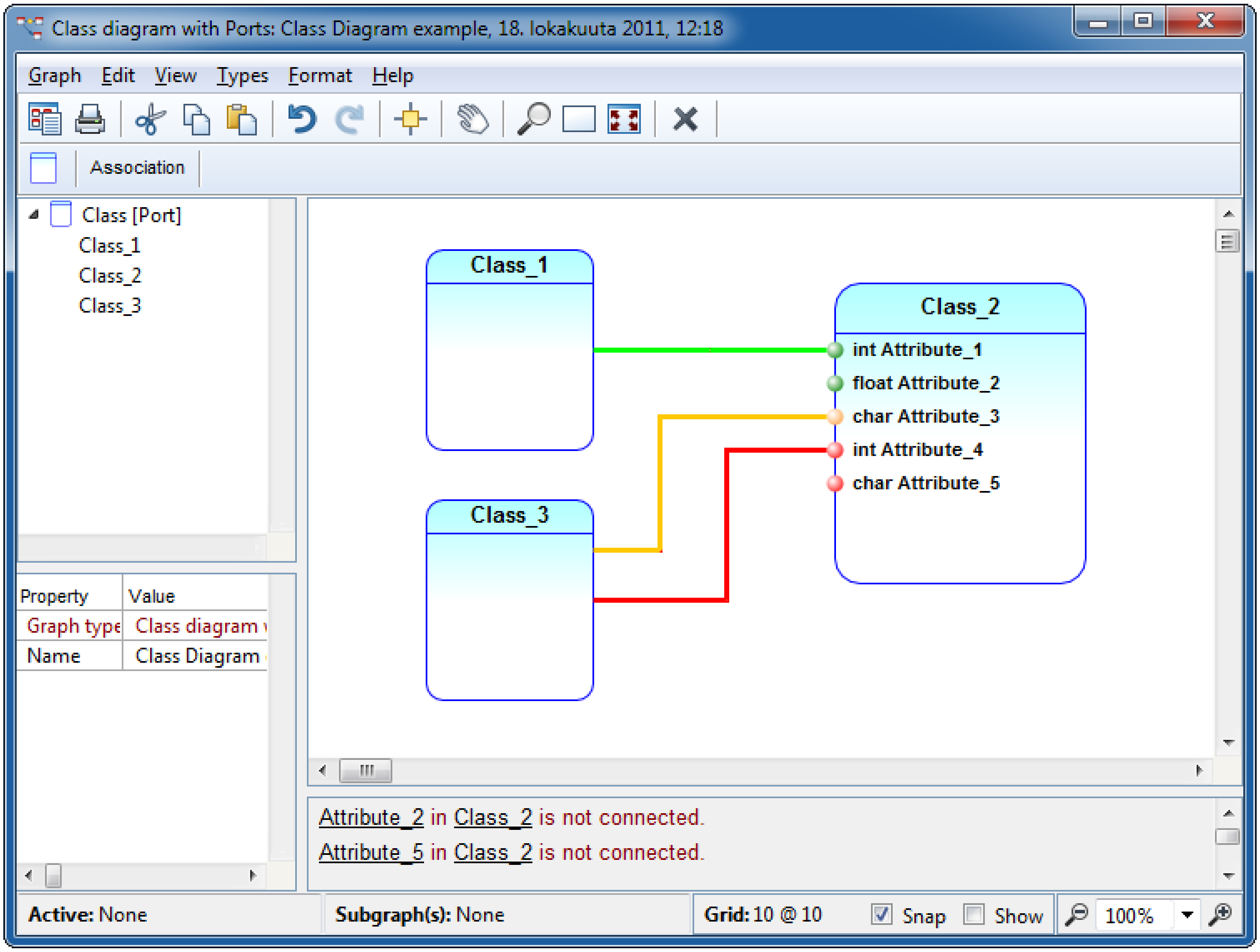
MetaEdit+ 5.0 Beta Primer

March 9th 2011 / Build 30



Property	Value
Graph type	Class diagram
Name	Class Diagram

Attribute_2 in Class_2 is not connected.
Attribute_5 in Class_2 is not connected.

1 Overview

Welcome to the MetaEdit+ 5.0 beta!

The feature highlights of 5.0 compared to 4.5 are:

- IDE integration for Visual Studio and Eclipse
- View changes graphically directly in the models
- Enhanced dynamic behavior of symbols
    - Templates to include common elements in multiple symbols
    - Dynamic ports shown based on component interfaces etc.
- Faster, more intuitive modeling
    - New tools for managing model hierarchies
    - Create large objects by dragging area
- New generator functions
    - Generator parameters, local variables
    - Conditional breakpoints, break into running generator
- New graphics engine with high-quality anti-aliasing
- Windows 7 support

For a more detailed technical rundown of new features, please refer to the "What's New" document included within the beta download zip file. This beta version is available for Windows platforms only; other platforms will follow later.

Obviously, you should have the normal expectations for a beta: great new functionality, but poor documentation and some bugs, possibly even serious ones. We would like you to report (metaedit.support@metacase.com) errors by attaching the meplus*.err file and possibly a zip of your database, explaining what you were doing when the error occurred. When we send out a new beta version, we would appreciate it if you could move to using it as quickly as possible, and follow any instructions provided for updating your databases to the new version - this way we save you from hitting bugs that others have already reported.

Liability: MetaCase shall not be liable for any special, incidental, indirect, consequential, punitive or exemplary damages whatsoever (including, without limitation, damages for loss of goodwill, business profits, business interruption, loss of business information, or any other pecuniary loss) arising out of the use of or inability to use the software referred to here as MetaEdit+ 5.0 beta.

2 Installation and getting started

This beta version is distributed as a zip archive that contains the following files
(* represents a version number):

- **testmep50Build*.exe**: MetaEdit+ program
- **demo***: the demo database
- **cairo.dll**: Cairo graphics libraries
- **What's New *.txt**: the list of new features, improvements and fixes
- **MetaEdit+ 5.0 Beta Primer.pdf**: this document

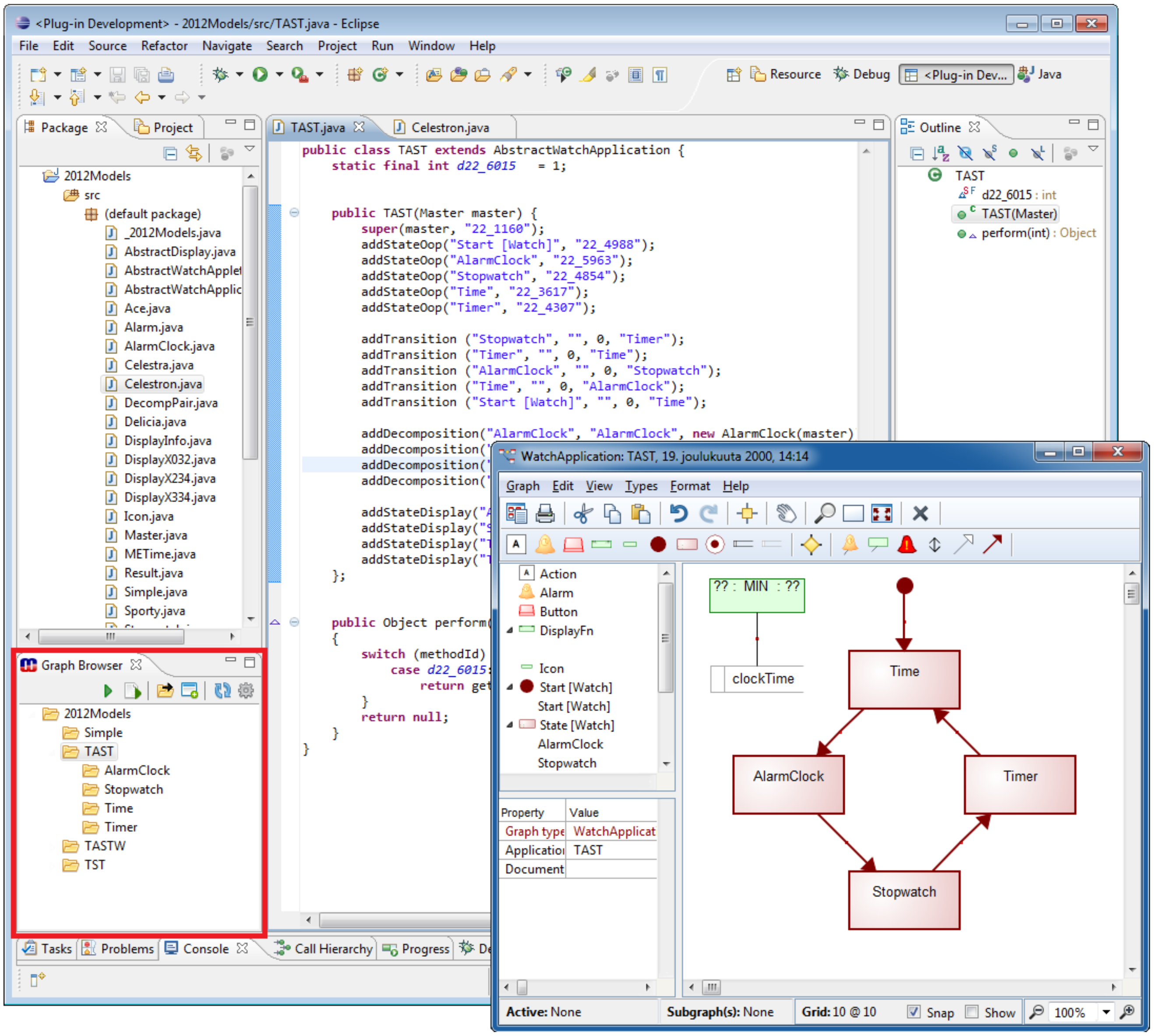
For installation, simply unzip the archive into a new folder created to host the beta. If you unzip into an existing beta folder, please delete the old beta executable, and be careful not to overwrite your existing repository. **We strongly advise NOT to unzip or copy these files into your existing MetaEdit+ 4.5 installation folder!**

To start MetaEdit+ 5.0 beta, simply double-click **testmep50Build*.exe** and proceed with the login as usual.

As for trying out and using the beta, you are encouraged to explore the examples in our 5.0 demo database supplied with the beta. Or you can use your existing database with the beta version (it will be converted for 5.0 on the fly on login), but note that changes made there cannot be back-ported to work with MetaEdit+ 4.5: 4.5 simply does not support the new features, which is where most of your changes will presumably be. There will be a series of beta versions, culminating in one or more release candidates. The beta versions will be time-limited, but we promise to keep you up and running until the next release is available, should you want to move your development completely to the beta version at some point.

3 Enhanced IDE Integration

Since version 4.0, organizations that use MetaEdit+ have integrated it with their IDEs via code generation, build script generation, and its API. For MetaEdit+ 5.0 we identified the common IDE integration scenarios and put them together in ready-made Eclipse and Visual Studio plugins. The plugins will work with any modeling language, allowing you to work with MetaEdit+ directly from your IDE: see the hierarchy of your models, open models, generate code, and even build and update whole IDE projects.



```
public class TAST extends AbstractWatchApplication {
    static final int d22_6015   = 1;


    public TAST(Master master) {
        super(master, "22_1160");
        addStateOop("Start [Watch]", "22_4988");
        addStateOop("AlarmClock", "22_5963");
        addStateOop("Stopwatch", "22_4854");
        addStateOop("Time", "22_3617");
        addStateOop("Timer", "22_4307");

        addTransition ("Stopwatch", "", 0, "Timer");
        addTransition ("Timer", "", 0, "Time");
        addTransition ("AlarmClock", "", 0, "Stopwatch");
        addTransition ("Time", "", 0, "AlarmClock");
        addTransition ("Start [Watch]", "", 0, "Time");

        addDecomposition("AlarmClock", "AlarmClock", new AlarmClock(master)
        addDecomposition('
        addDecomposition('
        addDecomposition('

        addStateDisplay("A
        addStateDisplay("S
        addStateDisplay("T
        addStateDisplay("T
    };


    public Object perform(
    {
        switch (methodId)
            case d22_6015:
                return get
        }
        return null;
    }
}
```

Property	Value
Graph type	WatchApplicat
Applicatio	TAST
Document	

Rather than trying to duplicate MetaEdit+ graphical modeling in the IDE, or IDE code editing in MetaEdit+, the plugin integrates the best of both tools into a cohesive whole, with the familiar UI of the IDE as the starting point. And of course the plugin itself is open source, so you can improve and extend it to fit your own needs.

You can try the plugin yourself with the MetaEdit+ 5.0 beta version, whose Digital Watch example has been updated to generate and build Eclipse and Visual Studio projects. The plugin download sites have the instructions for installing and using the plugin:

Eclipse:	http://code.google.com/p/metaedit-plugin-for-eclipse/
Visual Studio:	http://graphbrowser.codeplex.com/

Your existing modeling languages already work with the plugin, and you can use the plugin's hooks in your generator to have the plugin create IDE projects out of generated code. To do this, follow the IDE folder structure for projects (e.g. for a Java project in Eclipse, create src/ and bin/ folders under the project folder), and generate the project file listing the source files – see the plugin reference for details:
http://code.google.com/p/metaedit-plugin-for-eclipse/wiki/ExtendingThePluginGuide
http://graphbrowser.codeplex.com/wikipage?title=ExtendingThePluginGuide

As a starting point when trying out the IDE integration, we recommend you to take a tour around the Digital Watch example that now includes different integration examples. These have been implemented as different generation target platform options, selectable with the Generation target platform property in "WatchFamily: 2012Models":

- **C#: Windows**: C# generation and integration with .NET 4 / MS Visual Studio
- **C#: API**: C# generation with API support (model animation call-back)
- **C#: WP 7**: C# generation for Windows Phone 7 (requires appropriate SDK)
- **Java: Windows**: Java generator for Windows JDK, called by Eclipse
- **Java: API**: Java generation with API support (model animation call-back)
- **Java: Android**: Java generator for Android (requires Android SDK)
- **Java: MIDP**: Java MIDP generator for mobile phones.
- **Java: Linux**: Java generator for Linux platform.
- **Java: Mac OS X**: Java generator for Max OS X.

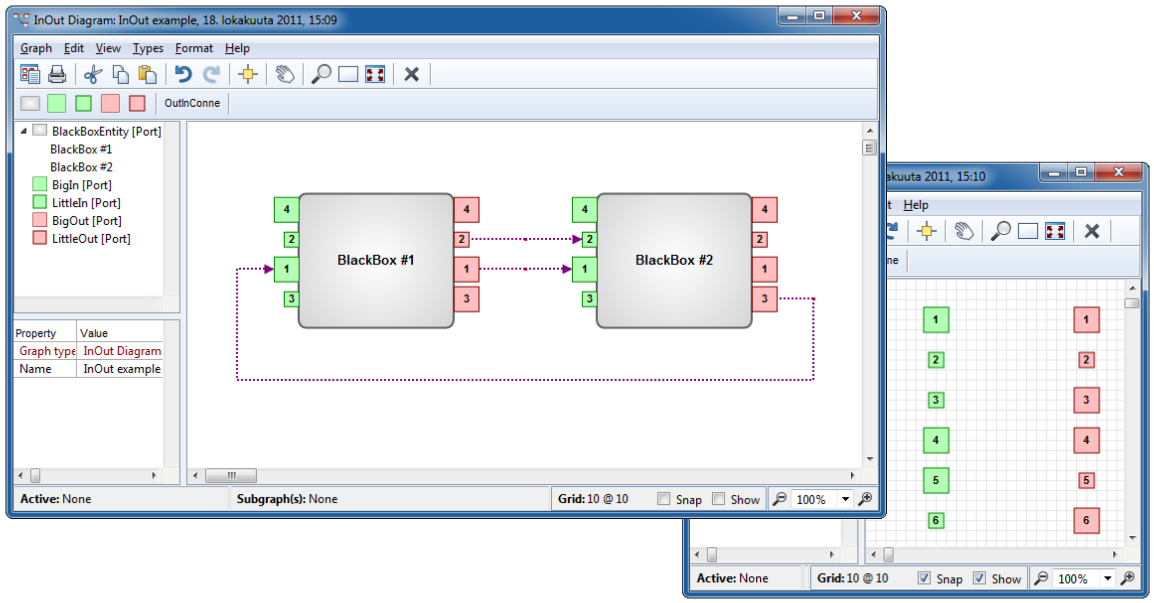
The Autobuild generator, run either from the plugin or manually, will use whichever of these is selected. When run from the plugin, an IDE project will be created to contain the generated source code, and the project will be built and run by the IDE.

4 Template Symbol Elements and Dynamic Ports

Template symbol elements offer a powerful way to construct symbols using pre-defined subsymbols that are fetched dynamically and displayed as part of the hosting symbol. These subsymbols can also act as dynamic ports for more precise role-object-connection semantics, in a similar fashion to the existing static ports. In the following we describe the basic use of templates and dynamic ports and explore some of their possible applications.

4.1 Dynamic Ports Fetched from Subgraph

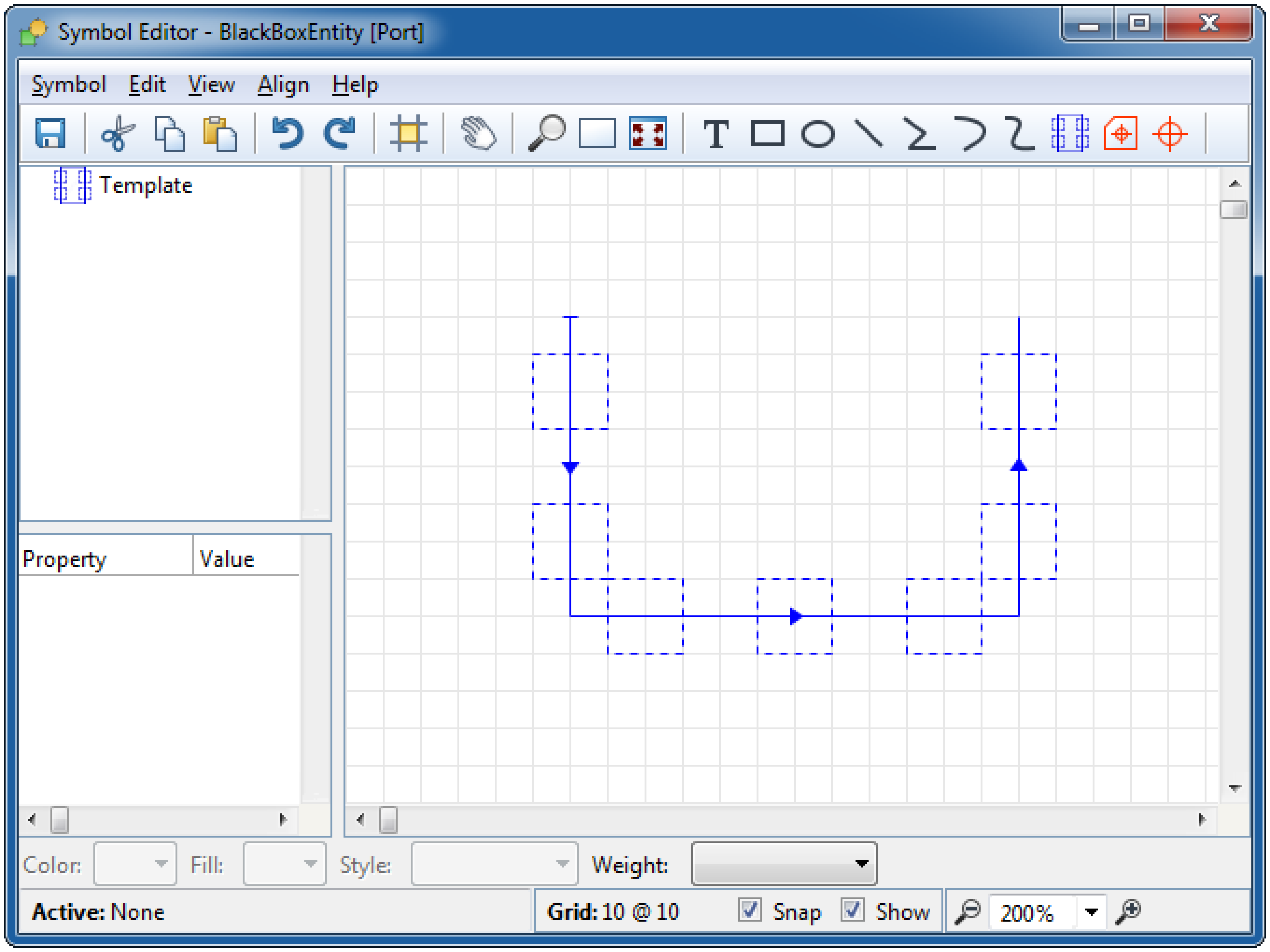
This example illustrates one of the most typical use cases for templates and dynamic ports. Quite often in various problems domains there is a need to be able to build hierarchies of components, where the component on the upper level hides its lower-level implementation in a black box fashion, revealing only the required interface elements from the lower level. A typical scenario, illustrated below, is that the black box component provides IN and OUT ports that are available for incoming and outgoing relationships on the upper level and appear as entry and exit points on the lower one.



The above InOut example can be found from the Port example project in the Demo repository. By examining this example we will show how to build such languages. In order to create something like this, we first need to create the following:

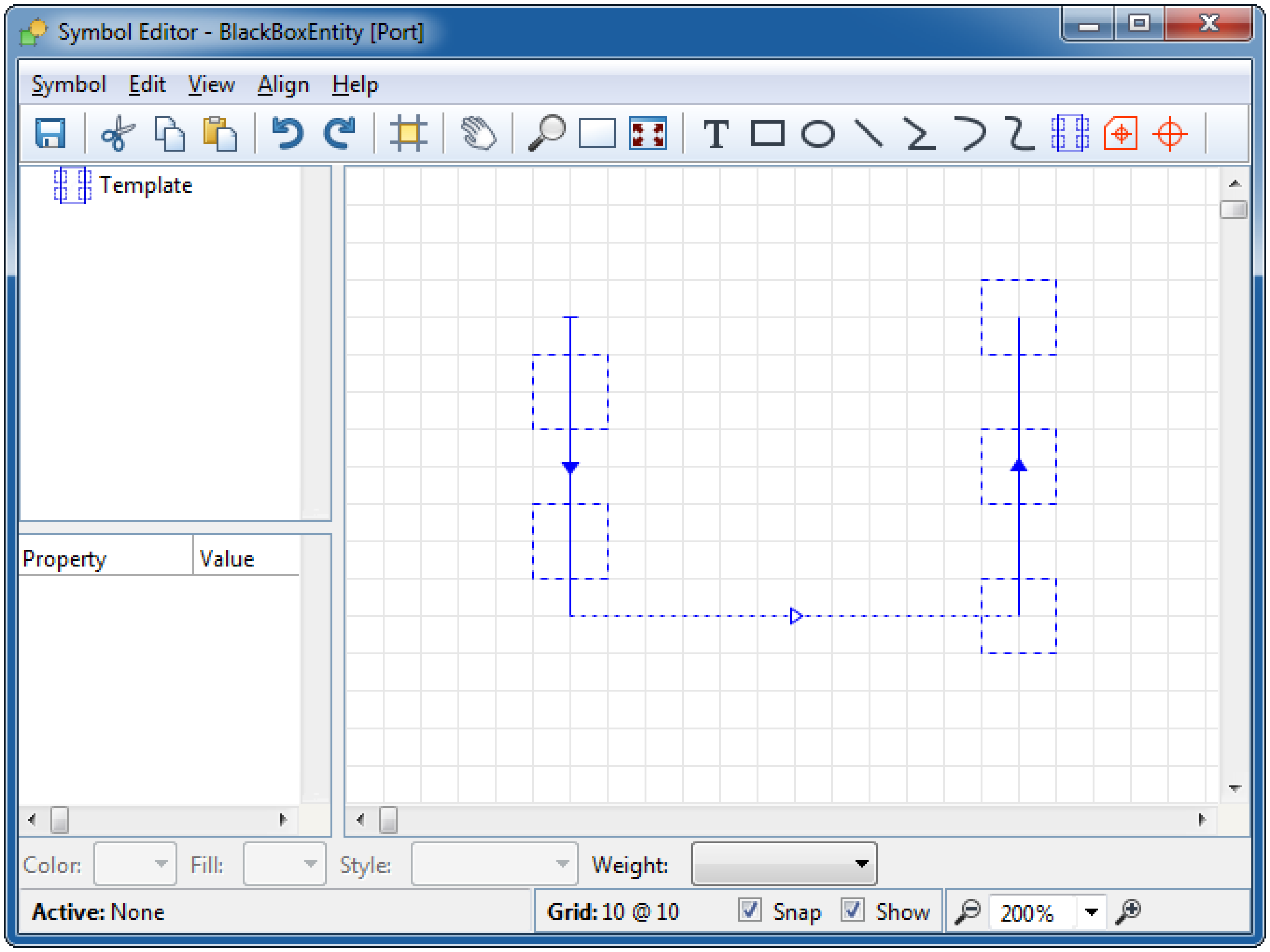
- An object type to host the template and dynamic ports (called **BlackBoxEntity** in our example)
- Object types that will appear as subsymbols and dynamic ports (**BigIn**, **LittleIn**, **BigOut** and **LittleOut** in our example)
- A graph type that contains the **BlackBoxEntity** objects and allows them to have subgraphs containing **BigIn**s, **LittleIn**s, **BigOut**s and **LittleOut**s. Often the main graph and subgraphs are the same type, as is **InOut Diagram** here.

When these concepts are available, we can start defining the symbol for **BlackBoxEntity**. The first thing we need is a template path along which the subsymbols are distributed. Open the Symbol Editor, select the Template creation tool from the toolbar, move the mouse back to the drawing area and create a template path as for a polyline by clicking each point along it, double-clicking the last. You will end up with something like this:

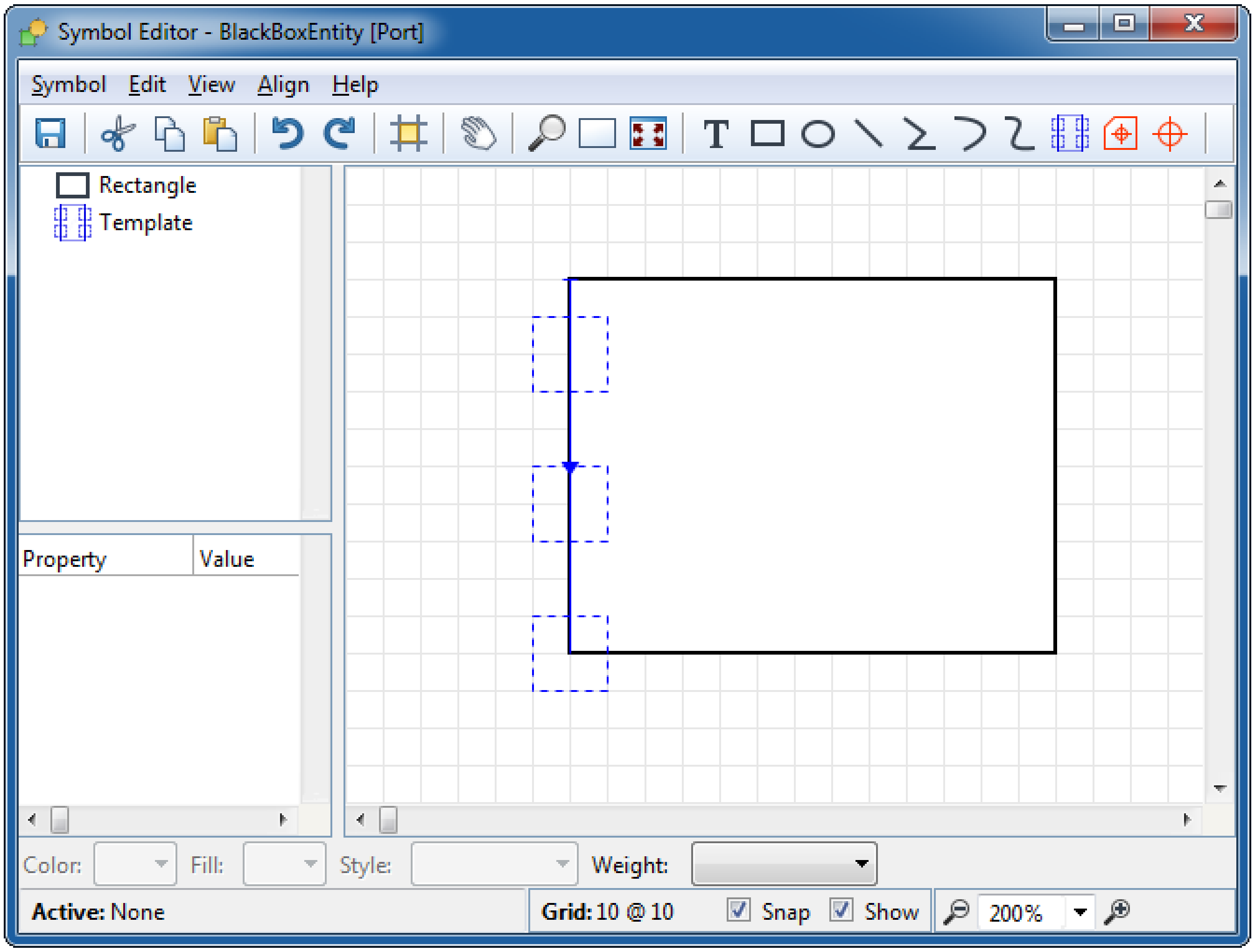


The solid blue line defines the template path and the blue rectangles with dashed perimeters stand for the slots where the subsymbols will be placed. The blue arrow heads in the middle of the path segments denote the direction of the subsymbol distribution.

The individual path segments can be toggled on and off – when toggled off, no subsymbols will be allocated on that segment. To toggle a segment on or off, select **Edit Points** from the template's pop-up menu, open the pop-up menu for a path segment and choose **Active Segment** to toggle it. The result will look like this:



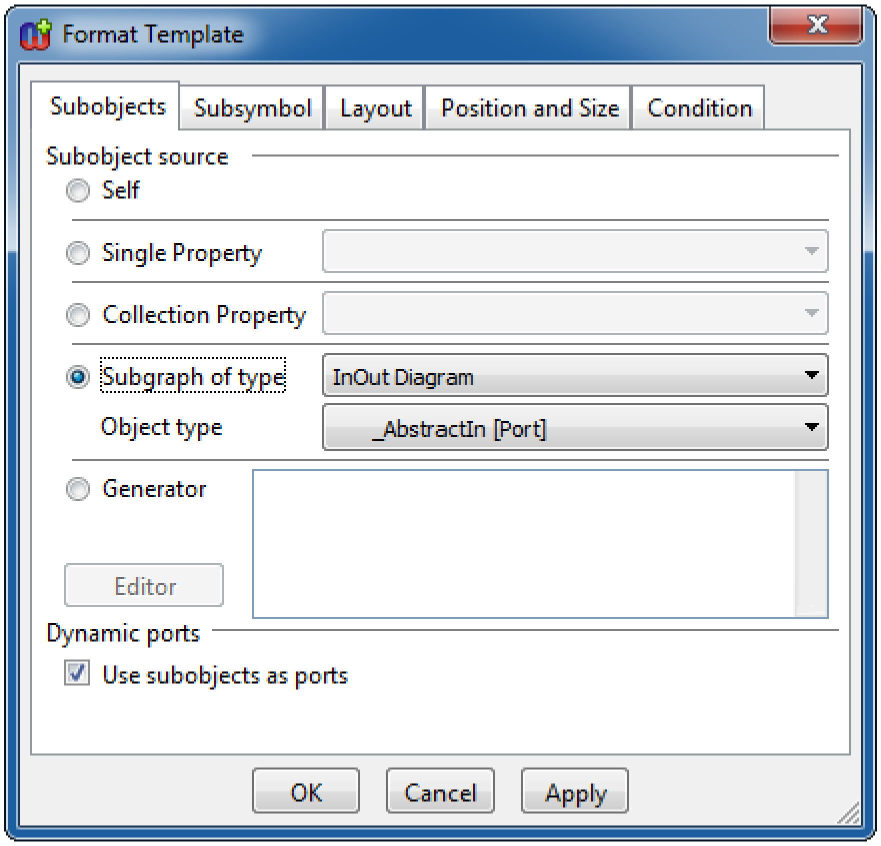
However, what we need now for our example, is just a straight top-down path that we will align with a rectangle representing the **BlackBoxEntity**, like below. Choose **Edit Points** and delete the two extra points on the right of the template by Ctrl-clicking them:



The next thing to do is to define where to get the subsymbols from. As the visible subsymbols always stand for real object instances, we need to first define how to fetch these subobjects. Open the format dialog for the template. The source for the subobjects will be defined on the first dialog page. We can choose from five possible sources:

- **Self**: use the hosting object as a subobject as well (would be BlackBoxEntity in our case)
- **Single Property**: use the object from a single property in the host object
- **Collection Property**: use the objects from a collection property in the host object
- **Subgraph of type + Object type**: from the host object's subgraph of the specified type, use the objects of the specified type
- **Generator**: use a set of objects retrieved by a generator and output one per line

As the subobjects in this example are contained in a subgraph, we could select the subgraph option and select the proper subgraph type and object type within:

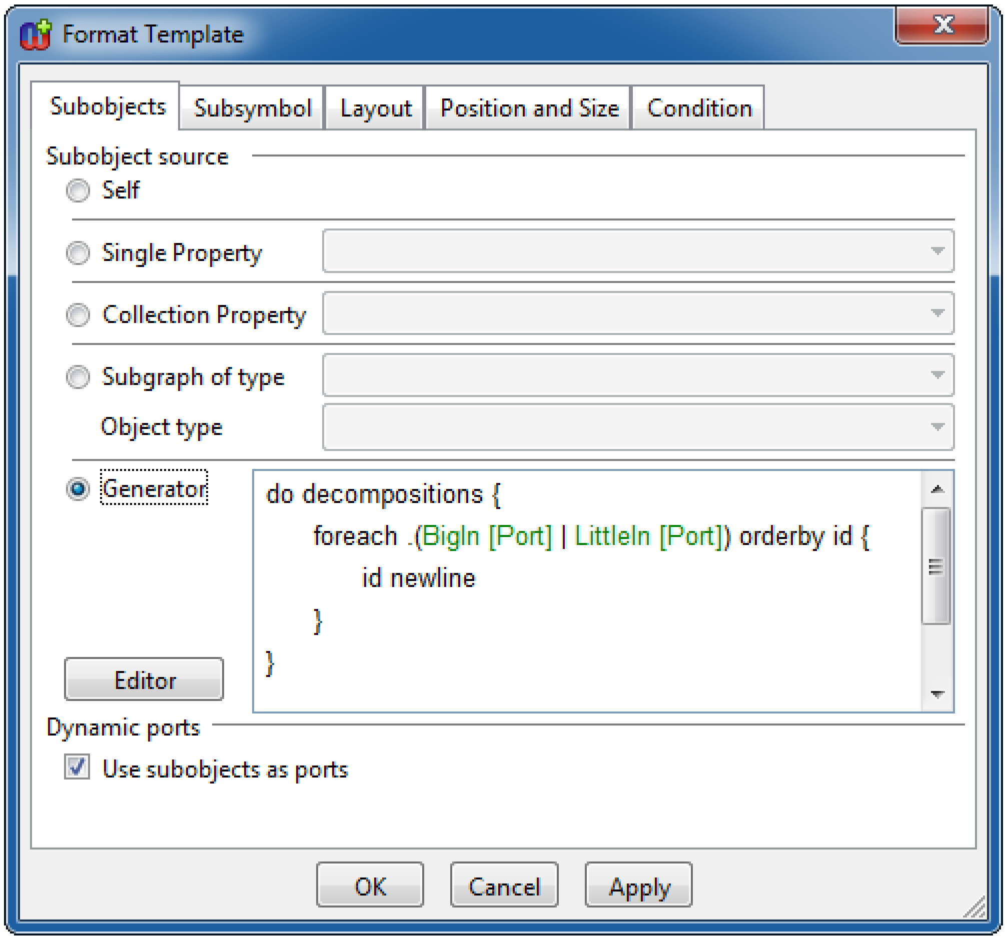


Subobject source
( ) Self
( ) Single Property
( ) Collection Property
(●) Subgraph of type: InOut Diagram
Object type: _AbstractIn [Port]
( ) Generator

Dynamic ports
[✓] Use subobjects as ports

This is a valid and workable definition but it will return all **BigIn**s before the **LittleIn**s – the default order in MetaEdit+ is to sort by type then by name.

Since we want to retrieve our subobjects sorted by their name, regardless of type, we will fetch the objects with a generator:



[✓ Generator]

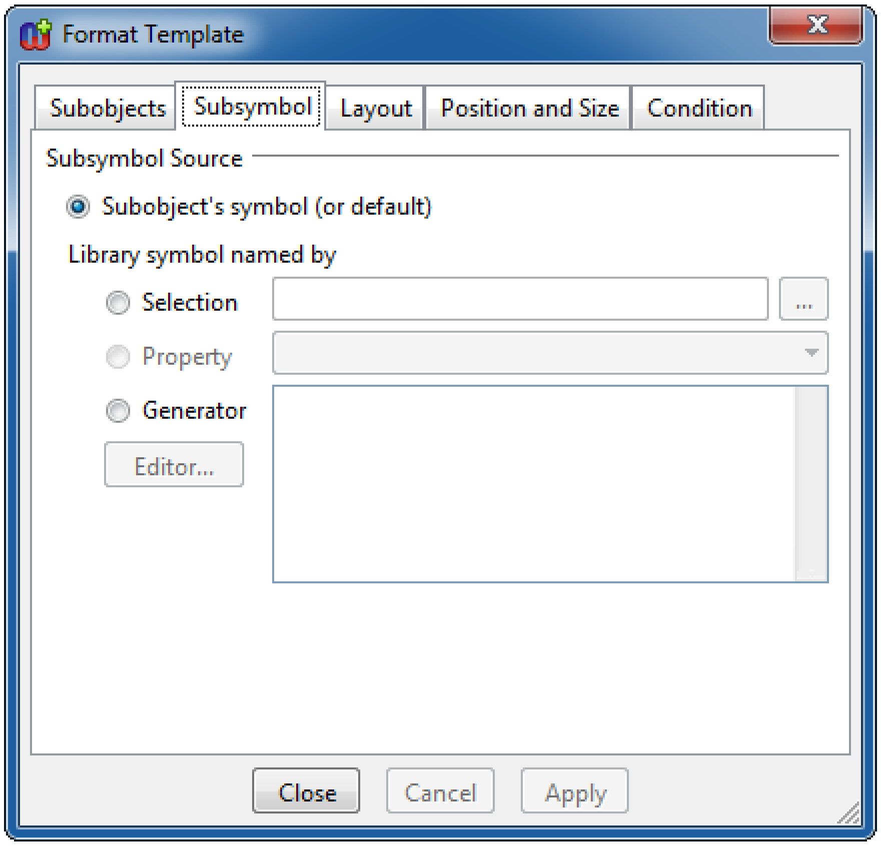
```
do decompositions {
    foreach .(BigIn [Port] | LittleIn [Port]) orderby id {
        id newline
    }
}
```

Dynamic ports

[✓] Use subobjects as ports

Please note the **Use subobjects as ports** check box at the bottom of the dialog. Enabling this means that the subobjects are allowed to become dynamic ports for the roles to connect to. If the check box is left unchecked, plain subsymbols will be displayed without the port behavior.

In addition to the objects for the subsymbols, we can also further define the look for the subsymbol itself. We can use either the default symbol of our subobject or we can get a named symbol from the symbol library. This can be chosen on the Subsymbol page in the template's format dialog. In this example we will stick with the default symbol, so our definition will look like this:



Subsymbol Source

[✓ Subobject's symbol (or default)]

Library symbol named by

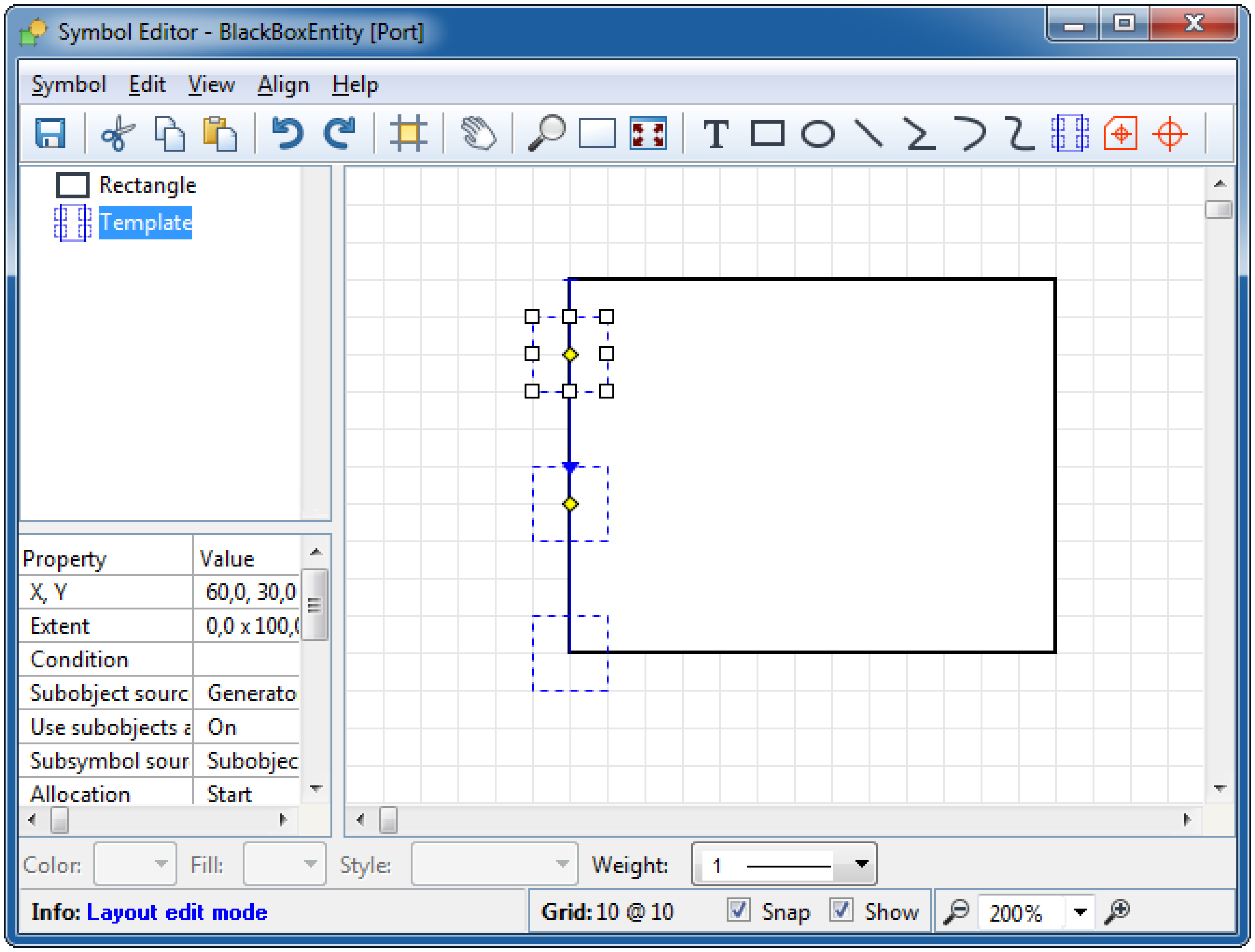
Selection

Property

Generator

In our example the default symbols of subobjects come in different sizes. **BigIn** and **BigOut** have 25x25 px symbols whereas the little ones are 15x15 px. The default subsymbol slot size is 20x20 px, which means that the bigger symbols would be displayed cropped: the slot perimeter also sets the clipping area. Also, the slots are now center aligned with the template path and we want the subsymbol to be completely outside the edge of the host object's rectangle.

To edit the template's layout, select **Edit Layout** from its pop-up menu. This will enable the layout editing mode:


Symbol Editor - BlackBoxEntity [Port]
Symbol Edit View Align Help
Rectangle
Template
Color:
Fill:
Style:
Weight:
1
Snap
Show
200%

Property	Value
X, Y	60,0, 30,0
Extent	0,0 x 100,(
Condition	
Subobject sourc	Generato
Use subobjects a	On
Subsymbol sour	Subobjec
Allocation	Start

Info: Layout edit mode

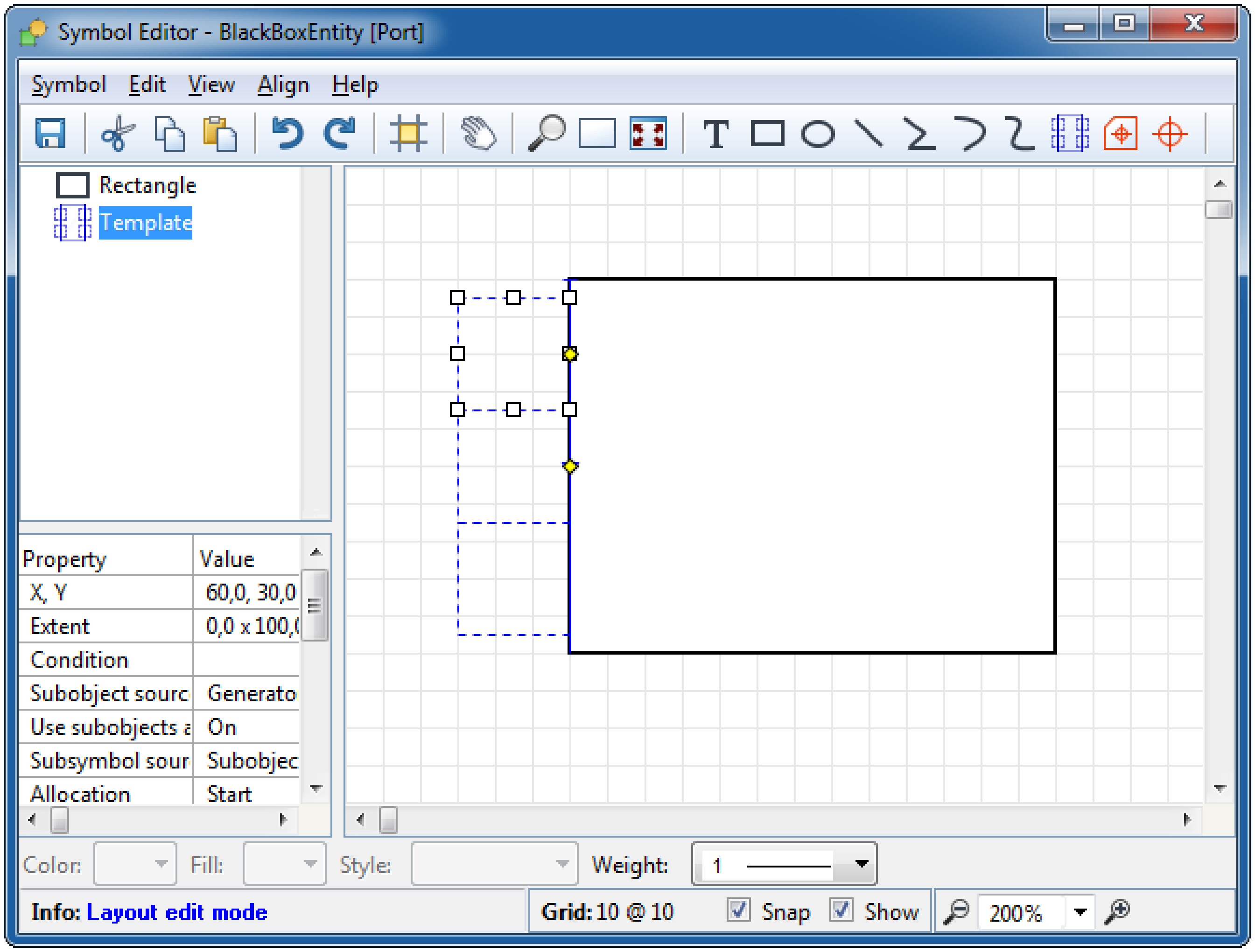
Grid: 10 @ 10

Edit Layout mode reveals several handles that you can use for direct manipulation:

- The first (top-most) yellow handle defines the distance between the start of the path and the first slot position on the path
- The second yellow handle defines the distance or interval between the slots
- The white handles around the first slot are used to define the size of the slot
- By dragging a point within the first slot, you can set the alignment of the slot relative to its position on the path

Moving these handles will affect the positioning of all the subsymbol slots *en masse*. So, drag the first slot left so it is completely on the outside edge. From its pop-up menu choose **Format** and on the Layout tab set **Max Width** and **Max Height** to 30. Move the lower yellow handle up so there are no gaps between the subsymbol slots.

You should now have a template that looks like this:

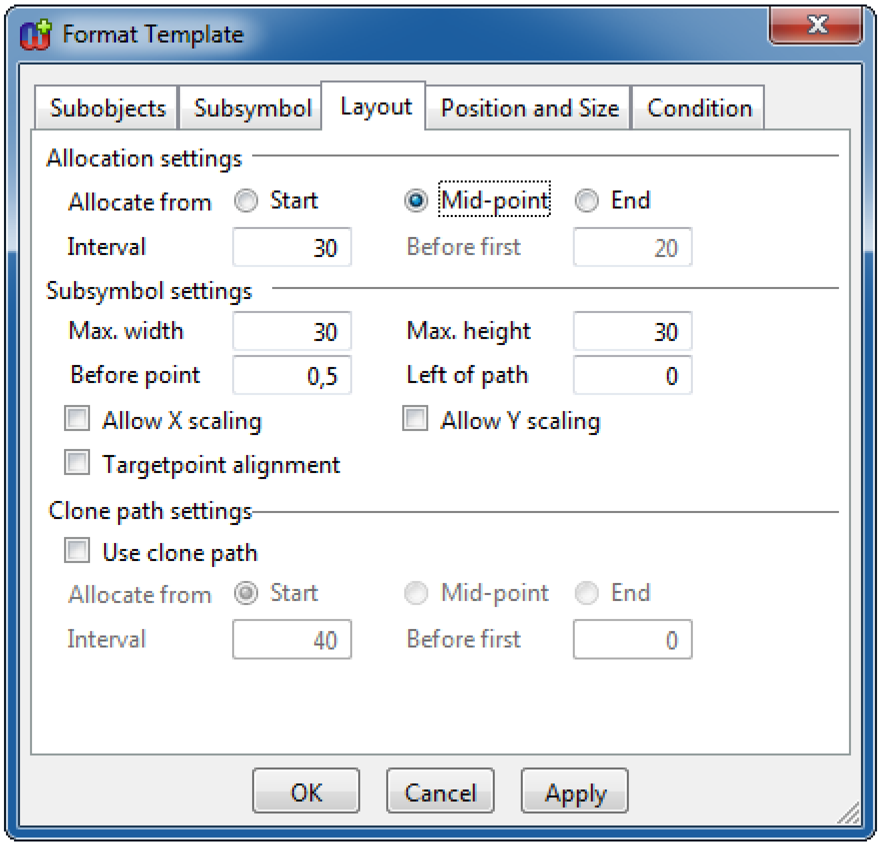

Symbol Editor - BlackBoxEntity [Port]
Symbol Edit View Align Help
Rectangle
Template

Property	Value
X, Y	60,0, 30,0
Extent	0,0 x 100,(
Condition	
Subobject sourc	Generato
Use subobjects a	On
Subsymbol sour	Subobjec
Allocation	Start

Color: Fill: Style: Weight: 1

Info: Layout edit mode Grid: 10 @ 10 ☑ Snap ☑ Show 200%

Finally, from the Format dialog's Layout page, set **Allocate from** to **Mid-point**. You can also check the rest of your settings from this page:


Format Template

Subobjects | Subsymbol | Layout | Position and Size | Condition

Allocation settings

Allocate from: Start [✓ Mid-point] End

Interval: 30 Before first: 20

Subsymbol settings

Max. width: 30 Max. height: 30

Before point: 0,5 Left of path: 0

☐ Allow X scaling ☐ Allow Y scaling

☐ Targetpoint alignment

Clone path settings

☐ Use clone path

Allocate from: [✓ Start] Mid-point End

Interval: 40 Before first: 0

OK Cancel Apply

The topmost section, **Allocation settings**, defines how slots are positioned on the path:

- **The allocation order**, i.e. shall we start distribution from the beginning, middle or end (in the mid-point case we spread from the middle outwards)
- **Interval**, i.e. the distance between the slots
- **Before first**, i.e. the distance from allocation start to the first slot

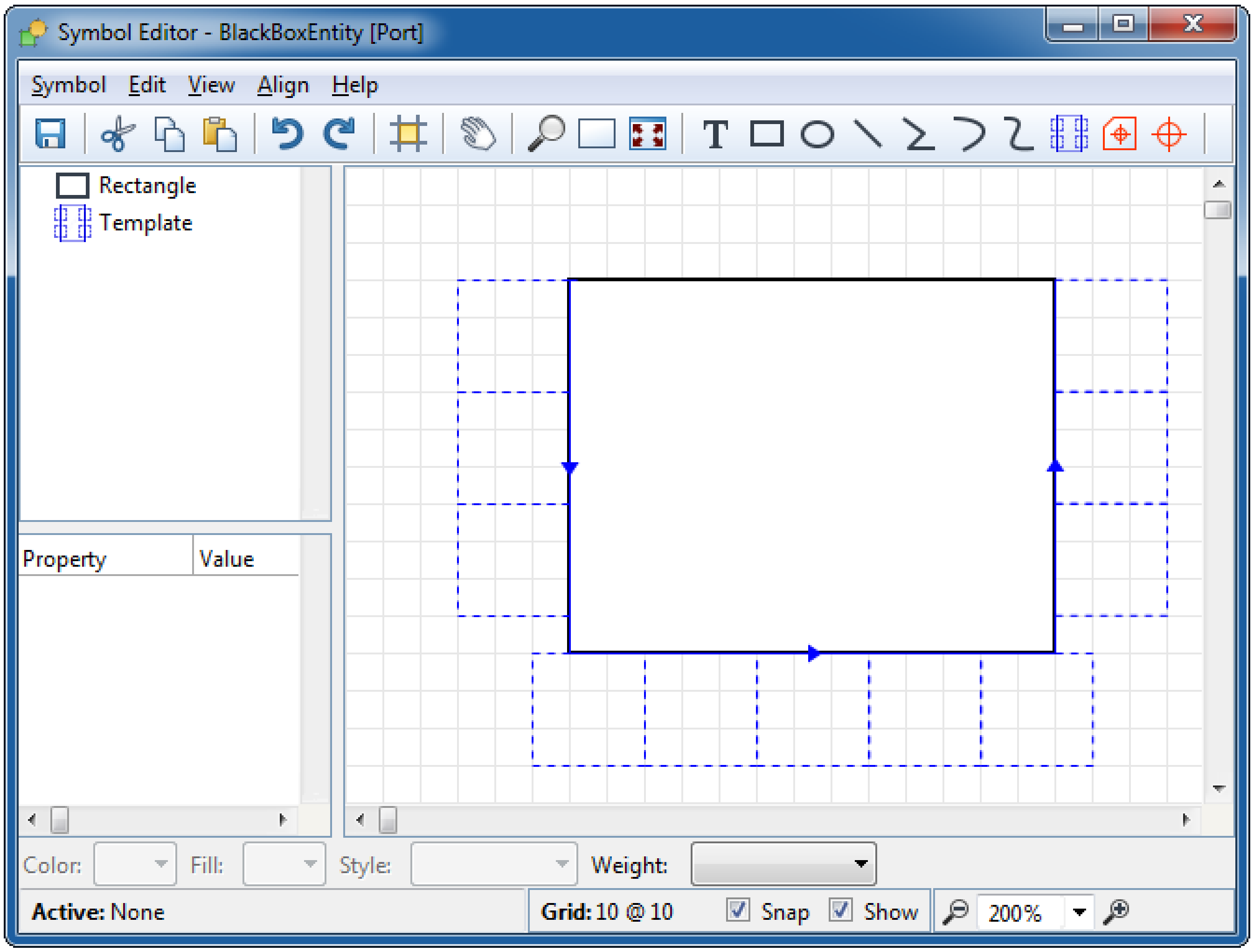
The middle section contains the settings for the subsymbol slot:

- **Max. width** and **Max. height**: the maximum width and height of the slot area
- **Before point** and **Left of path**: the alignment of the subsymbol slot. The value is the fraction of the subsymbol slot that is respectively before or to the left of its allocated point, from the perspective of one moving along the template path segment in the direction of its arrowhead.
- **Allow X** and **Y scaling**: whether the subsymbol is allowed to scale with the hosting object
- **Targetpoint alignment**: when this is set, the subsymbol will be aligned so that its default connectable's target point will be in that point around which the subsymbol slot is aligned (i.e. the first yellow handle)

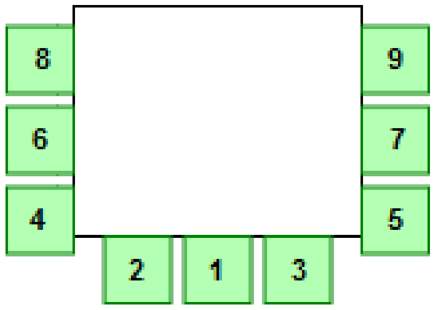
The bottom section remains disabled now, for more information please refer to Section 4.3, Button Layout.

As a side note, the subsymbol slot's alignment with the template path is not absolute but interpreted relative to the direction of individual path segments. In our case, the fraction of the subsymbol slot to the Left of Path is 0, so as we travel down the template path in the direction of its arrowhead, none of the subsymbol slot is to our left, i.e. none is inside the big rectangle.

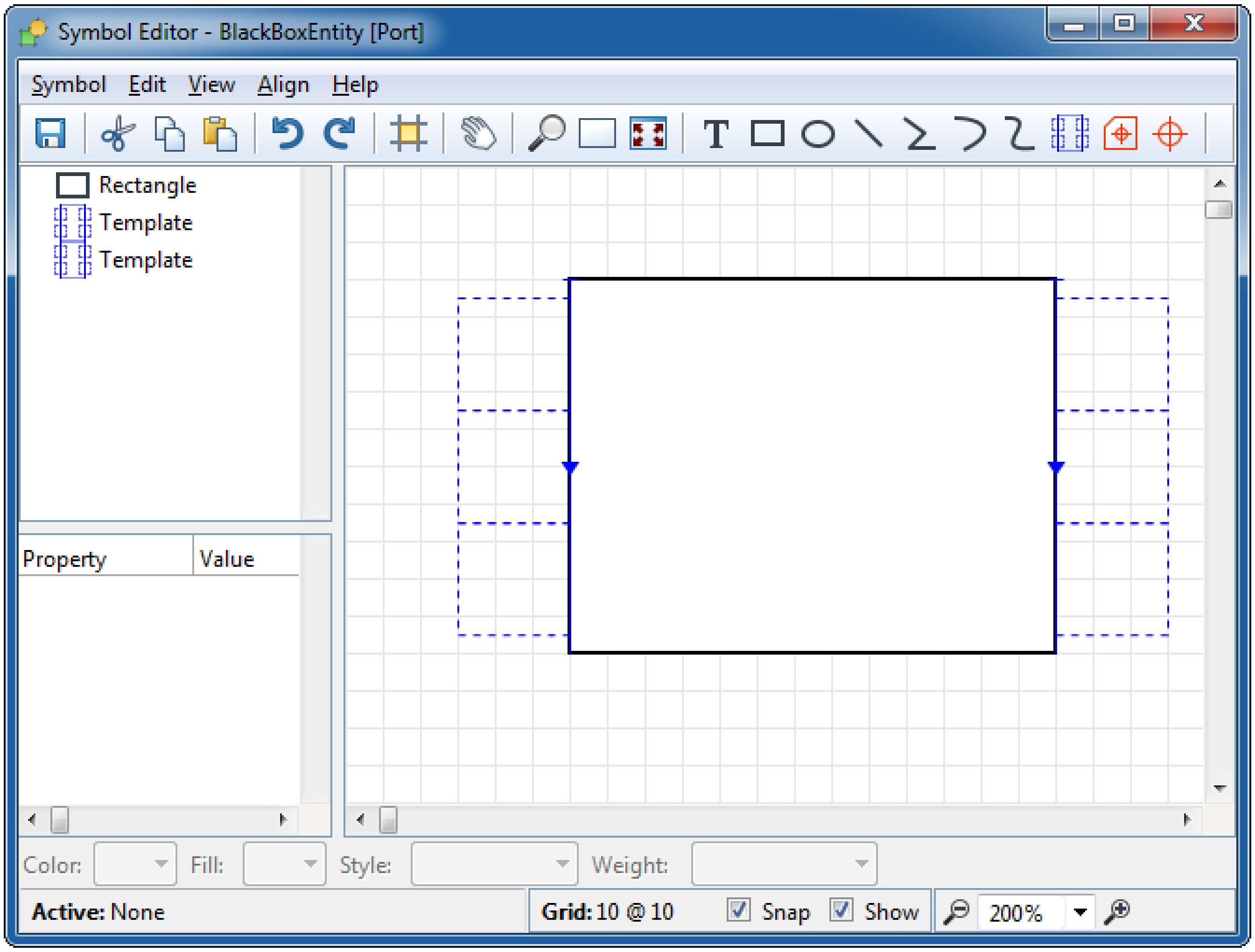
If we extend the path to carry on around the bottom and right edges of the big rectangle, our current alignment setting would look like this:



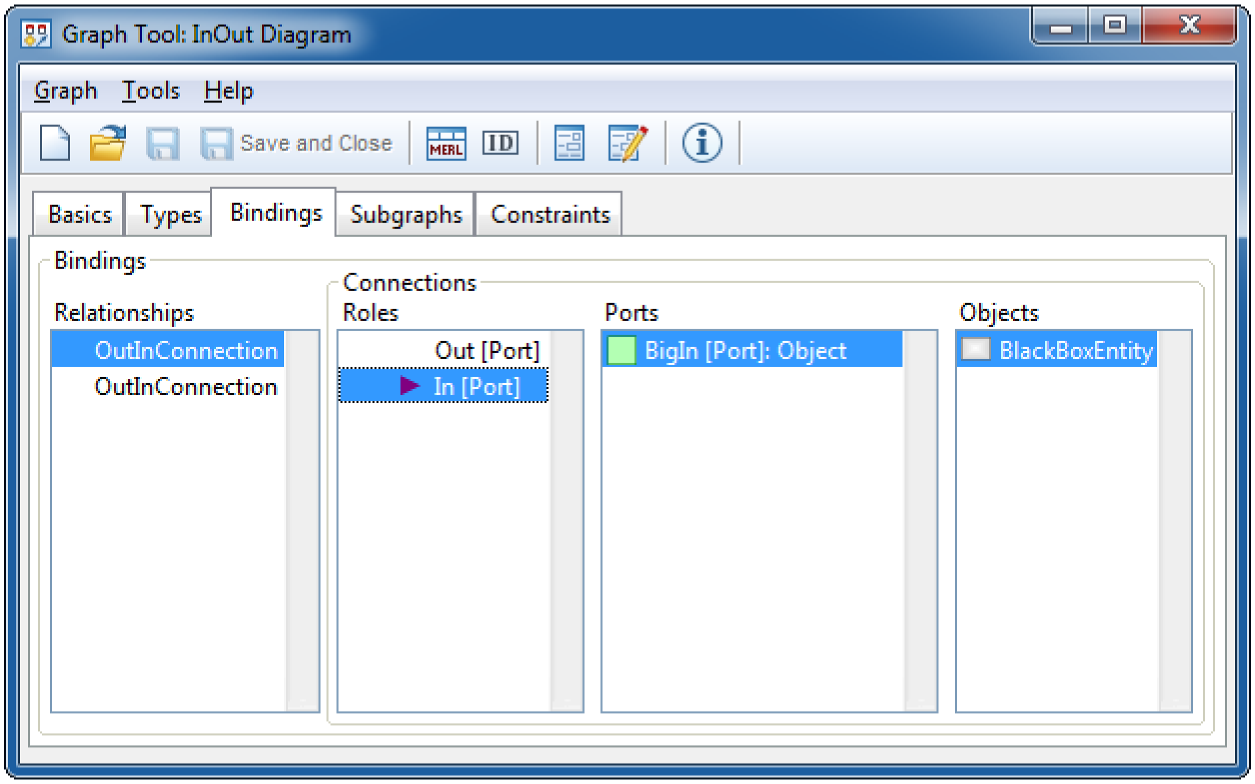
As you can see, because the alignment is defined relative to the path segment, the subsymbols all stay neatly on the outside edge of the big rectangle, which is indeed what we would want. In the Diagram Editor the actual subsymbols would thus be laid out like this:



For now we will, however, settle with our original example, having an individual template for the In ports on the left of the host object's rectangle. We can then go on to create a similar template for the Out ports on the right of the host object's rectangle, e.g. by copying, pasting and editing the first template:

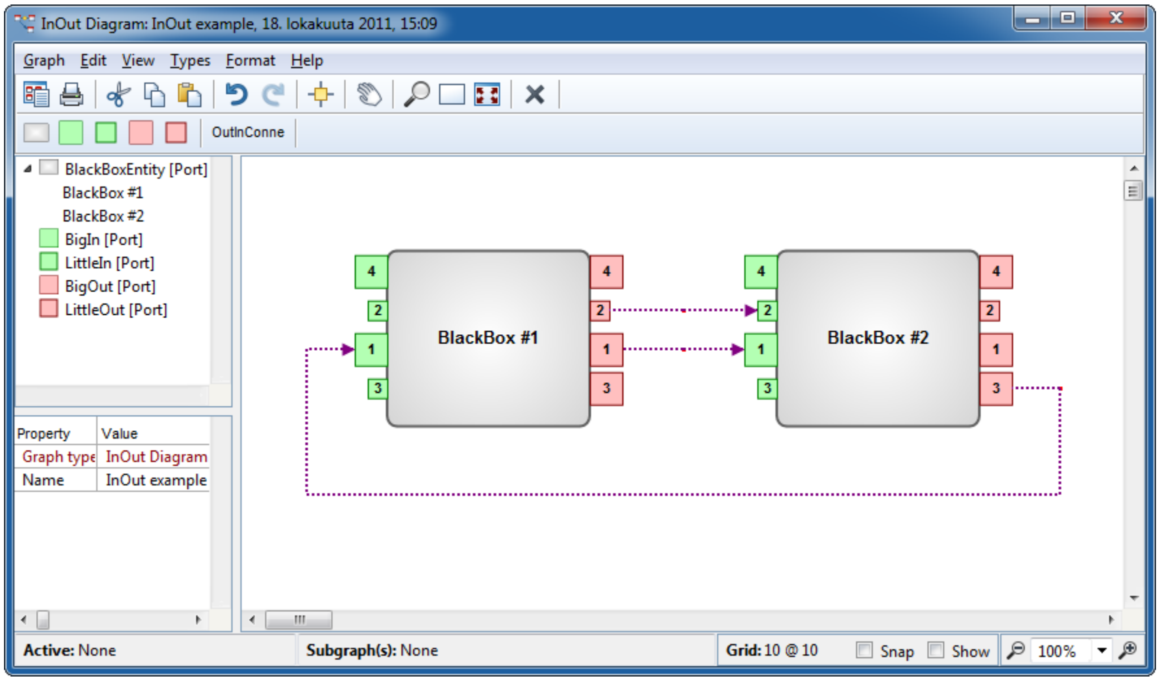


We have now defined how the subsymbols are fetched and displayed as part of the host objects. We have also allowed the subobjects to act as dynamic ports for incoming and outgoing connections. What remains to be done is to define bindings to use the subobjects as ports. This is similar to defining the normal static ports in a type binding definition in Graph Tool:



When attaching ports to a binding definition in the Graph Tool, MetaEdit+ will ask whether to use a traditional static port or a dynamic port. If a dynamic port is chosen, MetaEdit+ will provide a list of suitable object types to choose from. The selected type will then appear on the port slot of binding definition.

With the binding definition in action (and some fine tuning of symbols), we now have a metamodel that enables us to create component models in hierarchical black box fashion, as specified our original use case:



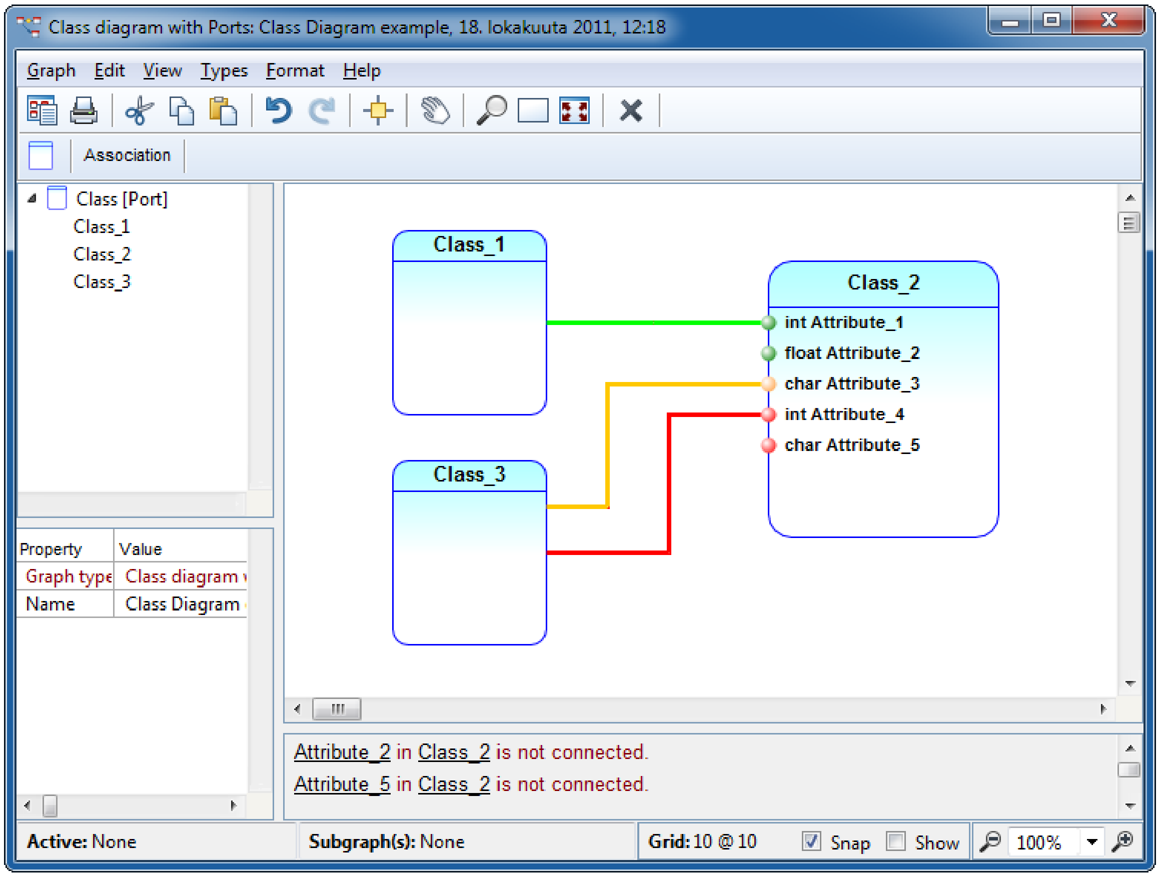
Property	Value
Graph type	InOut Diagram
Name	InOut example

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

With this example we have also walked through the basic tasks of template creation and definition of dynamic ports. In the next two sections we will look at a few variations of the same theme.

4.2 Class Diagram with Enhanced Visualization

As for another example of when and how to apply templates and dynamic ports, let us consider the following use case. We have a typical class diagram that describes a set of classes and their respective attributes. We want to enhance the diagram by forcing the association relationship to connect visually exactly to a certain attribute and retain that visualization even if the attribute's class symbol is moved or scaled. Furthermore, we also want a visual cue, like a color in this case, about the access level of each attribute. Finally, we want to display the connecting association with the same color, too. After these changes our diagram could look like this (the example can be found as Class Diagram example in the Demo database's 'Port example'):



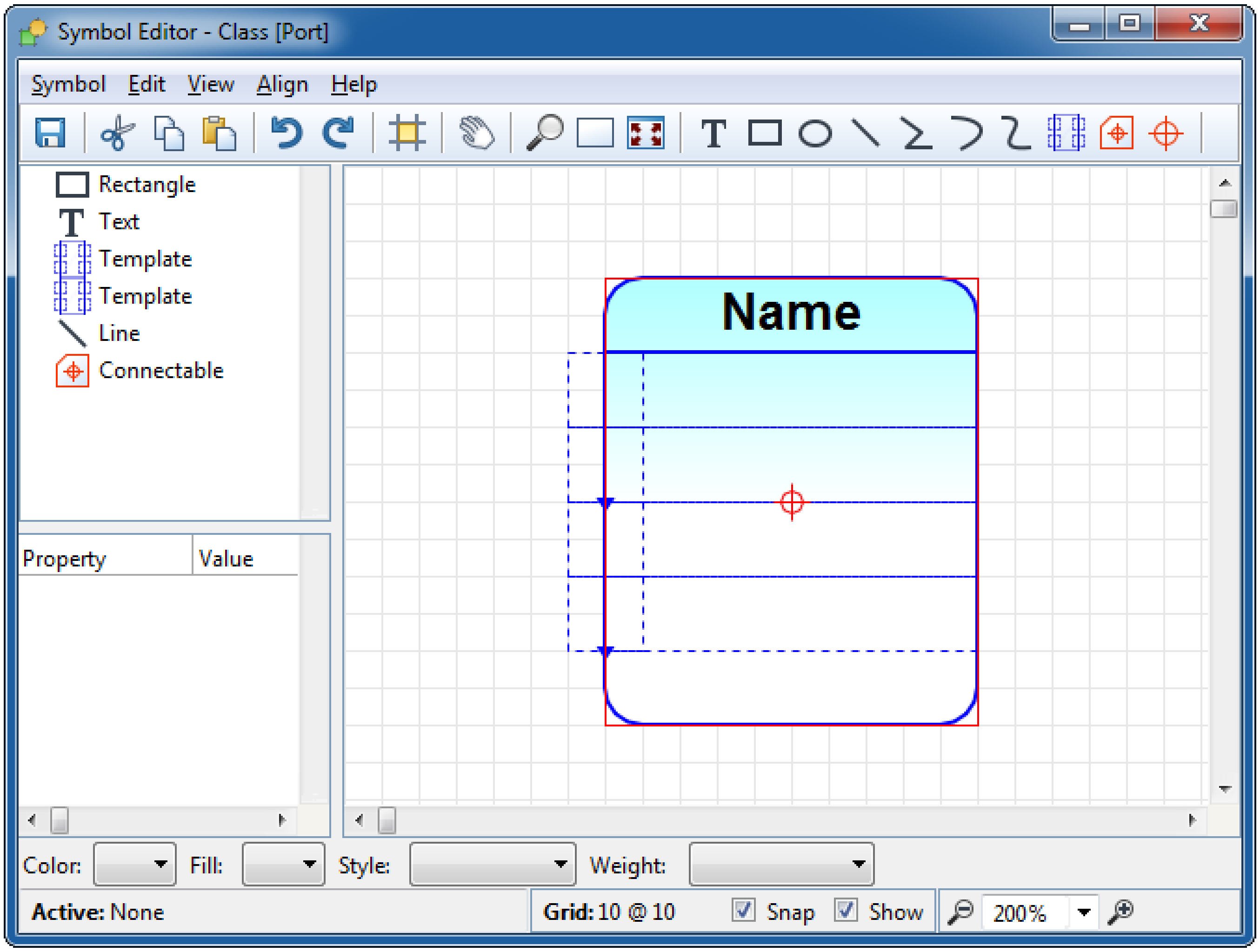
Property	Value
Graph type	Class diagram w
Name	Class Diagram

Attribute_2 in Class_2 is not connected.
Attribute_5 in Class_2 is not connected.

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

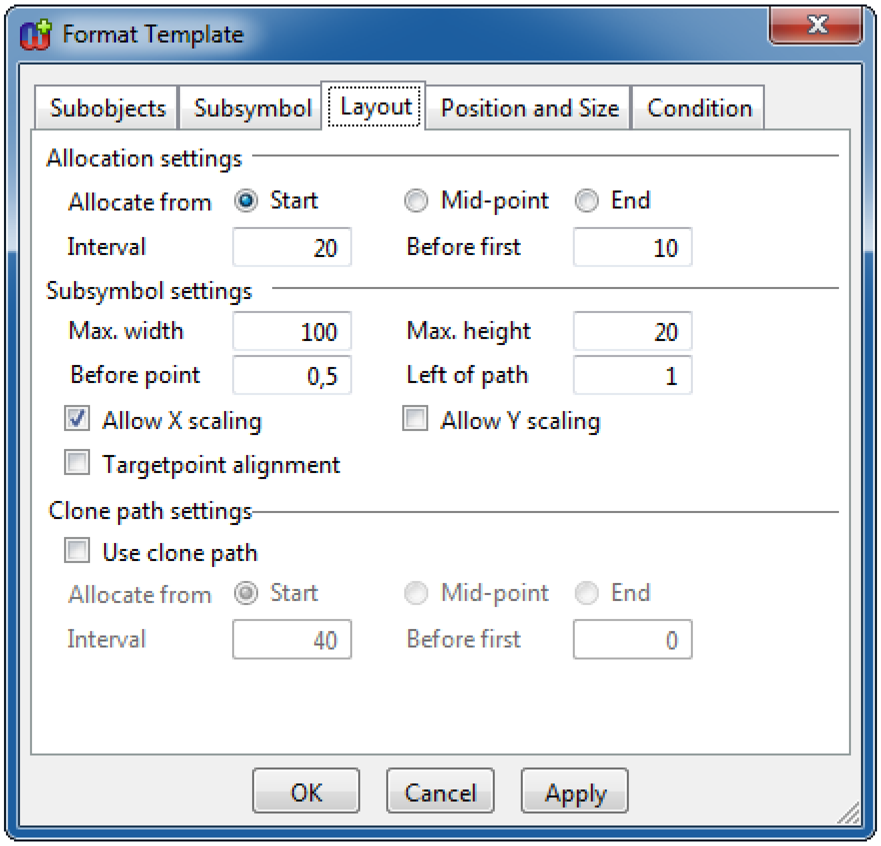
In the diagram above, green denotes the attribute being public, yellow means it is protected and red stands for private. In theory we shouldn't connect associations to private attributes, but let's agree for now what the red color provides a nice visual alarm that the connection is wrong. There are other problems with the semantics here, but then this is only an example of ports, not an attempt to add real attribute association semantics to UML.

Let us look at how the class symbol has been defined:



This example illustrates a situation where we are using two overlaid templates to provide a complex visualization: the template on the right takes care of displaying the normal list of attributes with their names etc., whereas the template on the left is used to display the cues about the access levels. Both templates use the same method to fetch the subobjects, collecting them from a collection property.

The template on the right uses the default symbol as subsymbol but does not allow the respective subobject to become a dynamic port. Its layout dialog page looks like this:



Allocation settings

Allocate from [✓ Start] Mid-point End

Interval 20 Before first 10

Subsymbol settings

Max. width 100 Max. height 20

Before point 0,5 Left of path 1

[✓] Allow X scaling [ ] Allow Y scaling

[ ] Targetpoint alignment

Clone path settings

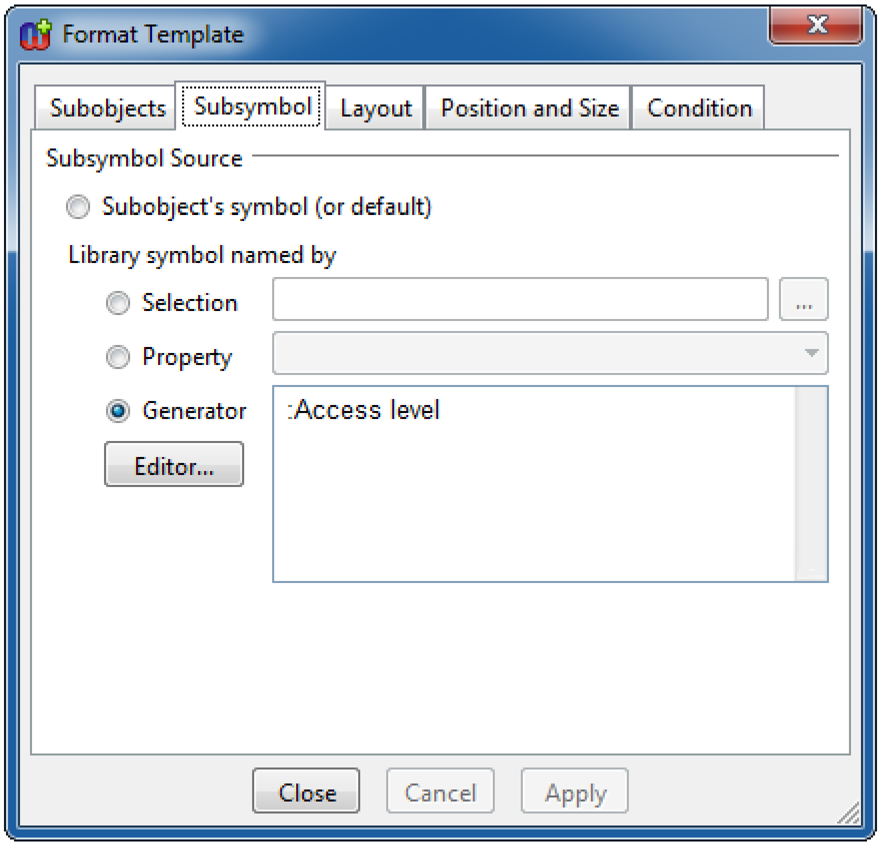
[ ] Use clone path

Allocate from [✓ Start] Mid-point End

Interval 40 Before first 0

What is noteworthy here is that this time we will allow the subsymbol to scale horizontally. This means that when the host object is scaled, the width of the subsymbol will grow with the host width (i.e. more space to display each attribute’s name). On the vertical axis, however, the height of each subsymbol slot remains constant, so if the host scales vertically, more subsymbol slots will be able to fit in to the host symbol (i.e. more attributes can be displayed in the list).

The template on the left retrieves its subsymbols from the symbol library according to a generated name, the value of the :Access level property:



Subsymbol Source

( ) Subobject's symbol (or default)

Library symbol named by

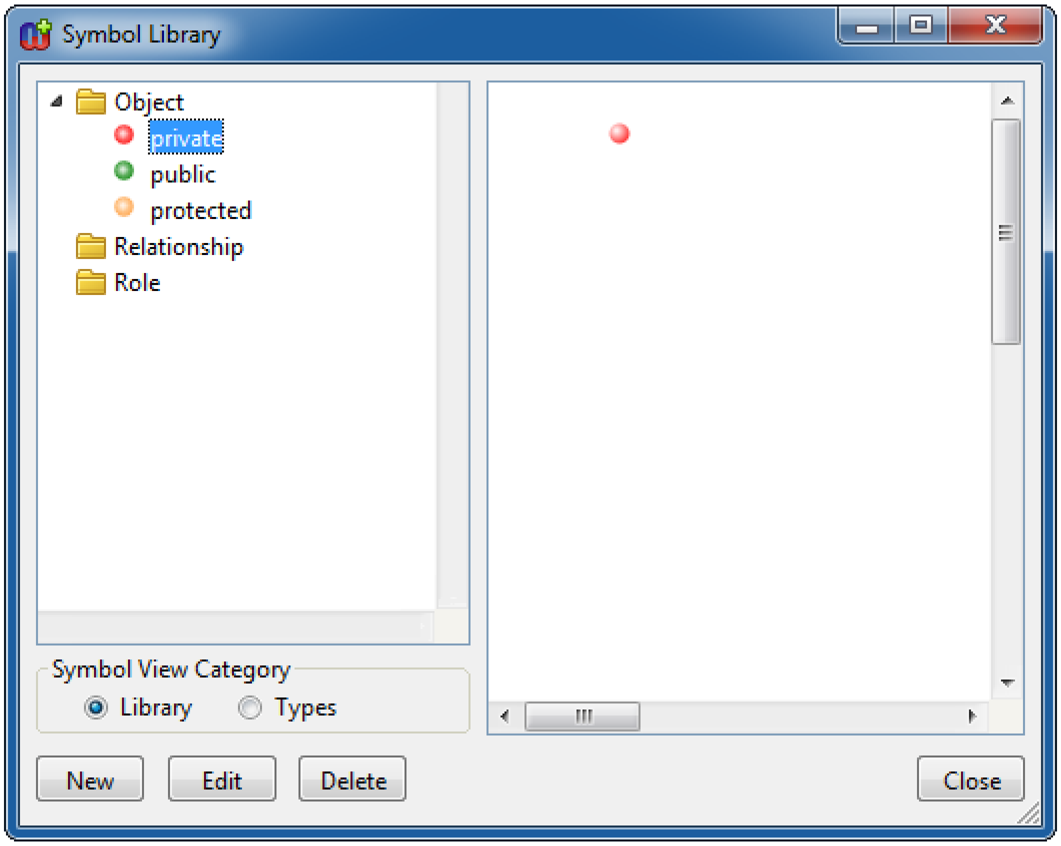
( ) Selection

( ) Property

(●) Generator

:Access level

Since the values can be 'public', 'protected' or 'private', subsymbols will be fetched by those names from the symbol library (**Metamodel** | **Symbol Browser**):

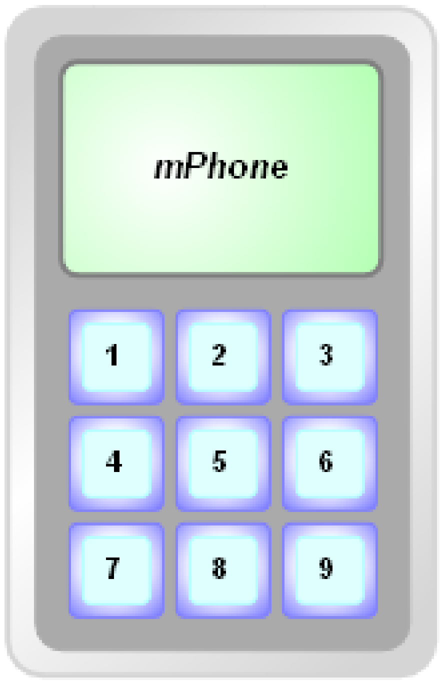


With these definitions we now have two templates: one takes care of the normal displaying of the attribute list and the other provides the access level cues and the connection ports.

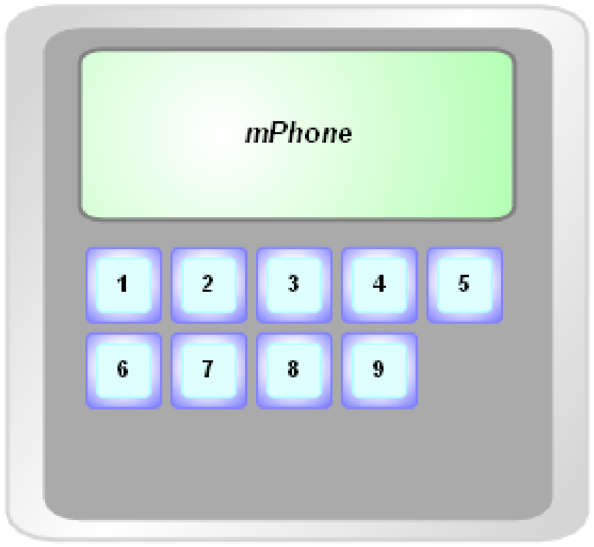
This example also includes the feature of role lines adapting their visualization according to their connections – for more information about how this is achieved, please see Section 5, Conditional Role lines.

4.3 Button Layout

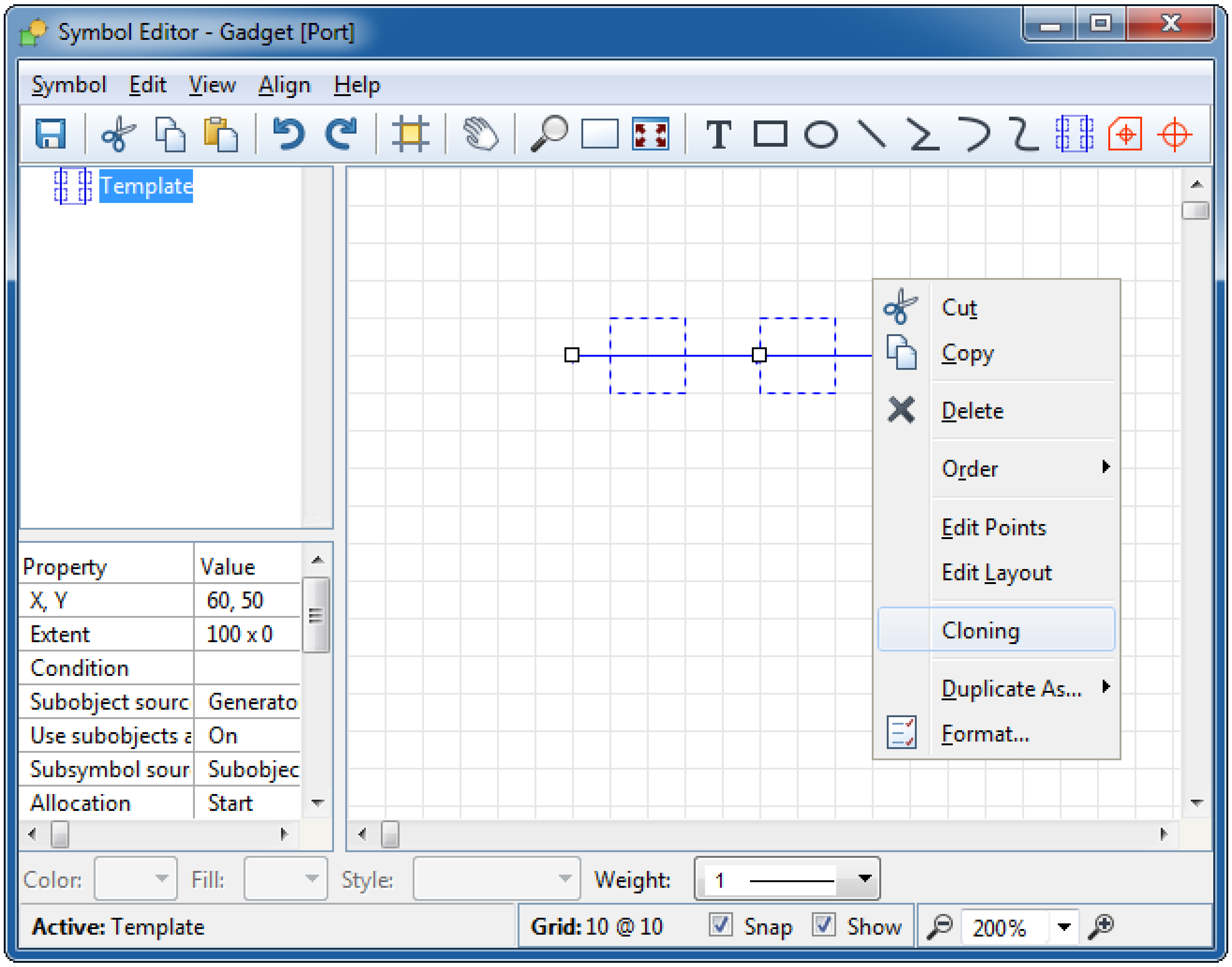
So far, our subsymbols have only been allocated along a single path, albeit one that can have multiple segments. The third template example describes the way to define a grid-like layout for the subsymbols. A typical use case for this is presented in the Gadget example diagram that can be found in the 'Port example' project in the Demo repository:



In this example the numeric buttons have been defined as individual objects stored in a collection property in the 'mPhone' Gadget object. When 'mPhone' is scaled, the layout will adjust itself to fit the buttons in the new area according to the layout parameters:



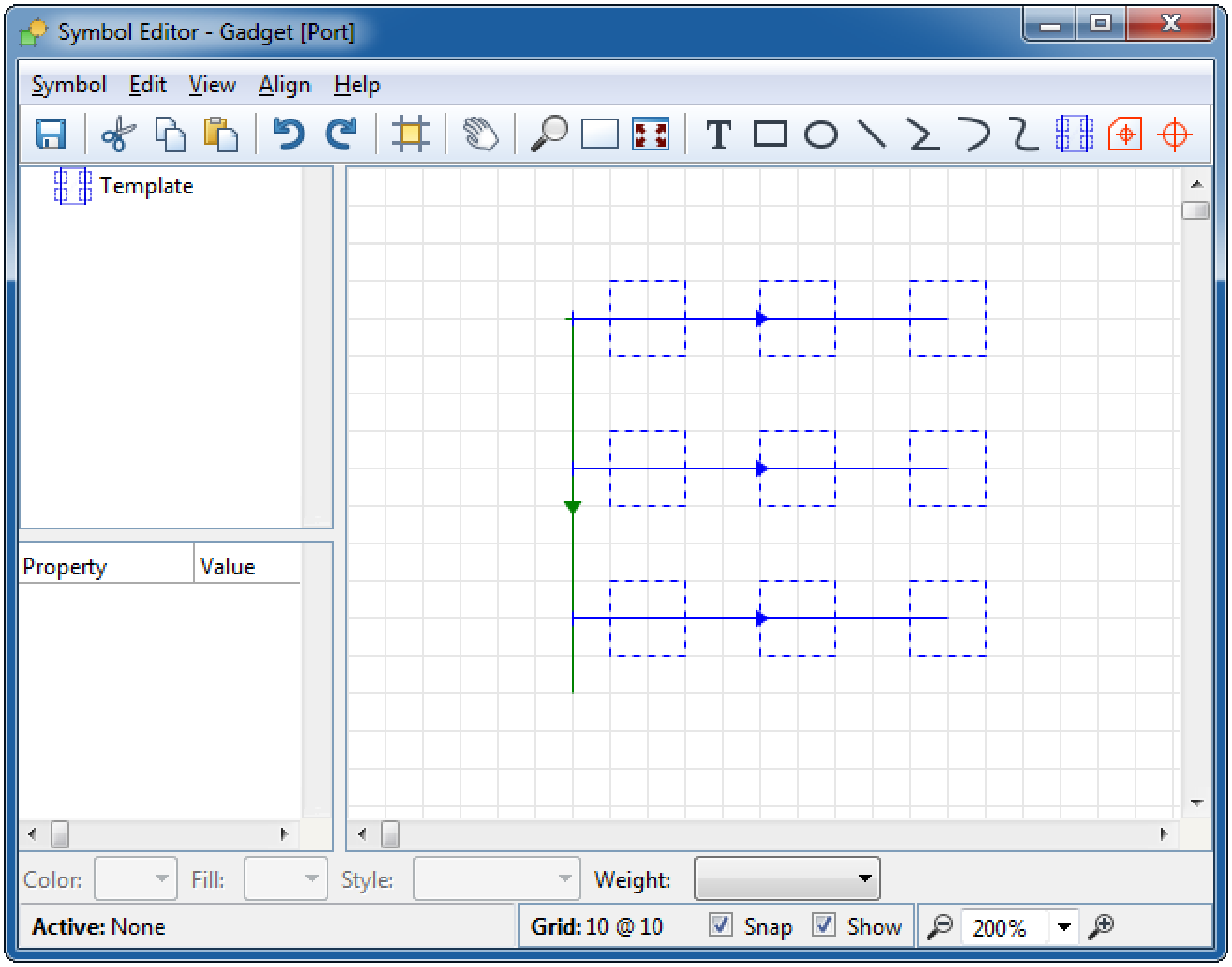
To create this kind of grid-like distribution, we need to use the template's cloning functionality. In the Symbol Editor, first draw a normal template path. Keep it selected and choose **Cloning** from its pop-up menu:



Property	Value
X, Y	60, 50
Extent	100 x 0
Condition	
Subobject sourc	Generato
Use subobjects a	On
Subsymbol sour	Subobjec
Allocation	Start

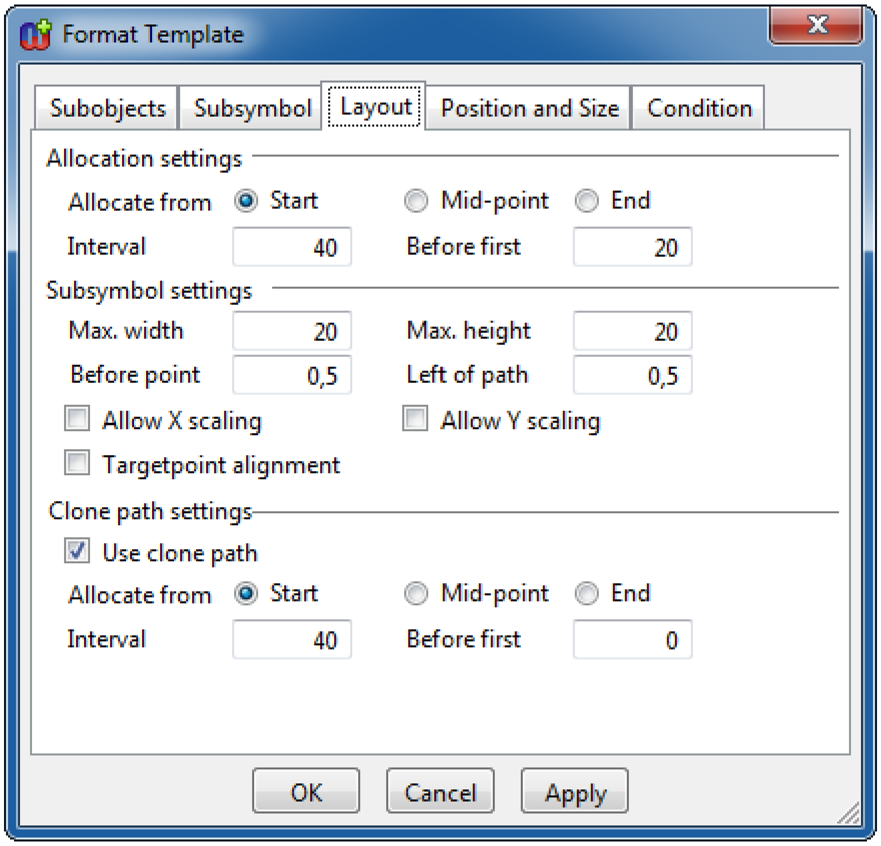
Active: Template **Grid:** 10 @ 10 ☑ Snap ☑ Show

The template will now show a new green cloning path along which several template paths have been distributed:



Active: None | **Grid:** 10 @ 10 | ☑ Snap | ☑ Show | 200%

Defining the retrieval of subobjects and subsymbols works in a similar way to non-cloning (i.e. non-grid) layouts but there are now more options available at the bottom of the layout format dialog:



Allocation settings

Allocate from [✓ Start] Mid-point End

Interval 40 Before first 20

Subsymbol settings

Max. width 20 Max. height 20

Before point 0,5 Left of path 0,5

☐ Allow X scaling ☐ Allow Y scaling

☐ Targetpoint alignment

Clone path settings

☑ Use clone path

Allocate from [✓ Start] Mid-point End

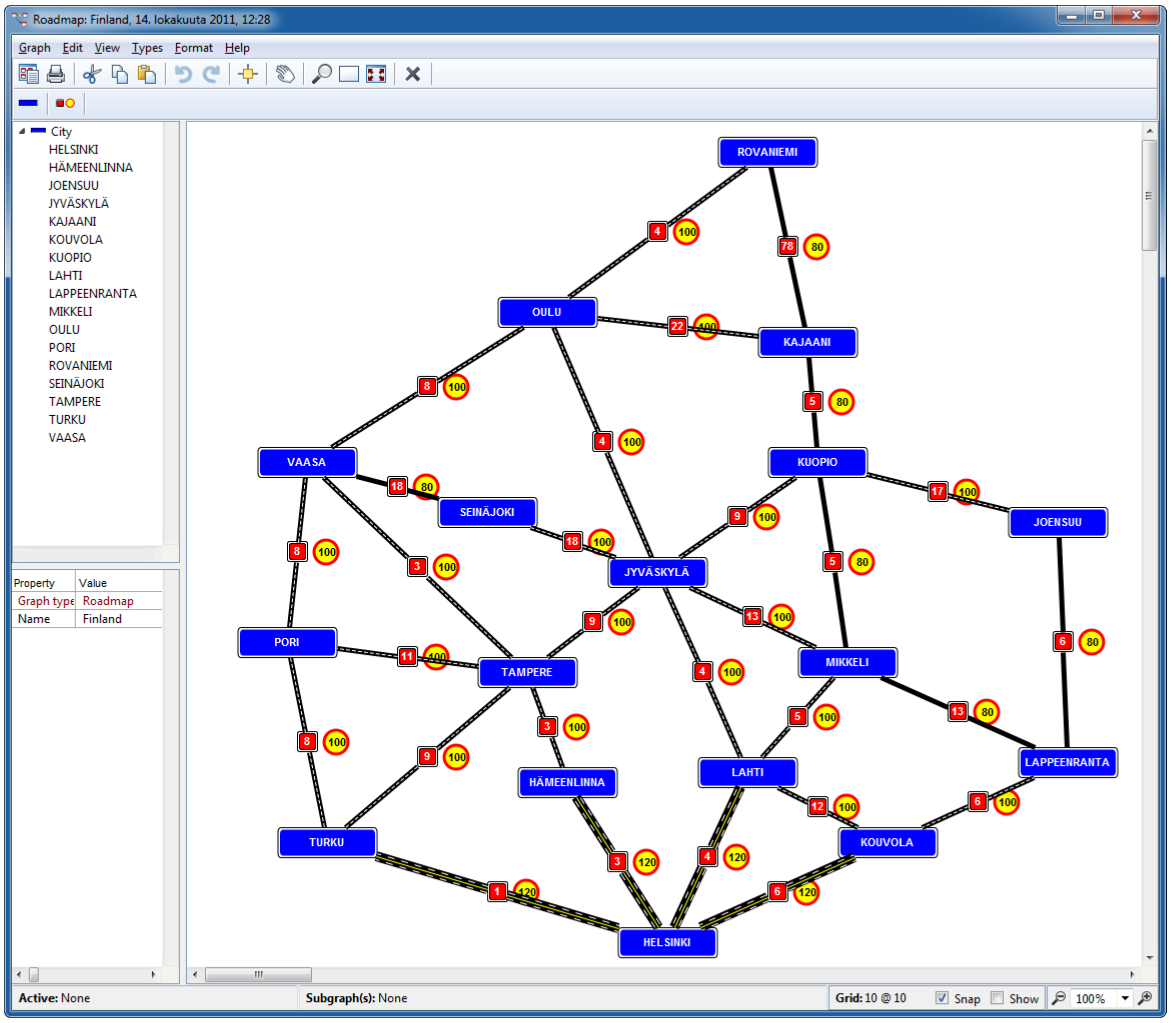
Interval 40 Before first 0

The clone path settings behave the same way as the template's general allocation settings in the upper part of the dialog. Whereas each position along a normal (blue) template path represents where a subsymbol will be placed, each position along the cloning (green) path represents where the whole normal path will be cloned to. The normal paths are cloned only as necessary: each normal path is filled first, and if there are still subobjects, the normal path is cloned and placed at the next point on the cloning path.

We can toggle the cloning on or off, set the allocation basis (start, middle or end) and define the interval and distance from the start (except in the mid-allocation). In Edit Layout mode, yellow handles appear for settings these distances on the cloning path. In Edit Points mode, standard handles appear for each cloning path segment.

5 Conditional Role lines

Another new feature of MetaEdit+ 5.0 are the conditional role lines that provide a powerful way to add visual semantics in the models. As an example of the possibilities of this new feature, let us look at the diagram below:

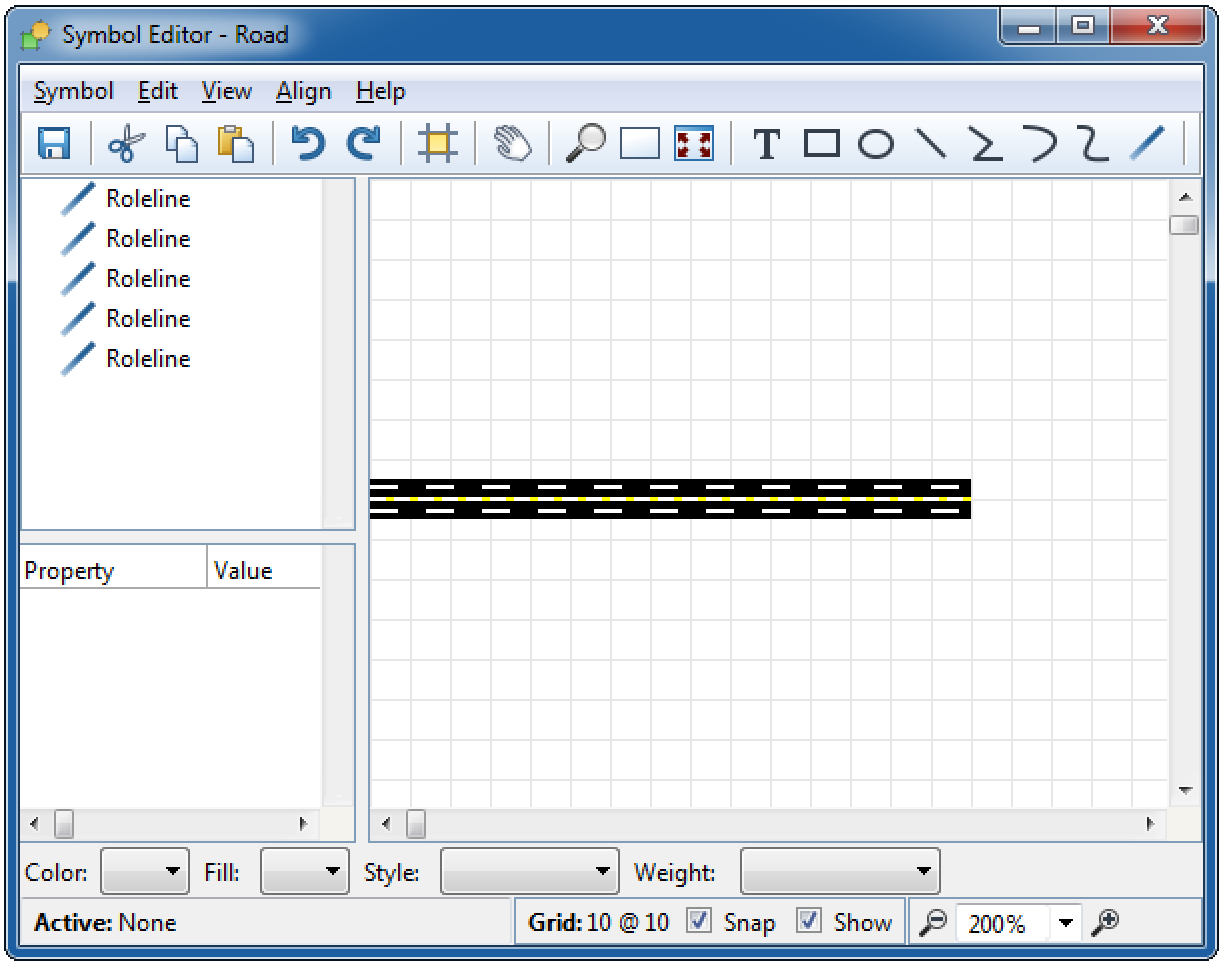


Property	Value
Graph type	Roadmap
Name	Finland

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

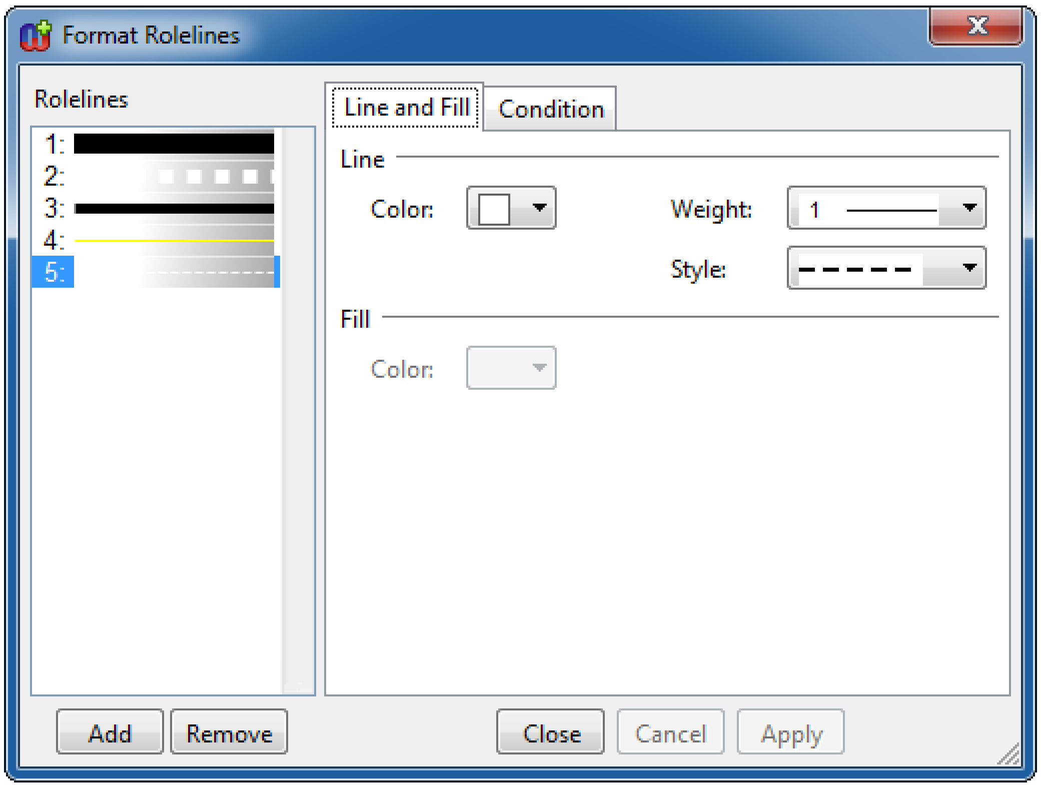
This diagram presents a traffic network between various cities and towns (and is available in the Demo repository in the ‘Roadmap’ project). The modeling language has been build so that when a new road is created between two cities, it automatically adjusts its representation according to the size of the cities it connects. If the joint population of the cities is more than 500,000, we can expect the traffic volume to require a wider road with the speed limit of 120 km/h, whereas the connections between smaller cities and towns will suffice with smaller roads and speed limits of 100 and 80 km/h. The visualization is now built dynamically according to the joint population calculation.

In order to see how this can be done, open the symbol editor for the **Road** role type:



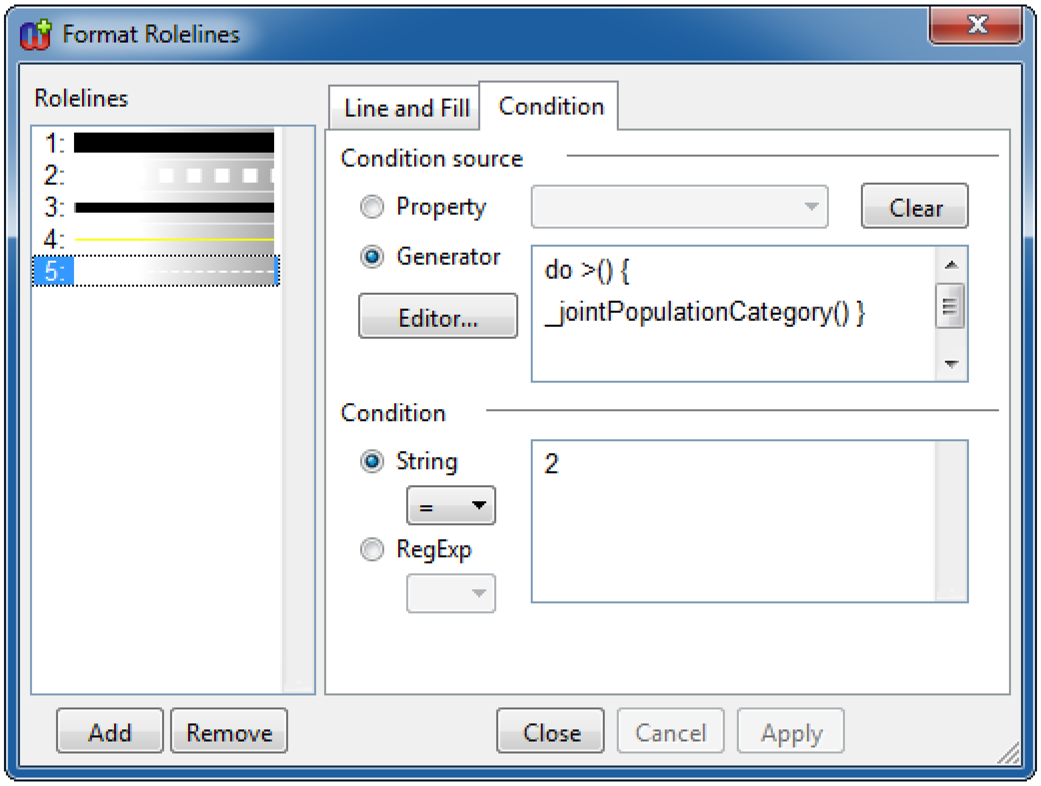
As we see, we now have a set of role line definitions on top of each other in the editor. Their color and line style can be edited in the role line format dialog that can be opened either by double-clicking any role line or choosing **Format…** from its pop-up menu, or by selecting **Edit | Format Role Lines…**.

The format dialog looks like this:



All role lines in the editor are listed on the left while the right side contains the format dialog for the selected role line, where its line style and color settings can be edited. To add or remove a role line, or to change its position in the Z order, use the buttons below the Rolelines list or the list's pop-up menu.

The format dialog for an individual role line also has a Condition page:

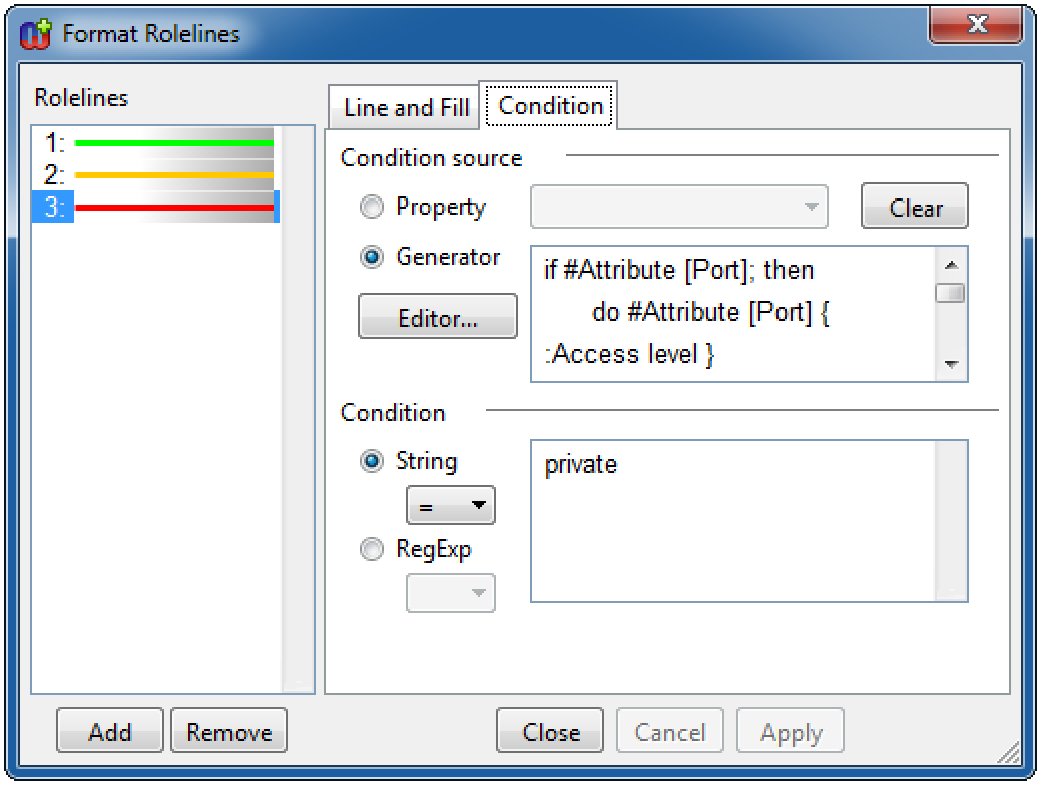


```
do >() {
_jointPopulationCategory() }
```

Condition: String [✓] = 2

On this page the user can set the visibility condition for the selected role line. The Condition dialog is similar to those of other symbol elements, i.e. the condition source can be either a property or a generator, and its value is compared with a string or a regular expression. The definition in the picture above simply states that the selected role line will be displayed if the execution of _jointPopulationCategory subgenerator returns 2. Similar conditions exists for other listed role lines as well, and the final visualization for the road segment is built up by selectively displaying the individual role lines (for example, lines #1, #2 and #4 will only be displayed if the _jointPopulationCategory is 1, but line #3 will always be displayed).

For another example, let us look back at the Class Diagram visualization example in Section 4.2. Part of the visualization in that example was the coloring of association relationship according to the access level of the attribute it connected to. That was implemented with conditional role lines in a similar fashion to the Roadmap example:

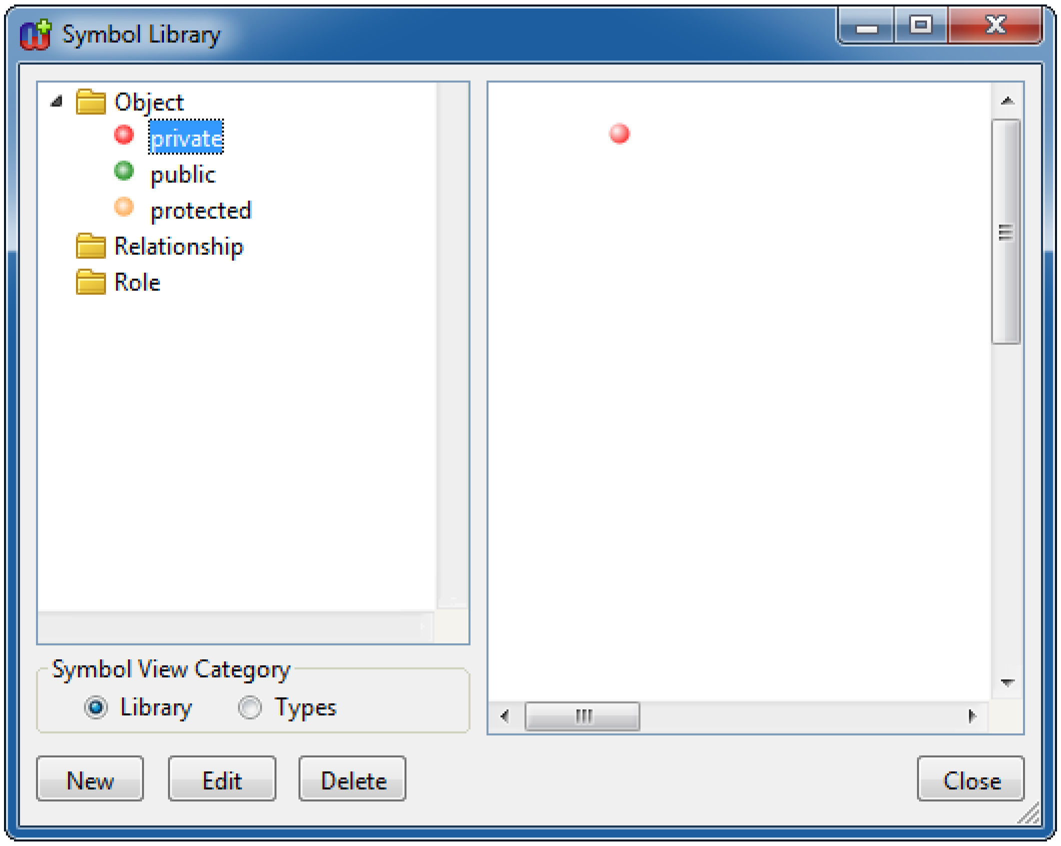


```
if #Attribute [Port]; then
    do #Attribute [Port] {
:Access level }
```

Condition: String = private

6 Symbol Browser

MetaEdit+ 5.0 also improves the Symbol Library. The first major change is that the library is now global within the database (not project-specific as before). The access and management of library and type symbols have also been made easier with the new Symbol Browser:



The Symbol Browser can be accessed from the Main MetaEdit+ Launcher (**Metamodel | Symbol Browser**), from Symbol Editors or from the template subsymbol dialog. Depending on where the browser has been opened from and for what purpose (i.e. selection, importing, saving, etc.), some of its functionality may have been disabled, but in general it provides the following options:

- Browsing symbols in library and types
- Opening symbols for editing
- Creating new library symbols
- Deleting library symbols

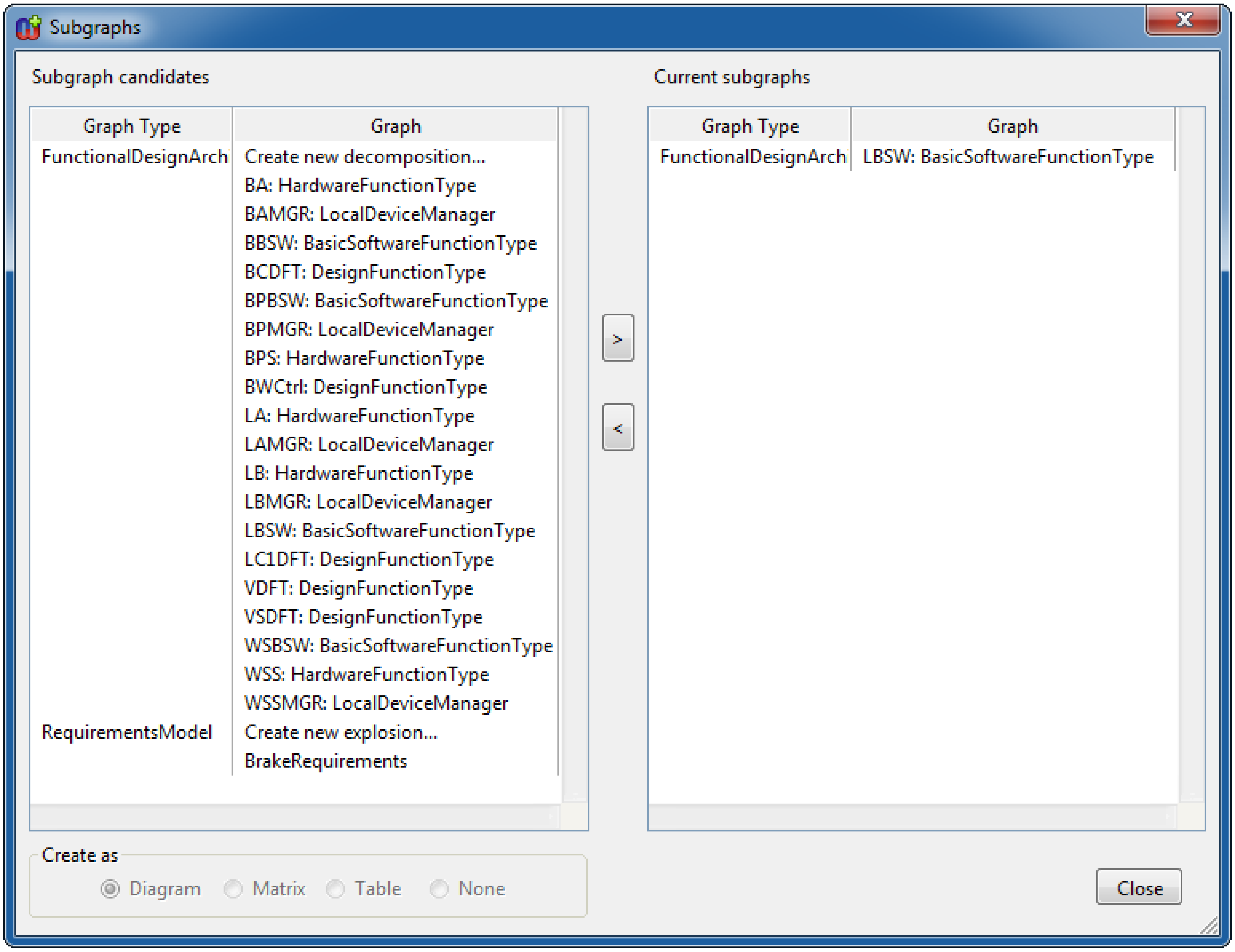
Other Symbol Library and Browser related functions are:

- Selecting a library symbol as a template's subsymbol
- Importing a symbol into the Symbol Editor
- Saving a copy of a symbol to the library from the Symbol Editor

7 New Tools for Handling Model Hierarchies

The subgraph management system has been completely re-designed for MetaEdit+ 5.0. The most visible changes from the old system are that all operations are handled in this single dialog, rather than a series of smaller dialogs, and that both subgraph categories, decompositions and explosions, are now managed together.

To add the first subgraph for an element or open its existing subgraph, you can Ctrl-double-click it or select **Open Subgraph** from its pop-up menu. For more advanced management of element's subgraphs, select **Manage Subgraphs…** from the pop-up menu. This will open the new Subgraphs dialog:



Subgraph candidates

Graph Type	Graph
FunctionalDesignArch	Create new decomposition...
	BA: HardwareFunctionType
	BAMGR: LocalDeviceManager
	BBSW: BasicSoftwareFunctionType
	BCDFT: DesignFunctionType
	BPBSW: BasicSoftwareFunctionType
	BPMGR: LocalDeviceManager
	BPS: HardwareFunctionType
	BWCtrl: DesignFunctionType
	LA: HardwareFunctionType
	LAMGR: LocalDeviceManager
	LB: HardwareFunctionType
	LBMGR: LocalDeviceManager
	LBSW: BasicSoftwareFunctionType
	LC1DFT: DesignFunctionType
	VDFT: DesignFunctionType
	VSDFT: DesignFunctionType
	WSBSW: BasicSoftwareFunctionType
	WSS: HardwareFunctionType
	WSSMGR: LocalDeviceManager
RequirementsModel	Create new explosion...
	BrakeRequirements

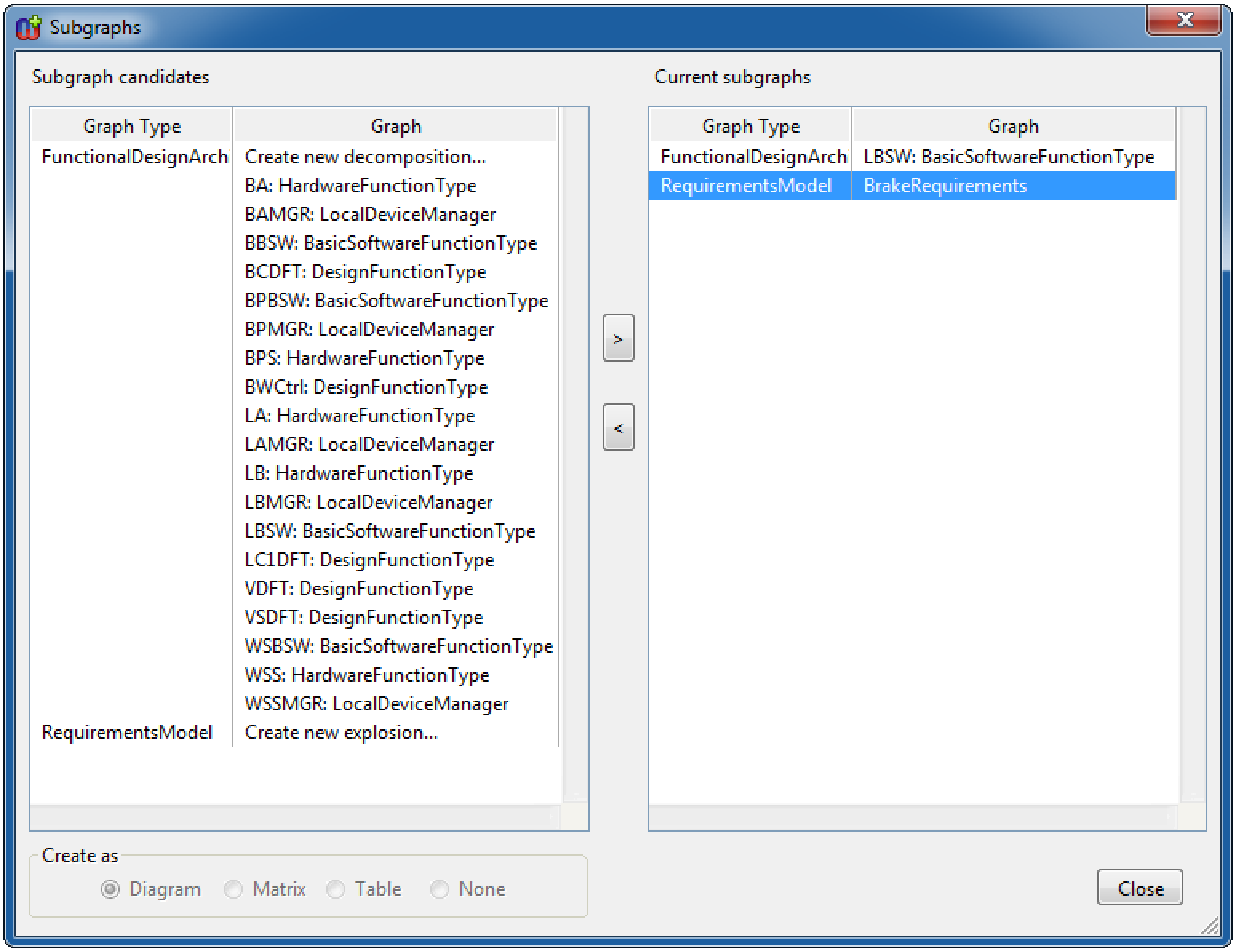
Current subgraphs

Graph Type	Graph
FunctionalDesignArch	LBSW: BasicSoftwareFunctionType

Create as: [✓ Diagram] Matrix Table None

All possible subgraph candidates are listed on the left (with the option to create a new subgraph) and the existing subgraph links are shown on the right. The pop-up menu for a candidate subgraph provides options to add the candidate to the current subgraphs list, open it for viewing, edit its properties, etc. A candidate can be added into the current list by double-clicking it or pressing the '>' button between the lists. Similarly a current subgraph can be removed by pressing the '<' button in the middle or selecting **Remove** from its pop-up menu. A pop-up menu also provides options for opening the current subgraphs (double-click does the same also), editing its properties, etc.

For example, in the example pictured above, if we want to add a RequirementsModel called 'BrakeRequirements' as a new explosion, we just double-click it (or select and press '>' or choose **Add** from the pop-up menu) and it will be moved from the candidate list to the list of current subgraphs:



Graph Type	Graph
FunctionalDesignArch	Create new decomposition...
	BA: HardwareFunctionType
	BAMGR: LocalDeviceManager
	BBSW: BasicSoftwareFunctionType
	BCDFT: DesignFunctionType
	BPBSW: BasicSoftwareFunctionType
	BPMGR: LocalDeviceManager
	BPS: HardwareFunctionType
	BWCtrl: DesignFunctionType
	LA: HardwareFunctionType
	LAMGR: LocalDeviceManager
	LB: HardwareFunctionType
	LBMGR: LocalDeviceManager
	LBSW: BasicSoftwareFunctionType
	LC1DFT: DesignFunctionType
	VDFT: DesignFunctionType
	VSDFT: DesignFunctionType
	WSBSW: BasicSoftwareFunctionType
	WSS: HardwareFunctionType
	WSSMGR: LocalDeviceManager
RequirementsModel	Create new explosion...

Graph Type	Graph
FunctionalDesignArch	LBSW: BasicSoftwareFunctionType
RequirementsModel	BrakeRequirements

Create as: [✓ Diagram] Matrix Table None

If we want to create a new subgraph, we double-click the desired subgraph type, whose row will show either **Create new decomposition…** or **Create new explosion…**. In this case we also have to select the representation type for our new subgraph, with the radio buttons at the bottom in the **Create as** box.

Please note that the subgraph link semantics still remain the same as before, i.e. there can be only one decomposition per element, whereas the number of explosion links is unlimited.

8 Generator Enhancements

MetaEdit+ 5.0 extends the generator tools and MERL in many ways. The main new features and improvements are described below.

8.1 Local variables

Local variables have been added to MERL and they are scoped to a single invocation of a generator. They can be expressed with a short syntax or a long syntax, similarly to global variables.

The short syntax uses the @ prefix (which is now a reserved character). It allows only a fixed variable name and can assign only the output of a single command.

```
@myVar = 'some value'
@myVar
```

The long syntax uses the **local** keyword. It allows the variable name to be built up by a number of commands, and similarly the value can be built up by a number of commands.

```
local 'myVar' write 'some value' close
local 'myVar' append ' plus '   'some more' close
local 'myVar' read
```

Please also note that the local and global variables are in separate namespaces, i.e. $foo has no relationship to @foo.

The lexical scope of a local variable is a single generator, and the runtime scope is a single invocation of that generator. For example, if we run this pair of generators:

```
report 'Main'
      @a = 'outer'
      subreport 'Sub' run
      @a
endreport
```

and:

```
report 'Sub'
      @a = 'inner'
endreport
```

the output will be 'outer': the subgenerator cannot read or write the supergenerator's local variables (even if Sub is the same generator as Main, i.e. for recursive generators).

8.2 Short syntax for generator definition and calling

Subgenerators can now be defined and called with a shorter syntax, myGenerator(). Note there can be no space before the opening parenthesis. The old style for defining and using generators is still allowed (useful for dynamically building up the name of the generator to call). We can compare the old syntax:

```
report 'myGenerator'
      subreport '_translators' run
      id%xml
endreport
```

with the new shorter syntax:

```
myGenerator()
      _translators()
      id%xml
```

Note there are no curly brackets around the content of the generator: curly brackets in MERL always mean a navigation step to a new model element, and a generator call does not cause such a step.

8.3 Generator parameters

Generators can now have parameters that can be defined with the new syntax:

```
myGenerator(@a, @b)
      'First argument is ' @a ' and second is ' @b
```

Note there can be no space before the opening parenthesis. The parameters are listed as comma-separated local variables within the parentheses, and behave within the body of the generator as local variables.

When calling a generator, values are supplied separated by commas:

```
myGenerator('MetaEdit+', '5.0')
```

Generators can be called in the old style with an extra **arguments** keyword and comma-separated arguments:

```
subreport 'myGenerator' arguments 'firstArg', 'secondArg' run
```

In either style, each argument can be made up of a string of MERL commands (as in the **orderby** clause for **do** and **foreach** loops):

```
subreport 'myGenerator' arguments
      'first' 'Arg',
      'second' 'Arg'
run
myGenerator('first' 'Arg', 'second' 'Arg')
```

Arguments to subreport calls can be nested, e.g.

```
subreport 'main' arguments
      subreport 'inner' arguments
            '1'
      run
run

main(inner('1'))
```

If the number of arguments supplied when calling a generator does not match the number of parameters expected by that generator, any missing parameters are given an empty string as a value, and any extra arguments are added as local variables called __extraParamN, where N is the index of the argument. The index starts from 1 for the first argument (not the first extra argument). A warning is raised when running from a Generator Editor, but not elsewhere.

8.4 runSuper

The new runSuper MERL command is similar to run but looks up the named generator in the supertype of the graph type containing the current generator. It can call subgenerators of a different name from the current generator but cannot be used in Graph, nor in generators not stored in a particular Graph type (e.g. identifier generators or the generator bodies found in symbol elements).

8.5 Conditional breakpoints

MERL breakpoints can now have conditions, which are evaluated as MERL snippets in the current context (including element stack and variables), and the output compared to the supplied string or regular expression.

Breakpoint | Edit Condition will edit the condition for the breakpoint in the selection (if none, a breakpoint with an empty condition is first added at the start of the selection). Setting the condition source to None or pressing Clear will remove the condition, i.e. make the breakpoint always active.

8.6 Background generator timeout

Users can now set the timeout in milliseconds for running a background generator such as an identifier generator or a text or condition generator in a symbol element. The default timeout is 200ms.

Things to remember and consider when setting the timeout:

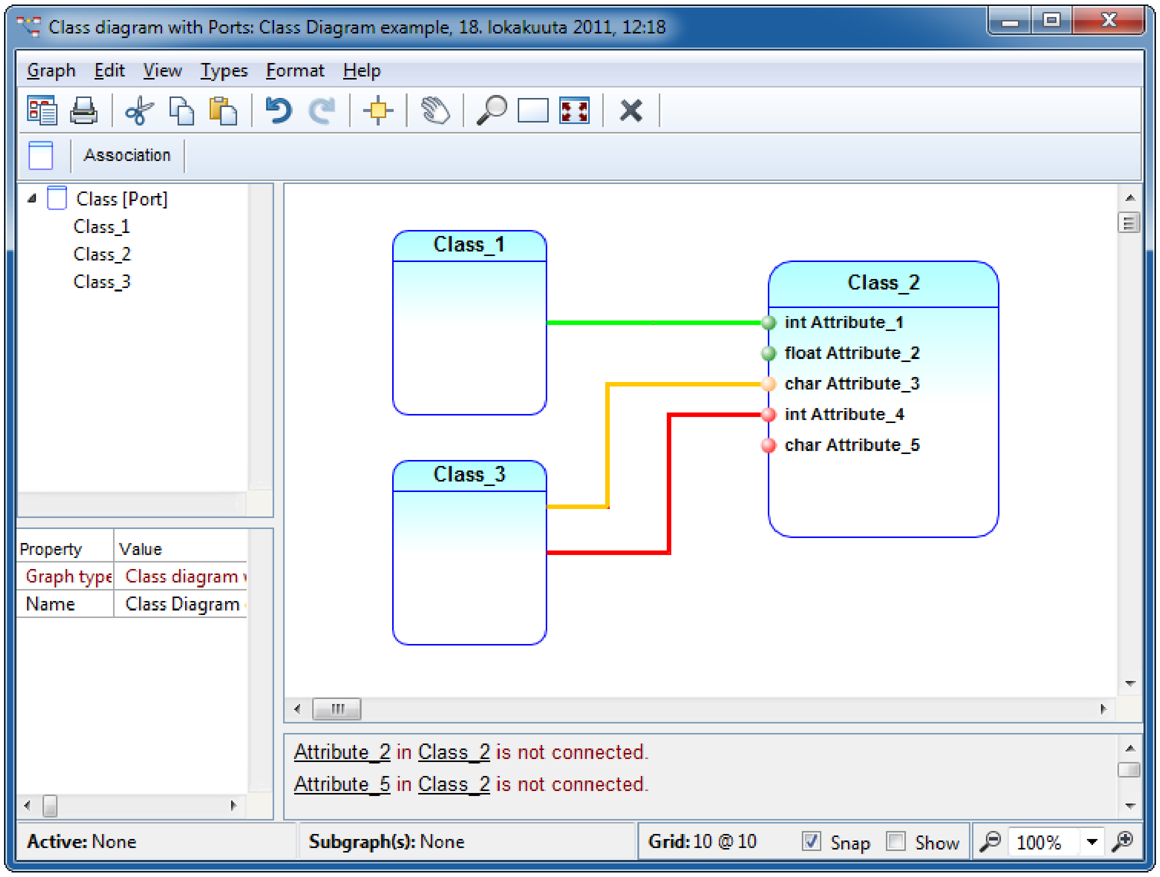
- Most background generators run in under 1ms, but things like iterating through all graphs or using representation information can take longer.

- The time is clock time, not CPU time used by MetaEdit+ or that generator, so another high priority program can occasionally cause spurious timeouts.
- If there is an infinite recursion in a background generator, it may well be run for many objects simultaneously, since many are visible on the screen. This can slow response to a crawl: with 20 objects and 200ms, you have to wait 4 seconds for each click to be processed.
- We advise leaving the default alone while building generators, but it can be increased or turned off entirely with a value of 0 in order to make sure all background generators are run to completion during normal modeling.

8.7 Live Check

IDE users are accustomed to seeing errors listed in a separate pane below their work. In a modeling language, it is often better to show errors directly in the symbol concerned. For some situations an error pane can be useful, so MetaEdit+ now offers one in the Diagram Editor. The contents of the pane are formed by running a generator called `__LiveCheck`, and the pane is only shown if such a generator exists.

The 'Class diagram with Ports' in the 'Port example' project uses `__LiveCheck` to list unconnected attributes, as an example:



Property	Value
Graph type	Class diagram
Name	Class Diagram

```
Attribute_2 in Class_2 is not connected.
Attribute_5 in Class_2 is not connected.
```

8.8 Other new features and improvements

Other notable new features and improvements in the generator section are:

- Generator diff with saved version, Generator | Compare with Saved
- A generator can have an icon. Currently it is only shown if the generator is included in editors' action toolbars by starting its name with **!**.
- A Break button is now added to the Progress Dialog when running a generator, allowing infinite recursion problems to be cleanly debugged
- Generator Debugger shows local as well as global variables (local variables are shown for the context of the generator currently selected in the generator stack list)
- Generators allow Unicode characters in type names without escaping
- Enter maintains indent in generators
- Generated Files status bar shows total size in kB and speed (kB/s)
- Existing Generated Files window is used rather than opening a new one
- Regex translators can have their subexpression matches translated, e.g. $1%upper; on the RHS will find the match for $1, then translate it with the %upper translator (looked up when used)
- If the name of a generator has changed in the source code, choosing Rename As... will offer the new name by default
- do graphs has been optimized for better performance with large repositories
- Processing translator definitions has been optimized, improving their usefulness even in intensively used identifier or symbol generators

9 New API commands

MetaEdit+ 5.0 also provides a number of extensions in the API. The new API commands have been summarized here.

API commands for creating and editing non-properties via dialogs:

Command	Purpose
MEAny createGraphDialog(METype)	Start the Create Graph dialog. Returns an MEAny containing either the new Graph MEOop or MENull if canceled.
MEAny createFromPropertyDialogInArea(METype, int)	Create a new non-property from a dialog, storing the result in the project specified by the int areaId. Returns an MEAny containing either the new MEOop or MENull if canceled.
xsd:boolean propertyDialog(MEOop)	Open a Property Dialog on an existing non-property. Returns true if the user presses OK, false otherwise.

Note that if the user takes a long time in the dialog, the API call may time out. In that case the dialog remains open, and actions taken there are valid - the API caller just doesn't hear about the results after receiving the timeout notification.

Other new API commands are:

Command	Purpose
Boolean forGraphRun(MEOop, string)	Run the generator for the specific graph. Returns false if the generator does not exist or gives a compilation error.
String generatorNames(METype)	Return a string containing the names of all the generators defined in a graph type, one per line

There is also a new command-line parameter **saveWSDL:**, which will save the WSDL to the file specified in the next argument. The settings for hostname and port will be taken from the API Tool most recently started via the API, or the defaults if none.