

eMoflon::IBeX



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

---

## Table of Contents

---

1	Introduction.....	1
1.1	Installation Guide.....	1
2	Creating a metamodel.....	4
2.1	Graph transformation with eMoflon .....	11
2.2	Creating a graph transformation project.....	11
2.3	Rules and Patterns .....	12
2.4	Application conditions .....	14
2.5	Parameters and attribute constraints.....	15
2.6	Finishing the graph transformation ruleset.....	17
2.7	Java implementation of the ruleset.....	21
2.8	Arithmetic Extension.....	24
2.9	Stochastic extensions.....	27
2.10	Static probabilities .....	28
2.11	Stochastic functions.....	29
2.12	Number Generation.....	30
2.13	Appendix for graph transformation.....	32
3	Bidirectional Transformation with Triple Graph Grammars .....	39
3.1	Administration metamodel.....	39
3.2	Creating a TGG project.....	40
3.3	Rules.....	44
3.4	Attribute Conditions .....	55
3.5	Running the TGG Project .....	58
3.6	Debugging in TGGs.....	63
3.7	Adding additional Information.....	64
3.8	Appendix for graph transformation.....	69
4	Troubleshooting .....	73

---

# 1 Introduction

---

Welcome to our guide for metamodeling with **eMoflon::IBeX**. In the first part of the tutorial, we will show you how to install the required software for using eMoflon::IBeX and how to create the 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 with Java. A fundamental understanding of metamodeling might come in handy but is not required.

---

## 1.1 Installation Guide

---

### Eclipse Modelling Tools:

The very first thing you need is a running version of **Eclipse Modelling Tools** which can be downloaded on the official Eclipse website: [Download link \(https://www.eclipse.org/downloads/packages/release/2021-03/r/eclipse-modeling-tools\)](https://www.eclipse.org/downloads/packages/release/2021-03/r/eclipse-modeling-tools) The installation of Eclipse should be self-explanatory, if you are unsure about the installation just follow the instructions on the Eclipse website.

### Eclipse Modelling Framework:

The Eclipse Modelling Tools version you just downloaded 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 the tutorial.<sup>1</sup>

### GraphViz:

Another standalone software you will need for the visualization of the created metamodels is GraphViz. We highly recommend using version **2.38 of GraphViz** since newer releases have not been

---

<sup>1</sup> See: Vogella EclipseEMF article (<https://www.vogella.com/tutorials/EclipseEMF/article.html>)

working reliably. You can download GraphViz [here](http://www.linuxfromscratch.org/blfs/view/7.9/general/graphviz.html) (<http://www.linuxfromscratch.org/blfs/view/7.9/general/graphviz.html>).

## PlantUML :

If you installed these two programs, you are ready to install the remaining software in Eclipse. You will also need PlantUML for the visualization of our class diagrams. The easiest way to install PlantUML is to install it via the **integrated Eclipse Marketplace**. The Marketplace can be accessed over the help drop-down menu where you select the Marketplace. A new window should pop up, which presents an overview of available software plugins. Now simply type **PlantUML** in the search window on the very top and **install the PlantUML plugin**. Chart 1 provides a cumulated view of the previous steps.

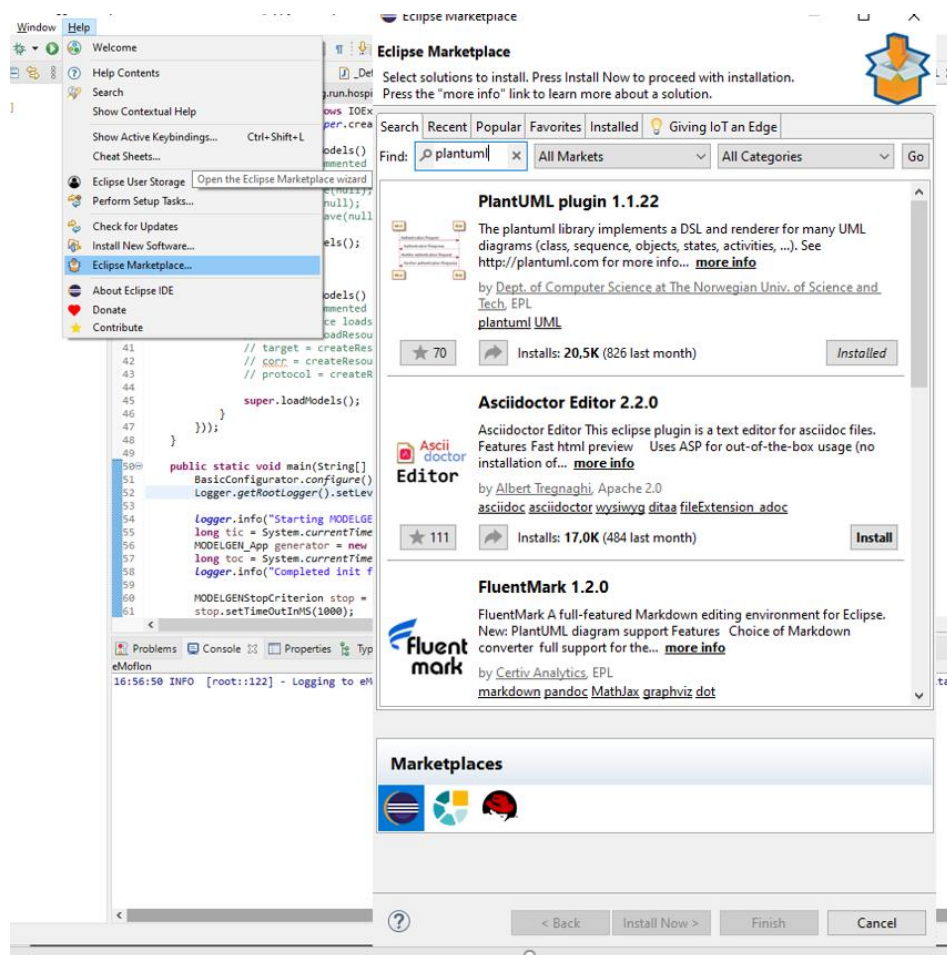


Chart 1: Screenshot of PlantUML installation.

## eMoflon::IBeX

Now we will install eMoflon::IBeX. You need to click on Help again, but now you select the **Install New Software** field. Another window should pop up and 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 Chart 2 presenting eMoflon::IBeX and the different modules in a drop-down menu. For this tutorial, you will need the Modules Democles and HiPE, although installing the whole suite does no harm. Start the installation by **clicking next, accepting the license agreement, and pressing install anyway afterward**. Throughout the process of installing, Eclipse will require a restart. If Eclipse restarts without any errors, you should be good to go and everything necessary is installed.

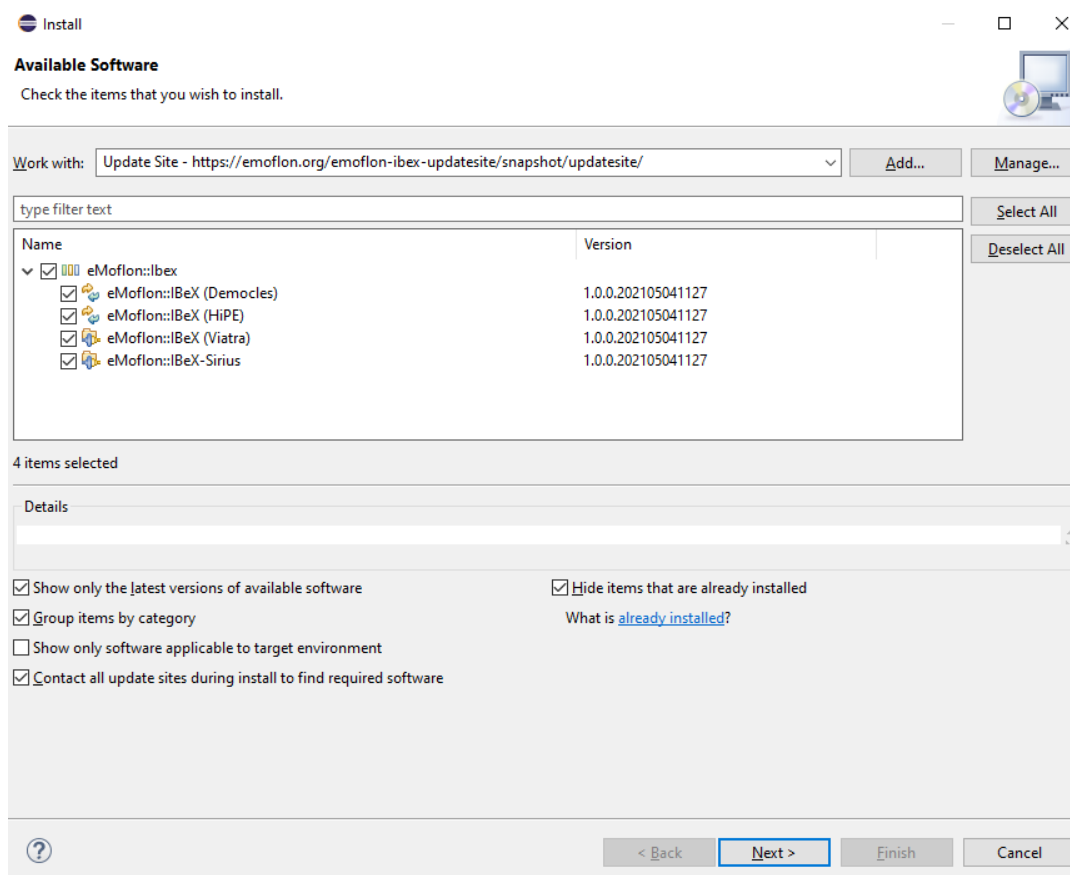


Chart 2: Screenshot of available eMoflon::IBeX modules.

Both Democles and HiPE are Eclipse projects which include pattern matching engines while Viatra provides a framework with an editor for model queries and a code generator to implement the model queries easily into java code. 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 Democles and HiPE. In contrast to Democles, HiPE is a parallel pattern matcher.

## 2 Creating a metamodel

A first question which you might wonder about is, what is metamodeling? Metamodeling can be described as the way of designing models by defining abstract rules and structures a concrete model has to fulfill.<sup>2</sup>

Now it is time to get to know the functionalities of eMoflon. First, you should add the eMoflon toolbar to your Eclipse as shown in chart 3. Go to the **Window Tab click on Perspective → Open → PerspectiveOther**. The eMoflon toolbar icon is highlighted in the red square in chart 3:

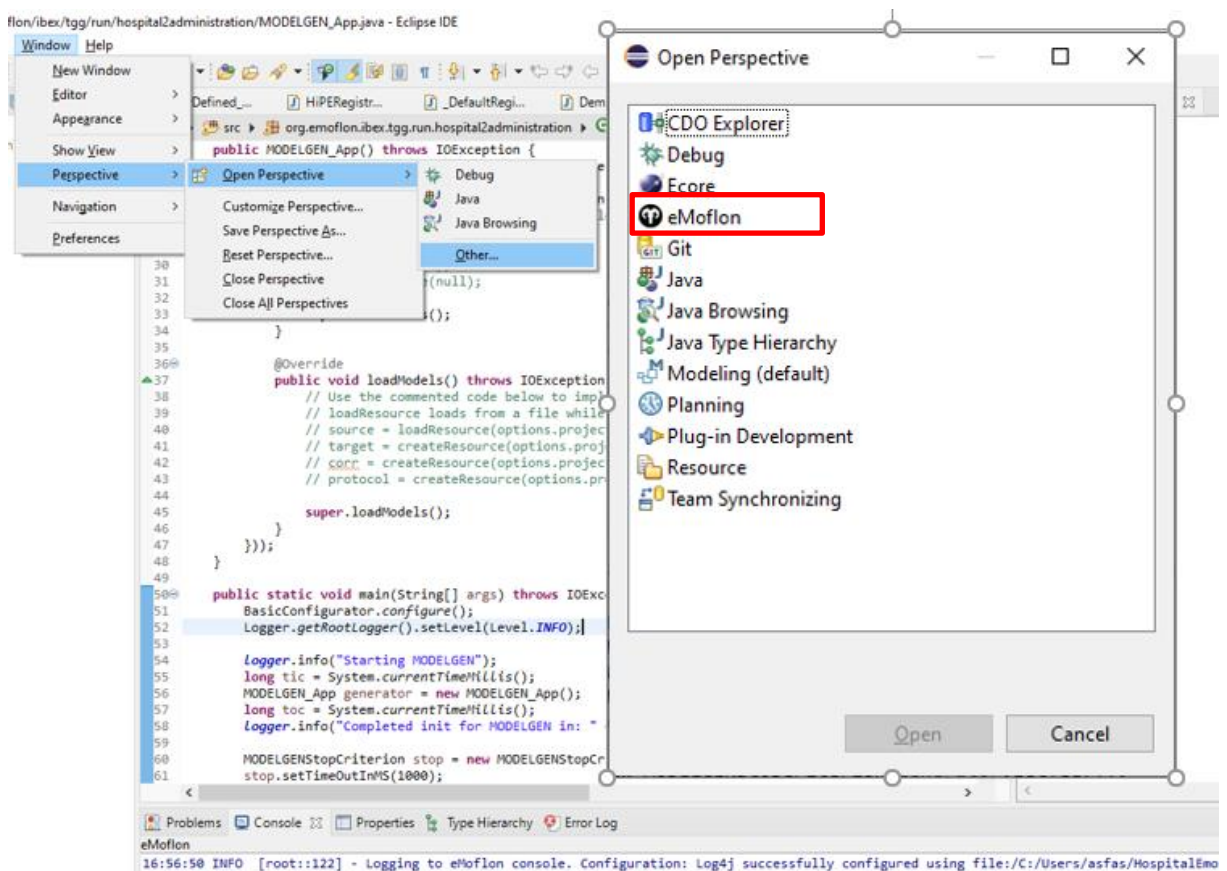
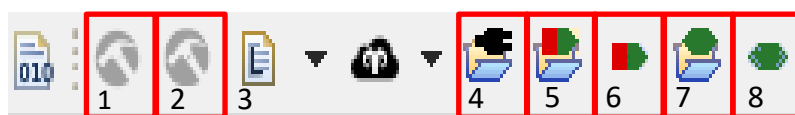


Chart 3: eMoflon toolbar perspective

This is how the eMoflon perspective should look like. Let us take a look at the functions we will need throughout this tutorial:



1. Click this button will build the selected projects fully.

<sup>2</sup> See. Sprinkle, Rumpe et. Al 2014.

2. Click 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.

Time to create a new metamodel. Click on the button to create a new eMoflon EMF Project and choose a new name for your project. For this tutorial, we will be trying to create a hospital scenario and the corresponding administration to show you the basic functions of eMoflon::IBeX. You should stick to the name conventions, we are using, since inconsistent names may lead to errors later. Please select **Generate default ecore model**. Please give the project the name **HospitalExample** as shown in chart 4. After you have pressed the finish button a new file should appear in the model folder. If you cannot see a file, try to refresh the folder.

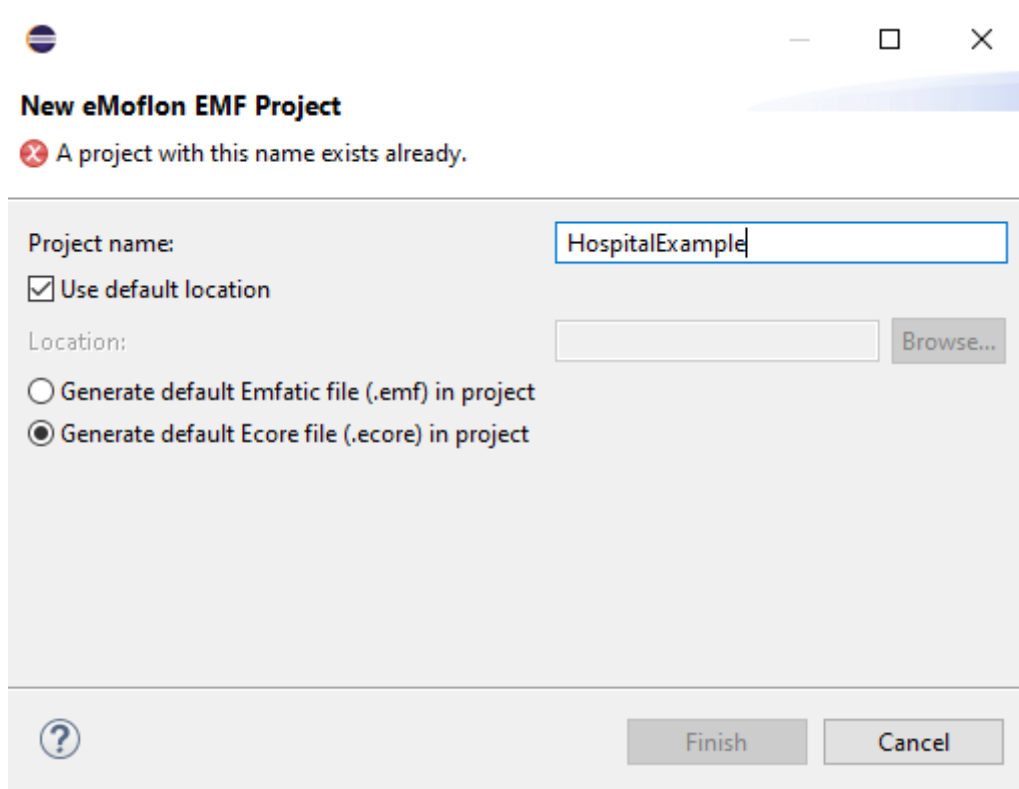


Chart 4: New eMoflon EMF Project.

Our goal for this part is to create a specification, which is like a UML model and allows us to create the corresponding java code via the EMF framework. The first class we need for our tutorial is the hospital itself, which will be the container for the following subclasses. To create the Hospital class, you need to **right-click on the HospitalExample package** and **select EClass** in the drop-down menu as a new child of the Metamodel (Chart 5).

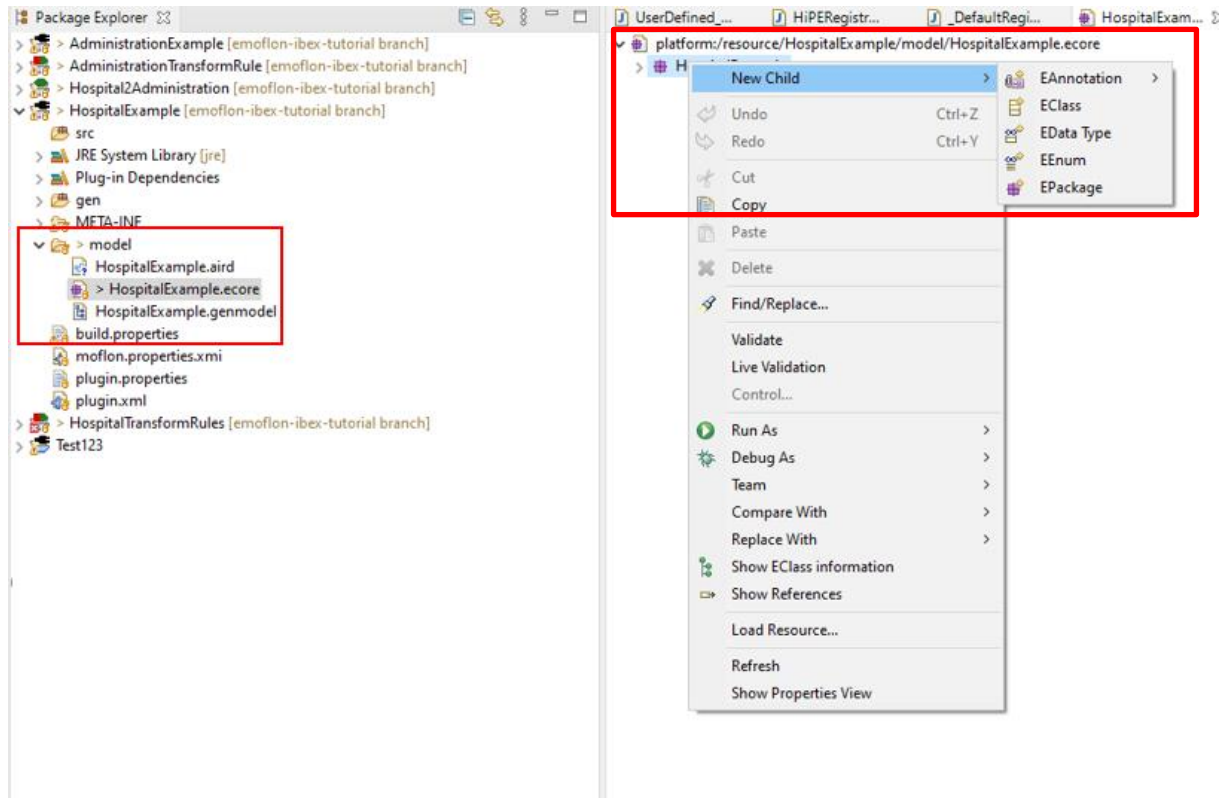


Chart 5: Overview for the fresh metamodel and class creation.

As you can see, it is also possible to create an enumeration, a data type, and other packages to split up your project. You maybe have noticed that every option begins with a capital E. This is merely the name convention used in EMF and it will be also used for things such as variable types. For example, a variable of the type of 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 just created and select properties on the bottom of the list or simply double left-click on the child. Let us name the class accordingly and **type in "Hospital"** in the name field. Within the next few steps, we will create our hospital meta-model.

Chart 6 on the next page provides a UML-based visualization of the finished model you will create throughout this part of the tutorial:



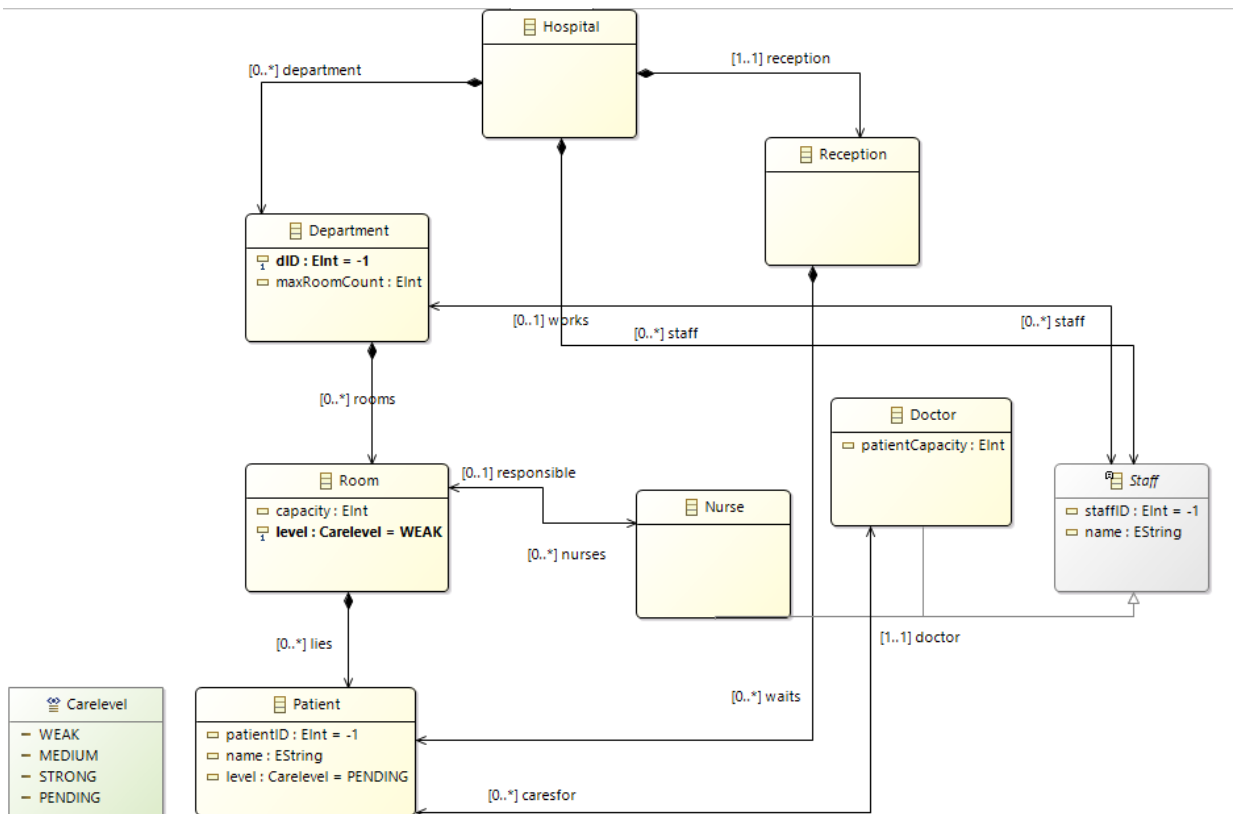


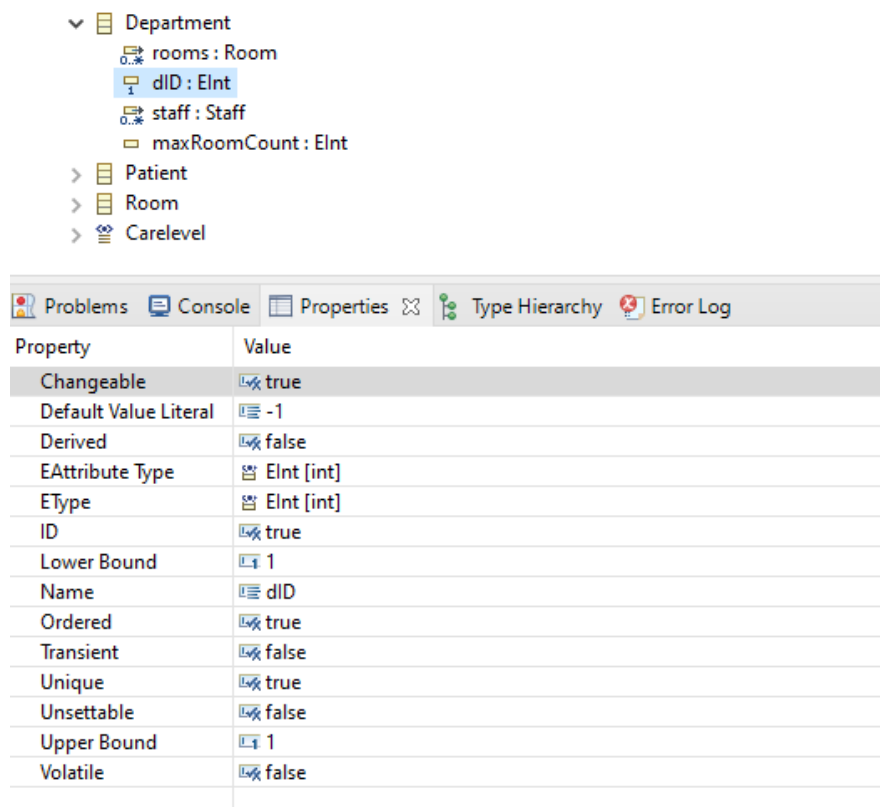
Chart 6: Complete hospital ecore Model.

To fill our Hospital, we start by adding a reception class and a department class. After creating the classes, we have to connect them with our hospital. This can be done by creating a new child in our hospital class. To connect them, you have to create an **EReference child**. The properties of a reference offer a large variety of options you can configure but let us keep it simple for now. To maintain comprehensibility and simplicity in your model we recommend using the name of the class that we are referring to or the purpose of the connection. Hence, we chose reception as the name as well. 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**. The department reference can be created analogously.

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**. Since we want every department to have an ID, we modify the relations in the same manner as our reference.



The screenshot shows a software interface with a class hierarchy on the left and a properties table on the right.

**Class Hierarchy:**

- Department
  - rooms : Room
  - did : EInt
  - staff : Staff
  - maxRoomCount : EInt
- Patient
- Room
- Carelevel

**Properties Table:**

Property	Value
Changeable	true
Default Value Literal	-1
Derived	false
EAttribute Type	EInt [int]
EType	EInt [int]
ID	true
Lower Bound	1
Name	did
Ordered	true
Transient	false
Unique	true
Unsettable	false
Upper Bound	1
Volatile	false

Chart 7: Overview of properties for attributes in the metamodel.

The **maxRoomCount** already indicated 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. The rooms need a connection to the departments they are assigned to, the patients who are lying in the room, and the responsible staff.

After creating the infrastructure for our hospital, we need the persons 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 just like it is shown in chart 8. 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.

Property	Value
Abstract	<input checked="" type="checkbox"/> true
Default Value	<input type="text"/>
ESuper Types	<input type="text"/>
Instance Type Name	<input type="text"/>
Interface	<input checked="" type="checkbox"/> false
Name	<input type="text" value="Staff"/>

Chart 8: Abstract settings field in properties.

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 class Doctor to inherit the references and attributes of the abstract Staff class. 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.

Finally, we need patients in our hospital. Patients also have **a name, a patientID**, and the level which defines the **carelevel** assigned to a patient. 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**.

To identify the **responsible doctor**, please create a **bidirectional reference** to the Doctor class. To achieve bidirectional references **select** the reference **caresfor:Patient** in the **EOpposite** property. Now go to the doctor class and **assign** the **doctor:Doctor** references as an **EOpposite** as well.

For the care level, we want to use an enumeration. The Carelevel enumeration can be created as a **child of the Hospital package**. The possible care levels are **WEAK, MEDIUM, STRONG, and PENDING** which can be added as children of the enumeration as **EEnum literals**. If you made it this far your hospital meta-model should be completed and the visualization in PlantUML should look like the right side of chart 9:

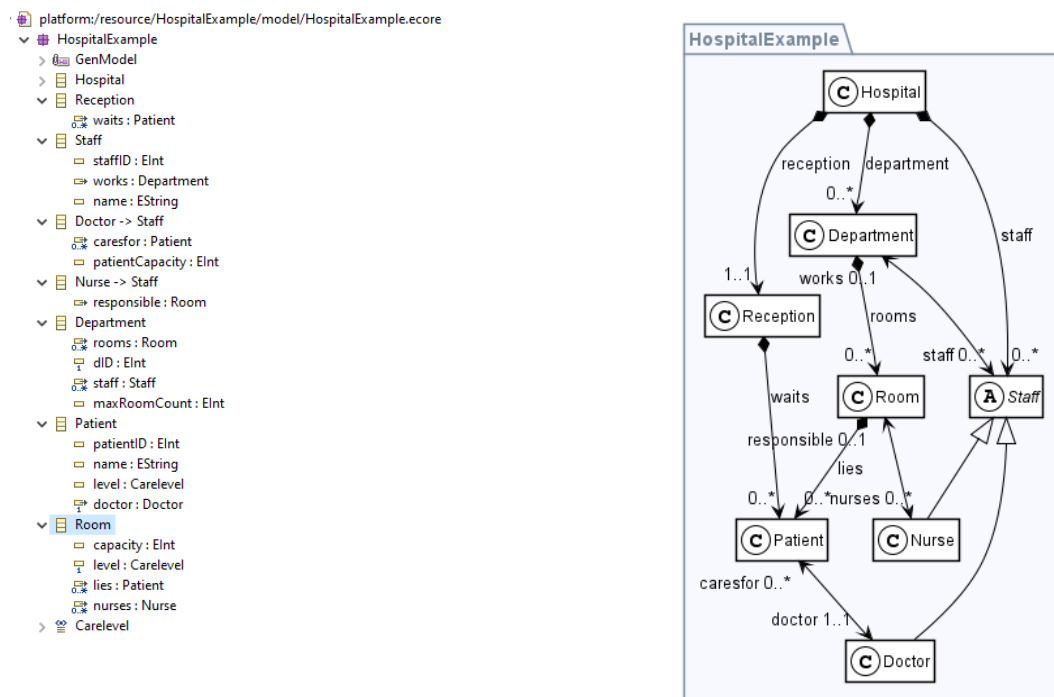


Chart 9: List with classes and their attributes and PlantUML visualization.

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 in the next chapter. Congratulations on completing the first part of this tutorial!

## 2.1 Graph transformation with eMoflon

In this second part of our tutorial, we will focus on graph transformation, and patterns to build and validate the hospital according to the constraints we previously specified in the metamodel.

## 2.2 Creating a graph transformation project

Let us get started by creating a new eMoflon Graph Transformation project. Please **click** on [the red and green arrow in the folder](#). Write **HospitalTransformRules** as the name and **press** the finish button to create the project. You will notice the Rules.gt 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.

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

Your file should be empty except for these two imports.

You might have noticed that the project has some [compilation errors](#). To solve the issues, you need to add the meta-model project as a dependency by opening the META-INF/MANIFEST.MF file. Now click on dependencies and add the HospitalExample from the drop-down menu. Chart 10 shows the different dialogue windows from left to right.

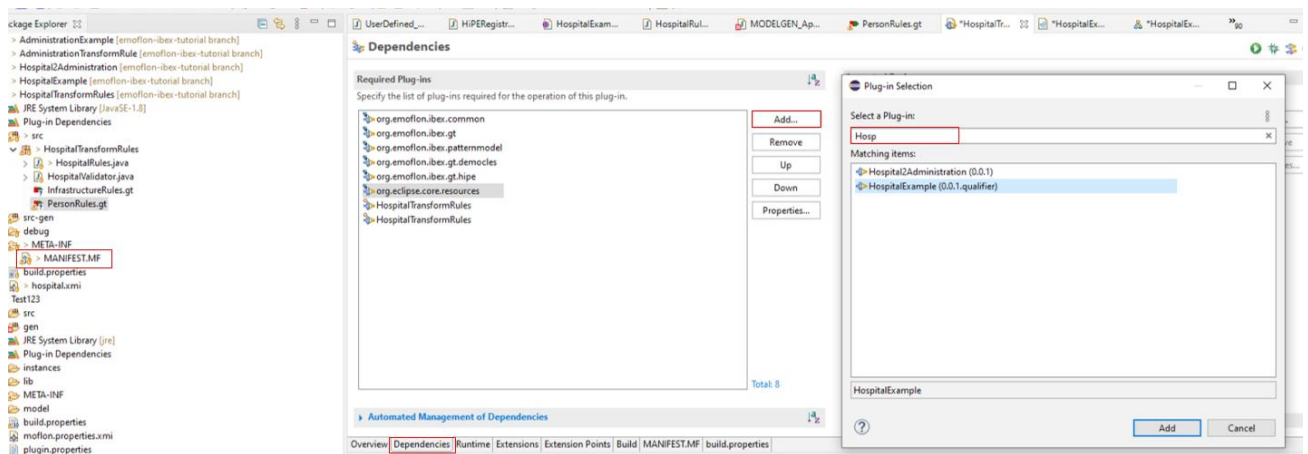


Chart 10: Management of dependencies for the MANIFEST.MF.

---

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.

---

## 2.3 Rules and Patterns

---

The GitHub site where you have found this PDF for the Tutorial also has different stages of the project stored for you in case you messed something up, skipped a part of the tutorial, or if you are unsure whether everything you did is correct. These different stages of the project are represented by different branches in the Git repository. Throughout this tutorial, we will be using four different branches which you can access to keep the project up to date for every section of the tutorial. Whenever you can switch to a new branch for a new section it will be announced at the beginning of the respective section. As an example, please use the branch ***Ecore+GT empty*** for the following section. This stage of the project includes the hospital ecore model and the graph transformation project you just created in the previous chapter but has some repetitive contents added to it so you do not get to learn the important functionalities more easily. Whenever you have to add code to the existing project it will be said explicitly and highlighted in the Java project.

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**. A graph pattern describes certain structures that must be present in a given instance graph. Consequently, a graph pattern matcher will find all sub-structures in the instance graph that match all structural constraints of a given pattern. Therefore, we will use the term graph pattern matches to refer to these matching instance graph sub-structures.

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.

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.

---

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 **++** and **--**. 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 none of the operators is used, the element is interpreted as context and colored as black. Every element that is not denoted with the **++** 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    }
```

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    }
```

A quick note on the visualization of PlantUML. When you click on the freshly created hospital rule for example 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. Chart 11 provides a visual example for the reception rule, which will be covered in a different version later.

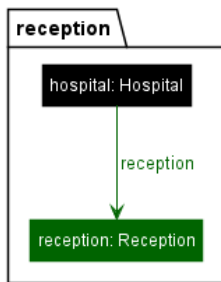


Chart 11: Visualization example of a reception rule.

## 2.4 Application conditions

Since our model requires only one hospital node, we should add a so-called application condition to prevent multiple applications of the hospital rule. A condition has to be defined and needs to be accessed by the keyword **when** once you are using it. To define a condition, you need to type the **condition** and its name. Please note there are two types of conditions you can either **forbid** the presence of matches of certain patterns or **enforce** them. An application condition that forbids the presence of matches is called a **negative application condition (NAC)** while a condition that enforces the presence of matches 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, simply add a **when** behind our hospital rule and **insert *forbidHospital*** as the name of the condition. Now we need to **specify** our condition to prevent the creation of another hospital object if we already have one. With our condition, we want to forbid the application of the rule if our ***findHospital*** pattern finds the structure ***hospital:Hospital***.

```

1  rule hospital() {
2      ++hospital: Hospital
3  }
4  when forbidHospital
5  condition forbidHospital = forbid findHospital

```

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 the type of reception. You can also write a condition to prevent the creation of multiple reception nodes. 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. You need to type in “++” once again but now we continue with a “--” since this the symbol for edges. 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 8.

```
6    rule reception() {
7        hospital: Hospital {
8            ++ -reception -> reception
9        }
10
11        ++reception: Reception
12    }
13    when forbidReception
14    condition forbidReception = forbid findReception
```

---

## 2.5 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. After creating these two rules we want to introduce some new syntax. As you remember both classes have attributes that need to be assigned. To assign attributes you need to use the operator “:=”. In the respective node and pressing the auto-completion behind the point should suggest the attributes defined in the meta-model. By typing “:=” you can **assign** a value to the attribute, e.g., the simplest assignment would be static. As an unrelated example to the project, you could assign the ID 1 to the department you create by writing:

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

Hence, 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 instead of the static assignment. The same has to be done for maxRoomCount.

```
1    rule department(dID: EInt, maxRoomCount: EInt)
2
3        hospital: Hospital {
4            ++ -department -> department
5        }
6        ++department: Department {
```

```

6          .dID:=param::dID
7          .maxRoomCount:=param::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:=param::cap
19          .level:=param::carelvl
20      }
21  }
22  }
23  #department.maxRoomCount>count(findRoomInDepartment)
24  }

```

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 dot 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.

Our goal for this pattern is to count the number of rooms which are assigned to a department. However, we want to avoid finding any matching room for a department node since this would lead

```

pattern findRoomInDepartment() {
    department: Department {
        -rooms -> otherroom
    }

    otherroom: Room {
    }
}

```

This pattern only finds any rooms in this department.

```

pattern findRoomInDepartment() {
    department: Department {
        -rooms -> room
    }

    room: Room {
    }
}

```

This pattern only finds a certain room in this department.

to  $n \cdot m$  matches for  $n$  rooms and  $m$  departments which would be of little use for us. Consequently, we want to **count** the matches for the rooms assigned to a **certain department** node. All we have to do to create the mapping between the nodes in the room rule and the *findRoomInDepartment()* pattern is to assign identical names for the nodes. The mapping itself is done by eMoflon. These kinds of pattern invocations are also called support pattern invocations just like NACs and PACs. You can note the difference when comparing these two slightly different patterns:

Hopping back to the visualization of PlantUML. When you click on the room rule you should now also see the parameters and attribute constraints we have added for the rule. Parameters and attribute constraints appear as black boxes on the top left and show you the type of parameters we have initialized with this rule and the requirements for our attribute constraints. You can also see the *findRoomInDepartment* pattern visualization which we require for our attribute constraint in chart 12.

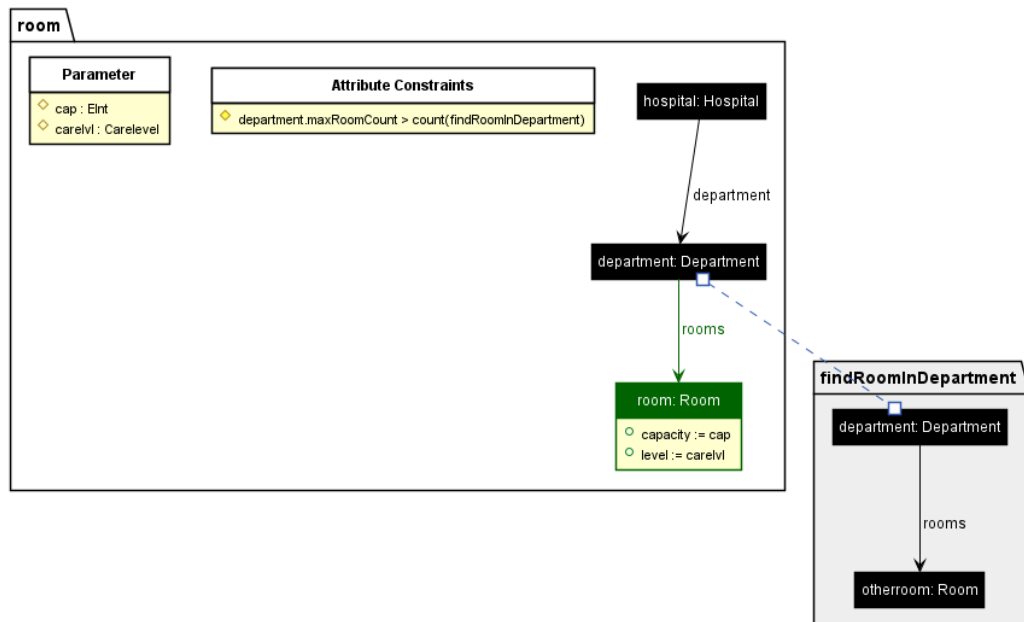


Chart 12: PlantUML visualization of the room rule and the corresponding pattern.

For a complete listing of all possible operators and instructions, that are allowed in attribute constraints, [please refer to the appendix](#).

## 2.6 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 persons in the hospital. In our case, we have the following person types in our hospital: the patients, the doctors, and the nurses.

---

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 *carelevel* which requires a certain kind of treatment. The implementation of the patient rule can be seen on the next page.

#### Patient:

```
1    rule patient(name: EString, patientId: EInt, level: Carelevel) {
2        hospital: Hospital {
3            -reception -> reception
4        }
5
6        reception: Reception {
7            ++ -waits -> patient      }
8
9        ++patient: Patient {
10           .name:=param::name
11           .patientID:=param::patientId
12           .level:=param::level
13       }
14    }
```

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.

#### Staff:

```
15    abstract rule staff(name: EString, staffID: EInt) {
16        hospital: Hospital {
17            ++ -staff -> staff
18            -department -> department
19        }
20
21        department : Department
22
23        ++staff: Staff {
24            .staffID:=param::staffID
25            .name:=param::name
26        }
27    }
```

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. 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 the pattern works analogously to the pattern we defined for the rooms.

```

30  rule doctor(capacity: EInt)
31      refines staff {
32          ++staff: Doctor {
33              .patientCapacity:=param::capacity
34          }
35          department : Department {
36              ++ -staff -> staff
37          }
38      } when departmentWithoutDoctor = forbid doctorInDepartment

39  pattern doctorInDepartment() {
40      someDoctor : Doctor
41      department : Department {
42          -staff->someDoctor
43      }
44  }

```

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() refines staff {
46      ++ staff: Nurse {
47          ++ -responsible -> room
48      }
49
50      room: Room
51
52      department : Department {
53          -rooms->room
54          ++ -staff-> staff
55      }
56  }
57  when forbidNurse
58
59  condition forbidNurse = forbid findNurseInRoom
60
61  pattern findNurseInRoom() {

```

```

63         somenurse: Nurse {
64             -responsible -> room
65         }
66
67         room: Room {
68
69         }
70     }

```

Another important use case that one can think of is the dismissal of a patient, which can be modeled with a simple rule. For this rule, we want to remove a patient with a given *patientID* by using “-”. 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 85 and 88.

```

80     rule releasePatient(patientID:EInt){
81         -- patient:Patient{
82
83         }
84         room:Room{
85             -- -lies->patient
86         }
87         doctor:Doctor{
88             -- -caresfor->patient
89         }
90     }
91 }

```

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 keep in mind. 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         }
96
97         room: Room {
98             ++ -lies -> patient
99         }
100         #room.capacity>count(findPatientInRoom)
101     }

```

---

```

102     doctor: Doctor {
103         ++-caresfor->patient
104     }
105
106     #doctor.patientCapacity>count(findOccupiedDoc)
107
108     hospital: Hospital {
109         -reception -> reception
110         -department -> department
111     }
112
113     department: Department {
114         -rooms -> room
115     }
116
117     reception: Reception {
118         -- -waits -> patient
119     }
120 }
121 when patientWithDoc
122
123 condition patientWithDoc = forbid findPatientWithDoc
124
125 pattern findPatientWithDoc() {
126     somedoctor: Doctor {
127         -caresfor->patient
128     }
129     patient: Patient {
130     }
131 }
132
133 condition docWithPatient = enforce findPatientWithDoc
134
135 pattern findDocWithPatient() {
136     somedoctor : Doctor
137 } when docWithPatient

```

Finally, we have created all the rules we need to build our hospital. So far, we have only written the rules without applying them. This will be our next task.

---

## 2.7 Java implementation of the ruleset

---

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 *HospitalValidator.java* in the *HospitalTransformRules* package, please.

The HospitalValidator file creates an empty Model with the URI hospital.xmi:

```

1      public class HospitalValidator extends HospitalTransformRulesHiPEApp {
2
3          public HospitalValidator() {
4              createModel(URI.createURI("hospital.xmi"));
5          }
6
7      }

```

We use this simple class to initialize a new model instance with the name ***hospital.xmi***.

Let us inspect HospitalRules.java a bit more closely since there is much of interest going on. The first important thing we want to look at is in lines 4 and 8 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.

Another important thing to note is happening on line 18 where we save our hospital model. It is important to note that the hospital instance we have initialized in the HospitalValidator will not be stored anywhere. If we want to keep it for usage in the future, we have to ***save it with a separate command*** as we are doing it in line 18. The URI ***hospital.xmi*** is saved in the project folder of the ***HospitalTransformRules*** project.

```

1      public class HospitalRules {
2          ...
3          private Random rnd;
4          public HospitalTransformRulesAPI api;
5
6          public HospitalRules(final long rndSeed) {
7              rnd = new Random(rndSeed);
8              api = new HospitalValidator().initAPI();
9          }
10
11          public static void main(String[] args) throws IOException {
12              HospitalRules hospitalrules = new
13              HospitalRules("someSeed".hashCode());
14
15              hospitalrules.createHospital();
16              hospitalrules.validateHospital();
17          try {
18              hospitalrules.api.getModel().getResources().get(0).save(null);
19          } catch (IOException e) {
20              // TODO Auto-generated catch block
21              e.printStackTrace();
22          }
23          hospitalrules.api.terminate();
24      }
25
26      public void createHospital() {
27
28          api.hospital().apply();
29          api.reception().apply();
30          for(int i=0; i<4; i++) {
31              api.department(i+2, 4).apply();

```



```
32         }
33         for(int i=0; i<16; i++) {
34             api.room(4, Carelevel.get(rnd.nextInt(3))).apply();
35         } ...}
```

Let us take a closer look at the createHospital method starting at line 26. As you can see we are accessing our rules via the variable name **api.<rulename>.apply()**. 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 do we 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().apply();
api.hospital().apply();
```

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 persons of the type of patient, doctor, and nurse for our hospital as well as the departments and the rooms for our hospital.

The simplest 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 countPatientsInHospital = api.findPatient().countMatches();
System.out.println(countPatientsInHospital + " Patients are in the hospital right now");
```

---

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:

```
11 Patients are in the hospital right now
11 Patients are in a room
One instance of a hospital has been created
One instance of a reception has been created
At least one deparment instance has been created
16 nurses are in the hospital right now and 16 nurses are busy
At least one doctor is in the hospital
4 doctors are in the hospital right now and 4 doctors are busy
At least one patient is in the hospital
The hospital consists of at least one room
11 Patients are in the hospital right now and 11 patients are in a room
```

Chart 13: Console output for the graph transformation project.

If you get a console output like the one in chart 13 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.

---

## 2.8 Arithmetic Extension

---

The sections 2.8 to 2.12 of this tutorial introduce other extensions independently of our hospital example. This part uses a simple network model as an example instead. The metamodel view can be seen in chart 14. The network model contains a certain number of devices and a random number of connections between the devices. A simple rule for this task could be written as follows:

```
1    rule connect{
2        network: Network{
3            ++ -connections -> connection
4        }
5        to: Device{
6            ++ -connections -> connection
7        }
8        from: Device{
9            ++ -connections -> connection
10       }
11     ++ connections: Connection
12     ++ -devices -> from
13     ++ -devices -> to
14     }
15     }when noConnection
```

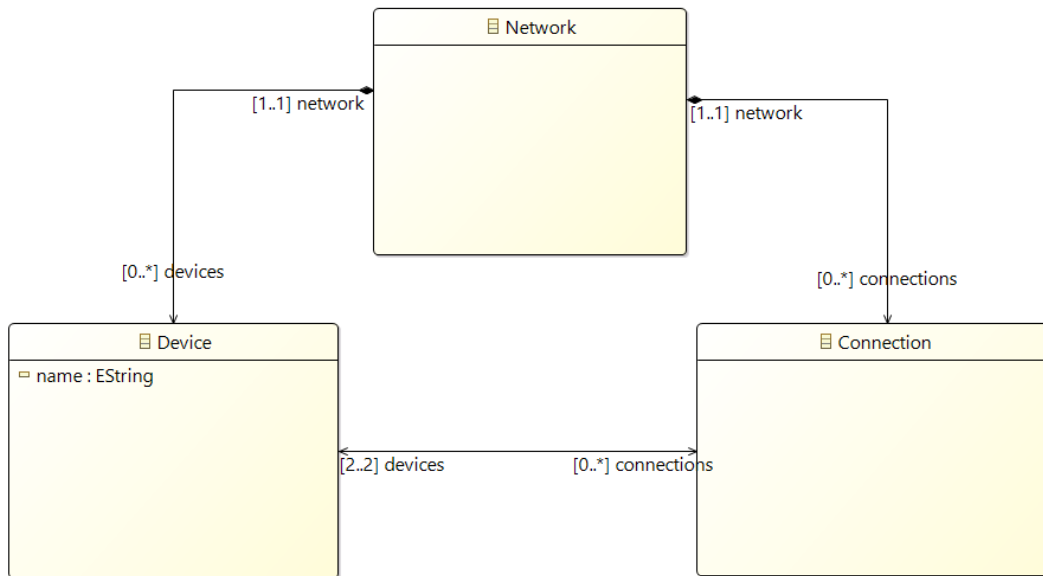


Chart 14: Network metamodel.

With this rule, it is possible to create a connection between two devices while ensuring that there was no connection between them before the rule execution, but it will always be applied to the underlying graph.

One feature included in eMoflon is the arithmetic calculation of values. Arithmetic expressions can be used in both the stochastic extension which will be covered later and for the calculation of values in graph transformation rules and patterns. The arithmetic extension allows a variety of arithmetic functions which are depicted in chart 15. With these functions, it is possible to write a variety of equations. The constraints of the functions will also be automatically checked by eMoflon.

Keyword	Mathematical Function	Constraint
a + b	$a + b$	
a - b	$a - b$	
a * b	$a \times b$	
a / b	$a/b$	$b \neq 0$
a % b	$a \% b$	
a ^ b	$a^b$	
exp(a)	$e^a$	
log(a)	$\log(a)$	$a > 0$
ln(a)	$\ln(a)$	$a > 0$
sqrt(a)	$\sqrt{a}$	$a > 0$
abs(a)	$ a $	
cos(a)	$\cos(a)$	
sin(a)	$\sin(a)$	
tan(a)	$\tan(a)$	

Chart 15: Arithmetic functionalities in eMoflon

When generating the network model using the Waxman algorithm which is an iterative method to determine the eigenstates of a Hamiltonian operator without using the diagonal matrix.<sup>3</sup> Then it is necessary to use the arithmetic expansion by defining the probability of the connect rule as  $p = \beta e^{-\frac{d}{\alpha d_{\max}}}$  where  $\alpha, \beta \in [0,1]$ ,  $d$  is the distance between two devices and  $d_{\max}$  is the maximum distance between two devices.

The connect rule is then expanded to:

```

23     rule connect{
24         ...}
25         when noConnection @network.beta*
25         exp(-sqrt((from.x-to.x)^2
26         +(from.y-to.y)^2))/
27         (network.alpha*network.maxDistance))

```

<sup>3</sup> See Chamberlain et. Al. 2018 (<https://iopscience.iop.org/article/10.1088/2399-6528/aaea3#:~:text=The%20Waxman%20algorithm%20%5B1%5D%20is,well%20as%20non-Hermitian%20operators.>)

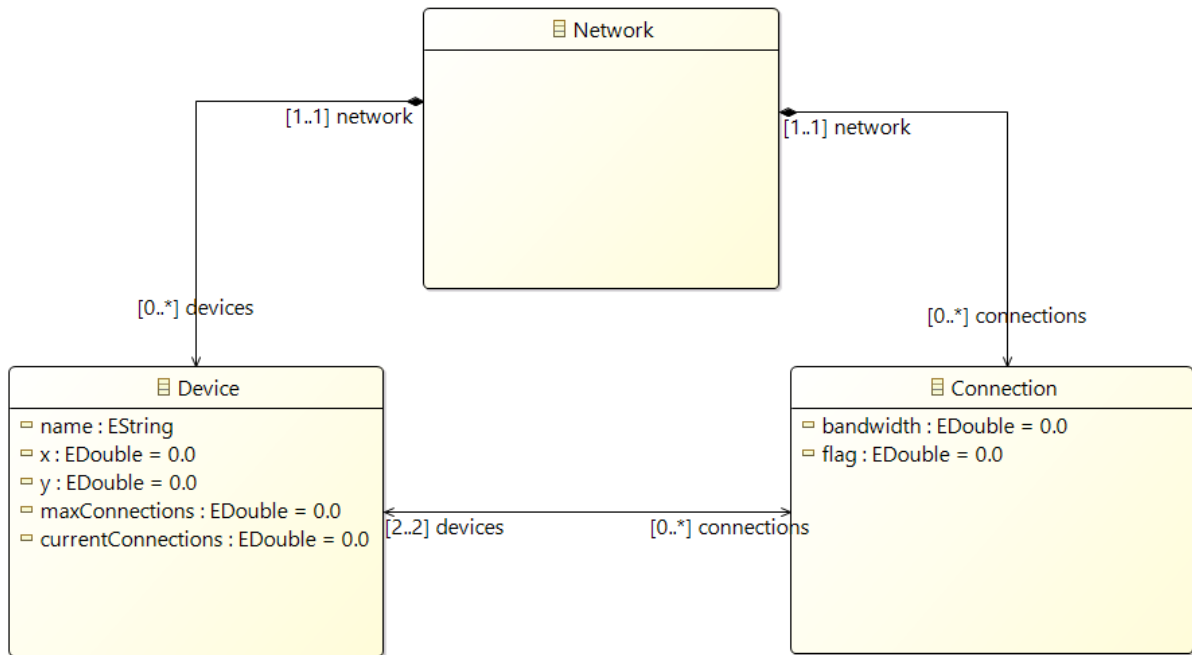


Chart 16: Expanded metamodel

With the previous **connect rule**, it is possible to generate a variety of random network models while writing just a few code lines. The arithmetic extension can also be used for conditional statements in rules or patterns.

## 2.9 Stochastic extensions

The probability of a rule can be written in two ways: With an arithmetic expression as we have seen in the previous chapter or with a stochastic function. Please note, only rules can have a probability since patterns are always deterministic. The probability is written at the end of the rule after a **@** and adds a probability to the condition of a rule. A simple example of a rule with a probability could be written as below:

```

1    rule connect{
2    }when noConnection @ 0.1
  
```

This rule will be applied with the probability  $P(r) = 0.1$ . The probability of the rule is now saved in a **StaticProbability** class, which itself is saved in the generated connect rule class. When using the stochastic extension in eMoflon::IBeX it is necessary to apply the rule with **.applyStochastic()** or

**.applyStochastic(M match)**, otherwise when using the functions **.apply()** or **.apply(M match)**, eMoflon::IBeX will apply the rule deterministically. When using **applyStochastic** it will return the corresponding match of the rule with a probability of 0.1 and with 0.9 an **optional.empty()**.

## 2.10 Static probabilities

The rule application probability may also depend on parameterized or non-parameterized attributes.

The network model and connect rule is expanded to make the probability dependent on the number of the current and maximum number of connections of a device:

```
1    rule connect{
2    }when noConnection @((to.maxConnections-
to.currentConnections)/.maxConnections)
```

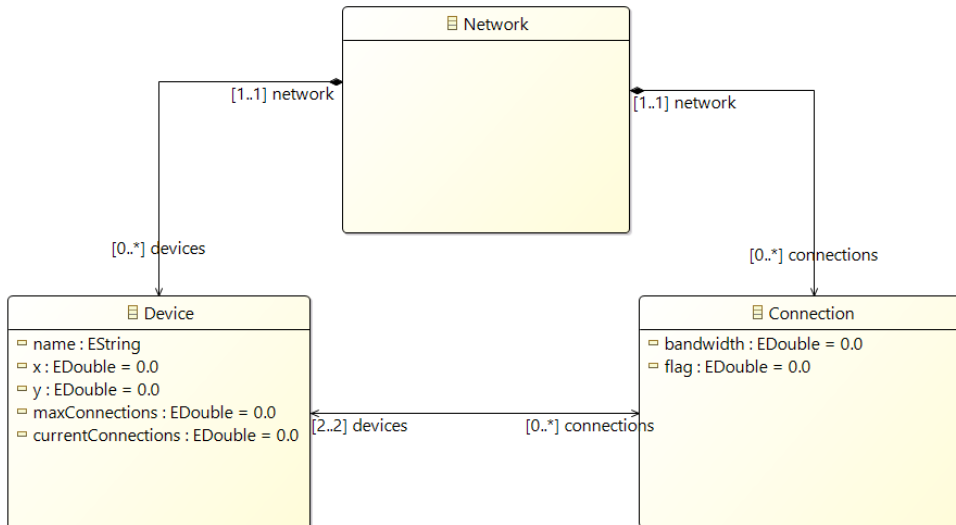


Chart 17: Expanded metamodel for static probabilities.

A rule is less likely to be applied the more connections a device has. It will also generate a specific probability class named **ConnectRuleProbability**. Generally, all probabilities that are dependent on a parameter will generate a class. If there is no dependency, then the StaticProbability class will be used for the probability calculation. The newly created probability class is depicted in chart 18.

```

/**
 * The probability class for the rule ConnectRule; calculates the probability
 * that the rule will be applied
 */
public class ConnectRuleProbability implements Probability<ConnectMatch, ConnectRule>{

    Random rnd = new Random();

    @Override
    public double getProbability(ConnectMatch match){
        if((match.getTo().getMaxConnections()!=0.0){
            if(((match.getTo().getMaxConnections()-match.getTo().getCurrentConnections()) / match.getTo().getMaxConnections())>= 0.0 &&
                ((match.getTo().getMaxConnections()-match.getTo().getCurrentConnections()) / match.getTo().getMaxConnections()) <= 1.0){
                return (match.getTo().getMaxConnections()-match.getTo().getCurrentConnections()) / match.getTo().getMaxConnections();
            }
            else{
                throw new IllegalArgumentException("The probability of the rule connect needs to be a value between zero and one");
            }
        } else{
            throw new IllegalArgumentException("There was an error with the arithmetic expression when calculating the probability");
        }
    }

    @Override
    public double getProbability(){
        return 0;
    }
}

```

Chart 18: ConnectRuleProbability class.

All probability classes implement the Probability interface, which declares the methods **getProbability(M match)** and **getProbability()**. A generated probability class can only receive correct values when using the **getProbability(M match)** method since it needs the match to get the values of the parameters. When using the **getProbability()** method it will simply return the value 0. The StaticProbability class can use both methods since it is not dependent on the match. When using parameterized probabilities, there is a need to pay attention to stochastic and arithmetic constraints. For example, the earlier defined probability needs to be a decimal value between zero and one and the arithmetic expression cannot be divided by zero. These constraints will be automatically checked in the probability class, but the user is responsible for not violating any constraints.

Ultimately, all arithmetic functions that are supported in eMoflon:IBeX can be used to define a probability for a rule.

## 2.11 Stochastic functions

Another way to describe the probability of a rule is with a stochastic function. With these functions, it is possible to define a randomly generated probability or to describe a probability defined as  $P(X < k)$ . The stochastic functions supported by eMoflon:IBeX and their constraints, which are automatically checked, are shown in chart 19.

Keyword	Distribution	Constraint
N(mean, sd)	Normal distribution	$sd \geq 0$
U(minValue, maxValue)	Uniform distribution	$minValue \leq maxValue$
Exp(lambda)	Exponential distribution	$\lambda > 0$

---

Chart 19: Table with all supported functions for stochastic functions

If the model generator should create connections with a random probability  $p \sim N(0,1)$ , then the connect rule is written as:

```
1    rule connect{
...
2    }when noConnection @N(0,1)
```

The probability of the rule will be calculated in eMoflon as  $p = \text{random.nextGaussian}()$ . If the probability should not be generated randomly but calculated, then it could also be defined as  $p = P(X < k)$ . Then it is necessary to define a new parameter k that is written after the distribution. For example, if the probability of the connect rule should be anti-proportionally dependent on the current number of connections, then the rule will look like this:

```
3    rule connect {
...
4    }when noConnection @((to.maxConnections -
    to.currentConnections)/to.maxConnections)
```

The probability is now defined as  $P(X < \text{to.maxConnections} - \text{to.currentConnections})$  and it will be calculated in eMoflon with the Apache Math library function *normalDistribution.cumulativeProbability (to.maxConnections - to.currentConnections)*.

---

## 2.12 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 connect{
2
3        ++ connections: Connection
4        flag := U(1,5)
5
6        ++ -devices -> from
7        ++ -devices -> to
8    }when noConnection @N(0,1) to.maxConnections-to.currentConnections
```



---

By assigning  $U(1,5)$  to the flag attribute it will get a uniform distributed value between 1 and 5. 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 below 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  $b \sim N(10,15)$ :

```
9      rule connect{
10
11          ++ connections: Connection
12          ++ -devices -> from
13          ++ -devices -> to
14          .flag := U(1,5)
15          .bandwidth := +N(10,15)
16
17      }when noConnection @ N(0,1) to.maxConnections-to.currentConnections
```

The range function will return the absolute or negative absolute of the generated value to make it positive or negative.

---

## 2.13 Appendix for graph transformation

---

### 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.

### Nodes

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.

### Edges

Edges also should have names that describe the type of connection between the nodes for reasons of traceability. Chart 20 shows a visual example of context nodes and context edges.

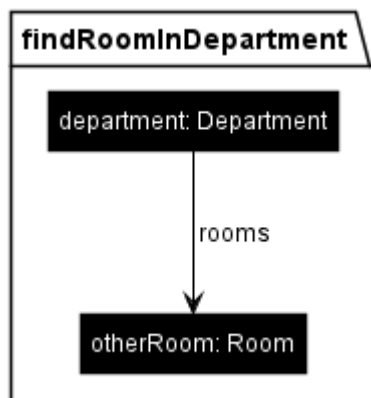


Chart 20: Example pattern for findRoomInDepartment.

### 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 enum:: for auto-completion)
  - parameters (type param:: 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 that 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. Furthermore, in concrete rules, the types of the created nodes are not allowed to be abstract.

### **Graph Conditions**

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.

---

## API methods

- **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.
- **Void terminate():** Terminates the interpreter
- **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 which is SPO but can be changed for the whole API as well.
- **PushoutApproach setPushoutApproach(PushoutApproach pushoutApproach):** Sets the pushout approach. This overwrites the default value on the API level.
- **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
- **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.
- **Double getTotalSystemActivity(Boolean doUpdate):** Helper method for the Gillespie algorithm; counts all the possible matches for rules in the graph that have a static probability

---

## Pattern methods

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

- **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 countMatches():** Returns the number of matches for the pattern.
- **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.

---

## Rule methods

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

- **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(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.
- **void enableAutoApply():** Enables instant automatic rule application: As soon as a match for the rule is found, the rule will be applied.
- **void disableAutoApply():** Disables instant automatic rule application.
- **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.

## 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 /*--- pattern/rule with parameters -----*/
4
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> [:=] enum::<VALUE>
14     .<attribute-name> [:=] param::<parameter-name>
15     .<attribute-name> [:=] <node-name>.<attribute-name>
16
17 /* References to other nodes */
18     [++|--] -<reference-name> -> <node-name>
19 }
20 }
21
22 /* Additional (application) conditions for pattern/rule: Combine conditions via OR */
23     [when <condition-name> [|| <condition-name> [|| ...]]
24
25 /* Probability of the rule; cannot be used on patterns */
26     [@ <arithmetic-expression> | [N|U|Exp](<arithmetic-expression>
27     [, <arithmetic-expression>)] [<arithmetic-expression>]]
28
29 /*--- conditions to be used in patterns/rules -----*/
30 /* Ensure that a certain pattern can be matched (positive application condition) */
31     condition <condition-name> = enforce <pattern-name>
32
33 /* Ensure that a certain pattern cannot be matched (negative application condition) */
34     condition <condition-name> = forbid <pattern-name>
35
36 /* Combine conditions via AND */
37     condition <condition-name> = <condition-name> && <condition-name>
```

---

38

39/\*----- summary of the arithmetic extension -----\*/

40     <arithmetic-expression> = <constant>|<node-name>.<attribute-name>|

41     <arithmetic-expression> [+|-|\*|/|%|^] <arithmetic-expression>|

42     [exp|log|ln|sqrt|abs|cos|sin|tan] (<arithmetic-expression>)



### 3 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.

#### 3.1 Administration metamodel

Let us look at the visualization of the **administration metamodel in chart 21**. 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.

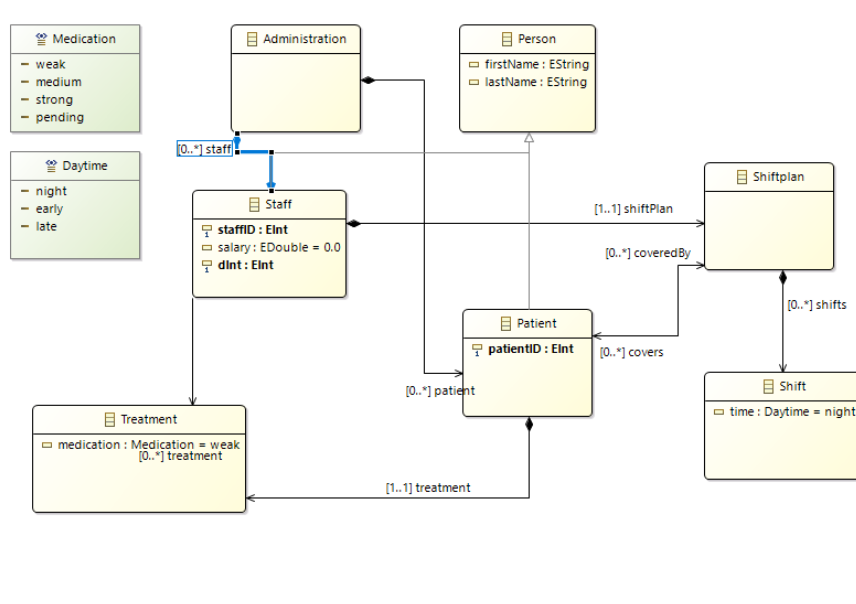


Chart 21: Visualization of administration metamodel

---

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](#).

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

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.

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 **shifts** 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. 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.

---

### 3.2 Creating a TGG project

---

For this section, we will be using the branch EcoreOnly2 from our Git repository. If you skipped the creation of the metamodel or you are unsure whether your model works correctly, please use this branch of our repository for the next section.

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.

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 the [symbol with green rhomb and the folder](#). Give it the name **Hospital2Administration** and select the option Project with **preselected metamodel** and 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 chart 22 and select **platform:/resource/HospitalExample/model/HospitalExample.ecore**

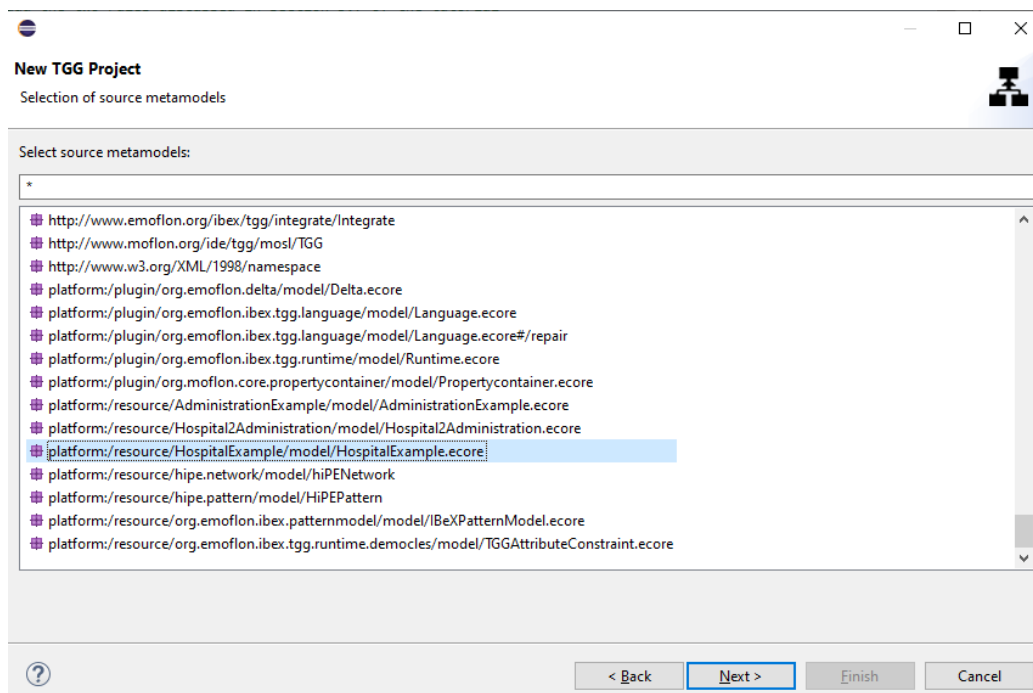


Chart 22: Selection option for the source metamodel

The second model to select will be our target model and you might have already guessed it we are selecting the **platform:/resourceAdministrationExample/model/Administrationexample.ecore** we created just created in the previous chapter. Press Finish and wait until the project is created. 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** and the **.tgg package**, you will find the **Schema.tgg**. 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.tgg also completes our triple of metamodel graphs. Our **triple** consists of **the hospital metamodel**, the **administration metamodel**, and finally, the **correspondence metamodel** which will be linking the target and source metamodel.

There are three important parts in the Schema file shown in the code snippet on pages 43 and 44. Some elements are missing in the Schema.tgg and have to be added throughout the next section. Do not worry we will explain everything in detail and walk you through every step.

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 6 to 12 where the **HospitalExample** metamodel was defined as the source model and the **AdministrationExample** metamodel was defined as the target model.

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 that is missing and 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. Go ahead and write **HospitalToAdministration** 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->** and the name of the element. 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. For the next correspondences (lines 14 to 42), 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. The remaining correspondences are given and explained in detail in the next section, where we are going to create some rules.

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 want to differentiate doctors from nurses on the target side as well. 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.

```
1      #import "platform:/resource/HospitalExample/model/HospitalExample.ecore"
2      #import
3      "platform:/resource/AdministrationExample/model/AdministrationExample.ecore"
4      #schema Hospital2Administration
5
6      #source {
7          HospitalExample
8      }
9
10     #target {
11         AdministrationExample
12     }
13
14     #correspondence {
15         HospitalToAdministration{
16             #src->Hospital
17             #trg->Administration
18         }
19
20         StaffToStaff{
21             #src->Staff
22             #trg->Staff
23         }
24
25         PatientToPatient{
26             #src->Patient
27             #trg->Patient
28         }
29     }
```

```

30         DoctorToStaff{
31             #src->Doctor
32             #trg->Staff
33         }
34         NurseToStaff{
35             #src->Nurse
36             #trg->Staff
37         }
38     }
39
40     #attributeConditions {
41         #userDefined doctorsalary(salary:EDouble){
42             #sync: [F],[B]
43             #gen: [F], [B]
44         }
45
46         #userDefined nursesalary(salary:EDouble){
47             #sync: [F],[B]
48             #gen: [F], [B]
49         }
50     ...
51 }

```

The third and last block consists of the **attribute conditions** (lines 40 to 49) where we can define custom attribute conditions. Before stepping into custom attribute conditions, it would make sense to explain attribute conditions in general, which we will be doing in another chapter of this tutorial. So, let us skip attribute conditions and continue our TGG project by looking at the creation of rules.

---

### 3.3 Rules

---

Please switch to the Git repository branch **TGG** for the following section. This branch includes everything you have created so far plus the base frame for the next few chapters. Since the ruleset and certain specifications for the triple grammar graph project are quite extensive. As a consequence, we do not want you to create everything by yourself but rather explain to you the ideas behind certain structures.

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**. Just **click** on the package and then select the right option **New TGG Rule**, give it the name **HospitalToAdministration** and **press finish**.

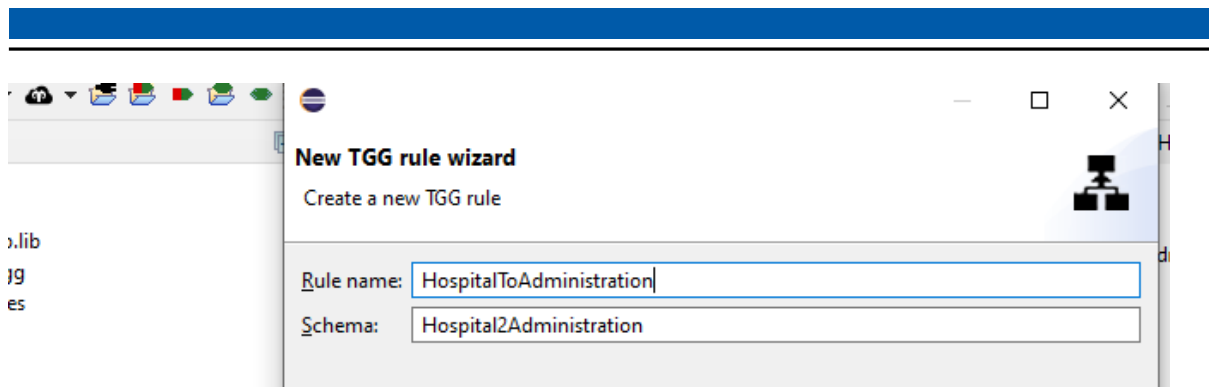


Chart 23: Creation of a new TGG with the rule wizard

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 our correspondence metamodel, we have defined in the Schema.tgg our 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. Name it *htoa* and define its type as the *HospitalToAdministration* correspondence we have created in the Schema.tgg. Now we add the hospital node we created in this rule to the source side and the administration node to the target side.

```

1  #source {
2      ++hospital:Hospital{
3          ++-reception->reception
4      }
5      ++reception:Reception{
6      }
7  }
8
9
10 }
11
12 #target {
13     ++administration:Administration
14 }
15
16 #correspondence {
17     ++htoa:HospitalToAdministration{
18         #src->hospital
19         #trg->administration
20     }

```

Once you are done, this is how our first TGG rule should be visualized in Plant UML and look like the picture in chart 24:

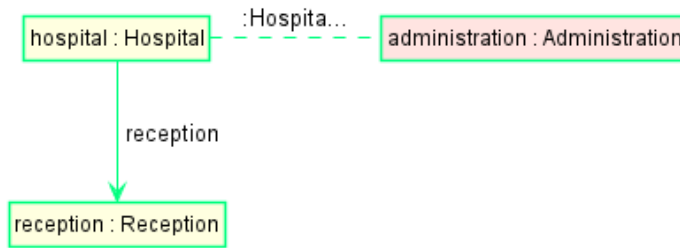


Chart 24: Visualization of the HospitalToAdministration.tgg.

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. As you 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.

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:

```

1  #source {
2      h:Hospital{
3          ++-department->department
4      }
5      ++department:Department{
6
7      }
8  }
9
10 #target {
11
12 }
13
14 #correspondence {
15

```



```

16     }
17
18     #attributeConditions {
19         incrementingDepartmentID(department.dID)
20         setDefaultNumber(department.maxRoomCount, 10)
21     }

```

Maybe you should try coding 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. The correct syntax can be found below if you are unsure.

```

22     #source {
23
24         department:Department{
25             ++-rooms->room
26         }
27         ++room:Room{
28             capacity := 4
29         }
30     }
31
32     #target {
33
34     }
35
36     #correspondence {
37
38     }
39
40     #attributeConditions {
41
42     }

```

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.

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 and its visualization is shown in chart 25. 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.

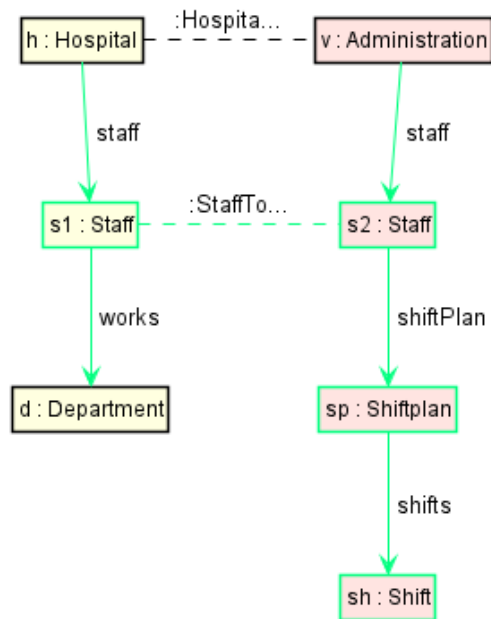


Chart 25: Visualization of the StaffToStaffRule.tgg

You might wonder: “But how do we distinguish doctors from nurses in the administration view?” That is the part where the **DoctorToStaff** and **NurseToStaff correspondences** we created previously come into play, where we assigned doctors and nurses to correspond with the 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 **#extends** which needs to be added after the name of the rule. If you remember from the GT part of our tutorial the keyword **#extends** 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. As well as an **#attributeCondition** for the **doctorsalary**. Please add the attribute condition starting on line 55 but do not worry about it because this function will be covered in the next chapter.

```

43  #rule DocToStaffRule #extends StaffToStaffRule #with Hospital2Administration
44
45  #source{
46      ++s1:Doctor
47  }
48
49  #target{
50      ++s2:Staff{
51
52      }
53  }
```

```

54
55     #attributeConditions{
56         doctorsalary(s2.salary)
57     }

```

After building your project you can click on the name of the **DocToStaff** rule and take a look at the UML visualization in chart 26. The type of **s1** has changed to a doctor.

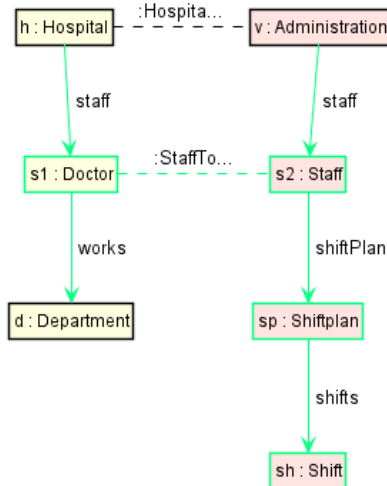


Chart 26: Visualization of the DocToStaffRule.

The **NurseToStaffRule** can be created similarly, maybe you want to **try it yourself before having a look at the syntax below**. All you have to do is to change the type of the staff member as well as adjusting the attribute Condition to the **nursesalary condition**.

```

58     #rule NurseToStaffRule #extends StaffToStaffRule #with
Hospital2Administration
59
60     #source{
61         ++s1:Nurse{
62
63         }
64     }
65
66     #target{
67         ++s2:Staff{
68
69         }
70     }
71
72     #attributeConditions{
73         nursesalary(s2.salary)
74     }

```

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 ShiftPlan rule**. 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. Then we require **patient** and **staff nodes** on the source side and the coverage as we have defined it in the administration metamodel for the target side. So, for the target side, we require a staff node with an edge to a **shiftPlan** node as well as a shift in that **shiftPlan**. For the Patient node, we want to add an edge to the **shiftPlan** to indicate a patient is covered in that **shiftPlan**. 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 this step should be pretty forward.

```

58     #abstract #rule ShiftplanRule #with Hospital2Administration
59
60     #source {
61
62         p1:Patient{
63
64         }
65         s1:Staff{
66
67         }
68
69
70
71     }
72
73     #target {
74
75         s2:Staff{
76             -shiftPlan->sp
77         }
78         sp:Shiftplan{
79             -shifts->sh
80
81         }
82         sh:Shift{
83
84         }
85         p2:Patient{
86             ++-coveredBy->sp
87         }
88
89     }
90
91     #correspondence {
92
93         pToP:PatientToPatient{
94             #src->p1
95             #trg->p2
96         }
97
98         sToS:StaffToStaff{
99             #src->s1
100            #trg->s2
101        }

```

```

102
103   }
104
105   #attributeConditions {
106
107   }

```

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. Open the **DoctorShiftplanRule** and the **NurseShiftplanRule** which are already given and let us compare the UML visualizations which are also depicted in chart 27.

We do not change anything on the target side compared to the original **ShiftplanRule**. As previously mentioned, the way our source side is structured differently because we have different edges we need to connect.

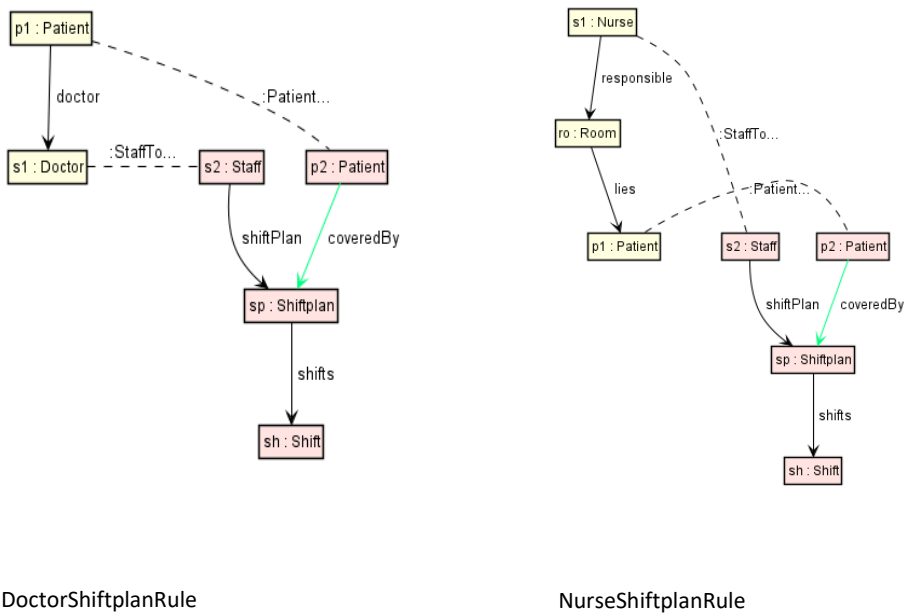


Chart 27: Comparison of the rules for doctors and nurses.

Now that we have covered the staff and their respective shifts, we need to handle the patients. The abstract **PatientToPatientRule** is simple and just 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 chart 28.

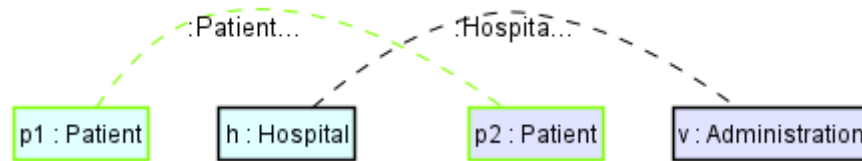


Chart 28: Visualization of the PatientToPatientRule.

As you might notice we have created two correspondences independently which 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:

```

108  #rule PatientInReception #extends PatientToPatient #with
Hospital2Administration
109
110
111  #source {
112
113      ++ p1 : Patient{
114
115      }
116      h:Hospital{
117          -reception->r
118      }
119
120      r:Reception{
121          ++-waits->p1
122      }
123  }
124
125
126  #target {
127
128      ++ p2 : Patient
129      v:Administration{
130          ++-patient->p2
131      }
132  }
133
134
135  #correspondence {

```

```

136
137     htov:HospitalToAdministration{
138         #src->h
139         #trg->v
140     }
141
142 }
143
144 #attributeConditions {
145
146 }

```

The second rule is already given in your project with the name **PatientInRoom**. This rule is rather simple because it just adds the lies edge from the Room context node to the Patient node the visualization of that rule is depicted in chart 29.

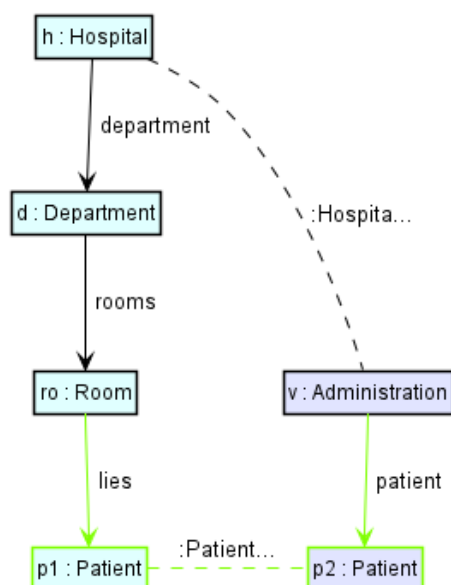


Chart 29: UML visualization of the PatientInRoomRule.

For the last rules, we want to cover in this section, let us hop to the assignment of the patients to a doctor and the nurses to a room. Both rules are already given in your project.

One rule has the name **DoctorToPatientRule**. For the source side of this rule, we need a direct connection between the doctors and the patient because a patient who is covered also has a doctor assigned to him. And to cover a patient, the patient also needs to receive the respective treatment given by a doctor to him. In chart 30 you can see the visualization of the **DoctoToPatientRule**.

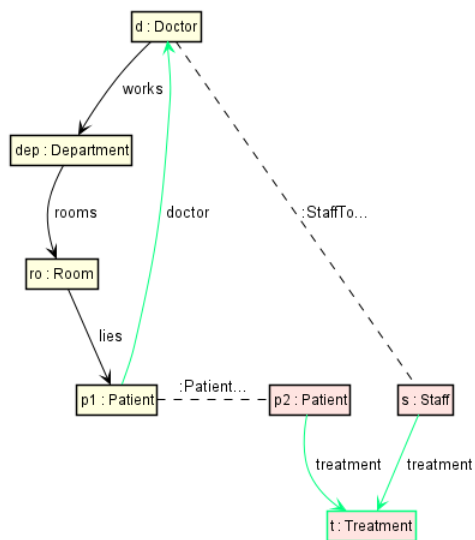


Chart 30: DoctorToPatientRule.

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 chart 31.

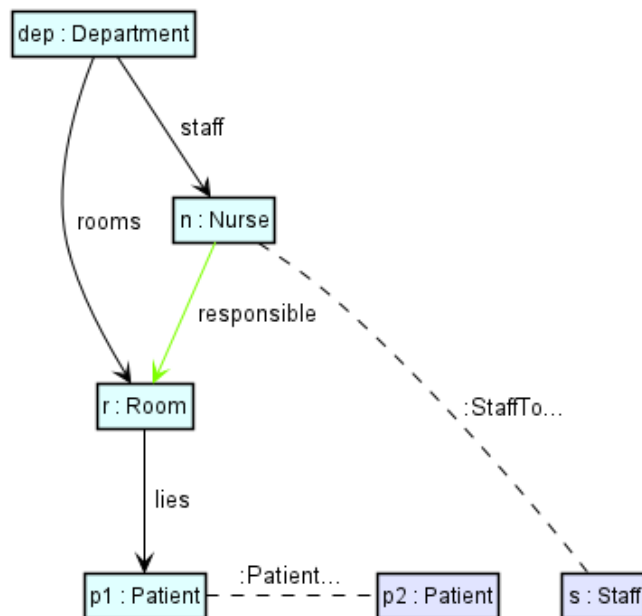


Chart 31: UML visualization of the NurseToRoomRule.

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.



---

### 3.4 Attribute Conditions

---

In this chapter we will be covering attribute conditions. Attribute conditions are used to alter attribute values for the elements of the rule we are using the condition in. For example, you have seen the invocation of the *incrementingDepartmentID* and *setDefaultNumber* in our *departmentRule.tgg*.

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

What do these two attribute conditions do? The *incrementingDepartmentID* is a user-defined attribute condition that increases the number of the *departmentID* by 1 whenever we apply this rule and henceforth creates a new department. User-defined attribute conditions will be explained soon. The *setDefaultNumber* is a predefined attribute condition that sets the attribute value of the *department.maxRoomCount* to the value 10 if the value has not been assigned already. Please, go to the package *org.emoflon.ibex.tgg.csp.lib* and open the *AttrCondDefLibrary.tgg* package which is shown in chart 32.

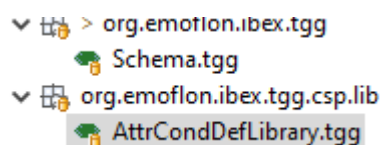


Chart 32: Picture of the *AttrCondDefLibrary*.

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 free. If an attribute value is free its variable value is not defined yet. In this case, it can be assigned a suited value. If an attribute value is bound it has already a value. This is determined by the different cases which are indicated via free “F” and bound “B”. For the *setDefaultNumber* it says that 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.

Depending on the number of variables an attribute condition can cover different cases for the operation you want to use in your rule. The *#sync: [B B], [F B]* line describes the different cases which can occur when you are synchronizing source and target model. In contrast to the generation: *#gen: [B B], [F B], [F F]* synchronization does not have the case free, free because if both

---

variables are not set, then there is nothing to synchronize. The differences between generating and synchronizing a consistent triple will be explained in detail in the next chapter.

You can find a comprehensive list of the predefined attribute conditions and a short explanation of each condition [in the appendix](#).

To create a user-defined attribute condition you have to **define it** in the **schema.tgg** first. Let us start with by inspecting the `incrementingDepartmentID` which is already given:

```
#userDefined incrementingDepartmentID(id:EInt)
{
    #sync:[B F]
    #gen:[F]
}
```

A user-defined attribute condition is defined by the keyword ***#userDefined*** followed by the name of the attribute condition and the parameters to alter. 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. Now head to the ***.constraints.custom.hospital2administration*** package and open up the ***UserDefined\_incrementingDepartmentID***.

```
1      public class UserDefined_incrementingDepartmentID extends
RuntimeTGGAttributeConstraint
2      {      private static int idIncrement = 1;
3              @Override
4              public void solve() {
5                  if (variables.size() != 1)
6                      throw new RuntimeException("The CSP -
INCREMENTINGDEPARTMENTID- needs exactly 1 variables");
7
8                  RuntimeTGGAttributeConstraintVariable v0 = variables.get(0);
9                  String bindingStates = getBindingStates(v0);
10
11                  switch(bindingStates) {
12                      case "F":
13                          v0.bindToValue(idIncrement++);
14                          setSatisfied(true);
15                          break;
16                      case "B":
17                          setSatisfied(true);
18                          return;
19                      default: throw new UnsupportedOperationException("This case in
the constraint has not been implemented yet: " + bindingStates);
20                  }
21
22
23      }
```

On line 7 we define the initial value of our incrementing ID which is incremented in the switch statement defined in lines 22 to 30. If the **departmentID** is free we switch to the case F and bind the ID to the value of **idIncrement** variable which is increased by 1 every time we invoke this user-defined attribute condition. In the next line, we set the Boolean variable **setSatisfied** to **true** to fulfill the requirement for our runtime variable. For the second case “B” we just set the setSatisfied value to true, since a department already has an ID and does not need one.

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**.

```
#userDefined incrementingRoomID(id:EInt)
{
    #sync:[B F]
    #gen:[F]
}
```

Please build the project by **pressing** either the **black** or the **green hammer symbol** 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. Now we have to adjust the user-defined attribute constraint to do what it is supposed to do. In this case, we need a counter variable that is incremented with every invocation of the attribute and the assignment of the cases **free** and **bound** just like in the example for **incrementingDepartmentID**.

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.

```
#userDefined nametoname(separator:EString, leftWord:EString, rightWord:EString,
result:EString) {
    #sync: [B B B B], [B B B F], [B B F B], [B F F B], [B F B B]
    #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]
}
```

However, 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. If you open the java file of the attribute constraint you can see two arrays with sample names which we are using to generate a random first and last name whenever we invoke the attribute constraint, and the name attributes are not set yet:

```
25     case "BFFF":
26
27         String firstName = firstNames[random.nextInt(firstNames.length)];
28         v1.bindToValue(firstName);
29         String lastName = lastNames[random.nextInt(lastNames.length)];
30         v2.bindToValue(lastName);
31         v3.bindToValue(firstName + v0.getValue() + lastName);
32         setSatisfied(true);
33         break;
```

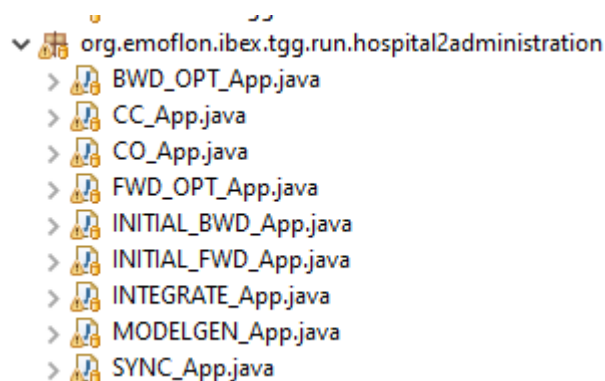
For the case above we want a bound separator character which is a blank space in our case. Then we select a random first name from our first name array and bind its value to the runtime variable **v1** which represents the **leftWord**. The same is done for the last name. **v3** is the **result** of combining the values of the previous runtime variables. As you can see the user-defined constraints are a powerful way to create constraints that are not covered by our predetermined library.

---

### 3.5 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 chart 33:

Chart 33: Application overview for the TGG project.

The first operation we are interested in is the **MODELGEN\_APP.java**. Let us start exploring this feature by **running** the **MODELGEN\_APP** in our java project. **Refresh** the project folder.

Once you open the **instances** folder you should be able to see four files chart 30 shows.

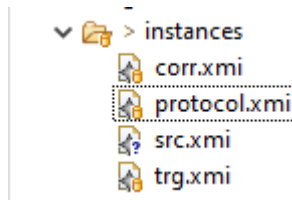


Chart 30: The different model instances.

The MODELGEN\_APP generates a consistent triple according to our rule sets from scrap. It generates a hospital model, an administration target model, and the correspondence model for the connections between our source and target model.

Open the src.xmi by clicking on it and discover the model instance. You should see Patients waiting in the reception or being assigned to their rooms and the staff members with their departments and for example nurses which are covering a room with patients. Look at the trg.xmi as well and you will notice we have the same patients and staff members in this model instance as well but this time we focused on the shifts and treatments we assigned to their patients. A model generator might also be helpful for testing purposes, especially scalability testing that requires large models.

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. The **INITIAL\_FWD** 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. The **INITIAL\_BWD** works in the same way but requires the target side. Hence, the names fwd for forward synchronization and bwd for backward.

---

Another option is the **CC\_App** which stands for Consistency Checking which compares the given source and target model instances and creates a respective correspondence model. 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. With the **CC\_App** a pair of source and target models can be checked for consistency by attempting to extend the pair to a consistent triple. If the operation succeeds it will additionally provide a correspondence model, which can be partial if the source and target models are inconsistent. A consistency check is particularly useful for verification purposes.

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 information loss 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 respective side. A possible solution would be the manual addition of these properties to the target metamodel, 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 by **updating an existing output model incrementally**. We only add what is necessary and is missing or changed. In this manner, unrelated parts of the model can be retained.

The second reason is efficiency. **SYNC** works **incrementally**, the time required for an update does not depend on the size of the current model triple, but rather on the size of the update itself. Please note 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. 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.

Now let us take a look at the apps because we have some options to configure them.

The first line of interest is the definition of the registration helper:

```
public static IRegistrationHelper registrationHelper = new  
HiPERRegistrationHelper();
```

In the package **hospital2administration.config** shown in chart 34 you see that you have the choice between three different registration helpers:

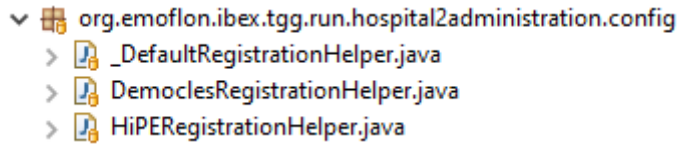


Chart 34: RegistrationHelper overview.

By default, a newly created project has the **\_DefaultRegistration** helper selected, but we recommend using the **HiPERRegistrationHelper()** for every application **except** the **BWD\_OPT.App** and the **FWD\_OPT.App** since both requires the DemoclesRegistrationHelper. And you might already guess the reason: The main difference between those two is the utilization of the different pattern matchers we have explained at the very beginning of our tutorial.

Then you can see the two methods for saving and loading models, as an example, you can see in the screenshot of the **MODELGEN\_App** in chart 35 below:

```
public MODELGEN_App() throws IOException {
    super(registrationHelper.createIbexOptions().resourceHandler(new TGGResourceHandler() {
        @Override
        public void saveModels() throws IOException {
            // Use the commented code below to implement saveModels individually.
            // source.save(null);
            // target.save(null);
            // corr.save(null);
            // protocol.save(null);

            super.saveModels();
        }

        @Override
        public void loadModels() throws IOException {
            // Use the commented code below to implement loadModels individually.
            // loadResource loads from a file while createResource creates a new resource without content
            // source = loadResource(options.project.path() + "/instances/src.xmi");
            // target = createResource(options.project.path() + "/instances/trg.xmi");
            // corr = createResource(options.project.path() + "/instances/corr.xmi");
            // protocol = createResource(options.project.path() + "/instances/protocol.xmi");

            super.loadModels();
        }
    })
}
```

Chart 35: Example for loadModels() and saveModels() methods in the MODELGEN\_APP

Right now, we are using the keyword **super** access these operations from the constructor. Depending on the operation you want to perform you can use the code which is commented out right now. If you swapped the commented code with the **super.saveModels()** line, you would simply save all four files you can see in the instances folder. For the **loadModels method**, we would load a source model from the given path and create the corresponding resources. Adjusting the line, we have commented out right now would also allow changing the path we are saving our models to or loading from. In the save method we could also change the name of our file.

Let us walk through this exemplarily by modifying the **SYNC\_APP** to load our **hospital.xmi** we created in the first part of this tutorial. For the first step we want to modify the loadModels method to load

---

the *hospital.xmi*. You need to adjust the path to the source variable to access the *hospital.xmi* which should be saved in your workspace in the project folder of the HospitalExample. The “../” command allows us to navigate through the folders of our project. This is a so-called relative file path. In this case, we switch to the parent folder of the folder where our project is stored and then we navigate to the *hospital.xmi* which is stored in the project folder of our graph transformation project. Using the relative path to our project is more convenient and less prone to failure.

```
source = loadResource(options.project.path() "../Hospital2Administration/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:

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

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

```
target.save(null);  
corr.save(null);  
protocol.save(null);
```

After you have **saved** your file and **ran** the **SYNC\_APP**, 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 now 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 ms. By **typing stop.** and 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](#).

---

### 3.6 Debugging in TGGs

---

A note on the debugging app you can find in the .debug package shown in chart 36.

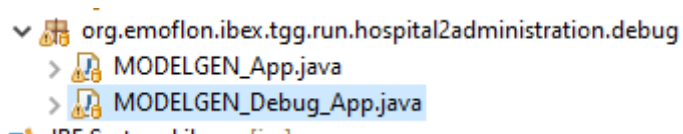


Chart 36: Location of the debugging application

This might be helpful when your set of triples is created without any errors, but the rules or correspondences are not working in the way you want them to work. Once you **run** the **MODELGEN\_Debug\_App.java** a new window like the one in chart 37 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.

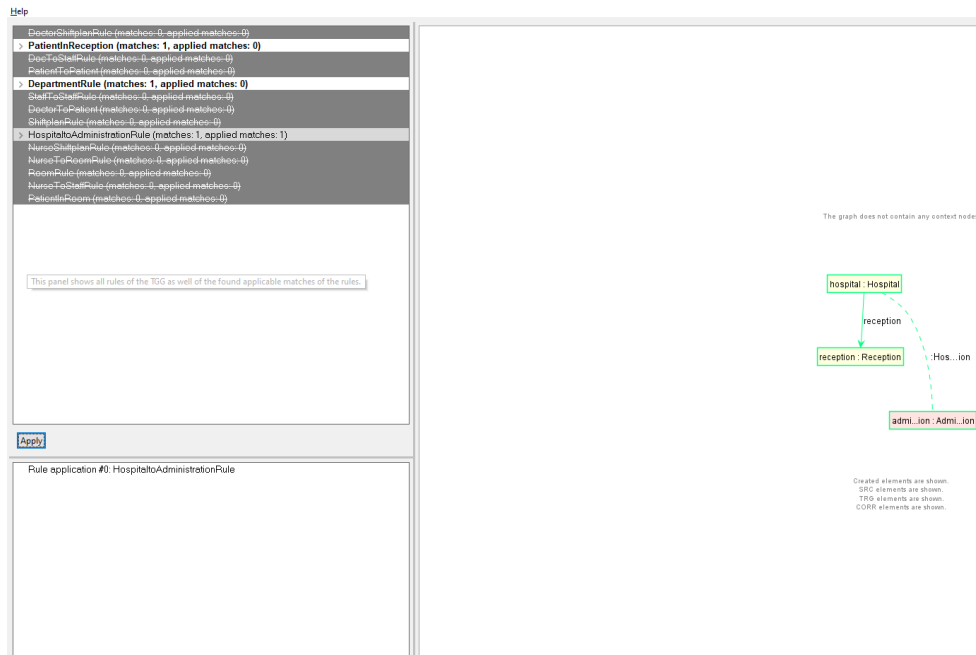


Chart 37: Visual example of the debugging process.

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.

### 3.7 Adding additional Information

Now let us head back one last time to graph transformation to finalize our tutorial by switching to the branch TGGTransform in the Git repository.

Even though TGGs are a powerful tool to ensure consistency between our model instances, we cannot map every information or specification we want to model for our hospital scenario.

In this case, we need an additional ruleset for the shifts and the shift plans because this information is unique to the perspective of the administration. So let us look at the administration example project which is already in your workspace.

Look at the **AdministrationValidator.java**:

```
1 public class AdministrationValidator extends
2     AdministrationTransformRuleDemoclesApp {
3
4     public AdministrationValidator() {
5
6
7         String filePathUrl = workspacePath +
8         "Hospital2Administration\\instances\\trg.xml";
9
10        loadModel(URI.createFileURI(filePathUrl));
```

```

11     }
12 }

```

In contrast to the HospitalValidator, where we created a new model, we do **load** the administration model on line 9. If you recall the synchronization operation the **trg.xmi** is the administration model instance, we have created in the triple graph grammar project.

The problem we want to solve for the last part of this tutorial is the coverage of the patients with the shifts and the shift plans. For now, we just added the treatments to the patients throughout the synchronization process since the treatments correspond with the care levels of the hospital model instance. If we want to simulate a running hospital to a certain degree, we need to extend this coverage of the patient, so a patient has not only a treatment assigned to him but is covered throughout the day. In the administration metamodel, we assumed that a day is split into three shifts and we want to cover a patient for the whole day. Consequently, we need to add additional information for this part of our model since the information about the shifts is only present in the target side of our model. Right now, we are lacking the information. So let us take care of the **staff rule**. For a staff member, we want to create a **shiftplan** node that includes the creation of two **shift** nodes. Since every shift is connected via a treatment node for a certain patient, we want to add a constraint to limit the number of treatments a staff member can handle to less than three. The concrete implementation and the visualization (chart 38) of the staff rule can be found below:

```

1  rule staff(time1:Daytime, time2:Daytime)
2  refines person {
3      ++person: Staff {
4          ++ -shiftPlan -> shiftPlan
5
6          ++ -treatment -> treatment
7
8      }
9
10     ++shiftPlan: Shiftplan {
11         ++ -shifts -> shift
12         ++ -shifts -> shift2
13     }
14
15     ++ shift: Shift {
16         .time:=param::time1
17     }
18
19     ++ shift2: Shift {
20         .time:=param::time2
21     }
22     treatment:Treatment{
23
24     }
25     #count(findTreatment)<3
26

```

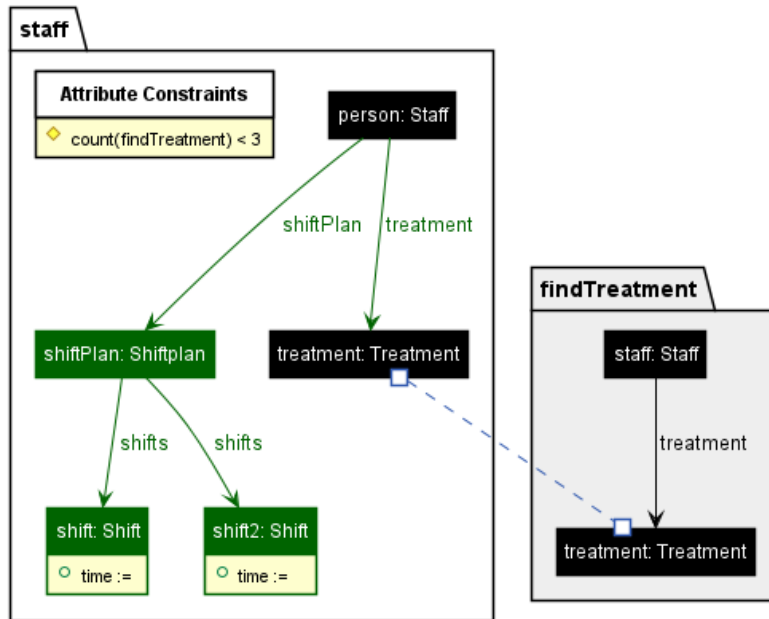


Chart 38: Visualization of the staff rule for the administration.

The other rather complicated rule for the administration example is the actual coverage of the patients which we name **patientCovered**. We assume that patient requires at least 2 staff members who take care of him. Consequently, we need to make a few assumptions for the staff members who are covering a patient. First, we need to ensure the two staff members we assign to a patient are not the same. Secondly, a patient has to be covered for the whole day. Hence, we need an early shift, a late shift, and a night shift for each patient. This assumption leads us to another requirement for our **patientCovered** rule: Since a staff member is only allowed to have two shifts per day, we need at least two different staff members to cover one patient. We will be using two different shiftplans to connect a patient with a staff member who will be responsible for the patient during his shift. At last, we need a condition which forbids the coverage of a patient who is already covered, in order to avoid overcrowding. For reasons of simplicity we assume that the first shiftplan covering a patient includes an early shift and a night shift. This leaves the late shift for the second shiftplan, namely **shiftPlan2**.

The whole **patientCovered** rule is depicted below:

```

28 rule patientCovered() {
29     patient: Patient {
30         ++ -coveredBy -> shiftPlan
31         ++ -coveredBy -> shiftPlan2
32         -treatment->treatment

```

```

33     }
34
35     shiftPlan: Shiftplan {
36         -shifts -> earlyshift
37
38         -shifts -> nightshift
39     }
40     shiftPlan2: Shiftplan {
41         -shifts -> lateshift
42
43     }
44
45     earlyshift: Shift {
46
47     }
48
49     lateshift: Shift {
50
51     }
52
53     nightshift: Shift {
54
55     }
56
57
58     s: Staff {
59         -treatment -> treatment
60         -shiftPlan->shiftPlan
61     }
62     s1:Staff{
63         -treatment -> treatment
64         -shiftPlan->shiftPlan2
65     }
66     treatment: Treatment {
67
68     }
69 } when patientAlreadyCovered
70 condition patientAlreadyCovered = forbid findCoveredPatient

```

As you can see in chart 28 the rule has become quite extensive even though we have only created two new edges from the patient towards a respective shift plan. The visualization in chart 39 contains the != symbol to indicate which nodes are allowed to be the same object. Due to the limitation of two shifts per staff member is it necessary that the staff member s is not the same as staff member s1. The same takes effect on *shiftPlan* and *shiftPlan2* as well as the three different shifts.

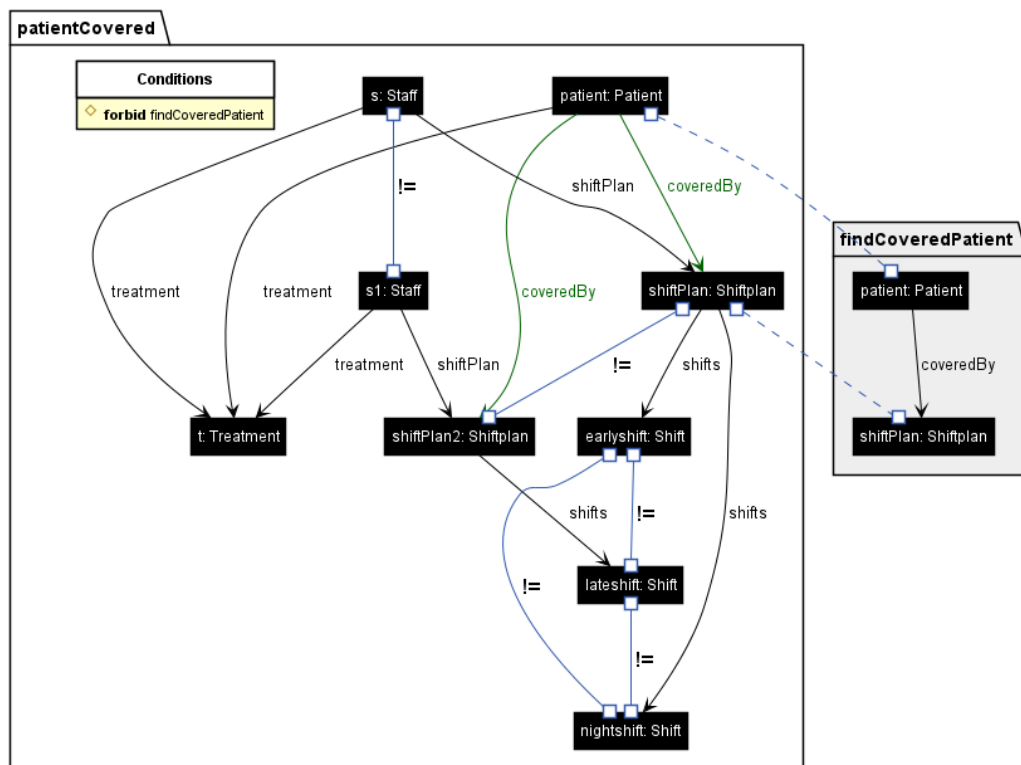


Chart 39: PlantUML visualization of the patientCovered rule.

After making sure every patient is covered it is time to head over to the **AdministrationRules.java** file and test the two rules we have created previously. The AdministrationRules.java is structured in the same way as the HospitalRules.java we have explained in detail [here](#).

First, we are creating the remaining parts of the administration by applying the staff and the patientCovered rule multiple times in the **createAdministration()** method, then we are trying to validate the functionalities of our rules with a console output. If you run the AdministrationRules.java, the following output of chart 40 should appear in your console:

```
11 Patients are in the hospital right now
26 staff members are in the hospital right now
Every Patient is covered
We have an early shift
We have a late shift
We have a night shift
```

Chart 40: Console output for the AdministrationRules.java.

Congratulations on completing the last step of our tutorial! We hope that you have learned plenty about the functions of eMoflon::IBeX and you are now well equipped to create your project. Do not hesitate to use this tutorial and the appendices as a reference in case further questions arise.

---

### 3.8 Appendix for graph transformation

---

#### 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, which 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, you will find two files: `ibex-patterns.xmi` and `democles-patterns.xmi`. These two files 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` so open this file and note that you can visualize the contained PIN in the PlantUML view. There is an overview visualization for the pattern set and another visualization for individual context patterns.

#### Attribute conditions overview:

Attribute Condition which ensures both given string variables are equal:

```
eq_string(a: EString, b: EString) {  
  
    #sync: [B B], [B F], [F B]  
    #gen:  [B B], [B F], [F B], [F F]  
}
```

Attribute Condition which ensures both given integer variables are equal:

```
eq_int(a: EInt, b: EInt) {  
    #sync: [B B], [B F], [F B]  
    #gen:  [B B], [B F], [F B], [F F]  
}
```

}  
Attribute Condition which ensures both given float variables are equal:

```
eq_float(a: EFloat, b: EFloat) {
    #sync: [B B], [B F], [F B]
    #gen:  [B B], [B F], [F B], [F F]
}
```

Attribute Condition which ensures both given double variables are equal:

```
eq_double(a: EDouble, b: EDouble) {
    #sync: [B B], [B F], [F B]
    #gen:  [B B], [B F], [F B], [F F]
}
```

Attribute Condition which ensures both given long variables are equal:

```
eq_long(a: ELong, b: ELong) {
    #sync: [B B], [B F], [F B]
    #gen:  [B B], [B F], [F B], [F F]
}
```

Attribute Condition which ensures both given char variables are equal:

```
eq_char(a: EChar, b: EChar) {
    #sync: [B B], [B F], [F B]
    #gen:  [B B], [B F], [F B], [F F]
}
```

Attribute Condition which ensures both given boolean variables are equal:

```
eq_boolean(a: EBoolean, b: EBoolean) {
    #sync: [B B], [B F], [F B]
    #gen:  [B B], [B F], [F B], [F F]
}
```

Attribute Condition which adds a prefix to a given word variable and handing over the prefix plus the word as the result:

```
addPrefix(prefix:EString, word:EString, result:EString) {
    #sync: [B B B], [B B F], [B F B], [F B B]
    #gen:  [B B B], [B B F], [B F B], [F B B], [B F F], [F B F]
}
```

Attribute condition which adds a suffix to a given word variable and handing over the suffix plus the word as the result:

```
addSuffix(suffix:EString, word:EString, result:EString) {
    #sync: [B B B], [B B F], [B F B], [F B B]
    #gen:  [B B B], [B B F], [B F B], [F B B], [B F F], [F F F], [F B F]
}
```

Concatenation attribute condition which combines a left word the separator and the right word in this order to a result:

```
concat(separator:EString, leftWord:EString, rightWord:EString,
result:EString) {
    #sync: [B B B B], [B B B F], [B B F B], [B F F B], [B F B B]
    #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]
}
```



Attribute Condition which sets a variable string to the defaultString if it is free (FB). If it already has a value (BB) then nothing is done, and the condition is still satisfied. The case (FF) does not make sense for #sync as this should be a fixed default string.

```
setDefaultString(variableString:EString, defaultString:EString) {
    #sync: [B B], [F B]
    #gen: [B B], [F B], [F F]
}
```

Attribute Condition which sets a variableNumber to the defaultNumber if it is free (FB). If it already has a value (BB) then nothing is done and the condition is still satisfied. The case (FF) does not make sense for #sync as this should be a fixed default string.

```
setDefaultNumber(variableNumber:EDouble, defaultNumber:EDouble) {
    #sync: [B B], [F B]
    #gen: [B B], [F B], [F F]
}
```

The attribute condition converts a stringValue into a double Value:

```
stringToDouble(stringValue:EString, doubleValue:EDouble) {
    #sync: [B B], [B F], [F B]
    #gen: [B B], [B F], [F B], [F F]
}
```

The attribute condition converts a stringValue into an int Value:

```
stringToInt(stringValue:EString, intValue:EInt) {
    #sync: [B B], [B F], [F B]
    #gen: [B B], [B F], [F B], [F F]
}
```

The attribute condition multiplies both operands for the result:

```
multiply(operand1:EDouble, operand2:EDouble, result:EDouble) {
    #sync: [B B B], [B B F], [B F B], [F B B]
    #gen: [B B B], [B B F], [B F B], [F B B]
}
```

An attribute divides the numerator by the denominator and the result contains the solution of the operation:

```
divide(numerator:EDouble, denominator:EDouble, result:EDouble) {
    #sync: [B B B], [B B F], [B F B], [F B B]
    #gen: [B B B], [B B F], [B F B], [F B B]
}
```

A simple attribute condition adding both summands for the result:

```
add(summand1:EDouble, summand2:EDouble, result:EDouble) {
    #sync: [B B B], [B B F], [B F B], [F B B]
    #gen: [B B B], [B B F], [B F B], [F B B], [F F B], [F B F], [B F F]
}
```

A simple attribute condition subtracting the subtrahend from the minuend for the result:

```
sub(minuend:EDouble, subtrahend:EDouble, result:EDouble) {
    #sync: [B B B], [B B F], [B F B], [F B B]
}
```

```

        #gen: [B B B], [B B F], [B F B], [F B B], [F F B], [B F F], [F B F],
        [F F F]
    }

```

The attribute condition selects the maximum value from the two given double variables:

```

max(a:EDouble, b:EDouble, max:EDouble) {
    #sync: [B B B], [B B F], [B F B], [F B B]
    #gen: [B B B], [B B F], [B F B], [F B B]
}

```

Attribute condition 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.

```

setRandomString(a:EString) {
    #sync: [B]
    #gen: [F], [B]
}

```

#### Stop Criterion's overview:

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

---

## 4 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.

**I have switched to another branch of the repository for the tutorial, and I get error markers for my project?**

- First, you should build your project. Try to build it multiple times if the errors persist.
- Secondly, you should look at the dependencies in the MANIFEST.MF, since the required bundles are not added automatically.
- If you still have errors in your project file you can try to delete the src-gen folder and rebuild the project.