Comparing graphical DSL editors: AToM^3, GMF, MetaEdit+

Nick Baetens

University of Antwerp
Middelheimlaan 1
B-2020 Antwerp
Belgium

Abstract

In this paper we will discuss and compare graphical DSL editors GMF and MetaEdit+ to AToM^3. We assume that the reader is familiar with the latter as its workflow and architecture will not be discussed here. For the comparison itself we will use features as multi-view, multi-user, the ability to use operational semantics, etc.

Keywords: AToM^3, GMF, MetaEdit+, comparison, architecture, workflow

1. Introduction

Domain-specific modeling (DSM) is a software engineering methodology for designing and developing systems, such as computer software. It involves systematic use of a graphical domain-specific language (DSL) to represent the various facets of a system dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique. DSM languages tend to support higher-level abstractions than general-purpose modeling languages, so they require less effort and fewer low-level details to specify a given system. [1, 2] To support DSM a number of tools or frameworks are available. In this report we will discuss three of them: MetaEdit+, GMF and AToM^3. The first two will be discussed quiet detailed as the latter one is supposed known to the reader. This report will

Email address: nick.baetens@gmail.com (Nick Baetens)

focus on the architectures and the development process using these tools. In the end we will compare them according to our findings.

Related to this work is a report by Steven Kelly [3], an employee of MetaCase. In that report a logic example is implemented both using MetaEdit+ and GMF, there is however no comparison on the workflow of both packages. They mostly focus on the time that was needed to implement the example, and MetaEdit+ came out as the winner. But this should perhaps not be considered a surprise from a writer that works with MetaCase.

Nowadays there are powerful tools for DSM. An ongoing problem is the insufficient tool interoperability which complicates the development of complete tool chains or the re-use of existing meta-models, models, and model operations. In the work of Heiko Kern [4] the approach of M3-Level-Based Bridges is presented and this approach is applied to enable the interoperability between MetaEdit+ and GMF.

In the next sections we will discuss both the architecture and workflow of MetaEdit+ and GMF, followed by a comparison of the two packages plus ATOM^3.

2. The Traffic Example

In this report we will refer to the traffic example, this example models traffic flows. The objects in this example are:

- Car
- RoadSegment: attributes *capacity* and *load*. Capacity defines the maximum number of cars that can be on the road segment, the number of cars on the road segment (= load) must be less or equal than the capacity.
- SplitSegment: subclass of RoadSegment; splits one incoming road segment in to two outgoing segments.
- MergeSegment: subclass of RoadSegment; merges two incoming road segments in to one outgoing segment.
- Car Generator: creates new cars.
- Car Collector: destroys cars.

- Traffic Light: attribute *color* with values "red", "yellow" and "green". When the light is red, cars on the road segment may not proceed.

We will try to model this example using MetaEdit+ and GMF. As we will see these packages can not implement the same features.

3. MetaEdit+

3.1. Specifications

In this section we will discuss some concepts used in MetaEdit+, for starters we will go into meta-modeling and take a look at MetaEdit's implementation. Next we will go into more details about the architecture of the environment.

3.1.1. GOPRR

Meta-modeling in MetaEdit+ is based on the GOPPRR (graphical) meta-modeling language. GOPPRR is an acronym formed from this language's base types which are Graph, Object, Port, Property, Relationship and Role.

Graph is the top-level structure of the meta-model. It defines one language or diagram technique such as Class Diagram or State Transition Diagram. The actual semantics of the graph are defined as the bindings of objects, relationships, roles and ports within the graph. Properties are characterizing attributes that can be attached to each of these other types. [5] Let us go through these language concepts in more detail: [6]

Graph specifies one modeling language, the details of each language are modeled with a separate meta-model. An object in a graph can contain subgraphs, this is called decomposition. But is also allowed for objects, relationships or roles to be linked to other graphs, this is called explosion. For these two techniques a meta-model for multiple graph types is needed. **Object** describes the basic concepts of a modeling language. Objects are the main elements of a design. They are elements that are connected together, the properties of such connections are defined by **relationships**. Inheritance (creating subtypes of other language concepts) is a good example of a relationship. Each object in a relationship plays his **role**, Superclass for example. In a graph objects, relationships and roles must be bound. A binding connects a relationship, two or more roles, and for each role, one or more objects in a graph. Binding is further specified with multiplicity. Objects can be grouped in Object Sets, these sets describe a collection of objects that can play the same role in a binding.

Graphs, Objects and Relationships can have attributes, information that these concepts carry with them. This information can be of different types: string, text, number, Boolean, collection etc, or they can contain a link to other modeling language concepts. We call this attributes **Properties**.

MetaEdit+'s implementation of the above described GOPPRR meta-modeling language was written in Smalltalk and provides useful flexibility for the meta-modeler. The meta-modeling state is always "live", i.e. the tool will automatically propagate changes in the meta-models into the models. [5]

3.1.2. The Environment

The functional architecture of MetaEdit+ is illustrated in Figure 1, as depicted there are several different tools. A tool, as the term is used within the MetaEdit+ environment, is a window type with its associated functionality, through which a user can view and possibly alter a design objects in a particular way. Most of them will be discussed in the next section. Here we will describe the environment as such. The heart of the environment is the MetaEngine, which handles all operations on the underlying conceptual data through a well-defined service protocol. [8] There is no direct communication between components at the same level, or over a common bus between components separated by more than one level: tools communicate only via the MetaEngine. The different tools request services of the engine in accessing and manipulating repository data. We will come to the repository later but due to the fact that the engine handles the communication between the repository and the different tools there is no need to duplicate the manipulation code and it will make things easier when extending the environment. The data is stored in a repository so it is possible to run MetaEdit+ as a single-user workstation environment, or simultaneously on many workstation clients connected by a network to a server. The data will be shared amongst the clients. Each client has a running instance of MetaEdit+, including all its tools and the MetaEngine, without the engine no communication would be possible. So what is in that repository? Quiet a lot actually, it is holding all the data contained in models, and also in the meta-models, in addition to user and locking information. To be more precise: object specification base containing all the method specifications represented as GOPRR concepts; symbol specification base containing all symbols needed to represent Objects, Relationships and Roles; tool related information base containing all information needed to represent conceptual objects in different tools (such

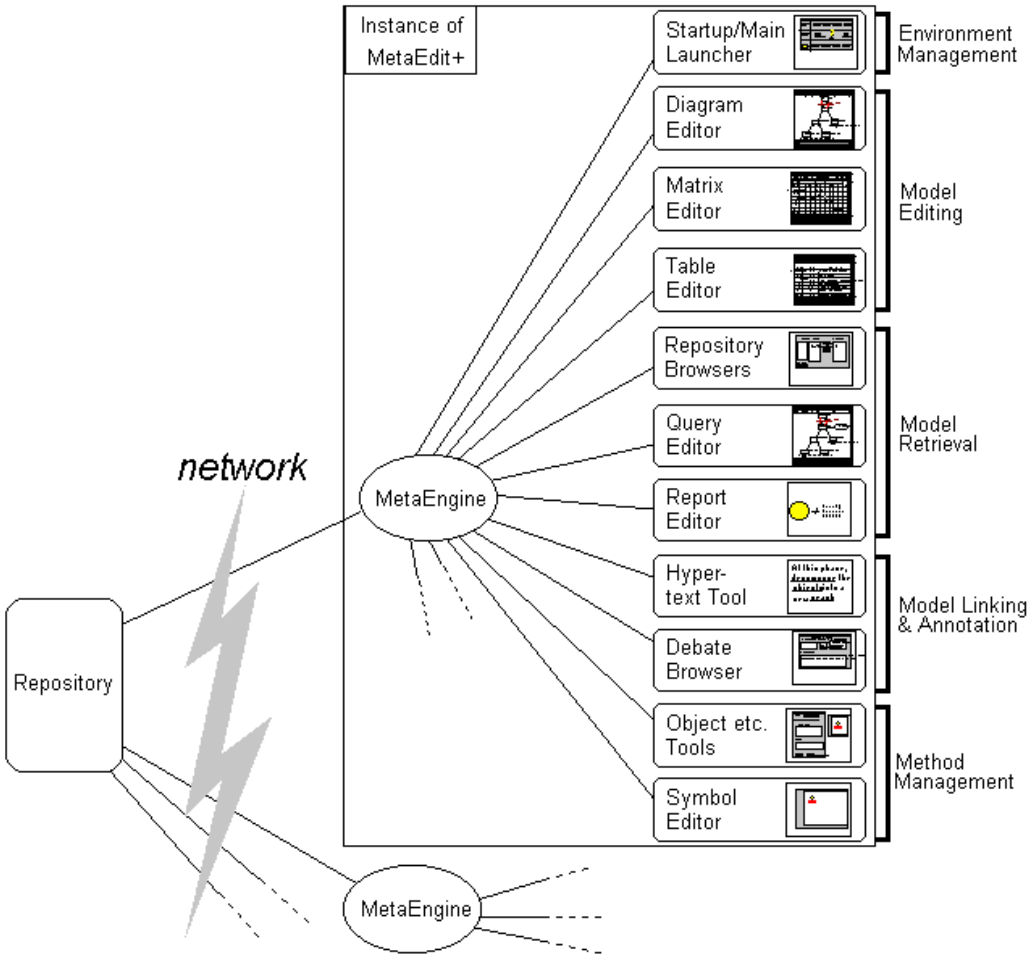

Figure 1: Multi-user environment [7]

as spatial coordinates, or size), user information base containing all information related to various users such as their passwords, access rights, or current locks held; report specification base containing all report and other output specifications. [8] When it comes to updating, it is important to know which policy is applied. It is vital to ensure that information in instance models created with the older version of the meta-model is not lost when the new version is deployed. During the development of MetaEdit+, a lot of effort has been invested in ensuring the seamless updates of meta-models and models. In many cases a conservative approach for modifying the existing meta-models and design data has been adopted. [7] For example, if a concept

is removed from the language, the creation of new instances of the type will not be possible, but existing instances of this concept are not removed from the models but rather be marked as obsolete.

In the design of the environment the tools are classified into five distinct families according to their purpose and underlying common functionality. [8]

- Environment management tools
- Model editing tools
- Model retrieval tools
- Model linking and annotation tools
- Method management tools

In the next section we will continue to describe the different tools as a workflow. Meaning that the tools will described in the order they show up in the development process.

3.2. Workflow

In this section we will guide the user through the workflow to develop a model and its underlying meta-model. When we start up MetaEdit+, the first screen that shows up is a login screen as depicted in Figure 2. The user can choose with which repository he wants to work. As already mentioned in the previous section, this repository can be local (on the user's desktop) or on a remote location (in this case a repository server runs on the network). After the user has entered the right credentials we move along to the next screen.

Figure 3 shows the meta-model browser. The middle pane shows all the meta-models in the project, the right one shows the types present in each meta-model. This browser will be the starting point to develop a new meta-model, as all the tools needed can be accessed from here. So after we created a new meta-model, we can add types to it, this is done with the Object Tool shown in Figure 4.

The Object Tool can't be called surprising, all the suspects one could expect when defining a type are there. The new type should have a name and the user can define some properties. Quiet a number of possible types for those properties are already build in. For example it is very easy to give an unique id to a type, the user can select that this id should be the creation

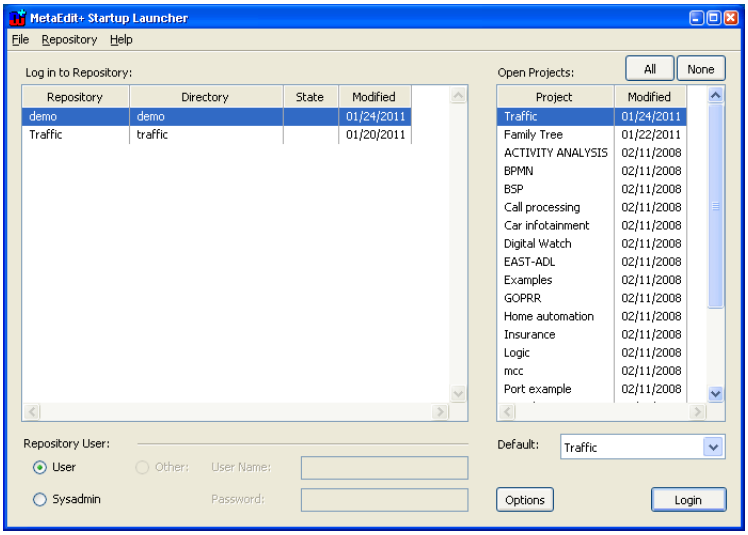

Figure 2: Repository Login

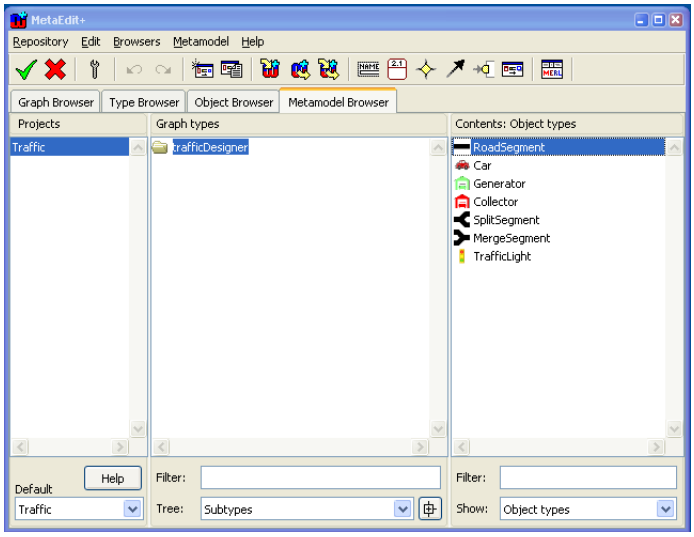

Figure 3: Metamodel Browser

time of the instance. Once the new type is saved, the user can define how this type should appear in a model. Once again MetaEdit+ offers a tool that is powerful enough to meet most of the user's requirements. The symbol editor is shown on the right side of Figure 3. The user has the freedom to draw his own symbol from scratch, but it is also possible to import some

images. The user may find it helpful to display the state of the properties, an example is shown in the picture. Both the load and capacity of a road segment will be displayed in the model. In principle the type is ready to be used in a new model. Not much can be done with it though, no relations, roles or restrictions are defined. So it is time to do so.

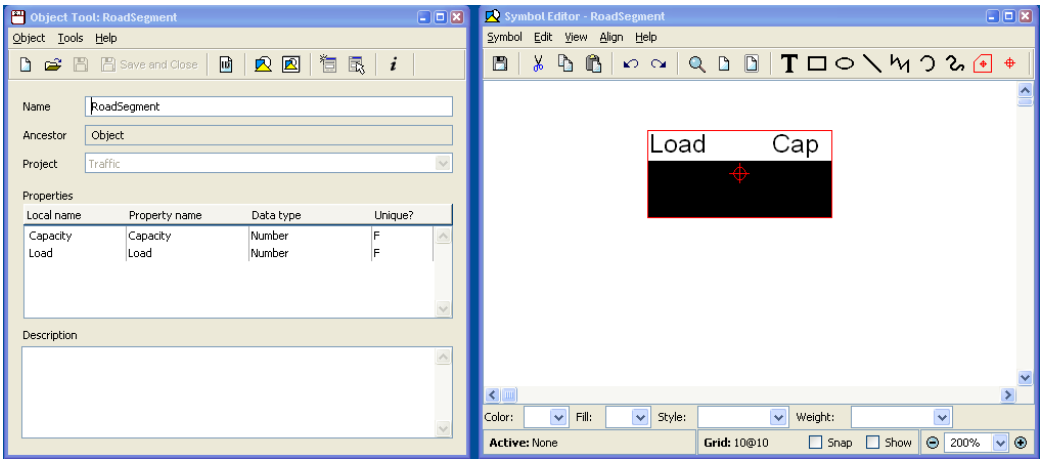

Figure 4: Object Tool and Symbol Editor

We need the Graph Tool to finish the meta-model, there are several tabs in this tool and some small features that do not feel natural. For starters there is the Types tab, there are 3 columns displayed here: Relationships, Roles and Objects. The first two are related, the third one just shows which types are defined. So we can create a new relationship in a similar way we created the types earlier, this includes the user defined symbols. If we select the newly created relationship the second column will be empty, it is necessary to define roles to go with the relationship. For example a source and a target role, here again a user defined symbol can be added. In the example in Figure 5 we choose to draw an arrow towards the target object. So speaking of unnatural, one could expect that the third column shows which types can play the selected role. For some reason MetaEdit+ chose to define bindings in a separate tab, Figure 6.

In this tab we see the same three columns again, with an additional Port column. Here every column displays the information as one could expect. A relationships has several roles, the user can select a port (this is optional) and in the last column we add every object that may play the selected role

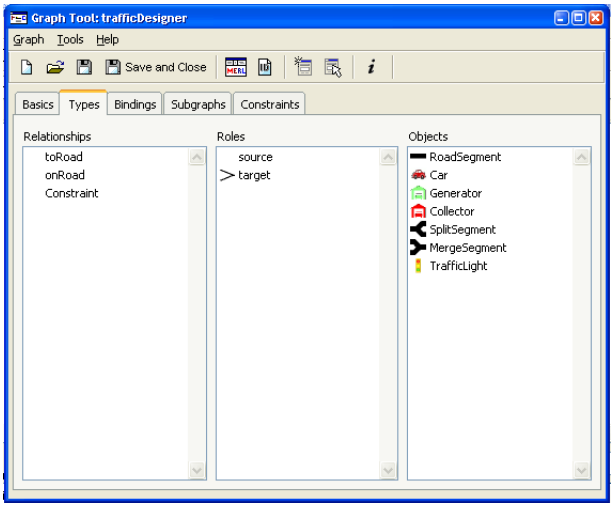

Figure 5: Graph Tool: Types

in the model. This is quiet straightforward but it can be time consuming when the number of relationships increases.

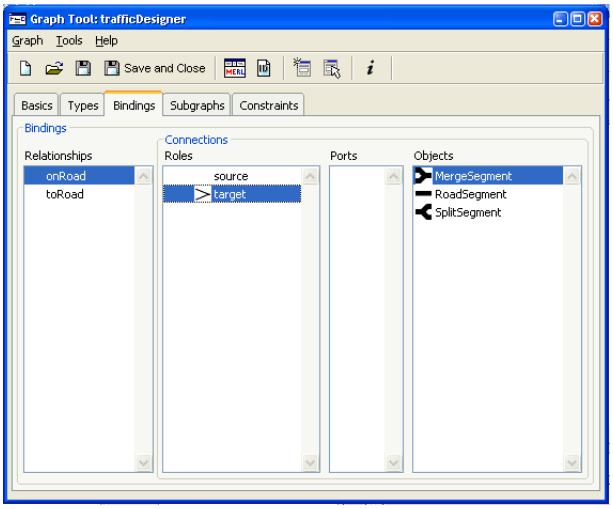

Figure 6: Graph Tool: Bindings

Now that types and relationships are defined, the user may want to constraint them. MetaEdit+ supports several kinds of constraints: [9]

- Object connectivity in the binding (e.g. an object may be in a certain

role at most a specified number of times).

- Object occurrence (e.g. an object type may have only a specified number of instances in a graph)
- Ports involved in binding (e.g. all ports of a specified type in a binding must have the same value for a specified property)
- Property uniqueness (e.g. all objects of a certain type in a graph must have unique values for a specified property).

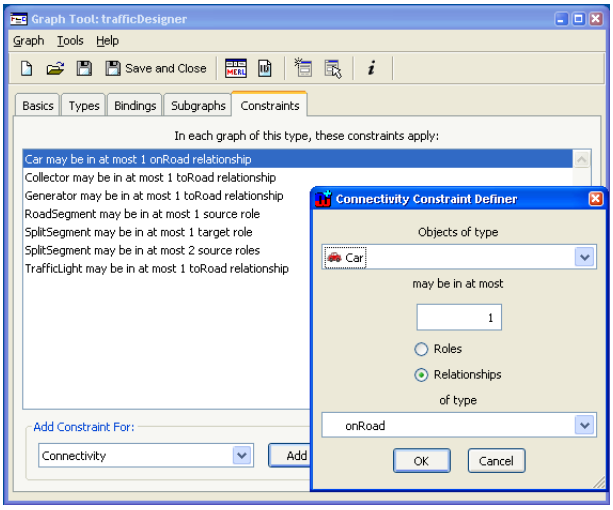

Figure 7: Graph Tool: Constraints

There is still a tab left in this window that we did not discuss, the subgraph tab. This can become useful if the meta-model becomes large, there are two kinds supported: Decomposition and Explosion. Differences between these two will not be discussed here.

Once we finished all of the described steps, our meta-model is ready for modeling. So in the main window we switch to the Graph browser, after creating a new graph the screen in Figure 8 shows up. All the defined types of the meta-model appear on top of the diagram and the user can start connecting them together. Here we see the result of the effort we put in the Symbol Editor. What if we want to put a constraint on the value of a property? The different kind of constraints mentioned above do not support

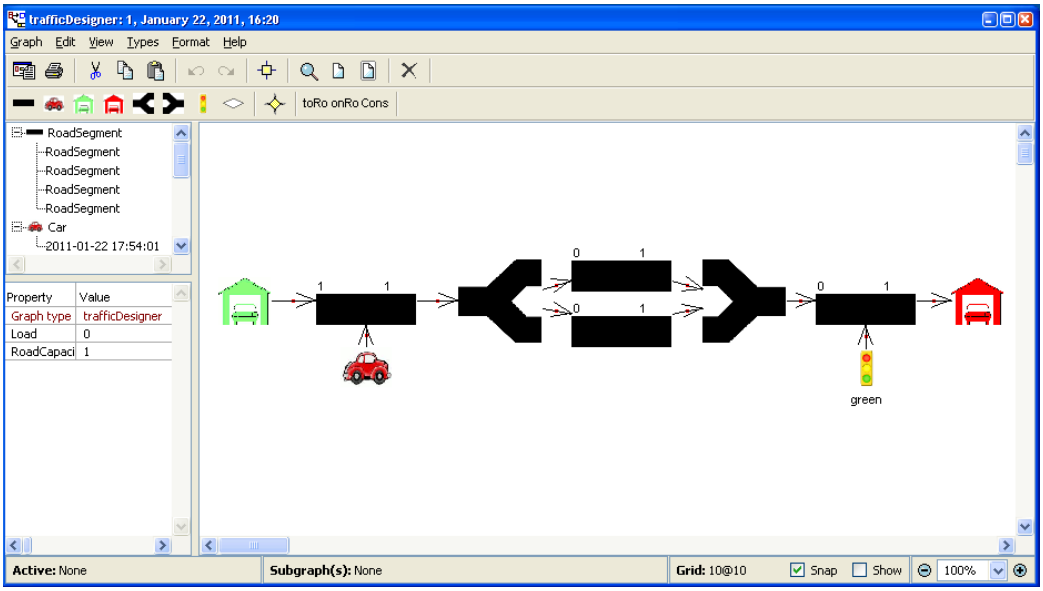

Figure 8: Model Editor

this feature. The user has to come up with his own solution, an example can be found in the example repository of MetaEdit+, the S60 phone application.

Here we want to make sure that the length of a zip code is 5 in order to be a valid zip code. These kinds of constraints have to build in the model, we are not aware of a way to build them in the meta-model somewhere.

If the user wants to generate code out of his models he should "implement" a generator. The Generator Editor is an interactive development environment for creating, editing and managing generators. It allows the user to view, edit and run available generators, and to create new generators for your needs. [9] An example Generator Editor window is shown in Figure 10. This completes the workflow of MetaEdit+, we will now switch to another package, GMF.

4. Graphical Modeling Framework

4.1. Specifications

GMF is based on two other frameworks, the Eclipse Modeling Framework (EMF) and the Graphical Editing Framework (GEF). How these two work together will be discussed later on, but let us first specify these technologies.

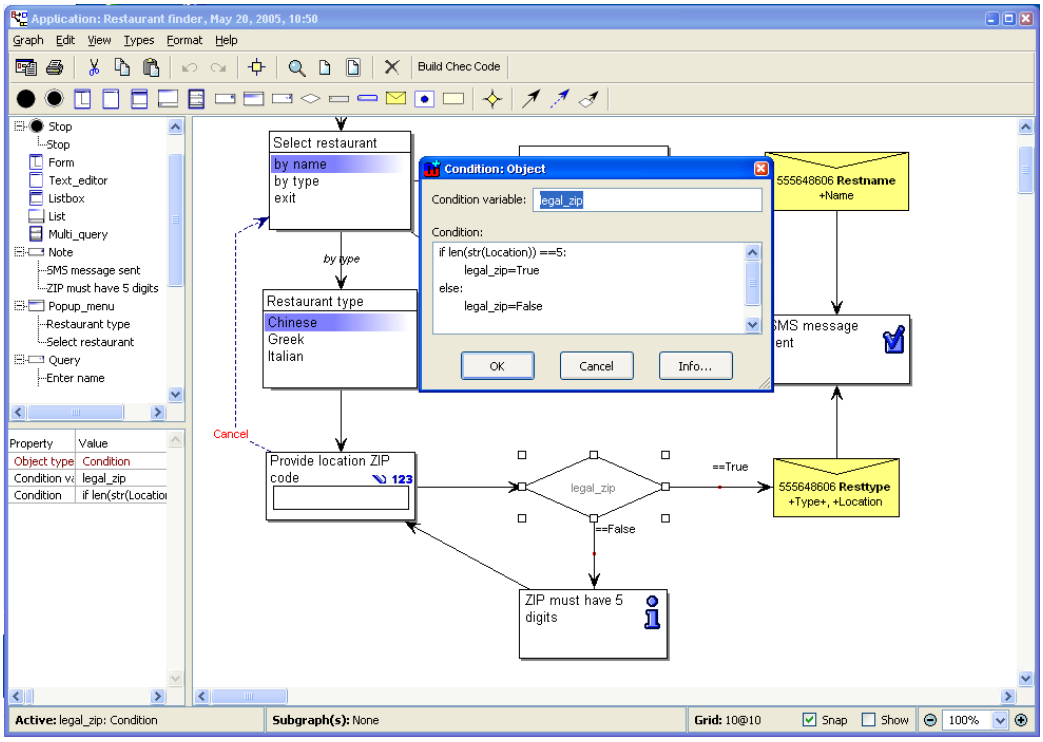

Figure 9: Condition in a model

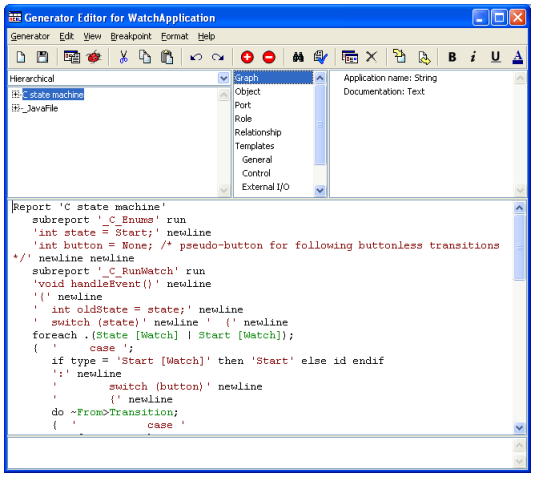

Figure 10: Generator

4.1.1. Eclipse Modeling Framework

The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. [10] EMF consists of three main parts:

- Core: EMF has its own implementation of the Essential Meta Object Facility (EMOF), called Ecore. These metamodels are domain models described in the meta-meta language MOF. The latter was designed to describe meta languages in such a way that it is possible to export and import (also across networks) created models. This is ensured by the OMGs standard format for model storage: The XML Metadata Interchange (XMI), which is based on eXtensible Markup Language (XML). [11]

- Edit: The main goal of this part is to display EMF Objects in JFace Viewers. The Eclipse user interface framework (JFace) includes a set of reusable viewer classes (for example TreeViewer, TableViewer) for displaying structured models. JFace viewers work with any kind of object. This is possible because the viewers, instead of navigating the model objects directly, access the model objects through an adapter object called a content provider. The EMF.Edit framework provides a generic content provider implementation class that can be used to provide content for EMF models. [12]

- Codegen: The EMF.Codegen provides the ability to generate Java code from a domain (Ecore) model. For each class in a given domain model, a Java interface with the needed getter and setter methods will be generated along with an implementation and factories to create instances of the domain model classes. [11]

4.1.2. Graphical Editing Framework

The Graphical Editing Framework (GEF) provides technology to create rich graphical editors and views for the Eclipse Workbench UI. [13] These editors consist of several components:

- The diagram editor including tool palette

- Figures which graphically represent the underlying data model elements
- EditParts which match figures and their respective model elements
- Request objects for user input
- EditPolicy objects which evaluate the requests and create appropriate command objects
- Command objects that edit the model and provide undo-redo

Some of these components will appear in the workflow described in the next section. Our goal is to create an Eclipse plug-in, which will implement a graphical editor for a specific domain model. It is important to note that GEF makes no restrictions on the underlying model, it can be an EMF model, Java code, etc. But as we will see later, we have to abandon this freedom. GEF follows the MVC (model-view-controller) concept, meaning that there is a separation between the model, its graphical representation (view) and the program logic (controller). [11] In MVC design, the controller is often the only connection between the view and the model. The controller is responsible for maintaining the view, and for interpreting UI events and turning them into operations on the model. We will explain what happens if we use EMF and GEF together in the next part.

4.1.3. Bringing the parts together

Our goal is to generate a GEF editor from an EMF model, so we want to build models in a graphical environment on top of a meta-model. The meta-model will be managed by EMF, the graphical part by GEF, but we need to stick these two together. GMF makes it easier to complete this job by offering a guided way to do so. There are two main parts: a Runtime Environment (which extends some feature of both EMF and GEF) and a Generation Framework. The last one contains special editors to handle the GMF models and a generator which produces the editor code from the GMF models. So what are these GMF models? There are three models to define the domain and visual representation, further there is one more to generate the code of the plug-in. These models will be discussed in the next section but we will descibe the general picture here. Figure 11 shows what we are aiming for.

On the left there is the meta-model (defined in EMF, a .ecore file), on the right there is a visual representation (GEF), what about the gap in the

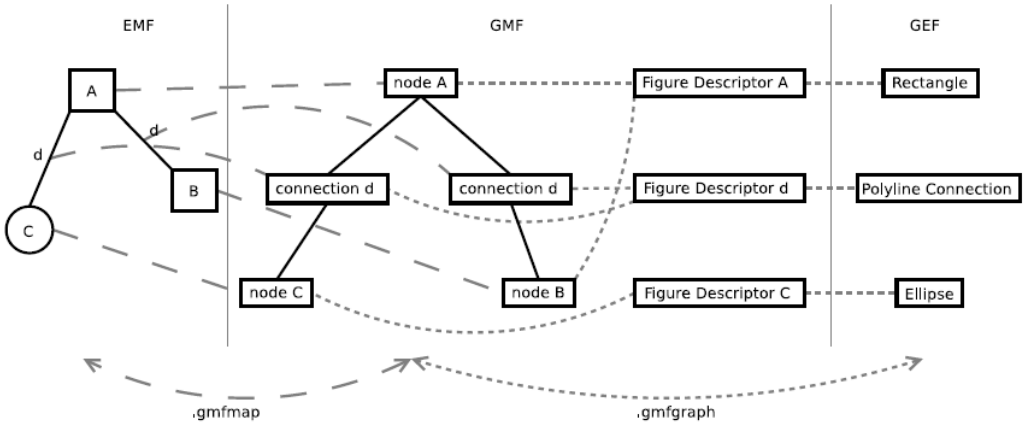

Figure 11: GMF: The glue between EMF and GEF

middle? Every element of the domain model which shall be displayed needs a representative in the graphical definition model. For every class there will be a node entry, for each connection a connection entry. There will be a figure descriptor assigned to each entry, as the name already reveals this descriptor defines the visual appearance. Or in other words it will tell GEF how to draw the according model element. Figure descriptors can be shared among several entries so these entries will have the same visual representation.

4.2. Workflow

Developers who create graphical modeling-like editors using GMF follow this simplified workflow as depicted in Figure 12:

- Create a domain model, this model defines the non-graphical information managed by the editor
- Create a diagram definition model, this model defines graphical elements to be displayed in the editor
- Create a diagram mapping model, this model defines mapping between domain model elements and graphical elements
- Generate the graphical editor
- Enhance the graphical editor by editing the generated plug-in code

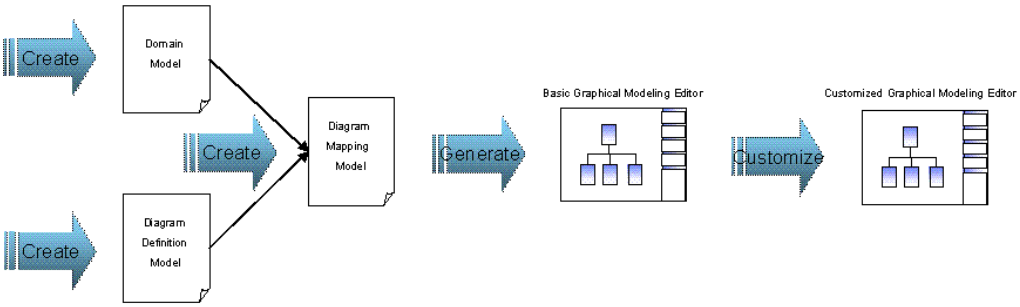

Figure 12: GMF Workflow

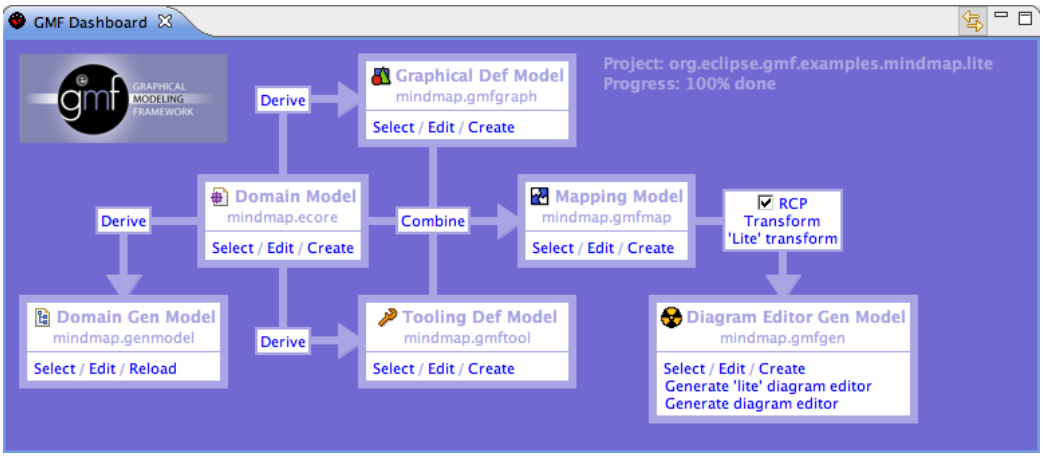

Figure 13: GMF Dashboard

To assist the user going through the different steps, the GMF presents the Dashboard. The Dashboard, as depicted in Figure 13, will tell the user what to do next. In theory nothing can go wrong if the user follows the arrows in the Dashboard, in theory that is. The view is very useful but it lacks a crucial step to come to working model editor. Tutorials that show you how to get started with GMF jump right into the wizards that are provided as part of the SDK. The wizards and Dashboard that are used to develop GMF applications are very powerful. With the exception of the data model, all of the configuration files can be generated from wizards. [14] Herein lies the danger, since the user is not always knowing what he is doing. We will explain the role of every model and mapping in the sequel.

4.2.1. Domain Model Definition

The first model we run into is the Ecore Model, as one could suspect from the Dashboard, this will be the heart of the model editor we are developing. All types and relationships between them are defined in this model. The meta-model is displayed as a tree (Figure 14). No visual representation is added here, so there is no such thing as a symbol editor involved in this stage. This will be the next step, introduce a new model to visualize things.

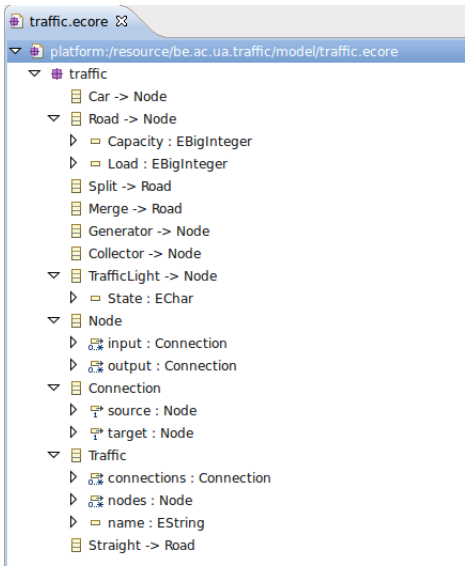

Figure 14: Ecore Model

4.2.2. Graphical Definition

So as we just said, in this step we define a gmfgraph file, which will be used in the diagram to display classes from the domain model. Basic building blocks are nodes, links, etc. But is also possible to link some images. Note that is not possible to create a symbol as it was with MetaEdit+. This representation can be generated automatically with a wizard. There is nothing of great interest in this gmfgraph file, the user can leave the default options in it or experiment with it.

4.2.3. Tooling Definition

The gmftool file is a tooling definition that defines what text you want to display on the tool palette and the button's tool tip. So when the user

starts the generated Eclipse plug-in and wants to start modeling there will be an editor with some palettes. One of them will contain all the types and relationships the user has defined earlier. In this step we add the meta-model to this palette.

4.2.4. Mapping Definition

So now that we have these different models it is time to bring the pieces of the puzzle together, this done in the gmfmap file. This is perhaps the most difficult step because errors in the previous discussed model will come to the surface here, on the bright side is the fact that there is a wizard for this model as well. In short we have to tell GMF what action to take when a tool is selected, what classes are to be created, and what figures to render when those classes are added to the diagram. As shown in the Dashboard it is this model that will as input to a transformation step which will produce our final model, the generation model.

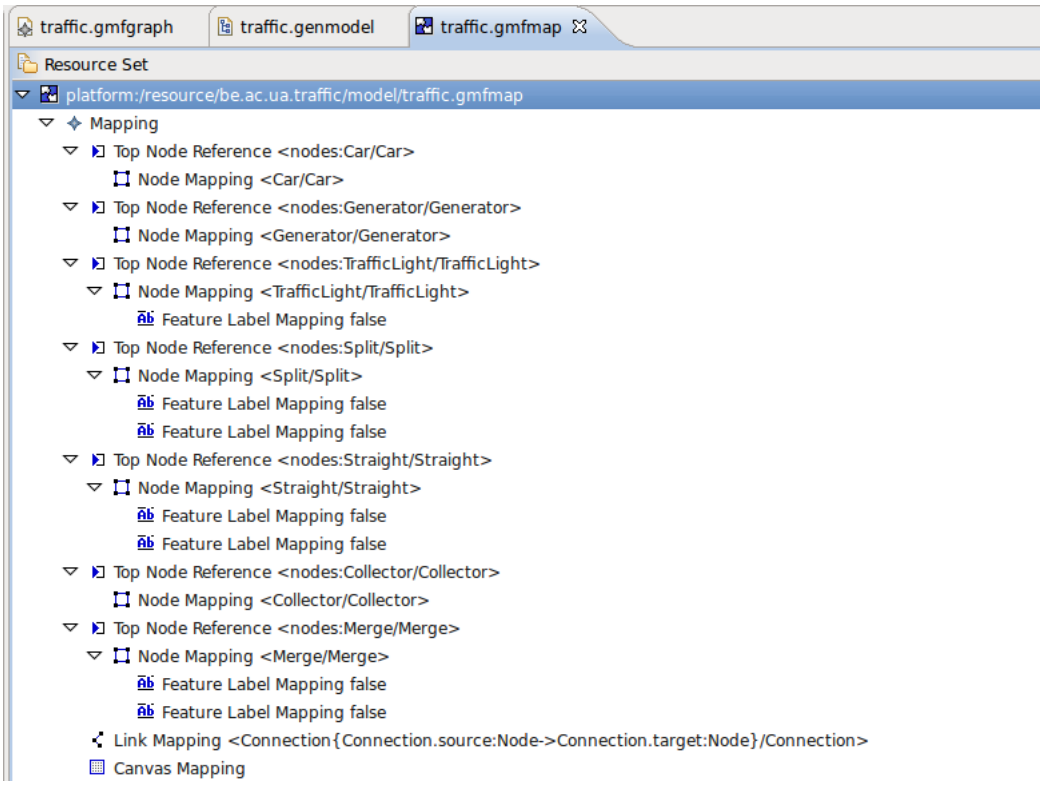

Figure 15: Mapping

4.2.5. Code Generation

Before we can actually run the plug-in we have to generate code out of the models we just developed. To do this we need a gmfgen file in order to set the properties for code generation. As a modeler there is no need to understand all things that are executed in the background once the "generate" button is clicked. The framework will take care, finally the plug-in is ready for an initial test.

4.2.6. Start Your Plug-in

If all the previous steps are executed correctly a similar screen as depicted in Figure 16 should appear. From our experience we can say that it can take a while before everything is working as it should be. Especially if we consider the time it takes to generate some models and the code over and over again in order to correct errors. In the next section we will compare the features of the different packages.

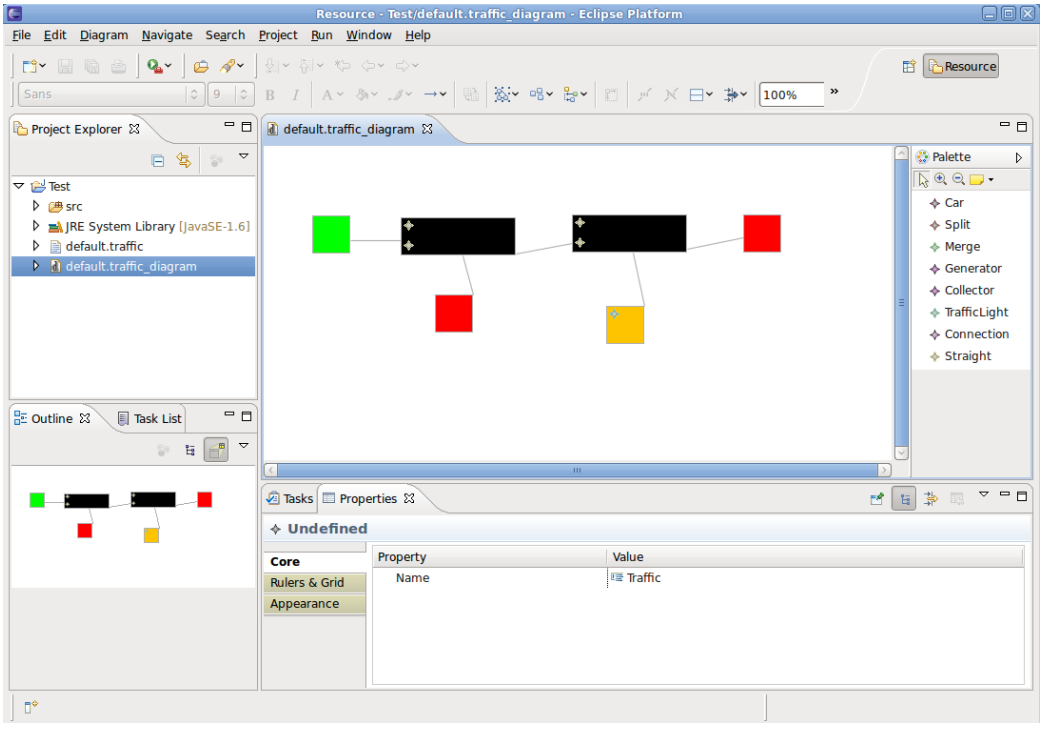

Figure 16: Model Editor

5. Comparison

In this section we will make a comparison between GMF, MetaEdit+ and AToM^3 . The first features we discuss can not be called spectacular but we go through some low level differences in the rest of the feature list we propose here.

5.1. Stand-Alone Application

AToM^3 and MetaEdit+ deliver a complete package which can be used as a whole. GMF on the other side is only a plug-in to go with Eclipse. This can be seen as an advantage as the user can already be familiar with Eclipse as an development environment. This will be more of a plus for the end-user (model designer, not the meta-model designer), as he can just switch perspectives from modeling to actual coding. For the non-Eclipse fans, the first two applications will be a more appropriate choice.

5.2. Meta-Model

Where AToM^3 has a graphical representation of a meta-model, GMF uses a tree representation. In MetaEdit+ the meta-model is spread among different tools and there is no place where the meta-model is displayed as a whole. It is hard to tell which one is best because this can be vary due to the users' habits and needs.

5.3. Build Models

For this feature GMF differs from MetaEdit+ and AToM^3 , the latter two let the user build models out of meta-models in the same environment. With GMF this is not strictly the case, here we build our meta-model and graphical mapping and generate a new plug-in. The editor for the models is in fact a new application, but it depends on GMF of course. For meta-models that are stable this will not be considered a drawback, but for evolving ones this makes the update cycle longer. On the other hand it can be seen as an advantage that the end-user only needs the generated plug-in, it is not necessary to ship your meta-models to them.

5.4. Symbol Editor

This will not be considered a major issue but it is worth mentioning that only MetaEdit+ and AToM^3 contain a symbol editor. In GMF the user can only choose between some basic shapes (such as a rectangle) which can be

colored. For users who do not consider themselves an artist, all three packages are able to import external files for the symbols. Another difference is that in MetaEdit+ and AToM^3 it is possible to display the status of an attribute of a model element. For example the load of a road segment in the traffic example. GMF does not seem to support this feature or it is pretty well hidden.

5.5. Multi-user

As we already discussed MetaEdit+ supports multi-users right out of the box as it uses a repository to manage the models. It will be another story for GMF and AToM^3 as no such feature is included in the standard release. It is of course possible to use a version control tool together with the modeling tools but that will only help to distribute the models amongst several users. Features like the conservative approach used in MetaEdit+ will have to implemented by the user himself, which makes it a lot harder to develop/use. With GMF we also take into account that different versions of a meta-model probably will not be detected as a new plug-in is generated every time the meta-model changes. Collaborative model (not meta-model) development can be achieved with the use of the Subclipse plug-in, as long as it is not simultaneous.

5.6. Multi-view

Related to the previous feature, but not quiet the same, is the multi-view feature. In MetaEdit+ the user can open a model as a diagram, a table or as a matrix. These three views will remain consistent all of the time. This is made possible by the GOPRR principle. Unlike OPRR the GOPRR model allows multiple representations of the same underlying conceptual object (e.g. graphical, matrix, text), and even different graphical representations of the same object in the same representation paradigm. This is achieved by making available mechanisms that can override the default representation. In this sense GOPRR forms a true conceptual kernel on which varied representations of data, including not only graphical diagrams but also hypertext, text and matrices, can be built. This allows GOPRR to support a wide range of methodologies including matrix, table or text oriented ones, and gives users the ability to see and manipulate design information in a variety of representations. [8] The other two packages do not include this feature in their standard distribution. However, this does not mean that multi-view can not be achieved. For AToM^3 it is shown that it is possible to implement

with the Observer design pattern. This allows easy to implement consistency management. To allow for the transformation between different view types, transformation rules have to be defined. The AToM^3 tool provided an easily extendable environment and already offered the support for defining transformation rules. [15] In GMF multi-view is almost impossible to achieve, the main reason is that it will be necessary to adapt the GMF framework to our needs. We will have to define several different approaches to generate a new plug-in out of a meta-model. Currently there is only one way available but the multi-view functionality itself should be possible to implement in a similar way as they did with AToM^3 . Also recall that MetaEdit+ uses different tools, each user can operate several tools simultaneously where each tool provides a different view to the same object. [8]

5.7. Consistency Model

For the consistency model we see three different approaches, this will allow us to make a ranking based on the time it is needed for an update. Let us start with MetaEdit+, this is without doubt the faster of the three, changes in the meta-model are reflected immediately in the model. We assume that the tool for meta-modeling and the model editor are both open. If we then add a property to one of the types and change its visual representation to display this additional property the model will be changed once we switch to the editor tool again. This is a good example why locking can be used in the repository, when two users are working simultaneously, one modeler and one meta-modeler, the modeler can lock the meta-model to prevent his model from updating. In GMF we see the complete opposite, as mentioned before GMF is strictly used to develop a plug-in with a meta-model in it. The actual modeling is done using that plug-in. So if we want to update the meta-model we have to regenerate the plug-in and update the Eclipse environment to use the new version. Remark that the update process is by no means automated, the user has to update his Eclipse version manually or at least give the command to actually perform the update. This is by far the slowest of the three approaches, therefore we would not recommend to use GMF with instable meta-models. Finally we come to AToM^3 , this approach can be placed in the middle of the previous ones. If the user wants his models to be updated according to the meta-model he has to reload it. So if one runs several instances of AToM^3 , one to change the meta-model and one to change the model, the changes will not be reflected in the other instance as long as the user does not reload the model. So basically if you

open a model with MetaEdit+ and AToM^3 the user knows that he is working with the latest version of the meta-model at that time. Once the model is loaded, only MetaEdit+ for fills this promise. In GMF you are working with the latest version at the time the plug-in was generated.

5.8. Static Semantics

Here we will focus on the way constraints can be defined in the different packages. In AToM^3 constraints can be defined in Python, it is possible to lay constraints on relationships (for example: a car can be only on one road segment at the same time) but also on attributes of an object (the number of cars on a road segment (attribute: load) must be lower or equal than the capacity of that road segment). As we discussed before MetaEdit+ only supports constraints on relationships out of the box. Constraints on attribute values can not be included in the meta-model, a trick must be applied at the model level. As shown in Figure 9, it is possible to enforce that a zip code is of length 5 for example but we hesitate to call this kind of construction a constraint. GMF on the other hand, supports constraints defined in the Object Constraint Language (OCL). OCL was introduced in an effort to complement the UML meta-language. This language is able to describe additional relevant aspects of a specification which cannot be described in UML itself. For example it can express some additional constraints about the objects in the model. Such constraints are often described in natural language. Practice has shown that this will always result in ambiguities. In order to write unambiguous constraints, so-called formal languages have been developed. The disadvantage of traditional formal languages is that they are usable to persons with a strong mathematical background, but difficult for the average business or system modeler to use. [16] This is where OCL comes in as it is a formal language but without the massive mathematical expressions, it aims to be easy to read and write. OCL a descendant of the Syntropy[1] method

[1]Syntropy is an object-oriented analysis and design method developed at Object Designers Limited in the UK in the early 1990s. The goal in developing Syntropy was to provide a set of modelling techniques that would allow precise specification and would keep separate different areas of concern. The approach was to take established techniques, as found in methods such as OMT and Booch, and reposition and refine them. Graphical notations were much favoured by the market but lacked rigour, so Syntropy adopted ideas from formal specification languages, specifically Z, to provide tools for both precise definition of the graphical notations and for the construction of richer models than are possible with pictures alone. [17]

and was originally intended as a business modeling language within the IBM Insurance division. Today it is part of the UML standard. In the sequel we will describe some properties and uses of OCL, and introduce some examples.

- **OCL is a pure specification language.** This means that the evaluation of an OCL expression will not change the state of the model. OCL can however describe the change of a state in order to evaluate a postcondition for instance but it will never actually execute the change. When an OCL expression is evaluated, it simply returns a value. As a specification language, all implementation issues are out of scope and cannot be expressed in OCL.

- **OCL is not a programming language.** OCL expressions are not by definition directly executable, it is not even possible to write program logic or flow control in OCL.

- **OCL is a typed language.** Each OCL expression has a type (a list of types can be found in the specification [16]). To be well formed, an OCL expression must conform to the type conformance rules of the language. (For example, it is not allowed to compare an Integer with a String.)

- **The evaluation of an OCL expression is instantaneous.** This means that the states of objects in a model cannot change during evaluation.

Some general examples:

- The number of bosses who have more than 10 reports:
  $company.employees \rightarrow select\ (title = \text{"Boss"}\ and\ self.reports \rightarrow size > 10)$

- The set of pilots who have enough training hours:
  $flight.pilot.training\_hours \geq flight.plane.minimum\_hours$

So how does this works in GMF? The constraints must be defined in the mapping model as shown in Figure 17.

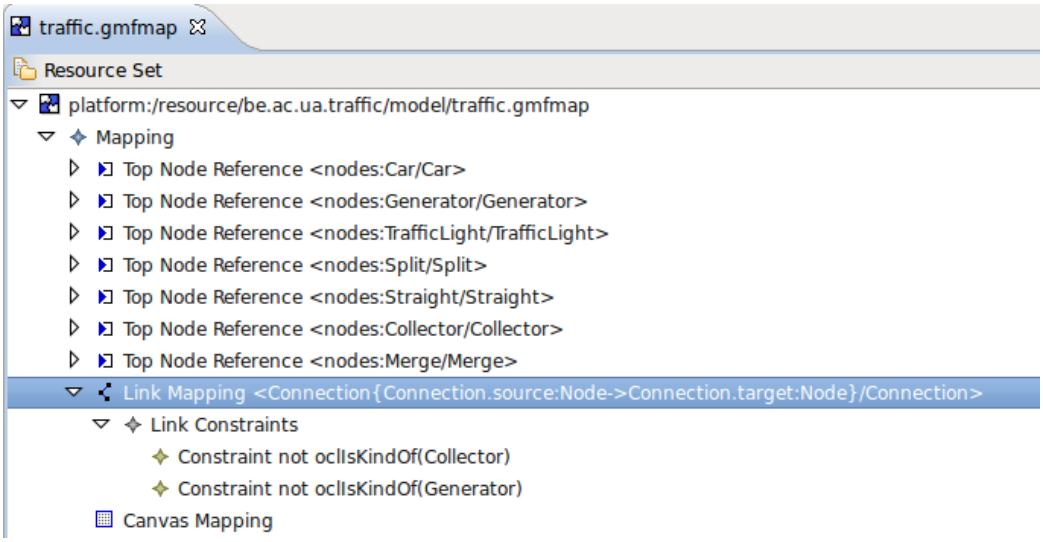

Figure 17: OCL constraint in GMF

The constraints in the example enforce that there may not be an incoming connection to a car generator and no outgoing connections from a car collector. Constraints on relationships will avoid the creation of an element if the constraint will be violated if the element was created. So we will not be allowed to draw an outgoing connection form a collector. The general examples can also be adopted to apply on the traffic example: $self.load \leq self.capacity$ with *self* a road segment. As such the domains of the attributes of an object can be defined. After the static semantics it is time for operational semantics.

5.9. Operational Semantics

In MetaEdit+ an API is included which provides the interface to read, create, and update model elements, as well as control MetaEdit+ for scripting or simulation support, it is important to note that the dynamic link is real-time. Modifications made through the API are reflected immediately in the different tools. There are several uses possible with the API, to name two of them: Simulation and Model Transformations. The first one let us the user animate models while running code. For instance a car will move from road to road in the traffic example. Model transformations support updating all models with similar changes. The MetaEdit+ API uses the widely supported and open SOAP / Web Services / .NET standard for application

integration, making MetaEdit+ functions accessible from almost any programming language and platform. GMF on the other hand will only support Java, but also is platform independent (thanks to Java). In GMF there is also an internal representation and API where we can play want if we want. The internal representation of the diagrams is again a tree. It is possible to change the properties of the elements such as their place and size. To put this in practice there is some coding work to do, in fact an additional plug-in must be developed in order to implement a simulation for example. Unlike AToM^3 there is no debug window to help the user, the same story as with consistency model applies here. It takes a long time to produce correct code. So why we need an additional plug-in, the main reason is the invocation of the simulator. Some button or menu-item must be added to Eclipse, the associated action will then be the actual simulation. Nevertheless, it is possible to implement and as a proof of concept we would like to refer to a tool developed by the writer. This tool is also a plug-in for Eclipse and serves to check UML models for consistency. In order to represent inconsistencies some notes are added to the diagram and moved to a position relative to the inconsistent model element, this is depicted in Figure 18. To implement this tool we actually took advantage of the mapping between EMF and GEF we discussed in the previous section. We added some figure descriptors, these are supplied to GEF which draws these figures on screen but we did not add a mapping to the actual model. So we altered the diagram without changing the underlying model. This makes sense because we do not want to add semantics to the model by representing inconsistencies, the representation only helps the user to understand where he made a mistake. To implement a simulation we need to change both the model and the visual parts. To create and delete model elements we need to mess around with the model, to change the place of objects in the representation we need to change the visual parts. Another issue is the fact that GMF does not provide direct access to the model, it is not possible to access the model that is loaded by GMF. The user has to build classes that load the model again, then the model can be altered and finally saved. Once it is saved the editor will refresh and display the changes. So when implementing a simulation the user has to save the model after each modification manually. AToM^3 support similar features as MetaEdit+ except that is restricted to Python as programming language. Transformation rules are only supported by AToM^3 in the standard distribution. But MetaEdit+ and GMF can be extended to do so, this will require some coding work. On top of that it will no be possible to program the rules

in a visual way like in AToM^3 but rather through the API's. This concludes our comparison of the different packages, it is time to come to a conclusion based on our findings.

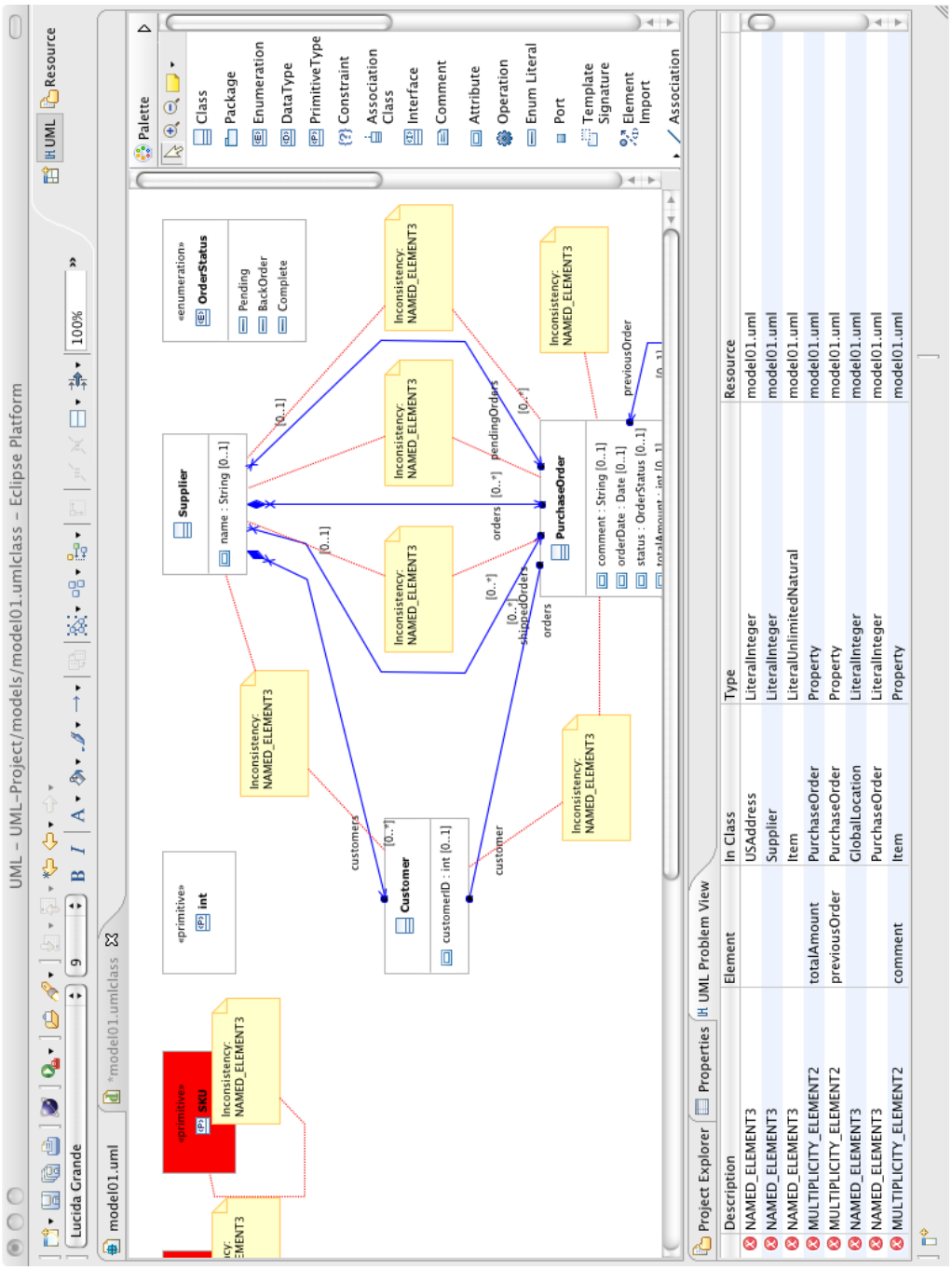

Figure 18: Resclipse

6. Conclusion

If it comes to a conclusion about what DSL editor is best, it is probably useful to distinguish industrial environments from research bases. In industrial environments one is looking for stable applications that work out of the box and suit the needs without having to develop some extra features. Many good things can be said about AToM^3 , but when it comes to stability we have to admit that there is room for improvement. This rules it out of the race at the moment. GMF offers a stable framework, but perhaps it lacks some essential features to make it a success in industry. Collaborative development for instance, is harder to achieve in GMF than it is with MetaEdit+. So if we round up, MetaEdit+ is probably the best choice in an industrial environment, it offers quite a number of features the other packages lack. But on the debit side is the price of the package, licenses are pretty expensive. We can not judge if this investments pays of, it probably does because MetaCase became a successful company.

If the user is doing research, he probably will not be willing to buy an expensive license. In that case we have to choose between AToM^3 and GMF. These two will be competitive in this setting and it will be more less a subjective choice. If the user is familiar with Eclipse he might choose the GMF, it will take some time before he really becomes acquainted with the smaller details of the package. GMF is perhaps not the most intuitive package, AToM^3 on the other hand suffers from the same disease sometimes with mouse commands that are not straightforward. But it probably all comes down to habit. AToM^3 has the advantage that supports operational semantics better than GMF, in that case it depends on the features the user actually needs which package is right for him. So making a comparison it not that hard, making a ranking is. There are a lot of aspects a user can base his decision on, that's why a sole conclusion can't be drawn. We hope that our work can contribute in making the right decision to suit the user's needs.

References

[1] "http://en.wikipedia.org/wiki/domain-specific_language/."

[2] "http://en.wikipedia.org/wiki/domain-specific_modeling/."

[3] S. Kelly, "Comparison of eclipse emf/gef and metaedit+ for dsm.," *In 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Workshop on Best Practices for Model Driven Software Development.*, 2004.

[4] H. Kern, "The interchange of (meta)models between metaedit+ and eclipse emf using m3-level-based bridges," in *8th OOPSLA Workshop on Domain-Specific Modeling at OOPSLA 2008* (J. Gray, J. Sprinkle, J.-P. Tolvanen, and M. Rossi, eds.), pp. 14–19, University of Alabama at Birmingham, 2008.

[5] R. Pohjonen, "Metamodeling made easy – metaedit+ (tool demonstration)," 2005.

[6] MetaCase, "The graphical metamodeling example," 2008.

[7] J.-P. Tolvanen and S. Kelly, "Metaedit+: defining and using integrated domain-specific modeling languages," in *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, (New York, NY, USA), pp. 819–820, ACM, 2009.

[8] S. Kelly, K. Lyytinen, and M. Rossi, "Metaedit+ a fully configurable multi-user and multi-tool case and came environment," in *Advanced Information Systems Engineering* (P. Constantopoulos, J. Mylopoulos, and Y. Vassiliou, eds.), vol. 1080 of *Lecture Notes in Computer Science*, pp. 1–21, Springer Berlin / Heidelberg, 1996.

[9] "http://www.metacase.com/support/45/manuals/meplus/mp.html."

[10] "http://www.eclipse.org/modeling/emf/."

[11] M. Rohs, "A visual editor for semantics specifications using the eclipse graphical modeling framework," Master's thesis, University of Paderborn.

[12] "http://org.eclipse.emf.doc/references/overview/emf.edit.html."

[13] "http://www.eclipse.org/gef/."

[14] "http://onjava.com/pub/a/onjava/2007/07/11/
gmf-beyond-the-wizards.html."

[15] B. Smets, "Multiview modelling using atom3," 2010.

[16] Object Management Group, "Ocl 2.2 specification," 2010.

[17] "http://www.syntropy.co.uk/syntropy/."