Version 5.6

Co-evolution Example

MetaCase Document No. EV-5.6

First Printing, 1st Edition, October 2025

MetaCase
Ylistönmäentie 31
FI–40500 Jyväskylä
Finland

E-mail: info@metacase.com
WWW: https://www.metacase.com

Preface

The goal of this example is to demonstrate how MetaEdit+ supports co-evolution of a modeling language, and the models made with it. This means that when a modeling language is modified, the changes are reflected in existing models and to the rest of MetaEdit+ functionality such as its editors.

By language evolution, we mean modification of its metamodel (in GOPPRR), its notational symbols and its generators. These modifications also affect the user interface of MetaEdit+, such as dialogs, property sheets and icons, but those updates are fully automated.

This example is based on making a set of changes to a small example language and then describing how those changes are reflected in the models. It also presents, in a cookbook-style manner, solutions to commonly occurring language evolution tasks. Please note that the example is designed so that you can work "hands-on" to gain a deeper understanding of the subject matter.

To explore the co-evolution example thoroughly hands-on, you need to have either MetaEdit+ Workbench or the evaluation version installed on your machine, along with the example language. The example language is introduced later in this guide, along with download instructions.

For further information about MetaEdit+, please refer to the 'MetaEdit+ Users Guide', 'MetaEdit+ Workbench Users Guide' or our web pages at https://www.metacase.com.

1 Co-evolution example

The co-evolution example presents a set of language changes that are common when maintaining and refining a modeling language, including adding, renaming, removing or changing links in the language definition. These modifications can target many parts of a modeling solution powered by MetaEdit+:

1) Metamodel: the language's concepts defined in GOPPRR (Graphs, Objects, Relationships, Roles, Ports and Property Types).
2) Symbols: graphical representations of the elements defined in the metamodel.
3) Generators: providing model checking, reporting, code generation etc.
4) User interface: the content and appearance of dialogs, menus and toolbars.

In this chapter we introduce the co-evolution scenarios together with a small example in which the changes are performed. Chapter 2 then explains each change in detail, allowing you to perform the same changes hands-on. While the tasks in Chapter 2 are carried out by a single person using the single-user version of MetaEdit+, Chapter 3 describes how to perform these changes in an industrial context having large models, multiple languages, multiple modelers, and possibly multiple repositories. We also discuss how to use the API to specify model updates where the automatic updates are not applicable.

Please note that walking through the co-evolution example requires a good knowledge of the metamodeling concepts of MetaEdit+ and basic knowledge on how to use MetaEdit+. A good starting point for gaining this knowledge is the Family Tree example in the 'Evaluation Tutorial'. The hands-on parts of language evolution can be completed within an hour.

1.1 CO-EVOLUTION SCENARIOS

In this guide we demonstrate co-evolution from two aspects: the nature of the change and the location of the change. The first aspect is the kind of change we are making: adding, renaming or removing a part of the language definition, or changing a link from one part of the language definition to another part.

The second aspect of evolution is the location of the change — whether it targets the language structures specified in the metamodel, constraints on how those structures can be used in models, or the notation of how they are displayed.

Based on these two aspects — four natures of change and three locations of change — we identify 12 different scenarios for co-evolution. Table 1-1 summarizes these scenarios, which we will demonstrate in detail throughout this guide. For example, Scenario 1 involves adding an element to the metamodel, while Scenario 2 adds a constraint. Although each scenario can be performed individually, a coherent sequence of changes provides a more realistic demonstration. For example, Scenario 2 would often add a constraint related to the element just added in Scenario 1. Therefore, we organize the scenarios as a sequence of 12 steps that reflect how such changes might occur in practice.

Location of Change	Nature of Change			
	Add	Rename	Remove	Change
Metamodel	1	4	7	10
Constraints	2	5	8	11
Notation	3	6	9	12

Table 1-1. Co-evolution scenarios.

1.2 AN EXAMPLE LANGUAGE

For our demonstration of co-evolution, we use the same MetaEdit+-based example as used in the book on Domain-Specific Languages by Martin Fowler[1]: a state machine for Gothic Security — a subset of the secret doors and revolving bookcases of spy films. Its metamodel is shown in Figure 1-1 using graphical GOPPRR.

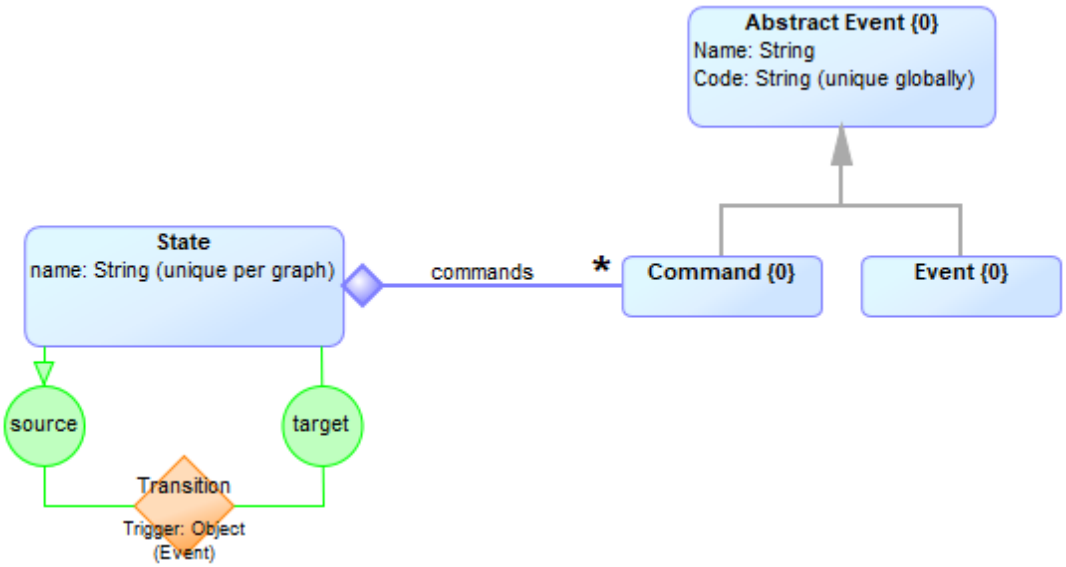

Figure 1-1. Metamodel of state machine.

The language models simple state machines: states may have commands, and transitions between states are triggered by events. Both commands and events have a name and a code. There are also constraints — though they are only evident from the code shown in Fowler's book — such as the requirement that state names must be mandatory and unique within the current state machine, and that event codes must be unique.

A model based on this metamodel is shown in Figure 1-2, which also illustrates the concrete syntax of the language. The model defines the functionality of a system for Miss Grant to open

[1] M. Fowler, Domain-Specific Languages, Addison-Wesley, 2010.

a hidden panel. From this kind of model, code can be generated for various targets; in this case, we use Fowler's controller code as an example.

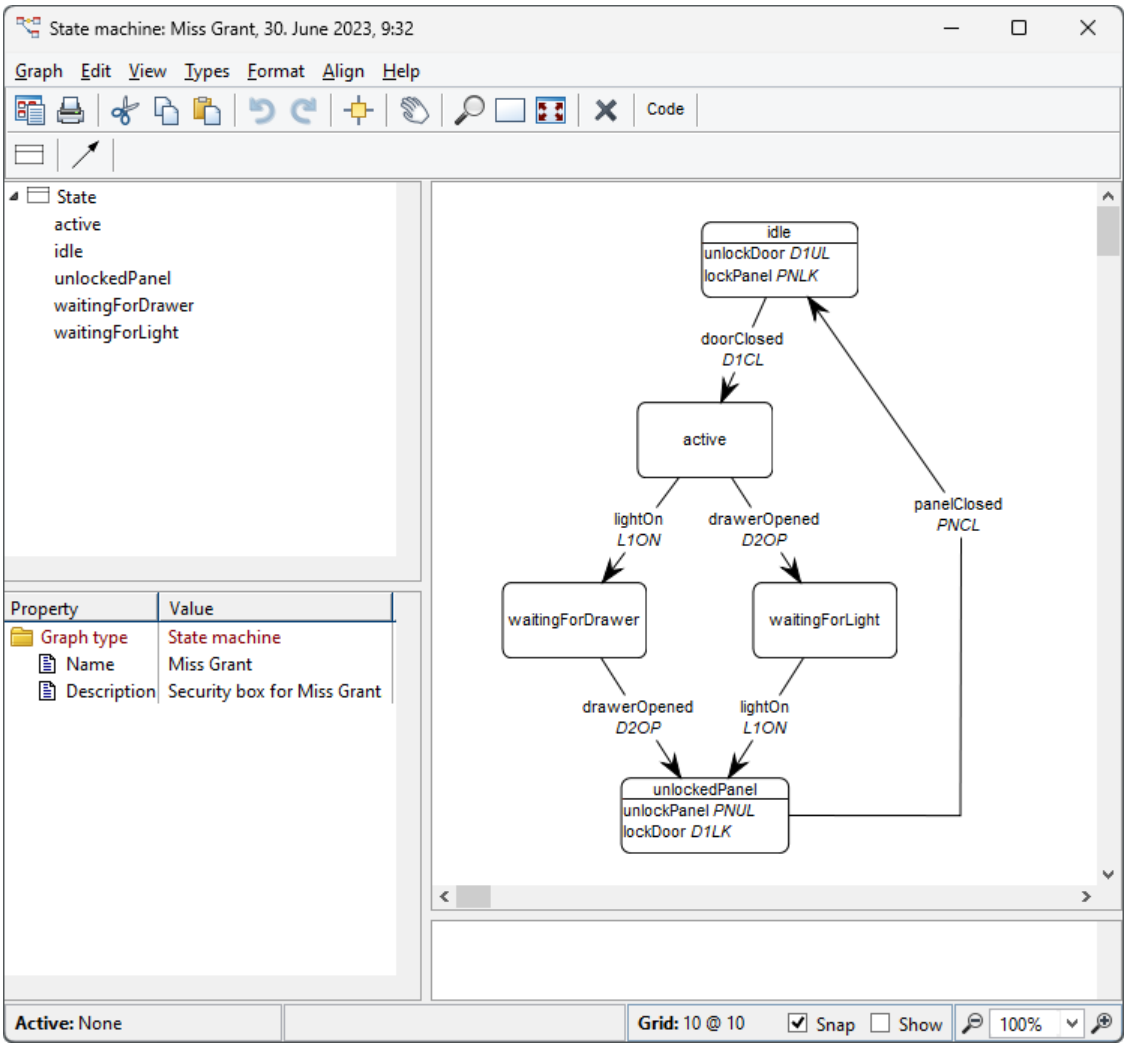

Figure 1-2. Example state machine model: Miss Grant.

1.3 INDIVIDUAL CO-EVOLUTION STEPS

Following the 12 different scenarios identified in Table 1-1, we have 12 different individual changes targeting the Gothic Security project's State machine language as follows:

1) Add element to metamodel: Add a new Reset element to State machine, with a set of events that trigger it.
2) Add constraint: Only one Reset can be defined in a State machine, and it can connect to only one State there.
3) Add notation: The symbol for Reset is created.
4) Rename element in metamodel: State is renamed to Situation.
5) Rename constraint: In MetaEdit+, constraints do not have names, so no change is needed.
6) Rename notation: The symbol for Situation is renamed.
7) Remove element from metamodel: The Reset element is removed from State machine.

8) Remove constraint: Reset is not allowed to have a relationship to Situation.
9) Remove notation: Reset's symbol is removed.
10) Change metamodel: The Transition relationship's Event property is moved to the Source role.
11) Change constraint: Add Start, then update old Reset constraints to point to Start instead and add Start into the original Transition binding.
12) Change notation: Make the Situation symbol refer to a different library symbol.

These 12 steps are designed to be implemented sequentially, resulting in 12 successive versions, as detailed in the next chapter. All the suggested changes are evolutionary, not revolutionary: If the language were to change completely, language engineers would likely create a new language instead. However, they could still reuse elements from the previously defined language and models.

2 Conducting language changes

In this chapter, we detail how to perform the language changes, the options to consider, and demonstrate how the resulting models — and the rest of MetaEdit+ functionality — work after the changes.

To obtain this example, go to https://github.com/mccjpt/Gothic and follow the Quick Start instructions there. They will give you a local copy of the MetaEdit+ repository with the initial language and model.

Start MetaEdit+, select the 'GothicAfter0' repository, choose the 'Gothic Security' project and the 'user' account, and press Login.

2.1 ADD ELEMENT TO METAMODEL

Add a new Reset element to State machine, with a set of events that trigger it.

To start modifying the language, open a Graph Tool by pressing the Graph Tool toolbar button or selecting **Metamodel | Graph Tool** in the MetaEdit+ launcher. Next, in the Graph Tool choose **Graph | Open…** and MetaEdit+ opens the only Graph Type from the repository: 'State machine' as shown in Figure 2-1.

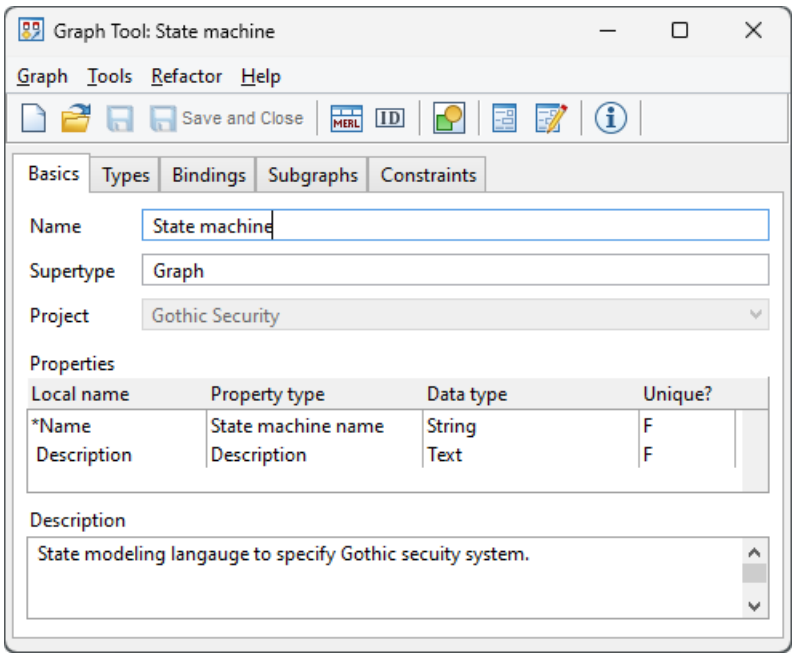

Figure 2-1. Graph Tool for the State machine graph type.

Next, in the Types tab's Objects list, open the pop-up menu and choose **Add New…** to open an Object Tool.

Enter ‘Reset’ as the name for the new object type and give a description: ‘Reset causing the state machine initialization’ (see Figure 2-2). This description text is used as part of the language help, available in modeling editors for language users.

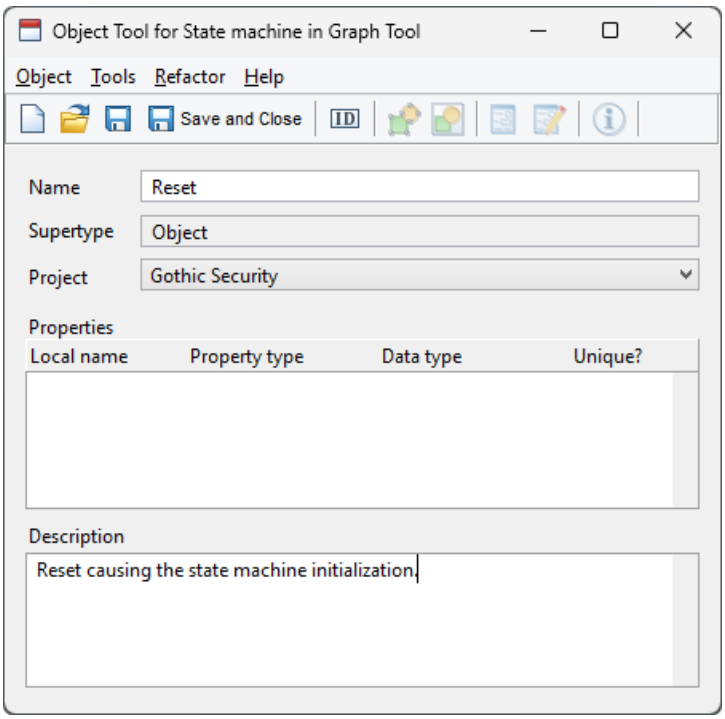

Figure 2-2. Adding ‘Reset’ in the Object Tool.

In this scenario, ‘Reset’ can refer to several Events — the Event object type is already defined in the metamodel (see metamodel in Figure 1-1). In the Properties list of the Object Tool, open the pop-up menu and choose **Add Property…**, and then ‘New Property Type’ from the dialog that opens.

In the Property Tool that opens, enter ‘Events’ as the name of the new Property type, and set its Data Type to be ‘Collection’. From the Item Type menu choose ‘Object...’, and select ‘Event’ as the Object type from the dialog that opens. Enter the description: ‘Events causing reset of state machine’ as shown in Figure 2-3.

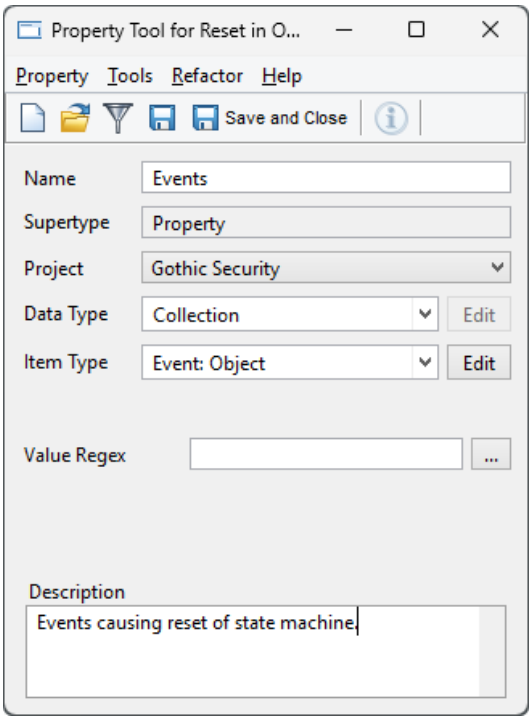

Figure 2-3. Adding 'Events' property in Property Tool.

To finalize the changes, press **Save and Close** in the Property Tool and Object Tool, and just **Save** in the Graph Tool, as we will continue working there.

→ *Note that after these changes the generator does not know about resets nor provides any code from them. In the Graph Tool for 'State machine' select* ***Tools | Generator Editor****, open the hierarchy for generator called '!Code' to see a generator '_events'. Extend it by adding a line to produce basic code for Reset as shown below. Then close the Generator Editor saving the changes.*

```
foreach .Reset {do :Events {oid newline}}
```

After the change, existing models are updated automatically, and modeling editors automatically adjust when next opened, allowing the modeler to create Reset objects with lists of Events. You may next open the existing state machine for 'Miss Grant' and add reset events to it, e.g. a Reset object with a new Event 'doorOpened' with code 'D1OP'. You can also try the generator updates by running the generator called 'Code' from the Diagram Editor toolbar.

If you are satisfied with the change, you may save the changes to the repository by pressing **Commit** in the main MetaEdit+ launcher.

2.2 ADD CONSTRAINT

Only one Reset can be defined in a State machine, and it can connect to only one State there.

We will break this down into a new binding, a Connectivity constraint and an Occurrence constraint.

The new binding will allow a Transition relationship to have its Source role connected to a Reset object, and its Target role connected to a State object. In the Graph Tool for State machine, build this new binding on the Bindings tab by choosing **Add…** in the various lists' pop-up menus as follows:

1) In the Relationships list, add a new binding for ‘Transition’ (see Figure 2-4).
2) With this new Transition binding selected, in the Roles list add both ‘Source’ and ‘Target’.
3) With the ‘Source’ role selected, in the Objects list add ‘Reset’.
4) With the ‘Target’ role selected, in the Object list add ‘State’.

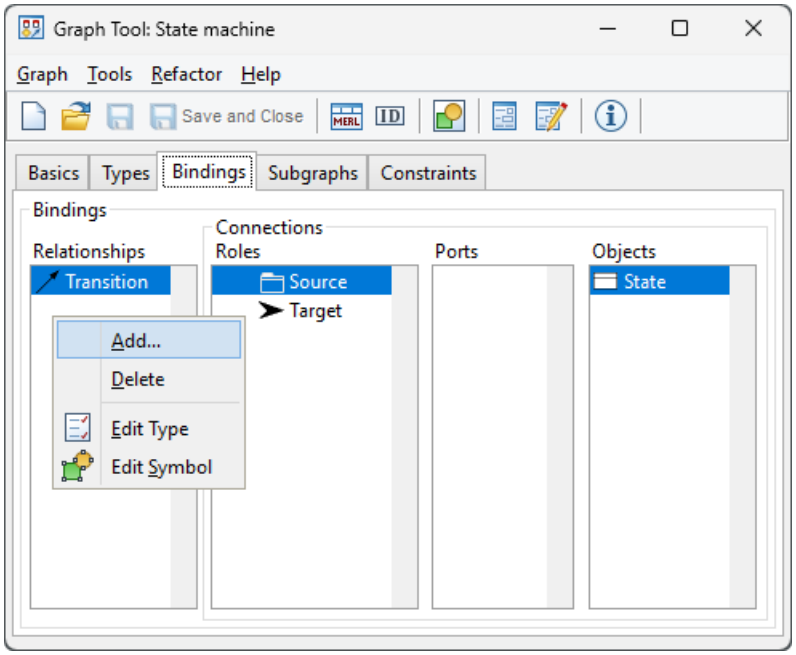

Figure 2-4. Adding binding.

Next, a connectivity constraint that allows ‘Reset’ to only be in one ‘Transition’ is defined in the Graph Tool’s Constraints tab:

1) From the ‘Add Constraint For’ pull-down list, select Connectivity, and press **Add**.
2) For Objects of type ‘Reset’, allow at most 1 relationship of type ‘Transition’, as in Figure 2-5, and press **OK**.

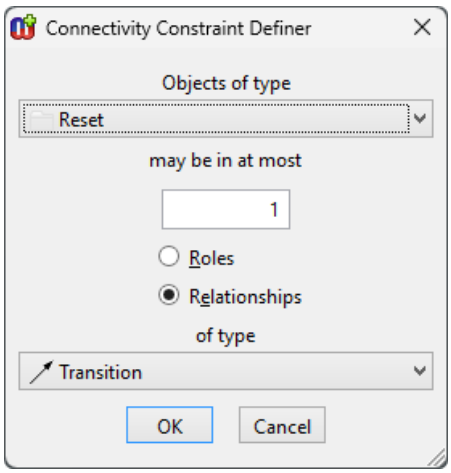

Figure 2-5. Adding connectivity constraint.

Finally, an occurrence constraint is set to limit the number of Resets to one per graph. In the Graph Tool’s Constraints tab:

1) From the ‘Add Constraint For’ pull-down list, select Occurrence, and press **Add**.

2) For Objects of type ‘Reset’, allow at most 1 occurrence in a graph, as in Figure 2-6, and press **OK**.

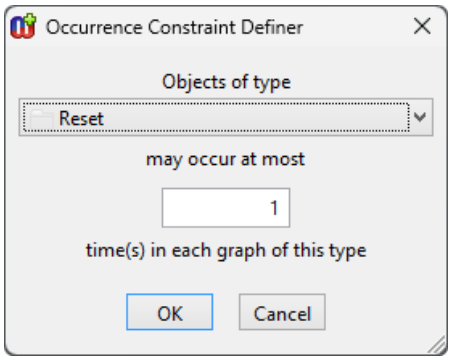

Figure 2-6. Adding occurrence constraint.

Save the new binding and constraints by pressing **Save** in the Graph Tool.

After adding these constraints, existing models might not follow the language definition anymore: modelers may have added multiple resets to a graph. Only one of those should be allowed to remain, but we cannot automatically know which. To support the modeler in updating the models to conform to the updated language definition, MetaEdit+ offers several facilities. We can define a LiveCheck report to show problems in the model, include checks in the code generators, or make graphical symbols show warning text or another indicators.

In this case, we will create a LiveCheck report to show errors at modeling time, in the bottom of the Diagram Editor.

From the Graph Tool for State machine, open its generators with **Tools | Generator Editor**. Select the existing __LiveCheck generator, and add a line at the end to call a new generator:

```
_resetCheck()
```

When you **Save** the __LiveCheck generator, you can open the hierarchical call tree under __LiveCheck and see that the _resetCheck generator is shown in red with a question mark, as it has not yet been defined. Select it and choose **New…** from the popup menu. This will define the new generator, and you can fill in the rest of the generator as below:

```
_resetCheck()
@resets = __(foreach .Reset {'1'})
if @resets > '1' num then
   'Warning: model contains too many resets: '
   dowhile .Reset {id ', '} newline
endif
```

After the generator is added, close the Generator Editor, saving changes.

The editors and models automatically follow the new language definition. You may now try to add more Resets, and Transitions from Resets to States, to see how the constraints are enforced. If a graph made earlier already contains more than one Reset, the problems will be listed at the bottom of the Diagram Editor, with hyperlinks to the Reset objects.

Press **Commit** in the MetaEdit+ launcher to save the changes.

2.3 ADD NOTATIONAL SYMBOL

The symbol for Reset is created.

In the Graph Tool for 'State machine', go to the Types tab, open the 'Reset' object type's pop-up menu and choose **Edit Symbol**.

In the Symbol Editor that opens, click the text button (T) on the toolbar to start creating a Text box. In the drawing area, drag the desired area for the text box illustrated in Figure 2-7, from the top-left corner to the bottom right corner. The Format dialog that opens allows you to define the details of the text box. For now, just go to the Line and Fill tab and change the line color to black and its style to a dash, and press **OK**.

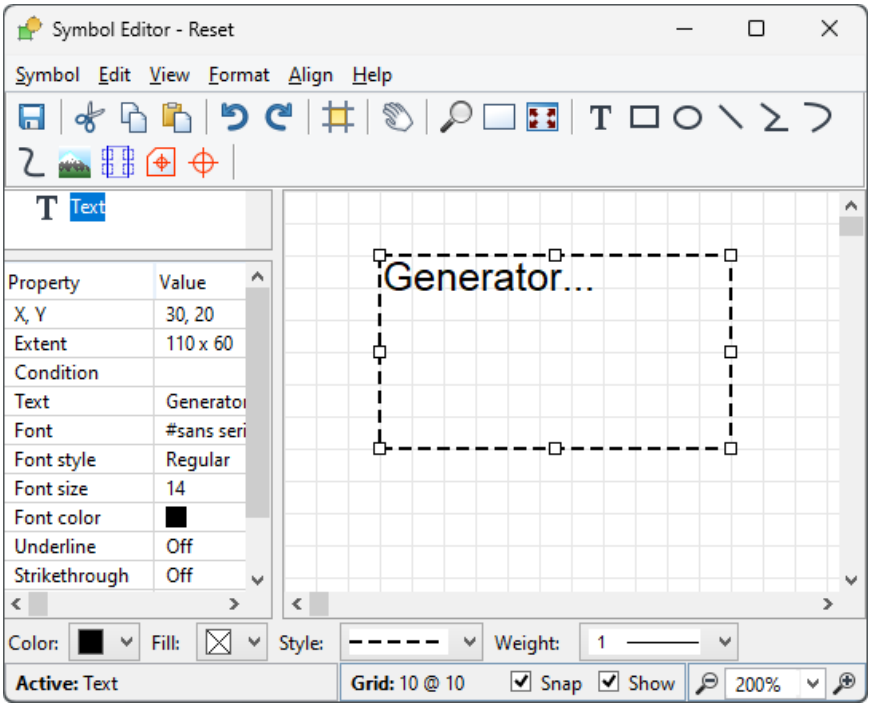

Figure 2-7. Adding a symbol for Reset.

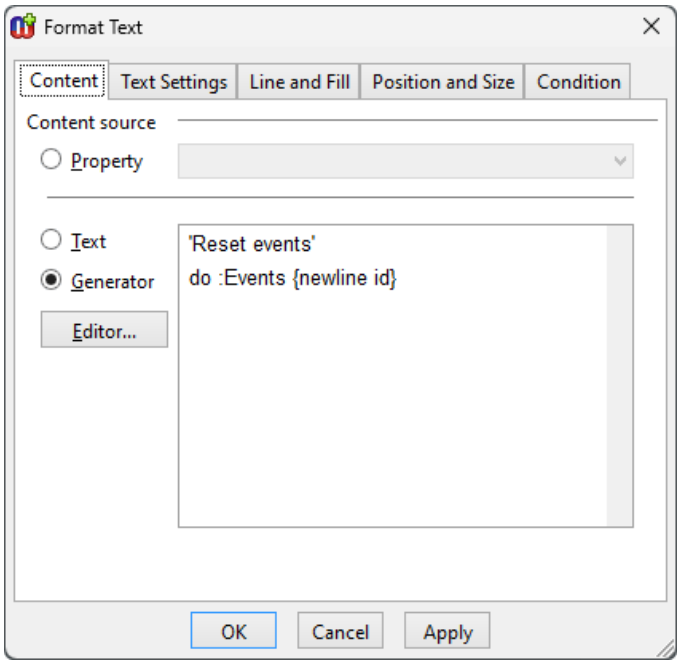

Figure 2-8. A generator to show resets in the text element.

With the text box selected, select **Format…** from its pop-up menu. On the Content tab, choose the Generator radio button to make the text come from a generator's output, rather than a property value or as fixed text. Write a generator to list all Reset events as follows (see Figure 2-8):

```
'Reset events'
do :Events {newline id}
```

Press **OK** in the dialog and close the Symbol Editor, saving changes.

After this change, you can open a model and see the newly created symbol for all resets. The symbol is also shown elsewhere in the user interface, as an icon in toolbars, tree views and browsers. If the symbol to be shown in models is not suitable when scaled down to icon size, you can define a dedicated icon with the Icon Editor of MetaEdit+ (opened with **Icon Editor** from the **Tools** menu of a metamodeling tool or the **Symbol** menu of a Symbol Editor).

Press **Commit** in the MetaEdit+ launcher to save the changes.

2.4 RENAME ELEMENT IN METAMODEL

State is renamed to Situation.

In the Graph Tool for 'State machine', go to the Types tab and double-click the 'State' object type to open it in an Object Tool. Change its name to 'Situation' and press **Save and Close**.

This renaming is automatically updated throughout the metamodel: no need to manually update bindings, constraints etc. Also, the concrete syntax updates automatically, as do editors and browsers.

→ *If there is already a generator that refers to 'State' by name, it must be changed. Open the Generator Editor for State machine and then find and replace. E.g. use **Edit | Advanced Find…** with string .State, and then for each Replace '.State' with '.Situation', **Save** and press F3 to go to the next occurrence.*

You may re-open the state machine for Miss Grant to see that States have been updated to Situations and run the Code generator.

Press **Commit** in the MetaEdit+ launcher to save the changes.

2.5 RENAME CONSTRAINT

In MetaEdit+, constraints do not have names, so no change is needed or indeed possible.

2.6 RENAME SYMBOL/NOTATION

The symbol for Situation is renamed.

Normally, symbols in MetaEdit+ are directly related to language elements and do not have names. Thus, renaming 'State' to 'Situation' earlier automatically kept its existing symbol with no need for a corresponding renaming of the symbol. For complex symbols, or reuse of symbol parts, it is also possible to save symbols by name in the symbol library. To illustrate this, we made the State symbol use a template element to pull in a subsymbol retrieved from the symbol library.

Open an Object Tool for 'Situation' and then open its Symbol Editor. Select the blue template symbol element in the left list and from its pop-up menu choose **Format…** (Figure 2-9).

→ *The symbol for states is a little clever, with conditional elements so that if there are no Commands, it just shows the name in the middle, but if there are Commands, the name is at the top, with a dividing line and the Commands underneath it.*

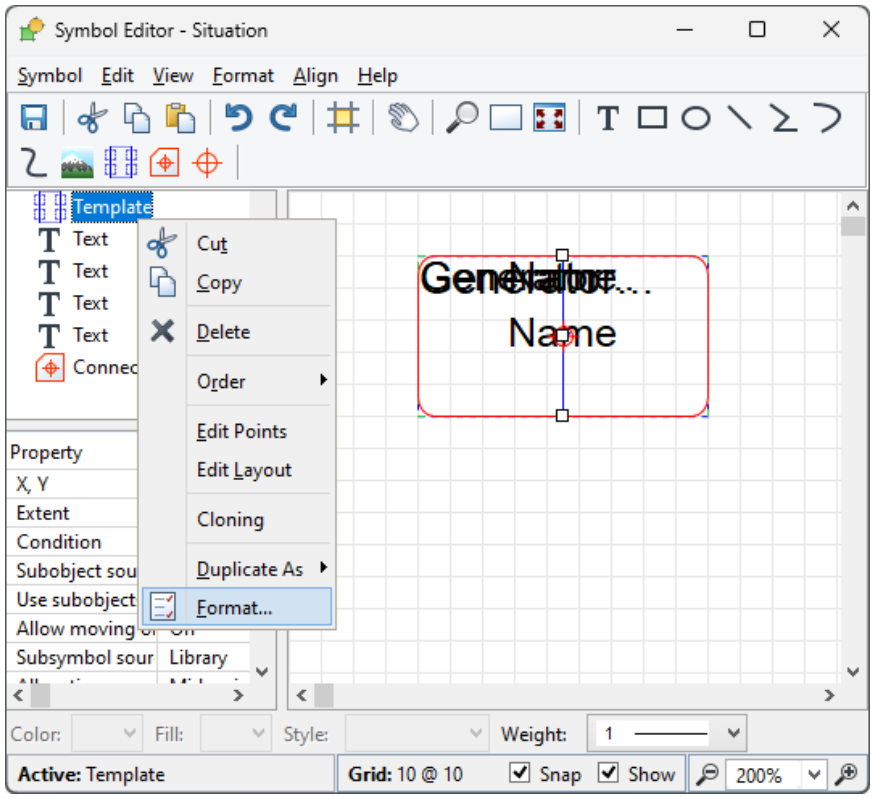

Figure 2-9. Choosing Format to edit the Template element.

Next, in the Subsymbol tab (Figure 2-10), click the button with three dots next to Selection: 'Rectangle' to open the Symbol Browser on the library.

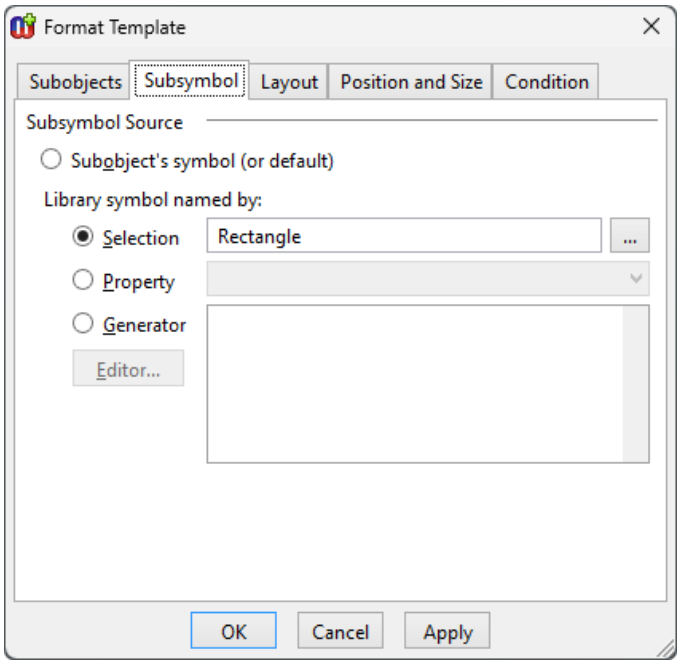

Figure 2-10. Choosing subsymbol from the library.

Select 'Rectangle' from the list of library symbols and press the Rename button as illustrated in Figure 2-11. Enter 'BlackRectangle' as its new name and close the dialog and Symbol Browser by pressing **OK**. To finalize, close the Format dialog and Symbol Editor, saving the changes.

Next, open the state machine for Miss Grant. The model should look the same as earlier as it still uses the library symbol we just renamed.

Press **Commit** in the MetaEdit+ launcher to save the changes.

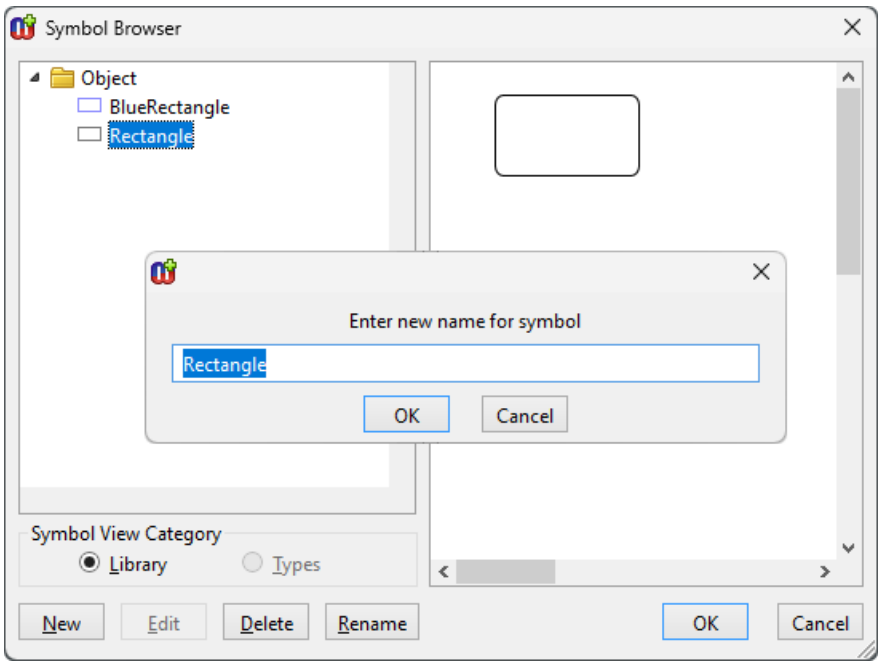

Figure 2-11. Renaming library symbol.

2.7 REMOVE ELEMENT FROM METAMODEL

The Reset element is removed from State machine.

Before deleting anything, we can first consider if it is better just to hide the metamodel element or make them no longer instantiable, rather than delete it and all its instances permanently. The gentler approach allows existing model data to be used for example when generating code – after all, the generator support for them already exists and works.

This approach of deprecation rather than hard deletion is popular with language users, allowing them to see and update design data from the past, while guiding them not to use the old language concept anymore. Its practical benefits are well-known from programming languages. Often, it is wise to enact deprecation and deletion in phases, with the tool offering guidance for each phase: first just show warnings, next prevent creation of the deprecated elements, then give notice of a deadline to remove existing deprecated elements. Here we will choose the middle option, preventing creation of new Reset objects but allowing existing ones to exist, with warnings shown for them.

The 'Reset' element can be removed in the Graph Tool for State machine by selecting it in the Types tab and choosing **Delete** from its pop-up menu. Press **Save** in the Graph Tool to finalize the removal. Removing Reset in this way removes it from the State machine language definition, and removes the user interface for adding new Reset instances to State machines. Reset is still referenced in constraints and has a symbol: we will deal with those in the following scenarios, moving further down the path of deprecation.

Existing Reset instances remain in the models. The _resetCheck generator, created in Section 2.2, can be updated to show a warning for Resets that already exist:

```
foreach .Reset {
   'Warning: model contains deprecated Reset: ' id newline
}
```

After the removal you may re-open the state machine for Miss Grant to confirm that adding new Reset objects is no longer possible, but earlier models open and can be updated. The models, modeling tools and generated code all continue to work without problems. We could later ask modelers to remove deprecated Resets (e.g. if human intervention is needed to handle those situations), or removal could be automated via the MetaEdit+ API (see Section 3.5 for details).

Press **Commit** in the MetaEdit+ launcher to save the changes.

2.8 REMOVE CONSTRAINT

Reset is not allowed to have a relationship to Situation.

In this case, the constraint to remove is a binding: the one allowing Reset to be the Source for a Transition. In the Graph Tool for 'State machine', go to the Bindings tab. At the moment there are two bindings. Select the 'Transition' whose 'Source' role has the 'Reset' object, open the Transition's pop-up menu and choose **Delete**. To finalize the change, press **Save** in the Graph Tool. New Transitions from Resets to Situations can no longer be made. Existing ones are allowed to remain for now, and considered as deprecated, as above.

We can update the _resetCheck generator to show a warning for the deprecated transitions:

```
foreach .Reset; where .() {
   'Warning: ' id ' has a transition to '
   dowhile .(){id ', '}'.' newline
}
```

After this removal of the constraint with deprecation, re-open the state machine for Miss Grant. Adding new Transitions from Reset to Situation is not possible anymore, and the Live Check lists any such deprecated Transitions.

Press **Commit** in the MetaEdit+ launcher to save the changes.

2.9 REMOVE SYMBOL/NOTATION

Reset's symbol is removed.

The symbol of a language element can be removed with the Symbol Editor. In the Graph Tool for State machine, go to the Types tab, open the pop-up menu for 'Reset' and choose **Edit Symbol**.

In the Symbol Editor choose **Symbol | New…** to clear the current symbol, answer **Yes** to confirm, and close the Symbol Editor, saving changes.

To see the results, open the State machine for Miss Grant. The default notation of a simple text box is used for Resets.

Press **Commit** in the MetaEdit+ launcher to save the changes.

2.10 CHANGE METAMODEL

The Transition relationship's Trigger property is moved to the Source role.

In MetaEdit+, changing a reference to an existing element in the metamodel, like moving the 'Trigger' property to the 'Source' role, is based on a direct link rather than an indirect reference by name. Rather than creating a new property type, as in scenario 1, we use the existing 'Trigger' property type in a new property slot in the 'Source' role.

To make this change, open a Role Tool for 'Source', for example from the Types pane of a Graph Tool opened on State machine. In the Role Tool choose **Add Property…** from the pop-up menu in the list of properties. Select the existing 'Trigger' as in Figure 2-12. To save the change press **Save and Close**.

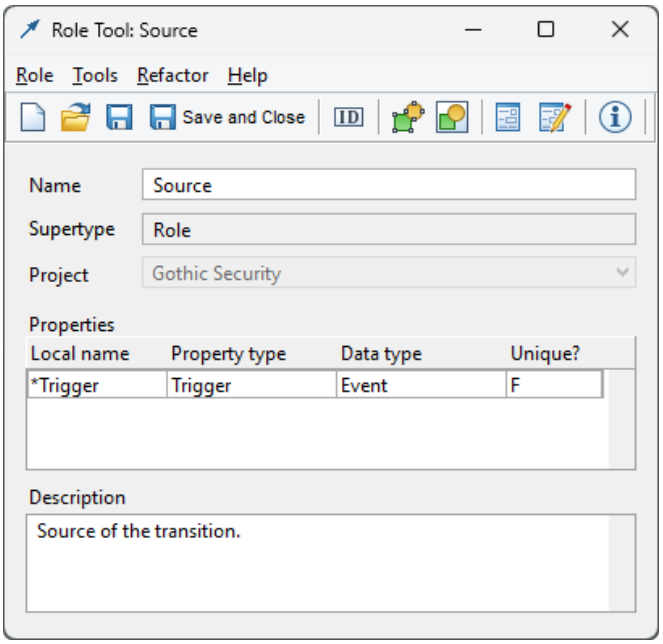

Figure 2-12. Adding Trigger to Source Role.

After this change the 'Transition' relationship still has its own 'Trigger' property slot, so information is not lost, and generators continue to work. Keeping 'Trigger' in 'Transition' is useful for this interim period, allowing current Trigger information to be moved to the 'Source' role. This can be done manually by adding the Transition's existing Trigger event into the Source, or automated by calling the MetaEdit+ API to complete the model transformation (see Section 3.5). It is also possible to prevent users creating new Triggers in Transitions by making this property read-only in Transition by prefixing its local name there with an underscore, '_'.

We can also make an annotation or report, similarly to scenarios 2, 7 and 8, to highlight the elements that need changing.

If we want to show the values of Trigger visually, the symbol for the Source role could have the same symbol element as in the Transition relationship. See Section 2.3 for the steps to update the generator for the contents of the text element in the symbol.

It is also possible to hide the trigger information visually from the Transition once it has been entered into the Source role. Change the generator in Transition symbol's text element:

```
if ~Source:Trigger; then
   do ~Source:Trigger {:Name; newline :Code}
else
   do :Trigger { :Name; newline :Code }
endif
```

After adding the property, we can update the four generators (names in bold below), to use the trigger from the Source role if given, and otherwise the trigger from Transition:

```
_commands()
foreach .Situation {do :Command {oid newline}}
```

```
_events()
foreach ~Source {
   if :Trigger; then
      do :Trigger {oid newline}
   else
      do >() {do :Trigger {oid newline}}
   endif
}

_target()
do :Trigger {:Name} ', '
do ~Target.() {id}
'State);' newline

!Code()
do _events() where id unique id {
   'Event ' :Name ' = new Event("' :Name '", "' :Code '");'
   newline
}
newline
do _commands()  where id unique id {
   'Command ' :Name 'Cmd = new Command("' :Name '", "'
   :Code '");' newline
}
newline
foreach .Situation; {
   'State ' id 'State = new State("' id '");' newline
} newline
foreach .(); {
   do :Command {
      id;1 'State.addAction(' :Name 'Cmd);' newline
   }
   do ~Source {
      if :Trigger; then
         id;1 'State.addTransition('_target()
      else
         do >() {
             id;2 'State.addTransition('_target()
         }
      endif
      newline
   }
}
```

After these changes, open the State machine for Miss Grant and see that Source roles may have trigger information too. You can generate the code, update trigger information into source roles and generate the code again.

Press **Commit** in the MetaEdit+ launcher to save the changes.

2.11 CHANGE CONSTRAINT

Add Start, then update old Reset constraints to point to Start instead, and add Start into the original Transition binding.

To set things up for a realistic constraint change, we first extend the language a little. A new object type (‘Start’) is added in the Types tab of the Graph Tool for State machine, similarly to

scenario 1. You may also create a symbol for it, e.g. a black filled circle, in a similar way as in scenario 3 for Reset.

Next, we change an existing constraint: the binding that specifies Transitions between Situations is changed to also allow 'Start' in the 'Source' role. This is done in the Graph Tool's Bindings tab. Select the 'Transition' relationship and then the 'Source' role, and in the Objects list add 'Start' (Figure 2-13).

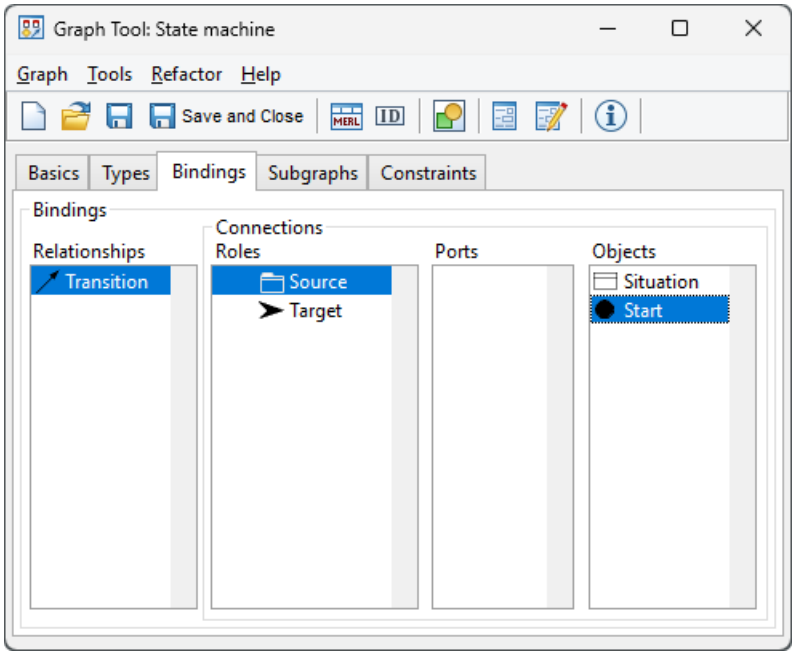

Figure 2-13. Adding binding constraint for Start

We continue by changing two other constraints: the constraints we created in scenario 2 for Reset are updated by changing them to Start. In the Graph Tool's Constraints tab, update the occurrence constraint and the connectivity constraint by double-clicking them and choosing 'Start' in place of 'Reset' as the Object type. These changes are illustrated in Figure 2-14.

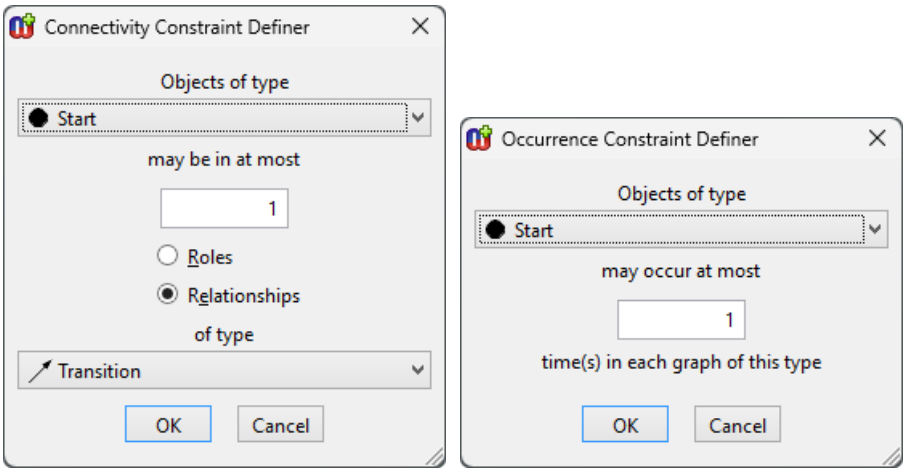

Figure 2-14. Updated constraints for Start.

→ *After this update, the generator called '!Code' needs to be updated to produce code for the Start state: Add the following before the first "foreach .Situation;" line in '!Code'.*

```
foreach .Start; {
   'State StartState = new State("start");' newline
}
```

After these changes, test them by re-opening the state machine for Miss Grant and see that Start has been added to the language and is visible in the toolbar and tree view. Add one Start to the diagram, connect it to a Situation and run the generator.

Press **Commit** in the MetaEdit+ launcher to save the changes.

2.12 CHANGE NOTATION

Make the Situation symbol refer to a different library symbol.

Normally in MetaEdit+ the notation is changed by changing it directly in the Symbol Editor. Since here we want a change in a reference, from one metamodel element to another, we again use the symbol library reference.

First, select 'Situation' on the Types tab of the Graph Tool for State machine, and choose **Edit Symbol** from its pop-up menu. Select the blue template element in the left list as in Figure 2-15. Next, open its pop-up menu and choose **Format…** to open the formatting dialog.

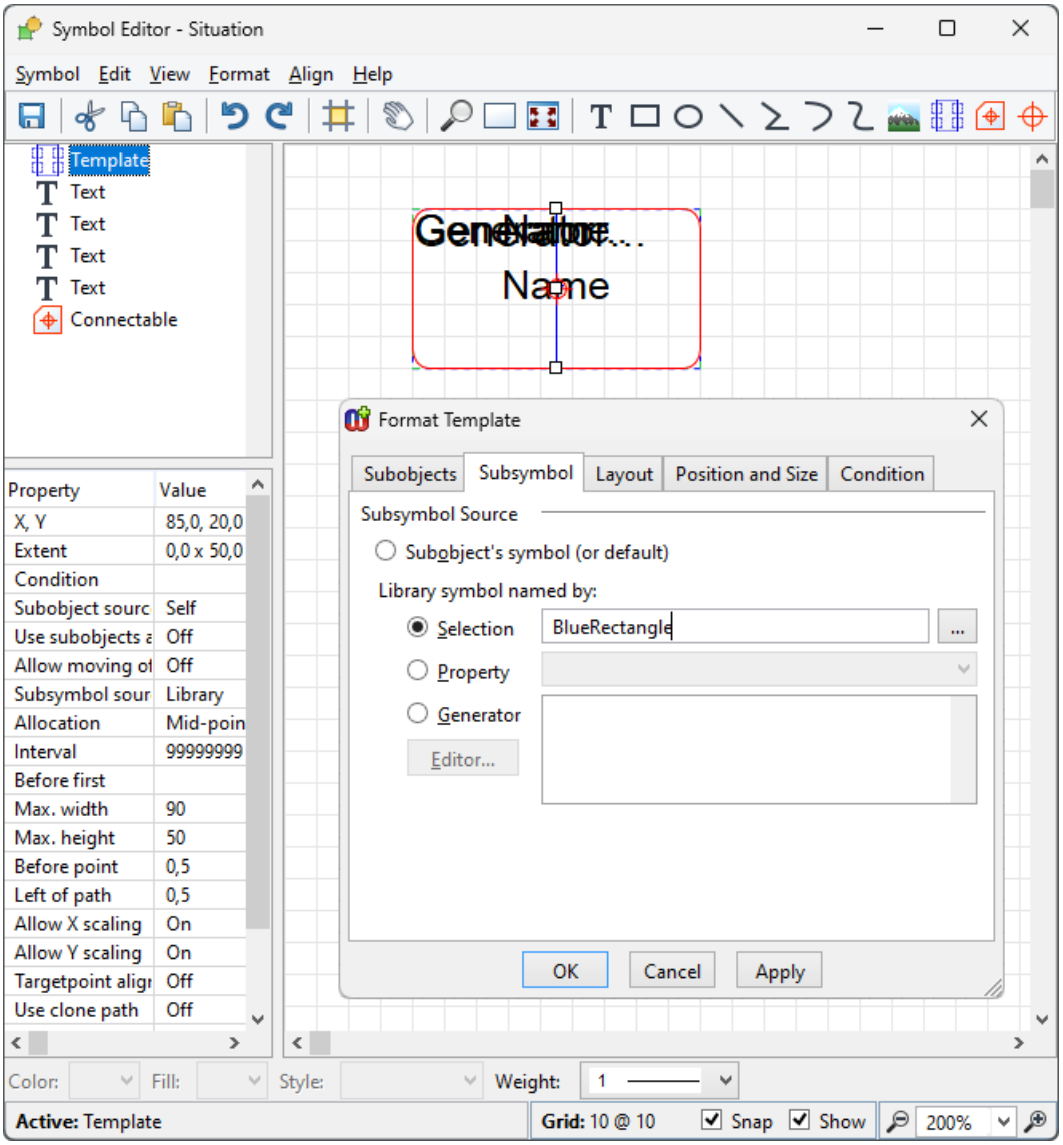

Figure 2-15. Changing library symbol for 'Situation'.

In the Subsymbol tab, click the button with three dots next to Selection and choose ‘BlueRectangle’ from the library. Press **OK** in the dialog and close the Symbol Editor, saving changes.

Next, open the State machine for Miss Grant and see that Situations are now represented with a blue rounded rectangle, as defined in the library symbol.

Press **Commit** in the MetaEdit+ launcher to save the changes.

3 Co-evolution on an industrial scale

So far, our changes have been made to a small language with one example model, in the single-user version of MetaEdit+. Next we will describe how to co-evolve the language on an industrial scale with a larger number of models, languages, users and repositories as well as using automated model updates via the MetaEdit+ API.

3.1 LARGE MODELS

In MetaEdit+, models update in the same way whether there is a single graph like Miss Grant or thousands of them in the repository. With each operation on the language definition, the models currently loaded are automatically updated to match the structure of the new language version, as described in the scenarios of Chapter 2. Models that are not currently loaded in memory are updated automatically when they are loaded. For most changes, the update is performed lazily, only being saved when the model element must be saved — such as when it is modified during regular modeling activities.

3.2 MULTIPLE LANGUAGES

With several modeling languages (different Graph types), the co-evolution would remain similar to Chapter 2. If an element in the metamodel is used across multiple languages, its links to other languages can be traced with the Info Tool **(Tools | Info Tool**), showing which other elements of the metamodel use it and which other metamodel elements it uses.

3.3 MULTIPLE USERS

In multi-user repositories, changes to the language become visible to other users only when a new transaction is started (such as logging in, making a commit or abandoning a change). At that point the models are updated as well. This allows the language engineer to freely modify the language and test its impact on existing models and generated outcomes. Only after pressing **Commit** are the changes saved and made available to others.

Additionally, in MetaEdit+ access rights can be set to say which users can modify metamodels, or restrict certain projects to read-only modeling rights for most users, allowing only certain users to modify models there. See the System Administrator's Guide for details.

3.4 MULTIPLE REPOSITORIES

In cases with large numbers of repositories, it is prudent to separate language development into its own repository. The language changes can then be tested offline. The language

development repository can contain (or import as necessary) models specifically created for testing, as well as a cross-section of production models, to verify the results of updating to and using the new language definition.

Language definitions covering metamodels, constraints, symbols, generators and user interface settings can be exported and imported to repositories in binary and XML formats.

Using multiple repositories is also an effective way to manage different versions of products being developed. For example, one repository can be dedicated to a product in the maintenance phase, containing many components, such as frameworks, compilers, or even hardware, that are frozen. In such cases, the language may not need to be changed either.

3.5 API FOR AUTOMATING TRANSFORMATIONS

The implementation of the co-evolution scenarios demonstrated that MetaEdit+ largely eliminates the need to create transformations to co-evolve models. This is mainly achieved through automatic, built-in co-evolution of models and modeling tools, with the remaining complexities simplified by following the established practice of deprecation rather than deletion.

The use of deprecation, like using older model data for generators while preventing their creation, makes co-evolution easy and fast for language users. However, it is also possible to automatically transform models via the MetaEdit+ API, enabling updates to both model data and its representations. While the API was not used in the co-evolution cases described earlier, its functionality is detailed in the MetaEdit+ Workbench Users Guide. For co-evolution scenarios 7, 8 and 10 the API could be applied to automate model changes.

The MetaEdit+ API can be called from virtually any language. Below is an example in C#, automating model updates in scenario 10 by moving Triggers from Transition relationships to Source roles.

```
METype graphType        = new METype() { name = "State machine" };
METype relationshipType = new METype() { name = "Transition" };
METype roleType         = new METype() { name = "Source" };

MetaEditAPIPortTypeClient api = new MetaEditAPIPortTypeClient();

foreach (MEOop graph in api.allGoodInstances(graphType))
{
  foreach (MEOop transition in api.contentsMatchingType(graph,
relationshipType, false))
  {
    MEOop[] sources = api.rolesForRel(graph, transition, roleType);
    MEAny trigger = api.valueForLocalName(transition, "Trigger");
    api.setValueForLocalName(sources[0], "Trigger", trigger);
  }
}
```

4 Conclusion

In this guide, we have demonstrated how MetaEdit+ supports the co-evolution of modeling languages and models. With MetaEdit+, your work is never lost, editors do not break, and the effort required to refine the language to meet new needs is minimal.

Most of the co-evolution scenarios in MetaEdit+ are achieved through the automatic, built-in co-evolution of models and modeling tools, with the remaining complexities simplified by following the established practice of deprecation rather than deletion.

Making language evolution simple, low-effort and low-risk can better ensure that languages evolve to continue meeting development needs. This contrasts with approaches and tools that require implementing and applying transformations each time the language changes. The knowledge and effort required to evolve the language is also modest, as demonstrated by being able to carry out the 12 different scenarios in a short time even as a new user.

This simplicity, enabled by supporting co-evolution as a fact of life from the start of MetaEdit+ use, has been proven in industrial use in cases up to hundreds of users and dozens of years.