in roleOfOrg(org)\}\,;$

$orgRoles == (orgRole\,;\ \mathbf{closure1}(Roles\_Hierarchy^{-1}\ )) \cup orgRole\,;$

$allOrgRoles == (\mathbf{closure1}(Org\_Hierarchy)\,;\ orgRoles) \cup orgRoles\,;$

$superRoles(ro) == \mathbf{closure1}(Roles\_Hierarchy)[ro]\,;$

$subRoles(ro) == \mathbf{closure1}(Roles\_Hierarchy^{-1}\ )[ro]\,;$

$superOrg(org) == \mathbf{closure1}(Org\_Hierarchy)[org]\,;$

$subOrg(org) == \mathbf{closure1}(Org\_Hierarchy^{-1}\ )[org]\,;$

$expOrgPermissionAssignement == \{org,\ ro,\ en,\ pp \mid pp \in PERMISSIONS \wedge pp \in \mathbf{dom}(PermissionAssignement)$
$\wedge\ org \in ORG \wedge$
$org \in \mathbf{union}(\mathbf{dom}(\mathbf{dom}(PermissionAssignement[\{pp\}]))) \wedge ro \in ROLES \wedge ro \in \mathbf{ran}(\mathbf{dom}(PermissionAssigneme$
$\wedge$
$en \in ENTITIES \wedge en \in \mathbf{ran}(PermissionAssignement[\{pp\}])\}\,;$

$$allEntityActions == \{pp,\ at \mid pp \in PERMISSIONS \land at \in ActionsType \land pp \in \mathbf{dom}(EntityActions) \land at \in EntityActions(pp)\};$$

$$orgPermEntitiesCreation == \mathbf{ran}(\{create\} \lhd (allEntityActions^{-1}; expOrgPermissionAssignement^{-1}));$$

$$orgPermOpCreation == (orgPermEntitiesCreation;\ constructorOf^{-1});$$

$$orgPermEntitiesDestruction == \mathbf{ran}(\{delete\} \lhd (allEntityActions^{-1}; expOrgPermissionAssignement^{-1}));$$

$$orgPermOpDestruction == (orgPermEntitiesDestruction;\ destructorOf^{-1});$$

$$orgPermEntitiesPRead == \mathbf{ran}(\{privateRead\} \lhd (allEntityActions^{-1};\ expOrgPermissionAssignement^{-1}));$$

$$orgPermOpPRead == (orgPermEntitiesPRead;\ (getterOf;\ AttributeOf)^{-1});$$

$$publicGetters == getterOf \rhd \mathbf{dom}(AttributeKind \rhd \{public\});$$

$$orgPermEntitiesRead == \mathbf{ran}(\{read\} \lhd (allEntityActions^{-1}; expOrgPermissionAssignement^{-1}));$$

$$orgPermOpRead == (orgPermEntitiesRead;\ (publicGetters;\ AttributeOf)^{-1});$$

$$orgPermEntitiesPModify == \mathbf{ran}(\{privateModify\} \lhd (allEntityActions^{-1};\ expOrgPermissionAssignement^{-1}));$$

$$orgPermOpPModify == (orgPermEntitiesPModify;\ (setterOf;\ AttributeOf)^{-1});$$

$$publicSetters == setterOf \rhd \mathbf{dom}(AttributeKind \rhd \{public\});$$

$$orgPermEntitiesModify == \mathbf{ran}(\{modify\} \lhd (allEntityActions^{-1}; expOrgPermissionAssignement^{-1}));$$

$$orgPermOpModify == (orgPermEntitiesModify;\ (publicSetters;\ AttributeOf)^{-1});$$

$$orgPermEntitiesAbsoluteRead == \mathbf{ran}(\{privateRead,\ read\} \lhd (allEntityActions^{-1};\ expOrgPermissionAssignement^{-1}));$$

$$orgPermOpReadOps == (orgPermEntitiesAbsoluteRead;\ (StereotypeOps[\{readOp\}] \lhd OperationOf)^{-1});$$

$$orgPermEntitiesAbsoluteModify == \mathbf{ran}(\{privateModify,\ modify\} \lhd (allEntityActions^{-1}; expOrgPermis-$$

$sionAssignement^{-1}$));
$orgPermOpModifyOps == (orgPermEntitiesAbsoluteModify; (StereotypeOps[\{modifyOp\}] \lhd OperationOf)^{-1})$;

$ExpandedMethodActions == \{pp, op \mid pp \in PERMISSIONS \land op \in Operations \land op \in MethodActions(pp)\}$;

$orgPermOpMethodActions == \{oro, op \mid oro \in allOrgRoles \land op \in Operations \land$
$oro \in \mathbf{dom}(\mathbf{ran}(\{op\} \lhd (ExpandedMethodActions^{-1}; expOrgPermissionAssignement^{-1})))\}$;

$actifUserOrgRole == (session\_user^{-1} \rhd \{currentSession\}; \{currentSession\} \lhd session\_orgRole)$;

$currentOrgRole\_Session == \{org, ro \mid org = currentOrg \land org \in \mathbf{dom}(\mathbf{ran}(\{currentUser\} \lhd actifUserOrgRole)) \land ro \in ROLES \land$
$ro : (\mathbf{ran}(\{org\} \lhd \mathbf{ran}(\{currentUser\} \lhd actifUserOrgRole)) \cup \mathbf{closure1}(Roles\_Hierarchy^{-1})[\mathbf{ran}(\{org\} \lhd \mathbf{ran}(\{currentUser\} \lhd actifUserOrgRole))])\}$;

$setOfActiveRoles == \mathbf{ran}(\mathbf{ran}(actifUserOrgRole))$;

$orgPermissions == orgPermOpCreation \cup$
$orgPermOpDestruction \cup$
$orgPermOpPRead \cup$
$orgPermOpRead \cup$
$orgPermOpPModify \cup$
$orgPermOpModify \cup$
$orgPermOpReadOps \cup$
$orgPermOpModifyOps \cup$
$orgPermOpMethodActions$;
$allOrgPermissions == \{org, ro, op \mid org \in ORG \land org \in \mathbf{dom}(\mathbf{dom}(orgPermissions)) \cup subOrg(\mathbf{dom}(\mathbf{dom}(orgPermissio$
$\land ro \in ROLES \land ro \in \mathbf{ran}((\{org\} \cup superOrg(\{org\})) \lhd \mathbf{dom}(orgPermissions))$
$\land op : (\mathbf{ran}(\{pOrg, role \mid pOrg \in ORG \land role \in ROLES \land pOrg \in superOrg(\{org\}) \land role = ro\} \lhd orgPermissions) \cup$
$\mathbf{ran}(\{(org \mapsto ro)\} \lhd orgPermissions))$
$\}$;
$allPermissions == \{org, ro, op \mid org \in ORG \land ro \in ROLES \land op \in Operations \land op \in \mathbf{ran}(allOrgPermissions)$
$\land org \in \mathbf{dom}(\mathbf{dom}(allOrgPermissions \rhd \{op\}))$
$\land ro : (\mathbf{ran}(\{org\} \lhd \mathbf{dom}(allOrgPermissions \rhd \{op\})) \cup subRoles(\mathbf{ran}(\{org\} \lhd \mathbf{dom}(allOrgPermissions \rhd \{op\}))))\}$

INITIALISATION

$AttributeKind := \{(PatientMedicalRecord \mapsto public),$
$(PatientDepartment \mapsto public),$
$(MedicalEmployeeDepartments \mapsto public),$
$(MedicalRecordCreator \mapsto public),$

$(patient\_SSN \mapsto public)$,
$(department\_Name \mapsto public)$,
$(medicalEmployee\_Name \mapsto public)$,
$(medicalRecord\_Data \mapsto private)$,
$(medicalRecord\_IsValidated \mapsto private)\}$
||
$AttributeOf := \{(PatientMedicalRecord \mapsto patient)$,
$(PatientDepartment \mapsto patient)$,
$(MedicalEmployeeDepartments \mapsto medicalEmployee)$,
$(MedicalRecordCreator \mapsto medicalRecord)$,
$(patient\_SSN \mapsto patient)$,
$(department\_Name \mapsto department)$,
$(medicalEmployee\_Name \mapsto medicalEmployee)$,
$(medicalRecord\_Data \mapsto medicalRecord)$,
$(medicalRecord\_IsValidated \mapsto medicalRecord)$
}
||
$OperationOf := \{(medicalRecord\_NEW \mapsto medicalRecord)$,
$(medicalRecord\_Free \mapsto medicalRecord)$,
$(medicalRecord\_SetMedicalRecordData \mapsto medicalRecord)$,
$(medicalRecord\_GetMedicalRecordData \mapsto medicalRecord)$,
$(medicalRecord\_GetMedicalRecordIsValidated \mapsto medicalRecord)$,
$(medicalRecord\_GetMedicalRecordCreator \mapsto medicalRecord)$,
$(medicalRecord\_validate \mapsto medicalRecord)$,
$(patient\_GetPatientMedicalRecord \mapsto patient)$,
$(patient\_GetPatientDepartment \mapsto patient)$,
$(medicalEmployee\_GetMedicalEmployeeDepartments \mapsto medicalEmployee)$,
$(patient\_GetSSN \mapsto patient)$,
$(medicalEmployee\_joinDepartment \mapsto medicalEmployee)$,
$(medicalEmployee\_leaveDepartment \mapsto medicalEmployee)$
}
||
$constructorOf := \{(medicalRecord\_NEW \mapsto medicalRecord)\}$
||
$destructorOf := \{(medicalRecord\_Free \mapsto medicalRecord)\}$
||
$StereotypeOps := \emptyset$
||
$setterOf := \{$
$(medicalRecord\_SetMedicalRecordData \mapsto medicalRecord\_Data)$
}
||
$getterOf := \{(medicalRecord\_GetMedicalRecordData \mapsto medicalRecord\_Data)$,
$(medicalRecord\_GetMedicalRecordIsValidated \mapsto medicalRecord\_IsValidated)$,
$(medicalRecord\_GetMedicalRecordCreator \mapsto MedicalRecordCreator)$,
$(patient\_GetPatientMedicalRecord \mapsto PatientMedicalRecord)$,
$(patient\_GetPatientDepartment \mapsto PatientDepartment)$,

$(medicalEmployee\_GetMedicalEmployeeDepartments \mapsto MedicalEmployeeDepartments)$,
$(patient\_GetSSN \mapsto patient\_SSN)$
}
||
$PermissionAssignement := \{(ReadMedicalRecord \mapsto ((\{Hospital\},\ Medical\_Employee) \mapsto medicalRecord))$,
$(NurseMedicalRecordPerm \mapsto ((\{Radiology\},\ Nurse) \mapsto medicalRecord))$,
$(DoctorMedicalRecordPerm \mapsto ((\{Hospital\},\ Doctor) \mapsto medicalRecord))$,
$(ValidateMedicalRecord \mapsto ((\{Hospital\},\ Doctor) \mapsto medicalRecord))$,
$(DirectorPerm \mapsto ((\{Hospital\},\ DepartmentDirector) \mapsto medicalEmployee))$
}
||
$EntityActions := \{(ReadMedicalRecord \mapsto \{read,\ privateRead\})$,
$(NurseMedicalRecordPerm \mapsto \{create,\ modify,\ privateModify\})$,
$(DoctorMedicalRecordPerm \mapsto \{create,\ modify,\ privateModify,\ delete\})$
}
||
$MethodActions := \{(DoctorMedicalRecordPerm \mapsto \{medicalRecord\_validate\})$,
$(DirectorPerm \mapsto \{medicalEmployee\_joinDepartment,\ medicalEmployee\_leaveDepartment\})$
}
||
$isPermitted := \emptyset$

OPERATIONS
setPermissions =
PRE
$isPermitted = \emptyset$
THEN
$isPermitted := allPermissions$
END;

secure_MedicalRecord_NEW($Instance$,$patientMedicalRecord\_patientValue$, $medicalRecordCreator\_creatorValue$)=
PRE
$Instance \in MEDICALRECORD \quad \wedge \quad Instance \notin MedicalRecord$
$\wedge\ patientMedicalRecord\_patientValue \in Patient \wedge medicalRecordCreator\_creatorValue \in MedicalEmployee$
$\wedge\ patientMedicalRecord\_patientValue \in \mathbf{dom}(patientDepartment)$
$\wedge\ patientMedicalRecord\_patientValue \notin \mathbf{dom}(patientMedicalRecord)$
THEN
SELECT
$medicalRecord\_NEW \in isPermitted[currentOrgRole\_Session] \wedge medicalRecordCreator\_creatorValue = currentUser$
$\wedge\ patientDepartment(patientMedicalRecord\_patientValue) = currentOrg$
THEN
MedicalRecord_NEW($Instance$,$patientMedicalRecord\_patientValue$, $medicalRecordCreator\_creatorValue$)
END

END;

secure_MedicalRecord_Free(*Instance*)=
  PRE
    $Instance \in MEDICALRECORD \quad \wedge \quad Instance \in MedicalRecord$
    $\wedge\ MedicalRecord\_IsValidated(Instance) = \mathbf{FALSE}$
  THEN
    SELECT
      $medicalRecord\_Free \in isPermitted[currentOrgRole\_Session]$
      $\wedge\ (medicalRecordCreator(Instance) = currentUser \vee Doctor \in setOfActiveRoles)$
      $\wedge\ patientDepartment(patientMedicalRecord^{-1}(Instance)) = currentOrg$
    THEN
      MedicalRecord_Free(*Instance*)
    END
  END;

secure_MedicalRecord_SetMedicalRecordData(*Instance*, *data*) =
PRE
  $Instance \in MedicalRecord \wedge data \in STR$
  $\wedge\ MedicalRecord\_IsValidated(Instance) = \mathbf{FALSE}$
THEN
  SELECT
    $medicalRecord\_SetMedicalRecordData \in isPermitted[currentOrgRole\_Session]$
    $\wedge\ (medicalRecordCreator(Instance) = currentUser \vee Doctor \in setOfActiveRoles)$
    $\wedge\ patientDepartment(patientMedicalRecord^{-1}(Instance)) = currentOrg$
  THEN
    MedicalRecord_SetMedicalRecordData(*Instance*, *data*)
  END
END;

result ← **secure_MedicalRecord_GetMedicalRecordData**(*Instance*)=
PRE
  $Instance \in MedicalRecord$
THEN
  SELECT
    $medicalRecord\_GetMedicalRecordData \in isPermitted[currentOrgRole\_Session]$
  THEN
    result ← **MedicalRecord_GetMedicalRecordData**(*Instance*)
  END
END;

result ← **secure_MedicalRecord_GetMedicalRecordIsValidated**(*Instance*) =
PRE
  $Instance \in MedicalRecord$
THEN
  SELECT
    $medicalRecord\_GetMedicalRecordIsValidated \in isPermitted[currentOrgRole\_Session]$
  THEN

result ← **MedicalRecord_GetMedicalRecordIsValidated**(*Instance*)
END
END;

secure_MedicalRecord_validate(*Instance*)=
PRE
Instance ∈ *MedicalRecord* ∧ *MedicalRecord_IsValidated*(*Instance*) = **FALSE**
THEN
SELECT
medicalRecord_validate ∈ *isPermitted*[*currentOrgRole_Session*]
∧ *patientDepartment*(*patientMedicalRecord* $^{-1}$ (*Instance*)) = *currentOrg*
THEN
MedicalRecord_validate(*Instance*)
END
END;

result ← **secure_Patient_GetPatientMedicalRecord**(*Instance*)=
PRE
Instance ∈ *Patient*
THEN
SELECT
patient_GetPatientMedicalRecord ∈ *isPermitted*[*currentOrgRole_Session*]
THEN
result ← **Patient_GetPatientMedicalRecord**(*Instance*)
END
END;

result ← **secure_Patient_GetPatientDepartment**(*Instance*)=
PRE
Instance ∈ *Patient*
THEN
SELECT
patient_GetPatientDepartment ∈ *isPermitted*[*currentOrgRole_Session*]
THEN
result ← **Patient_GetPatientDepartment**(*Instance*)
END
END;

result ← **secure_MedicalEmployee_GetMedicalEmployeeDepartments**(*Instance*)=
PRE
Instance ∈ *MedicalEmployee*
THEN
SELECT
medicalEmployee_GetMedicalEmployeeDepartments ∈ *isPermitted*[*currentOrgRole_Session*]
THEN
result ← **MedicalEmployee_GetMedicalEmployeeDepartments**(*Instance*)
END
END;

result ← **secure_MedicalRecord_GetMedicalRecordCreator**(*Instance*)=
PRE
Instance ∈ *MedicalRecord*
THEN
SELECT
medicalRecord_GetMedicalRecordCreator ∈ *isPermitted*[*currentOrgRole_Session*]
THEN
result ← **MedicalRecord_GetMedicalRecordCreator**(*Instance*)
END
END;

result ← **secure_Patient_GetSSN**(*Instance*)=
PRE
Instance ∈ *Patient*
THEN
SELECT
patient_GetSSN ∈ *isPermitted*[*currentOrgRole_Session*]
THEN
result ← **Patient_GetSSN**(*Instance*)
END
END;

secure_MedicalEmployee_joinDepartment(*Instance*,*dep*)=
PRE
Instance ∈ *MedicalEmployee* ∧ *dep* ∈ *Department* ∧ *dep* ∉ *medicalEmployeeDepartments*[{*Instance*}]
THEN
SELECT
medicalEmployee_joinDepartment ∈ *isPermitted*[*currentOrgRole_Session*] ∧
medicalEmployeeDepartments(*Instance*) = *currentOrg*
THEN
MedicalEmployee_joinDepartment(*Instance*, *dep*)
||
add_userAssign(*Instance*, *role*, *currentOrg*)
END
END;

secure_MedicalEmployee_leaveDepartment(*Instance*, *dep*)=
PRE
Instance ∈ *MedicalEmployee* ∧ *Instance* ∈ **dom**(*medicalEmployeeDepartments*) ∧
dep ∈ *Department* ∧ *dep* ∈ *medicalEmployeeDepartments*[{*Instance*}]
THEN
SELECT
medicalEmployee_leaveDepartment ∈ *isPermitted*[*currentOrgRole_Session*]
∧ *medicalEmployeeDepartments*(*Instance*) = *currentOrg*
THEN
MedicalEmployee_leaveDepartment(*Instance*, *dep*)

END
END ;

changeCurrentSession(*ss*) =
PRE
ss : (*sessions* ∪ {*noSession*})
THEN
changeSession(*ss*)
END ;

session_roleSet ← **connect_user**(*user*, *ss*, *co*, *roleSet*) =
PRE
user ∈ *USERS* ∧ *co* ∈ *ORG* ∧ *roleSet* ∈ $\mathcal{P}_1$ (*ROLES*) ∧ *co* ∈ **dom**(**ran**({*user*} ◁ *user_assign*)) ∧ *ss* ∈ *SESSIONS*-{*noSession*} ∧ *ss* ∉ *sessions* ∧
roleSet ∈ $\mathcal{P}_1$ (*ROLES*) ∧ *roleSet* ⊆ **ran**({*co*} ◁ **ran**({*user*} ◁ *user_assign*)) ∧ *roleSet* ⊆ **ran**({*co*} ◁ *allOrgRoles*) ∧

∀ (*r1*,*r2*).(*r1* ∈ *roleSet* ∧ *r2* ∈ *roleSet* ∧ *r1* ≠ *r2*
⇒ *r2* ∉ **closure1**(*Roles_Hierarchy*)[{*r1*}]) ∧

∀ (*rs*, *org*).(*rs* ∈ $\mathcal{P}_1$ (*ROLES*) ∧ *org* ∈ *ORG* ∧ (*org* = *co* ∨ *org* ∈ **closure1**(*Org_Hierarchy*)[{*co*}]) ∧ *org* ∈ **ran**(**dom**(*DSD_mutex*)) ∧ *rs* ∈ **dom**(**dom**(*DSD_mutex*) ⩤ {*org*}) ⇒
card((**closure1**(*Roles_Hierarchy*)[*roleSet*] ∪ *roleSet*) ∩ *rs*) < *DSD_mutex*(*rs*, *org*))
THEN
connect(*user*, *ss*, *co*, *roleSet*) ||
session_roleSet := *roleSet* ∪ (**closure1**(*Roles_Hierarchy*))[*roleSet*]
END ;

secure_add_role_session(*ro*, *ss*) =
PRE
ss ∈ *SESSIONS* ∧ *ss* ∈ **dom**(*session_user*) ∧ *ro* ∈ *ROLES* ∧ *ro* ∉ **ran**(**ran**({*ss*} ◁ *session_orgRole*)) ∧
ro ∈ **ran**(*currentOrg_session*[{*ss*}] ◁ **ran**({*session_user*(*ss*)} ◁ *user_assign*)) ∧
∀ (*rs*, *org*).(*rs* ∈ $\mathcal{P}_1$ (*ROLES*) ∧ *org* ∈ *ORG* ∧ (*org* ∈ **dom**(**ran**({*ss*} ◁ *session_orgRole*)) ∨ *org* ∈ **closure1**(*Org_Hierarchy*)[**dom**(**ran**({*ss*} ◁ *session_orgRole*))]) ∧ *org* ∈ **ran**(**dom**(*DSD_mutex*)) ∧ *rs* ∈ **dom**(**dom**(*DSD_mutex*) ⩤ {*org*}) ⇒
card((**closure1**(*Roles_Hierarchy*)[**ran**(**ran**({*ss*} ◁ *session_orgRole*)) ∪ {*ro*}] ∪ **ran**(**ran**({*ss*} ◁ *session_orgRole*)) ∪ {*ro*}) ∩ *rs*) < *DSD_mutex*(*rs*, *org*))
THEN
add_role_session(*ro*, *ss*)
END ;

secure_drop_role_session(*ro*, *ss*) =
PRE
ss ∈ *SESSIONS* ∧ *ss* ∈ **dom**(*session_user*) ∧ *ro* ∈ *ROLES* ∧ *ro* ∈ **ran**(**ran**({*ss*} ◁ *session_orgRole*))
THEN
drop_role_session(*ro*, *ss*)
END ;

disconnect_user($user$, ss) =
PRE
$user \in USERS \wedge ss \in sessions \wedge session\_user(ss) = user$
THEN
disconnect($user$, ss)
END

END

Bibliographie

[BCDE07] David A. Basin, Manuel Clavel, Jürgen Doser, and Marina Egea. A metamodel-based approach for analyzing security-design models. In *MoDELS*, pages 420–435, 2007.

[FK] David F. Ferraiolo and D. Richard Kuhn. Role-based access controls. *Computing Research Repository.*

[GBR07] Martin Gogolla, Fabian Büttner, and Mark Richters. Use : A uml-based specification environment for validating uml and ocl. *Science of Computer Programming*, 69(1-3) :27–34, 2007.

[LB08] Michael Leuschel and Michael J. Butler. Prob : an automated analysis toolset for the b method. *Software Tools for Technology Transfer*, 10(2) :185–203, 2008.

[LBD02] Torsten Lodderstedt, David A. Basin, and Jürgen Doser. Secureuml : A uml-based modeling language for model-driven security. In *UML*, pages 426–441, 2002.

[LIM+11] 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 *CAiSE Workshops*, pages 592–606, 2011.