Scenario: Public Transport System (PTS)

Project Description “Modelling Systems and Languages (MoSyL)”

We are PTS a globally operating company in the public transport sector. In our core business, we manage and operate passenger trains in various countries. We have recently acquired the competing company Global Transport (GT) along with the rights to their network and are now faced with heterogenous software systems to plan our passenger trains. In consequence, we have decided to rebuild our core software system to adequately manage various networks and their train traffic. We are now looking for software developers to create a software system that can adequately represent our practices for the respective users. In particular, we have four main business areas that need to be represented adequately:

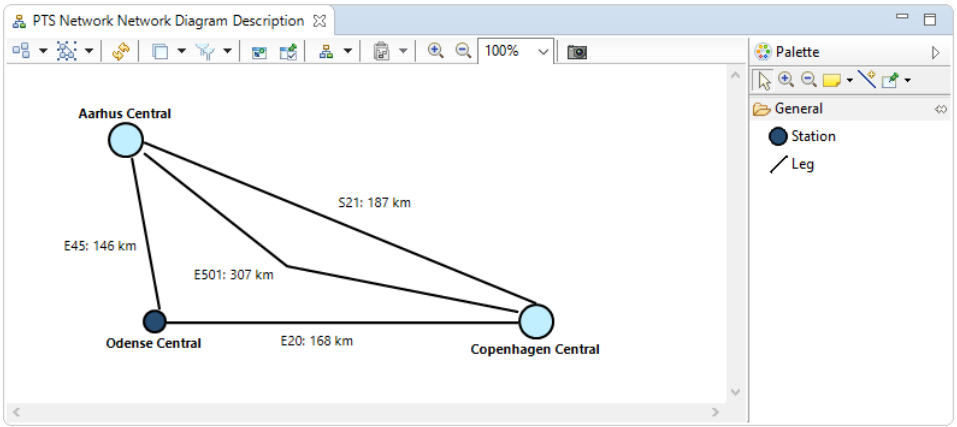
1. Network Planning (Network)

We service networks in different countries (e.g., Denmark) and regions (e.g., East Germany). A network contains all the stations that we service in the area along with their connections.

A station’s main characteristic is its name, e.g., “Copenhagen Central”. In the PTS jargon, the connection between two stations is called a “leg”. A leg connects two stations, which can be navigated back and forth, and defines the distance to be traveled between them in kilometers. A leg may carry a name but, for cases where there is only one leg between two individual stations, we usually omit it. However, if two individual stations are connected by more than just one leg, each of the legs must carry a name so that we can uniquely identify the route a train takes (see Schedule). As we operate multiple networks, we need each network to be uniquely identifiable, e.g., via a name.

Together with our network operators, we have already defined a metamodel capturing this information. However, the respective models can still violate well-formedness. We also need means to define our own network as well as to import data from the acquired company GT, who used a different format (see below).

Network operators that define our networks in the software systems are trained in various fields. However, they are seldom computer specialists so that we would like the definition of networks to closely align with the real-world equivalence of connected train stations on a map, i.e., in a graphical notation. We have prepared a mockup of a possible representation in the software system to be built:



Global Transport (GT) defined their network in what they called a Station Plan. While their format is different from ours, the respective files contain sufficient information to be imported into our network. We now need a mechanism to transfer their (model-based) data into our models.

Fleet Planning (Depot)

For each of our networks, we operate different fleets of trains, where trains are stored in a depot[1]. Our depot managers combine different coaches (see below) to form specific trains that each have a name unique throughout the entire depot, e.g., "ICE121".

We operate two different types of trains: *Regional* and *Intercity* trains, which, among others, affects the trains' potential layout, i.e., which coaches they may be comprised of. A layout is individual to each train and needs to be defined by our depot managers, e.g., number and order of coaches. Trains can principally consist of three different types of coaches: *Locomotive*, *Passenger Coach* and *Dining Coach*.

A Locomotive pulls the train and, thus, needs to be the first or last coach of each train. In certain cases, it is also possible to have a locomotive on both ends of a train but never in between further coaches.

A passenger coach is meant to transport passengers. It can either be of first class or second class. However, within each train, all passenger coaches of one class (first class or second class) must be in sequence, i.e., no coaches of another class must be in between them and there must be no more than one sequence for each class per train. Furthermore, each intercity train is required to have at least one first-class passenger coach.

Finally, a dining coach serves food and beverages to interested customers. Each train may have at most one dining coach but intercity trains are required to have one. If a train is comprised of passenger coaches of both first and second class, the dining coach has to be located in between the classes.

In addition, each of the inner coaches (i.e., not locomotives) carries a number unique within the train that identifies it for passengers.

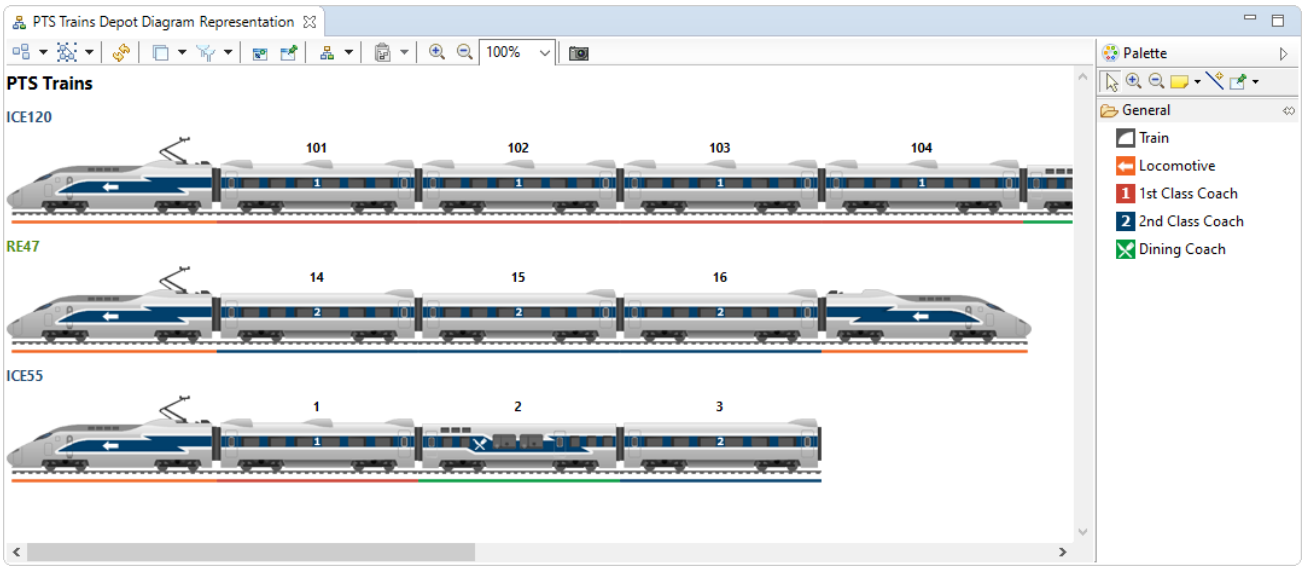
We want to define and, ideally, modify trains within a depot with a visual editor. The editor should allow to create various trains by placing coaches of the respective types. It should also allow to modify the trains' names and the (inner) coaches' numbers. It must also be possible to delete coaches from a train. The editor must warn our depot managers if they disobey rules for defining trains, e.g., placing coaches in the wrong order. Ideally, the editor will also allow to rearrange a train's coaches via drag and drop.

To align closely with depot managers' experience, trains and their coaches should be represented by visually expressive symbols, i.e., graphics of different train coaches. To further aid visual navigation, each type of train coach is associated with a specific color: locomotives are represented by orange, first-class passenger coaches by red, second-class passenger coaches by blue and dining coaches by green. We supply graphical material for our favored representations.

We have created a mockup of a potential editor for depots:

---

[1] We do not represent a fleet explicitly but perceive it as the sum of trains from relevant depots, i.e., for schedule planning (see below), it must be possible to use trains from multiple depots.



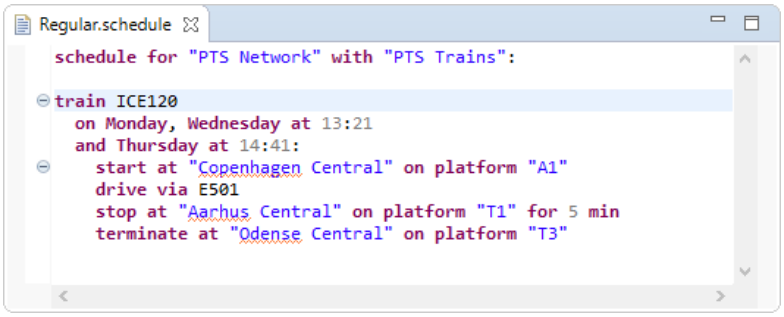
Schedule Planning (Schedule)

A schedule associates trains from one or many depots with a network by assigning traveling routes and departure times/frequencies. As this is the part of our software system where we currently face many heterogeneous approaches, we have less of a clear picture regarding an overall solution. Hence, we define requirements on a satisfactory solution (rather than instructions):

- An individual schedule uses a single network but can use trains from potentially many depots.
- Trains must be able to navigate through the stations of the network along routes where the route is specific to a particular train schedule, e.g., there may be a train from Copenhagen to Aarhus going directly but there may be another going via Odense.
- If there is only one leg connecting two stations, we do not want to be required to specify it explicitly (it may not even have a name to reference). However, when there is more than one leg connecting the same two stations, we must be able to specify which of the two legs should be taken.
- For each stop along the route, the platform number the train will arrive at/depart from has to be specifiable. Furthermore, a train stops for a number of minutes that must be specifiable for each station along the route as well.
- At certain stations along the route, the train must turn due to the station's track layout. We want to manually mark the stops on a route that require a turn. However, when a route contains a turn, it must be ensured that the train driving the route has a locomotive as leading *and* trailing coach.
- Train schedules are often repetitive so that a given train may drive the same route on different days at the same time (e.g., Monday and Wednesday at 13:21) or on the same day at different time (e.g., Monday at 13:21 and 14:41). Further combinations of days and times are also possible for repeat schedules. We need a compact representation of these repeat schedules to not have to redefine too much of the respective route.

Our train schedules are subject to frequent chance and we have export schedulers working on them. To allow swift editing of schedules, we envision a textual scheduling language that adequately represents above requirements.

Below is an example of (an excerpt of) the syntax we can envision for the schedule language but we are open for your suggestions:

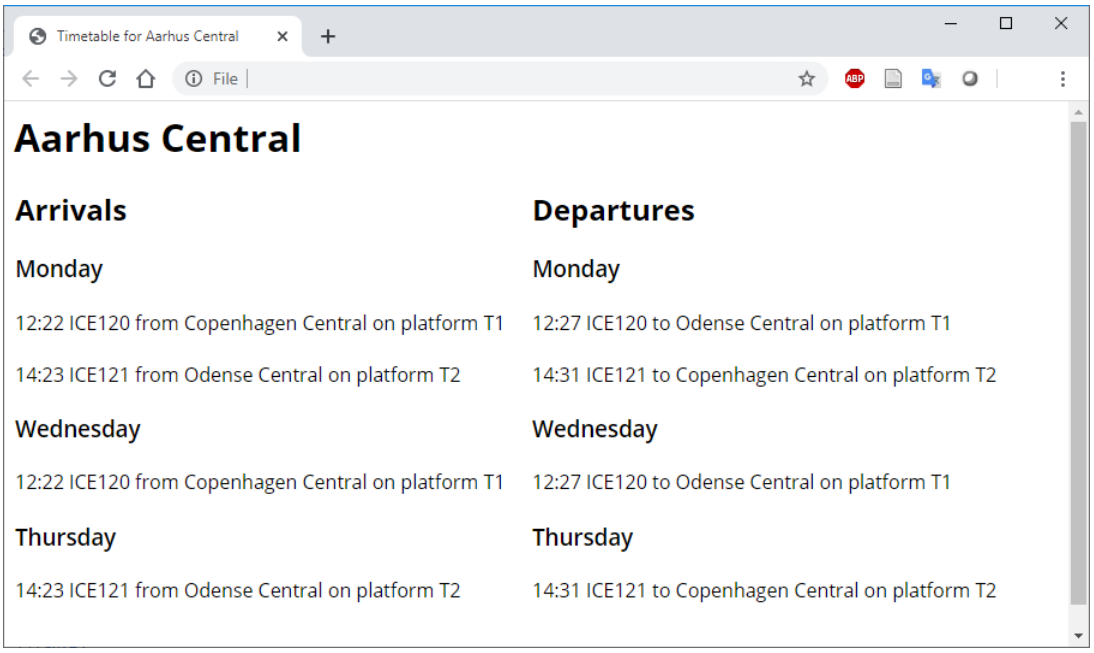


Timetable Announcement (Timetable)

Communication with our customers (train users) is of utmost importance to us. For this purpose, we supply timetables for each station we operate, which shows the arrival and departure of each train along with the respective time and track. Up to now, we have created these timetables manually in a tedious and error-prone operation. With the new software system, we wish the timetables for each station to be generated automatically from the information supplied in a particular schedule. Ultimately, we wish the information of a timetable to be represented in an HTML file, which we can post online, but we would also like the timetable information to be represented within the software system so that we can later add further generators, e.g., properly designed posters to be printed.

A timetable contains arrivals and departures for a station. For each individual arrival/departure, a set of data has to be collected containing: the respective train, the day of the week, the time on that day and the platform. For arrivals, the timetable also has to store from which station a train arrived (if not starting at the timetable's station) and, for departures, to which station the train will drive next (if any).

Below is a representation of the HTML format of our timetables, which should also be used for generation:

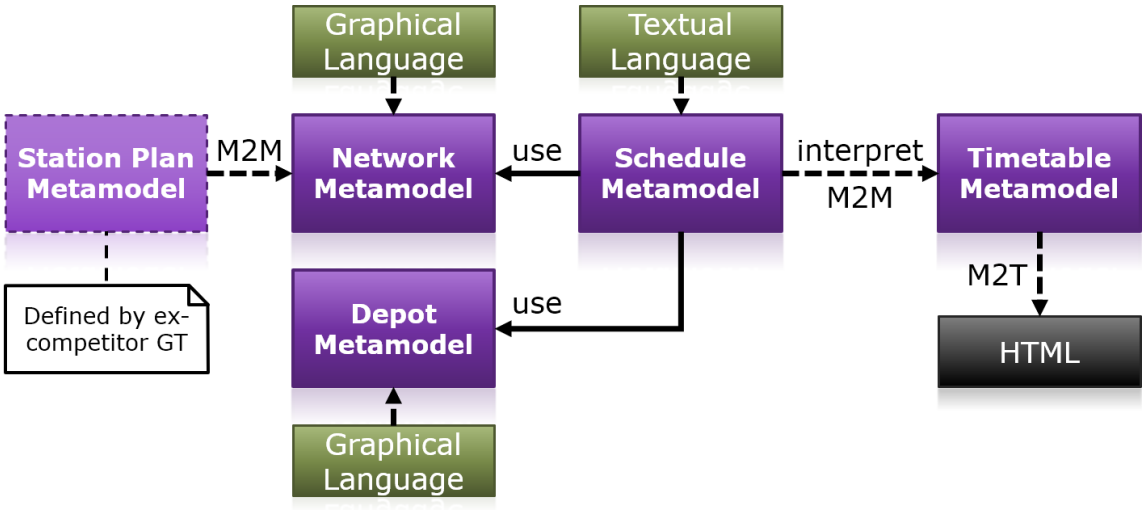


We envision the information for a timetable to be collected by evaluating a particular schedule (and, indirectly, its referenced network and depots): According to the schedule, a train drives along a certain route passing multiple stations, which can be used to calculate arrivals and departures along with their days of the week and times on that day. This information can be used to advance a train from the initial station to the next in the route and so on. The travel time between two stations is calculated via the distance on the respective leg of the network and a trains average speed. Depending on the type and layout of a train, different average speeds are assumed: A regional train drives at an average of 80 kph and an intercity train drives at an average of 150 kph. However, due to their length, intercity trains that have eight or more coaches can only drive at an average of 130 kph. When traveling from a station A to a station B, the arrival time at a station B is then calculated as the departure time at station A plus the travel time from station A to station B. In turn, the departure time at station B is calculated as the arrival time at station B plus the standing duration during that stop of the schedule.

Hints and Tips

Help in Completing the Project in "Modelling Systems and Languages (MoSyL)"

The project will result in 4 core metamodels for *Network*, *Depot*, *Schedule* and *Timetable*. The *Station Plan* metamodel was defined by the (ex-)competitor GT and serves merely as input to a M2M transformation. The following overview relates the metamodels and procedures to be performed upon them:



You will be supplied the overall project structure to get you started (but you will have to define additional projects). In the respective archive, you will find the following structure:

- Development
  - `org.mdse.`**`gt.stationplan`**
    Project containing the GT Station Plan metamodel.
  - `org.mdse.pts.`**`depot`**
    Project to define the Depot metamodel. Also add constraints where appropriate.
  - `org.mdse.pts.`**`depot.dsl`**
    Project to define the graphical Depot DSL. During development, it might be worth moving this project to the runtime instance to develop language dynamically. The folders `icons` and `symbols` contain resources you should use in your editor.
  - `org.mdse.pts.`**`network`**
    Project containing the (already defined) Network metamodel. Add constraints where appropriate.
  - `org.mdse.pts.`**`network.dsl`**
    Project to define the graphical Network DSL. During development, it might be worth moving this project to the runtime instance to develop language dynamically. The folder `icons` contains resources you should use in your editor.

    - `org.mdse.pts.`**`schedule`**
      Project to define the Schedule metamodel. Add constraints where appropriate.
    - `org.mdse.pts.`**`schedule.dsl`**
      Project to define the textual Network DSL.
    - `org.mdse.pts.`**`timetable`**
      Project to define the Timetable metamodel.
- Runtime
    - **`GTFiles`**
      Defines a GT Station Plan model that serves as input to the M2M transformation from GT Station Plan to PTS Network.
    - **`PTSFiles`**
      A place to go wild with modeling PTS artifacts. Can also serve as target of the M2M transformation from GT Station Plan to PTS Network.
- Additionally, in the project `org.mdse.pts.common`, you will find utility classes that help you deal with Eclipse and Ecore (they work for me but no guarantees that they will work in all cases; either way, a good place to get you started):
    - Class `org.mdse.pts.common.util.`**`EclipseUtil`**
      Various methods that help you deal with finding and modifying files in Eclipse.
    - Class `org.mdse.pts.common.util.`**`EcoreIOUtil`**
      Various methods to load and save Ecore models.

In the following, you will find a collection of hints regarding technology and concrete realizations within the project for metamodels, constraints, textual/graphical languages, model-to-model/text transformation and model interpretation.

Metamodels

- You can define metamodels in whatever format you prefer but the obvious choices would be directly with the Ecore editor, graphically with an Ecore diagram or textually with EMFatic. For EMFatic, be aware of the (unfortunate) interplay with OCLinEcore: OCLinEcore creates annotations for the *.ecore file best defined by editing the *.ecore file in the textual OCLinEcore editor. However, EMFatic uses its own format an generates the *.ecore file, i.e., whenever you recreate your *.ecore file from EMFatic, you overwrite the constraints written in OCLinEcore. To my knowledge, there is no convenient way around this so that careful development is needed: Be relatively content with the metamodel before you write your OCL constraints and, if you have to do further changes to the metamodel in EMFatic be sure to copy your constraints back into the generated metamodel.
- You may want to use opposite references in your metamodels. If you are using EMFatic, be aware of its (a bit obscure) notation for defining eOpposites (see Section 4.3 in https://www.eclipse.org/epsilon/doc/articles/emfatic/)
- You may discover that some of your metamodels share common classes. In that case, feel free to create a new metamodel that is referenced by all sharing metamodels.

Constraints

- You may define constraints in OCL or a GPL of your liking (most likely Java). The choice of technology has different consequences: OCL provides a compact syntax for constraints and integrates directly with

the respective editors. However, it will only show messages of the kind "The XYZ constraint is violated on ObjectXYZ" and ignore the message you provide for your constraint. This is due to Ecore (intentionally, for compatibility with other technologies) not reading OCL's error messages. If you define constraints in a GPL, such as Java, you have full control over evaluating the constraints, can supply custom error messages but lose the comfort of a DSL for constraints. Unfortunately, there is no silver bullet but, for the project, an advice would be to use a GPL to define the constraints for the Schedule metamodel (to get proper error messages in the textual editor) and use OCL to define the constraints for the other metamodels.

- To write constraints in Java, you will have to manually provide an override of the EValidator of the targeted metamodel. There is an example of this in the exercise material.
- Be wary of the element you define a constraint on. This defines not only the performance when a constraint is evaluated but also creates an error marker at the respective object when the constraint is violated, e.g., for the Depot models, it is much more helpful to have an error displayed with an offending coach than just the entire train.

Textual Languages

- The obvious choice for defining the textual language of the project is to use Xtext. However, in principle, you may also use EMFText.
- When referring to other elements in your textual language (i.e., non-containment references), you will have to tell your language's infrastructure how to find the respective elements via their textual representation. For this, there are *two* options (the lecture/exercise only discussed the first):
  - Scope Provider: Provides a set of candidates for each reference and has Xtext find the matching object via the object's String attribute called *name*.
  - Linking Service: Parses the provided string for the reference and allows you to find the (one!) matching element programmatically.
- To create a custom Linking Service for your project, you have to do the following:
  - Create a new class that extends `org.eclipse.xtext.linking.impl.DefaultLinkingService`
  - Override the method
    ```
    public List<EObject> getLinkedObjects(EObject context, EReference reference,
    INode node) throws IllegalNodeException
    ```
  - Within the method's body, you can check which particular reference should be resolved via:
    ```
    if (reference == MyModelPackage.eINSTANCE.getModelElement_Reference())
    ```
  - Within your custom Linking Service, the following method can retrieve the originally provided String for a reference, which can then be linked manually
    `getCrossRefNodeAsString(INode node)`
  - In the DSL project (where the Xtext grammar is located), find the *MyDSL*`RuntimeModule` class and provide a method (where *MyDSL* is a placeholder for your language's name):
    ```
    @Override
    public Class<? extends ILinkingService> bindILinkingService() {
          return MyDSLLinkingService.class;
    }
    ```

Graphical Languages

- The obvious choice for defining the graphical languages/editors of the project is to use Sirius. However, in principle, you may also use any of the other EMF technologies for graphical languages (e.g., GEF, GMF, EuGENia).
- The Sirius help is an invaluable aid: https://help.eclipse.org/2020-03/index.jsp?topic=%2Forg.eclipse.sirius.doc%2Fdoc%2Findex.html&cp=24
- When using Sirius, be aware that the graphical notation can be developed "live" in the runtime instance on a model (and then migrated to the development instance when done). This saves enormous amounts of development time.
- Sirius uses a Begin block to assign executable operations to some of its blocks. In the Begin block, if more than one operation is needed, add an empty "Change Context" operation, which permits multiple children. (Unfortunately, this workaround is Sirius developers' preferred solution.)
- Be aware that Sirius uses multiple languages for the fields that contain content that can be evaluated, most notably AQL (Acceleo Query Language) and MTL (Acceleo Model Transformation Language) (see https://www.eclipse.org/sirius/doc/specifier/general/Writing_Queries.html):
  - AQL is used as `aql:expression`, MTL is used as `[expression/]`
  - AQL seems to be the preferred solution within Sirius these days and some functionality seems to only work using AQL (not MTL). Unfortunately, it is hard to find information on this.
- Sirius also integrates OCL for its queries esp. as operations on objects and navigating a model. This allows to create compound expressions of AQL/MTL and OCL, e.g., `[if (name.oclIsUndefined()) then '' else name endif /]`. You can find a reference of OCL operations used in Sirius within the Acceleo documentation (as Acceleo and Sirius are developed by the same developer and use the same OCL integration): https://wiki.eclipse.org/Acceleo/OCL_Operations_Reference
- Sirius permits the definition and use of what they call *Service Methods*, i.e., methods defined in Java within the Sirius project that can be used from the Sirius model. This can come in handy when using complex operations that are tedious to define in Sirius or when wanting to reuse operations. One caveat to note: Call service methods seems to only be possible from AQL and not from MTL, e.g., when you have defined a service method `formatLabel()`, calling it on an appropriate object via `aql:self.formatLabel()` (AQL) should work but `[self.formatLabel()/]` (MTL) will not work. You can find information on service methods under "Sirius Specifier Manual" -> "Queries and Interpreted Expressions" -> "Writing Java Services" in the Sirius help (see above).
- In your graphical notations (esp. in the Depot graphical notation), you may want to use the Sirius block *Container*. I have noticed some peculiar behavior with that block: It seems that a Container must only have *Bordered Node* and *Container* elements as children to work properly but not regular *Nodes* or *Sub-Nodes*. However, I was not able to confirm my suspicions with official material.
- When you have defined constraints on your metamodel (e.g., via OCL or Java), their evaluation results should be integrated into the editor automatically, i.e., there should be error markers at the respective elements if a constraint is violated. However, you have to trigger evaluation of constraints manually via the graphical editor's menu under "Diagram" -> "Validate".

Model-to-Model (M2M) Transformation/Interpretation

- For defining M2M transformations, you may use any of the dedicated technologies (e.g., QVT-O or ATL) or a GPL such as Java.
    - For the transformation of GT's Station Plan model to PTS's Network model, dedicated technologies (e.g., QVT-O or ATL) are adequate.
    - For the transformation of Schedule to Timetable models, the use of Java is advisable (see below).
- The transformation of Station Plan to Network models should be relatively straight forward. However, notice that the two metamodel (and respectively their models) use different units of measure for distances. To avoid frustration: OCL defines a method `div()` on values of type integer that performs integer division, i.e., that does not produce decimals.
- The transformation of Schedule to Timetable models relies heavily on deriving information from the Schedule model used as input, e.g., train runs have to be simulated to determine the arrival and departure times at all relevant stations.
    - Determining this information is done by a model interpreter, which is best written in a GPL (similarly to the exercises).
    - The result of these calculations is a set of Timetable *models*, i.e., the procedure also has characteristics of an M2M transformation.
    - It is principally possible to write the entire procedure in an M2M language or a GPL. However, the computational part of the interpreter is the more challenging one than the M2M transformation so that I would strongly recommend writing this functionality in Java.
- For an easy integration of the Timetable generation mechanism with the Schedule language, you may want to use the generator of your schedule DSL, which you will find as Xtend file under `org.mdse.pts.schedule.dsl.generator.ScheduleGenerator`.
- When using a GPL for M2M, you can use the generated source of your metamodel to build models programmatically, e.g.,
```
Arrival arrival = TimetableFactory.eINSTANCE.createArrival();
arrival.setTrain(train);
```
- The M2M transformation/interpretation will result in a set of Timetable models. You can either save these models to disk or hand them to the M2T generator directly. The prior requires a bit of juggling with Eclipse functionality to determine how and where to store the models in the runtime instance, e.g., a sub-folder of the Schedule model that was used as input. The provided utility classes `EcoreIOUtil` and `EclipseUtil` may help you with this.

Model-to-Text (M2T) Transformation

- For defining M2T transformations, you may use any of the dedicated technologies (e.g., JET, Acceleo, EGL) or a GPL. While, for use of a GPL, Java is possible, it really has no redeeming qualities for comprehensive text generation. However, Xtend has a nice mechanism for generated text by using the integrated Xpand language. Except for Java, either one of these technologies is adequate for the task.
- If you decide to use Xtend, then you will have to deal with saving files in the runtime instance of Eclipse. The provided utility class `EclipseUtil` may help you with this.

- When generating the HTML code for each Timetable model, make sure that both arrivals and departures are sorted by their day of the week (primary) and their time of the day (secondary).
- If you want to be fancy in the generated HTML code for Timetable models, you can include a visual representation for each train and its coaches. For that purpose, you can use the images supplied for the graphical editor for Depot models, copy them as part of the M2T and reference them from the resulting HTML file.