Automatic Layout and Structure-Based Editing of UML Diagrams

Hauke Fuhrmann, Miro Spönemann, Michael Matzen, and Reinhard von Hanxleden
Real-Time and Embedded Systems Group, Department of Computer Science
Christian-Albrechts-Universität zu Kiel, Germany
{haf,msp,mim,rvh}@informatik.uni-kiel.de

***Abstract*—Graphical modeling languages, as defined in the UML, are appealing in their relative ease of comprehension. A well-structured model can provide a compact representation of complex designs. However, the development of graphical models is still hampered by modeling tools that force the user to perform low-level graphical editing steps, instead of focusing on the underlying model. We here propose novel, efficient modeling paradigms that build on an automatic layout capability, and present a prototypical implementation in the KIELER framework that supports a range of UML modeling tools.**

I. Introduction

Model-based engineering has gained an important role for the development of embedded systems. Graphical models are appealing and help to provide a common base for experts from different domains, especially if standardized like the UML. The graphical aspect aims at introducing intuitive language semantics and better displaying an abstraction of a system. A two-dimensional canvas gives more freedom to clearly present a view on a system than the one-dimensional textual representation, either in a low-level programming language or a high-level specification scheme. However, such freedoms demand new design decisions from the developer, who now has to spend effort into the graphical representation in order to reap its benefits. Petre quotes a professional developer as follows: "I quite often spend an hour or two just moving boxes and wires around, with no change in functionality, to make it that much more comprehensible when I come back to it" [1]. One estimate from industrial users puts the time spent with unproductive editing/formatting activities at about 30% of overall developing time[1].

In this paper we present a generic approach on interaction mechanisms that allow to create, edit, and analyze graphical models without the hassle of manual layouting of diagrams. In Sec. II we discuss the context of this work and related work. The first step is to completely automate the layouting task in the development environment, and to offer a layout of such quality that the developer is willing to dispense with manual placing. We implement an open layout interface to the *Eclipse* platform, link existing layout libraries and implement or adapt algorithms. This is presented in Sec. III. The next step is to consistently build upon the layout services and create new interaction paradigms. As one example of such, we present *structure-based editing* in Sec. IV in the context of UML editors in Eclipse. This editing scheme is based on structural transformations on the domain model, which are specified using a textual notation. In Sec. V we conclude and give an outlook on future work in this area.

II. Related Work

The KIEL project [2] evaluated the usage of automatic layout and structure-based editing in the context of *Statecharts*. It provided a platform for exploring layout alternatives and has been used for cognitive experiments evaluating established and novel modeling paradigms. However, it was rather limited in its scope and applicability, hence it has been succeeded by the KIELER project, the Kiel Integrated Environment for Layout Eclipse Rich Client[2], which is the context of the work presented here. KIELER aims at enhancing the *pragmatics* of graphical modeling, i. e. the way to interact with graphical models [3], and implements generic approaches applicable for a wide variety of graphical model types, including different UML diagrams. KIELER integrates into the Eclipse platform, which has a large user community in the modeling domain[3]. It makes use of the projects around the Eclipse Modeling Framework (EMF) and especially aims at providing its services to all graphical editors created with the Graphical Modeling Framework (GMF). The case studies applying the KIELER approaches to UML presented in this paper are carried out on the *Papyrus* UML tool suite and the UML2Tools which are both also official part of the Eclipse Modeling project.

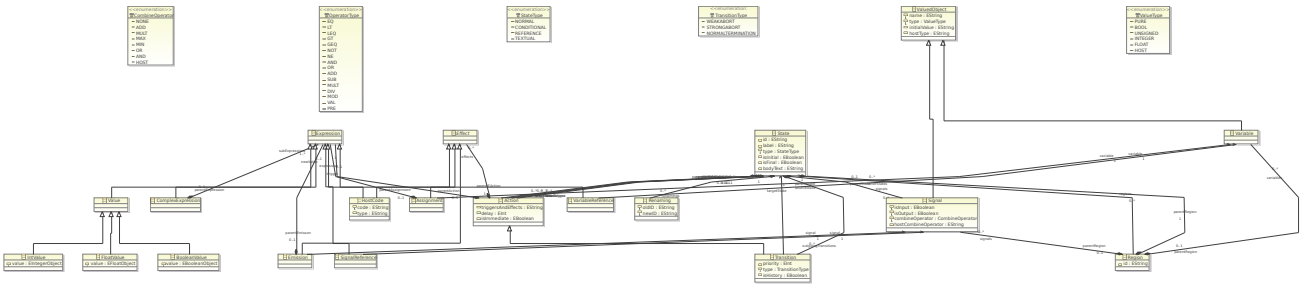
Model transformations are a well-established technique to achieve consistency between model entities produced at different model life-cycle stages. GenGEd [4] modifies visual languages using graph grammars and graph productions with predefined production sequences. We here instead propose responsive model manipulation by interactively triggering in-place model transformations like QVT [5] or, as presented here, *Xtend* from the Eclipse Model to Text (M2T) project.

The graph layout problem is studied by a large research community, which has developed a wide variety of different layout algorithms [6], [7], [8]. Such algorithms have been implemented in commercial tools such as yFiles (yWorks GmbH) and ILOG JViews [9] as well as non-commercial

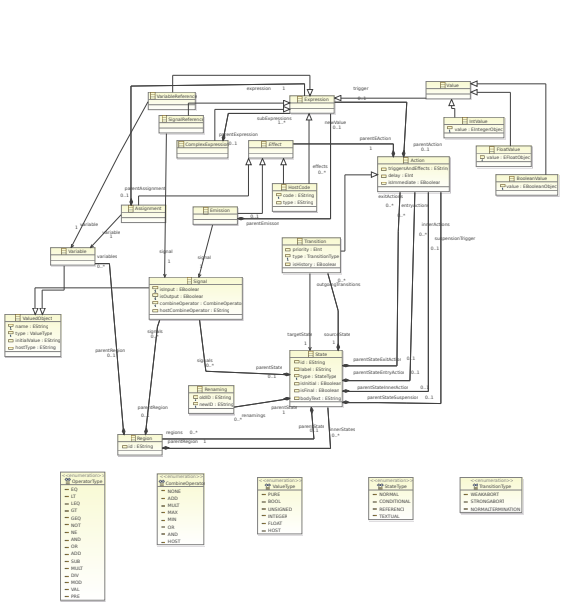
[1] L. K. Klauske (Daimler Center for Automotive IT Innovations), personal communication, Oct. 2009.

[2] http://www.informatik.uni-kiel.de/rtsys/kieler

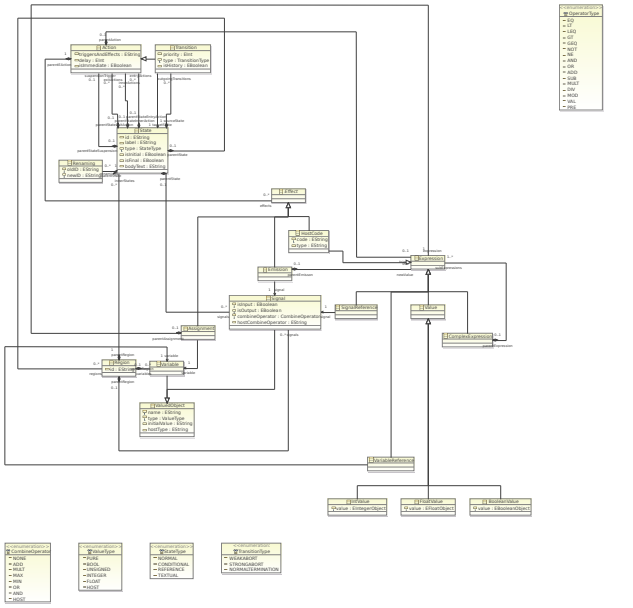
[3] http://www.eclipse.org/modeling/



(a) The default GMF layout



(b) Manually arranged layout, taking much time and not consistently following notation guidelines



(c) The layout provided by OGDF, with manually improved label positioning

Fig. 1. Layout comparison for an EMF class diagram.

tools such as Graphviz [10] and Zest, which is part of the Eclipse Graphical Editing Framework (GEF). A variant of the hierarchical layout algorithm [11] is also integrated in GMF (the *Arrange All* button that appears for GMF diagrams); however, that integration lacks flexibility and the resulting layouts are not useful in many cases, as seen in Fig. 1a.

The Open Graph Drawing Framework (OGDF) is a C++ class library that contains sophisticated graph algorithms for automatic layout [12]. Among these is a specialized algorithm for class diagrams [13], which should not be drawn with plain layout algorithms because of the different notation standards for the edge types *association*, *generalization*, and *dependency*. Other approaches for layout of class diagrams have been proposed by Eiglsperger [14], Seemann [15], and Eichelberger [16].

III. Automatic Layout

Our approach to automatic layout of graphical diagrams is best described as *meta layout*: instead of forcing the user to accept a fixed layout algorithm, we offer a flexible interface that allows to set different layout options for each diagram, or even for each part of a diagram. These options include the selection of a specific layout algorithm, which in turn can be contributed using Eclipse *extension points*.

A. General Procedure

Diagram editors in the Eclipse GMF are structured with the Model-View-Controller (MVC) paradigm: the *model* is built on an EMF structure that stores the semantic data, the *view* consists of *figures* that are used to draw the actual diagram, and the *controller* consists of a set of *edit parts* that are assigned to the model elements in order to control their visualization and editing. By exploiting the standardized structure of these edit parts, we are able to derive an annotated graph from any GMF diagram editor at run time. Since diagram elements may be hierarchically structured, the resulting graph G is a *compound graph* $G = (V, H, E)$ with a set of vertices V, a set of *inclusion edges* H, and a set of *adjacency edges* E [17]. The *inclusion graph* (V, H) must form a tree, where each inclusion edge $(v_1, v_2) \in H$ is interpreted as v_1 *contains* v_2. The annotations that are attached to the elements of G are either layout options to select and configure layout algorithms

or layout results such as the coordinates and size of each element.

We apply the following procedure to layout GMF diagrams:

1) Derive a compound graph G from the structure of edit parts of the diagram and add layout options as annotations.
2) Recursively perform the layout on the inclusion tree of G, starting with the leaves, considering the individual layout algorithm associated with each inner vertex of the inclusion tree. The resulting object positions are attached as annotations.
3) Apply the layout results that are attached to the elements of G to the respective edit parts of the diagram.

This procedure allows to extend any graph layout algorithm to compound graphs, furthermore we gain the flexibility of applying different algorithms on the layers of the inclusion tree. However, for compound graphs where adjacency edges may connect vertices from different hierarchy levels, e. g. UML state machines, this procedure does not perform well, as it cannot take into account these cross-hierarchy edges. A compound graph layout algorithm is required for these cases [17], [18].

B. Extension Points and Interfaces

Eclipse extension points consist of an XML schema document that defines an interface which can be extended by Eclipse plug-ins. Such an extension is expressed in the XML format, but can be edited with an intuitive user interface in Eclipse. With this extension point mechanism we open the automatic diagram layout to several contributions:

- Plug in new layout algorithms.
- Define new layout options and specify which options can be handled by each layout algorithm.
- Define diagram types and set default layout algorithms for each type.
- Configure default layout options for parts of specific diagram editors.

The contributed layout algorithms and layout options can be selected and changed at run time using an Eclipse view, as seen in Fig. 2.

For diagram editors that are directly generated by GMF the automatic layout can be used without any modification. The extension points mentioned above are optional and can be extended to customize the default behavior of layout algorithms for each editor. However, some editors contain changes of the GMF code that inhibit our layout interface from detecting them properly, e. g. the Papyrus GMF editors, which contain multiple diagrams in one editor and switch between them using tabs. For such cases another extension point can be used to add some bridge code that enables the automatic layout for the respective editors.

C. Algorithms and Limitations

Contributions of the Graphviz and Zest layout algorithms for our layout interface are already available, so that nice drawings can be created for a large number of diagram types.

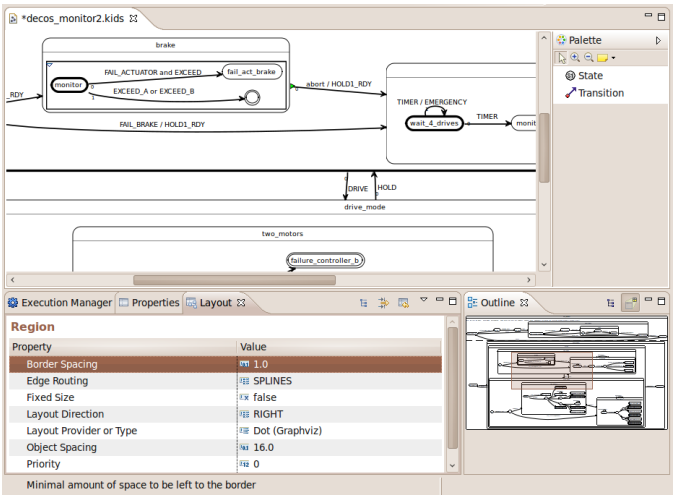


Fig. 2. The layout of a diagram can be customized in the Layout view, here a Statechart with compound graph layout.

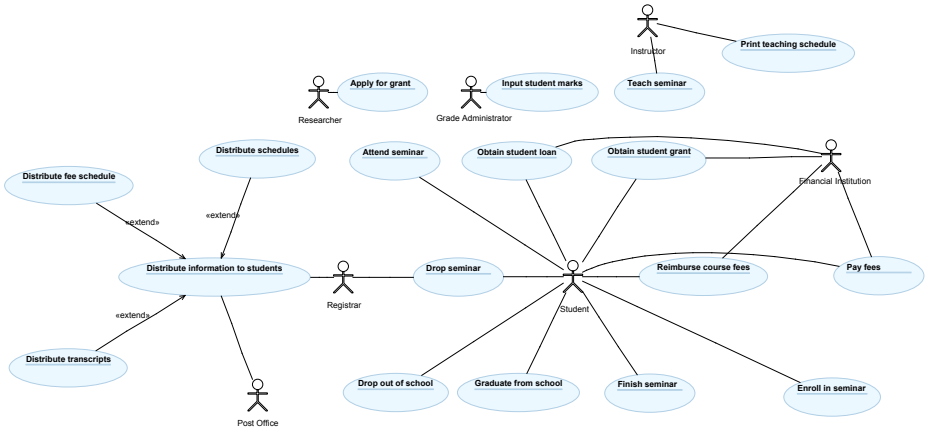
Figs. 3a and 3b show the result of automatic layout for a use case diagram and an activity diagram. The first layout follows an energy-based approach proposed by Kamada and Kawai [19], while the second layout follows a layer-based approach proposed by Sugiyama, Tagawa, and Toda [11], [20].

Despite the success of such graph drawing algorithms, some kinds of diagrams cannot be handled by their basic variants, but impose special constraints on their drawings. Data flow diagrams for embedded systems design, for example, require all edges to be connected to specific *ports* of their source and target nodes. The development of algorithms for the layout of such diagrams is a topic of ongoing research [21], and one implementation is included in KIELER.

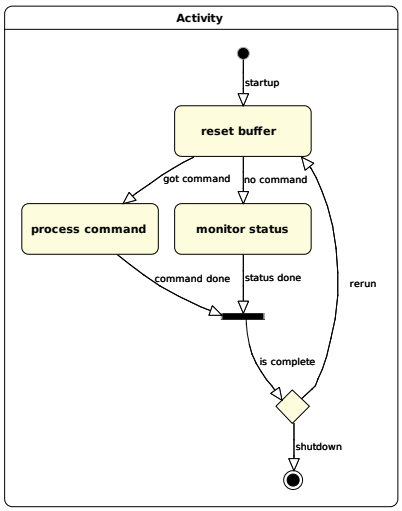
The UML also contains diagrams that need special treatment: class diagrams contain associations, generalizations, and dependencies, with different graphical notations. Fig. 1b shows a class diagram with manually arranged layout, which was very effort prone, took many iterations and hours to get to it, and still does not completely follow the notation guidelines. However, spending the efforts, the result is quite compact. Hence we do not claim that automatic layout always reveals better results, but it will always have a much better cost-benefit ratio than manual layout, and it enables new interaction methodologies as the one presented in the next section.

Since there already exist graph drawing libraries with specialized algorithms for class diagrams, it is one of our goals to create an interface to these tools. Fig. 1c shows a class diagram with an adapted *topology-shape-metrics* layout [22], [13], which is far more readable than the layout of the same diagram shown in Fig. 1a and compares well to Fig. 1b with the manual layout.

The meta layout approach presented in this section is implemented in KIELER, and all figures in this paper are real screen shots of diagrams processed with it.



(a) A use case diagram with Neato layout



(b) An activity diagram with Dot layout

Fig. 3. Papyrus diagram examples with different Graphviz layout methods.

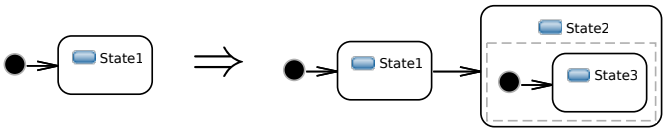


Fig. 4. State Machine: Insertion of a hierarchical composite state.

IV. Structure-Based Editing

In this section we first look at the standard state-of-the-practice editing process, and then present alternative approaches.

A. Drag-and-Drop Editing Process

In the Drag-and-Drop (DND) editing process, applying an *editing schema* generally involves the following action sequence [2]: (1) If needed, create free space (e. g. move existing elements or expand hierarchical ones). (2) Focus on a diagram element and interact with a modification tool (e. g. a toolbar, palette, or menu). (3) Apply the editing schema. (4) If needed, rearrange the modified diagram to improve readability. Steps 1 and 4 may consume much time depending on the current diagram context, in particular when modifying existing diagrams.

An editing schema itself consists of another set of low-level editing steps. For a UML State Machine diagram, the schema to "add a composite state with initial contents" uses the following steps to achieve the result shown in Fig. 4: (A) add a new composite state (in the Eclipse UML2Tools editor it already contains a region), (B) add a transition connector from original to new composite state, (C) add an inner initial state, (D) add an inner state, (E) add a connector from initial to new state.

Especially for introduction of composite elements, which do not make sense without any contents, the number of low-level steps is considerable.

B. The Transformational Approach

In order to avoid manual execution of all these steps, we propose an interactive transformational approach on the domain model, i. e., the semantic structure, in contrast to the graphical information, which in some editors is stored in a different model.

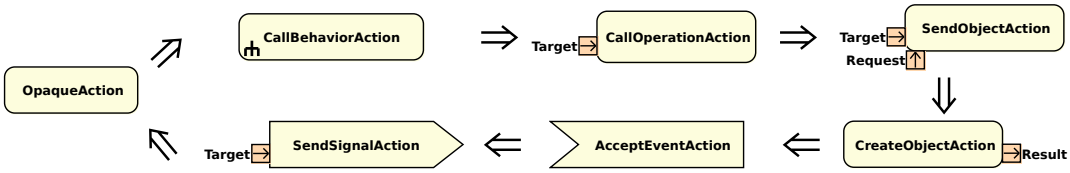
Listing 1 shows an in-place transformation in Xtend that specifies the editing scheme of Fig. 4 discussed above. It is a pure semantical model transformation and does not involve graphical information.

The structure-based editing approach proposed in this paper uses transformations as the above (or other transformation languages such as QVT) hand in hand with automatic layout of the corresponding graphical representation of the model as presented in Sec. III. The user interface of the resulting integrated development environment (IDE) provides a set of predefined transformations that are suitable to create and modify models through a toolbar, context menu, or keyboard shortcuts. Due to the application of automatic layout of the corresponding diagram, the manual editing action sequence from above is boiled down to: (1') Focus a graphical model element for modification or augmentation, (2') apply an editing transformation. The cleanup steps (1) and (4) from above vanish as well as the low-level sub-steps of the editing scheme itself (A)-(E).

C. Approaching the UML

As the Eclipse modeling project contains an open UML meta model, which is used by different graphical editor implementations such as the UML2Tools or Papyrus, transformations can simply be defined for this meta model to be re-used in multiple graphical editors. Structure-based editing has been explored for the KIELER *SyncCharts* editor (a synchronous Statecharts dialect); the editing schemes defined for SyncCharts can already be transferred to similar UML state ma-

(a) Insertion of a successor action



(b) Toggling transformation between different action classes

Fig. 5. Activity diagram transformations.

chine transformations. Other diagram types can be supported with corresponding transformations, e. g. activity diagrams, for which Fig. 5a shows a simple example. In principle, all UML diagram types are supported; in practice, however, this approach is feasible for those diagrams for which suitable automatic layout algorithms are provided, as discussed in Sec. III-C.

D. Object Class Transformations

The transformations so far are quite straight-forward and for Statecharts editing implemented in KIELER the number of transformations is clear and small.

For the UML meta model the situation is rather different. It extensively uses class hierarchy to express different object refinements. For example, in an Activity model there is the class *Action* which has the different subclasses *OpaqueAction*, *CallAction*, *InvocationAction*, *SendSignalAction*, *CallOperationAction* and *CallBehaviorAction*. All metamodel classes have a graphical representative with slight differences and a creation tool in the palette (which makes palettes of UML editors usually quite crowded). There are two major drawbacks of this approach: (1) If a developer inserts any of the above actions and then later realizes that a different class would be more appropriate, the object instance has to be exchanged completely. (2) To provide a sufficient set of transformations for structure-based editing, one would require similar transformations for each of the classes. E. g. the "insertion of a successor action" in Fig. 5a, which is shown for an *OpaqueAction*, would require a similar transformation rule for all other Action classes.

As an alternative, one could provide only a small set of transformations for one specialization of an abstract class, e. g. only for *OpaqueActions*. These rules suffice to create the basic graphical structure of the diagram employing automatic layout. Then another set of transformations would be responsible to change the concrete class of a model object. To reduce the number of transformations and user interface items, one single conditional transformation could *toggle* between all different subclasses of a given class. For the Action example, the sequence is shown in Fig. 5b.

Such *toggling transformations* free the developer of two manual tasks next to exchanging the class itself: First, copy all common attributes from the old class instance to the new one, and second, fix all incoming and outgoing connections. In this example we see that it is not always possible to the full extent when classes have different types of connections and ports.

For example, the creation of a new *CallOperationAction* would come down to the steps: (1) focus an existing action, (2) call the "insertion of a successor action" and (3) call the toggle transformation twice. It is up to the toolsmith to find a good trade-off between the number of transformations provided for a language and how directly specific editing schemes should be supported. However, especially the toggle transformations are a real benefit not only for pure structure-based editing, but also for any DND editing, where in general changing object classes requires a lot of manual steps.

Listing 1. Xtend transformation for insertion of a composite state

```
Void createComposite(State originalState):
  let compState = new State:
  let initState = new State:
  let simpleState = new State:
  let innerRegion = new Region:
  let oTrans = new Transition:
  let iTrans = new Transition:
  let parentRegion = originalState.container:
  parentRegion.transition.add(oTrans) ->
  oTrans.setSource(originalState) ->
  oTrans.setTarget(compState) ->
  compState.region.add(innerRegion) ->
  innerRegion.subvertex.add(initState) ->
  innerRegion.subvertex.add(simpleState) ->
  innerRegion.transition.add(iTrans) ->
  iTrans.setSource(initState) ->
  iTrans.setTarget(simpleState);
```

E. Implementation

The benefits of structure-based editing in general have been evaluated in the KIEL project for Statechart models. KIELER implements the approach for general GMF based editors in Eclipse. Once a set of model transformations has been defined for the metamodel of a specific graphical editor, they can be made available as operations in the user interface by using an Eclipse extension point. Thus the technical effort to connect this editing approach to new editors is kept minimal. For a new Domain Specific Language (DSL) the toolsmith provides

(1) a set of semantical model transformations (e.g. a file with Xtend transformations) and (2) a few plug-in configurations which are done via XML according to an Eclipse extension point. The latter configures names and places (menu, toolbar, popup) of transformations in the user interface.

While currently the specification of structure-based editing transformations is provided statically and loaded during Eclipse startup, we also experiment with user interfaces to dynamically create, save and share new transformations at tool run-time in order to give tool users themselves the ability to customize the user interface.

V. Conclusion and Outlook

We presented a flexible interface for automatic layout of graphical diagrams in Eclipse, and proposed a novel editing paradigm that builds on this layout capability. Our approaches build on standardized Eclipse technology, thus minimizing the effort of applying them to new graphical editors. The layout can be easily customized and be extended by other layout algorithms and layout options. By applying model transformation directly on the domain model and performing automatic layout on the transformed model, we are able to define structure-based operations, which can be triggered from the user interface. This promises a significant increase in designer productivity.

The UML with its graphical notation is a good application for the structure-based editing paradigm, and in this paper it has been shown which kind of model transformations can potentially help in editing UML diagrams. What is left as future work is the definition of a concrete set of transformations on the UML metamodel and the evaluation of the resulting structure-based editing operations in the context of actual development projects that employ the UML. The same approaches can be applied to other graphical languages for model-based engineering, e. g. Statecharts and data flow languages such as Simulink (The MathWorks), SCADE (Esterel Technologies), or Ptolemy II [23].

Structure-based editing is just one example of a new interaction scheme that consistently builds upon automatic layout of graphical models. Another approach, textual modeling with synchronization to a graphical diagram, is also a current topic of research. Since many developers are more skilled in editing with the keyboard only, this could further reduce the efforts of creating and maintaining models. Furthermore, there are other aspects next to editing that benefit of the automatic placement, e. g. navigation/browsing of models and analysis of models during simulations. Resulting improvements are dynamically created graphical views that can display models in different levels of details and present only model parts that are relevant to the user for the specific context under which the model is currently investigated. This topic is also subject of ongoing research under the term *view management* [3].

References

[1] M. Petre, "Why looking isn't always seeing: Readership skills and graphical programming," *Communications of the ACM*, vol. 38, no. 6, pp. 33–44, Jun. 1995.

[2] S. Prochnow and R. von Hanxleden, "Statechart development beyond WYSIWYG," in *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)*, Nashville, TN, USA, Oct. 2007.

[3] H. Fuhrmann and R. von Hanxleden, "On the pragmatics of model-based design," in *Proceedings of the 15th International Monterey Workshop on Foundations of Computer Software, Future Trends and Techniques for Development (2008)*, ser. LNCS (to appear), Budapest, 2010, also available as Technical Report 0913, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2009.

[4] R. Bardohl, "GenGEd – a visual environment for visual languages," *Science of Computer Programming, Special Issue of GraTra '00*, 2002.

[5] Object Management Group, "MOF 2.0 Query/Views/Transformation RFP," Apr. 2004, http://www.omg.org/docs/ad/02-04-10.pdf.

[6] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis, *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[7] M. Kaufmann and D. Wagner, Eds., *Drawing Graphs: Methods and Models*, ser. LNCS. Berlin, Germany: Springer-Verlag, 2001, no. 2025.

[8] M. Jünger and P. Mutzel, *Graph Drawing Software*. Springer, Oct. 2003.

[9] G. Sander and A. Vasiliu, "The ILOG JViews graph layout module," in *GD 2001: Proceedings of the 9th International Symposium on Graph Drawing*, ser. LNCS, vol. 2265. Springer-Verlag, 2002, pp. 469–475.

[10] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *Software—Practice and Experience*, vol. 30, no. 11, pp. 1203–1234, 2000.

[11] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for visual understanding of hierarchical system structures," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 11, no. 2, pp. 109–125, Feb. 1981.

[12] M. Chimani, C. Gutwenger, M. Jünger, K. Klein, P. Mutzel, and M. Schulz, "The Open Graph Drawing Framework," Poster at the 15th International Symposium on Graph Drawing (GD07), Sydney, 2007.

[13] C. Gutwenger, M. Jünger, K. Klein, J. Kupke, S. Leipert, and P. Mutzel, "A new approach for visualizing UML class diagrams," in *SoftVis '03: Proceedings of the 2003 ACM Symposium on Software Visualization*. New York, NY, USA: ACM, 2003, pp. 179–188.

[14] M. Eiglsperger, "Automatic layout of UML class diagrams: A topology-shape-metrics approach," Ph.D. dissertation, Faculty of Information and Cognitive Science, Eberhard-Karls-Universität Tübingen, 2003.

[15] J. Seemann, "Extending the Sugiyama algorithm for drawing UML class diagrams: Towards automatic layout of object-oriented software diagrams," in *Proceedings of the 5th International Symposium on Graph Drawing (GD '97)*, ser. LNCS, vol. 1353. Springer, 1997, pp. 415–424.

[16] H. Eichelberger, "Aesthetics and automatic layout of UML class diagrams," Ph.D. dissertation, Bayerische Julius-Maximilians-Universität Würzburg, 2005.

[17] K. Sugiyama and K. Misue, "Visualization of structural information: automatic drawing of compound digraphs," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 21, no. 4, pp. 876–892, Jul/Aug 1991.

[18] G. Sander, "Layout of compound directed graphs," Universität des Saarlandes, FB 14 Informatik, 66041 Saarbrücken, Tech. Rep. A/03/96, Jun. 1996.

[19] T. Kamada and S. Kawai, "An algorithm for drawing general undirected graphs," *Information Processing Letters*, vol. 31, no. 1, pp. 7–15, 1989.

[20] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo, "A technique for drawing directed graphs," *Software Engineering*, vol. 19, no. 3, pp. 214–230, 1993.

[21] M. Spönemann, H. Fuhrmann, R. von Hanxleden, and P. Mutzel, "Port constraints in hierarchical layout of data flow diagrams," in *Proceedings of the 17th International Symposium on Graph Drawing (GD'09)*, ser. LNCS, vol. 5849. Springer, 2010, pp. 135–146.

[22] R. Tamassia, G. D. Battista, and C. Batini, "Automatic graph drawing and readability of diagrams," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 18, no. 1, pp. 61–79, 1988.

[23] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity—the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, Jan 2003.