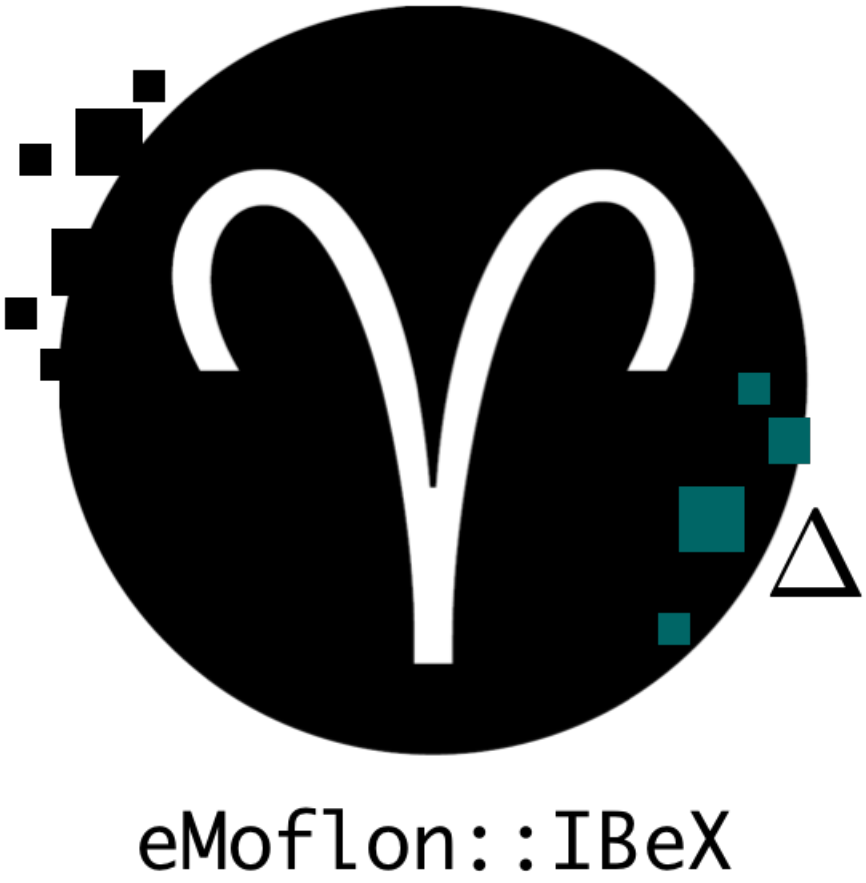
eMoflon::IBeX - Tutorial

Marek Daniv
August 5, 2023



Contents

1 Introduction

Welcome to our guide for metamodeling with **eMoflon::IBeX**. In the first part of this tutorial we will show you how to install the required software for using eMoflon::IBeX and how to create your first metamodel.
The second part of our tutorial will teach you the functions of **eMoflon::IBeX-GT** our interpreter for graph transformation rules, that employs various incremental graph pattern matching tools to provide solid performance even on large-scale models.
The third and final part is focused on bidirectional model transformation with **triple grammar graphs**.
There are almost no prerequisites to tackle this tutorial. The only thing you might need is some basic knowledge of Eclipse and programming in Java. A fundamental understanding of metamodeling might come in handy but is not required.
If you ever have problems with solving the tutorial or you just can´t figure out what you have done wrong, there is a solution of the whole tutorial in the **emoflon-ibex-tutorial** repository.

2 Install instructions

2.1 General

Basically there are three ways to install eMoflon. This tutorial describes the usual way of installing it in the following chapter. For interested people the two other ways are:

- the **Pre-built Eclipse Application** and
- the **Pre-built Virtual Machine (VM)**

How to install these two is specified on their download pages, accesible through the link.

Eclipse Modelling Tools:
First of all you need a running version of **Eclipse** that includes the **Eclipse Modelling Tools** package.
You can install it by running the regular Eclipse installer and choosing **Eclipse Modeling Tools** in the installation wizard. Afterwards just follow the instructions.
You can download the newest installer here.

Eclipse Modelling Framework:
The Eclipse Modelling Tools version you just installed includes, among other features, a set of plug-ins for Eclipse which enable the modulation of data models, the generation of corresponding code, and outputs based on this model.
This set of plug-ins is summarized under the name **Eclipse Modelling Framework (EMF)**.
The user can create metamodels via different means such as UML or XML schemes. The metamodels created with EMF consist of two parts: the **ecore** and the **genmodel** description files which you will get to know better throughout this tutorial[1].

2.2 Windows specific

GraphViz:
Another standalone software you will need for the visualization of the created metamodels is **GraphViz**.
You can download the newest installers for Windows here.
Just follow the instructions from the installer.

You can continue with step **2.4** now.

2.3 Linux specific

GraphViz:
Another standalone software you will need for the visualization of the created metamodels is **GraphViz**. On Linux it is requiered to install the corresponding package.
You can look at an overview for the different Linux plattforms here.

[1] See: Vogella EclipseEMF article here

2.4 Eclipse plugins

PlantUML:
If you have completed the previous section, you are ready to install the remaining software in Eclipse. You will need *PlantUML* for the visualization of our class diagrams. The easiest way to install this plugin is to use the integrated **Eclipse Marketplace**.
The *Marketplace* can be accessed over:

→ *Help* drop-down menu → *Eclipse Marketplace*

A new window should pop up, which presents an overview of available software plugins. Now type *PlantUML* in the search bar on the very top and install **the PlantUML plugin**.The chart below provides a cumulated view of the previous steps.

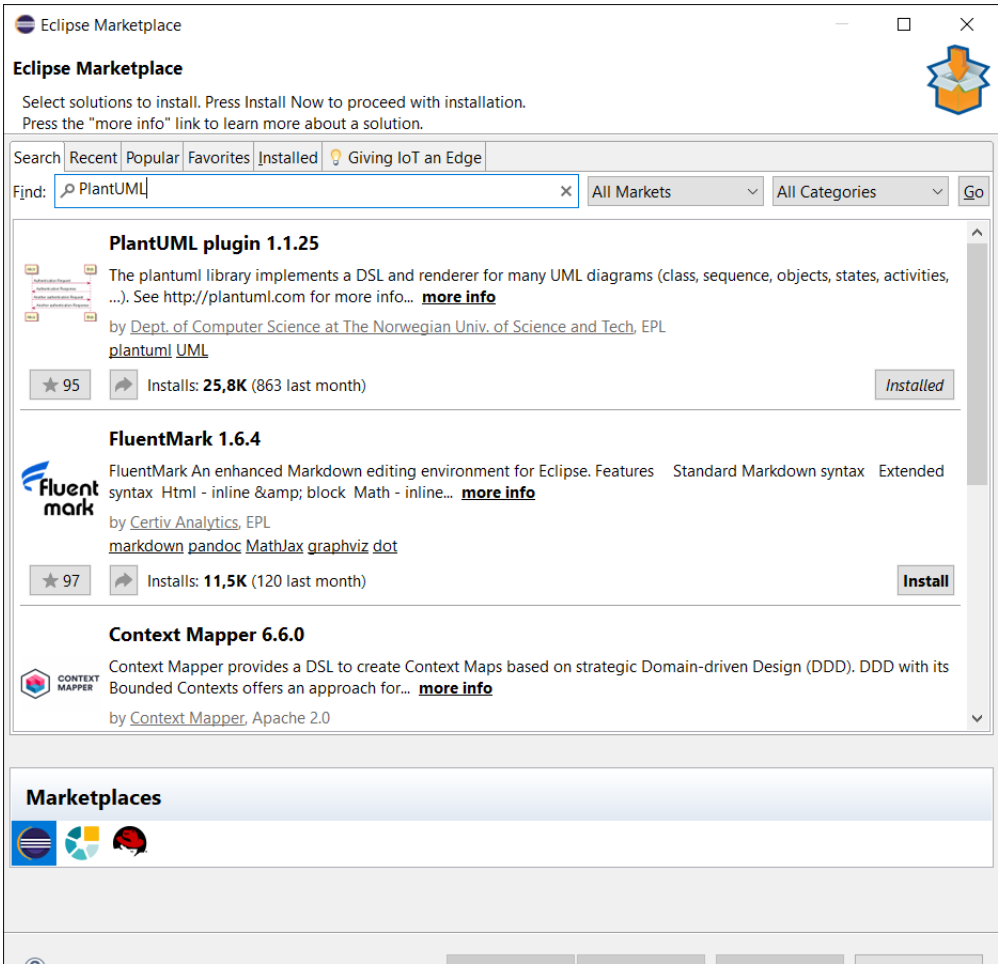


Figure 1: Screenshot of the *Marketplace*

HiPE:

Furthermor you need to install the plugin *HiPe*. You need to click:

→ *Help* → *Install New Software*

A new window should pop up and you can insert the following URL into the *Work with* field:

https://hipe-devops.github.io/HiPE-Updatesite/hipe.updatesite/

After pressing *Enter* you should now see the window from the screenshot below presenting *HiPE* next to a checkbox. Start the installation by selecting it, clicking *next*and accepting the license agreement. In some cases you need to press *install anyway* afterwards. Throughout the process of installing Eclipse may require a restart. If Eclipse restarts without any errors, everything necessary is installed.

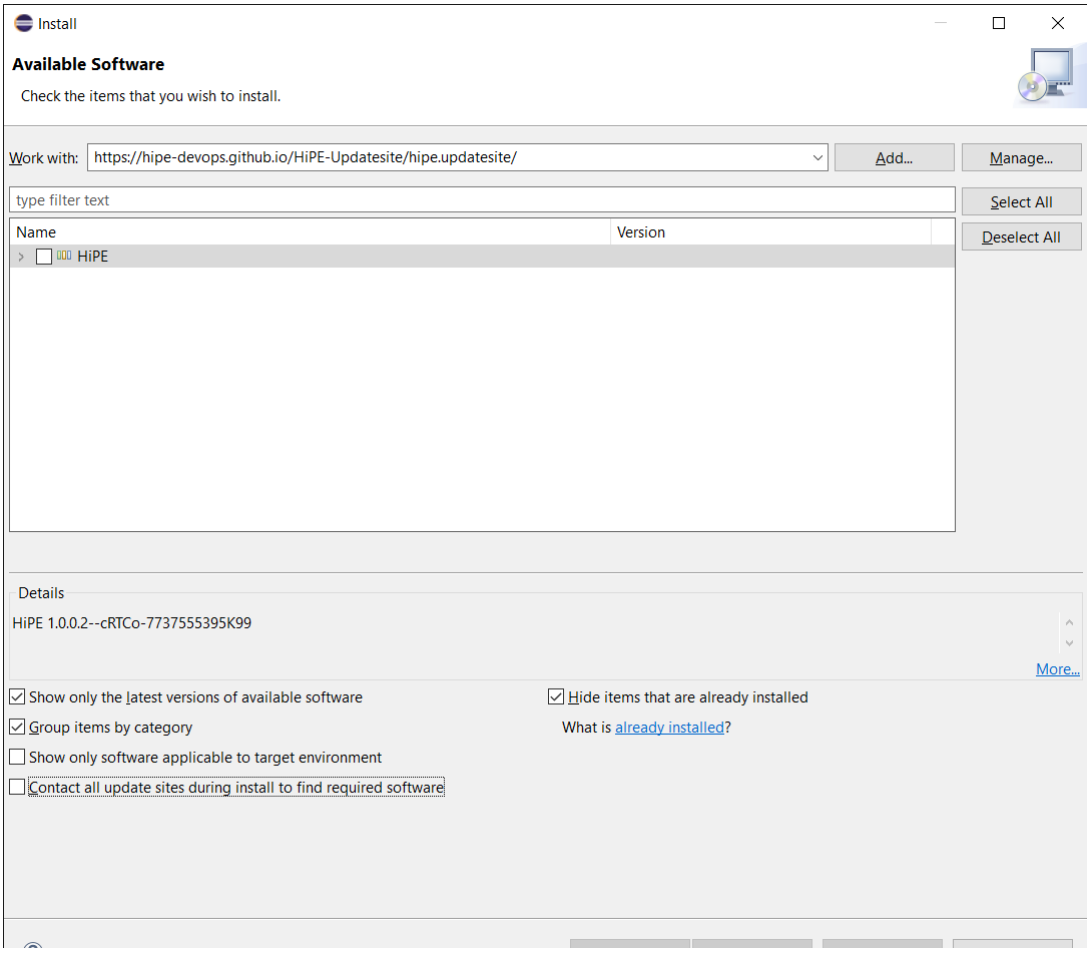


Figure 2: Screenshot of the *Install New Software* window for *HiPE*

eMoflon::IBeX:

Lastly we will install **eMoflon::IBeX**. This is done in the same way as the *HiPE* plugin. You can insert the following URL into the *Work with* field:

https://emoflon.org/emoflon-ibex-updatesite/snapshot/updatesite/

You should now see the window from the screenshot below presenting *eMoflon:IBeX* and it's different submodules. Although you will only need *Democles* and *HiPE* for this tutorial, it is advisable to install the whole suite.
Continue with the installation in the same way as in the previous step.

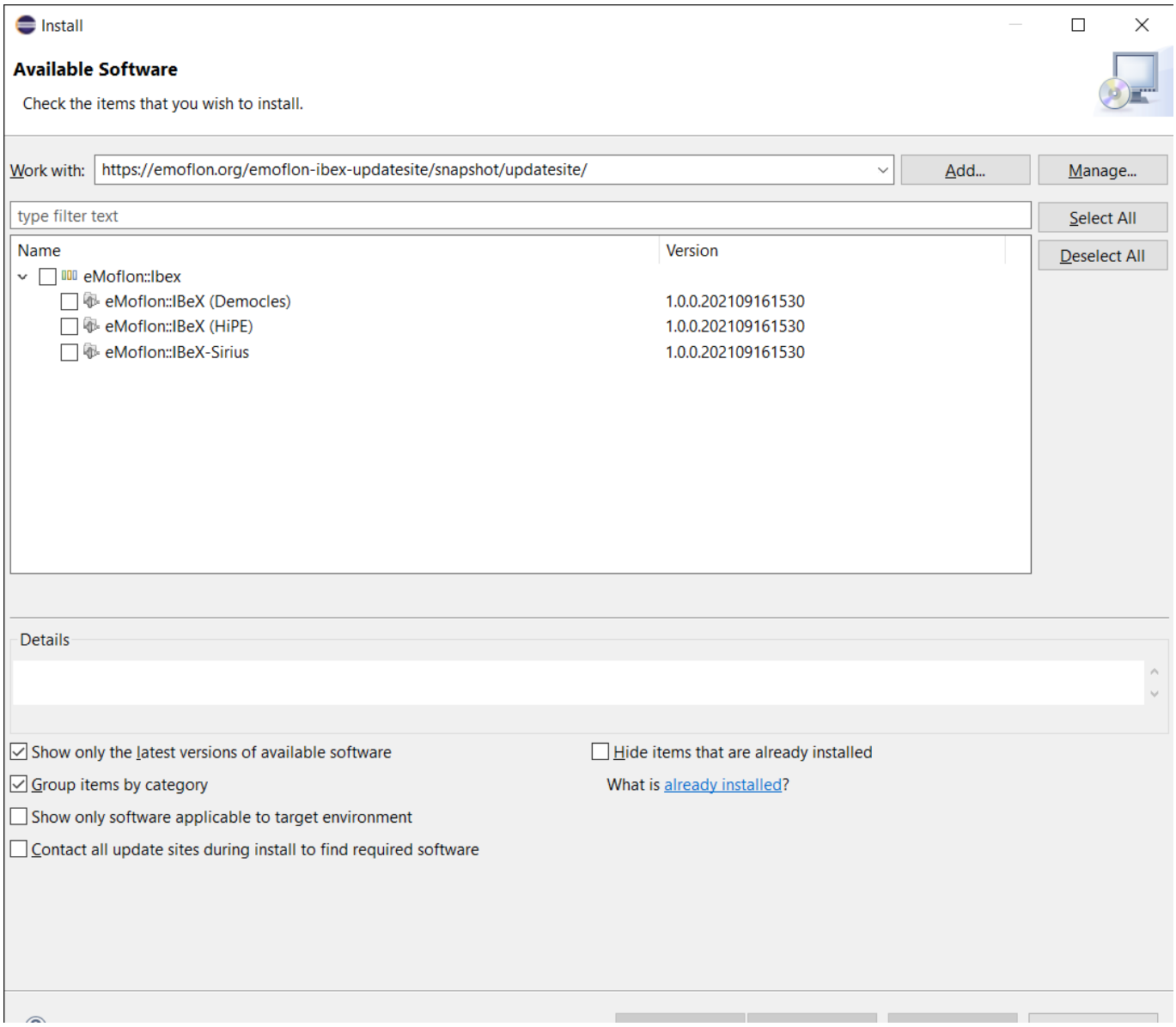


Figure 3: Screenshot of the *Install New Software* window for *eMoflon::IBeX*

Both *Democles* and *HiPE* are Eclipse projects which include pattern matching engines.
Sirius is a framework for visualization which implements a graphic editor for triple graph grammar rules.

In this tutorial we will just need the incremental pattern matchers of the first two. In contrast to *Democles HiPE* is a parallel pattern matcher.

3 Creating a metamodel

3.1 Metamodeling introduction

The first question you might be asking yourself is: What is metamodeling?
Metamodeling can be described as creating the model of a model. That means defining abstract rules and structures an instance of such a metamodel has to fulfill[2].

For a basic introduction into metamodeling and graph transformation see[3].

3.2 Adding the eMoflon toolbar

Now it is time to get to know the functionalities of eMoflon. Firstly you should add the **eMoflon toolbar** to your **Eclipse** as shown in the screenshot below.
Look for the Window tab in the top left corner and press:

→ *Window* → *Perspective* → *Open Perspective* → *Other*

Then open the **eMoflon toolbar icon** which is highlighted via the red square in the screenshot below. Note that sometimes the toolbar is displayed very tiny. This can be fixed by moving the toolbar via the four dots infront of it:

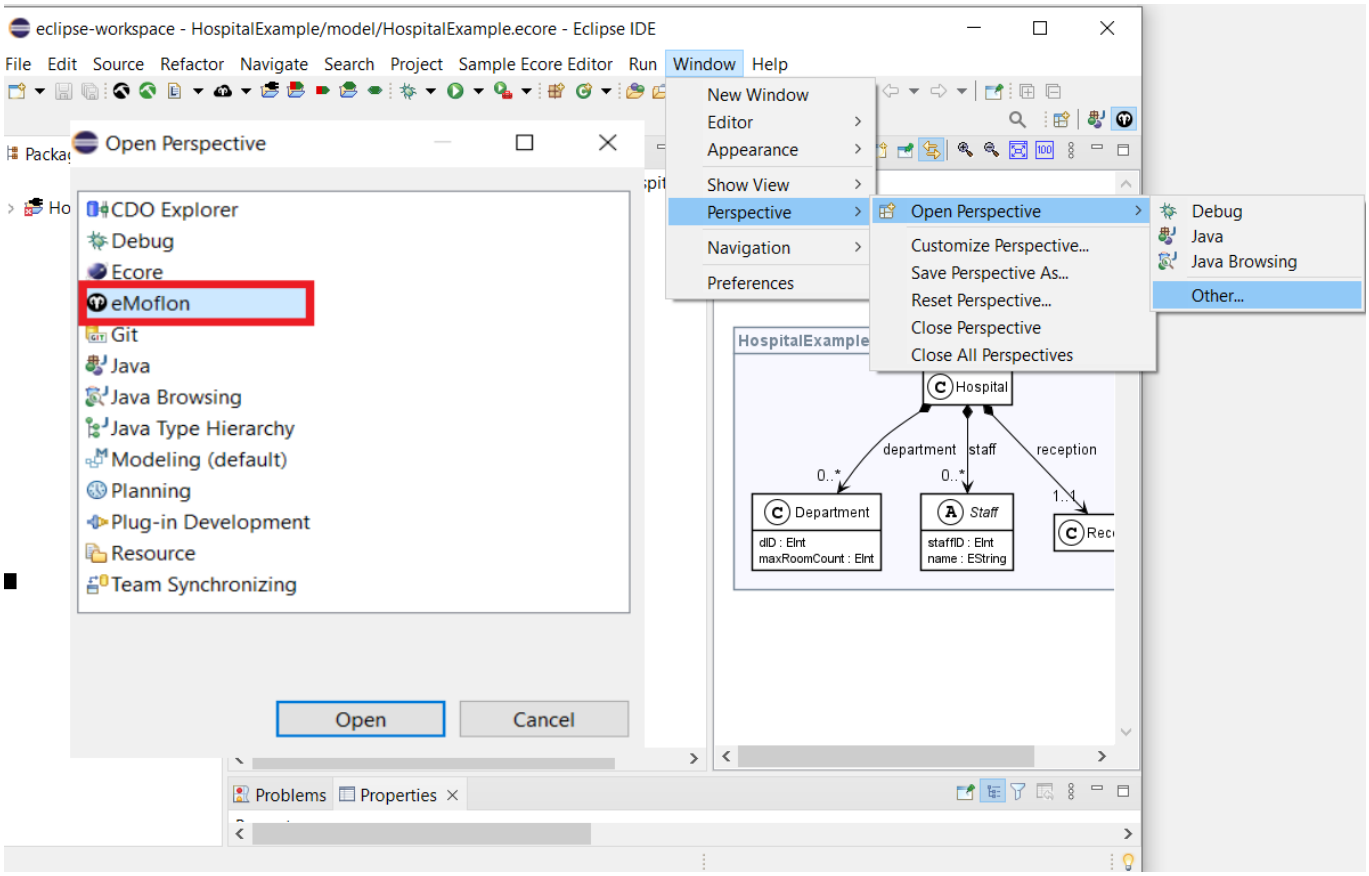


Figure 4: Add the eMoflon toolbar icon

[2] See Sprinkle, Rumpe et. Al 2014
[3] Graph Transformation in a Nutshell

Functions:

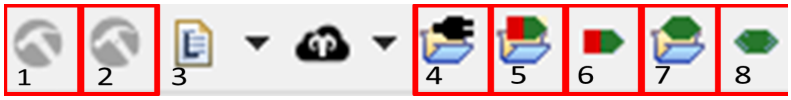


Figure 5: This is how the eMoflon perspective should look like

Let us take a look at the functions we will need throughout this tutorial:

1. Clicking this button will **build** the selected projects **fully**.
2. Clicking this button will **build** the selected projects **incrementally**.
3. Clicking this will show you the **logging configuration** of the eMoflon Console.
4. The folder with the plug **creates a new eMoflon project**.
5. The folder with the green and red arrow **creates a new graph transformation project**.
6. The green and red arrow **creates a simple graph transformation file**.
7. The folder with the green trapezoid is used for **creating triple grammar graph projects**.
8. The green trapezoid button **creates a single triple grammar graph file**.

3.3 Creating a new EMF project

First of all we have to create a new **metamodel**. Click on **button 4** to create a new **eMoflon EMF Project** and choose a new name for your project.
For this tutorial we are going to create a scenario which models a hospital and its administration. This is supposed to show you the basic functions of eMoflon::IBeX.
You should **stick to the naming conventions** we are using, since inconsistent names may lead to errors later on!

→ Please select *Generate default Ecore file*

Please give the project the name *HospitalExample* as shown in the screenshot below. After you have pressed the *finish* button a new file should appear in the model folder. If you cannot see it, try to refresh the folder.

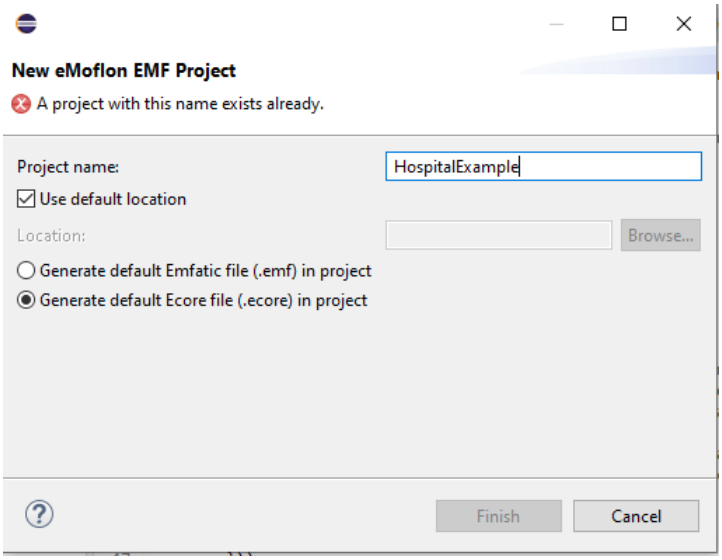


Figure 6: Creation of a new EMF project

3.4 Filling the model with content

The chart below provides an UML-based visualization of the finished model you will create throughout this part of the tutorial. If you are not sure whether your model is correct, you can use this as a comparison:

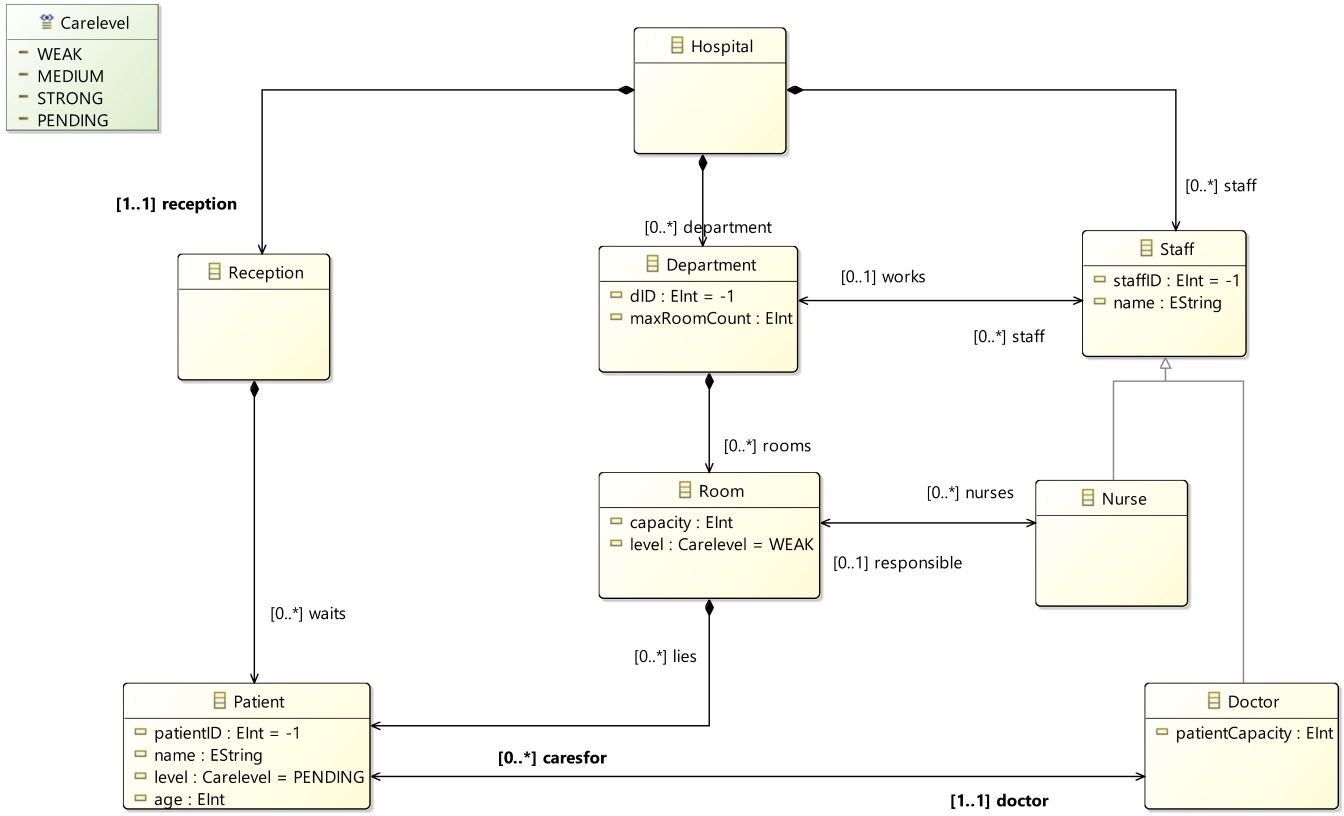


Figure 7: Complete model

Note:
To visualize your current model, left-click on *HospitalExample* and look at the **PlantUML tab** in the top right corner. This will look a bit different than this UML chart, but should be equal in regard to its content.

The first class we need for our tutorial is the hospital itself. To create the **Hospital class**, you need to open the **HospitalExample.ecore package** in the model folder of your package and **right-click** it to select **EClass** as a new child of the metamodel in the drop-down menu.

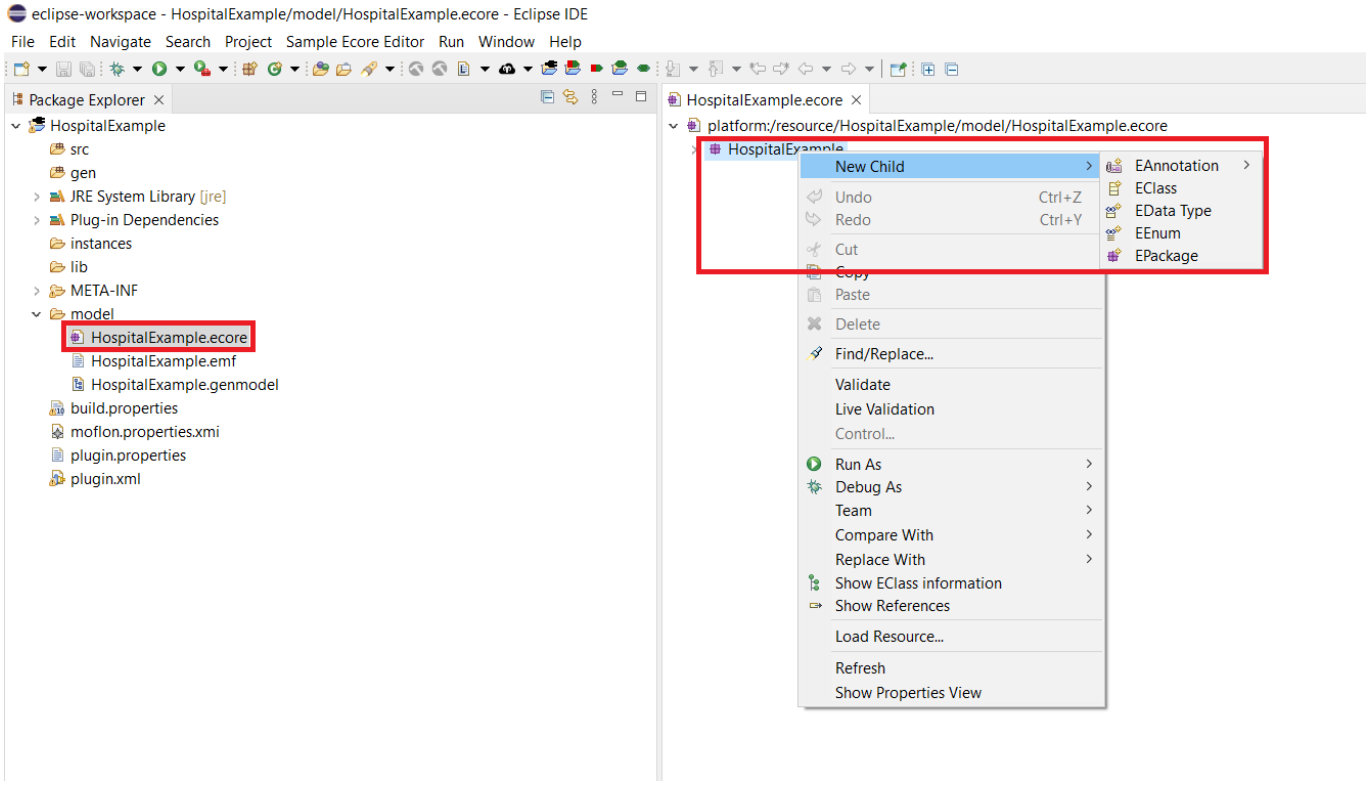


Figure 8: Create Hospital class as child of the metamodel

As you can see, it is also possible to create an **enumeration**, a **data type**, and **other packages** to split up your project.
You may have noticed that every option begins with a capital "E". This is merely a **naming convention** used in EMF and it will be also used for things such as variable types. For example, a variable of type Integer has to be defined as "EInt".

After you have created the Hospital, you can define its **properties** in the *properties* window. If you cannot see the properties window, right-click on the *Hospital* child you have just created and select *Show Properties View* on the bottom of the list or **double left-click** on the child.
Let us name the class accordingly by typing *Hospital* in the name field. All other fields can be left set to their default value.

To fill our Hospital, we start by adding a *Reception* class and a *Department* class. This is done in the same way as for *Hospital*.

After creating the classes, we have to connect them to our hospital. This can be done by creating a **new child** of type **EReference** in our hospital class. The properties of a **reference** offer a large variety of options you can configure but let's keep it simple for now.
To maintain comprehensibility and simplicity in your model **naming your reference** after the corresponding class or its purpose is recommended.
Hence, we chose the name *reception*.

Furthermore, we also need a **type** for our reference which can be defined in the **EType field**, where we select the type *Reception*.

Naturally, a hospital has only one reception, so we have to modify the **multiplicities** of our references as well. **Multiplicities** can be configured via the "Lower Bound" and "Upper Bound" options. By changing the **lower bound** value to **1** we ensure that our hospital has **at least one** reception. To avoid having **more than one** reception in the Hospital you have to set the value for the **upper bound** to **1** as well. If you want to model an **N-multiplicity** you have to set the **upper bound** to **-1**.

Please note that it is also possible to change the **boundaries of attributes**. However, meddling with the attribute boundaries can

easily mess up your model. Hence, we advise you to leave the boundaries of attributes as they are by default.

The model depicts the relation between the Hospital and the Department class with a **rhomb** this means the relationship is a **containment**. You have to set the value of the "Containment" field to true.

Please create the **department reference** analogously. This time the reference has an N-multiplicity, which can be modeled like shown above. Note that you can get the correct multiplicities and the correct type of an attribute by looking at Figure 7.

A short note on **containment objects**: If you delete a container object, e.g., the Hospital in our case, you will also delete every object that is contained by the Hospital. In addition, an object can only be contained by one container, once a new containment edge is assigned to an object the old containment edge will be deleted.

For the next step, we want to assign attributes such as department numbers and a maximum number of rooms for our departments, to our Department class. Add a **new child** of the type **EAttribute** to the departments, name it "dID", and give it the type "EInt" in the EType field. Since we want every department to have an ID, we modify the **multiplicities** in the same manner as our previous references. Also set the field "ID" to true. This tells the model whether the value of this attribute identifies the object uniquely in the containing resource.
Create the Attribute "maxRoomCount" analogously except it won't be used as an **ID**.

Before we continue we want to create something in our model that symbolizes the different care levels in a hospital. Suitable for this is an **enumeration**. The Carelevel enumeration can be created as a child of the Hospital package. The possible care levels should be WEAK, MEDIUM, STRONG and PENDING which can be added as **children of the enumeration** of type **EEnum literals**.

Naturally we need patients in our hospital as well. **Patients** have the following attributes: A name, a patientID, and a level representing the **care level** assigned to a patient. Please create the **Patient class** accordingly.

When a patient arrives at the hospital he will be waiting in the reception until he has been assigned to a room and diagnosed with necessary treatment. For the first condition, we need a **reference in the reception class** to the patient and while the patient is waiting in the reception he has not been diagnosed yet. Hence, we want to set the **default value literal** of the **level attribute** to PENDING.

The attribute maxRoomCount already indicates the need for a **room class**. Regarding the **attributes** of the rooms, we need a **capacity** and a **care level** to indicate the necessary medication for the patients and the room. For the latter you can select our freshly created enumeration Carelevel as its **attribute type**. The **default value** for the level should be WEAK. Please create the room class accordingly.

The rooms also need a **connection to the departments** they are assigned to and **contain** multiple patients. Please add **references** in the **Department class** to model this. You can get their characteristics from the final model in fig 7.

After creating the infrastructure for our hospital, we need the people, which are required to keep a hospital running and of course the patients. Starting with the Staff, create an **abstract class** with the **attributes** staffID of the type "EInt" and the name of the staff member which should be an "EString". You can set the class to abstract by selecting true for the field Abstract in the properties window.
Similar to the concept of inheritance in Java it is not possible to instantiate such a class. However, attributes and variables defined in an abstract class will be inherited by a subclass of this abstract class.

The staff should be **contained in the hospital** and needs a **reference** to the department the staff member is working in. The staff needs to be differentiated into nurses and doctors. Create a new **Doctor class** and change the ESupertype in the properties window to Staff, which allows the Doctor class to **inherit the references and attributes** of the **abstract Staff class**. This can be done by pressing Add and confirming in the windows that will pop up. Additionally, we want to add the **attribute** patientCapacity to define the number of patients a doctor can care for and a relationship to the patients he oversees.
For the Nurse class, we want a reference to the rooms a nurse is responsible for.
To model this responsibility we want a **bidirectional reference** between the **Doctor class** and the **Patient class**. To achieve bidirectional references please create a **reference** in the **patient class** to the doctor called doctor. On the other hand create a **reference** to the patient class in the **doctor class** called caresFor. Multiplicities can be read from fig 7. Now for the latter select the reference caresfor:Patient in the **EOpposite property**. This should be done automatically but if not, go to the **doctor class** and assign the doctor:Doctor reference as an **EOpposite** of the created reference as well.
Note: This works only if you have set the type correctly and if the opposite reference has been created already.

Please follow the same procedure to make the existing **reference between the nurse class and the room class bidirectional**.

Also add a **bidirectional reference** between the **staff** and the **department** they are working in.

If you made it this far your hospital meta-model should be completed and the visualization in PlantUML should look like this:

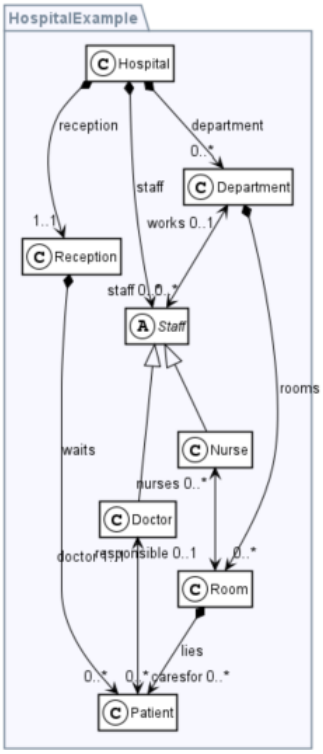


Figure 9: Goal model in PlantUML

Please also compare your **Multiplicities**, **attributes**, **default literal values**, **containment values** and all **names** to fig 7. Differences can result in errors later on.

Congratulations on completing the first part of this tutorial!

4 Graph transformation with eMoflon

The next part of our Tutorial is dedicated to the creation of a graph transformation rule set which allows us to create a concrete model instance based on constraints and multiplicities we defined in this metamodel. The goal of the ruleset we want to define in the next chapter is to achieve the defined structure and behavior of operations in our hospital while creating a certain kind of dynamic through the application of rules. This might sound a bit confusing for now, but no worries we will explain it to you step by step.

4.1 Rules and patterns introduction

Now that you are good to go, it is time to get familiar with the two most common constructs for eMoflon Graph Transformations: **The rules and patterns**. For a more detailed description you can read the second chapter of this essay[4].

Graph pattern:
A Graph pattern describes certain structures that must be present in a given instance graph. A structure consists of an arangement of nodes with specific types and specific connecting edges. A pattern could also be made up by a single node of a specified type.

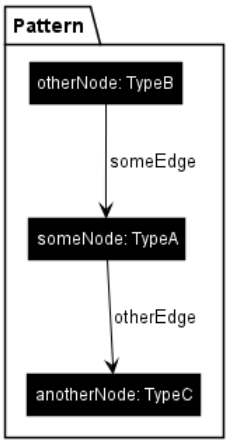


Figure 10: Pattern example in PlantUML

Consequently, a graph pattern matcher will find all sub-structures in the instance graph that match all structural constraints of a given pattern.

[4] Graph Transformation in a Nutshell

Graph transformation rule:

A graph transformation rule consists of a so-called left-hand side (LHS) and right-hand side (RHS). The former describes certain preconditions, like a graph pattern that must be present in a given instance graph, otherwise a rule is not applicable. If a match of the LHS is found, the rule can be applied. When a rule is applied, the graph is changed in such a way, that the rule's RHS is satisfied. More precisely, any graph pattern node and edge that is present in the LHS, but is not present in the RHS, is deleted. In turn, any node that is not present in the LHS, but is present in the RHS, is created.

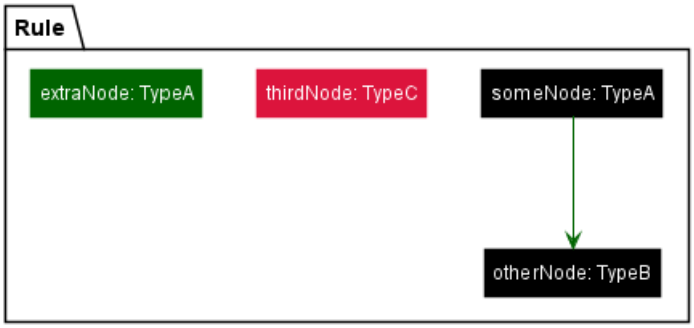


Figure 11: A rule example in PlantUML

4.2 Creating a graph transformation project

Let us get started by creating a new eMoflon Graph Transformation project. Please click on the button 5 with the **red and green arrow in the folder**.

→ Please write "HospitalTransformRules" as the name → Press "Finish"

You will notice the Rules.gt file which has been generated automatically. In this file, you will be defining the rules to construct our Hospital.
Start with deleting the automatically generated contents in the ".gt" file except for the Ecore import.
The next thing you want to do is to **import the metamodel** you have created in the previous section. Just type import and use code completion (Ctrl+Space) to obtain the suggested URI to your metamodel HospitalExample.

Afterwards your Rules.gt file should be empty exept these two imports:

```
1  import "http://www.eclipse.org/emf/2002/Ecore"
2  import "platform:/resource/HospitalExample/model/HospitalExample.ecore"
```

You might have noticed that the project has some **compilation errors**. To solve the issues do the following:

Open "META-INF/MANIFEST.MF" file → dependencies → Add "HospitalExample" to the Required Plug-ins menu.

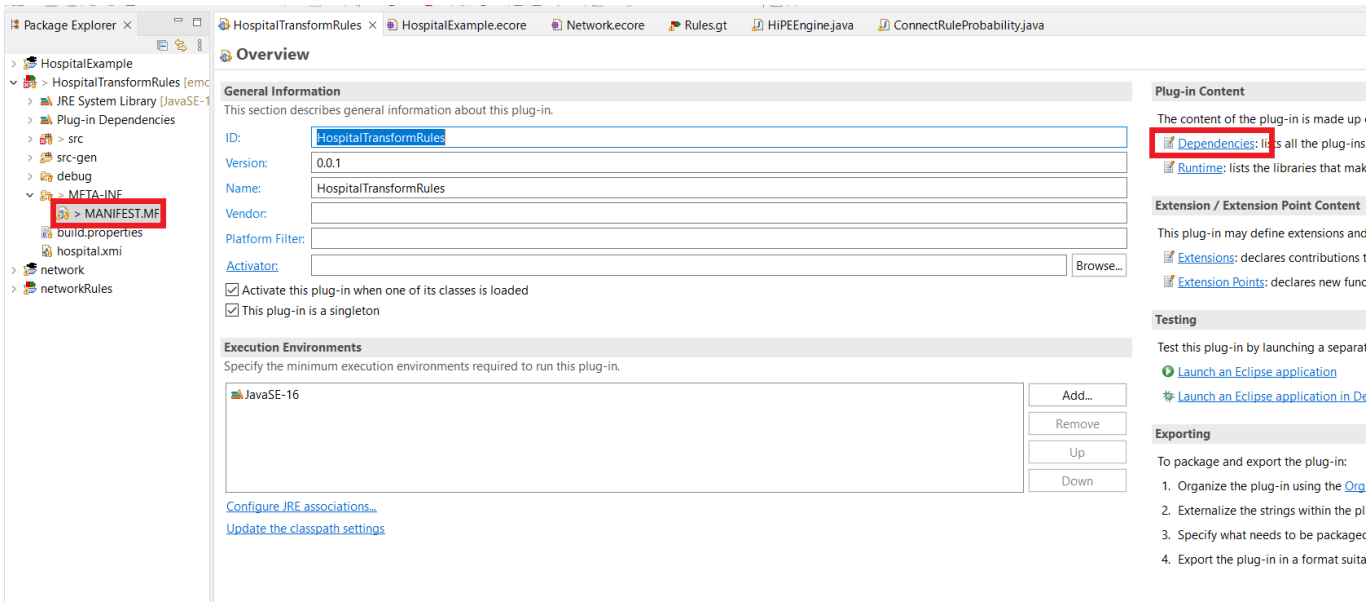


Figure 12: Manifest.MF

The errors should be fixed now, and your project should compile properly. If the **errors persist** check if the Build automatically option is turned on in the project drop-down menu of your eclipse. **Refreshing and cleaning** all the projects are the remaining options if you are still encountering errors. Should there still be errors remaining in the manifest, check the code of the manifest for the **Export-Package** and **Import-Package** sections and check if the names are written properly. Sometimes the commas and names get mixed up or put in the wrong line. After fixing that you should see no more errors.

4.3 Rules and patterns in eMoflon

Anyhow, let us get started by writing the very first rule for our hospital rule set. The complete hospital ruleset will allow us to create a concrete hospital model instance according to the specifications of our metamodel from chapter one. Graph transformation rules are perfectly suited for this task since they allow us to consider constraints and requirements directly.

Rules:
For a rule, we need the **keyword rule** and a **name for the rule**. As previously mentioned, you should **name the rule after its purpose** to keep the project comprehensive.

As a brief introduction to our syntax, we would like to present the two most important operators:

- [+]
- [–]
- [=]

Any Element that is prepended with a [+] is **created** when the rule is applied. This is indicated by the green syntax color.

In turn, the [–] operator will lead to a **deletion** of the denoted element. If an element is deleted by the application of a rule the syntax will be colored red.

If the [=] operators is used, the element is interpreted as **context** and **colored black**.

Every element that is **not denoted** with an operator describes the **LHS** of the rule, which means that these elements must be present in a given instance graph.
In practical terms, a valid match of the LHS must be present before the rule can be applied.
By using the [+] operator before the hospital graph pattern node in the hospital rule, we enable the rule to create a new hospital Object upon application. Since this rule has no black or red context elements, it can **always** be applied.

```
1    rule hospital() {
2        [+] hospital: Hospital
3    };
```

Patterns:
If we want to check whether a hospital object is present within a given model, we can create a pattern that finds a hospital. Here we create the pattern findHospital() that defines a node named hospital of the type Hospital. Since no elements are annotated with any of the operators, findHospital() only contains black elements, i.e., context and will not lead to the creation or deletion of any objects. Therefore, findHospital() is not a rule, but a pattern, which solely will instruct the pattern matching engine to **find matches of this pattern**.

```
4    pattern findHospital() {
5        [=] hospital: Hospital
6    };
```

Please create rule and pattern in the Construction.gtl file of your HospitalTransformRules project.

Note on patterns:
Just like shown in this example, all the other patterns for all the other types should be created by you on your own.

Note on the visualization of PlantUML:
When you click on the freshly created hospital rule you should see the visualization in the PlantUML window on the right. It shows you the rule and the way the rule is defined as a graph structure. The black boxes are pattern nodes that represent context, as a consequence these nodes must be present in an instance graph. Analogously, black arrows define context edges between nodes, that must be present in the same instance graph. The green elements are the ones we are creating with a rule application.

4.4 Application Condition

Since our model requires only one hospital node, we should add a so-called **application condition** to prevent multiple applications of the hospital rule.

Please note there are two types of conditions. You can either forbid the presence of graph structures attached to pattern matches or enforce them. An application condition that forbids the presence of such structures is called a **negative application condition (NAC)** while a condition that enforces the presence of such structures is called a **positive application condition (PAC)**.
The utilization of a construct consisting of a condition and a pattern is called a **support pattern invocation** while the pattern itself is called a **support pattern**.

Back to the example. To extend our hospital rule with a negative application condition, add a forbid inside our hospital rule and insert findHospital into the round brackets as the name of pattern to be forbidden.
With our condition, we want to forbid the application of the rule if our findHopistal pattern finds the structure hospital:Hospital.

```
1    rule hospital() {
2        [+] hospital: Hospital
3        forbid(findHospital) []
4    };
```

After creating a rule for the construction of a hospital node we need to fill our hospital with the same objects of our meta-model. So, let us continue with a **reception**. We start by creating the rule reception and adding a reception node of type reception.

In contrast to the previous rule, we also have to **link our reception to the hospital**. To create such a reference, we require the hospital node and add an edge from the hospital node to the reception node as we have done previously in the tutorial.
You need to use the [+] operator once again but now we continue with a "**-**". This marks the created structure as an edge. The **"-**" is followed by the **type of the edge** and with **"->"** we are pointing towards the **node we want to connect**.
The syntax should look like depicted below and the crucial part for the creation of an edge is shown in **line 9**:
Additionally we now need to add some code into the brackets after the forbid() statement. There we need to specify which node of a pattern **maps** to a corresponding node of a second pattern. We do this by adding **hospital=hospital** into the brackets.

```
5    rule reception() {
6        [=] hospital: Hospital {
7            [+] -reception -> reception
8            }
9        [+] reception: Reception
10       forbid(findReception) [hospital=hospital]
11   };
```

Please implement both rules above this chapter in your rules file.

An example for an application of enforce would be:

```
1    rule releasePatient(patientID:EInt) {
2        ...
3        enforce(findPatientInRoom)[patient=patient]
7    };
```

4.5 Parameters and attribute constraints

The focus of this chapter will be the handling of parameters in rules and attribute constraints. Regarding the infrastructure of our hospital, we have to address the **departments** and **rooms** in our next rules.

You should be able to create the **department rule and the branch** from the hospital node yourself. We will expand them now.

After creating these two rules we want to introduce some new syntax. Remember that both classes have attributes that need to be assigned. To assign **attributes** you need to use the **operator "."**. Pressing the **auto-completion** behind the "." should suggest the **attributes defined in the meta-model**.
By typing the **operator ":="** you can **assign a value** to the attribute, e.g. the simplest assignment would be a static one.
As an **unrelated** example for this, you could assign the ID 1 to the department you create by writing:

```
1    .dID:=1.
```

In reality we want to hand over a **parameter**, not unlike it is done in Java methods.
We need to define the parameters the rule expects in the **round brackets** after the rule name and type **parameter.**paramName; instead of the static assignment. The same has to be done for maxRoomCount:

```
1   rule department(dID: EInt, maxRoomCount: EInt){
2       [=] hospital: Hospital {
3           [+] -department -> department
4       }
5       [+] department: Department {
6           .dID:=parameter.dID;
7           .maxRoomCount:=parameter.maxRoomCount;
8       }
9   };
```

```
10   rule room(cap: EInt, carelvl: Carelevel) {
11       [=] hospital: Hospital {
12           [=] -department -> department
13       }
14       [=] department: Department {
15           [+] -rooms -> room
16       }
17       [+] room: Room {
18           .capacity:= parameter.cap;
19           .level:= parameter.carelvl;
21       }
22       [#]department.maxRoomCount>count(findRoomInDepartment)[department=department];
23   };
```

Please implement both rules above this chapter in your rules file.

Attribute constraints:
For the room rule, we want to add a **constraint** that limits the creation of rooms in a certain department because we have previously limited the maximum number of rooms a department can contain. To specify a so-called **attribute constraint (or attribute condition)** you have to use the **"[#]"** operator. Such an attribute constraint can be defined at any point in the pattern. For this constraint, we want to get the maxRoomCount of the given Department and check if the limit of rooms in this department is not exceeded.
To **access an attribute value** of a specific pattern node, we provide a syntax that is remarkably similar to accessing attributes in java, namely using the "**.**" operator.
For example, we access the parameter of the department node via [#]department.maxRoomCount. The count(findRoomInDepartment) instruction **counts the matches** for a certain pattern, in our case for the findRoomInDepartment pattern which we have to create in the next step.

Support pattern invocations with mapping:
Our goal for this pattern is to count the number of rooms which are assigned to a department. However, we want to avoid finding matching rooms connected to any department node since this would lead to n·m matches for n rooms and m departments. Consequently, we want to count the matches for the rooms assigned to a **single** department node. In other words we want the **graph pattern matcher** to only find matches of our **pattern** findRoomInDepartment() that involve the **specific department node** we are currently referencing in our **rule** room. To accomplish this we have to create a **mapping** between the nodes in the rule and the elements in the support pattern. eMoflon will do exactly that if we **assign identical names** for the nodes in rules and patterns.

Generally, patterns which are used like this, are invocated from the **'perspective'** of the currently evaluated element. eMoflon will try to resolve identifiers from the pattern as **references** to structures from the surrounding element. If matching names are found the corresponding element will be regarded as a placeholder for any node, reference or else of matching type. These kinds of pattern invocations are also called **support pattern invocations** just like **NACs** and **PACs**.

You can note the difference when comparing these slightly different patterns:

```
1       pattern findRoomsInAnyDepartment(){
2           [=] otherdepartment:Department {
3               [=] -rooms->otherroom
4           }
5           [=] otherroom: Room
6       };
```

When called as a support pattern **in context of the rule** room this will find $n \times m$ matches since it doesn't reference any structure from our rule and consequently would be valid for any rooms in any department node.

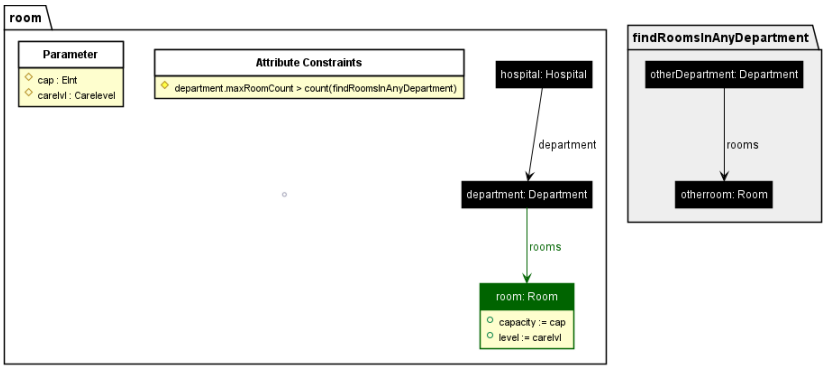


Figure 13: PlantUML visualization of findRoomsInAnyDepartment

```
7       pattern findRoomsInDepartment(){
8           [=] department:Department {
9               [=] -rooms->otherroom
10          }
11          [=] otherroom: Room
12      };
```

When called as a support pattern **in context of the rule** room this will find this pattern will find all rooms for the **specific** department node referenced in our rule.

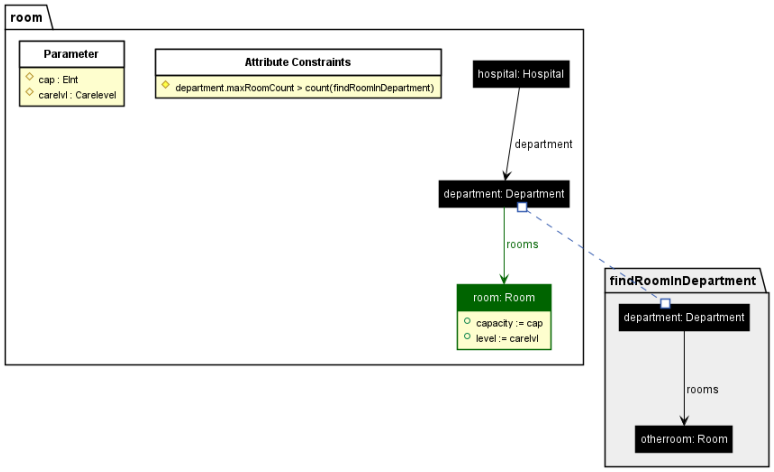


Figure 14: PlantUML visualization of findRoomsInDepartment

```
13      pattern findRoomInDepartment(){
14          [=] department:Department {
15              [=] -rooms->room
16          }
17          [=] room: Room
18      };
```

When called like the others this pattern will find exactly find 1 match. Because we are not only referencing the department node from our rule but the specific room we have just created as well.

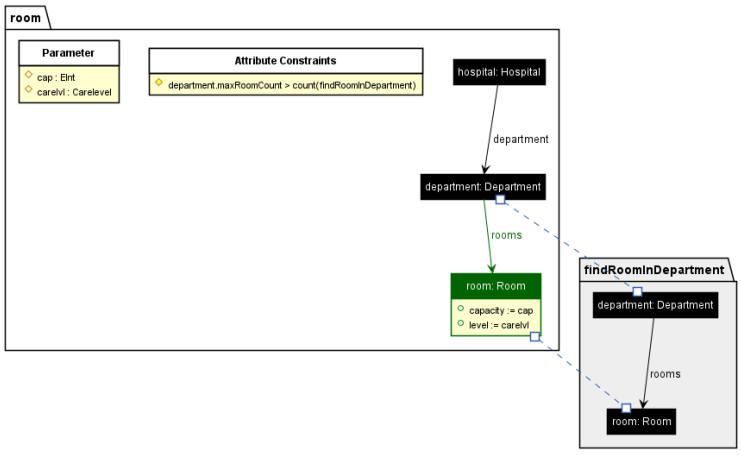


Figure 15: PlantUML visualization of findRoomInDepartment

Note that you can look at the PlantUML visualization of rules and pattern by clicking on them in the Construction.gtl and looking at the PlantUML view.

Please use the findRoomInDepartment pattern for our room rule.

Disclamer control flow:
While attribute constraints and conditions can be used to model complex application conditions for rules and patterns it's not possible to model **control flow** directly in eMoflon. For example you can not specify on which of all fitting elements a rule will be applied or in what order new elements will be created.
This has to be done in the actual code implementation.

4.6 Finishing the graph transformation ruleset

After we have created the ruleset for the infrastructure of the hospital, we need rules to define the behavior of the people in the hospital. In our case, we have the following people types in our hospital: the **patients**, the **doctors**, and the **nurses**.
Note: Repeatedly pressing the F5 key during the creation of a GT or TGG model helps preventing errors, since the automatic build mode sometimes doesn´t work correctly.

Patients:
For the patients, we assume that a patient appears in the reception and waits there until he has been assigned to a room. Naturally, a patient has a name and will be assigned a patientID and a carelevel which requires a certain kind of treatment.

```
1   rule patient(name: EString, patientId: EInt, level: Carelevel) {
2       [=] hospital: Hospital {
3           [=] -reception -> reception
4       }
5       [=] reception: Reception {
6           [+] -waits -> patient
7       }
8       [+] patient: Patient {
9           .name:=parameter.name;
10          .patientID:=parameter.patientId;
11          .level:=parameter.level;
12       }
13  };
```

Another important use case that one can think of is the dismissal of a patient, which can be modeled with a the following rule.
For this rule, we want to **remove** a patient with a given patientID by using the **operator** [–].
To keep our model clean, we also want to **avoid dangling edges**. The EMF framework removes those dangling edges automatically, but we can also remove the edges manually. An example for the edges we remove in the rule releasePatient would be in **lines 84 and 87**.

```
80  rule releasePatient(patientID:EInt){
81      [–] patient: Patient{
82      }
83      [=] room: Room{
84          [–] -lies -> patient
85      }
86      [=] doctor: Doctor{
87          [–] -caresfor -> patient
88      }
89      [#]patient.patientID==parameter.patientID;
90  };
```

Staff:
According to our metamodel, the class Staff is **abstract**. Every staff member is assigned to a department and has a name and a staffID as parameters to identify staff members.
A note on abstract rules: Just like it is not possible to instantiate an abstract class in Java it is **not possible to apply an abstract rule**. However, elements such as nodes, edges, and parameters will be inherited by a refining rule.

```
15  abstract rule staff(name: EString, staffID: EInt) {
16      [=] hospital: Hospital {
17          [+] -staff -> staff
18          [=] -department -> department
19      }
20      [=] department : Department
23      [+] staff: Staff {
24          .staffID:=parameter.staffID;
25          .name:=parameter.name;
26      }
27  };
```

Doctors:
Since the doctor rule is a specification of the staff rule, we can **refine** this rule to save some writing effort and improve readability. This can be done by adding the **keyword** refines and the rule which should be refined to the head of your rule. Furthermore we need to add a **@refines+** before each section where we refine something from staff. The **+** denotes the binding of the object **staff**. In the second refines statement you see that the binding of dpartment is **=** so we use **@refines=** instead.
To make things more interesting we could add another condition to our doctor rule. The condition only assigns a doctor to a department if the given department has no doctor who is already responsible for the department. Hence, we need a fitting pattern in addition to our doctor's rule.
This pattern works analogously to the pattern we defined for the rooms.

```
28  rule doctor(capacity: EInt) refines staff {
29      @refines+ staff.staff
30      [+] staff: Doctor {
31          .patientCapacity:=parameter.capacity;
32      }
33      @refines= staff.department
34      [=] department : Department {
35          [+] –staff -> staff
36      }
37  forbid(findDoctorInDepartment)[department=department]
38  };

39  pattern findDoctorInDepartment() {
40      [=] doctor: Doctor
41      [=] department : Department {
42          [=] –staff -> doctor
43      }
44  };
```

Nurse:
eMoflon is capable of distinguishing between the two different types of staff members. Hence, we can name the nodes for both types of staff.
For the nurse rule, we want to combine the creation of a nurse node with the assignment to a room. This assignment requires the creation of an edge to a room, as well as the creation of a staff node of the type nurse and a staff edge to the departments.
In our case, we want to assign only one nurse per room, which requires another condition and a pattern to find nurses who are already assigned to a room.

```
45  rule assignNurseToRoom(capacity: EInt) refines staff {
46      @refines+ staff.staff
47      [+] staff: Nurse {
48          [+] –responsible –> room:=parameter.capacity;
49      }
50      @refines= staff.department
51       [=] department : Department {
52          [=] –rooms –> room
53          [+] -staff –> staff
54      }
55  forbid(findNurseInRoom)[room=room]
56  };

57  pattern findNurseInRoom() {
58      [=] nurse: Nurse
59          [=] –responsible –> room
60      }
61      [=] room: Room
62  };
```

Patterns with conditions:
Conditions can also combined with patterns to allow more complex matching criteria.
This can be useful if we are trying to find out if there is a room that has no responsible nurse:

```
63   pattern findRoomWithoutNurse(){
64      [=] room: Room
65      forbid(findNurseInRoome)[room=room]
66  };
```

To finish our hospital ruleset, we need one last rule which assigns the patients to their rooms. This rule will be the most complex so far and there is a lot we need to take into account.
First, we want to assign the patient to a room by **adding** a lies edge to a room and **remove** the waits edge from the reception.
But we have to keep in mind that we can only assign a patient to a room if the **room is not full**, and we want to assign a patient to a doctor, which is also only possible if the **patient limit** of the doctor is **not exceeded**. Hence, we need **two attribute constraints** for our patient rule and a **condition** that forbids the assignment to room if the patient already has a doctor.

```
92  rule assignPatientToRoom() {
93      [=] patient: Patient {
94      }
95      [=] room: Room {
96          [+] –lies –> patient
97      }
98      [#] room.capacity>count(findPatientInRoom)
99      [=] doctor: Doctor {
100         [+] –caresfor -> patient
101     }
102     [#] doctor.patientCapacity>count(findOccupiedDoc)
103     [=] hospital: Hospital {
104         [=] –reception –> reception
105         [=] –department –> department
106     }
107     [=] department: Department {
108         [=] –rooms –> room
109     }
110     [=] reception: Reception {
111         [–] -waits -> patient
112     }
113     forbid(findOccupiedDoc)[patient=patient]
114 };

116 pattern findOccupiedDoc() {
117     [=] doctor: Doctor {
118     [=] patient: Patient
119         [=] –doctor –> doctor
121     }
122 };
```

Please implement all rules and patterns of this chapter in your Construction file.

4.7 Java implementation of the ruleset

So far, we have only written the rules without applying them. This will be our next task.
To understand how we implement our graph transformation ruleset we should take a look at the creation of a new model instance. For this step open the HospitalRules.java in the hospital.util package, please.

Let us inspect HospitalRules.java a bit more closely since there is much of interest going on:

```
1   public class HospitalRules {
2   ...
3       private Random rnd;
4       public HospitalTransformRulesAPI api;
5       public HospitalRules(final long rndSeed) {
6           rnd = new Random(rndSeed);
7           api = new HospitalValidator().initAPI();
8           try {
9               api.addModel(URI.createURI("hospital.xmi"));
10          }
11          catch (Exception e) {
12              e.printStackTrace();
13          }
14          api.initializeEngine();
15      }
16      public static void main(String[] args) throws IOException {
17          HospitalRules hospitalrules = new HospitalRules("someSeed".hashCode());
18          hospitalrules.createHospital();
19          hospitalrules.validateHospital();
20          hospitalrules.api.terminate();
21  }
22  public void createHospital() {
23      api.hospital().applyAny(true);
24      api.reception().applyAny(true);
25      for(int i=0; i<4; i++) {
26          api.department(i+2, 4).applyAny(true);
27      }
28      for(int i=0; i<16; i++) {
29          api.room(4, Carelevel.get(rnd.nextInt(3))).applyAny(true);
30      }
    ...
    }
```

Initialization:
The first important thing we want to look at is in **line 4** where we define and initialize the **api variable** for our hospital transformation rules.
The API command is used to access and apply our rules and patterns.

Rule application:
Let us take a closer look at the createHospital method starting at **line 22**.
As you can see we are accessing our rules via the variable name **api.<rulename>.applyAny(true)**.
Then we construct our hospital subsequently by applying all the rules necessary for our hospital.

The application of the hospital rule in **line 28** creates **one hospital object** according to our rule. Keep in mind we can only apply rules and not patterns. As previously mentioned we can access the eMoflon:IBeX functionalities via the **.api command**.

Writing .api and pressing the hotkeys for auto-completion will show you the extensive list of commands we can use for our graph transformation project. In the appendix for this part of the tutorial, you can find a list with a short explanation of the respective command.

Be aware that a rule can only be applied if the precondition is met, i.e., matches to its left-hand side exist.
For example, if we switch the application order of the hospital and reception rule, we would not be able to create a reception because we are missing the context of the hospital node and the corresponding connection.
Here is a faulty example if you changed the order of the rule application:

```
api.reception().applyAny(true);
api.hospital().applyAny(true);
```

This order would have **severe consequences** for our hospital since the rules which require a reception such as the patients would not be applied as well as the reception itself.

With rule applications below the reception rule, we will create people of the type of patient, doctor and nurse for our hospital as well as the departments and the rooms for our hospital.

Printing results:
A basic way to check whether our rule applications had the desired effect is, to display relevant match counts on the console, using the System.out.println(...) **method**. This is what we are doing in the validateHospital() **method** where we count the matches we can find for a given pattern and print them to the console.

```
long patientsInHospital = api.findPatient().countMatches(false);
long patientsInRoom = api.findPatientInRoom().countMatches(false);

System.out.println(patientsInHospital + " Patients are in the hospital right now and " + patientsInRoom + " patients are in a room");
```

Testing:
You can run the java application by **right-clicking** on the HospitalRule.java and selecting the Run as Java Application option. If you look at the output in the console, and it should look like this:

```
One instance of a hospital has been created
One instance of a reception has been created
At least one department instance has been created
32 nurses are in the hospital right now and 32 nurses are busy
8 doctors are in the hospital right now and 8 doctors are busy
At least one patient is in the hospital
22 Patients are in the hospital right now and 22 patients are in a room
```

If you get a console output like this one without any errors your Java code is most likely correct, but this does not guarantee that the model instance we have created is the one you intended to create.
For example, if some rules could not have been applied it would not be presented as a compilation error. Carefully examining the output for the validation of our hospital is the key to find misconceptions in our ruleset.

4.8 Arithmetic Extension

eMoflon also supports a lot of mathematical calculations in the form of arithmetic expressions.
They are very useful when assigning attribute values or defining attribute constrains that involve some sort of calculation.

The arithmetic extension allows a variety of arithmetic functions which are depicted in the table below. With these functions, it is possible to write all sorts of equations.
eMoflon will automatically check if given input parameters are within domains of definitions for specific functions. If not an error will be displayed.

Here is a list of supported operations, their respective domains of definition are of course equal to their mathematical counterparts:

- Basic operators +, -, *, /, %
- Exponents
- Exponential function
- Logarithmic functions
- Trigonometric functions cosinus, sinus, tangens
- Absolute value function

You can find the corresponding syntax in the **appendix**.

4.9 Number Generation

Another useful feature of the stochastic distribution extension is the value generation function. With this feature, we can assign a randomly generated number of the specific distribution to an attribute. If the connect rule is expanded to designate a random flag to every connection, then the rule can be written as:

```
1  rule createRandomPatient {
2      [+] patient: Patient{
3          .name :="John Doe"
4          .patientID := N(100,15)
5          .level :=0
6          .age := U(0,99)
7      }
8  }
```

By assigning U(0,99) to the age attribute it will get a uniform distributed value between 0 and 99. The new extension also supports so-called range assignments to the stochastic functions.
The user may define whether the generated value should be only positive, only negative, or both by adding a plus for positive values and adding a minus for negative values in front of the brackets of the distribution.

The example above depicts an example with positive values. With this example rule, it is possible to generate a positive value for the bandwidth attribute that is distributed as $b \sim N(100, 15)$:

5 Appendix Graph Transformations

5.1 Concepts

Patterns and Rules:
The keywords **pattern** and **rule** are used to distinguish graph structures containing only context and graph structures that create, delete or change the model.
Patterns only contain the context while rules contain context and change the model.

Node naming conventions:
Nodes should be named in **lowerCamelCase** and may contain small and capital letters and numbers. They may not contain underscores except as a first letter. Node names starting with an underscore are local nodes, i.e., they would not appear in matches. Nodes are visualized as boxes in PlantUML which provide context and can be created and deleted via the application of graph transformation rules.

Edge naming conventions:
Edges also should have names that describe the type of connection between the nodes for reasons of traceability.

Parameters:
Parameters are a way to pass values to attribute assignments and conditions at runtime. They must be declared in the signature. Note that parameters can only have **primitive data types** such as EInt, EDouble, EChar, EString, EBoolean, EShort, ELong, EByte.

Attribute Assignments and Conditions:
Via an attribute assignment, you can set new values for an attribute or add condition filters that can be applied to already existing attributes.
Supported values are:

- constants
- enum values (type enums. for auto-completion)
- parameters (type parameter. for auto-completion)
- the attribute value of another attribute
  (given by the node and attribute name <typeName>.<attributeName>)

Pattern Refinement:

Pattern refinement is a modular concept that allows nodes to inherit traits from super patterns to avoid writing the same graph structures repeatedly. All nodes from a pattern and its super patterns are combined to one pattern shown in the PlantUML visualization. This process is called flattening.
Note: Pattern refinement only includes graph structures, not the conditions of super patterns. The pattern refinement hierarchy must be defined such that the refinement hierarchy does **not contain cycles**, i.e., a rule is not allowed to refine itself directly or indirectly. Abstract patterns and rules exist only to be used in pattern refinement. Since they cannot be applied directly, they are not available in the generated API.

Conditions and Mapping:

Patterns may specify conditions to be used as an additional filter for matches. Conditions are used as application conditions for a rule. All nodes in patterns used in conditions are mapped to nodes with the same name in the pattern for which the application condition is specified. This mapping is depicted in the visualization of the pattern.

Patterns used in application conditions using the keywords enforce or forbid **cannot have parameters and may contain at most one application condition**. This restriction is made due to constraints when transforming the pattern specification into a pattern network for a pattern matching engine.

The API:

The API class is a factory for patterns and rules: For each pattern and rule, there is a method to create a new instance of a pattern or rule. Note that the returned instances are no singletons which means that you can create instances of the same pattern with different parameterization. If the pattern or rule has parameters, those must be initialized during creation.

5.2 Syntax and Documentation

5.2.1 API methods

Return Type	Signature	Description
void	updateMatches()	Triggers an incremental update of the matches
Map<M>	getAllPatterns()	Returns all the Rules and Patterns of the model that do need a parameter that has to be initialized with
PushoutApproach	getPushoutApproach()	Returns the pushout approach which would be used for the rule application. If the pushout was not set explicitly, the default pushout approach of the API is used. This is "SPO" but can be changed for the whole API as well
PushoutApproach	setPushoutApproach (PushoutApproach pushoutApproach)	Does the same as setDPO() and setSPO() but accepts an enum as input
PushoutApproach	setDPO()	Sets the pushout approach to DPO (Double Pushout). This means that a rule is only applicable if the deleted nodes do not leave dangling edges, i.e., if adjacent edges are not deleted with the node
PushoutApproach	setSPO()	Sets the pushout approach to SPO (Single Pushout). In contrast to DPO, SPO deletes adjacent edges implicitly as well
PushoutApproach	getDefaultPushoutApproach()	Returns the default pushout approach
void	terminate()	Terminates the interpreter
boolean	getTotalSystemActivity(boolean doUpdate)	Helper method for the Gillespie algorithm; counts all the possible matches for rules in the graph that have a static probability
ResourceSet	getModel()	Returns the resource that contains all models
boolean	applyGillespie(boolean doUpdate)	Applies a rule to the graph after the Gillespie algorithm but only rules that do not have parameters are counted
double	getGillespieProbability (GraphTransformationRule <?,?> rule, Boolean doUpdate)	Returns the probability that the rule will be applied with the Gillespie algorithm; only works if the rules do not have parameters and the probability is static

5.2.2 Pattern methods

Each pattern P initialized via the API has setters for all parameters and bind methods for all nodes.

Return Type	Signature	Description
Map<string, object>	getParameters()	Returns all parameters and bound nodes
p	bind(IMatch match)	Binds the nodes to the objects that are bound in the given match
p	bind (GraphTransformationMatch<?,?> match)	Binds the nodes to the objects that are bound in the given match. You can pass any match returned by any pattern or rule here
optional<M>	findAnyMatch()	Returns any matches for the pattern. Note that the resulting Optional object can be empty if no match exists
collection<M>	findMatches()	Returns all matches for the pattern
void	forEachMatch (Consumer<M> action)	Executes the given action for all matches
boolean	hasMatches()	Returns whether any matches for the pattern exist
int	hasMatches()	Returns whether any matches for the pattern exist
void	subscribeAppearing (Consumer<M> action)	Subscribes to any future match for the pattern. Whenever a new match for the pattern appears, the given action will be executed
void	unsubscribeAppearing (Consumer<M> action)	Unsubscribes the action such that the action will not be executed for new matches anymore
void	subscribeDisappearing (Consumer<M> action)	Subscribes to any disappearing matches for the pattern. Whenever a match for the pattern disappears, the given action will be executed
void	unsubscribeDisappearing (Consumer<M> action)	Unsubscribes the action such that the action will not be executed for disappearing matches anymore
void	subscribeMatchDisappears (M match, Consumer<M> action)	Subscribes to the given match. As soon as the match disappears, the given action will be executed
void	unsubscribeMatchDisappears (M match, Consumer<M> action)	Unsubscribes the action such that the action will not be called in the case the match disappears

5.2.3 Rule methods

Each rule R supports all methods provided for patterns and additional methods for rule application.

Return Type	Signature	Description
boolean	isApplicable()	Checks whether the rule can be applied, i.e., a match for the rule can be found
optional<M>	apply()	Applies the rule on an arbitrary match if any match exists and returns the co-match, i. e. the match after rule application. Note that the Optional will be empty if (1) no match exists or (2) the rule cannot be applied due to pushout semantics
optional<M>	apply(IMatch match)	Binds the nodes to the objects that are bound in the given match
optional<M>	apply(M match)	Applies the rule on the given match if possible and returns the co-match
collection<M>	apply(int max)	Applies the rule at most max times as long as there are matches the rule can be applied on
collection<M>	apply (predicate<collection<M» condition)	Applies the rule as long as the given condition based on the co-matches returns true
optional<M>	bindAndApply(IMatch match)	Binds the nodes from the given match and applies the rule
optional<M>	bindAndApply (GraphTransformationMatch<?,?> match)	Binds the nodes from the given match and applies the rule
collection<M>	bindAndApply (Supplier<? extends GraphTransformationMatch<?,?» matchSupplier)	Binds the nodes to the ones bound in the match given by the supplier and applies the rule. This is repeated until the supplier returns null
int	countRuleApplications()	Returns how often the rule has been applied
void	subscribeRuleApplications (Consumer<M> action)	Subscribes rule applications: Whenever the rule is applied, the given action will be executed
void	unsubscribeRuleApplications (Consumer<M> action)	Unsubscribes the action such that the action will not be executed for future rule applications
void	unsubscribeRuleApplicationsAll()	Removes all subscriptions for rule applications

5.2.4 Syntax reference

The list below provides a summary of the textual concrete syntax using the EBNF-style notation and <..> as placeholders for actual values:

```
1   /*— meta-models for node and parameter types ———*/
2   import"<URI of an Ecore meta-model>"
3
4   /*— pattern/rule with parameters ———————*/
5   [abstract] [pattern|rule] <pattern-name>[(<parameter-name: <parameter-type[, ...])]
6   [refines <super-pattern-name>[, ...]] {
7
8       /* Node: context, ++ for create, – for delete */
9       [[+]|[=]|[–]] <node-name>: <node-type> {
10
11          /* Attributes */
12          .<attribute-name> [:=] <constant>;
13          .<attribute-name> [:=] enums.<VALUE>;
14          .<attribute-name> [:=] parameter.<parameter-name>;
15          .<attribute-name> [:=] <node-name>.<attribute-name>;
16
17          /* References to other nodes */
18          [[+]|[=]|[[–]] -<reference-name> -> <node-name>
19      }
20  }
21
22  /* Probability of the rule; cannot be used on patterns */
23  [@ <arithmetic-expression> | [N|U|Exp](<arithmetic-expression>
24  [, <arithmetic-expression>]) [<arithmetic-expression>]]
25
26  /*— conditions to be used in patterns/rules ———-*/
27  /* Ensure that a certain pattern can be matched (positive application condition) */
28  enforce (<pattern-name>) [<node1>=<node2>]
29
30  /* Ensure that a certain pattern cannot be matched (negative application condition) */
31  forbid (<pattern-name>) [<node1>=<node2>]
32
33  /*—— summary of the arithmetic extension ————*/
34  <arithmetic-expression> = <constant> | <node-name>.<attribute-name> |
35  <arithmetic-expression> [+|-|*|/|%|^] <arithmetic-expression> |
36  [exp|log|ln|sqrt|abs|cos|sin|tan] (<arithmetic-expression>)
```

5.2.5 Syntax math functions

Syntax	**Function**
N(mean, standardDerivation)	Normal distribution
U(minValue, maxValue)	Uniform distribution
exp(lambda)	Exponential distribution
$x \wedge y$	Exponential function
exp(x)	Exponential function with e
log(x)	Logarithmic function
ln(x)	Natural logarithmic function
sqrt(x)	Square root
abs(x)	Absolute value function
cos(x)	Cosine
sin(x)	Sine
tan(x)	Tangens

6 Bidirectional Transformation with Triple Graph Grammars

For the third part of our tutorial, we will be introducing bidirectional transformations via triple graph grammar to our hospital example. Triple Graph Grammar (TGG) is a rule-based and declarative way to describe the language of all pairs of models that are considered to be consistent with each other. While these rules can be used to generate arbitrary consistent models from scratch, a TGG can also be used to automatically derive consistency management operations such as translators or synchronizers. In the case of our hospital, we created a metamodel view of the way the hospital might be perceived by a patient. From the viewpoint of the administration, other factors have to be considered to keep the very same hospital running, such as the salary of each staff member or their shift plans. This second administrative perspective contains new information but also information that overlaps with the patient perspective we already covered in the previous chapters and we have to make sure changing information is updated for both perspectives. For this purpose, we will be creating another metamodel from a different point of view. And throughout this chapter, we will be linking those two sides together to create our consistent triple grammar graph.

6.1 Administration metamodel

Let us look at the visualization of the administration metamodel in the chart below. You might notice some familiar classes like the staff and the patients, but from the administrative point of view, we want to manage our staff to cover the patients differently. Moreover, we want to keep track of the shifts a staff member covers, so every patient is cared for throughout the day.

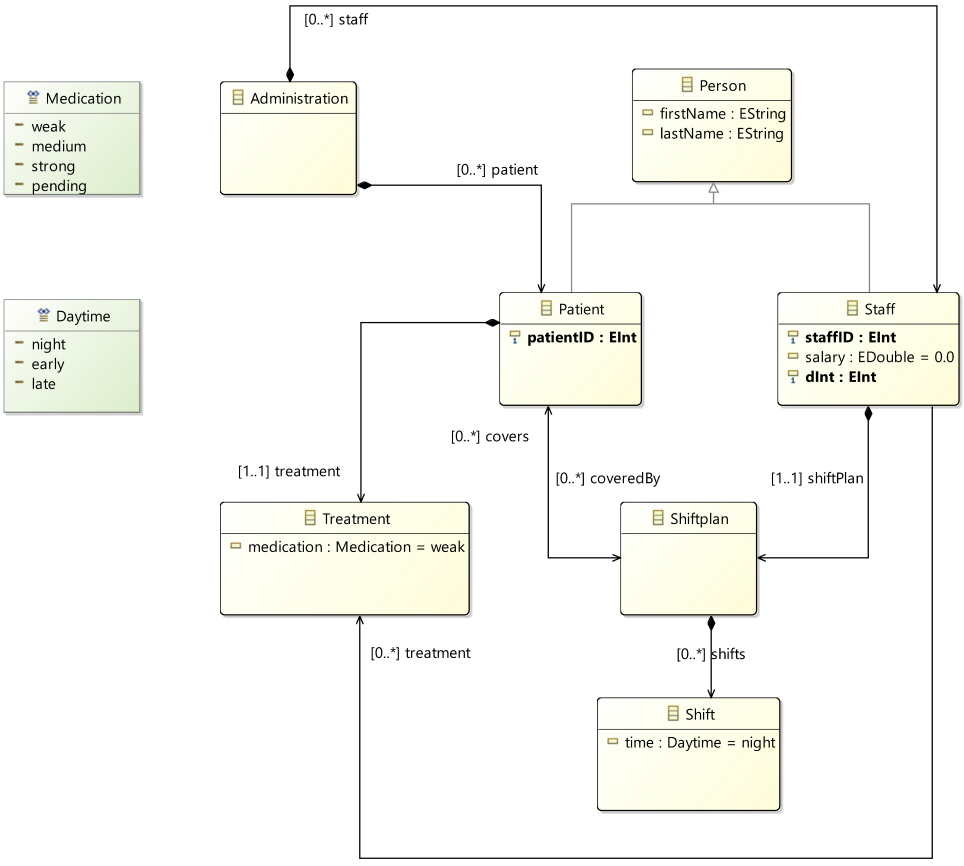


Figure 16: Administration model

For this purpose, we will be creating this new **administration metamodel** from scratch. If you do not want to exercise the creation of another metamodel yourself, you can skip this part and continue here with the **creation of a new TGG Project** in the next section.

For our administration metamodel, we need the administration container first. So go ahead and create it **as we did it for the hospital**.

Persons:
Then we need the persons which we want to cover for our administration. So, the next class you want to create are the **persons**, who should have a **first name** and a **last name** of the type **EString**.

Shifts:
The two next classes are the **patients** and the **staff**, which are of the **ESuperType** person so both patients and staff members are extended by the persons and have names as well. For the staff, we also want additional information such as the **staffID**, a **salary**, and the **dInt** variable to indicate the department a staff member belongs to. A patient only needs a **patientID** as an additional attribute. After creating the two classes we need to connect them to the administration by adding a **staff relation** for the staff and **patient relation** for the patients.

Shiftplans:
The next classes shiftplan and shift will be responsible for the coverage of the patients. So go ahead and create the class **Shiftplan** it does not require any attributes. But we want to add a **relation towards the staff** which ensures every staff member has **exactly one** shiftPan. Create the relation **shiftPlan** and set the upper and lower bounds to 1 to fulfill the desired multiplicities. We also want the patients to be covered by the shiftplan. Hence, we need a **bidirectional relation between the patients and the shift plans**.
The class **shift** will fill up a shiftPlan with shifts for the respective daytime. The times can be considered via the creation of the **Daytime Enumeration** with the literals **night**, **early**, and **late** and adding them as an attribute to the shifts class.

Medication and Treatment:
After covering the care throughout the day, the treatment of patients is of importance as well. So, we need to create the **enumeration Medication** with the following four types: **weak**, **medium**, **strong**, and **pending**.
Then we have to create a class that includes the medication. Create the class **Treatment** and a **medication attribute** that accesses the medication enumeration. The last step we need to do is to **connect the treatment class to our staff and the patients**.

The created metamodel should be completed now.

6.2 Creating a TGG project

We have now two metamodels containing information that are partly the same and partly different, but how do we connect the two sides and create a consistent triple? As a reminder, a consistent triple consists of three models which are related. The first model of a consistent triple is a so-called **source model**, which is represented by the hospital model in our case. The second model is the so-called **target model**, which is the administration model we have created in the previous segment. The third model is the **correspondence model** which puts the target and source model with each other.

Project creation:
Let us start with creating a new TGG Project to link our hospital metamodel with the administration metamodel. To create a new TGG Project just click on button 7 with the **green rhomb and the folder**. Then proceed with:

Use the name "Hospital2Administration" → Select option "Project with preselected metamodel"
→ Press "next"

Now you have to select a **source metamodel** which will be the hospital example in our case. Scroll down through the list of possible options shown in the chart below and select:

platform:/resource/HospitalExample/model/HospitalExample.ecore

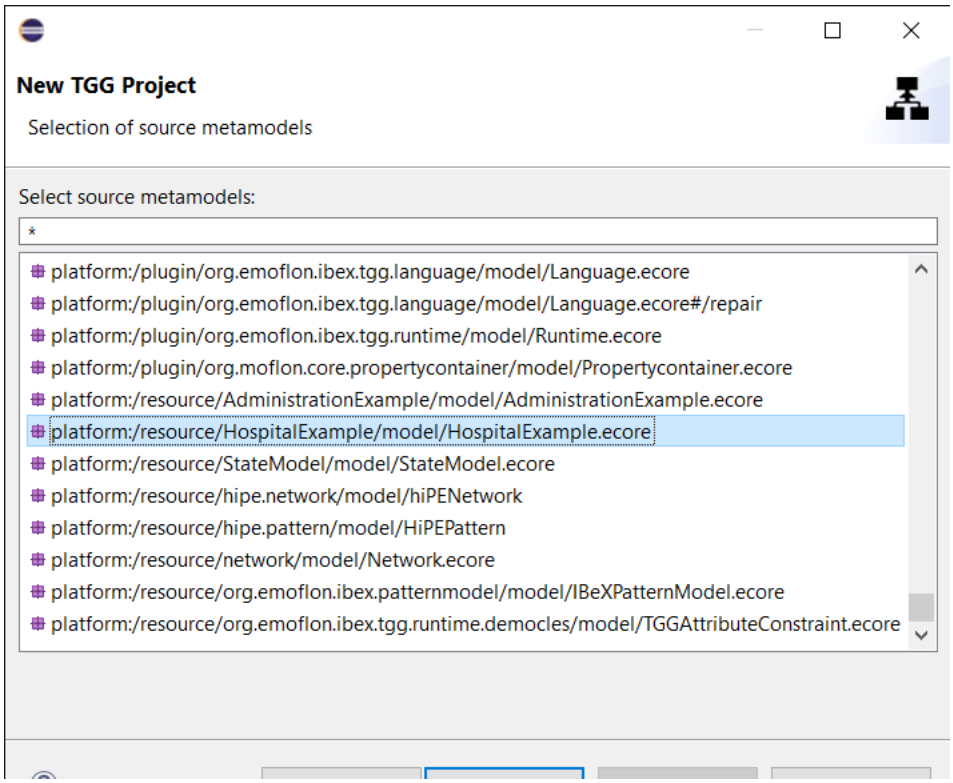


Figure 17: Source model selection

The second model to select will be our **target model** and you might have already guessed it we will select our Administration metamodel:

platform:/resource/AdministrationExample/model/AdministrationExample.ecore

-> Afterwards press "Finish"

Right now, we have a hospital metamodel and an administration metamodel but they are not connected. As a consequence, we need means to create relations between our two metamodels. These relations are defined in the files we will inspect now.
In the source folder you will find the **tgg package** and in it you will find the **Schema.tggl** file. This file is the central element linking both the source and the target side since it defines the **correspondence metamodel** for the connections between both the source and the target side. The correspondence metamodel defined by the Schema.tggl also completes our triple of metamodel graphs.

Warning: Please always start the name of your project with a **capital letter**. The automatic file generation will not work otherwise.

There are three important parts in the Schema file. Also there are a lot of desired elements missing in the Schema.tggl which will be added throughout the next section. Do not worry we will explain everything in detail and walk you through every step.

6.3 Writing a TGG schema

The automatically generated Schema.tggl should look like this:

```
1   import "platform:/resource/AdministrationExample/model/AdministrationExample.ecore" using HospitalExample as HE
2   import "platform:/resource/HospitalExample/model/HospitalExample.ecore" using AdministrationExample as AE
3   schema Hospital2Administration {
4   source {
5       HospitalExample
6   }
7   target {
8       AdministrationExample
9   }
10   correspondence {
11   }
12   }
```

Source and target model:
First, the Schema.tgg defines the **source** and the **target** which was done automatically in our case because we defined the source and target model when we created the project. If you want to start with a blank project you have to do this yourself. You can see this definition in lines 5 and 8 where the HospitalExample metamodel was defined as the source model and the AdministrationExample metamodel was defined as the target model.
Additionally you can define **aliases**. Those are meant for you to adress your source and target model in the way you want. In the Schema.tggl file you can see those aliases defined by using the keyword **as** and the name you intend, here **HE** for HospitalExample and **AE** for AdministrationExample. In the correspondences we will define on the next page you can see that we use the aliases to specify the model we want to adress instead of the full name.

Correspondences:
The second block consists of the **correspondences** between our meta models. The connections between our models we have talked previously about are called correspondences and their purpose is to define the elements which are connected. This might be a bit confusing so let us take a look at our example to make things clearer. The very first correspondence we want to create is the correspondence between our hospital and the administration. We can create a correspondence by writing the name of our correspondence and defining which elements on the source and target side should be connected in this correspondence. We recommend using the names of the elements we want to correspond with each other as the **names for the correspondence** for comprehensibility reasons.

Staff and patients:
Go ahead and write Hospital2Administration within the *correspondence* bracket and **add curved brackets** to it. Now we need to choose an element of the source/hospital side we want to correspond with an element on the target/administration side. As the name of the correspondence already indicates we want the **hospital class** to correspond with the **administration class** on the target metamodel.
A source element can be added via typing src-> <nameOfElement>. In the case of our hospital, we have to type src->Hospital. Since we are still missing an element on the target side, we add the administration by writing trg->Administration. Our correspondence metamodel has now one correspondence between the hospital and the administration.

However, we have still more elements we want to correspond with each other. Furthermore we want to connect the **staff** and the **patients** of both sides to each other. Let us create the new correspondence StaffToStaff and assign the staff classes of both sides as target and source to this correspondence.

Try coding the PatientToPatient correspondence yourself. For the correspondence between patients, we want to connect the patients on the source side with their respective counterparts on the target side.

Doctors and nurses:
Now we have covered the rather obvious correspondences between our two models, but some classes contain information we want to be present in the other model too. We only have staff members on the target side but we still want to differentiate **doctors** from **nurses** there. Since we do know that the doctors and the nurses are related to the staff class on the source side, it makes sense to let the nurses and doctors correspond with the staff on the target side.
Let us create two further **correspondences** then. Firstly, we need the **DoctorToStaff** correspondences which ensures a doctor on the source side corresponds with the staff on the target side. Secondly, the nurses should correspond with staff in the same way, and hence we need the **NurseToStaff** correspondence.

Finally your *correspondence* section should look like this:

```
10  correspondence {
11      Hospital2Administration{
12          src-> HospitalExample.Hospital
13          trg-> AdministrationExample.Administration
14      }
15      StaffToStaff{
16          src-> HE.Staff
17          trg-> AE.Staff
18      }
19      PatientToPatient{
20          src-> HE.Patient
21          trg-> AE.Patient
22      }
23      DoctorToStaff{
24          src-> HE.Doctor
25          trg-> AE.Staff
26      }
27      NurseToStaff{
28          src-> HE.Nurse
29          trg-> AE.Staff
30      }
31  }
```

6.4 TGG Rules

Since the ruleset and certain specifications for the triple grammar graph project are quite extensive, we do not want you to create everything by yourself but rather explain to you the ideas behind certain structures. For this reason there are two Hospital2Administration projects. **Hospital2Administration** is the template you should be using for this part of the tutorial. It contains all of the predefined stuff you need for it and you can just add the new stuff to it. **Hospital2AdministrationSolution** contains the solution for this part of the tutorial[5].

First rule:
In contrast to the Schema.tgg where we defined the correspondence metamodel for the hospital and the administration, rules are responsible for the **creation of the actual structure** of our triple grammar graph. You will notice they work similarly to the rules we defined in the graph transformation project. Start with the creation of a new TGG rule in the package org.emoflon.ibex.tgg.rules.

Please create this rule in the described way.

Click on the package → Press button 8 → Name it "HospitalToAdministration" → "Finish"

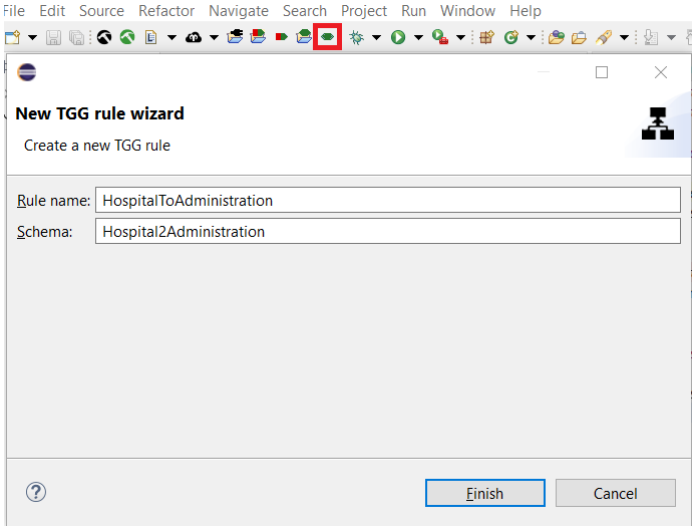


Figure 18: TGG rule creation

As we can see the proper schema is assigned automatically, at least if you have selected the right package in your TGG project.

For our first TGG rule, we want to create a connection for the hospital and the administration, as well as the nodes themselves. For this rule, we want a **hospital node** and a **reception node** with an **edge between them** on the source side.
The target side will require the creation of an **administration node**.
So far, the syntax of our TGG rule is almost the same as in graph transformation projects.

According to the correspondence metamodel that we have defined in the Schema.tgg this rule also needs to consider the correspondence between our hospital node and the administration node. We do this by **adding a new correspondence** (lines 16 to 20) in the correspondence bracket.
This is similar to the principle of graph transformation rules. We first have define in the ecoremodel what relations different elements in our model have between each other, i.e references. When creating instances of these elements via the application of a rule we also need to add an instance of those relations explicitly.

[5] Please notice that the packages in the solution projects also contain the string `solutions`. With this differentiation, you are able to import both projects to the same workspace simultaneously.

Please name the correspondence htov and define its type as Hospital2Administration. This is the correspondence we have created in the Schema.tggl.
Now we are going to add the hospital node we have created in this rule to the source side and the administration node to the target side.

Finally your rule should look like this:

```
source {
    [+] h:HE.Hospital{
        [+] -reception->r
    }
    [+] r:HE.Reception{
    }
}
target {
    [+] v:AE.Administration
}
correspondence{
    [+] htov:HospitalToAdministration{
        src-> h
        trg-> v
    }
}
attributeConditions{
}
```

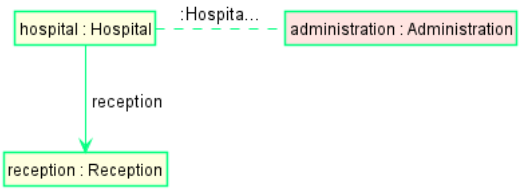


Figure 19: PlantUML visualization HospitalToAdministration

Note on the visualization:
On the left side, we see our target side which creates the hospital and the reception and on the right side, we can see a new administration. The dotted line between the hospital and administration stands for the correspondence between the two models.

Since you now have a grasp of how to create a rule let us take a step back and recapitulate its function. The rule creates a consistent triple consisting of the hospital and the reception nodes on the source side, the administration node on the target side, and their connection to each other in form of the correspondence. In other words, whenever we create a hospital and a reception on the source side, our correspondence requires the creation of an administration on the target side.

Departments and rooms:
The next two rules we will create are the rules for the departments and the rooms. The department rule creates a department with the respective edge to the hospital and works just as the departmentRule in the GT project. This rule allows the creation of a department if we already created a hospital node in our project.
So go ahead and create a new tgg rule with the name DepartmentRule for which we need to add the following contents:

```
rule DepartmentRule {
source {
    [=] h : HE.Hospital{
        [+] -department->d
    }
    [+] d : HE.Department{
    }
}
target {
}
correspondence {
}
attributeConditions {
    Hospital2AdministrationLibrary.incrementingDepartmentID(d.dID)
    Hospital2AdministrationLibrary.setDefaultNumber(d, 10)
}
}
```

Note:
We will explain the attributeConditions section later on. Please just copy these sections for now.

Please try to code the RoomRule yourself.

For this rule want to add a **room node** in case we have a department and assign the **default capacity** of that room to 4.
It should look like this in the end:

```
rule RoomRule {
source {
    [=] d:HE.Department{
        [+] -rooms->ro
    }
    [+] ro:HE.Room{
         .capacity:=4;
    }
}
target {
}
correspondence {
}
attributeConditions {
}
```

As you can see, we are only operating on the source side of our consistent triple since we do not have and do not need a correspondence for these elements on the target side because these elements are not present on the administration side.

Abstract rules:
Let us continue with the remaining rules since we are still lacking the personnel and the patients. Look at the StaffToStaffRule which is already present in your Java project. Its visualization is shown in the chart below.
For this rule, we assume that a hospital node, at least one department node, an administration node and the correspondence between source and target side exist.
Then we want to create a staff node and its respective connections on the source side. When creating a staff node on the target side we also want to connect them to the target side by creating the StaffToStaff **correspondence** and the staff for the target side itself. Upon creating a staff member on the target side, we also want to assign them a shift plan and a shift.

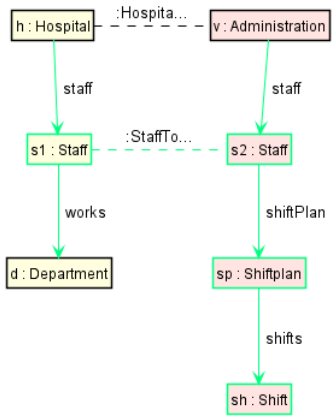


Figure 20: PlantUML visualization StaffToStaffRule

You probably have noticed the keyword abstract in the definition of this rule. Similar to **abstract** rules from graph transformations this means that this rule cannot be applied directly.

Extending abstract rules:
You might also wonder how we are going to distinguish doctors from nurses in the administration view. That is the part where the DoctorToStaff and NurseToStaff **correspondences** that we have created previously come into play. With those we have established a correspondence between doctors and nurses on the source side to staff on the target side.
This requires a further extension of our Staff rule which can be added just below the StaffToStaffRule. The keyword for the further specification of an existing rule in TGG is refines which needs to be added after the name of the rule. If you remember from the GT part of our tutorial the keyword refines works similar to the concept of inheritance in Java. We reduce redundancies for the new rules because they inherit the elements defined in the StaffToStaffRule as well. For the creation of the DocToStaff rule, which can be done in the present StaffToStaff rule, we need to add a staff node of the type doctor on our *source* side and a new staff node on the *target* side.

```
1   rule DocToStaffRule refines StaffToStaffRule
2   source {
3       [+] s1d:HE.Doctor
4   }
5   target {
6       [+] s2d:AE.Staff
7   }
8   correspondence{
9   }
10  attributeConditions{
11      Hospital2AdministrationLibrary.doctorsalary(s2.salary)
12  }
```

The NurseToStaffRule can be created similarly. Please try it yourself before having a look at the syntax below. Change the type of staff member accordingly and adjust the attribute condition to nursesalary.

```
1   rule NurseToStaffRule refines StaffToStaffRule
2    source {
3        [+] s1n:HE.Nurse
4   }
5    target {
6        [+] s2n:AE.Staff
7   }
8   correspondence{
9   }
10   attributeConditions{
11        Hospital2AdministrationLibrary.nursesalary(s2.salary)
12    }
```

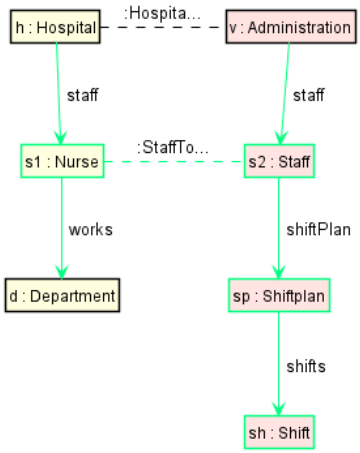


Figure 21: Visualization of NurseToStaffRule. DocToStaffRule is similar with changed staff type.

Shiftplan rule:
The next function we want to cover is the shift plans. In the ShiftplanRule, we want to add an edge on the target side to make sure a patient is covered by a shift plan. So let us start by creating the new rule ShiftPlanRule.

First, we want to make that rule **abstract** because as we have defined it in the GT part of our tutorial the coverage of patients via doctors and nurses works differently. While doctors are assigned to a patient directly, nurses cover patients indirectly with the rooms they are responsible for.
We require patient and staff nodes on the source side while having a staff, shift, shift plan and a patient node on the target side. Furthermore this shift plan has to be related to the specific staff and shift nodes via an edge.
For the patient node in our administration model, we want to add an edge to the shift plan to indicate a patient is covered in that shift plan.
In the last step for this rule, we need to define the **correspondences** with patients and staff members between the source and the target side. Since we have all elements on both sides defined this step should be pretty forward.

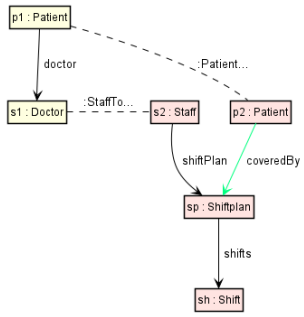
You can look at the complete rule on the next page.

```
1   abstract rule ShiftplanRule{
2    source {
3       [=] p1:HE.Patient{
4       }
5       [=] s1:HE.Staff{
6       }
7   }
8    target {
9       [=] s2:AE.Staff{
10              [=] -shiftPlan->sp
11       }
12       [=] sp:AE.Shiftplan{
13              [=] -shifts->sh
14       }
15       [=] sh:AE.Shift{
16       }
17       [=] p2:AE.Patient{
18              [+] -coveredBy->sp
19       }
20    }
21    correspondence {
22       [=] pToP:PatientToPatient{
23              src ->p1
24              trg ->p2
25       }
26       [=] sToS:StaffToStaff{
27              src ->s1
28              trg ->s2
29       }
30    }
31    }
```

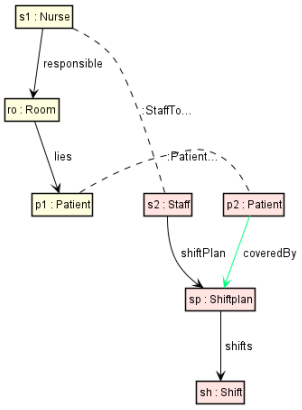
Specific shiftplans:

For this case, we also need differentiation between doctors and nurses, due to the different paths towards the coverage of a patient. Hence, we need two more rules.

These are already given in your project. You can look at the respective visualization of the DoctorShiftplanRule and the NurseShiftplanRule in your project:



(a) Visualization of DocShiftplanRule



(b) Visualization of NurseShiftplanRule

Rules regarding patients:
Now that we have covered the staff and their respective shifts, we need to handle the patients.
The **abstract** PatientToPatientRule adds patients and the required correspondences for further rules.
This rule is already given in the project and you might want to take a look at it. The visualization is shown in the chart below:

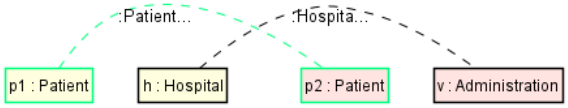


Figure 23: Visualization of PatientToPatientRule

As you might have noticed we are adding two correspondences to our model which are independent from each other. This requires us to add two rules extending the *PatientToPatient* rule to fit the patients into our hospital view.
For the first rule, we assume that our patients are waiting in the reception until they are moved to a room where they are treated. So, we want to create the rule PatientInReception where we require the context of the hospital node, the edge towards the reception and the reception node on the source side. Additionally, we require the context of the administration node and the HospitalToAdministration correspondence between the hospital node and the administration node.
Regarding the creation of nodes and edges, we want to add a patient node and a waits edge from the reception to the patient on the source side.
For the target side, we want to create the patient node and its respective edge from the administration towards the patient node.

Please try creating this rule yourself, the syntax can be found below:

```
1   rule PatientInReception refines PatientToPatient {
2    source {
3       [+] p1r : HE.Patient
4       [=] hr:HE.Hospital{
5           [=] -reception->r
6       }
7       [=] r:HE.Reception{
8           [+] -waits->p1
9       }
10   }
11    target {
12      [+] p2r : AE.Patient
13      [=] vr:AE.Administration{
14          [+] -patient->p2
15      }
16   }
17   correspondence {
18      [=] htovr:HospitalToAdministration{
19          src-> hr
20          trg-> vr
21      }
22   }
23   attributeConditions {
24   }
25   }
```

The second rule PatientInRoom is already given in your project. This rule adds the lies edge from the context node room to the node patient.

Its visualization is depicted in the chart below:

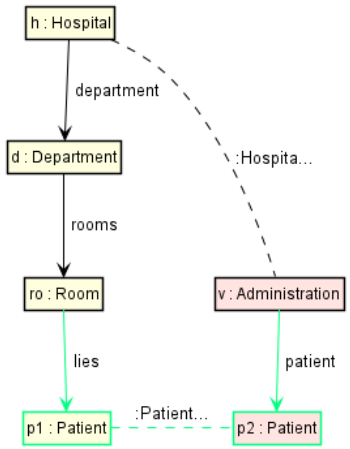


Figure 24: Visualization of PatientInRoomRule

NurseToRoomRule:
The last remaining rule is the NurseToRoomRule, where we add the edge from the nurse node to the room node according to the assumptions in our metamodel. The rule is also given in your project and the visualization is depicted in the chart below:

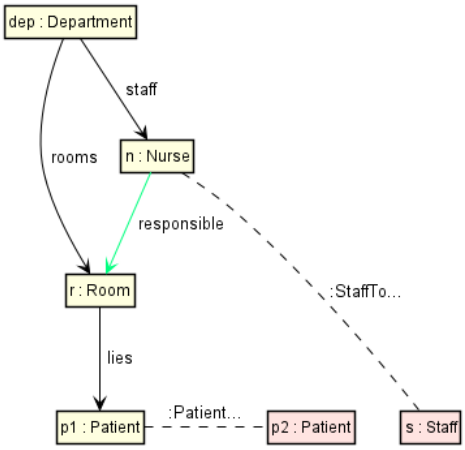


Figure 25: Visualization of PatientInRoomRule

After creating the rules for the infrastructure and the persons in the hospital you should have a grasp of the concept of creating rules for triple graph grammars and the way they are working. So let us continue with another important function we have skipped previously, the **attribute conditions**.

6.5 Attribute Conditions

In this chapter we will be covering **attribute conditions**. Attribute conditions can be used to assign meaningful values to attributes when creating a coherent graph triplet. They can also modify already set attribute values of an existing target and source graph pair to synchronize them.
They will be applied when their corresponding rule is.

You have already seen how to invoke them in the previous chapters:

```
...
attributeConditions {
    Hospital2AdministrationLibrary.incrementingDepartmentID(department.dID)
    Hospital2AdministrationLibrary.setDefaultNumber(department.maxRoomCount, 10)
}
```

To introduce you to the inner workings of attribute conditions we will have a look a the pre-defined condition setDefaultNumber(variableNumber:EDouble, defaultNumber:EDouble).

Please, go to **line 75** in the file **DefaultAttributeConditionLibrary** in the shown location:

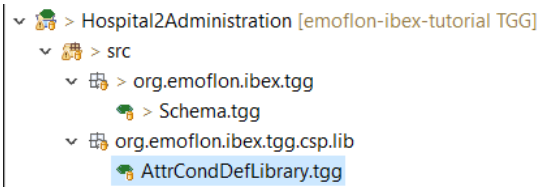


Figure 26: Location of setDefaultNumber

This file contains the library of predefined attribute conditions. Let us inspect the setDefaultNumber condition which was used previously. This attribute condition ensures that an attribute value is set to a defined default number if it is unset.

Control flow:
Attribute conditions in general use a control flow that will be defined and is implemented as a **switch case**. Each case condition is made up by **all valid combinations of the bounding states** of referenced attribute values, i.e the input parameters.

An attribute can be **free** meaning that there is no value assigned to it. This will be represented as "F" in the condition.
The other option is that the attribute is **bonded** which means that it already has a set value. This will be represented as "B" in the condition.
These cases will be defined in the definition of the attribute condition.

Note that there are two different sets of cases for the two application scenarios:

- *sync*: Keeping two existing graphs synchronized. [F F ... F] is not available since unset values don't need to be synchronized.
- *gen*: Generation of a new graph pair.

Pre-defined conditions:
Let's have a look at the definition of setDefaultNumber:

```
75    setDefaultNumber(variableNumber:EDouble, defaultNumber:EDouble) {
76        sync { [B B], [F B] }
77        gen { [B B], [F B], [F F] }
78    }
```

Because this condition is pre-defined the effects of each case are already implemented. If an attribute has no value and is perceived as free, it would be assigned the defaultNumber:EDouble we defined in the attribute condition of our rule. If the variableNumber:EDouble is already set and hence bound, the bound variableNumber will be kept.
Note that there is no case [B F] because a default value has to be specified for this to work.

You can find a comprehensive list of the predefined attribute conditions and a short explanation of each condition in the **appendix**.

User-defined conditions:
To create a user-defined attribute condition you have to define your own **library**. We will call it **Hospital2AdministrationLibrary**. Let us start by inspecting the incrementingDepartmentID which is already given:

```
88    incrementingDepartmentID(id:EInt) {
89        sync    { [B], [F] }
90        gen     { [F] }
91    }
```

A Library is defined by the keyword library followed by the **name** of the library. You can define as many libraries as you want.
In our case, we want to modify the ID of a department. The different boundary states are defined in the curved brackets just like in the predefined attribute conditions.

Be aware that after the application of an attribute condition all involved attributes have to be **bound**.

Maybe we should practice a bit by creating a fresh user-defined attribute condition. Our rooms also have IDs that we want to assign identically to the departmentIDs. Start by defining the attribute constraint incrementingRoomID in the schema.tgg. Its definition should look very similar:

```
108   incrementingRoomID(id:EInt)
109   {
110       sync{[B],[F]}
111       gen{[F]}
112   }
```

Please **build** the project by pressing either the **black or the green hammer button** 1 in the eMoflon Toolbar. Once you refresh your project folder you can see that the new attribute condition was created automatically and tucked into our project by eMoflon.

Complex conditions:
To further get to know user-defined attribute constraints let us look at the UserDefined_nametoname.Java file which is a rather complex condition and already given in our project. As you can see in the schema.tgg the number of different boundary states has increased notably since we are handling more parameters.

```
82  nametoname(separator:EString, leftWord:EString, rightWord:EString, result:EString) {
83      sync{ [B B B B], [B B B F], [B B F B], [B F F B], [B F B B]}
84      gen{ [B B B B] , [B B B F], [B B F B], [B F F B], [B F B B], [B F B F], [B B F F], [B F F F]}
85  }
```

We should start by explaining the purpose of this attribute constraint. If you compare the name attributes of the patients and staff members in the hospital metamodel with the name attribute of the Person class in the administration metamodel, you will notice that the name attributes on the administration side are split up in first and last name. This specification forces us to split up the full name consisting of first and last name on the hospital side into two attributes if we want to propagate the name information from the hospital side to the administration side or concatenate two names if we want to execute the opposite operation. This leads to the four parameters we need for the attribute constraint.
The separator:EString defines the character we are using to distinguish first from last name in case both are in the same attribute. The leftWord:EString will define the first name while rightWord:EString will be handling the last name. The result:EString combines the three previous Strings to one String.

6.6 Running the TGG Project

Now that we have finally defined all the elements and their respective correspondences of our models it is time to get back to our primary goal for this part of the tutorial. The generation of a consistent triple and its consistency maintenance.

As a short reminder: A consistent triple includes the source model, a correspondence model, and a target model. Once information changes in one model we want to propagate the information to the other models to achieve consistency. eMoflon provides a different set of Java applications for this purpose and we will go through the important ones step by step.

Open the package org.emoflon.ibex.tgg.run.hospital2administration and take a look at the different java applications you can see in the chart below:

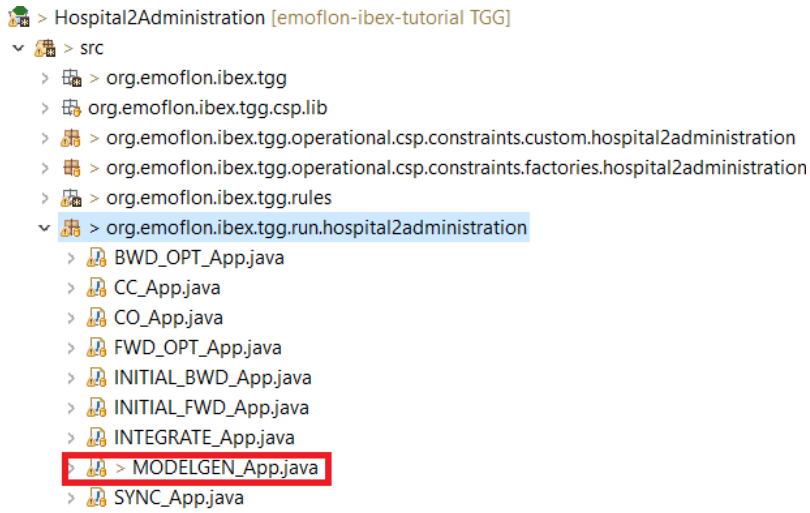


Figure 27: run.hospital2administration package

The first operation we are interested in is the *MODELGEN_APP.java* which is marked above. Let us start exploring this feature by **running** it in our java project. Please **refresh** the project folder afterwards.

Once you open the instances folder it should look like this:

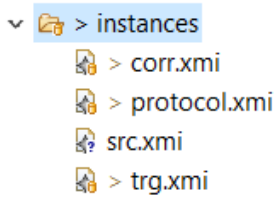


Figure 28: instances folder

Model generation via MODELGEN:
The MODELGEN_APP generates a consistent triple according to our rule sets without any preexisting models. It generates a hospital model, an administration target model, and the correspondence model for the connections between our source and target model.

By opening the src.xmi you can take a look a the final model instance. You should see patients waiting in the reception or being assigned to their rooms as well as the staff members within their departments. Nurses for example which should be covering a room with patients. Look at the trg.xmi and you will notice that we have the same patients and staff members in this model instance. But here we focus on the shifts of the latter and treatments we assigned to their patients.

A model generator might also be helpful for testing purposes, especially scalability testing that requires large models.

Other running options:
Let us explore the other run options as well. Although we are mainly interested in the synchronization function for our current example. It might be useful to take a step back and start with the forward and backward transformations as special cases of the general task of consistency management.

INITIAL_FWD:
This application requires a source model and can create or restore the target model instance in case it is given the correspondences. You might try it by deleting the trg.xmi in your instances folder and running the app.

INITIAL_BWD:
This works in the same way but requires the target side. Hence, the names "fwd" for forward synchronization and "bwd" for backward.

CC:
Another option is the CC_App which stands for consistency checking. It is used to compare the given source and target model instances by creating a respective correspondence model.

CO:
If you want to check a complete triple of source, correspondence, and target models for consistency you can use the CO_App which stands for check only.

Model synchronization via SYNC:
Throughout the rest of this tutorial, we will be concentrating on model transformation via the **SYNC** operation.
There are two reasons why initial forward and backward transformations are limited: First, they are not incremental and might incur loss of information if your bidirectional transformation is not bijective. Since both forward and backward transformation only translates the given side into a new model instance for the missing side. This can happen if one of the graphs doesn't have any correspondence for a specific node type on the other side, shifts for example.

One possible solution would be to add such properties manually to the respective metamodels, but it should be clear that this defeats the purpose of having multiple metamodels for different domains or applications. The **SYNC** operation can deal with this problem because it is updating an existing output model **incrementally**. We are only adding what is necessary, missing or has changed. In this manner, unrelated parts of the model can be retained.

Another reason to use **SYNC** is the improved performance. Since it works **incrementally**, the time required for an update of the current model triplet does not depend on its current size, but rather on the size of the update itself.
Please note that you can only update either source or target model during one synchronization operation.If you changed the information on both sides of your model and want to propagate them to the respective side you should use the **integrate** operation via the INTEGRATE_APP application.
The **SYNC** operation is of course the best way to handle small updates when you are working on your eMoflon project. And henceforth we recommend using this option.

Pattern matcher configuration:
Now let us take a closer look at these apps because there are some options to configure them. The first line of interest is the definition of the registration helper in very application:

```
20    public static IRegistrationHelper registrationHelper = new HiPERegistrationHelper();
```

As you can see in the package hospital2administration.config shown in the chart below you have the choice between the different registration helpers:

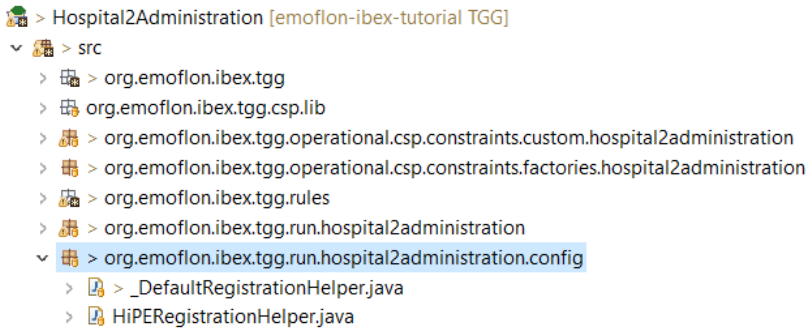


Figure 29: Available registration helpers

By default, a newly created project has the _DefaultRegistration helper selected, but we recommend using the HiPERegistrationHelper for every application. **Exceptions** are the **BWD_OPT** and the **FWD_OPT** applications since both require the DemoclesRegistrationHelper.
The main difference between those two is the utilization of **different pattern matchers** which we have explained at the very beginning of our tutorial.

Saving and loading models:
Let's look at further options within the applications:

```
18 public class MODELGEN_App extends MODELGEN {
19
20     // eMoflon supports other pattern matching engines. Replace _DefaultRegistrationHelper with one of the other
21     public static IRegistrationHelper registrationHelper = new HiPERegistrationHelper();
22
23     public MODELGEN_App() throws IOException {
24         super(registrationHelper.createIbexOptions().resourceHandler(new TGGResourceHandler() {
25             @Override
26             public void saveModels() throws IOException {
27                 // Use the commented code below to implement saveModels individually.
28                 // source.save(null);
29                 // target.save(null);
30                 // corr.save(null);
31                 // protocol.save(null);
32
33                 super.saveModels();
34             }
35
36             @Override
37             public void loadModels() throws IOException {
38                 // Use the commented code below to implement loadModels individually.
39                 // loadResource loads from a file while createResource creates a new resource without content
40                 // source = loadResource(options.project.path() + "/instances/src.xmi");
41                 // target = createResource(options.project.path() + "/instances/trg.xmi");
42                 // corr = createResource(options.project.path() + "/instances/corr.xmi");
43                 // protocol = createResource(options.project.path() + "/instances/protocol.xmi");
44
45                 super.loadModels();
46             }
47         }));
```

Figure 30: Example for loadModels() and saveModels() methods in the MODELGEN_APP

Depending on the operation you want to perform you can use the code which is commented out right now.

saveModels():
This method is used to save specific components or the whole graph triplet. super.saveModels() saves all four files you can see in the instances folder. You can swap this out for one or multiple of the file specific save commands and insert a specific path for saving.

loadModels():
This method is used to load specific components or a complete graph triplet. For the individual load methods, we would load a model from the given path and create the corresponding resources.

Example:
Let us walk through this exemplarily by modifying the SYNC_APP to load the hospital.xmi we have created in the first part of this tutorial. First we want to modify the loadModels method to load the hospital.xmi.
You need to adjust the **path** to the source variable to access it. It should be saved in your workspace in the project folder of the HospitalExample. we will use a relative path to switch to the parent folder of our project directory then we navigate to the hospital.xmi which is stored in the project folder of our graph transformation project:

```
39     source = loadResource(options.project.path() "../HopsitalTransformationRules/hospital.xmi");
```

Since we want to synchronize our triple from the given source model, we have to create the other resources by using the createResource command which is commented out right now. Please remove the slash symbols for the following lines shown below:

```
40    target = createResource(options.project.path() + "/instances/trg.xmi");
41    corr = createResource(options.project.path() + "/instances/corr.xmi");
42    protocol = createResource(options.project.path() + "/instances/protocol.xmi");
```

After creating these resources, we still need to save them somewhere. Just uncomment the following lines in the saveModels method:

```
40    target.save(null);
41    corr.save(null);
42    protocol.save(null);
```

After you have saved your file and **ran** the **SYNC** application, look at the new instances. They should look different. If they are still the same, try deleting the files in the instances folder and rerun the App.
Now open up the trg.xmi and inspect it. The number of patients and staff members is much smaller now, but why?
In this case, we have loaded the hospital.xmi from our graph transformation project and in this model instance we have created the elements such as patients and staff members manually and the SYNC_APP has generated our triple consisting of source, target, and correspondences according to the rules we defined in the TGG project.
On the contrary, the MODELGEN_APP created our consistent triple according to our TGG ruleset as well but **none of the values were bound** and hence the triple was created with random values as we defined it in the attribute conditions. Additionally, it would have done it **infinitely** if we had not defined the criteria to stop the execution of the application.

You can find these criteria in the main method of the applications. For example, we set a stop criterion based on the runtime:

```
stop.setTimeOutInMS(1000);
```

For this example, we terminate our application after approximately 1000 mswith the keyword stop. By using the auto-completion function you can see the other options you can set as **stop criteria**.
For example, if we want to execute a rule a certain number of times we can type:

```
stop.setMaxRuleCount("HospitaltoAdministrationRule", 1);
```

In this case, we limited the number of applications of the HospitalToAdministration to 1. You can find a list of possible stop criteria in the **appendix** below.

6.7 Debugging in TGGs

A note on the debugging app you can find in the .debug package shown below:

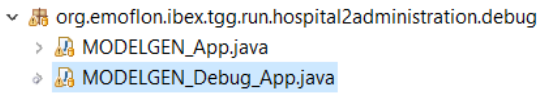


Figure 31: .debug package

There is no need for this when your set of triples is created without any errors. But when the rules or correspondences are not working in the way you want them to work you can run the generation in debug mode.
Once you run the MODELGEN_Debug_App.java a new window like the one in chart below pops up. On the top left, you can see the rules which have already been applied, the other rules are crossed out. On the right, you can see the respective visualization, and on the bottom left you can see the order in which the rules were applied.

If you hit the **apply** button on the left now you will apply the next rule according to the way, we have defined it in the java files. This works like debugging in Java and leads you step by step through the application of our ruleset and updates of the visualization.

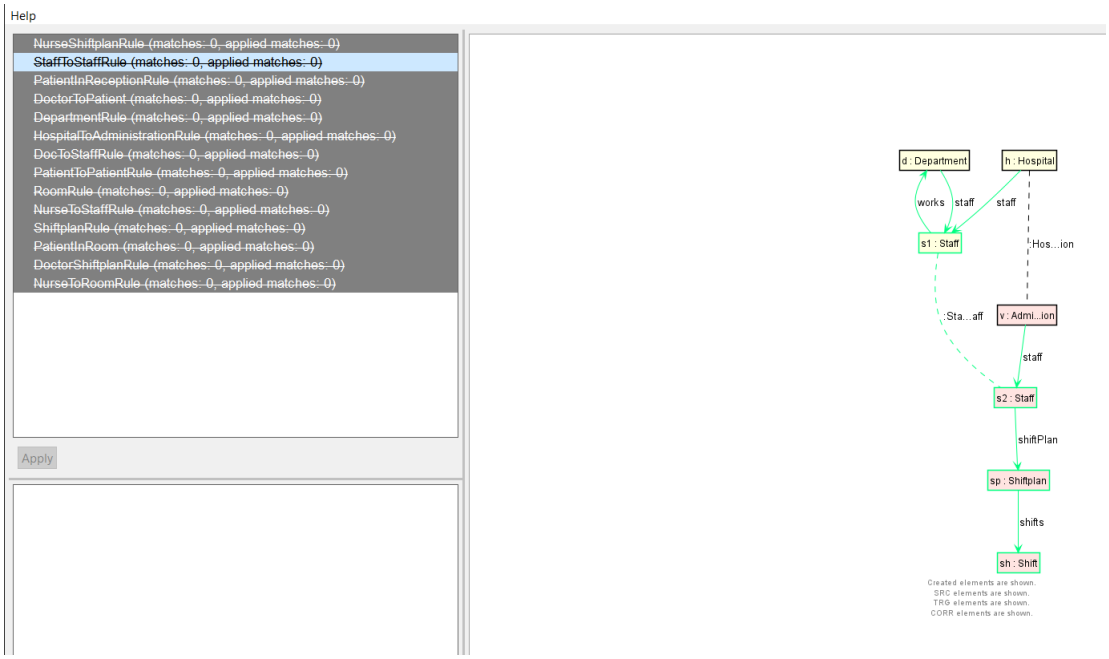


Figure 32: Debugging window

7 Appendix for Triple Graph Grammars

7.1 Debugging the generated Pattern Invocation Networks (PIN):

If you want to understand what is being provided as input to the underlying pattern matcher, or you are developing a new feature and need to “see” what is being passed as patterns, you can persist and visualize the so-called Pattern Invocation Network (PIN), which serves as input for the pattern matcher.

1. In the app of your choice, e.g. MODELGEN_App.java, locate the createIbexOptions method and append a debug(true) to the default statement in the method. This should return _RegistrationHelper.createIbexOptions(). If you wish to switch on debug globally, you can change the value set in the _RegistrationHelper

2. Run the app with your change and refresh your TGG project. If you did things right, you should notice a newly created /debug folder in the project. If you open it,there should be two files: ibex-patterns.xmi and democles-patterns.xmi. These contain the same PIN but on different levels of abstraction: The former is independent of a specific pattern matcher, while the latter is precisely what Democles, the default pattern matcher, requires.

3. In most cases, you will be interested in ibex-patterns.xmi. Opening this file will enable you to visualize the contained PIN in the PlantUML view. There is an overview visualization for the pattern set on the one hand and on the other is a visualization for individual context patterns.

7.2 Attribute conditions overview

Here you can find a list of all default attribute conditions with a short description each. To look up all specific cases they cover take a look at the AttrCondDefLibrary.tgg in the org.emoflon.ibex.tgg.csp.lib package in your TGG project.

Attribute condition	Description
eq_string(a: EString, b: EString)	Ensures both given string variables are equal
eq_int(a: EInt, b: EInt)	Ensures both given integer variables are equal
eq_float(a: EFloat, b: EFloat)	Ensures both given float variables are equal
eq_double(a: EDouble, b: EDouble)	Ensures both given double variables are equal
eq_long(a: ELong, b: ELong)	Ensures both given long variables are equal
eq_char(a: EChar, b: EChar)	Ensures both given char variables are equal
eq_boolean(a: EBoolean, b: EBoolean)	Ensures both given boolean variables are equal
addPrefix(prefix:EString, word:EString, result:EString)	Adds a prefix to a given word variable and handing over the prefix plus the word as the result
addSuffix(suffix:EString, word:EString, result:EString)	Adds a suffix to a given word variable and handing over the suffix plus the word as the result
concat(separator:EString, leftWord:EString, rightWord:EString, result:EString)	Combines a left word the separator and the right word in this order to a result
setDefaultString(variableString:EString, defaultString:EString)	Attribute condition which sets a variable string to the defaultString if it is free.
setDefaultNumber(variableNumber:EDouble, defaultNumber:EDouble)	Sets a variableNumber to the defaultNumber if it is free
stringToDouble(stringValue:EString, doubleValue:EDouble)	converts a stringValue into a double value
stringToInt(stringValue:EString, intValue:EInt)	Converts a stringValue into an int value
multiply(operand1:EDouble, operand2:EDouble, result:EDouble)	Multiplies both operands for the result
divide(numerator:EDouble, denominator:EDouble, result:EDouble)	Divides the numerator by the denominator and the result contains the solution of the operation
add(summand1:EDouble, summand2:EDouble, result:EDouble)	Adding both summands for the result
sub(minuend:EDouble, subtrahend:EDouble, result:EDouble)	Subtracting the subtrahend from the minuend for the result
max(a:EDouble, b:EDouble, max:EDouble)	Selects the maximum value from the two given double variables
setRandomString(a:EString)	Which sets a Variable to a random string. If it already has a value (B) then nothing is done and the condition is still satisfied

7.3 Stop criterions overview

Syntax	Description
stop.setMaxElementCount(int maxElement-Count)	Sets a stop criterion for the maximum number of elements allowed for the triple
stop.setMaxRuleCount(String ruleName, int maxNoOfApplications)	Sets a stop criterion for the defined number of applications for the named rule
stop.setMaxSrcCount(int maxSrcCount)	Sets a stop criterion for the maximum number of elements allowed on the source side of a model instance
stop.setMaxTrgCount(int maxTrgCount)	Sets a stop criterion for the maximum number of elements allowed on the target side of a model instance
stop.setTimeOutInMs(long timeOutInMs)	Sets a stop criterion after the defined time in milliseconds has passed

8 Troubleshooting

What are the best practices when specifying graph transformation rules?

- Use lowerCamelCase for patterns, rules, nodes, edges, and parameter names.
- Name the entities to methods and describe their purpose.
- Do not use node names like x, y, and z. Use descriptive names as you would do when writing a program. So, you do not lose track of their purpose. As the generated API uses the node names in methods this will lead to more traceable method names rather than getX().
- Try not to put everything into one file. You can reference patterns, rules, and conditions from other files within the same package.

I cannot specify a certain condition with the textual syntax. What can I do?

If your constraint cannot be expressed with eMoflon::IBeX application conditions, you can always apply additional arbitrary filtering conditions on the matches you get via the API using Java code.

My meta-model code is not in a subpackage named the same as the metamodel. How can I fix the error in the generated API code?

By default, eMoflon::IBeX assumes that the code for your metamodel is in a package named like the package name in the Ecore file. If that is not the case for your metamodel, you can fix that with one of the following proposals:

- Create a moflon.properties.xmi in the project's root directory.
- Create a new "Import Mapping" within the "Moflon Properties Container".
- Set the key to the URI as you reference your metamodel in the .gt files.
- Set the value to the name of the package containing generated code for your metamodel.

An EPackage seems to be in a different package. How can I fix such imports in the generated API code?

Similar to the question above, this problem can be resolved as follows:

- Create a moflon.properties.xmi in the project's root directory.
- Create a new "Import Mapping" within the "Moflon Properties Container".
- Set the key to the error value of the EPackage import, which you would like to correct.
- Set the value to the corrected value of the import. Rebuild and check if the fix is as desired.

After the project build, errors the project has error markers. What do I have to do to resolve this issue?

- If the MANIFEST.MF is affected by the errors, try to build the project again and checks whether the error markers are removed. Try to build the project multiple times since some relations might not be built. Check for missing dependencies if rebuilding does not help.
- If the plugin build creates the markers for missing packages before they are generated by the GT build, the error markers are removed as recently as the project is built again.
- If the errors are in generated code, check that you have added the metamodel project as a dependency of your project. Otherwise, Eclipse cannot find the metamodel classes on the build path of the GT project and reports errors.
- If none of the proposals above worked, you can try deleting the src-gen folder and building the whole project again.