An ECLIPSE Framework for Rapid Development of Rich-featured GEF Editors based on EMF Models - Long Version -

Tony Modica, Enrico Biermann, Claudia Ermel
Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin
E-Mail: {modica,enrico,lieske}@cs.tu-berlin.de

Abstract: Model-based development has an increasing importance in modern software engineering and other domains. Visual models such as Petri nets and UML diagrams proved to be an adequate way to illustrate many structural and behavioral system properties. However, while tooling for textual modeling is pretty mature now, visual tool builders are faced with a much higher complexity regarding the representation of model properties, and the interplay of the concrete syntax (the views) with the underlying abstract model representation, e.g. based on Java, XML or the Eclipse Modeling Framework (EMF). In order to ease the development of visual editors, the Graphical Editing Framework (GEF) offers layout and rendering possibilities, as well as an architecture that allows to integrate models based on EMF, Java or XMI with their visual views and editors. Unfortunately, the structure of GEF is quite complex to use if editors are not simply one-to-one representations of model elements, or if more than one view is needed at a time for more complex models.

Based on several years of experience in teaching the development of GEF-based visual editors for complex visual models to students, we developed MUVITORKIT (Multi-View Editor Kit), a framework for rich-featured visual editors, which is presented in this paper. MUVITORKIT is based on EMF and GEF, and supports nested models, models needing multiple graphical viewers, and animated simulation of model behavior. The architecture of MUVITORKIT is designed in a way that encapsulates the complex underlying mechanisms in GEF and simplifies the integration in the ECLIPSE workbench.

1 Introduction

Model-based development grows more and more important in modern software engineering. For a long time, visual models were restricted to pencil drawings on paper, used in the early software development phases to illustrate structural and behavioral system aspects. While textual modeling had been supported by tools pretty early, visual modeling tools have been developed much later. Tool builders of visual modeling tools still face a number of problems such as layouting, pretty-drawing, view management and version control. The ECLIPSE Graphical Editing Framework (GEF) allows editor developers to implement graphical editors for existing models. GEF provides the layout and rendering toolkit DRAW2D for graphics and follows the model-view-controller (MVC) architectural

pattern to synchronize model changes with its views and vice versa.

Our research group has its main focus on applying formal techniques to visual modeling languages. For five years now, we have held a visual languages programming project (VILA) for graduate students, concerning the development of a graphical editor as an ECLIPSE plug-in, using GEF and the Eclipse Modeling Framework (EMF).

Generally, with a complex framework as GEF there are always many different possibilities to approach the implementation of a feature, but usually there are better and worse ways, especially if you want to reuse code later on. Several frameworks support generation of code for from abstract editor specifications, like the ECLIPSE Graphical Modeling Framework (GMF) or MOFLON [AKRS06]. Unfortunately, the visual languages we usually want to implement editors for and their simulation operations seem not appropriate to be specified in these frameworks, which assume models to be displayed in a single pane only. Moreover, if we used e.g. GMF to generate code as far as possible, GEF apprentices without deeper knowledge of the mechanisms in GEF would surely struggle when laying hand on the generated code to extend it with complex features.

Some experiences we made Every year, we choose a specific visual language of our current research, i.e in the first two years a new visual notation for OCL constraints and later more complex models that integrate approaches for different aspects of a system, e.g. class diagrams and typed graphs (structure), Petri nets and activity diagrams (behavior), and graph or Petri net transformation rules (behavior and reconfiguration).

In the first phase of each project, we had to deal with the steep learning curve of GEF and ECLIPSE programming. We did not want to spend much time on occupying the students with analyzing solely the example editors shipped with GEF and producing simple editors that they could not use directly for the project's goal, because the example editors did not use generated EMF models. Besides, in the early years of GEF, the only available documentation was the IBM *RedBook* [MDG+04] (recently, a growing number of introductory tutorials and overviews have been made freely accessible). We let the students adapt a simple editor for a similar model, e.g. a graph editor to a Petri net editor, or vice versa, to get an impression of the main principles while modifying the existing code to work on another newly generated EMF model. Unfortunately, we noticed that many students were discouraged by the complexity of the code or/and were not able to distinguish code for the editor functionalities from code that is simply needed to integrate the editor into the ECLIPSE workbench. Moreover, with the more complex models, requirements arose that could not be fulfilled by GEF's default editor implementation, which has just one panel to operate on, showing just one graphical viewer at a time.

Another problem stemmed from the project scheduling, which stipulates starting with easy-to-build components. For example, we cannot simply start to develop a particular editor component if we first need a more complex component to create elements for the simple component to edit, at all. Unfortunately, reducing the EMF model to the parts we actually want to build an editor component for is not a practical solution, because the integration and adaption of such a component to the whole EMF model with encompassing model elements turned out to be very tedious and error-prone.

A new framework We used our past experiences to generalize recurring code fragments for many editor features and to document them properly for simplifying the familiarization process for the students as well as the editor implementation. This development lead to our GEF-based framework MUVITORKIT (Multi-View Editor Kit). MUVITORKIT supports nested models, models needing multiple graphical viewers, and animated simulation of model behavior. The architecture is designed in a way that encapsulates complex underlying mechanisms in GEF and simplifies the interaction with the ECLIPSE workbench.

used our past experiences to generalize recurring code fragments for many editor features into code templates and to document them properly for simplifying the familiarization process for the students as well as the editor implementation. This development lead to our GEF-based framework MUVITORKIT (Multi-View Editor Kit). MUVITORKIT supports nested models, models needing multiple graphical viewers, and animated simulation of model behavior. The architecture is designed in a way that encapsulates complex underlying mechanisms in GEF and simplifies the interaction with the ECLIPSE workbench.

We present the MUVITORKIT framework in this article, which is structured as follows: In the next section we give a short overview over the requirements that rise from the models we usually want to implement editors for, and which are supposed to be supported by MUVITORKIT. In Section 3, we present actual implementations based on MUVITORKIT. Section 4 describes the architecture of MUVITORKIT, its advantages and how to use it. The fifth section presents a package of MUVITORKIT that allows developers to define flexible animations in editors based on it. In the conclusion we mention some future work on MUVITORKIT. The appendix contains sample code fragments for selected presented features.

2 Functional and Pragmatic Requirements

GEF in combination with EMF models is adequate for building editors with a single panel showing one diagram at a time. However, for visual languages whose components do not only consist of single graph-like diagrams, we have to come up with additional mechanisms to manage the different components of such models[1].

In this section we give an overview of the main additional editor concepts we need to support in order to enable visual editor users to edit their models conveniently. For illustration, we introduce some recurring components of visual languages from our VILA projects.

Nested Models Fig. 1 on the following page shows an example of a special kind of high-level Petri nets, i.e. a *higher-order* Petri net. Here, the token *WF* on place *Workflow* is a (simple) Petri net itself[2]. According to [Val98], we say that the (shaded) *system net* contains the *object net WF*. Thus, for a higher-order net editor, we need some facility to access nested objects like the *object net* tokens, e.g. to open a special editor component

[1] We call the instances of a visual language simply 'models' in this article, in contrast to the notion 'EMF model', which is in fact a meta-model describing the visual language.

[2] You may ignore other details of this net for now.

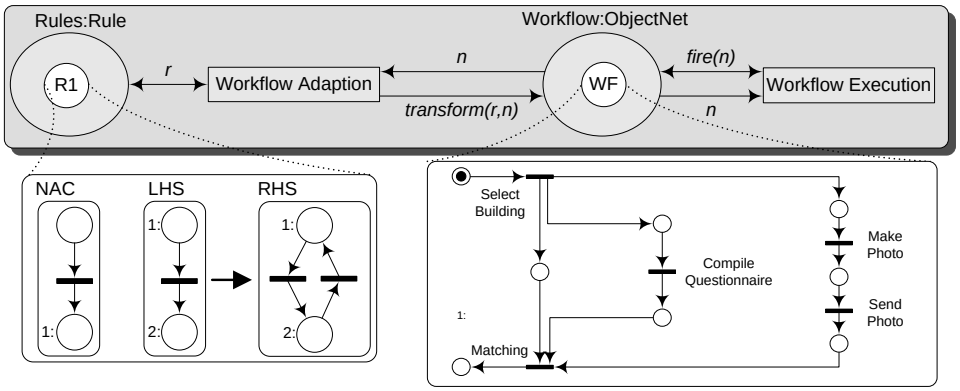

Figure 1: Example higher-order Petri net

for editing simple Petri nets in addition to one for the high-level *system net*.

Components with more than one Graphical Viewer In the left of the *system net* in Fig. 1, you can see another complex token *R1* that is not a Petri net but a transformation rule for Petri nets [HME05]. Such rules consist of a left-hand (LHS) and a right-hand side (RHS), similar to formal grammars for strings. Additionally, there may be some negative application conditions (NAC), which can prevent rule applications in certain cases.

If we want to build an editor component for rules, we have to integrate three single panels into it and to provide means for specifying the relations of the rule's elements (here indicated by the small numbers next to the places) and managing the NACs.

In addition, we surely want to be able to check the *system net* and its *object nets* while editing a rule for a special purpose. Therefore we cannot simply overlay the main editor's panel with the net/rule/token we want to edit, we rather need a parallel simultaneous presentation of the different editor components similar to Fig. 1.

Animated Simulation Petri nets are behavioral models and we want the editors not only for editing but for simulating them as well, i.e. the firing steps, which consume and produce tokens. Continuous animation of the involved tokens would let the user comprehend the performed firing step better than simple switching to the new system's configuration. So, we need an easy to integrate mechanism to state that consumed token's figures should move (by animation) from the incoming places to the firing transition and vanish, whereas the newly created tokens should appear at the firing transition and then move to their corresponding places. For an example, see Fig. 2 on the next page where the dashed arrows show the animation paths that the consumed and the created tokens will follow during a firing step of the transition.

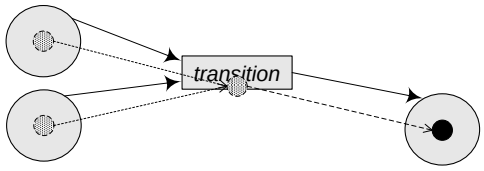

Figure 2: Example animation for a firing step in a Petri net

Support Stepwise Development and advise Good Practices In addition to the former functional requirements derived from specific visual language features, we also want to deal with the difficulties of getting acquainted with ECLIPSE and GEF and nevertheless writing coherent code. For this, we would like to support the students by suggesting practices we found useful for producing code that is easy to extend and to maintain.

3 Sample Editors based on MUVITORKIT

In this section, we present two editors based on the MUVITORKIT to demonstrate how to realize our multiple view requirements. The editors are working on complex models that integrate approaches for different aspects of a system, e.g. typed graphs, Petri nets, and activity diagrams (structure); and graph or Petri net transformation rules (behavior and reconfiguration).

3.1 RONEditor for Reconfigurable Object Nets

The visual language of this editor, Reconfigurable Object Nets (RON), are just a variant of the higher-order nets in the previous section, simplified for the VILA students' project [BEHM07]. RONs are higher-order system nets consisting of transitions (without algebraic expressions as before, but with fixed semantics) and of places that carry object Petri net and rule tokens. The editor is freely available [RON].

Fig. 3 on the following page shows a screenshot of the editor with an example RON (a producer-consumer model for distributed producers and consumers, see [BEHM07] for a detailed explanation). In the left, you see the main tree editor component (1), showing all elements as transitions, places, and tokens on places. (2) is the graphical component for editing the RON *system net*. If you open the editor on a RON file, it will immediately show (1) and (2). This, displaying the tree editor and the graphical view for the main element (the 'top' element of the model), is the default behavior in all MUVITORKIT implementations.

Consider the place *Producers* in (2). If you double-click on the token *Prod1*, GEF triggers an 'open' request, which by default MUVITORKIT handles such that the graphical

component (3) for the object Petri net will be opened. Similarly, ‘opening’ the rule token *mergePC* on place *MergeRules* causes the rule editor component (4) to appear. Note that this component has a single tool palette shared by three graphical viewers, showing the NAC, LHS, and RHS of the rule.

Opening graphical views for model elements is also always possible in the main tree editor component. The user may double-click entries in the tree to open corresponding graphical views, such (3) and (4).

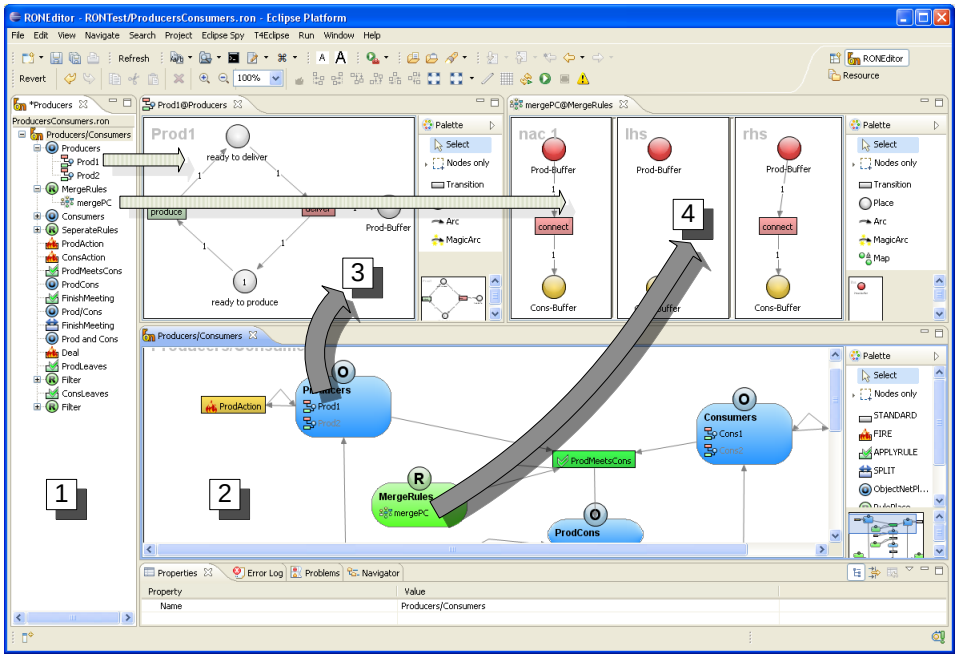

Figure 3: Screenshot of the RONEditor

All components (1)-(4) are regular ECLIPSE views that can be arranged in any way. Moreover, the RONEditor demonstrates how to define a perspective, so that the components for rules and object nets are automatically arranged as depicted and, if multiple views of the same kind are opened, they are gathered with tabs as the views for the object nets *Prod1*, *Prod2*, and *Cons1*.

3.2 ActiGra Editor

In the ActiGra model we combined activity diagrams with graph transformation rules as activities. When simulating an activity diagram, the correspondent rules will be applied to a graph representing some data. In Fig. 4 on the next page, we have again in (1) the all-

encompassing tree view where you can open views for the data graphs, e.g. for *PizzaOrder* in (2), and for activity diagrams, e.g. for *orderPizzaDiagram* in (3). Note that rule views, as for *orderBeverage* in (4), are accesible via the tree view as well as via the activity nodes, e.g. (4) could have been opened by double-clicking the corresponding activity in (3).

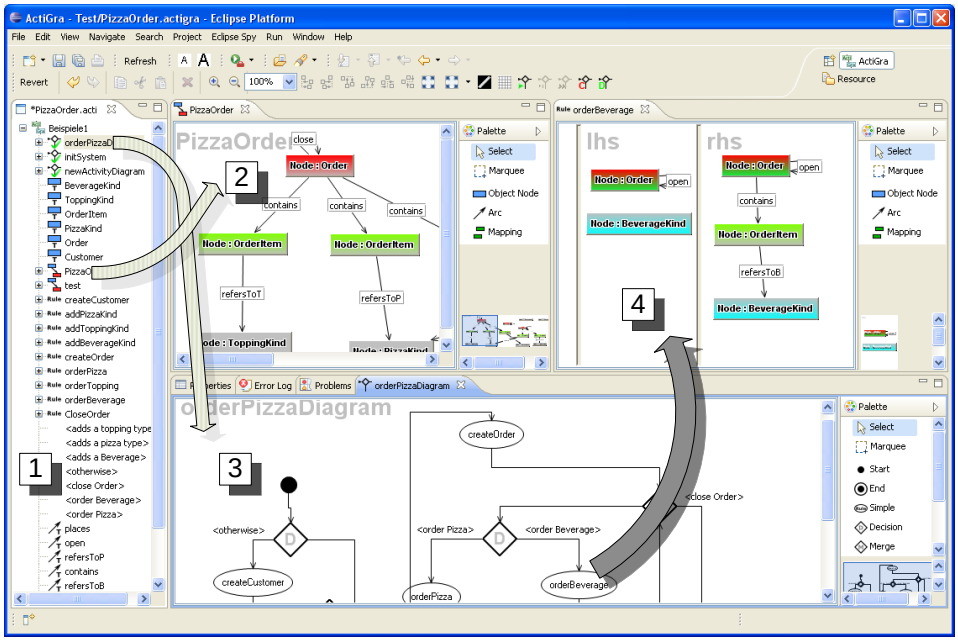

Figure 4: Screenshot of the ActiGraEditor

4 Structure and Features of the MUVITORKIT

In this section, we describe the MUVITORKIT, its main parts, and how they can be used to quickly build a GEF editor meeting our previously mentioned requirements.

In general, implementing a GEF editor involves two main tasks. On the one hand, you have to use the GEF architecture to implement `EditParts` as controllers that mediate between a model element and the view part, which are DRAW2D figures, according to the MVC principle. For our models, we use EMF to design the model and to automatically generate code featuring i.a. a qualified notification mechanism. On the other hand, the editor has to be integrated properly into the complex ECLIPSE workbench. To keep the second task as simple as possible, MUVITORKIT contains abstract classes for building editors and graphical classes with GEF's editing capabilities, which need only the GEF-specific information to be implemented. In short, most parts that are not GEF-related and integrate the editor into the workbench are already configured reasonably in MUVITORKIT to provide

many features to every MUVITORKIT implementation with little effort for the developer. Nevertheless, the editor can be controlled via well-documented special methods in the abstract classes, following our requirement for encapsulating good practices in simple-to-use methods[3]. Note that, in contrast to GMF, where you specify and generate an editor via an abstract model, the aim of MUVITORKIT is to help users with easy-to-use and extendable default implementations for building complex GEF editors.

Most generalizations of editor features are only possible because we assume to have a generated EMF model and make use of the generated code's special features. The whole MUVITORKIT framework and its parts are tailored to be used together with an EMF model.

We describe the different parts in detail to give an impression about building an editor based on MUVITORKIT. At first, let us state a nomenclature: In ECLIPSE, all workbench components are called 'views' and an editor is just a special kind of view with an input. In MUVITORKIT, there is always only one class implementing the ECLIPSE interface `IEditorPart` (see next paragraph), so in the following, we will refer to 'views' as the graphical components that are not the main tree-based view, which we call the *editor*. In a view or an editor, there may reside several GEF 'viewers', which actually display the graphical or tree-based representation of the model for editing, and that must not be confused with 'views'! In general, view classes have to be registered in the plug-in's configuration file *plugin.xml* with some identifier to be used in the workbench.

Main Tree Editor The central part of an editor based on MUVITORKIT is an implementation of the abstract class `MuvitorTreeEditor`. It integrates a basic but comprehensive tree-based editor component into the workbench, as a base to access all the graphical views for specific model elements from. As for all ECLIPSE editors, this class has to be registered with the editor extension point of the editor plugin you want to build (i.e. in *plugin.xml*).

Implementing a subclass of `MuvitorTreeEditor` is basically connecting the components responsible for the GEF parts to the editor. For this, there are several abstract methods in which you need to instanciate the following parts: a default EMF model instance (for empty or corrupt files); an `EditPartFactory` for assigning GEF `EditParts` to EMF model elements; a `ContextMenuProvider` for the tree editor component, and optional some custom actions. See the appendix A.1 for a sample code fragment realizing the RONEditor. Implementing the edit part factory and the context menu provider is pure GEF developing; there is no need for the developer to deal with workbench integration at this point.

There are other methods to realize further editor features as well, but this is the necessary set. In addition, you need to associate the types of EMF model elements (e.g. graphs, Petri nets etc.) that you want to display graphically to appropriate views. We will see later how to implement these views and how they can be opened via the `MuvitorTreeEditor`'s *showView* mechanism.

All `MuvitorTreeEditors`, even resulting from minimal implementations, have the

[3] In this way, the students can first simply use the API (after we advised them of the corresponding methods) to implement basic features and later analyze and extend parts of the code selectively with more complex features.

following features:

- When opened, the editor activates a perspective (general layout for the views on the workbench) if optionally registered in *plugin.xml*. When closed, it restores the previous perspective and remembers all currently opened graphical views. These are reopened when the editor is activated again.
- A manager for persistency operations on EMF models is provided and handled properly. No additional load and save implementations are needed.
- Most generic actions provided by ECLIPSE and GEF are automatically installed to the action bars as save, save as, undo, redo, print, direct edit, delete, alignment, revert. `MuvitorTreeEditor` automatically takes care about updating the states of these actions properly. A basic revert mechanism allows to reload the model from the current file.
- If the developer lets the editor create special problem markers, editor users may 'open' these in the ECLIPSE Problem View, which causes the editor to be activated (if necessary) and to display and to select the problematic part of the model via the *showView* mechanism.
- Registering the class `MuvitorFileCreationWizard` with the new wizard extension point in *plugin.xml* is all you need for a wizard dialog that creates a new empty file. `MuvitorTreeEditor` fills this with the empty default model. The file extension to which the editor is bound is retrieved automatically from *plugin.xml*.
- Several special technical adjustments are made for better support of the multi view concept, e.g. to keep the editor's action bar enabled if the user selects something in a graphical view.

Graphical Views Using MUVITORKIT, the main editor can open different views for certain types of model elements (e.g. for the graphs, Petri nets, rules etc.) that support an arbitrary number of (GEF) viewers hosted inside a view (e.g. a view for transformation rules containing two viewers displaying the rule's LHS and RHS, maybe even with a third one displaying a NAC).

`MuvitorPage` is an abstract implementation of an ECLIPSE view that is prepared to be opened via the method `MuvitorTreeEditor.showView(EObject)`[4]. If the EMF model type of the passed `EObject` (i.e. its `EClass`) has been assigned to the identifier of a registered (in *plugin.xml*) view, calling this method will open the correponding view and make the `EObject` accessible for this view. See Fig. 5 on the following page for a diagram outlining how the `MuvitorTreeEditor` interacts with the ECLIPSE platform

[4]Strictly speaking, `MuvitorPage` itself is not an ECLIPSE view (implementing `IViewPart`) but it contains all the editing functionality. MUVITORKIT provides an abstract class `MuvitorPageBookView`, which has to be implemented for each `MuvitorPage` and has to be registered in the *plugin.xml*. It is very easy to implement and its sole purpose is to host `MuvitorPage` instances. This is a rather technical construction ECLIPSE demands, so in the following we consider the `MuvitorPage` as the view.

and workbench to open a `MuvitorPage` for a model element. This mechanism makes use of the EMF notification mechanism for automatically closing views showing elements that have been deleted in the model or whose hostig (tree) editor has been hosting closed. Alternatively, you may call `closeViewsShowing(EObject)`, as well.

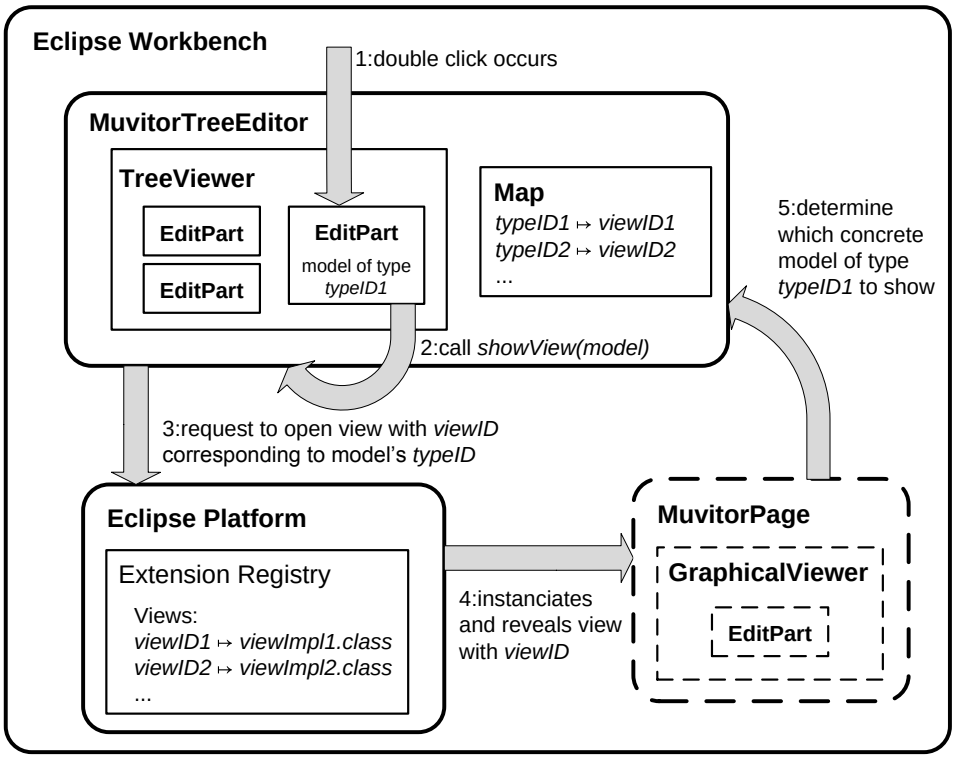

Figure 5: How to open graphical views in MUVITORKIT

In principle, `MuvitorPage` is very similar to the default editor implementation GEF provides, but changed to be integrated in the workbench as a view instead of as an editor. The main additional contribution of `MuvitorPage` is to handle the changes of the currently active GEF viewer of this page for proper integration into the main editor and the workbench. With this, you can e.g. change the zoom level of each viewer independently via a single action in the editor's main toolbar.

Implementing a `MuvitorPage` is very similar to subclassing `MuvitorTreeEditor` before; it defines again the vital parts for GEF viewers, in this case graphical ones, like an `EditPartFactory`, a `ContextMenuProvider`, and optionally some actions that may be shared with the main editor. For the graphical viewers, we need additionally a palette for the editing tools[5]. The most important abstract method requires to return a list of

[5] `MuvitorPaletteRoot` is a default palette for MuvitorPages, which encapsulates and documents installation of new `ToolEntries` in a flexible method.

`EObjects` as viewer contents, each to be displayed in an own viewer on this page. For example to show a rule as mentioned before, if the view has been opened via *showView(rule)*, the returned list must contain *rule.getNAC()*, *rule.getLHS()*, and *rule.getRHS()*. See the appendix A.2 for a sample code fragment realizing the RONEditor's rule component.

Furthermore, there are additional API methods e.g. for hiding single viewers, e.g. if you delete the NAC of a rule.

The main additional contribution of `MuvitorPage` is to handle the changes of the currently active GEF viewer of this page for proper integration into the main editor and the workbench. With this, you can e.g. change the zoom level of each viewer independently via a single action in the editor's main toolbar.

`MuvitorPages` are designed to be able to reuse actions that have been created in the main editor, like the generic copy and paste action in `RulePage`, and updating actions is handled similarly to the `MuvitorTreeEditor`.

Furthermore, there are additional API methods e.g. for hiding single viewers, e.g. if you delete the NAC of a rule.

By implementing `MuvitorPage` you get features for free again, in this case a number of generic actions that work as they are in every single GEF viewer of the page:

- ExportViewerImageAction exports the components contents into a graphic file. TrimViewerAction moves the figure in the upper left corner so that the viewer's size is minimized but still showing all figures.
- GenericGraphLayoutZESTAction and GenericGraphLayoutAction apply the ZEST or DRAW2D graph layout algorithms to the selected component.
- MoveNodeAction changes the location of selected nodes by keys.
- SelectAllInMultiViewerAction selects all parts in the current viewer.
- GenericCopyAction copies any EMF model in form of a serialized String into the system's clipboard. GenericPasteAction pastes the clipboard into the current edit part's EMF model if allowed. Supports undo operation and derinition of flexible PasteRules, e.g. to prevend rule tokens to be pasted into *system net* places for *object nets* and vice versa.
- MuvitorToggleGridAction and MuvitorToggleRulerVisibilityAction toggle the grid and ruler visibility of a viewer.

Implementing `EditParts` as Controllers Now that we have the main editor and the views, we need to some place to invoke the *showView* mechanism. For this, MUVITORKIT offers its own extended `EditParts` called `AdapterEditParts`, which are the controllers of the elements displayed in the GEF viewers and which are supposed to be used exclusively in MUVITORKIT. Besides other enhancements[6], by default they call *showView*

[6] E.g. to support closing the views for deleted models, and providing a generic `EObjectPropertySource` for the Properties View, showing all of the model's attributes and their values.

on their model if they receive GEF's open request, which is dispatched to an edit part when a double-click occurs on it (see the first step in Fig. 5 on page 10). This is the key to the behavior of the MUVITORKIT examples we described in Sect.3: we configure the views correctly and associate them with the types of elements they are supposed to display and we use these special `AdapterEditParts`. The types of models (rules, nets etc.) can be arbitrarily nested and we can navigate as deep as we want, as long as we have an edit part for a closed representation of each nested part, like a rule token, and a view for its detailed representation like the rule view.

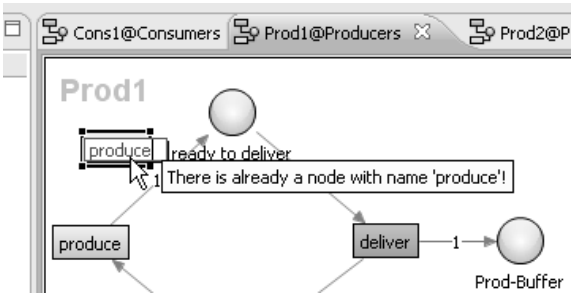

Figure 6: Direct Editing with validator for unique node names

An editing feature, useful for almost all edit parts, is direct editing of a name or some other attribute value. In previous projects, we commissioned one of the student groups to give a little lecture about how to implement this features as this is not a trivial task. With presuming an EMF model, we could generalize and encapsulate this feature to all `AdapterEditParts`. All we need to do is to let a custom edit part implement our interface `IDirectEditPart`, which just has one mandatory method that returns the EMF identifier of the model's attribute that should be edited[7]. This is all information the edit part needs to run a direct edit manager as in Fig. 6. Optionally, you may specify a validator checking the input and generating an error message as in this picture.

Some additional supporting MUVITORKIT classes `MappingCreationTool` is a special GEF `ConnectionCreationTool`, allowing connections between parts in different viewers like the relation of places and transitions in the Petri nets of a transformation rule. In contrast to the regular `ConnectionCreationTool` this tool remains in valid state when the mouse leaves the viewer's border.

Registering the class `MuvitorFileCreationWizard` with the new wizard extension point in *plugin.xml* is all you need for a wizard dialog that creates a new empty file. `MuvitorTreeEditor` fills this with the empty default model. The file extension to which the editor is bound is retrieved automatically from *plugin.xml*.

The class `IconUtilTemplate` proposes a universal way to access image files in the project path via definable keys and uses the caching mechanism of the plug-in's `ImageRegistry`.

[7] e.g. `EcorePackage.ENAMED_ELEMENT__NAME`

5 A Package for Defining Functional Animations

MUVITORKIT supports the definition of continuous animations for selected model elements. The main class in the animation package is `AnimatingCommand`, which can be given information about how some model elements (EMF `EObjects`), visualized by figures in a graphical viewer, should be animated[8]. For this, you specify stepwise absolute (`Points`) or relative (some model elements) locations and optional size factors for the model elements to be animated.

Why not use GEF's Animation Mechanism? GEF also contains classes `Animator` and `Animation`, which can be used to animate figures, i.e. sliding a figure on a straight path to the new location it is set to. The following main advantages of using MUVITORKIT's animation package instead allow to specify more powerful animations more flexibly:

- You can specify which *model element* to animate instead of the corresponding *figure*. This means that you can specify animations independently from actual viewers and model states, and rely on the package's mechanism to find the corresponding figures automatically at animation runtime.
- Several elements can be animated independently along complex paths with flexible timing/speed (by interpolating intermediate steps). You can even alter the animation paths to sine, circle, or elliptic curves.
- Each animated figure can be zoomed smoothly while it moves on its path, according to absolute size factors. Such figures resize independently of all other (possibly animated) figures in the same viewer.
- You can easily specify parallel animations in several independent viewers.
- You can easily integrate your animation definitions into the `Commands` your editor invokes to change the model on editor operations. Moreover, `AnimatingCommand` automatically takes care of reversing performed animations to support the 'undo' operation.

We provide a detailed tutorial example of the documentation of `AnimatingCommand` [RON] in appendix B.

[8] According to the MVC principle, we strictly distinguish a model element (some EMF `EObject` instance) from its 'figure', which is just its graphical representation in GEF viewers

6 Conclusion

We presented MUVITORKIT, a framework to facilitate the implementation of rich-featured GEF editors, and described the important aspects of this framework's main classes and how they can be used to build editors for complex visual languages consisting of different nested components. Many more details about benefits of using MUVITORKIT can be found in the Java documentation of its classes.

We are permanently extending the MUVITORKIT. When our students come up with useful features in the projects or have feature requests, we try to generalize them in MUVITORKIT if possible or to include at least a documented example implementation in the RONEditor.

Future Work We plan to extend the MUVITORKIT by further features and to eliminate some minor deficiencies:

- When developing custom editor plugins based on MUVITORKIT, each editor has to reference its own exclusive copy of the MUVITORKIT plugin in its dependencies. This is due to the fact that the mechanisms accessing the editor's plugin registry are located in MUVITORKIT. Generalization is possible but would impose more preparation steps when implementing a custom editor on the developer.
- The animation package is going to be restructured to support more animation-related aspects like highlighting figures and annotating them with other figures like labels during animation.

References

[AKRS06] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In A. Rensink and J. Warmer, editors, *Model Driven Architecture - Foundations and Applications: Second European Conference*, volume 4066 of *Lecture Notes in Computer Science (LNCS)*, pages 361–375, Heidelberg, 2006. Springer Verlag, Springer Verlag.

[BEHM07] E. Biermann, C. Ermel, F. Hermann, and T. Modica. A Visual Editor for Reconfigurable Object Nets based on the ECLIPSE Graphical Editor Framework. In *Proc. 14th Workshop on Algorithms and Tools for Petri Nets (AWPN'07)*. GI Special Interest Group on Petri Nets and Related System Models, 2007.

[HME05] K. Hoffmann, T. Mossakowski, and H. Ehrig. High-Level Nets with Nets and Rules as Tokens. In *Proc. of 26th Intern. Conf. on Application and Theory of Petri Nets and other Models of Concurrency*, volume 3536, pages 268–288. 2005.

[MDG+04] B. Moore, D. Dean, A. Gerber, G. Wagenknecht, and P. Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Corporation, International Technical Support Organization, 2004.

[RON] Website of the RONEditor and MUVITORKIT. `http://tfs.cs.tu-berlin.de/roneditor`.

[Val98] R. Valk. Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In *ICATPN '98: Proceedings of the 19th International Conference on Application and Theory of Petri Nets*, volume 2987 of *LNCS*, pages 1–25. Springer, 1998.

A Sample code fragments from the RONEditor

A.1 Example for RONTreeEditor

```
public class RONTreeEditor extends MuvitorTreeEditor {
static String objectNetViewID = "ObjectNetPageBookView";
static String ruleViewID = "RulePageBookView";
static String ronViewID = "RONPageBookView";

// define the views for specific EMF model elements
{ registerViewID(RonmodelPackage.Literals.RON, ronViewID);
  registerViewID(RonmodelPackage.Literals.RULE, ruleViewID);
  registerViewID(RonmodelPackage.Literals.OBJECT_NET,
      objectNetViewID);
}

// create default model for     empty or corrupt files
protected EObject createDefaultModel() {
  RON newRon = RonmodelFactory.eINSTANCE.createRON();
  newRon.setName("<default>"); return newRon;
}

// factory for assigning GEF EditParts to model elements
protected EditPartFactory createTreeEditPartFactory() {
  return new RONTreeEditPartFactory();
}

// define a context menu for the tree editor component
protected [...] createContextMenuProvider(TreeViewer v) {
  return new RONEditorContextMenuProvider(v, getActionRegistry());
}

// create some additional actions for the editor
protected void createCustomActions() {
  registerAction(new GenericCopyAction(this));
  registerAction(new GenericPasteAction(this));
} } // end class
```

A.2 Example for RulePage

```
public class RulePage extends MuvitorPage {
```

```
// define a context      menu for the rule editor
protected [...] createContextMenuProvider(EditPartViewer v){
  return new RulePageContextMenuProvider(v, getActionRegistry());
}

// share additional actions from the editor
protected void createCustomActions() {
  registerSharedActionAsHandler(ActionFactory.COPY.getId());
  registerSharedActionAsHandler(ActionFactory.PASTE.getId());
}

// define a palette with editor tools
protected MuvitorPaletteRoot createPaletteRoot() {
  return new RulePaletteRoot();
}

// this array determines number and contents of the viewers
public EObject[] getViewerContents() {
  Rule rule = (Rule) getModel();
  return new EObject[] {
        rule.getNac(), rule.getLhs(), rule.getRhs() };
}

// factory for assigning GEF EditParts to model elements
protected EditPartFactory createEditPartFactory() {
  // reuse factory because lhs, rhs, and nac are object nets
  return new ObjectNetEditPartFactory();
}  } // end class
```

B Stepwise Animation Definition - an Example

To illustrate the principle of stepwise animation definition, we discuss the example time schedule matrix below. It represents the definition of an animation of three arbitrary model elements (*element1-3*). We have a column for each element to be animated and a row for each defined animation step, each with a positive integer value *d1-3* representing the duration of this step in msecs (except for the first, which is the starting position row):

	element1	***element2***	***element3***
start	*element1*	`Point(0,0)`	-/-
d1	*element2*	-/-	*element1*
d2	*element3*	-/-	*element2*
d3	*element1*	`Point(100,100)`	-/-

First, you have to understand what the performed animation will look like. After that you shall see how you can define this animation with an `AnimatingCommand`.

In short, each table entry tells when (after the sum of durations till this step) the figure of the column's element has to at a specific absolute or relative location. For example, *element1* is supposed to start at its own original position, to move in *d1* msecs to the position of *element2*, after that to move in *d2* msecs to the position of *element3*, and to return in *d3* seconds back to its initial position.

Note that when specifying relative locations, they will be resolved to the absolute locations that the related figure has before the animation. Thus, *element1*'s figure will move to the locations of *element2*'s and *element3*'s figures that they have before the animation, even if these will be animated themselves in this particular animation. Of course, only those elements will be animated, for which a figure (being related to this element by some `EditPart`) can be found. There is no need for *element1-3* having figures in the same viewer at command-execution nor command-definition time, but only figures in open viewers that exist at execution-time will be animated. `AnimatingCommand` will not regard it as an error if no figure can be found for an element at all.

Let's continue with *element2* to understand step interpolation: As you can see in the table, only an absolute starting position and a position after the third step has been specified. Generally, undefined steps between defined steps will be interpolated linearly. So, after *d1* msecs *element2*'s figure would reach the point `(33,33)`, after *d1*+*d2* msecs `(66,66)`, and finally `(100,100)`. Note that an animation of a particular figure always starts at the first specified step and ends at the last specified step. If no start or end location for the whole path of a model's figure can be resolved or interpolated, it will not be animated at all, but also no error message will be raised.

How *element3*'s figure would be animated should be obvious by now. Now we implement this animation. Suppose we have a container model element *parent*, being the contents of a viewer (e.g. a Petri net), and we want to animate the figures for *element1-3* (e.g. places or tokens):

```
AnimatingCommand aniComm = new AnimatingCommand();
// initialize objects for animation
aniComm.initializeAnimatedElement(element1, parent, null);
aniComm.initializeAnimatedElement(element2, parent, null);
// specify starting position
aniComm.specifyStep(element1, element1);
aniComm.specifyStep(element2, new Point(0, 0));
// finish "starting step", declare next step with duration d1
aniComm.nextStep(d1);
// specify step 1
aniComm.specifyStep(element1, element2);
aniComm.initializeAnimatedElement(element3, parent, null);
aniComm.specifyStep(element3, element1);
// finish definition of step 1, declare step with duration d2
aniComm.nextStep(d2);
// specify step 2
aniComm.specifyStep(element1, element3);
aniComm.specifyStep(element3, element2);
// finish definition of step 2, declare step with duration d3
aniComm.nextStep(d3);
```

```
// specify step 3
aniComm.specifyStep(element1, element1);
aniComm.specifyStep(element2, new Point(100, 100));
```

In this short code snippet, we first create a plain `AnimatingCommand` and declare *element1* and *element2* as to be animated in the viewer that 'shows *parent*', i.e. that has *parent* as contents. This must be done for an element before specifying any steps for it. The last parameter (`null`) can optionally alter the path curve form. Now, look at the matrix table above again: In the first step (which you have to imagine always as initial placement of the animated figures, i.e. happening with the duration 0msec) we specify *element1*'s and *element2*'s figures to be at the location of the *element1*'s figure of itself and at `Point(0,0)`, respectively.

With this we have completed the specification of the first step and can proceed to the next step by calling `nextStep(d1)`, which states that the next step should take *d1* msecs to complete. The next row in the table says that in this step *element1*'s figure should move to the original position of *element2*'s figure, and that *element3*'s figure should start its animation at the original position of element1's figure. For this we have to initialize *element3* belatedly. It should be clear by now, how the remaining lines correspond to the table entries.

Optional Animation Features Additionally, you may specify a size factor (relative to the figure's original size) for each step via `specifyStep(Object, Object, double)` that will affect the figure's size at this step's location. A value of -1 will mark this element's size factor as to be interpolated for the current step for a smooth animation, similar to the locations.

With `setDebug(boolean)` you can force the figures to mark their animation path with a red line, which will remain visible after animation.