B4MSecure

User Manual – Starting guide

Team VASCO
Laboratoire d'Informatique de Grenoble

B4MSecure

B4MSecure is an Eclipse platform dedicated to formally reason about functional UML models enhanced by an access control policy which follows the RBAC model.

The platform acts on three steps:

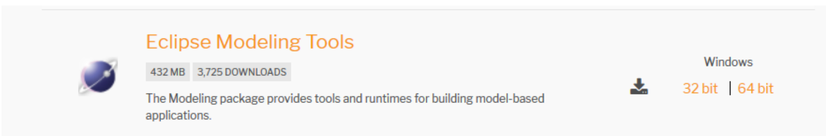
- Graphical modeling of a functional UML class diagram.
- Graphical modeling of an access control policy using a UML profile for RBAC (Role Based Access Control) and which is inspired by SecureUML.
- Translation of both models into B specifications, in order to formally reason about them.

The following sections guide you to the process of installing B4MSecure, defining your functional model and access control policy using the Papyrus modeling environment, and finally generating the corresponding B specifications.

1 Setting up B4MSecure

The B4MSecure tool is distributed as an extension to the Eclipse platform so, before starting, you need to install an Eclipse distribution.

We advise you to use the **Eclipse Modeling Tools** (Oxygen version) distribution[1], as it already includes some of the base technologies required by B4MSecure. Eclipse distributions are available at https://www.eclipse.org/downloads/eclipse-packages/.



The next step after installing Eclipse is to add the **B4MSecure** extensions. Eclipse provides facilities for adding new software to the platform or updating software in the system, the *update site* location is the only required item to update or install software within Eclipse.

The B4MSecure *update site* is available at the following URL:

http://vasco.imag.fr/tools/b4msecure/updates/releases/2.0/

To add the B4MSecure site, start your Eclipse and go to the “Install/Update > Available Software Sites” preference page, click the “Add” button, and type the name and URL of the update site, as shown in Figure 1.

[1] ***Note***: B4Msecure has been tested with Eclipse Modeling Tools (Neon and Oxygen versions), but you can also setup B4Msecure on an existing Eclipse installation of your choice. We warn you, however, that B4Msecure requires at least the following versions of Eclipse Modeling projects: EMF 2.12, UML2 5.2, OCL 6.2, Acceleo 3.7, QVTo 3.6

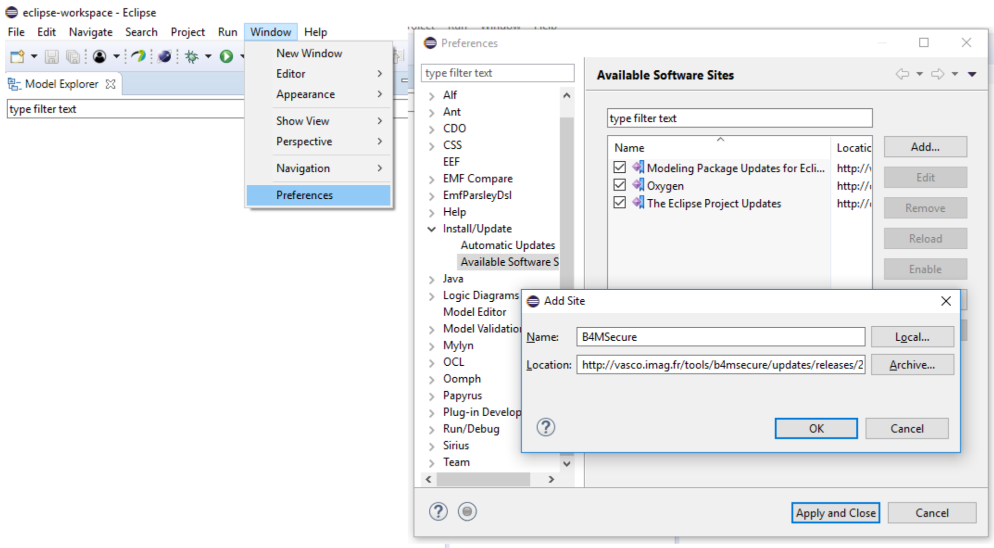


Figure 1 Adding B4Msecure update site

After adding the *Update Site*, you can proceed with the B4MSecure installation. Go to the install wizard ("Help" > "Install New Software…") and select the B4MSecure site in the "Work with" drop-down list, as shown in Figure 2.

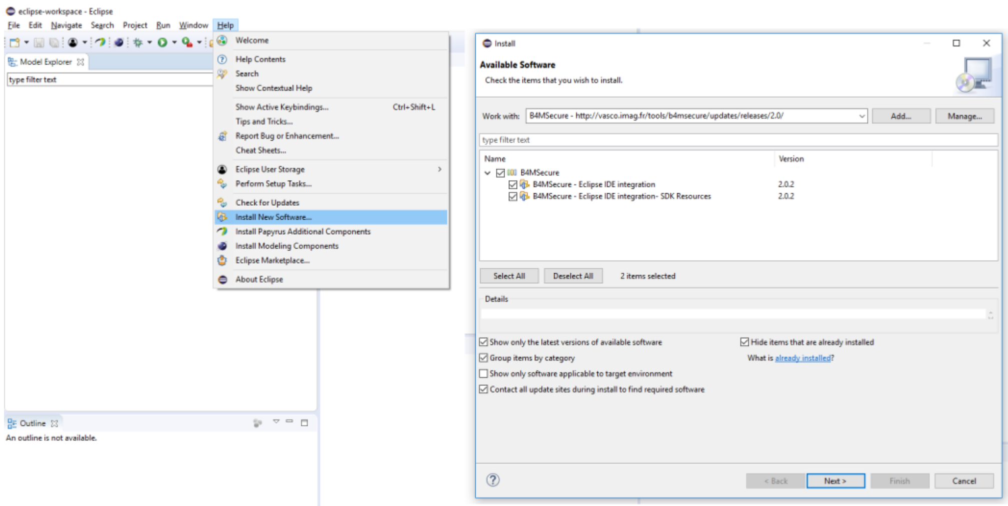


Figure 2 Installing B4MSecure

To complete the installation, just follow the instructions[2]: you will be prompted to select the software to install, accept the software license, and probably to restart your Eclipse.

Once you have installed B4MSecure, you also need to install an UML modeling tool. We use the **Papyrus Modeling environment**.

If you use Eclipse Modelling Tools the easier way is to install Papyrus is to use the discovery interface ("Help" > "Install Modeling Component") and select "Papyrus", as shown in Figure 3. Detailed instructions for installing Papyrus, and version compatibility requirements, are available at https://www.eclipse.org/papyrus/download.html.

[2] At some point during the installation, a warning will inform you that you are installing unsigned software. Just trust our software, and continue with the procedure.

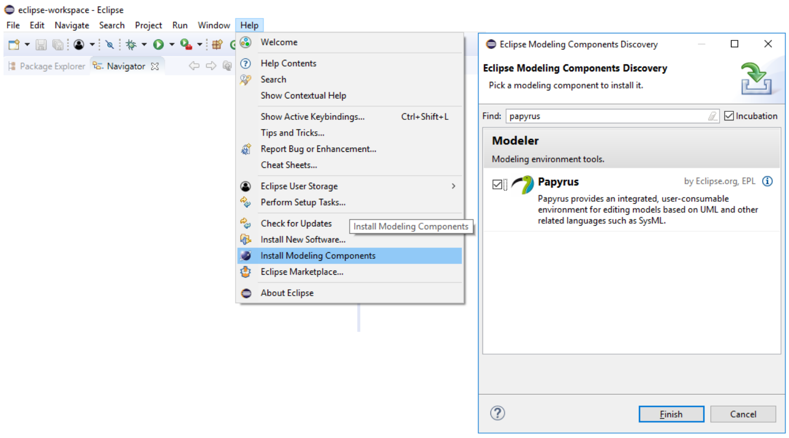


Figure 3 Papyrus installation

At this point you have successfully completed the environment setup. All the required software is installed and you are ready to start using **B4MSecure**. The next step is to create an UML model describing the functional and security aspects of your system.

2 System Modeling

We will use **Papyrus** to create the models used by B4MSecure. We expect the reader to be familiar with the UML class diagram editor, otherwise please refer to a Papyrus tutorial. Papyrus documentation is available at https://www.eclipse.org/papyrus/documentation.html

Use the new project wizard ("File" > "New" > "Project") to create a new Papyrus project named "*HospitalSystem*", with a root Class Diagram.

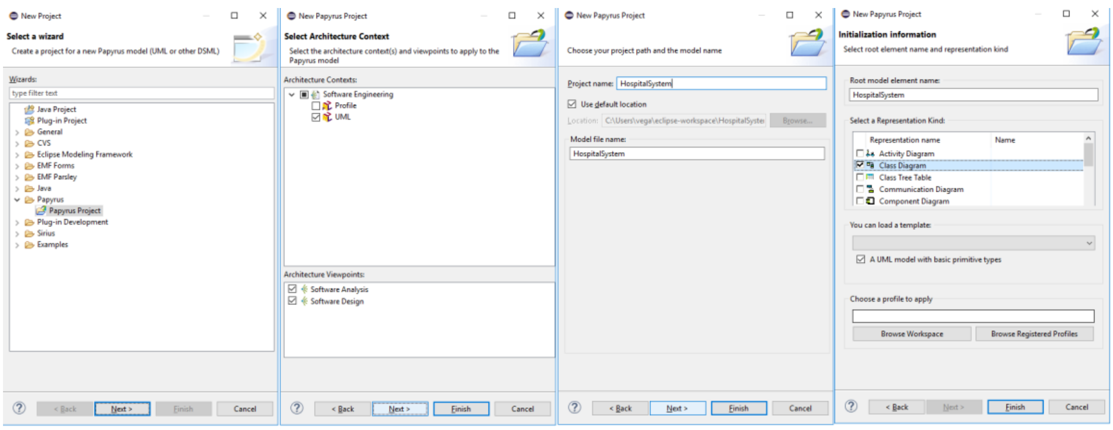


Figure 4 Papyrus project's creation

If you go to the Papyrus perspective, you should see a project named "*HospitalSystem*", with a single empty UML model file (named "*HospitalSystem"* as well). We are going now to create the functional and access policy packages.

2.1 Structuring the model

First, use the *palette* of the root class diagram to create two packages: the first one will store the functional model, and the second one will store the security model. Let us name these packages "***Functional***" and "***Policy***". Use the *Model Explorer* view to create a class diagram for each package, as shown in the following figure.

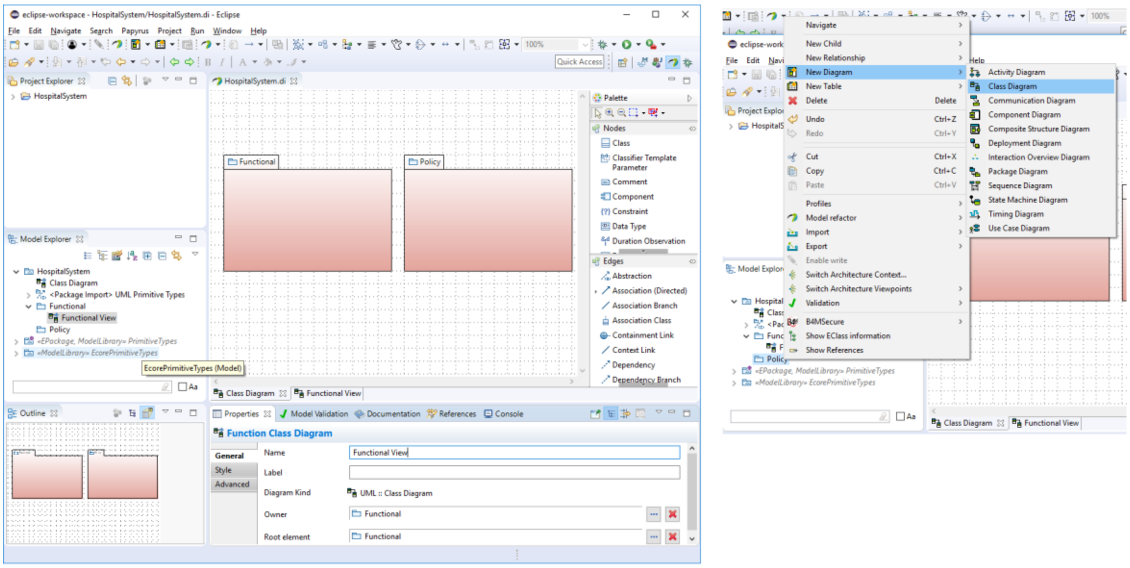


Figure 5 Structuring the model

2.2 Creating the functional model

Select the class diagram of the "***Functional***" package, and build the following model:

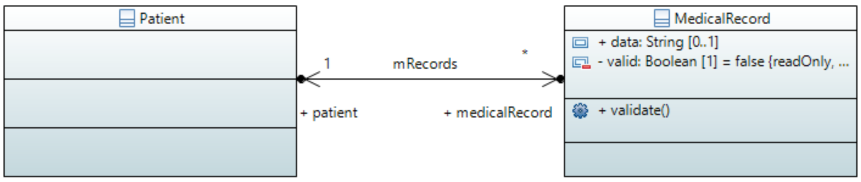


Figure 6 Hospital System, functional model

Attribute "***data***" is declared *public* and *optional*[3]. Attribute "***valid***" is *private*, *mandatory* and has a *default value* of *false*; it is also declared as *read only*, this means that no modification operation will be generated in the B specification for this attribute. Association "***mRecords***" links medical records to a single patient, while a patient may have several medical records. This association can be *navigated* in both directions. All these features are specified in the *Properties* view of the editor. Method "***validate***" is a *public* operation of class "***MedicalRecord***", with no parameters or result.

[3] *optional* attributes have multiplicity **0..1**, *mandatory* attributes have multiplicity **1**

2.3 Generation of the B functional specification

At this point, we have a functional model of the system; we can now use **B4MSecure** to generate the corresponding B specification. **B4MSecure** is available in the context menu associated to an UML package (it is activated by selecting a package and right-clicking on it).

In our scenario, we are going to select the "***Functional***" package in the *Model Explorer*, open the contextual menu and choose the B4MSecure transformation, as shown in the following figure.

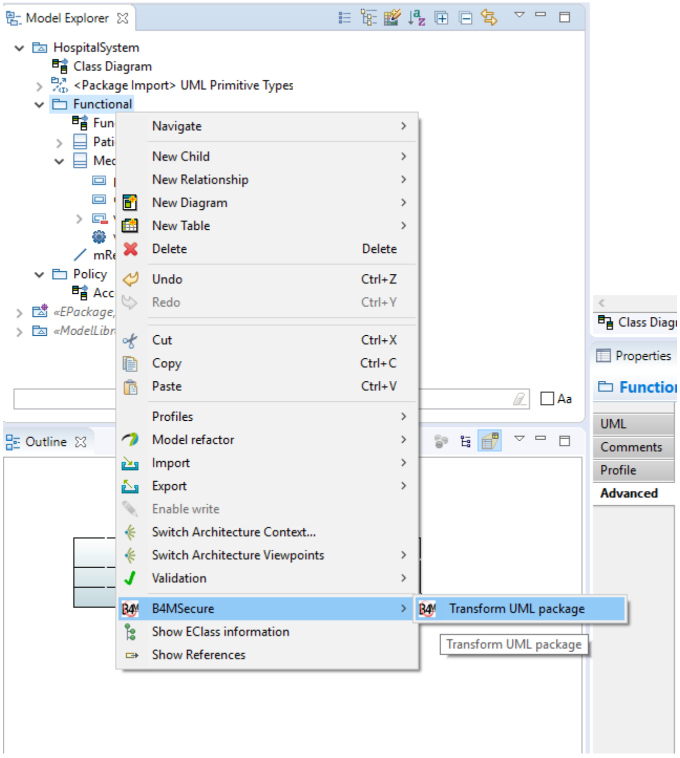


Figure 7 Generation of the B specification using B4MSecure

After execution of the B4mSecure transformation[4], several files are created in the same folder of the UML model, as shown in Figure 8. The generated **B specification machine** has the same name of the selected package, with the file extension "mch". So, in our case it is named "*Functional.mch"* (we can ignore, by now, the other generated B machine files, which are related to the security policy).

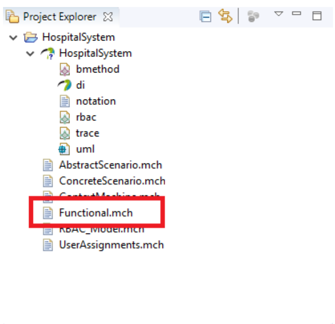


Figure 8 B4MSecure generated files

[4] You can see the log of the execution of the transformation in the *Console* view

In the resulting B specification, shown in Figure 9, classes are represented by sets (e.g. *PATIENT* and *MEDICALRECORD*). The existing instances of the class are represented by variables (e.g. *Patient* and *MedicalRecord*). Attributes are translated to functions (e.g. *MedicalRecord_data* and *MedicalRecord_valid* [5]). Associations are represented as a relation between the sets of instances (e.g. *mRecords*[6]).

```
MACHINE
    Functional

SETS
    STR;
    PATIENT;
    MEDICALRECORD

ABSTRACT_VARIABLES
    Patient,
    MedicalRecord,
    mRecords,
    MedicalRecord_data,
    MedicalRecord_valid

INVARIANT
    Patient : FIN(PATIENT) &
    MedicalRecord : FIN(MEDICALRECORD) &
    mRecords : MedicalRecord --> Patient &
    MedicalRecord_data : MedicalRecord +-> STR &
    MedicalRecord_valid : MedicalRecord --> BOOL

INITIALISATION
    Patient := {} ||
    MedicalRecord := {} ||
    mRecords := {} ||
    MedicalRecord_data := {} ||
    MedicalRecord_valid := {}

OPERATIONS
    MedicalRecord_validate(aMedicalRecord) =
    PRE aMedicalRecord : MedicalRecord

    THEN skip
    END;

    Patient_NEW(aPatient) =
    PRE aPatient : PATIENT &
        aPatient /: Patient

    THEN Patient := Patient \/ {aPatient}
    END;
```

Figure 9 B functional specification

Notice also that each method defined in a class in the model is translated to a B operation (e.g. *MedicalRecord_validate*), that takes as parameter an instance of the class. The generated operation has an empty (skip) substitution.

B4MSecure also generates basic operations to manipulate the model: class constructors (e.g. *Patient_NEW*), class destructors, attribute's accessors, attribute's setters, association navigation, and association's setters. A detailed explanation of the B4MSecure translation is beyond the scope of this manual; see the *publications* section of the site for a more thorough description.

[5] Notice that optional attributes, for example *data,* are translated to a partial function

[6] Notice that in this particular case, the *Patient* end of the association *mRecords* has multiplicity 1, so the relation is a total function

2.4 Adding invariants to the B specification

The generated B specification has invariants that essentially represent typing and multiplicities defined in the UML model. You may want to add domain invariants to your specification.

One possibility is to directly edit the generated file; however, your changes will be lost the next time that you execute the B4MSecure transformation. B4Msecure then offers the possibility to directly specify your B invariants in the UML model file.

Consider for instance the following invariant for the *Hospital System*: "a patient cannot have more than one medical record that is not validated". Given the generated B specification, this invariant can be expressed by the following B predicate:

!(p).(p : Patient => card((mRecords~[{p}] <| MedicalRecord_valid) |> {FALSE}) <= 1)

To add this invariant to your UML model, select the "***Functional"*** package in the *Model Explorer*, and a new *Constraint*, as shown in the following figure:

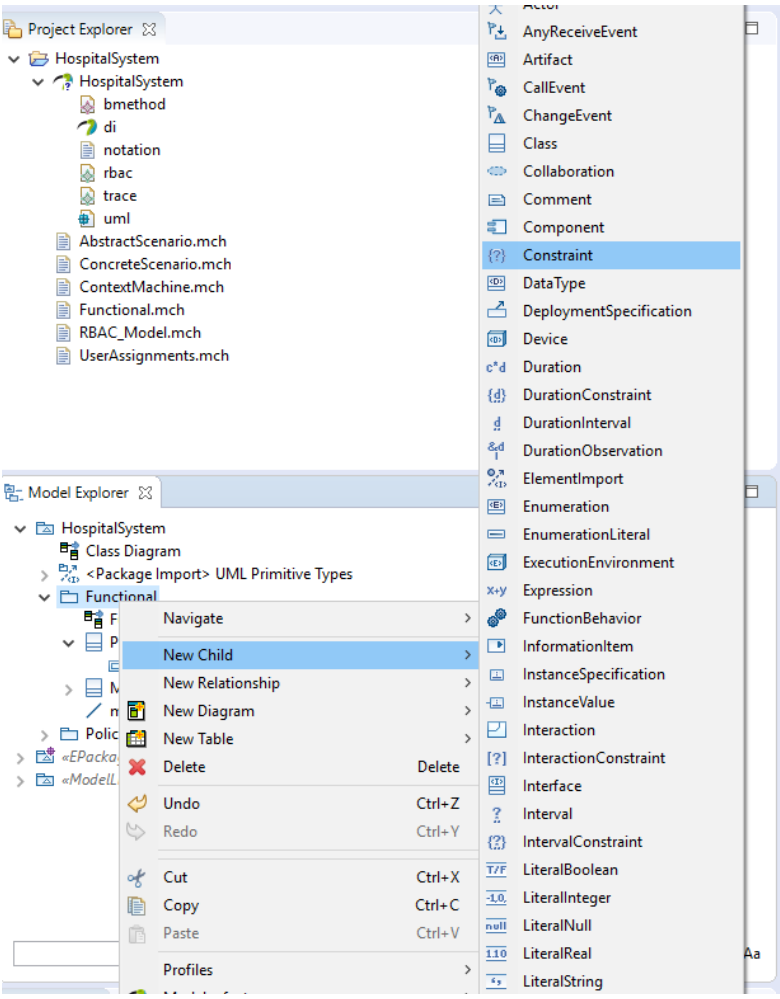


Figure 10 Adding constraints to the functional package

By default, Papyrus will create an OCL constraint, as shown in Figure 11; but it is possible to specify constraints in other languages. To add a new language: select the constraint value in the *Model Explorer*, go to the *Properties* view, click the "+" sign in the language section (red-circled in the figure). You will be prompted to add the new language (as shown in the inset, at the right of the figure), add the **B** language.

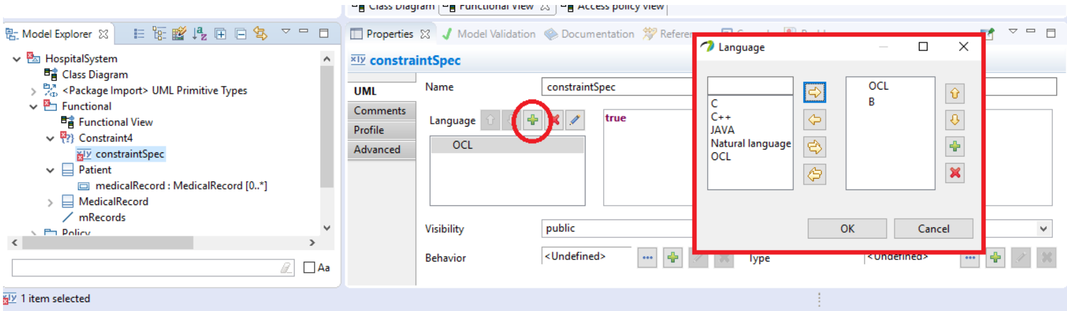


Figure 11 Papyrus: adding a new language to a constraint

Once you have added the B language, select it, and type your B predicate in the *Properties* view, as shown in Figure 12. Notice that we have also removed the unneeded OCL constraint, and renamed the constraint to give it a meaningful name.

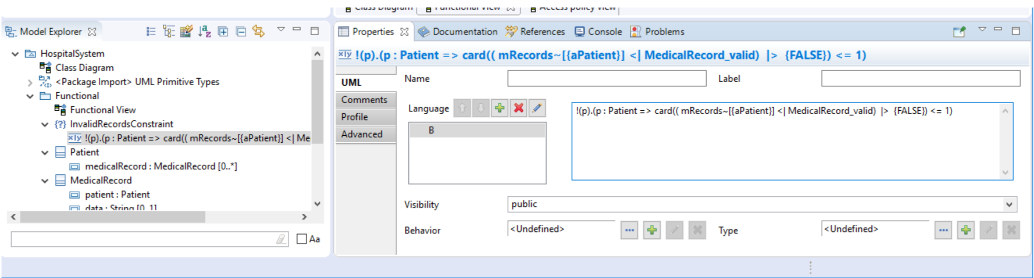


Figure 12 Specifying a B constraint to the model

At this point, the B invariant has been added to the UML model. We can now execute B4Msecure (see section 2.3) and regenerate the B specification. It will include the corresponding invariant.

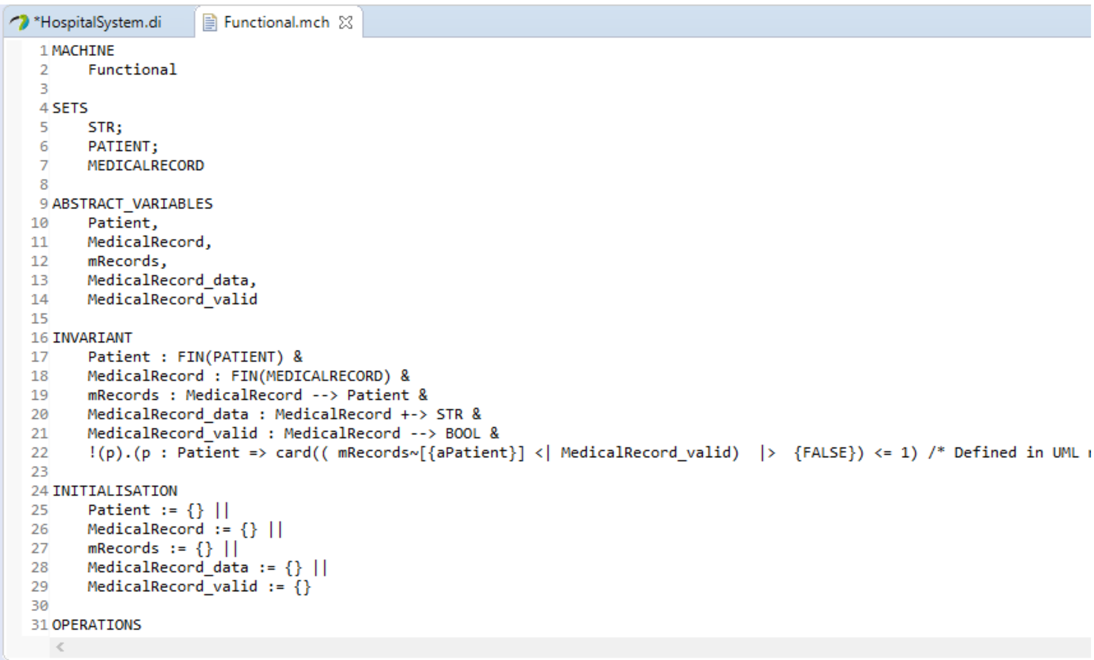


Figure 13 Domain invariants in generated B specification

2.5 Adding preconditions and operation's body

As we pointed out, methods defined in the UML model are translated by default into an empty B operation skeleton. To complete the B specification, we may need to add preconditions and specify the behavior of these methods.

Consider for instance the method "***validate"*** in the *Hospital System*. We can add a simple pre-condition to express the fact that validation only occurs for invalid records. This can be expressed in the B specification as predicate over the parameter of the operation *MedicalRecord_validate*:

MedicalRecord_valid(aMedicalRecord) = FALSE

In a similar way, one effect of the operation is to record the validation in the corresponding attribute. This can be expressed by the following B substitution:

MedicalRecord_valid(aMedicalRecord) := TRUE

As was the case for the invariants, we want to keep this information in the UML model (so that we do not need to edit the generated file). B4Msecure offers this possibility using the same approach that for invariants: we add constraints to the method in the UML model.

In our example, we select the "***validate"*** method in the *Model Explorer*, and add two B constraints (the procedure is the same as for the invariants, except that we select the method not the package), the result is shown in Figure 14.

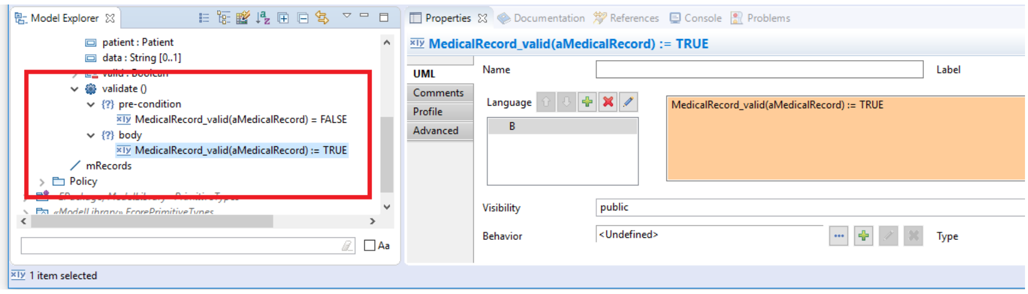


Figure 14 Defining pre-conditions and body

After that, select the "***validate"*** method, and in the *Properties* view change its body (as shown in the following figure) by selecting the constraint just defined.

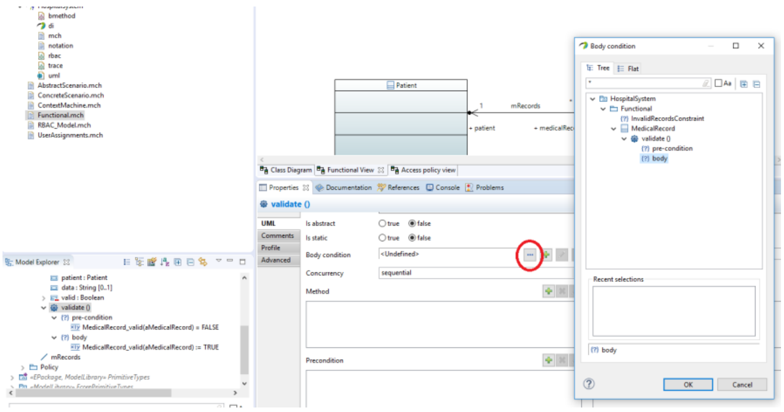


Figure 15 Specifying operation's body

In a similar way, go to the *precondition* section in the Properties view, and select the corresponding constraint, as shown in Figure 16.

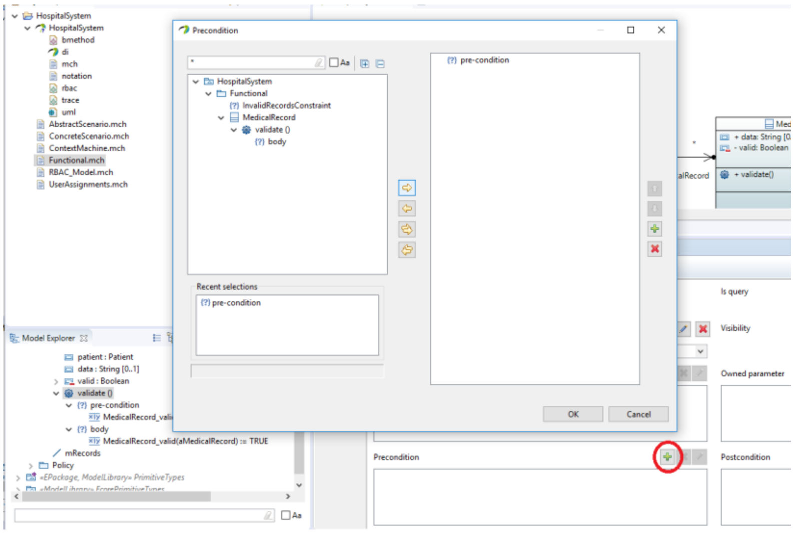
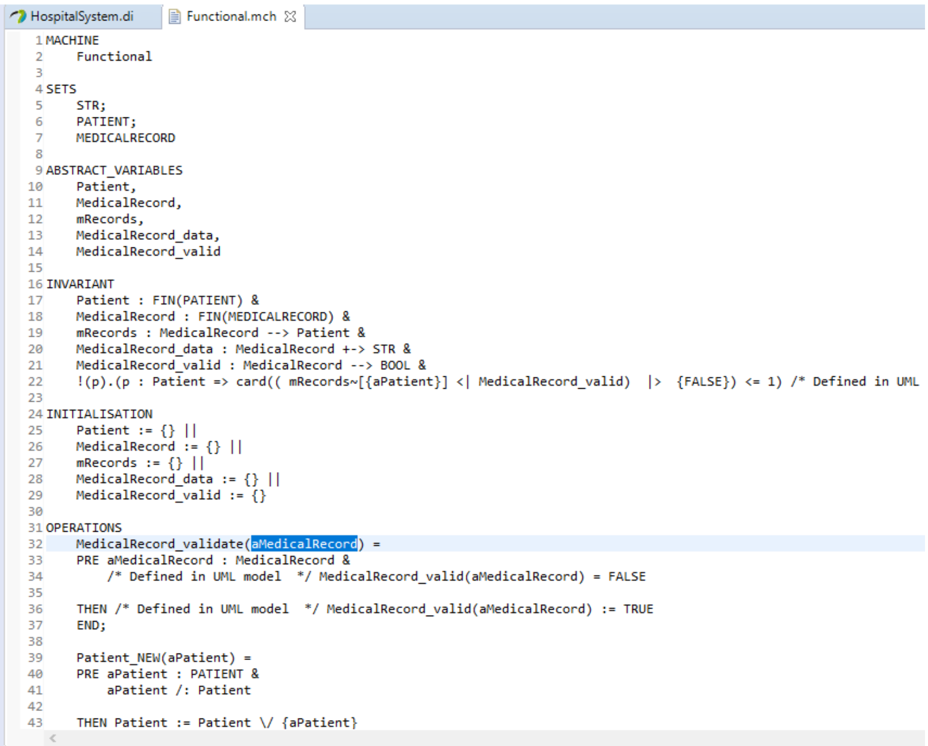


Figure 16 Specifying preconditions

At this point, the B precondition and body have been added to the UML model. We can now execute B4MSecure (see section 2.3) and regenerate the B specification.



```
MACHINE
    Functional

SETS
    STR;
    PATIENT;
    MEDICALRECORD

ABSTRACT_VARIABLES
    Patient,
    MedicalRecord,
    mRecords,
    MedicalRecord_data,
    MedicalRecord_valid

INVARIANT
    Patient : FIN(PATIENT) &
    MedicalRecord : FIN(MEDICALRECORD) &
    mRecords : MedicalRecord --> Patient &
    MedicalRecord_data : MedicalRecord +-> STR &
    MedicalRecord_valid : MedicalRecord --> BOOL &
    !(p).(p : Patient => card(( mRecords~[{aPatient}] <| MedicalRecord_valid)  |>  {FALSE}) <= 1) /* Defined in UML

INITIALISATION
    Patient := {} ||
    MedicalRecord := {} ||
    mRecords := {} ||
    MedicalRecord_data := {} ||
    MedicalRecord_valid := {}

OPERATIONS
    MedicalRecord_validate(aMedicalRecord) =
    PRE aMedicalRecord : MedicalRecord &
        /* Defined in UML model  */ MedicalRecord_valid(aMedicalRecord) = FALSE

    THEN /* Defined in UML model  */ MedicalRecord_valid(aMedicalRecord) := TRUE
    END;

    Patient_NEW(aPatient) =
    PRE aPatient : PATIENT &
        aPatient /: Patient

    THEN Patient := Patient \/ {aPatient}
```

Figure 17 Body and pre-conditions in generated B specification

We have finished our functional system model. Now we can start defining the associated security policy.

3 Security Modeling

To model the access control policy, B4MSecure uses a **UML profile** for RBAC (Role Based Access Control) that is inspired by **SecureUML**. In this section we will present how to use the profile in Papyrus, to define the security policy. We do not discuss in depth the concepts of SecureUML, the interested reader can refer to the paper [SecureUML: A UML-Based Modeling Language for Model-Driven Security]() by Torsten Lodderstedt, David Basin, and Jürgen Doser.

3.1 Applying the SecureUML profile

The first step to use an UML profile is to ***apply* the profile** to your model. In Papyrus, this is done by selecting the *root* of your model, and using the *Profile* tab of the *Properties* view, as shown in Figure 18. To select the profile, click on the "*apply registered profile*" button (the *plug* icon, red-circled in the figure) and choose the B4MSecure - SecureUML profile.

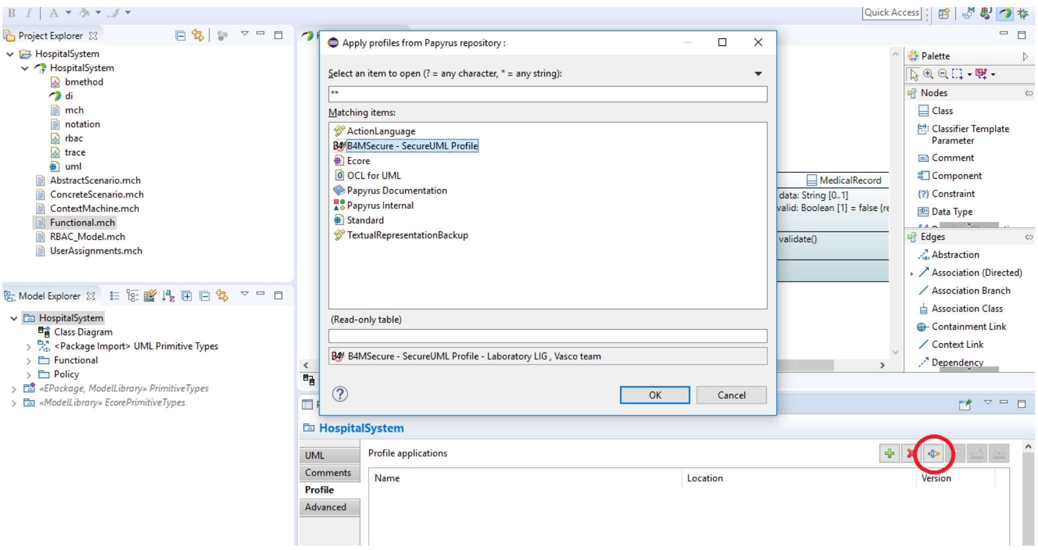


Figure 18 Applying the SecureUML profile

3.2 Defining the Role model

In SecureUML, security *roles* are represented by classes, and the *role hierarchy* is represented using a class generalization hierarchy. In our scenario, we are going to use the following role model:

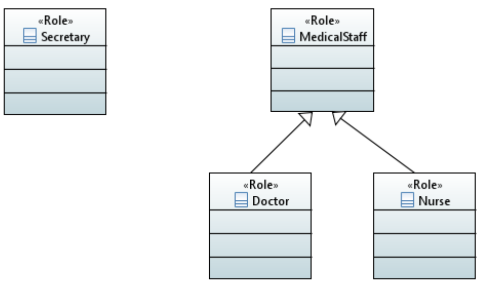


Figure 19 Role model (*Hospital System*)

To create the model open the class diagram associated to the "***Policy***" package, then create the classes and inheritance hierarchy shown in the previous figure. To differentiate domain classes from security roles, the SecureUML profile defines a **stereotype** named *Role*.

We need then to ***apply* the stereotype** to the classes in our UML model that represent roles. In Papyrus, this is done by selecting the class, and using the *Profile* tab of the *Properties* view, as shown in Figure 20. To select the stereotype, click on the "+" button (red-circled in the figure) and choose the *Role* stereotype.

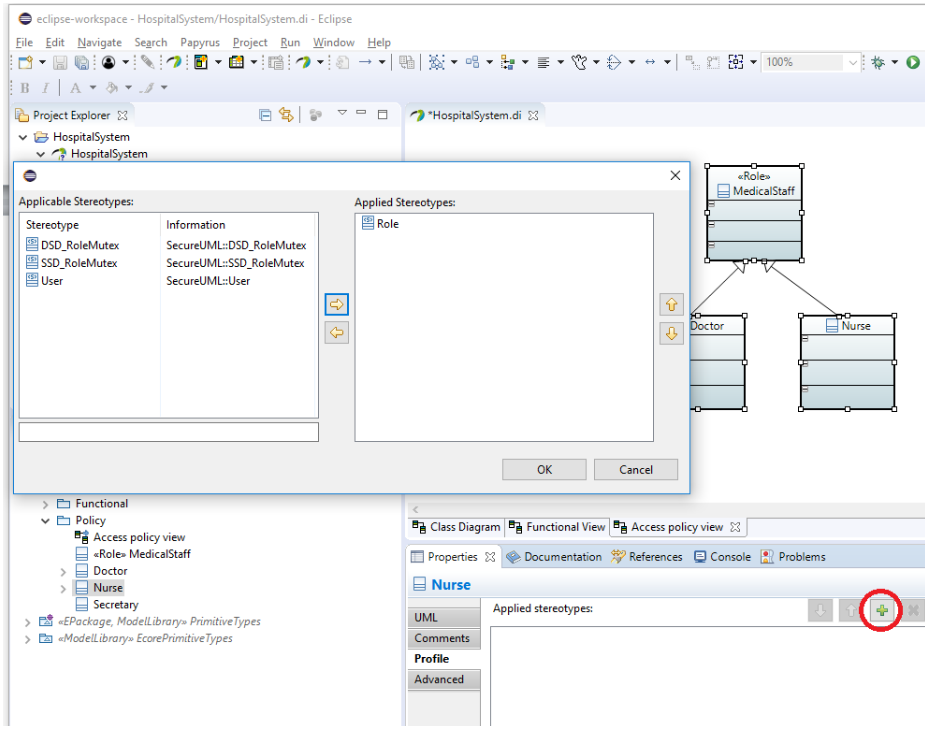


Figure 20 Applying an stereotype

At this point, we have finished defining the role model. We can now execute B4Msecure (see section 2.3) and regenerate the B specification. The corresponding B security specification is generated in file "*UserAssignments.mch".* A detailed explanation of the B4MSecure security translation is beyond the scope of this manual; but you can have a glimpse of the generated machine in Figure 21.

Figure 21 Role model in B

3.3 Defining the User Assignment model

In SecureUML, security *users* are represented by classes distinguished with the stereotype *User*. A *user assignment* (assigning a *role* to a *user*) is represented by defining an association between the class representing the *user* and the class representing the *role*.

In our scenario, we are going to use the following user assignment model:

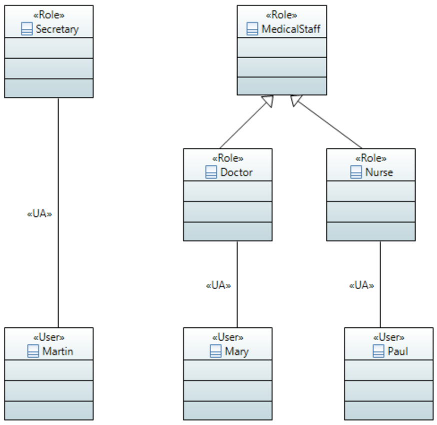


Figure 22 User Assignment

To create the model in Papyrus, just follow the same procedure explained in the previous section, and apply the stereotypes as appropriated. Notice that the associations have the applied stereotype *UA*. The procedure to apply a stereotype to an association is similar, just select the association and use the *Profile* tab in the *Properties* view.

Notice that Papyrus by default doesn't display the stereotypes of associations (and always displays ends' names and cardinalities). You can customize the display of an association in the diagram by selecting it and using the contextual (right-click) menu "Filters", as shown in the following figure.

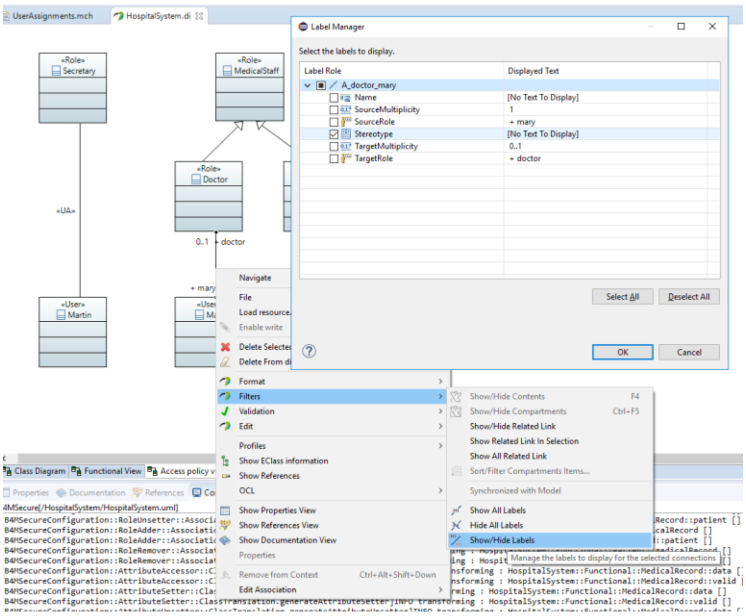


Figure 23 Customizing associations 'display

At this point, we have finished defining the user assignment model. We can now execute B4Msecure (see section 2.3) and regenerate the B specification. The corresponding B security specification is generated in files “*UserAssignments.mch”*.and “*ContextMachine.mch”* as shown in the following figure.

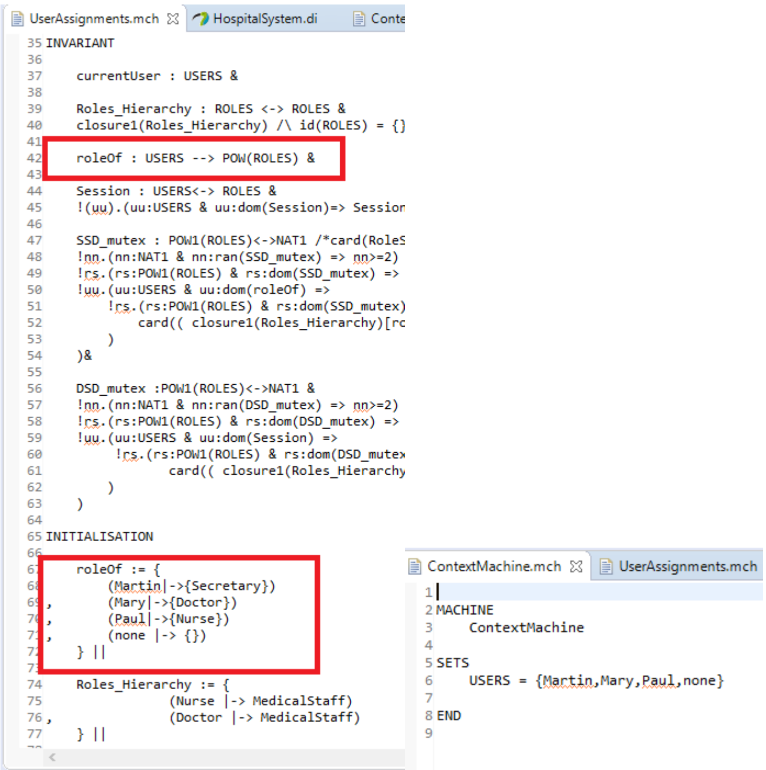


Figure 24 User asigment model in B

3.4 Defining the Permission model

In SecureUML, security *permissions* are represented by an association class between the class representing the *role* and the domain class representing the secured entity. Allowed actions are represented by methods of the permission.

For example, Figure 25 shows a permission policy expressing that a “*secretary may create a patient object”*. Notice the *Permission* stereotype applied to the association class, and the *EntityAction* stereotype applied to the allowed action *create*[7].

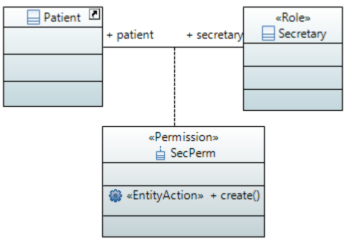


Figure 25 Secretary's permission

[7] The allowed entity actions that can be specified in SecureUML are : create, read, update/modify, delete

Figure 26 shows a policy expressing the following access rules: “*Every member of the medical staff may read the public information of medical record (this means that nurses and doctors have read access to the “Data” attribute)*”. “*Nurses may only create a medical record*”. “*Doctors may modify the data of a medical act and validate it”.* Notice the *MethodAction* stereotype in operation ***validate***[8], it is used to express that the method can be executed by a user assigned to the specified role.

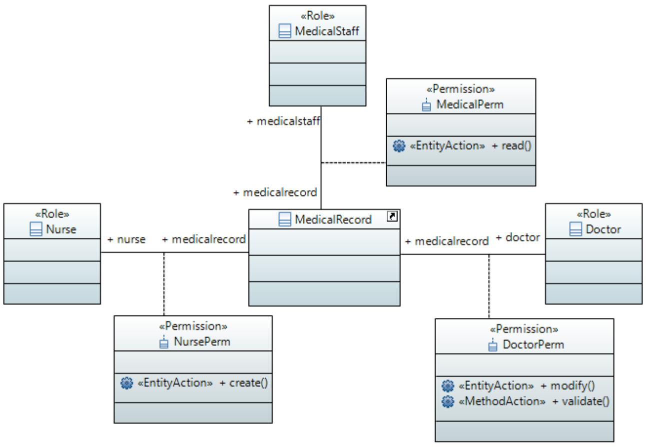


Figure 26 Security permission model

At this point, we have finished defining the *permission* model. We can now execute B4Msecure (see section 2.3) and regenerate the B specification. The corresponding B security specification is generated in file “*RBAC_model.mch”*, as shown in the following figure.

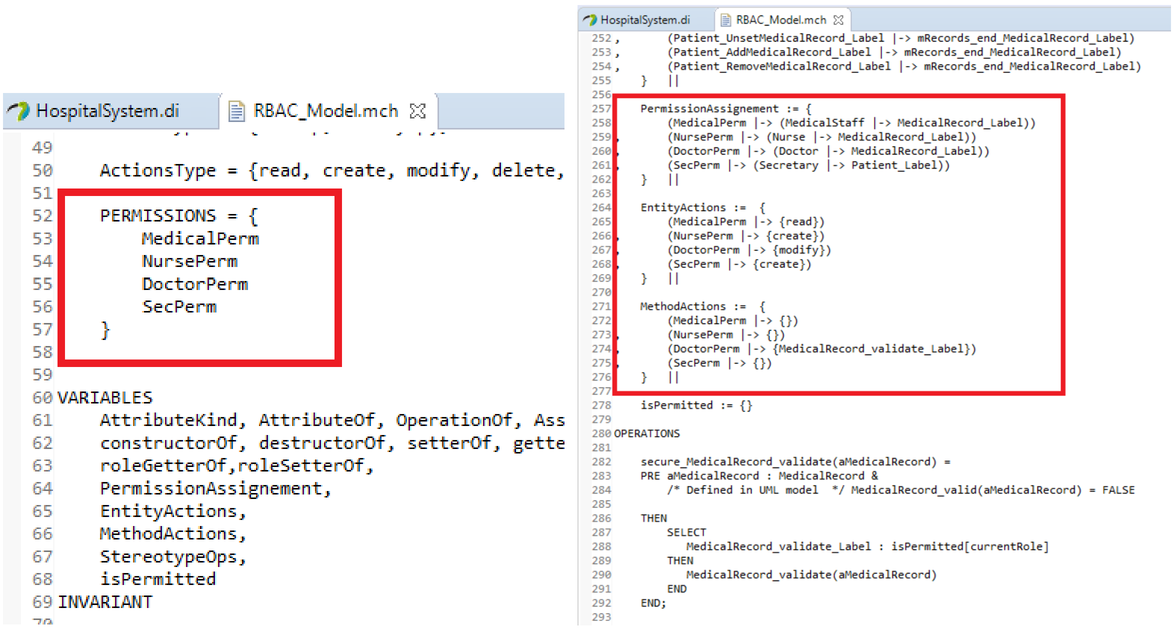


Figure 27 Permission specification model in B

[8] The name of the *Entity Action* method must match a method in the secured class

4 Conclusion

This guide has provided you with step-by-step instructions on how to use **Papyrus** and **SecureUML** to model secured systems in UML. Once you have completed your model, we have shown you how to use **B4MSEcure** to generate the corresponding B specification.

B4MSecure enables you to perform formal reasoning on your model. You can, for instance, take the B specification generated by **B4MSecure** and use the available tools of the B Method ecosystem for verifying proof obligations, or perform model checking.

We hope that **B4MSecure** will be a useful tool for you, but please contact us if you experience problems or simply want to request an enhancement.