LWC11 – MetaEdit+

Version 1.0.1, May 20, 2011
Based on LWC Task Version 1.1, October 15, 2010

Steven Kelly
stevek@metacase.com

Abstract

MetaEdit+ is a mature language workbench that supports graphical diagram, matrix and table representations. We demonstrate it by completing the tasks of the 2011 Language Workbench Competition from http://languageworkbenches.net.

Getting started

You can obtain an evaluation version of MetaEdit+ from www.metacase.com/download. Follow the instructions in the evaluation email / below to install MetaEdit+. Do not log in yet.

Windows

Run the installer and accept defaults. After installation you will have:

- MetaEdit+ app → C:\Program Files\MetaEdit+ 4.5 Evaluation
- User files & repositories → ...\My Documents\MetaEdit+ 4.5

MetaEdit+ starts automatically after installation. You can also start it from the Start Menu.

Mac OS X

Open the distribution .dmg and drag the MetaEdit+ application into your Applications folder.

Execute MetaEdit+ by clicking its icon in Applications.

- First execution of MetaEdit+ will create required folders and files under your Documents folder, which takes about 20 seconds.

Linux

Execute the following set of commands from the shell (requires root privileges or sudo):

```
cd /usr/local
sudo mkdir mep45eval
cd mep45eval
sudo tar -xzvf ~/Desktop/Linux/MetaEdit.tgz
export PATH=$PATH:/usr/local/mep45eval
cd ~
mesetup
```

Executing mesetup will create ~/metaedit and place user files and repositories there.

Start MetaEdit+ by executing:

```
cd ~/metaedit
metaedit
```

Do not log in to the demo repository: we want to get the LWC11 repository.

The repository for the LWC tasks can be downloaded from:

- http://www.metacase.com/support/45/repository/LWC11.zip

Unzip the LWC11 repository directory structure into your MetaEdit+ working directory:

• Windows:	My Documents\MetaEdit+ 4.5
• Mac OS X:	Documents/MetaEdit+ 4.5
• Linux:	~/metaedit

In the MetaEdit+ Startup Launcher, press F5 to update the repository list, which should now show LWC11 as well as the standard demo repository. (If you can't see it, make sure you unzipped correctly, giving paths like My Documents\MetaEdit+ 4.5\LWC11\manager.ab.)

Select the LWC11 repository, and you will see the projects it contains in the list on the right.

• Done:	The finished state of the main LWC tasks
• Empty:	An empty project, ready for you to implement the tasks yourself
• GOPRR:	The graphical GOPRR modeling language

You are now ready to begin with the tasks. The rest of this document gives instructions for completing the LWC tasks yourself. If instead you want to follow along with the tasks in a read-only mode, you can open the Done project instead of the Empty project, and simply open the existing graphs by double-clicking them, rather than creating new ones. It won't be as much fun though!

Phase 0 – Basics

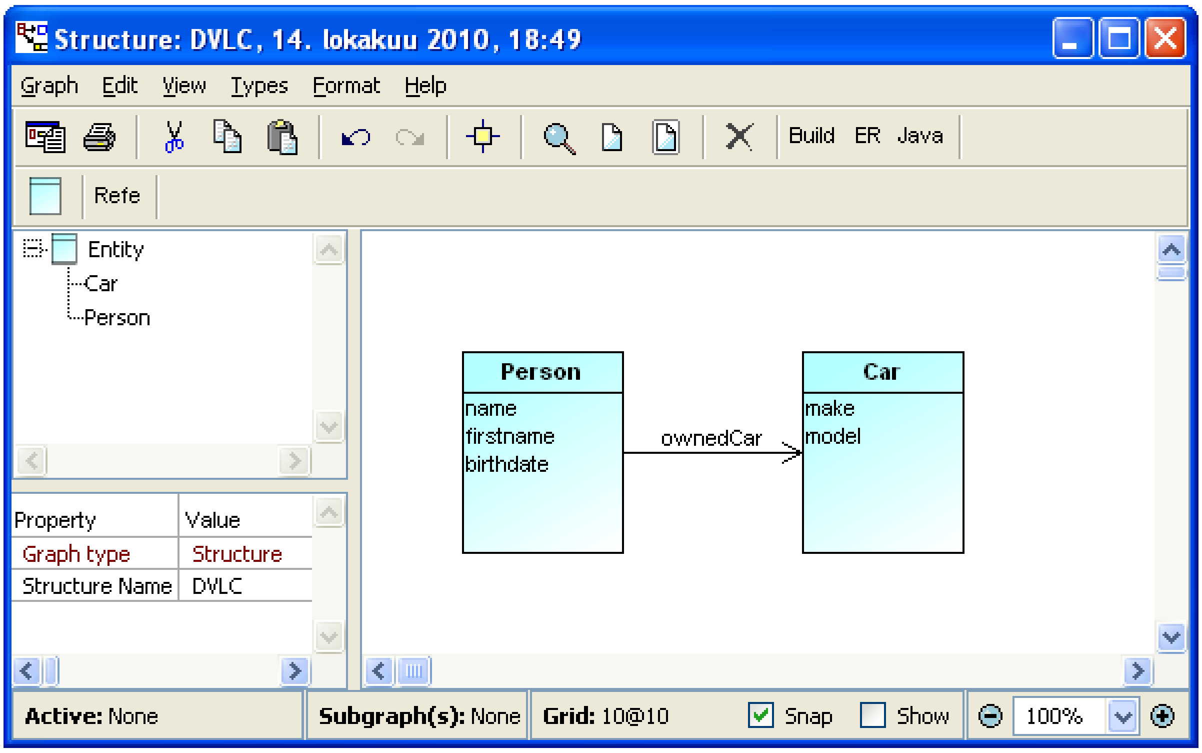
This phase is intended to demonstrate basic language design, including tool support (e.g. for textual languages IDE code completion, syntax coloring, outlines, etc., or for graphical languages the editor and expected modeling tool functionality).

0.1 Simple (structural) DSL without any fancy expression language or such.

- Build a simple data definition language to define Entities with Properties. Each Property has a name and a data type. The data type should be either a primitive type or an Entity.

```
entity Person  {
   string name
   string firstname
   date birthdate
   Car ownedCar
}

entity Car {
   string make
   string model
}
```

First let's see what that example model might look like as a graphical diagram:



Property	Value
Graph type	Structure
Structure Name	DVLC

Active: None Subgraph(s): None Grid: 10@10 Snap Show 100%

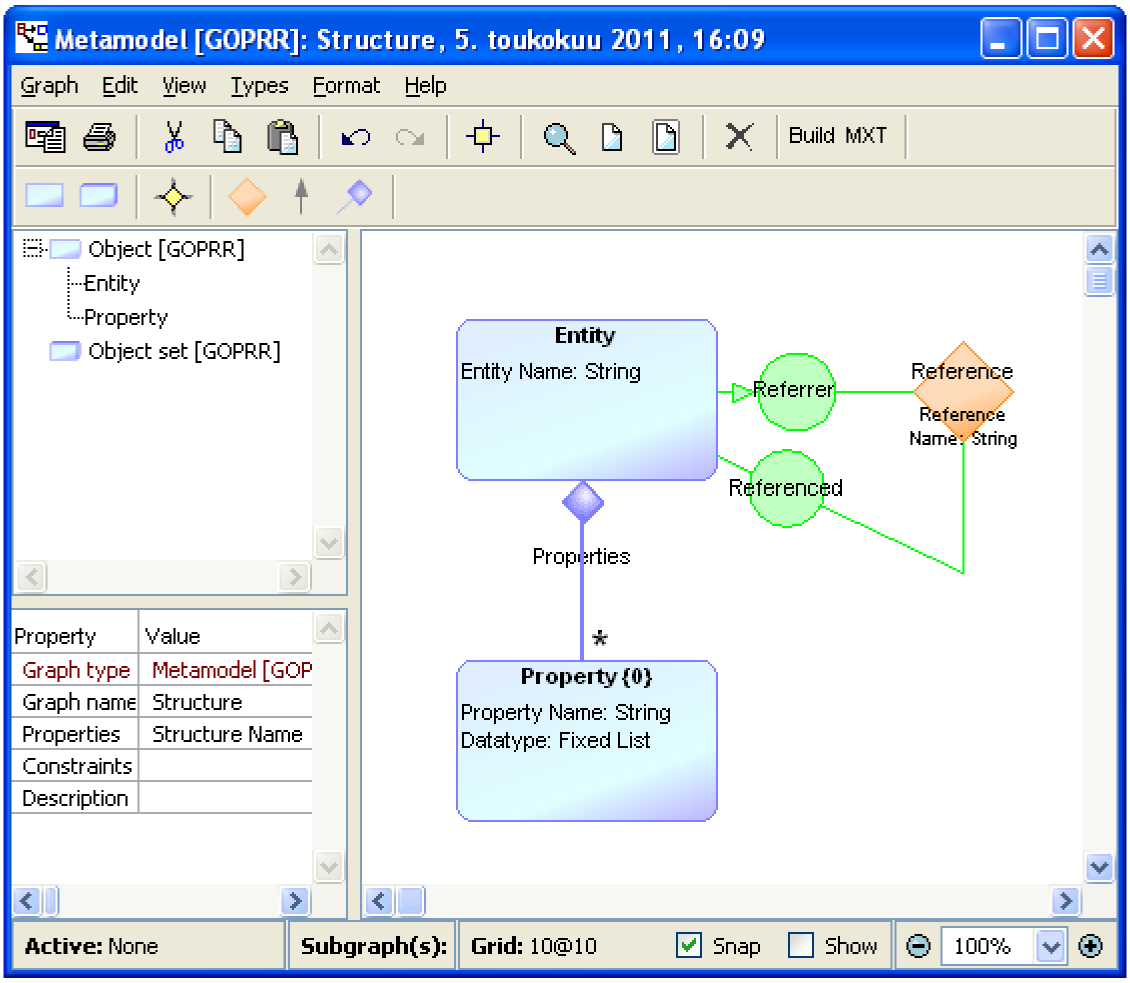
MetaEdit+ uses the GOPRR concepts to talk about models and metamodels:

- A **Graph** is one individual model, often shown as a diagram
- **Objects** are the main elements of graphs, often shown as boxes or circles
- A **Property** is an attribute characterizing another element, often shown as a label
- A **Relationship** connects objects together, often shown as a label over the connection
- A **Role** connects an object into a relationship, often shown as a line and arrow-head

Here we have Person and Car as Entity objects, with a binding connecting them in an ownedCar Reference relationship. Each Entity has a Name property (e.g. 'Person') and a property containing a list of Property objects, e.g. birthdate with Name property 'birthdate'.

The birthdate Property object also has a Datatype property ‘date’ which is not made visible in the diagram – a decision we made about the concrete syntax, about which more later.

We will define this language, which we will call “Structure”, using the graphical GOPRR notation. The diagram shows the basic structure of the language that we want to end up with: An Entity object has an Entity Name property[1], a set of Property subobjects representing primitive values, and Reference relationships to other Entities. The Reference relationship is specified in a *binding*, which connects objects with their roles in the relationship. In this case, each Reference relationship connects an Entity in the Referrer role to an Entity in the Referenced role.



Metamodel [GOPRR]: Structure, 5. toukokuu 2011, 16:09

Property	Value
Graph type	Metamodel [GOP
Graph name	Structure
Properties	Structure Name
Constraints	
Description	

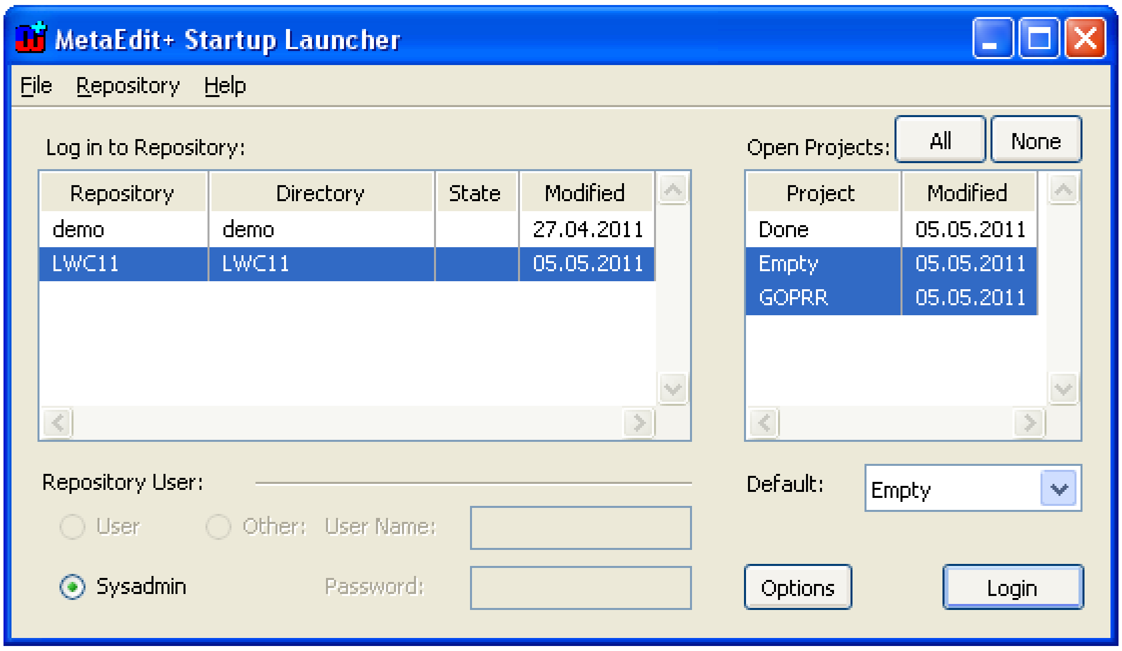
Active: None | Subgraph(s): | Grid: 10@10

Let’s now create that diagram from scratch, in the Empty project.

[1] It’s unfortunate but not uncommon that the terms of the modeling language overlap or clash with those of the language used to define it. We’ll use lower case for GOPRR terms like property or object, and uppercase for types defined in this language like Entity or Property.

Creating the Structure language

Start MetaEdit+, select the LWC11 repository, select Empty and GOPRR projects, check that Empty is chosen as the Default project (where new models will go), and **Login**.



Log in to Repository:

Repository	Directory	State	Modified
demo	demo		27.04.2011
LWC11	LWC11		05.05.2011

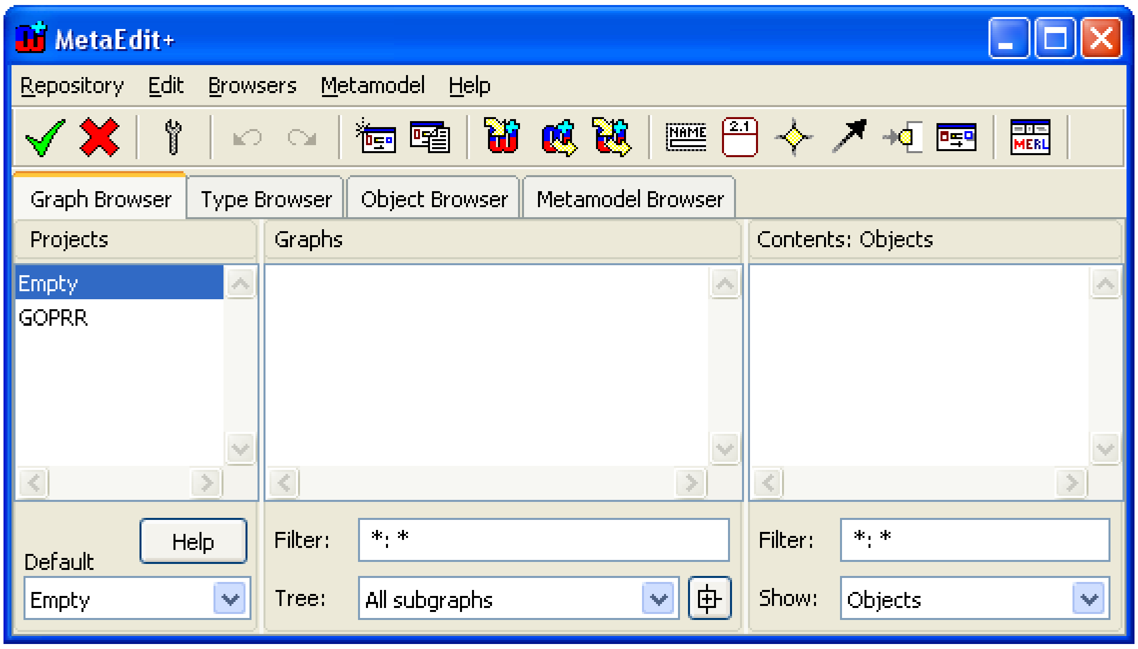
Open Projects:

Project	Modified
Done	05.05.2011
Empty	05.05.2011
GOPRR	05.05.2011

Repository User: ○ User ○ Other: User Name: ● Sysadmin Password:

Default: Empty

You will be prompted for your evaluation code if this is your first login: enter it from your email. The main MetaEdit+ launcher window opens, showing the default Graph Browser tab:



Projects: Empty, GOPRR

Graphs

Contents: Objects

Default: Empty

Filter: *; *

Tree: All subgraphs

Filter: *; *

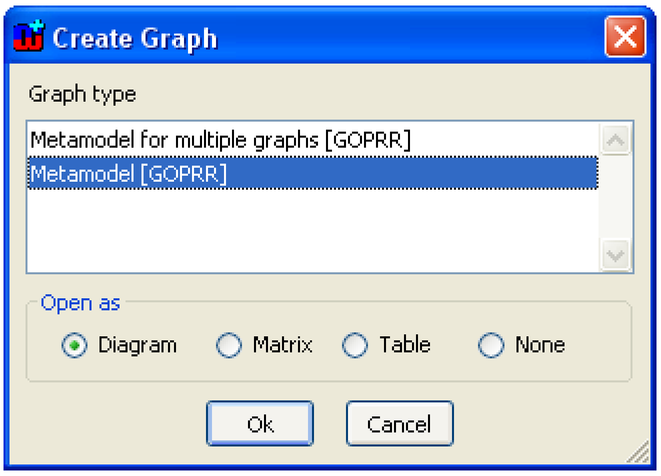
Show: Objects

On the left we can see a list of projects we have open, and the Default project. The middle list shows all the graphs that are in the projects selected on the left. The list on the right shows the objects in the selected graph. Below the lists are wildcard filters and selections for what to show in the list. Above the list are tabs for the other browsers, which are similar to the Graph Browser but use different routes to their data, e.g. whereas the Graph Browser uses containment (projects containing graphs containing objects), the Type Browser uses instantiation (for the selected type and projects, all instances of that type in those projects).

The various lists have their own pop-up menus, and many operations in MetaEdit+ are performed from there. If you can’t see what to do, try opening a pop-up menu for an element or in empty space! Double-clicking is also common: the operations are thus near the cursor.

Choose **Create Graph...** (e.g. from the middle list pop-up menu, the main MetaEdit+ toolbar , the Edit menu, or Ctrl+N). The Create Graph dialog (right) lets you select from the existing graph types, in this case just the two that are part of the graphical GOPRR language. 'Metamodel' is for defining a single graph type, 'Metamodel for multiple graphs' for integrating multiple existing graph types into a coherent interlinked language.

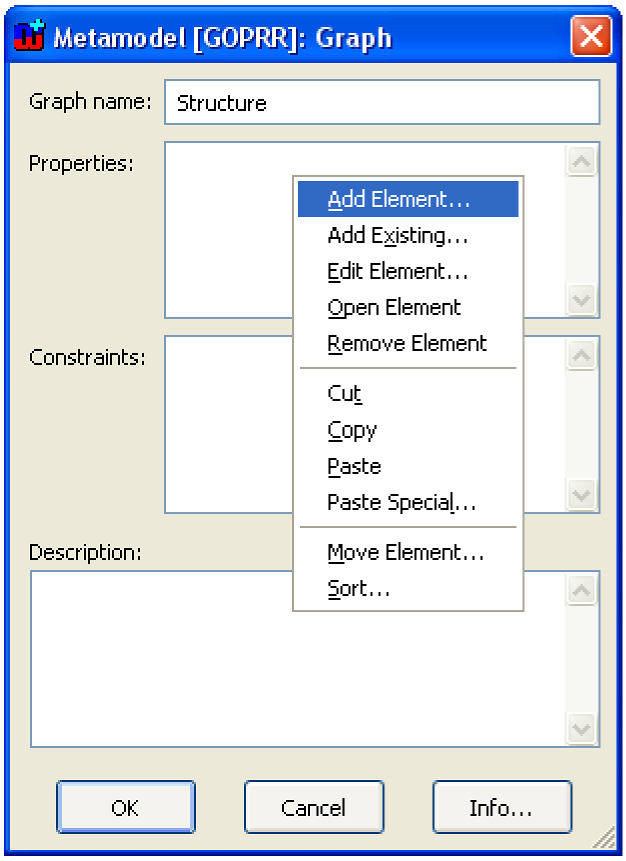
Select the "Metamodel [GOPRR]" graph type, and press OK.



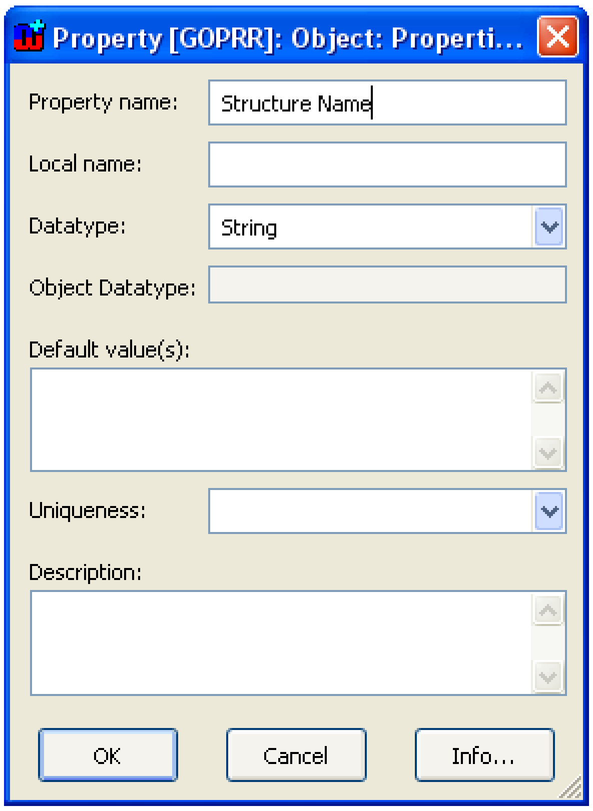
Graph type: Metamodel for multiple graphs [GOPRR]; [✓ Metamodel [GOPRR]]

Open as: [✓ Diagram] Matrix Table None

This will open the property dialog for a new Metamodel graph. Fill in 'Structure' for the Graph name, and in the Properties list open the pop-up menu and choose **Add Element...**. Fill in 'Structure Name' for the Property name, so that each instance Structure graph we make can have a name, and OK both dialogs.



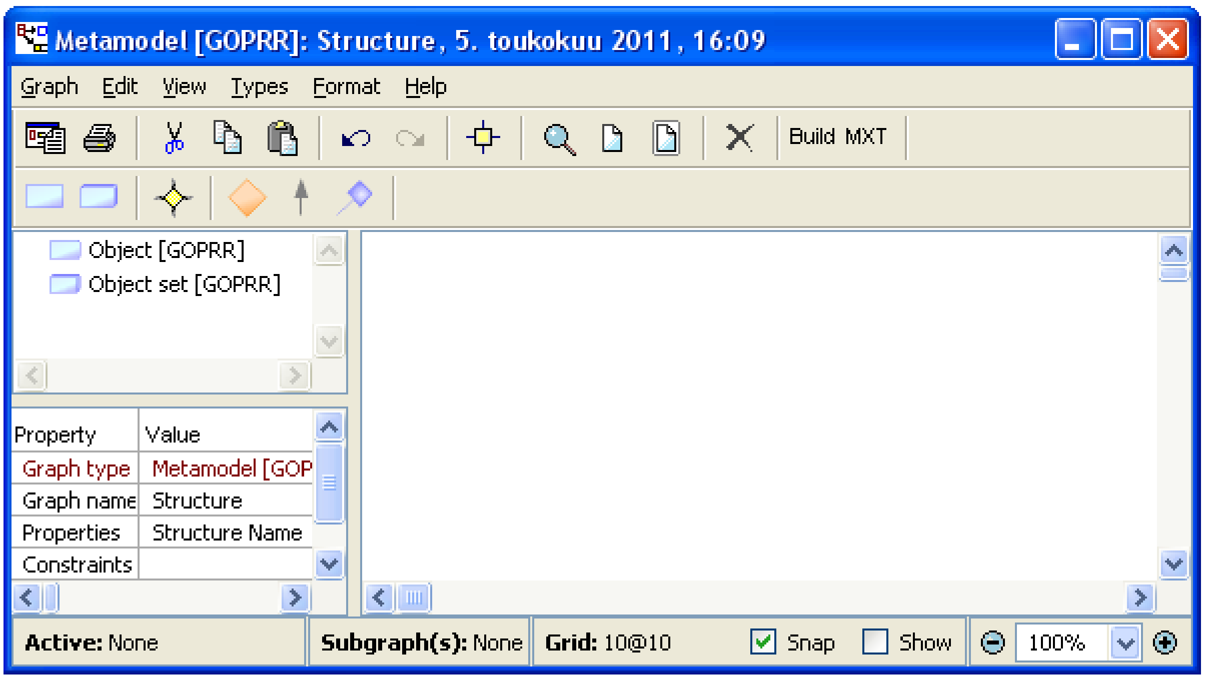
Graph name: Structure



Property name: Structure Name

Datatype: String

A Diagram Editor for the new Metamodel will open:


Metamodel [GOPRR]: Structure, 5. toukokuu 2011, 16:09
Graph Edit View Types Format Help
Build MXT
Object [GOPRR]
Object set [GOPRR]

Property	Value
Graph type	Metamodel [GOP
Graph name	Structure
Properties	Structure Name
Constraints	

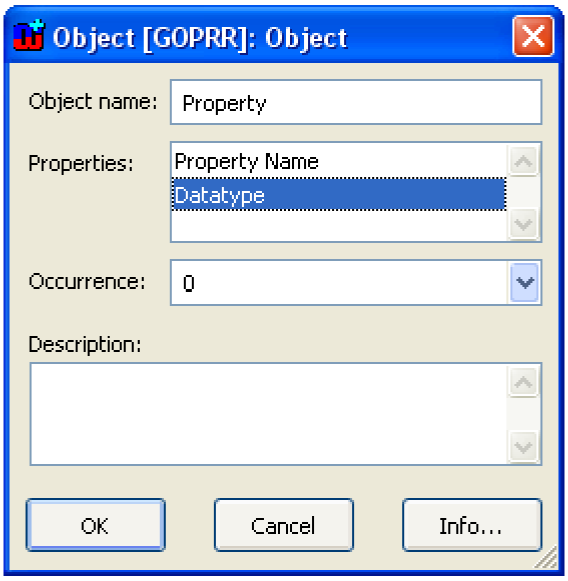
Active: None **Subgraph(s):** None **Grid:** 10@10 ☑ Snap ☐ Show 100%

The top row of the toolbar is the Action Toolbar, with the normal command buttons. The second role is the Type Toolbar, which contains the object and relationship types that we can use in this diagram.

To create our metamodel's first object type, Entity, click the leftmost button on the Type Toolbar, 'Object [GOPRR]'. MetaEdit+ is now in object creation mode, and you can click in the diagram to create a new object at that position. This will open a new property dialog, rather similar to that for the Graph type earlier. Fill in 'Entity' for the Object name, and add a new property 'Entity Name' in the same way as before. Try and be accurate with these names, as our generators will refer to them later.

Click the 'Object [GOPRR]' toolbar button again, and click in the diagram to create the Property object type. Fill in 'Property' for the Object name, give it a property 'Property Name', and then back in the properties list pop-up menu, choose **Add Element…** again. Fill in 'Datatype' as the name of the new element, and this time you will have to make some other choices too. The Datatype is not a freeform string, like the other properties we have defined so far, but can only be chosen from a fixed list of values: string, int or date. Select 'Fixed List' as the Datatype (you may have to scroll in the list) and fill in string, int and date one per line in the Default Values box. OK the dialog for the Datatype property.

The Property objects themselves cannot be directly in the instance models, but only as part of an Entity, so select the Occurrence of the Property object type to be 0. Now you can OK the dialog for Property, and it will appear in the diagram.

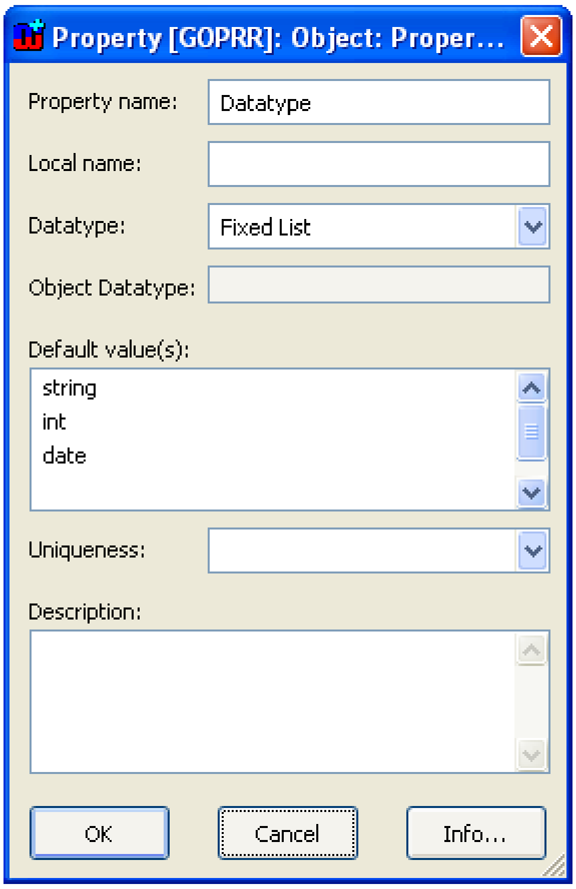

Object [GOPRR]: Object

Object name: Property

Properties: Property Name, Datatype

Occurrence: 0

Description:


Property [GOPRR]: Object: Proper...

Property name: Datatype

Local name:

Datatype: Fixed List

Object Datatype:

Default value(s): string, int, date

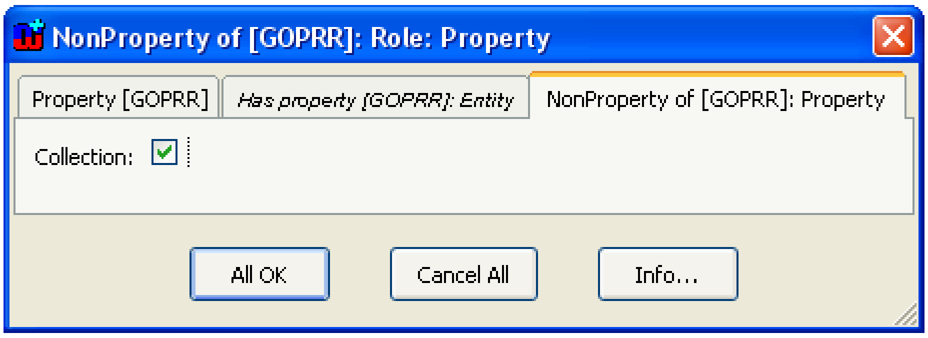
Uniqueness:

Description:

Above: dialog defining Property details

Right: dialog defining Datatype details

Next we need to connect the Entity object type to the Property object type, so that each Entity (e.g. Person) can have many Properties (e.g. name, firstname, birthdate). In the Type Toolbar, click the last button (a blue diamond), click your Entity (you'll see a red crosshair), and click your Property. This creates a relationship from Entity to Property. A dialog will open for extra information about the new relationship. The dialog has three tabs, one for the relationship, one for the first role ('Has property', connected to Entity), and one for the second role ('NonProperty of', connected to Property). In the first tab, fill in 'Properties' as the Local name. In the last tab, select the checkbox to say that we want Entity to have a Collection of Property objects, not just a single Property object:


NonProperty of [GOPRR]: Role: Property

Property [GOPRR]
Has property [GOPRR]: Entity
NonProperty of [GOPRR]: Property

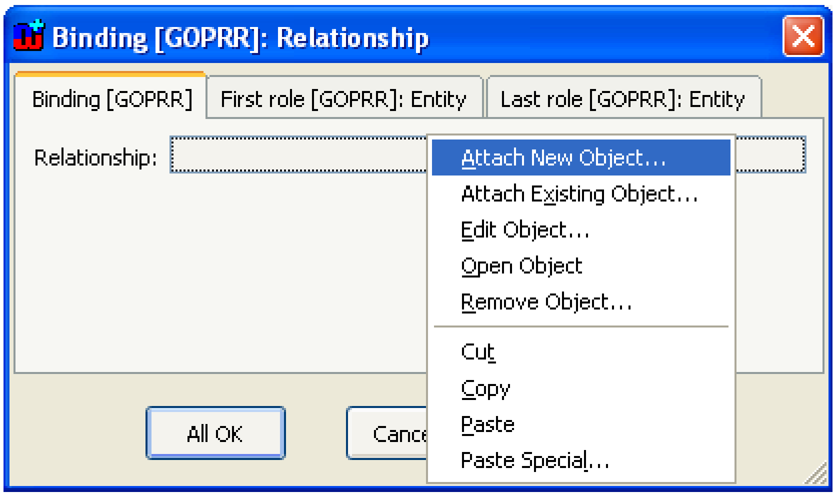
Collection: ✓

All OK
Cancel All
Info...

OK it with the **All OK** button, and the relationship will appear, showing its name and an asterisk. We thus now have defined that Entity has an 'Entity Name' and 'Properties', which is a collection of Property instances, each of which has a 'Property Name' and a 'Datatype'.

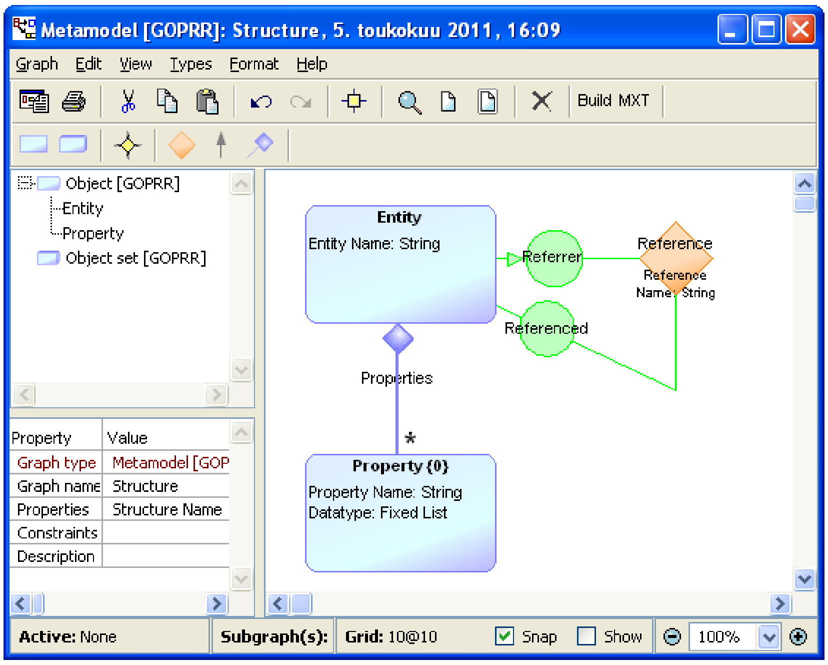
As the last step, we need to say that an Entity can also have a Reference to another Entity. (We could also try making each Entity instance one of the possible Datatypes, but this way is nicer!) We'll define a binding from Entity in a Referrer role, to Entity in a Referenced role, via a Reference relationship that specifies a Reference Name.

In the Type Toolbar, click the orange diamond, 'Binding [GOPRR]'. Click Entity as the start point, then click in space to its right for a position for the Reference relationship, click down from there for an extra point, and then click back in Entity. As you move the cursor you'll notice there is still a line following it: MetaEdit+ allows n-ary relationships with many roles, so it's letting you create more roles if you want. In this case two is enough, so right-click to finalize the binding. (You can also double-click when selecting the last object, to tell MetaEdit+ you don't want any more roles.) Again we get a dialog for more information about this binding:



We have drawn the binding, but now we need to define the Relationship and Role types that take part in it. In a more complicated language, the same Relationship and Role types might be used in many bindings, but here we're creating each for the first time. In the first tab's Relationship box, open the pop-up menu and choose **Attach New Object...**. Name the new relationship type 'Reference', and give it a property 'Reference Name'.

In the next tab, 'First role', choose **Attach New Object...** and name the new role type 'Referrer', but don't make any properties for it. In the last tab, 'Last role', add a role type 'Referenced' (note the spelling!), again with no properties. You can leave the other dialog fields as they are, and press All OK.



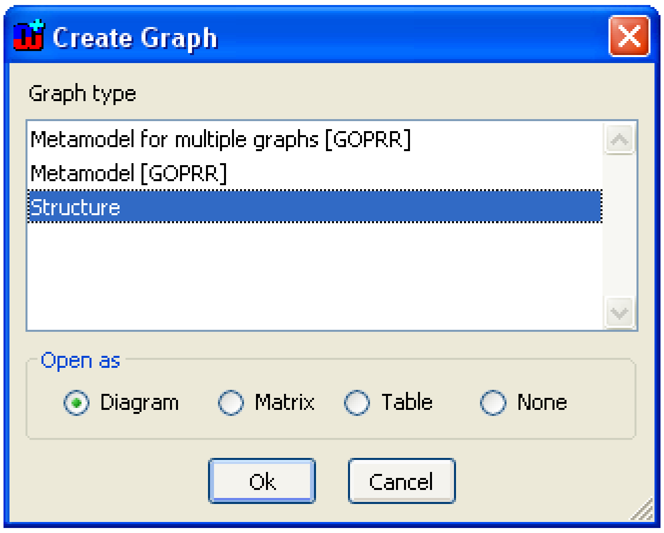
Property	Value
Graph type	Metamodel [GOP
Graph name	Structure
Properties	Structure Name
Constraints	
Description	

That's our language defined! Now would be a good time to press **Commit** on the main MetaEdit+ toolbar or its Repository menu, to save our changes to the repository.

Testing the Structure language and adding symbols

Press the **Build** button at the end of the action toolbar of our Structure Metamodel. This transforms this diagram into a language ready for instantiation (you can close the Generated Files window). In MetaEdit+ terms, this creates a new Graph type called "Structure".

Choose **Create Graph...**, e.g. from the main MetaEdit+ toolbar or Edit menu, and you will see MetaEdit+ now offers Structure as a possible graph type:

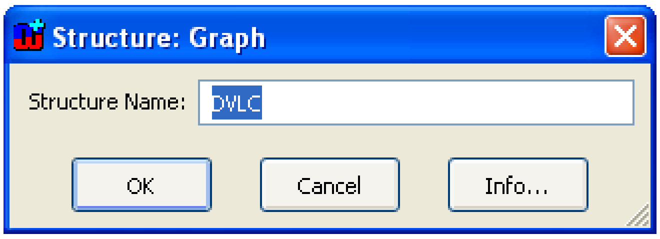

Create Graph

Graph type

Metamodel for multiple graphs [GOPRR]
Metamodel [GOPRR]
Structure

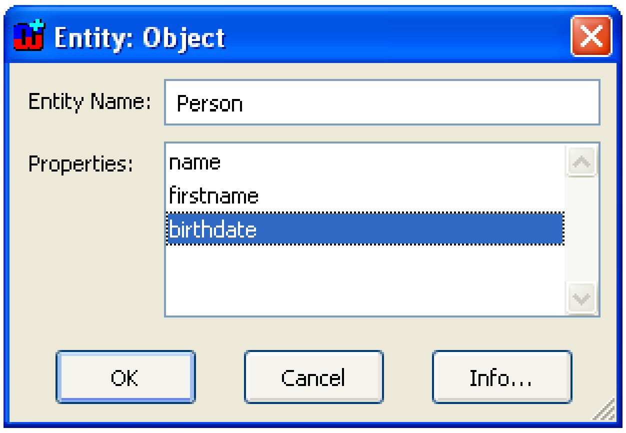
Open as: [✓ Diagram] Matrix Table None

Choose that, and you are asked to give a name for your new Structure graph – let's call it DVLC, which is the Driver and Vehicle Licensing Centre in the UK, and OK the dialog:


Structure: Graph

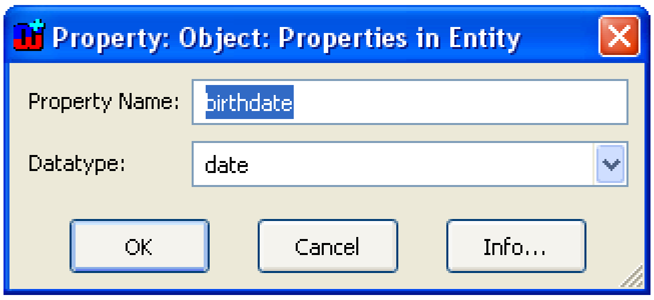
Structure Name: DVLC

In the Diagram Editor that opens, click the 'Entit[y]' button on the types toolbar, and click in the diagram to add the new Entity there. In the property dialog that opens, fill in 'Person' as the Entity Name. To add a new Property to the Properties list, choose **Add Element...** from the pop-up menu in the list widget:


Entity: Object

Entity Name: Person

Properties:
name
firstname
birthdate

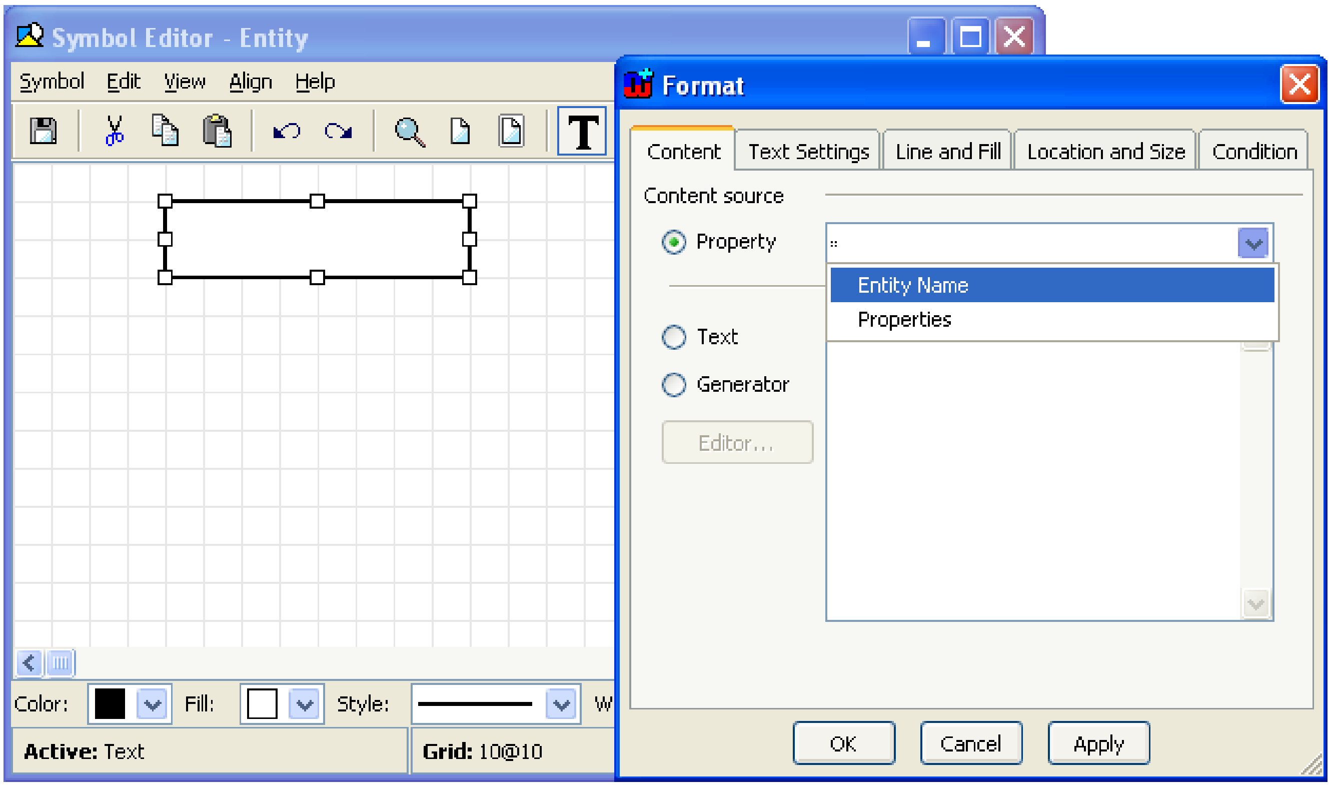

Property: Object: Properties in Entity

Property Name: birthdate

Datatype: date

As you'll see, we don't yet have a symbol defined for Entity, so MetaEdit+ will just show a box with the name 'Person'. Let's add a proper symbol: select the Person object, and shift-double-click the red text in the property sheet row 'Object type: Entity'. (This is the quick and dirty way :-). You can also get there from the main MetaEdit+ window Metamodel Browser tab.)

In the Symbol Editor that opens, first go to the Grid settings in the status bar at the bottom and turn on Snap and Show. Click the T[ext] button in the toolbar and drag to create a box for the Entity Name, say 8 squares wide by 2 high. In the Format dialog that opens, set the Content source to Property and choose Entity Name from its list. (A text element can show a fixed text, the value of a property, or the result of running a generator snippet for this object in this graph.)



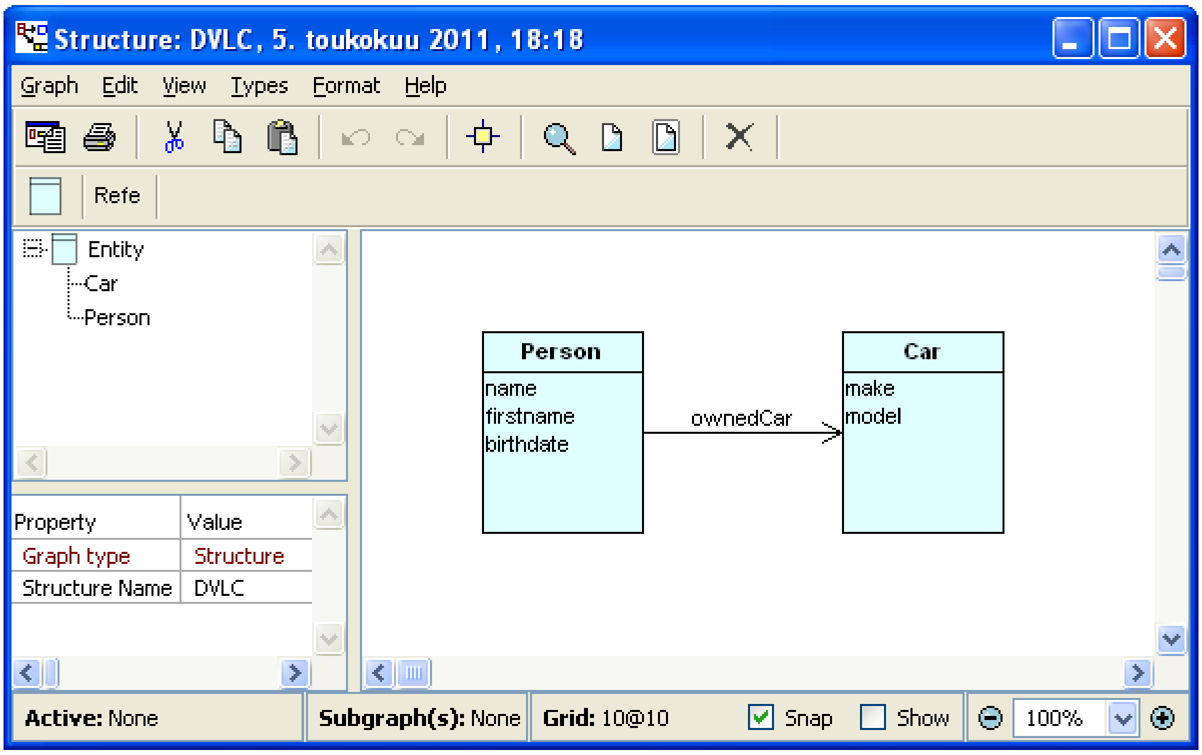
On the Text Settings tab, set the Font style to Bold and the Alignment to centered, and OK the Format dialog. Create another Text box to sit underneath this one, maybe 8x8 squares, and set its Content source to Property: Properties (the default Text Settings are fine for this). Select both text elements by shift-clicking, and from the status bar set their Fill color to the lightest blue.

As you can see, the Symbol Editor follows standard behavior of vector graphics editors, adding features specific to modeling language design. You can use Tab to cycle the selection through all the elements, and move the selected element with cursor keys (moving by grid increments, or by single pixels with shift down). You can import bitmaps and SVG, and save symbols to the MetaEdit+ symbol library or SVG files. With the Condition tab of the Format dialog, symbol elements can be made conditional, only appearing for certain values of their properties or even of a generator reading information from anywhere in the model.

Save the symbol from the toolbar, and you'll see that MetaEdit+ automatically adds a connectable: a red polygon and crosshair that together determine where lines connecting to this object point to, and where they stop at the edge of the object. Close the Symbol Editor, and you will see that your Person object now looks a lot better!

Since the toolbar still shows just the name Entit[y], let's close our DVLC diagram. This also lets us practise re-opening it by double-clicking 'DVLC: Structure' in the Graphs list of the main MetaEdit+ window. (If you don't see it, check you are in the Graph Browser tab, and that Empty is selected in the Projects list.)

Now you can finish filling in the empty Structure diagram with the Entities we want for DVLC: “Person” and “Car”, their properties and the relationship between them.



Property	Value
Graph type	Structure
Structure Name	DVLC

To create the Reference relationship, click ‘Refe’ on the type toolbar, click Person, and click Car. You will be prompted for the Reference Name: fill in ‘ownedCar’. As you’ll see, there’s no symbol for the relationship or roles yet.

Select the relationship by clicking the red dot at its center, and shift-double-click the red text in the property sheet to edit its symbol. Draw an 8x3 Text element for the Reference Name, and set its Alignment to top-centered, and Color and Fill to Transparent. Select **View | Choose Grid…** to change the grid to 5x5. Click the Point Connectable button at the end of the toolbar (the red crosshair), and click right in the centre of the text box. The connectable determines how the role lines connect: we want them to meet in the middle, under the first line of text. Close the Symbol Editor and Save, and you’ll see the ‘ownedCar’ text is now shown.

Select the Referenced role by clicking on the role line near the Car end. Shift-double-click the red text in the property sheet to edit the role symbol. The Symbol Editor displays an example role line, ending on the right. Use the straight Line tool from the toolbar to draw an arrow: click on the end point, move two squares left and one up and click again; repeat, creating a new Line that goes left and down. Close the Symbol Editor and Save, and you’ll see the arrow so you can tell which way your References go.

There you go! You’ve now learned the basics of MetaEdit+ metamodeling and modeling: the GOPRR concepts, the graphical GOPRR metamodeling language, the main MetaEdit+ window and Graph Browser, the Diagram Editor, the Symbol Editor, and the standard windows like property dialogs. Oh, and you also created the language and example model of Task 0.1, but that was hopefully the easy bit.

Now would again be a good time to **Commit** from the main MetaEdit+ window.

0.2 Code generation to GPL such as Java, C#, C++ or XML

- Generate Java Beans (or some equivalent data structure in C#, C++ etc.) with setters, getters and fields for the properties.

A generator is defined for a particular Graph type, and navigates through the contents of a graph of that type, outputting values from the model along with fixed text boilerplate, punctuation etc. Generators are written in MERL, the MetaEdit+ Reporting Language. Double-click 'DVLC: Structure' in the main MetaEdit+ window to open it in a Diagram Editor, and open the Generator Editor with **Graph | Edit Generators…**. Let's first create a new generator called Java: choose **New** from the toolbar (or Generator | New…, or press Ctrl+N) and enter Java as the name, and the header and footer will be inserted for you:

```
Report 'Java'

endreport
```

The cursor is on the blank line, ready for you to enter the body of the generator. MERL commands are written literally, and fixed text is quoted. Enter this for the body: (you can just type it, but if you like you can try using the top-center list's Control Templates and Object lists: double-click in the top-right list to enter a command or a type name from the metamodel.)

```
foreach .Entity
{  subreport '_JavaFile' run
}
```

We first iterate through each Entity, calling a subgenerator called '_JavaFile' for each. The subreport…run command follows a pattern common in MERL: the statements between the two keywords produce the output which forms the argument for the command. In this case there is just a single statement, a fixed string '_JavaFile', but any number of statements can be included in more complex cases, as we shall see.

Save the Java generator, and it will appear in the hierarchical list of generators with a '+' to show it has a subgenerator. A quick way to define the subgenerator is to click open the '+': you will see the _JavaFile subgenerator listed in red and prefixed with '?' to show it is not yet defined. Select it and press Ctrl+N to create it. Add this as the body of the _JavaFile generator:

```
filename id '.java' write
   subreport '_Java' run
close
```

The filename…write…close command directs output to a file. It has two arguments, the file name and the file contents. The file name is simply **id** plus '.java' for an extension. **id** is always the name of the current element to which the generator has navigated, in this case the name of an Entity. We will thus end up with multiple output files for a single Structure graph: one Java file for each Entity. The file contents for each file are formed by the subgenerator called _Java, which we can start defining as above: Save the _JavaFile generator, open it in the hierarchy, select its subgenerator _Java and press Ctrl+N. The body starts with the class and attribute definitions, which you can paste in – pressing Ctrl+E will check and highlight the syntax:

```
'public class ' :Entity Name; ' {' newline
do :Properties
{  $datatype = :Datatype
   subreport '_JavaField' run
}
do ~Referrer>Reference
{  $datatype = ~Referenced.Entity:Entity Name;
   subreport '_JavaField' run
}
```

We want to generate Java field definitions for each Property and each referenced Entity, so we make a generic _JavaField subgenerator. Since the datatype will be found from different places in the two cases, we get it here and set it in a variable called $datatype. Note the way we can use navigation in the second do loop: first from the Entity to its Referrer role and on to the Reference relationship, then within the loop on to the Referenced role, the Entity it attaches to, and its Entity Name. Having a concise syntax for navigation plays an important role in the effectiveness of generator languages. There are only four type prefix characters to learn:

```
:  property
.  object
~  role
>  relationship
```

We can save _Java for now, and define its subgenerator _JavaField: the body is simple:

```
'  private ' $datatype ' ' id ';' newline
```

Go back to the _Java generator (if the tree collapsed, click open the top-level Java → _JavaFile → _Java, or just Java then press Num Pad *, or use the Alphabetical view). We add the constructor: note how we can have line breaks within the fixed strings:

```
'
   public ' :Entity Name; '() {
   }

'
```

After that we generate the accessors, again for both Properties and References, before closing the final brace:

```
do :Properties
{  $datatype = :Datatype
   subreport '_JavaAccessors' run
}
do ~Referrer>Reference
{  $datatype = ~Referenced.Entity:Entity Name;
   subreport '_JavaAccessors' run
}

'}'
```

Save, then define the _JavaAccessors generator, which is mostly boilerplate:

```
'  public ' $datatype ' get_' id '() {
       return ' id ';
   }

   public void set_' id '(' $datatype ' value) {
       this.' id ' = value;
   }

'
```

You can now select the top-level Java generator and run it with **Generate**, or from any Structure graph with the **Generate** toolbar button. Running from a Generator Editor shows an extra warning for the benefit of the generator developer: you can ignore it with 'Yes to All'.

Extra Task 3.1 adds CamelCase for getFoo() rather than get_foo().

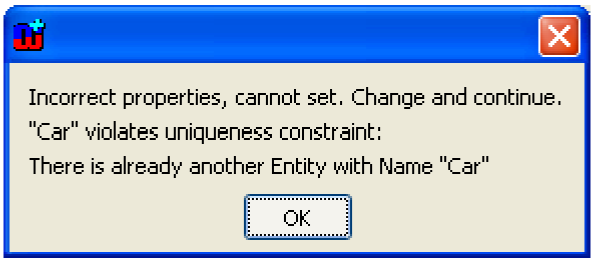
Extra Task 3.2 adds a toolbar button for running the Java generator from a diagram.

Extra Task 3.5 shows how to use the Generator Debugger

0.3 Simple constraint checks such as name-uniqueness.

- For example, to stop there being two Entities called "Car", or to stop there being two Properties called "make" in "Car".

In the Structure metamodel, double-click Entity and in its property dialog double-click Entity Name. Set its Uniqueness to "globally", OK the dialogs, and press Build in the toolbar. Now if you try to add a second Entity also called "Car" in the DVLC Structure diagram, you will be prevented with a dialog explaining the problem:


Incorrect properties, cannot set. Change and continue.
"Car" violates uniqueness constraint:
There is already another Entity with Name "Car"

This rule is a hard constraint, checked and enforced when creating a model element. You can also make soft constraints, e.g. if you prefer to let the user create a second Car temporarily, then delete the first when he has finished e.g. copying values from the old one. Soft constraints are generally made with generators in the symbols, so the symbols show a visual indication if the constraint is broken.

Edit the Entity symbol, e.g. from the Metamodel Browser or (faster) in the DVLC Structure diagram by selecting the "Car" Entity and shift-double-clicking the red "Entity" text in the property sheet on the left. Double-click the Text element for its Properties, change the Content source to Generator, and paste the following as the generator:

```
dowhile :Properties
{  $times = '0'
   do :Properties;1 where id = id;1
   {   $times = $++times
   }
   if $times > '1' num then '*' id '*' else id endif
   newline
}
```

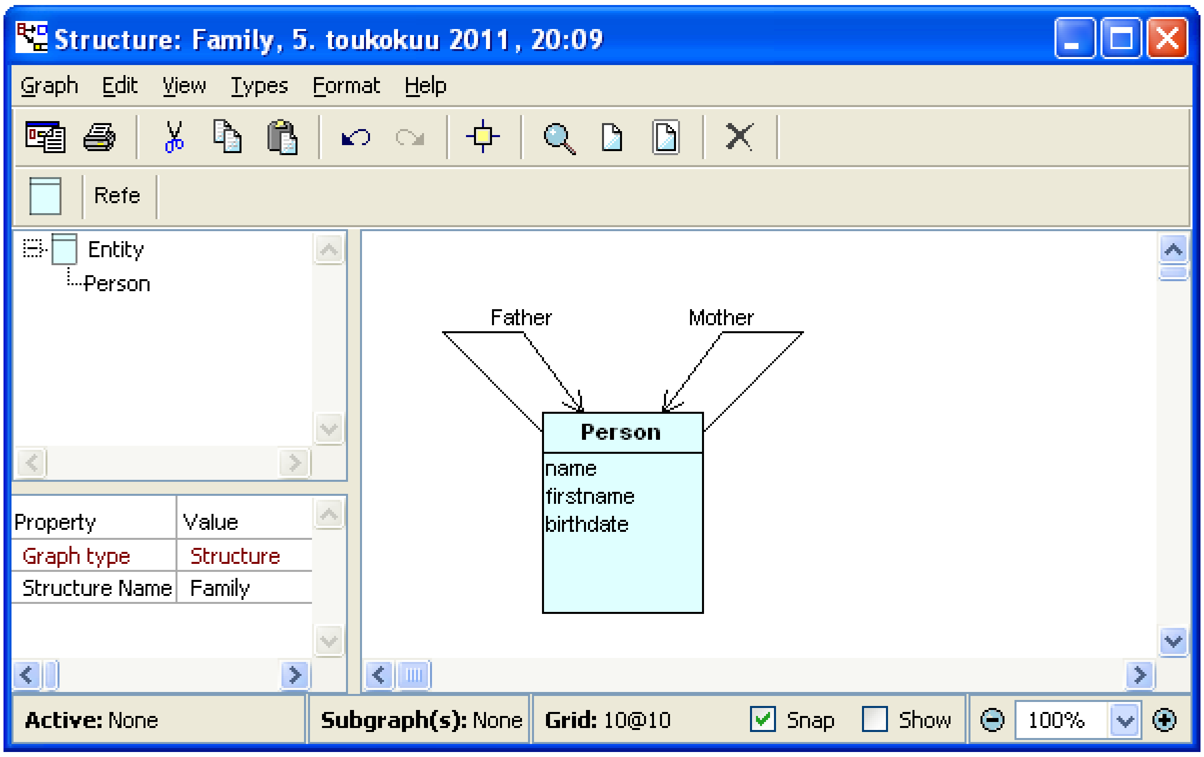
If you want you can press the **Editor…** button to add some formatting, e.g. **Format | Color…** to make the asterisks and id red, as here. OK the dialog and **Save** the symbol, then back in the DVLC structure diagram try adding a second "make" property in "Car": it will be allowed, but will be highlighted in the diagram as an error *like this*.

This generator shows something of how the stack of navigated model elements is used in MERL. The third line shows how to get information from different levels of the stack: **;1** means "evaluate this statement one stack level further out". At the start of the third line, the stack contains the original Structure graph, the Entity, and the Property we are currently iterating over. The third line thus does not want to ask the current element, a Property, for Properties (it has none), but rather to use "do :Properties**;1**" to ask the Entity one level out: we're going to iterate over the same set of Properties again, so we can spot the duplicates for the current Property. Within the "where" constraint we are one level deeper, the same as within the body of the inner do loop: the current element is a Property, as is the element one level out. The third line thus means "iterate over Properties of the Entity, where the Property's id is the same as the id of the current property".

0.4 Show how to break down a (large) model into several parts, while still cross-referencing between the parts.

- For example, put the Car and Person entities into different files or diagrams, while still having the Person -> Car reference work.

Simply create a new Structure graph, e.g. Family. You can copy and paste Person into there, and both graphs will use the same Person, and any existing References will work fine. The Family graph could have two References, Father and Mother[2].


Structure: Family, 5. toukokuu 2011, 20:09
Graph Edit View Types Format Help
Refe
Entity
Person
Father
Mother
Person
name
firstname
birthdate

Property	Value
Graph type	Structure
Structure Name	Family

Active: None | Subgraph(s): None | Grid: 10@10 | ✓ Snap | Show | 100%

This is an important feature of MetaEdit+: we can reuse an object in another model, without the 'baggage' of its relationships from the initial model. You can also trace to see all the places where a model element like Person exists by choosing **Info…** from the element's pop-up menu.

The cross-referencing even works on the instance level, as you will be able to see once you have completed Task 1.1: if you Build your Family Structure graph, you can instantiate it and copy and paste your existing "Markus Voelter" Person into the new Family graph.

[2] This does however reveal the simplicity of the Structure language as specified by the LWC Task requirements: we'd really need to add cardinality to the Structure language, so that when we define Family we'd have a way to prevent multiple Fathers etc.

Phase 1 - Advanced

This phase demonstrates advanced features not necessarily available to the same extent in every LWB.

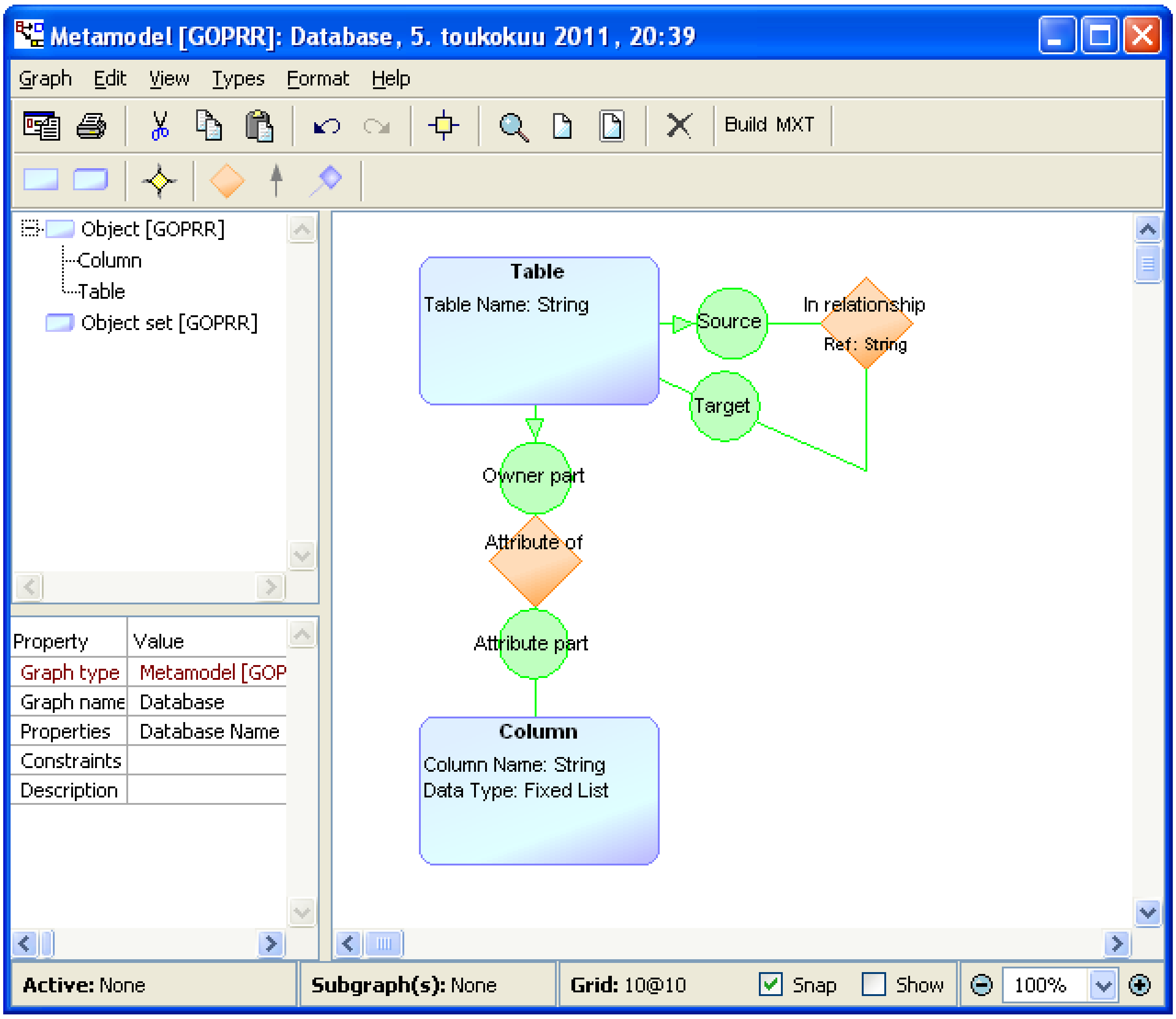
We will handle Task 1.3 first, as doing that will help in understanding the other tasks.

1.3 Show how to do a model-to-model transformation.

- Define an ER-meta model (Database, Table, Column) and transform the entity model into an instance of this ER meta model.

Create a new graph of type "Metamodel [GOPRR]", calling it "Database" and giving it a new Property, "Database name". Add the following content to the diagram, being careful to use exactly the type names specified here – the model-to-model generator we will create will use the same names. We'll make sure there's no cheating by not reusing anything from the earlier metamodel, e.g. rather than reusing the existing 'Datatype' we'll make a new property type 'Data Type' with a similar Fixed List of string/int/date. The relationship and role types in the Bindings have no properties, except for Ref in the top 'In relationship'.

When you have made the model, **Commit** and press the **Build** button on the toolbar to turn it into an instantiable graph type.



Property	Value
Graph type	Metamodel [GOP
Graph name	Database
Properties	Database Name
Constraints	
Description	

MetaEdit+ doesn't add a separate language for model-to-model translations, using instead the existing MERL and XML import facilities. We thus write a MERL generator that outputs the XML representing the desired model, and imports that XML. Select the DVLC: Structure graph in the main MetaEdit+ window and press the MERL button at the end of the toolbar to open a Generator Editor for Structure graphs. Create a new generator called ER, with the following body:

```
filename id '.mxm' encoding 'utf-8' write
   '<?xml version="1.0" encoding="UTF-8"?>'
   subreport '_ER' run
close

internal 'fileInPatch: "' id '.mxm"' execute
```

In the filename…write…close command we use an optional parameter, encoding, to write the file as UTF-8. The body of the XML file will be created by the _ER subgenerator. When the foo.mxm MetaEdit+ XML model file has been written, the generator will import it. It uses the internal…execute command to call a command-line parameter: fileInPatch: "foo.mxm".

The _ER generator starts by defining the standard translators, and creating the graph XML:

```
subreport '_translators' run
'
<gxl xmlns="http://www.metacase.com/gxlGOPRR">
 <graph type="Database">
  <slot><value><string>' id%xml '</string></value></slot>
'
```

As is common with XML, most of the generator is boilerplate. Inside the graph we create a Table for each Entity, and a Column for each Property of each Entity. The type names in the red boilerplate match those we defined in the target Database metamodel, whereas the type names in green are those from the source Structure metamodel:

```
foreach .Entity
orderby y num, x num
{  '  <object id="_' oid '" type="Table">
'  '   <slot><value><string>' id%xml '</string></value></slot>
'  '  </object>
'  do :Properties
   {   '  <object id="_' oid '" type="Column">
'      '   <slot><value><string>' id%xml '</string></value></slot>
'      '   <slot><value><string>' :Datatype%xml '</string></value></slot>
'      '  </object>
'  }
}
```

We sort the Entities from top to bottom and left to right, i.e. by their y co-ordinate and x co-ordinate as numbers (default sorting would be as strings, with the familiar problem that '100' < '20'). The optional orderby can be used with foreach and do loops, and can take any number of parameters separated by commas.

The oid command outputs for the current element an identifier unique in this repository. We need unique identifiers later in the XML, to refer back to these objects when creating relationships between them. The %xml translator escapes any reserved characters from the model into legal XML. The single quotes and tabs at the start of lines allow us to have indentation for MERL code separately from indentation in output code – I find this little extra effort makes reading and working with the generators easier, but of course you are welcome to ignore all line breaks and indentation in both XML and MERL.

Next we iterate over Entities and their Properties again, this time to create the 'Attribute of' relationships. Each relationship is in a binding with one connection via an 'Owner part' role to the Table (made from the Entity, one level out in the stack), and another connection via an ‘Attribute part’ role to the Column (made from the Property):

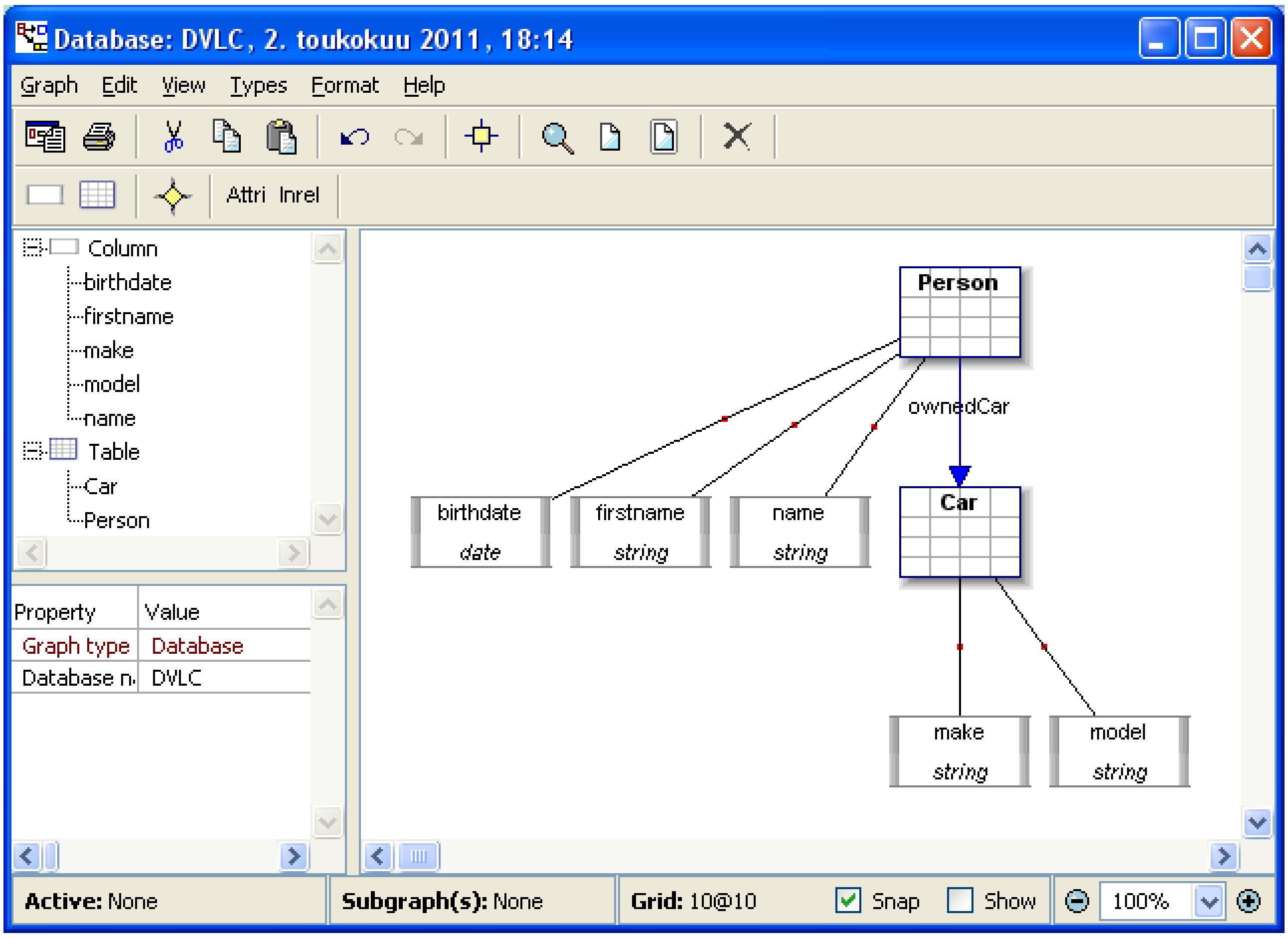
```
foreach .Entity
orderby y num, x num
{  do :Properties
   {   '   <binding>
'      '    <relationship type="Attribute of" />
'      '    <connection>
'      '     <role type="Owner part" />
'      '     <object href="#_'  oid;1 '" />
'      '    </connection>
'      '    <connection>
'      '     <role type="Attribute part" />
'      '     <object href="#_' oid '" />
'      '    </connection>
'      '   </binding>
'  }
}
```

Finally we create the bindings for the Reference relationships, and close the outermost tags:

```
foreach >Reference
{  '   <binding>
'  '    <relationship type="In relationship">
'  '     <slot><value><string>' id%xml '</string></value></slot>
'  '    </relationship>
'  '    <connection>
'  '     <role type="Source" />
'  '     <object href="#_' do ~Referrer.() { oid } '" />
'  '    </connection>
'  '    <connection>
'  '     <role type="Target" />
'  '     <object href="#_' do ~Referenced.() { oid } '" />
'  '    </connection>
'  '   </binding>
'
}
' </graph>
</gxl>'
```

Since we know the Referrer role can only be connected to an Entity, we can omit its name and just use parentheses to allow any object type. The parentheses can also be used to list several acceptable types, e.g. “do ~Referrer.(Entity | Foobar)”.

Running the ER generator on our DVLC Structure graph will create a new DVLC Database graph, which you can open by double-clicking it in the main MetaEdit+ window. Our XML only included the conceptual information, not any representational information, so MetaEdit+ will prompt you to create a new Diagram. In the new diagram we can get a sensible layout by choosing **Graph | Layout...**, pressing the **All** button at the top, and OK.



Property	Value
Graph type	Database
Database n.	DVLC

Active: None **Subgraph(s):** None **Grid:** 10@10 ☑ Snap ☐ Show 100%

To define your own symbols for the types, select an object of that type and shift-double-click the red name of the type in the property sheet on the left. You can use **Symbol | Browse Library...** to see some existing symbols that you can reuse. Note that as the library is metamodel-independent, you will have to edit any property Text elements to set their Content source to the correct property.

1.1 Show the integration of several languages.

- Define a second language to define instances of the entities, including assignment of values to the properties. This should ideally really be a second language that integrates with the first one, not just "more syntax" in the same grammar. We want to showcase language modularity and composition here.

```
Person p = {
   name = "Voelter"
   firstname = "Markus"
   birthdate = 14.02.1927
   ownedCar = c
}

Car c = {
   make = "VW"
   model = "Touran"
}
```

We could define a second language using the normal language workbench facilities of MetaEdit+, as we did for the first language. However, this language is clearly the instantiation of the first language, so it would be nicer if we could have our existing DVLC Structure graph automatically create it for us: all the information is there already. Knowing that MetaEdit+ has an XML format for metamodels which is similar to that for models, we can use a similar approach to the second half of Task 1.3. We write a generator for Structure graphs like the ER generator, but instead of generating a .mxm model XML file, we generate a .mxt metamodel XML file. In fact, this mirrors the existing "Build" generator in the graphical GOPRR Metamodel, which itself is just a normal MetaEdit+ generator in a normal MetaEdit+ graph type. In MetaEdit+, you can use any language you like to define your metamodels!

Obviously, this approach is slower if we just want to do Task 1.1 – it would be faster to quickly draw the graphical GOPRR model with Person and Car. Still, it seems fair to assume users would draw more than one Structure model, so let's follow our heart and do it right. Create a new generator for Structure graphs called Build, similar to Task 1.3 except now we're now dealing with .mxt files:

```
filename id '.mxt' encoding 'utf-8' write
   '<?xml version="1.0" encoding="UTF-8"?>'
   subreport '_Build' run
close

internal 'fileInPatch: "' id '.mxt"' execute
```

Its _Build subgenerator starts by defining some translators, which we'll put in a subgenerator:

```
subreport '_StructureTranslators' run
```

Save that for now, and create its _StructureTranslators subgenerator. This defines the usual generic translators, and adds two of its own. The first maps Structure data type names into MetaEdit+ data type names:

```
subreport '_translators' run

to '%datatype
$string $String
$int $Number
$date $String
' endto
```

The second maps Structure data type names into regular expressions that check input of that data type:

```
to '%regex
$string $.\*
$int $0|([1\-9][0\-9]\*)
$date $[0\-3]?[0\-9]\\.[0\-1]?[0\-9]\\.[1\-2][0\-9][0\-9][0\-9]
' endto
```

Save that, and return to the _Build generator, where we continue after the call to _StructureTranslators. The XML structure is broadly similar to that of .mxm files, except that <graph> and <property> here mean graph *type* and property *type*:

```
'
<gxl
  xmlns="http://www.metacase.com/gxlGOPRRType"
  xmlns:sym="http://www.metacase.com/symbol">
 <graph typeName="' id%xml '">
  <slot name="Name">
   <property typeName="Graph name">
    <dataType>
     <simpleType>String</simpleType>
    </dataType>
   </property>
  </slot>
'
```

For each Entity, we generate an object type, and for each of its Properties, we generate a property slot containing a new property type:

```
foreach .Entity
orderby y num, x num
{  '  <object id="_' oid '" typeName="' id%xml '">
'  do :Properties
   {   '   <slot id="s_' oid '" name="' id%xml '">
'      '    <property typeName="' id%xml '">
'      '     <dataType>
'      '      <simpleType>' :Datatype%datatype '</simpleType>
'      '     </dataType>
'      '     <regex>' :Datatype%regex '</regex>
'      '    </property>
'      '   </slot>
'  }
   '  </object>
'
}
```

For each Reference, we generate a new relationship type. Let's also add a symbol for that relationship type, so it can show its name over its line. Symbols are defined as standard SVG graphics, with a few additions needed for use in a modeling tool.

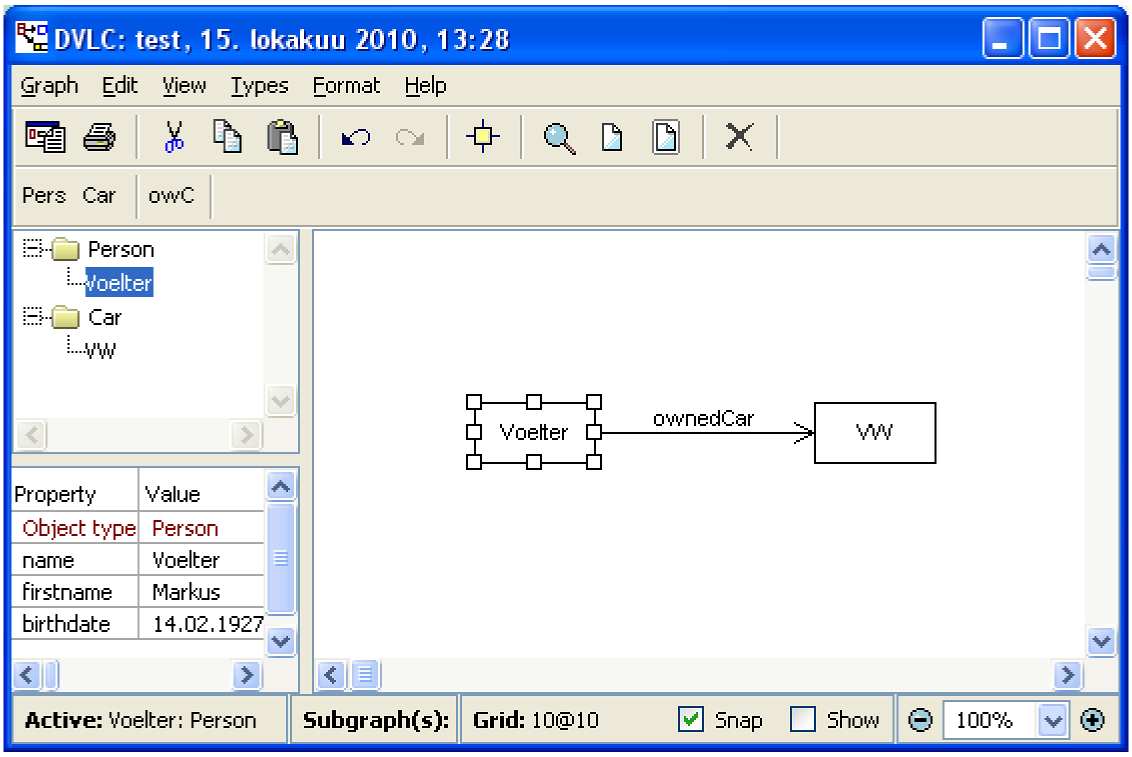
```
foreach >Reference
{  '  <relationship id="_' oid '" typeName="' id%xml '">
          <relationshipSymbol offset="60,40"
xmlns="http://www.metacase.com/symbol">
              <defaultConnectable targetPointX="100" targetPointY="55" />
              <svg height="70" width="140" xmlns="http://www.w3.org/2000/svg">
                  <textArea font-size="14" display-align="before" text-
anchor="middle" x="60" y="40" height="30" width="80">
                      ReportTextSource:id
                      <metaInfo xmlns="http://www.metacase.com/symbol" />
                  </textArea>
              </svg>
          </relationshipSymbol>
     </relationship>
'
}
```

Finally we create a pair of simple role types, generate the bindings for each Reference relationship, and close the outer tags:

```
'  <role id="From" typeName="From" />
  <role id="To" typeName="To" />
'
foreach >Reference
{  '  <binding>
'  '   <relationship href="#_' oid '" />
'  '    <connection>
'  '     <role href="#From" />
'  '     <object href="#_' do ~Referrer.() { oid } '" />
'  '    </connection>
'  '    <connection>
'  '     <role href="#To" />
'  '     <object href="#_' do ~Referenced.() { oid } '" />
'  '    </connection>
'  '   </binding>
'
}
' </graph>
</gxl>'
```

With that groundwork done, we can now turn any Structure model into an instantiable graph type by simply running its Build generator. Let's do that now: select the top-level Build generator, choose **Generator | Generate…** and select the DVLC Structure graph.

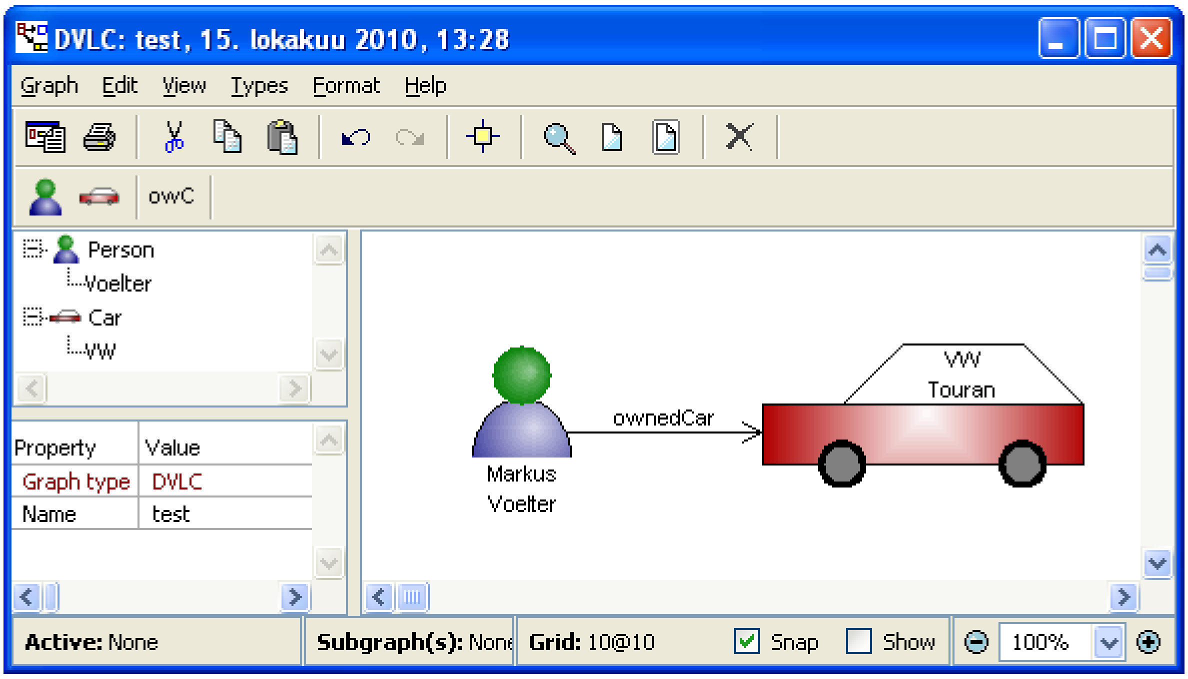
We can then create a new graph of this new type, DVLC, called Test, where we can add Markus and his car, as specified in the task.


DVLC: test, 15. lokakuu 2010, 13:28
Graph Edit View Types Format Help
Pers Car owC
Person
Voelter
Car
VW
Voelter
ownedCar
VW

Property	Value
Object type	Person
name	Voelter
firstname	Markus
birthdate	14.02.1927

Active: Voelter: Person **Subgraph(s):** **Grid:** 10@10 ✓ Snap ☐ Show 100%

If that felt too easy, go ahead and show you're a better graphics designer than I am, by adding symbols for Car and Person. OK, that's not a very hard task either… :-)


DVLC: test, 15. lokakuu 2010, 13:28
Graph Edit View Types Format Help
owC
Person
Voelter
Car
VW
VW
Touran
ownedCar
Markus
Voelter

Property	Value
Graph type	DVLC
Name	test

Active: None **Subgraph(s):** None **Grid:** 10@10 ✓ Snap ☐ Show 100%

1.2 Demonstrate how to implement runtime type systems.

- The initialization values in the instance-DSL must be of the same type as the types of the properties.

This was already handled by the solution to Task 1.1, in the data types and regular expressions in _StructureTranslators. You can see it when entering the birthdate: entering a random string or even 99.99.1999 will be refused. (And no, this isn't meant to be a full implementation of date checking: to keep our regex simple for this example, we only check the range of each digit, so 39.19.2999 is accepted. Better date regexes can be found online; leap centuries are left as an exercise for the reader!).

1.4 Some kind of visibility/namespaces/scoping for references.

- Integrate namespaces/packages into the entity DSL

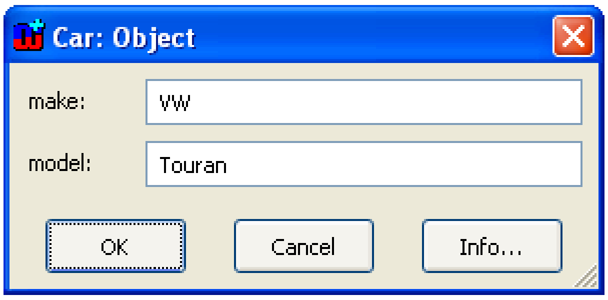
```
package p1 {
      entity Person  {
          string name
          string firstname
          date birthdate
          Car ownedCar
      }
}

package p2 {
      import p1
      entity Car {
         string make
         string model
      }
}
```

and make sure in the instance DSL you can only assign values to the properties of the respective entity, i.e. make sure that writing

```
Car c = {
   birthdate = …
}
```

is illegal.

This is already handled: when creating a Car, you only have property fields for make and model, so you can't add a birthdate.


Car: Object

make: VW

model: Touran

1.5 Integrating manually written code (again in Java, C# or C++).

- Integrate derived attributes to entities.

```
entity Person  {
   string name
   string firstname
   date birthdate
   Car ownedCar
   derived int age // somehow somewhere implement the code
                   // to calculate age
 }
```

Note that if you want, you can also define or reuse an expression language that allows defining the algorithm for calculating the age directly in the model. Ideally, you will show both (manually written 3GL code as well as an expression language).

Manually written code can be integrated in many ways in MetaEdit+. The easiest and best way is to have objects in the model that refer to a manually written piece of code, e.g. a module, class or function. Just the name or filename of that code element is part of the model: the actual implementation is in a separate manually written code file. For a filename, the user can open the file from within MetaEdit+ by using the External Element property datatype / widget.

Although this is the best way, it is also trivial, so here we shall show another way: allowing users to edit the generated file. This may not be best practice, but users find the idea reassuring, and it can sometimes be appropriate for languages that lack mechanisms like #include and partial classes. By allowing editing only in clearly marked protected regions, we can minimize the downsides.

The MERL filename…write…close command has a counterpart, filename...merge…close, which merges the output with any existing file of that name. The command can specify the protected region delimiters, and MD5 checksums are generated for each region so merging can choose correctly between the newly generated contents for that region and the existing manually written contents.

In the DVLC Structure graph, choose **Graph | Edit Generators…**, and open the hierarchy for the Java generator to select _JavaFile. Change "write" there to "merge":

```
filename id '.java' merge
   subreport '_Java' run
close
```

We will use the default region delimiters, /* MEPMD5 … */. We must still specify where in the file the protected regions are: let's have one at the end of the field list, where we can add our derived attribute, and another at the end of the accessor methods, so we can add the implementation of the calculation.

In the _Java subgenerator, after the first do ~Referrer>Reference {…} block, and just before the constructor, add an md5id…md5block…md5sum command to create a protected region:

```
'	' md5id 'insert manual fields between here and MD5 checksum' md5block
'	' md5sum
```

Each protected block in a file must have a unique md5id, in this case just the string above: this will be output as a comment using the region delimiters. Between md5block and md5sum go the automatically generated contents, which the user can manually override by editing. Sometimes this is many lines of generated output, but here we want it to be basically empty, so we just have a tab to align the layout. The md5sum will output a comment containing the MD5 checksum of the contents of the protected region. The output for this piece of code will thus be:

```
    /* MEPMD5 insert manual fields between here and MD5 checksum */
    /* MEPMD5 5e732a1878be2342dbfeff5fe3ca5aa3 */
```

At the end of the _Java generator, after the second do ~Referrer>Reference {…} block, and just before the closing brace } for the class, add a second block:

```
'  ' md5id 'insert manual field accessors between here and MD5 checksum' md5block
'  ' md5sum
```

Save the generator and run the top-level Java generator on the DVLC Structure graph. In the Generated Files dialog you can double-click the Person.java output file to edit it with MetaEdit+'s text editor, or press **Run Output** to edit the selected file in the system default editor for .java files. Add something between the first two /* MEPMD5 */ lines, e.g. "derived int age;", and save the file, making sure it overwrites the original in your user MetaEdit+ 4.5 directory. Now when you generate again, you will see that your change is maintained. Similarly, if you add a new Property to the Person object in the model and generate again, you will see that the changes are merged: Person.java has the attribute for the new Property as well as the manually added derived attribute.

1.6 Multiple generators.

- Generate some kind of XML structure from the entity model.

This has already been done in Tasks 1.3 and 1.1. MetaEdit+ allows you to freely combine multiple generators, on multiple graphs, of multiple graph types, producing multiple files.

Phase 2 - Non-Functional

Phase 2 is intended to show a couple of non-functional properties of the LWB. The task outlined below does not elaborate on how to do this.

2.1 How to evolve the DSL without breaking existing models

MetaEdit+ has unparalleled ability to evolve modeling languages without breaking existing models. Since the space in this document would only allow trivial examples, the best way to see this is in practice.

A good opportunity is in the MetaEdit+ Hands-On at Code Generation 2011 and other conferences, where we incrementally build 5 steadily improving versions of a language for a domain. For the majority of the changes, the models automatically update in response to the language evolution; only when we make what is effectively a completely new language do we discard existing models (or build a model-to-model transformation (as in Task 1.3), or simply decide to keep old models in the old language and build new ones with the new language).

For those who cannot attend the Hands-On, we have a short screencast showing evolution:

http://www.metacase.com/webcasts/DSM_Part3_ModifyMetamodel.html

As in the screencast, the best results for evolution are achieved with the native form-based metamodeling tools of MetaEdit+. In the LWC tasks we have used graphical GOPRR, which is intended for simpler use and therefore works in a more name-based than identity-based fashion. We have also used the even simpler Structure language for metamodeling in Task 1.1.

Even with that simple Structure 'metamodeling' language, we can show a little of the power of MetaEdit+ for evolution:

Addition: Open the DVLC Structure diagram, double-click Person to edit it, and from the pop-up menu in a blank area of the Properties list add a new Property 'country'. Run the Build generator again on that diagram with **Graph | Generate...** (or see extra Task 3.2 to add Build as a toolbar button), and the existing Test DVLC model will automatically be updated: the Voelter object now has a new field for country.

Deletion: Remove the 'country' property from Person, Build again, and the field is removed from Voelter.

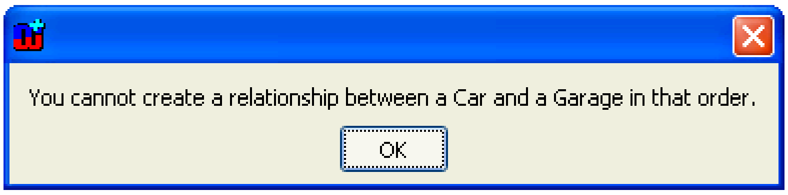
Insertion: In Person, select the property 'birthdate' (a new element will be inserted before the selection, if any), choose **Add Element...** from the pop-up menu, and add a new Property 'title'. Build again, and see that Voelter now has a title field before birthdate.

Update: In Person, double-click the 'name' Property and change it to 'surname'. Build and confirm that Voelter is now the value of the surname field.

For Entity types, we defined the MXT generator to use name-based rather than identity-based lookup when updating metamodels. In this mode, generally more familiar to end-users than strong object-oriented identity, we can make something mean the same by giving it the same name. Entities also work differently from Properties because they contain a larger amount of information, and are more interlinked with other model elements (e.g. via Reference relationships). Many things however work the same:

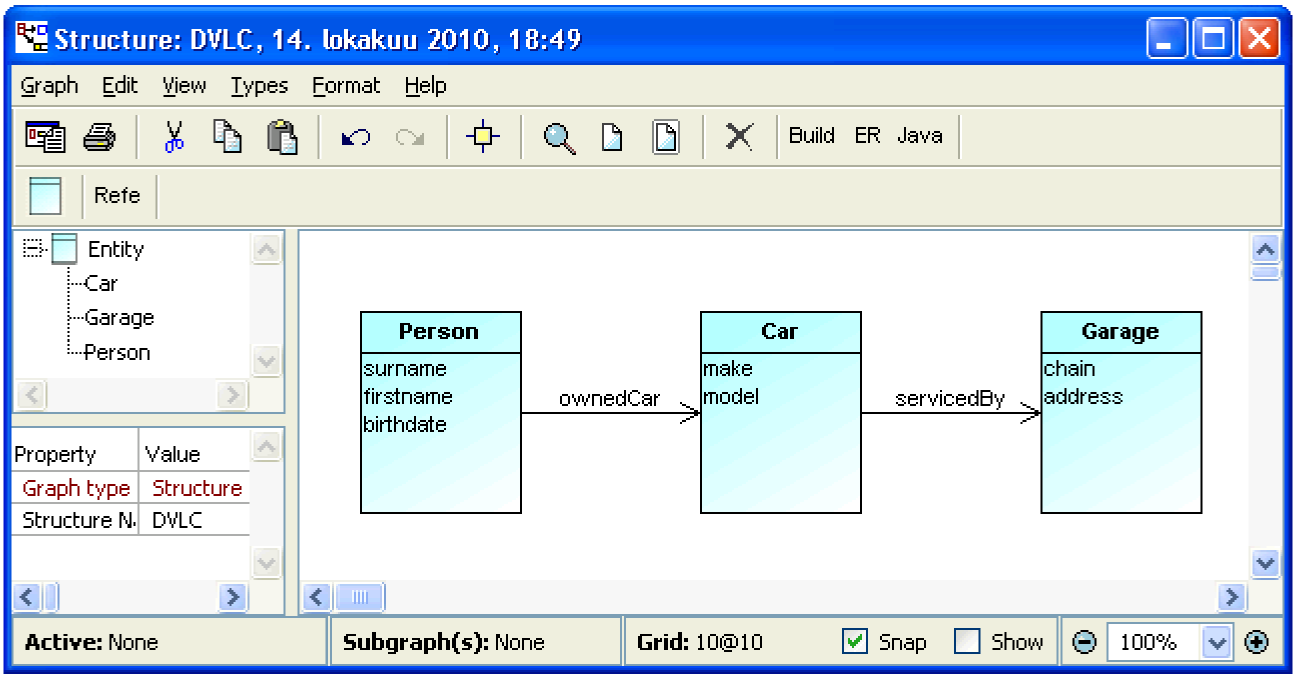
Addition: In the DVLC Structure diagram, create a new Entity called Garage, with properties 'chain' and 'address'. Build, and create a new Garage for 'Texaco' at '10 Main Street'. (If Test was open when you chose Build, you'll need to close and reopen it to update the toolbar with the new Garage type.)

Try creating a relationship between the VW Touran and Texaco, and you'll be prevented, because there are no legal Reference relationships defined for Garage and the other types:



You cannot create a relationship between a Car and a Garage in that order.

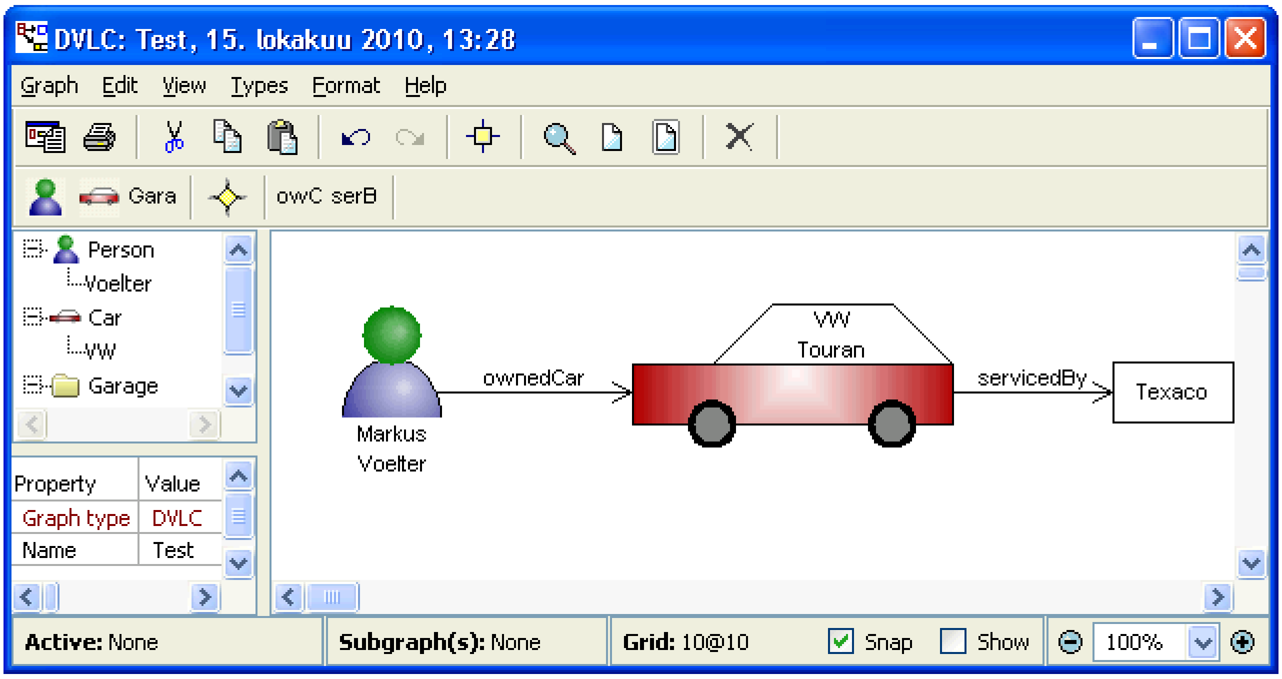
Rule change: Close Test, and back in the DVLC Structure diagram add a Reference from Car to Garage called 'servicedBy', then Build again.



Property	Value
Graph type	Structure
Structure N.	DVLC

Active: None **Subgraph(s):** None **Grid:** 10@10 ✓ Snap ☐ Show 100%

Now in the Test diagram, you can create a servicedBy relationship from the VW Touran to the Texaco Garage:



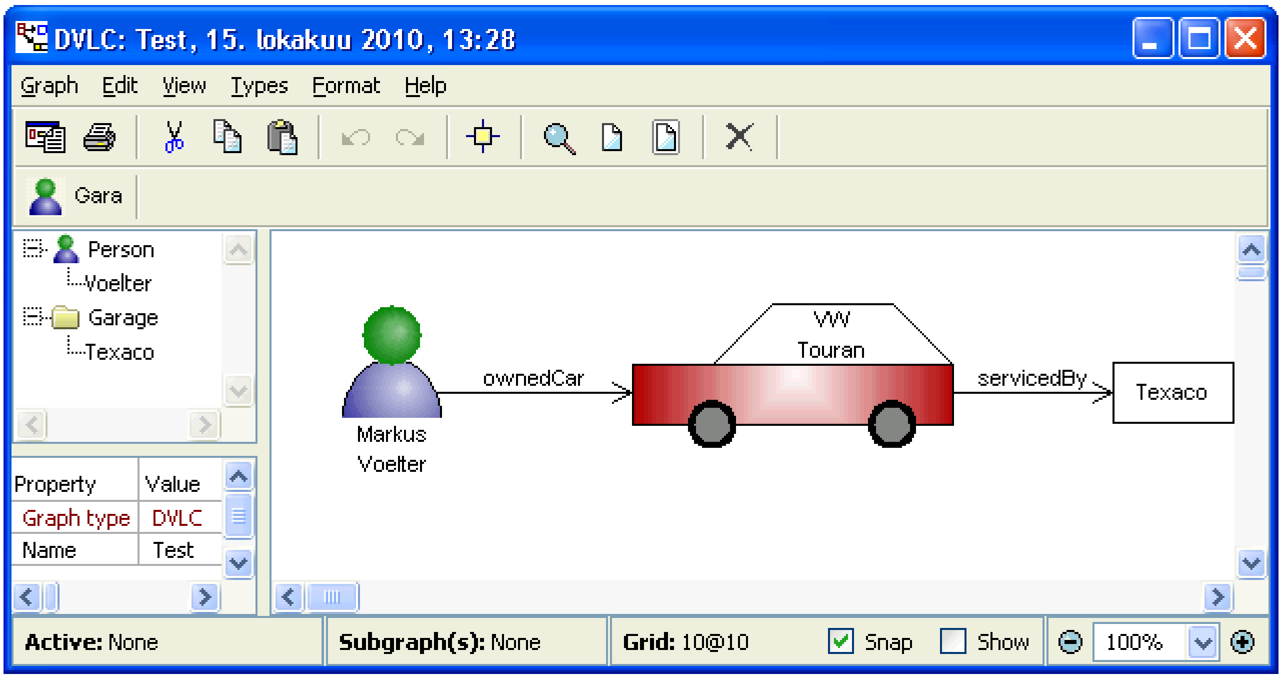
Property	Value
Graph type	DVLC
Name	Test

Active: None **Subgraph(s):** None **Grid:** 10@10 ✓ Snap ☐ Show 100%

Try creating other kinds of relationships, and you'll see they are correctly prevented.

Similarly, if you use the yellow diamond on the toolbar to start creating a relationship, MetaEdit+ will automatically figure out what are the legal relationships between the pair of objects, and create an ownedCar or servicedBy relationship as appropriate, or refuse.

Deletion: Close Test, and in DVLC Structure delete the Car Entity. Build, then open Test again. The results might surprise you:



Property	Value
Graph type	DVLC
Name	Test

Active: None **Subgraph(s):** None **Grid:** 10@10

At first glance, it looks like nothing has happened – the old Touran is still sitting there. On closer inspection, we see that Car has disappeared from the toolbar, as have the ownedCar and servicedBy relationships. MetaEdit+ has deliberately left the Touran and its relationships in the model, because there is significant modeling effort invested in them. If deleting a type instantly deleted all its instances and all connections with other data, the screams from the modelers would be loud indeed – and rightly so. When a keyword is removed from a programming language, it is normal to first mark it as sunsetted or obsolete, before final deletion after a few more language revision cycles. And removing a keyword from Java most certainly doesn't update all Java programs on the planet, deleting just that piece of text! Similarly in human language, when a word becomes obsolete or considered incorrect, existing texts still have that word, and they still "work": people still understand it.

The situation with a modeling language is just the same: the models are still understandable, and any code generators built for the language will still work, able to process these obsolescent elements. The user will just not be able to create new instances of this type. The metamodeler can mark the type as obsolescent, e.g. by adding a red X in its symbol, and can make generators that list all occurrences, or even require them all to be corrected before agreeing to continue processing the models.

In real-world cases, unlike in academia, it is rare for such modeling language changes to be completely automatable – e.g. replace all Cars with Vehicles. If we want to do that in MetaEdit+, we simply change the name of the type in the metamodel (our simple Structure language can't do this, but the form-based tools can: select the Touran, control-double-click the red text 'Car' in the property sheet on the left, change the name to Vehicle and press 'Save and Close'). If there ever really is a case where the transformation would be completely automatable, but not already automatically handled in MetaEdit+, the metamodeler could write an MXM generator to produce the updated models, or write a program in his language of choice to update them in place using the MetaEdit+ API. In the entire history of MetaEdit+, I don't recall a single case where this would have been necessary – despite languages continuously evolving over 15 years, in fast-changing domains, with hundreds of users.

2.2 How to work with the models efficiently in the team

The MetaEdit+ multi-user version allows multiple users to work on the same set of models simultaneously. Its fine granularity of concurrency, down to single objects and even single properties, even allows multiple users to work on different objects within the same diagram. Long design transactions with true ACID properties ensure a consistent view of the repository, and allow roll-back of changes and automatic re-synching to the current state of committed changes in the repository. See the following links for more information:

http://www.metacase.com/mep/multi-user.html

http://www.metacase.com/support/45/manuals/meplus/Mp-6_1_2.html

2.3 Demonstrate scalability of the tools

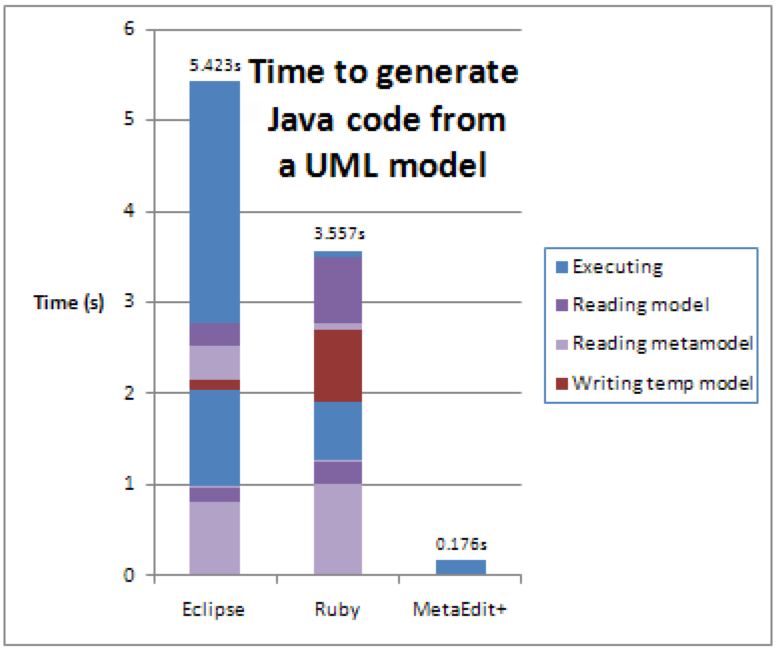
MetaEdit+ is not limited by scale, thanks to its solid foundations, long history, and the efficiency and performance required by the early versions being implemented in the early 1990s — on 486 processors with 16MB of RAM! It has been used on projects with hundreds of users spanning over 15 years and many gigabytes of models – but is equally at home as a single user tool on a mini-laptop.

Scalability also means scaling to small cases, for which a powerful tool can sometimes be cumbersome. MetaEdit+'s power does not however lead to a large initial learning curve for the tools, or large expenditure of effort in creating modeling tool support for a new language. Indeed, the opposite is the case: new users can build their first full languages with code generation within an hour. The time to create an industrial strength, deployment-ready solution averages 5-15 workdays, measured consistently over many cases.

A common scalability bottleneck for flexible tools is memory usage – this has been the case at least since Emacs ('Eight Megs And Constantly Swapping', which was huge in the 1980s). With all the metamodels and models of the LWC tasks open (including graphical GOPRR), and all generators run, memory usage is less than 40MB.

Another frequently encountered bottleneck is code generator performance. For MetaEdit+ figures see the graph below, where we duplicated a published experiment in which Ruby had been somewhat faster than Eclipse in generating Java code from a UML model. MetaEdit+ generated the same code from the same classes in 0.176 seconds — 30 times faster than Eclipse. More details are in this blog entry:

http://www.metacase.com/blogs/stevek/blogView?showComments=true&entry=3385914921



Phase 3 - Freestyle

3.1 Translators – extension to Task 0.2

When we generated the Java code in Task 0.2, we generated accessors like "get_foo()". It would be nicer to have "getFoo()", so let's add that.

MERL uses *translators* to filter, transform, translate or map characters and strings from the user-friendly format in models, into the computer-friendly format required by output files. Such translations are common in many kinds of generators, e.g. to turn spaces in user-entered names into underscores to make legal variable names in output code, or to escape reserved characters like & into &amp; for XML. MERL thus offers its own little language and mechanism for defining translators, and MetaEdit+ comes with a set of commonly-used translators for you to use, edit and extend. For instance, here's the definition of %upper, which says "replace each lower-case character with the corresponding upper-case character".

```
to '%upper
a-z A-Z'
endto
```

In this case, %upper is not quite what we want – it would give us "getFOO()" – so we'll make a couple of new translators. Since translators may vary depending on the modeling language and syntax of the output file, they are "included" by defining them, normally at the start of the outermost generator. Go back to the earlier 'Java' generator and add this line at the start:

```
subreport '_JavaTranslators' run
```

Save that and define the new _JavaTranslators generator. Start its body by calling _translators to define the generic ones supplied with MetaEdit+, including %upper, then add two regular expression translators: %first that just outputs the first character, and %rest for everything after the first character. Note that the right hand side of each is a string (the first $ prefix), whose contents are $1, the match of the first parenthesized subexpression on the left hand side.

```
subreport '_translators' run

to '%first
/(.).*/ $$1'
endto

to '%rest
/.(.*)/ $$1'
endto
```

Now with those defined, we can use them in the _JavaAccessors generator (if you can't see it, open Java, _JavaFile, _Java in the Hierarchical view, or switch to the Alphabetical view). Delete the underscore after get and set, and replace the id after them with the first character of the id converted to uppercase (noting that we can chain translators), followed by the rest of the id:

```
'  public ' $datatype ' get' id%first%upper  id%rest '() {
      return ' id ';
   }

   public void set' id%first%upper  id%rest '(' $datatype ' value) {
      this.' id ' = value;
   }

'
```

If you want to learn a little more about translators, you can look at the examples in the _translators subgenerator, which we called in _JavaTranslators. Each translator is defined with a to…endto block, whose contents (generally a single fixed string) form the definition. The first line is a name starting with %, and each subsequent line is a rule. The left hand side of the rule is a character, string or regular expression to search for, separated with a space from the right hand side, which is the replacement character or string. Characters are written literally or as ranges like a-z; strings are prefixed with $, and regular expressions surrounded with /slashes/. For more details, see the manual:
http://www.metacase.com/support/45/manuals/mwb/Mw-5_3_6.html#Index204

With a little more experience, you could make a more complicated translator to handle the whole CamelCase operation at once, rather than in two parts:
http://www.metacase.com/forums/forum_posts.asp?TID=115.

3.2 Toolbar buttons to run generators

MetaEdit+ allows you to add the most commonly used generators as buttons on the Action Toolbar of the Diagram Editor, for the convenience of your modelers.

Open a Generator Editor for Structure graphs, e.g. by select the DVLC: Structure graph in the main MetaEdit+ window and pressing the MERL button at the end of the toolbar. Press the New button in the Generator Editor, and give a short name starting with an exclamation mark, e.g. !Java. The exclamation mark means this generator will be shown in the toolbar.

For the body of the generator, call the existing generator you want to run with this button:

```
subreport 'Java' run
```

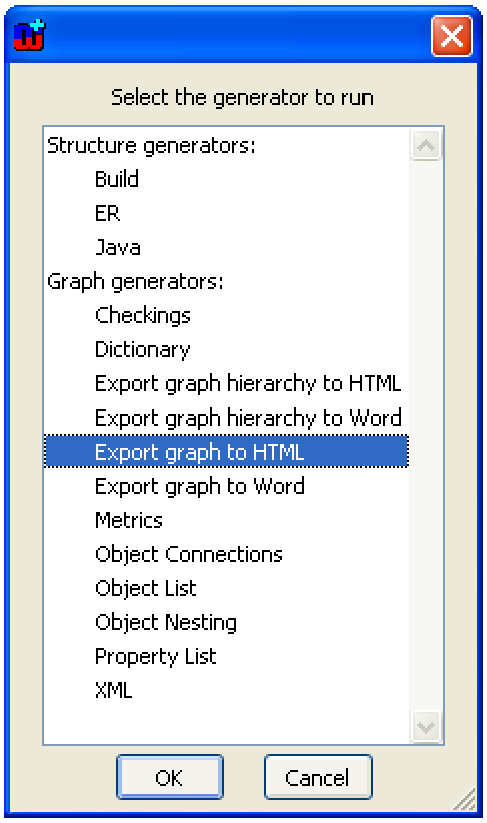
Save the generator, open the DVLC: Structure graph, and you will see the new button at the end of the Action Toolbar.

You can repeat this for any of the other generators you make in subsequent tasks, e.g. !ER and !Build.

3.3 Generic generators and inheritance

There are also predefined generators that you can run via Graph | Generators… and they are defined with MERL too. These generators are available for all languages, since they are defined on Graph itself, the supertype of all Graph types. GOPRR and MERL both support inheritance for the modeling languages.

The list of available generators for our DVLC Structure diagram is shown on the right. Try the 'Export Graph to HTML' generator for a quick taste of the supplied generic generators.


Select the generator to run

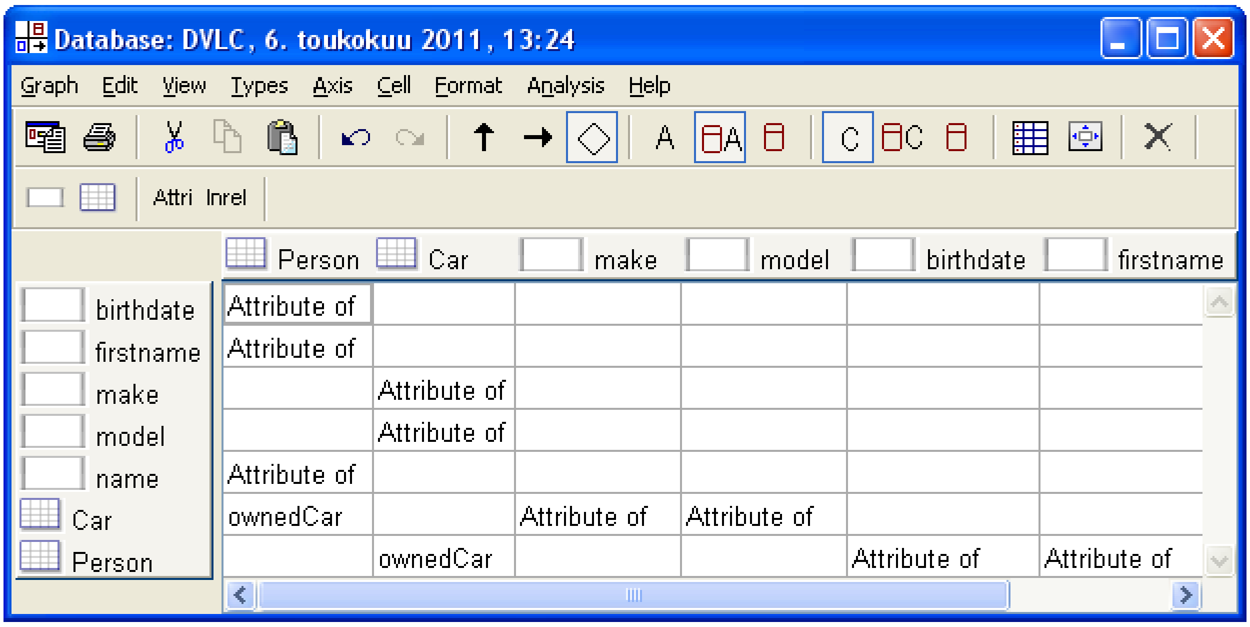
Structure generators:
- Build
- ER
- Java

Graph generators:
- Checkings
- Dictionary
- Export graph hierarchy to HTML
- Export graph hierarchy to Word
- Export graph to HTML
- Export graph to Word
- Metrics
- Object Connections
- Object List
- Object Nesting
- Property List
- XML

3.4 Matrix and Table Editors

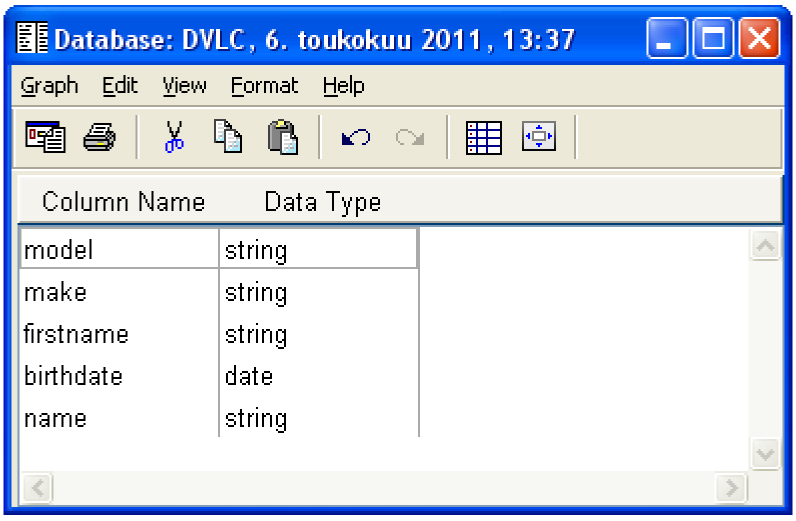
In addition to graphical diagrams, MetaEdit+ supports matrices and tables. Any graph can be viewed as a matrix and table, and the same graph can even have several diagrams, matrices and tables. All of these are representations of the same underlying conceptual data. Unlike projections, whose layout is auto-generated each time they are opened, leading to problems when the layout changes significantly from the previous time, these are true representations which can have their own layout information.

In the main MetaEdit+ window, select the DVLC: Database graph created in Task 1.3, and from its popup menu select **Open as Matrix…**. Each row and column is an object, and each cell show relationships between that pair of objects. You can turn on symbol display for the row and column headers with **Axis | Show Text + Symbol**, automatically set columns' width with **Format | Autowidth**, and diagonalise the matrix with **Analysis | Diagonalise**.


Database: DVLC, 6. toukokuu 2011, 13:24
Graph Edit View Types Axis Cell Format Analysis Help
Attri Inrel

	Person	Car	make	model	birthdate	firstname
birthdate	Attribute of					
firstname	Attribute of					
make		Attribute of				
model		Attribute of				
name	Attribute of					
Car	ownedCar		Attribute of	Attribute of		
Person		ownedCar			Attribute of	Attribute of

Similarly, you can open a Table Editor for the same graph with **Open as Table…**. Tables list objects showing all their properties in the columns, so you are asked to select the type of object to display: choose 'Column'.


Database: DVLC, 6. toukokuu 2011, 13:37
Graph Edit View Format Help

Column Name	Data Type
model	string
make	string
firstname	string
birthdate	date
name	string

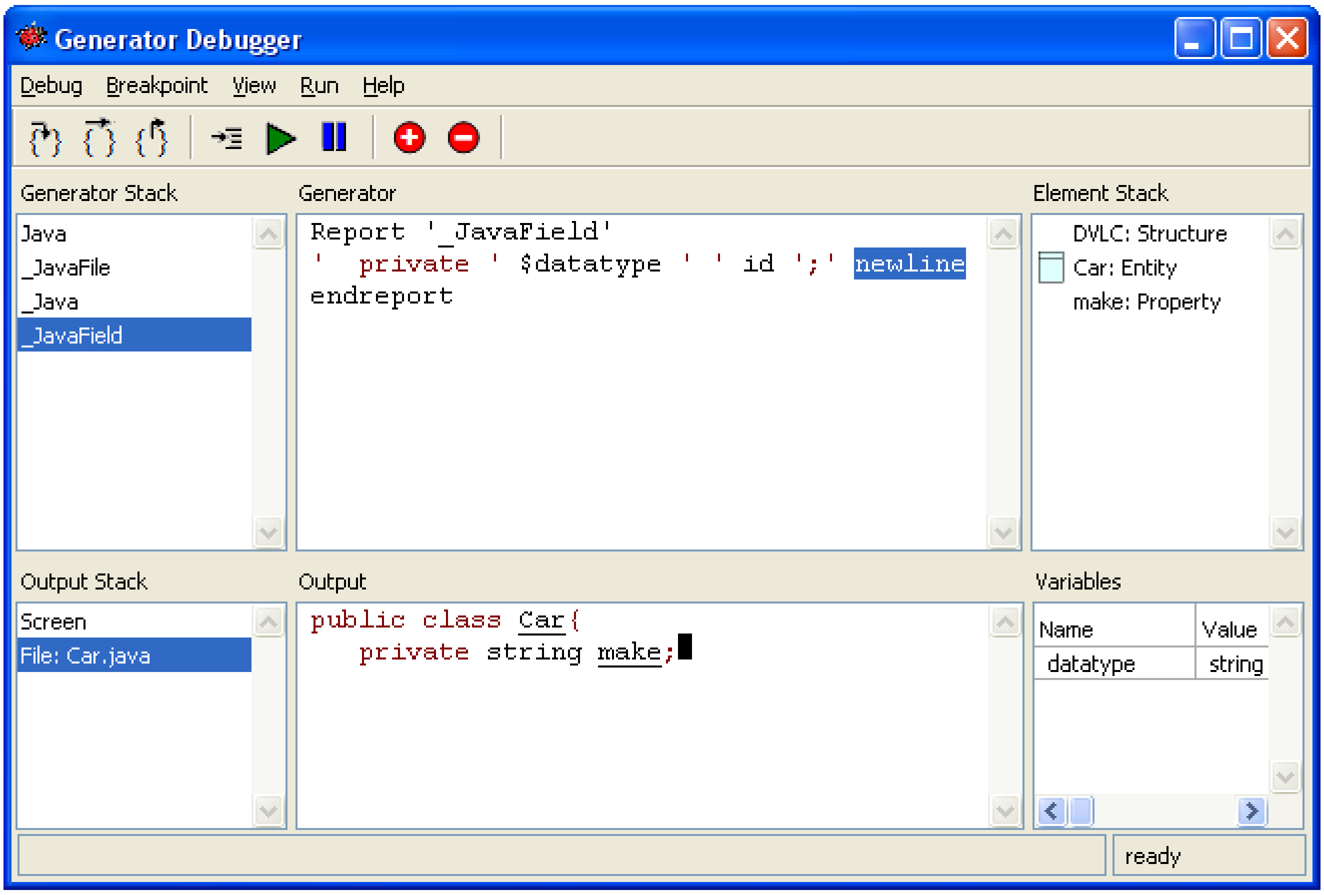
Now when you open DVLC: Database by double-clicking, you'll be prompted to choose which of its three representations you want to open. You can delete no longer needed representations from the graph's pop-up menu in the main MetaEdit+ window, with **Delete Representation…**.

MetaEdit+ will take care of the conceptual data consistency over multiple representations, while still ensuring the representational freedom of each separate representation. For instance, you can make a simplified diagram for managers, showing only some objects. When you update the objects in the main diagram, the simplified diagram will be kept up to date. When you add more objects in the main diagram, they won't be forced into that simplified diagram, but you can choose to add them with **Graph | Import Graph** or individually from the diagram pop-up menu item **Add Existing…**.

3.5 Generator Debugger

MetaEdit+ provides a full debugger for MERL generators. In the main MetaEdit+ window, select DVLC: Structure and press the MERL button at the end of the toolbar to open a Generator Editor. In the Generator Editor, select Java from the top left list and choose **Generator | Debug…**. If you have several Structure graphs, you will be asked which you want to run the generator on: choose DVLC.

Here's the debugger after pressing the 'Step Into' button several times:



Java
_JavaFile
_Java
_JavaField

```
Report '_JavaField'
'  private ' $datatype ' ' id ';' newline
endreport
```

DVLC: Structure
Car: Entity
make: Property

Screen
File: Car.java

```
public class Car{
   private string make;
```

Name	Value
datatype	string

The top half shows the call stack of generators to reach this point, the position within the current generator, and the stack of model elements navigated through to reach this point. You can investigate the model elements from their pop-up menu, to see where you are in the model and check the model data.

The bottom half shows the stack of output files, the state of the current output file so far, and the values of any MERL variables. You can double-click a MERL variable to see it in full, and even edit it and continue debugging.

Breakpoints can be added and removed here or in the Generator Editor. (The latter can also list all generators that have breakpoints: choose the view type from the menu box above the generator list.)