Structure, Abstraction and Direct Manipulation in Diagram Editors

Oliver Köth and Mark Minas

Lehrstuhl für Programmiersprachen
Universität Erlangen-Nürnberg
Martensstr. 3, 91058 Erlangen, Germany
minas@informatik.uni-erlangen.de

Abstract. Editors for visual languages should be as simple and convenient to use as possible; at the same time, programmers should be able to create such editors without prohibitive effort. We discuss the benefits that can be gained from combining the following aspects in an editor-generator approach:

- direct-manipulation editing (as in drawing programs)
- structure-based editing (as in common diagram tools)
- structural analysis and a common formal model

As a major practical example, we present an editor for UML class diagrams. We show that direct-manipulation editing capabilities can enhance the usability of such an editor in comparison to standard tools. A further improvement is obtained by including selective abstraction features similar to the well-known "fisheye-viewing" and "semantic zooming" paradigms. We show that the proposed generator architecture provides an excellent base for implementing such features. The resulting technique can be applied to a wide range of different diagram languages; in contrast to other general solutions, it takes into account the abstract structure and specific abstraction features of the individual languages.

1 Introduction and Overview

Working with visual languages requires appropriate tools, particularly editors, that are specially tailored to the respective language. It is possible to use a general-purpose drawing program to create, for example, UML diagrams. But this support immediately turns out to be insufficient when it comes to *working* with a language, i. e. modifying and extending those diagrams iteratively and extracting their "meaning" for use with other tools, e. g. a code generator. Consequently, for visual languages, language-specific editors are a must.

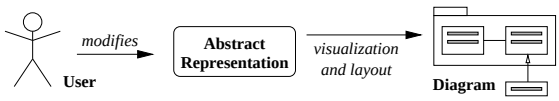
In this paper we present DiaGen, an editor-generator toolkit, which allows to create editors for specific diagram languages[1] with comparatively little effort. In the next section we contrast two different editing modes and corresponding

[1] In this paper, we use the terms "visual language" and "diagram language" as synonyms.

architectures, syntax-directed editing and direct manipulation editing, and show how they have been combined in a general concept (Section 3). Section 4 gives an example how this approach can be applied to a complex VL by the example of UML class diagrams, and why direct manipulation can be an improvement over conventional tool behavior. Section 5 explains why support for diagram abstraction is important for working with large diagrams and Section 6 shows how language-specific abstraction views can be supported in a diagram editor through the use of diagram transformations and diagram parsing information. We also consider how these features affect the automatic layout correction. Section 7 discusses the advantages and disadvantages of the presented abstraction technique, and Section 8 shows how it relates to other work. Finally, Section 9 repeats the main results of this work and indicates directions for future research.

2 Structure-based Editing and Direct Manipulation

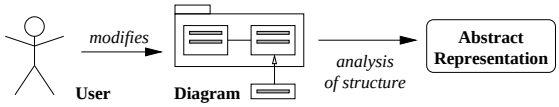
Programming specialized VL editors requires a lot of effort. A conventional architecture for such editors is built around an abstract internal representation of the diagram; in the case of UML diagrams that would be an object model based on the UML metamodel [18]. Most diagram modifications are accomplished through structure-based editing operations that modify this internal model, e. g. to add an attribute to a UML class. The actual diagram is a visualization of this internal model according to the rules of the diagram language as they are stated, for example, in the UML Notation Guide.



The whole editor architecture is highly dependent on the internal model and, since a VL typically has its own internal model, an editor for a new language must be implemented from scratch. That means that the effort to implement new experimental languages is quite high.

Editor toolkits or generators like DiaGen [14] aim to improve this situation and support the programmer in creating a specialized editor for a given diagram language. The programmer only needs to specify the graphical symbols of the language and the rules that describe how these symbols can be arranged to form correct diagrams. The toolkit then generates a complete visual editor that can typically output some sort of "abstract description" to be used by other tools.

As the specification of a new VL should be as easy and concise as possible, the generator should not force the programmer to define a complete set of language-specific editing operations that allows convenient creation and manipulation of all language features. Instead, generated editors often allow the user to directly manipulate the visual symbols of the language and the abstract internal model is generated by analyzing the visual representation of the diagram.



An important characteristic of this editing style is that it is possible for the user to create "diagrams" that violate the rules of the language. While this may seem to be a disadvantage, we have found that it often greatly simplifies the editing process. For example, it is possible to duplicate parts of a diagram using "copy & paste" features and fix up the arising errors (unconnected links etc.) later.

In contrast to a simple drawing program, such a diagram editor knows about the rules of the diagram language and the abstract "meaning" of the diagram. It can thus indicate, which parts of the diagram conform to the rules, and it can execute automatic layout corrections and beautifications; as a very simple example, when a class box in a UML diagram is moved, all attached links should stay connected to it.

3 DiaGen

The DiaGen toolkit combines the principles of direct manipulation and syntax-directed editing and allows to modify the visual and abstract representations of a diagram at the same time: A concise specification language is used to define the syntax of the VL; the specification document is then processed by a program generator to create a simple direct manipulation editor customized to that language. (Direct manipulation takes place by dragging "handles" that determine the shape and position of symbols.) If necessary, the specification can be extended with syntax-directed editing operations that manipulate the internal diagram representation [12, 16]; those "diagram transformations" combine elementary modifications of the diagram in the way of a powerful macro capability. The toolkit as well as the generated editors are all expressed in the Java programming language.

To achieve this integration of direct manipulation and syntax-directed editing, DiaGen uses attributed graphs[2] as a common base for the internal representation of different VLs. This allows us to use graph grammars as a common means for the analysis of diagrams; graph transformations can be used to model syntax-directed editing operations in a language-independent manner. It should be noted that this internal graph representation does by no means restrict the application of the DiaGen toolkit to graph-like VLs; the representational form is suited to almost any kind of diagrammatic language. It has been applied to UML class diagrams, petri nets, finite automata, control flow diagrams, Nassi-Shneiderman diagrams, message sequence charts, visual expression diagrams, sequential function charts, ladder diagrams etc.

[2] Actually, DiaGen uses hypergraphs whose edges are allowed to visit more than just two nodes.

Figure 1 shows the architecture of a diagram editor based on DiaGen (for details see [15, 16]): Direct manipulation is supported by the *drawing tool*. The resulting diagram is transformed into a (hyper-)graph model by the *modeler* after each diagram manipulation. *Reducer* and *parser* analyze this model; this is similar to traditional compiler techniques for textual languages: Reducer and parser correspond to lexical and syntactic analysis, resp., however based on graph grammars instead of string grammars. Syntax-directed diagram modifications are carried out by the *hypergraph transformer* which modifies the hypergraph model by graph transformations. Please note that results of any "diagram transformation" are analyzed by reducer and parser, i. e. syntax-directed editing operations cannot change the syntax of the VL [15, 16].

Figure 1 also shows how semantic analysis *attribute evaluation* and *layout correction* are integrated into the generated editor: Structural information which is gathered during diagram analysis is used by the *layouter* which adapts the diagram layout after each diagram modification (cf. Section 6).

Generated editors can be easily extended to support more convenient editing operations and new language constructs. The tight integration with the Java language allows to freely extend the user interface as well as other aspects of the editors and to use them as modules (Java-Beans) in larger programs. The toolkit as well as some documentation and publications is available at `http://www2.cs.fau.de/DiaGen`.

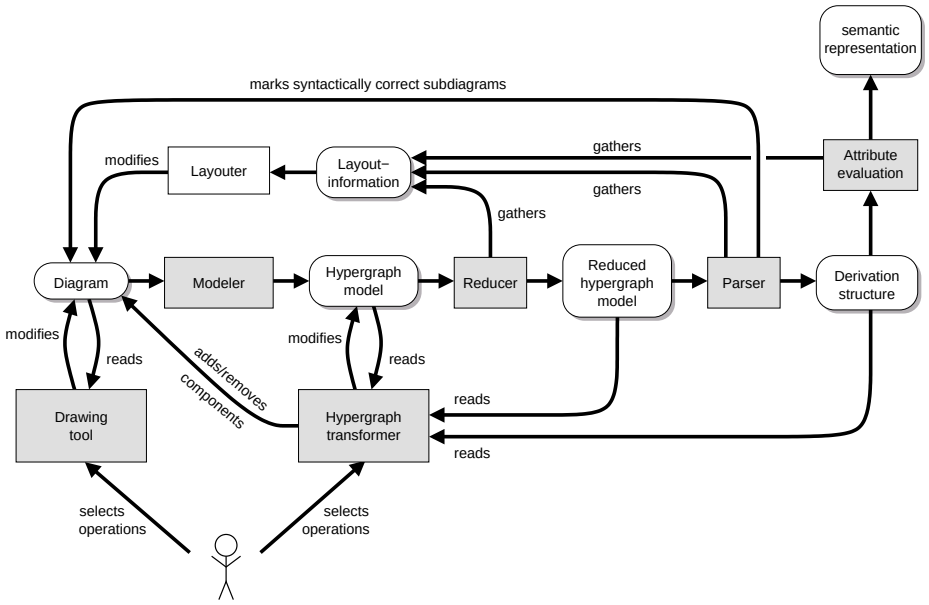


Fig. 1. Architecture of a diagram editor based on DiaGen.

4 Diagram Editing Using Direct Manipulation

Former publications on DiaGen have centered on the theoretical concepts and underlying mechanism for diagram analysis and transformation. In this paper we want to focus on the application of the toolkit and show the advantages that the approach presents in contrast to other methods.

As a continuous example we will use (a subset of) UML class diagrams; we assume that the reader is familiar with the most frequently used parts of this notation (classes, the different link types, attributes and packages). We have implemented a DiaGen editor for this type of diagrams and we believe that it strongly supports our claim that the techniques outlined above can indeed be used for complex diagram languages with practical relevance. With relatively little effort, it was possible to include support for a lot of notation variants that even commercial tools do not generally support (graphically nested packages, association classes, associations between more than two classes etc.). The internal graph representation of this editor is modeled after the standard UML metamodel and thus provides an adequate base for interfacing the editor with other CASE tools. To demonstrate this, we have implemented an XMI export feature, so we are able to exchange the created diagrams with other XMI-conforming tools. e. g. Together[3].

When we compare such a direct manipulation editor to standard (commercial) editing tools for UML diagrams, we find in fact that it often provides more convenient and direct ways to achieve a desired diagram modification.

As a simple example, consider the task of moving an attribute from one class to another. If a syntax-directed editor does not provide a special operation "move attribute", the only possible solution is to delete the attribute and recreate it in the destination class. In a direct manipulation editor, the user can simply drag the graphic symbol for the attribute from one class to the other. The diagram analysis recognizes the modified structure and updates the abstract representation; the layout correction takes care of resizing the class frames and aligning the attribute properly at its new place. Other examples one could think of could be "redirecting" an association that has some associated information (name and multiplicities) from one class to another or the duplication of arbitrary diagram parts using "copy & paste", when the resulting diagram is not structurally correct, but needs to be completed by additional editing operations.

From a more general point of view, a syntax-directed editor always forces the user to think in terms of "deep" modifications of the abstract model represented by the diagram. Direct manipulation features, in contrast, permit the user to additionally consider "shallow" modifications of the visual representation, which may sometimes be more direct and convenient and fit better into the immediate visual perception.

Figure 2 shows a screenshot of the UML editor. Several features relating to direct manipulation are visible here:

[3] `http://www.togethersoft.com/`

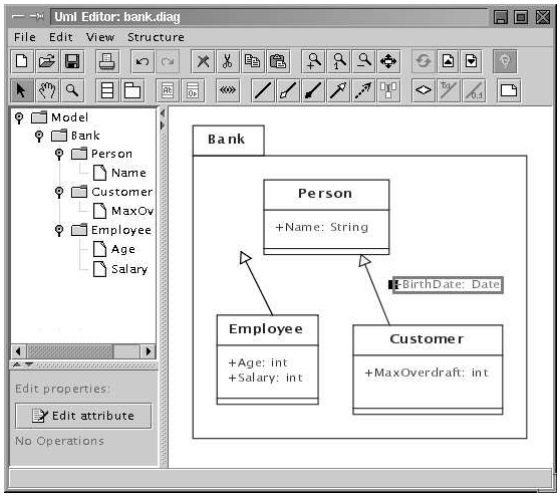


Fig. 2. Screenshot of a UML editor supporting direct manipulation

- The toolbar at the top of the window allows to create all kinds of elements that may occur in the supported UML class diagrams; there is no need to search through any menus, e. g. to add a stereotype to a class or a link.
- The selected diagram element (the "BirthDate" attribute) has been dragged out of the "Customer" class and is going to be moved to the "Person" class. Currently it is, of course, not placed correctly and therefore not recognized by the diagram analysis.
- The generalization arrow originating from the "Employee" class does not end at a class and is therefore incorrect. This is indicated by giving it a different color, which cannot be seen in the figure due to the grayscale display. To correct this error, the user would have to select the arrow and drag the endpoint into the "Person" class.
- Whenever parts of the diagram are manipulated, the automatic layout correction tries to adapt the whole layout. For example, the generalization arrow between the "Person" and "Customer" classes ends exactly at the boundaries of those elements, and the "Bank" package frame is shaped as the enclosing rectangle of its parts plus some border width.

The lower left window pane allows to execute syntax-directed editing operations; for the selected element (an attribute), there are no such operations available. Like in many other CASE tools, the tree display on the upper left window pane shows a different view on (parts of) the abstract internal diagram representation; in our case, it is generated from the results of the diagram analysis

Of course, similar features can also be included in standard (syntax directed) editors; for example, it is possible to implement drag & drop support for attributes and trigger operations to modify the respective classes accordingly. The

point is, that all these manipulations would have to be programmed explicitly, whereas the combination of direct manipulation, diagram analysis and automatic layout correction provides them almost "for free"; the programmer only needs to specify the structure of the language (and the layout), but does not need to care about every possible way to modify it. In fact, direct manipulation may permit manipulations of the diagram that the programmer has not even thought of.

For more complex modifications that are frequently needed, our architecture still allows to define powerful syntax-directed editing operations. As an additional feature, because all manipulations are ultimately broken down into a small set of elementary steps (insertion/deletion of graph parts, modification of attributes), it is comparatively easy to implement a multi-level undo/redo capability that works across all editing operations.

5 Working With Abstraction

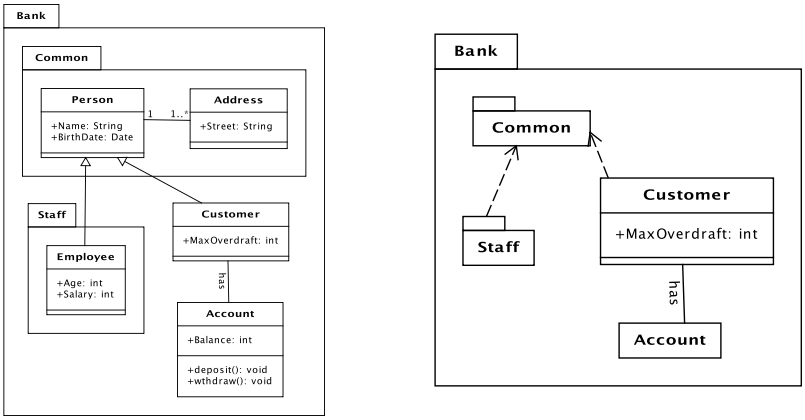


Fig. 3. A sample UML diagram and a "Focus and Context" view

We have found that the combination of graph parsing and graph transformation that DiaGen uses to analyze and modify diagrams is particularly well suited to express the concept of *abstraction* in diagram editing. Editor support for abstraction is necessary, because, with most VLs, when you scale them from toy applications to practical problems, the visualizations often get uncomfortably large and it is not possible to display all the information readably on a single computer screen. To permit the user to keep an overview of the whole system, it is necessary to suppress detailed information and display it in a simplified form (a principle that has been termed "semantic zooming" [2]).

To permit convenient working with large diagrams, an additional key feature is *selective abstraction*: Normally the user is working on a restricted region of the diagram (the "focus" area) and needs to see that in detail. The surrounding

regions (the "context"), however, should still be visible so that it is clear how the focus area fits into the context and what other diagram parts may be affected. Therefore it is desirable to "shrink" the context visually (by using abstraction) so that a larger context area can be displayed. This can be achieved by selectively abstracting the context while keeping the focus area unchanged.

Figure 3 demonstrates how this idea can applied to UML class diagrams. The left side shows a (very simplistic) model of a banking system that contains several nested packages. The diagram on the right side has been modified to facilitate working with the "Customer" element (the focus): all the details of this element remain visible, while the surrounding diagram parts only show their name and their interconnections and hide all details that are assumed to be irrelevant for the current task. The abstraction reduces the total size of the diagram, therefore it can be displayed at a larger scale (using standard optical zoom) so that the interesting details are better visible. Evidently, this technique yields most benefits, when it is applied to large models, where even a screen full of information can only show part of the whole system at an acceptable detail (text) size.

This *focus and context* viewing principle has been introduced by Furnas in 1981 [9] and since been used in a variety of applications; the best known implementations are probably optical nonlinear magnification systems, also known as "fisheye lenses" [10, 23]. In contrast to such systems, DiaGen offers the possibility to use structure-based abstraction instead of optical distortion, because the generated editors know about the abstract structure of the diagram and the graph-based internal models offer a standardized way of dealing with it.

Most visual languages already provide explicit means for abstraction. Two examples from the UML that we have integrated into our editor example are:

- the abstraction of classes by "hiding" their attributes and operations (cf. the "Account" class in Figure 3), and
- the abstraction of packages by "hiding" all contained elements (cf. the "Common" and "Staff" packages in Figure 3).

As can be seen in the example, the abstract forms for both classes and packages have visualizations that differ slightly from the standard forms, to distinguish them from elements that are merely empty and do not contain any hidden information. This implies that, like many other parts of the editor, the handling of abstraction must be tailored to the specific VL.

In fact there may be even more language-specific adaption required; in the case of UML package abstraction we need to redirect links that connect hidden package contents to the "outside" so that they end at the package. (In Figure 3, this is the case for the two generalization arrows ending at the "Person" class). However, according to the UML semantics, this way of redirecting generalization or association arrows changes the meaning of the diagram and may not even lead to a correct diagram. We can deal with this problem by turning those arrows into dependency arrows, which simply imply that there is some connection between the outside element and the package (which, in our case, cannot be

shown in detail unless the package contents are expanded again). Here we have an example for an adaption that obviously needs to be coded explicitly with specific knowledge about the VL semantics.

6 Abstraction by Means of Diagram Transformations

Now the architecture of the DiaGen system, which we have sketched in Section 2, allows to treat this kind of abstraction as just another form of diagram transformation. That means that the specification of a diagram language is augmented with syntax-directed editing operations that

- remove details from the diagram (and store them, so they can be re-inserted later),
- mark abstract elements visually or (optionally) replace them with different abstraction-representations,
- and perform other necessary adaptations like the arrow modifications mentioned above.

Of course, "reverse" transformations are also required to refine the diagram and re-insert the abstracted details. Note that this concept does not require specific geometric relationships between the "detail" and "higher-level" elements; in most cases, the details are visually contained in the higher-level elements, but, to give a different example, we have also used the same technique to implement an editor for rooted trees that allows to selectively hide all subtrees rooted at designated nodes. In this case, the "details" (lower-level nodes) are indicated not by containment but by the interconnection of the nodes; still the results of the diagram analysis allow to find all subordinate elements for a certain node easily.

Like the UML example shows, such abstraction transformations are usually based on the structural hierarchy of the diagram (although the concept also allows for other applications like the abstraction of selected parts of "flat" petri nets). This hierarchical structure is also expressed in the grammar for a diagram language. In our example, all the contents of a UML package are derived from a "Package" nonterminal symbol at some level of the grammar. DiaGen uses a sort of extended context-free graph parser for the analysis of diagrams, which creates a derivation structure (cf. Figure 1) in the form of a directed acyclic graph (DAG). The graph transformations that describe abstractions can make use of this DAG structure to select the diagram parts that are to be transformed. (If necessary, they may additionally include selection patterns that refer to the geometric relations between diagram parts, like "arrow a connects to class c" or "package p_1 is contained in package p_2".)

Figure 4 gives an idea of how the derivation structure can be used for selecting the details of a certain entity in the diagram, i. e. all the diagram elements that should be hidden by an abstraction transformation. It shows part of the (simplified) derivation structure that is created by analyzing the UML diagram on the left side of Figure 3. The lowest level contains the visual elements of the

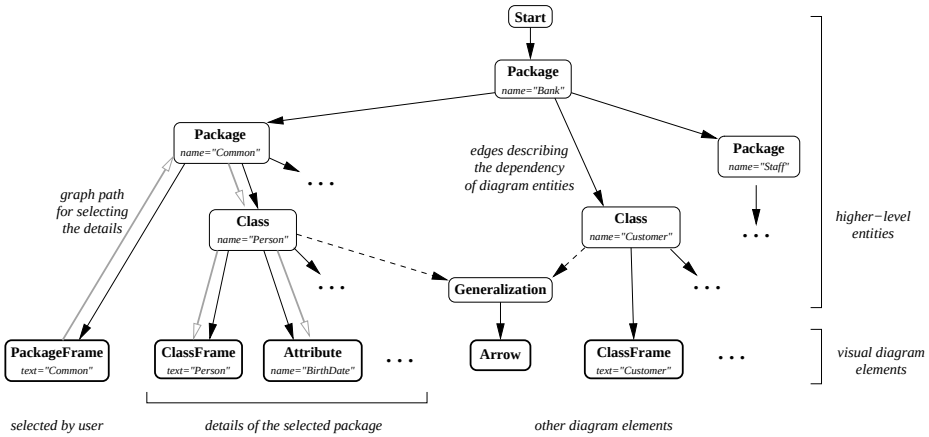


Fig. 4. Retrieving hierarchy information from the derivation structure of a diagram

drawing, which serve as terminal symbols for the grammar.[4] The other elements of the graph structure are higher-level entities (nonterminal symbols) found by the analysis module; the two types of black arrows represent how they depend on each other. Selected attributes of the symbols are also shown to make clear how the derivation structure relates to the actual diagram.

We have found that, in most cases, abstraction transformations can be defined by a simple standard pattern: The selection starts from a diagram element selected by the user (e. g. a package frame), then it walks "up" in the derivation structure to the first nonterminal symbol of a certain type (e. g. "Package") representing the desired higher-level entity; finally it follows the edges "down" again to include everything derived from that nonterminal. The gray arrows in the figure illustrate this selection process; it can be described by *path expressions* on the derivation DAG quite similar to *XPath* expressions [6] for XML documents. Matching entities are then handled by graph transformation rules, i. e. by the hypergraph transformer (cf. Figure 1). Thus the graph-grammar approach is very well suited to describe diagram abstraction.

The basic abstraction transformations for individual diagram units (like a single class or package) can be combined by higher-level transformations to create different views of the whole diagram. For example, the principles described by Furnas [9] can be used to compute a "degree of interest" for every diagram element, which can then be used to determine how far it should be abstracted, resulting in a view with a selected focus element shown in detail and a simplified context.

[4] In fact, these terminal symbols are found by the *reducer* (cf. Figure 1) which performs a preprocessing step that also takes into account the geometric relations (containment etc.) between the elements; a more detailed description of the analysis process can be found in [14, 16].

This desired focus and context effect stems from the fact that abstraction visualizations take up less screen space than the diagram part they represent. Obviously, if they just remained in their previous positions, the result would be a very "sparse" diagram of almost the same size with lots of free space between the small abstractions. To achieve the desired result, the layout correction mechanism must be able to handle these size changes properly.

Layout corrections for a DiaGen editor (cf. Figure 1) must, of course, be specified individually, along with the diagram language. DiaGen supports a constraint-based layout mechanism[5] as well as directly programmed layout algorithms. Since most diagram languages do not permit overlapping elements, the case of "growing" diagram parts (after refinement) can typically be handled by the standard layout procedure that pushes overlapping elements apart.

The case of "shrinking" diagram parts (after abstraction), on the other hand, must be addressed explicitly, because a "spread-out" layout does generally not violate the rules of the diagram language and should therefore not always be corrected automatically. Therefore the abstraction operation must explicitly request a "layout compaction" to use the free space. In the case of a constraint-based layout this might be done by inserting temporary "weak" constraints, which strive to reduce the distance between certain diagram elements as long as this does not violate other "hard" correctness constraints.

In the UML example, we have used a programmed layout based on *force-directed* layout algorithms (cf. [4]) to handle the class and package placement. The definition of adequate attracting or repelling forces between layout elements (which take into account their size and shape) allows to achieve the desired expansion or contraction of the diagram while preserving the relative element positions. We have found that the force-directed layout principle is a very flexible and powerful tool for expressing such interactive layout corrections; even if multiple elements change their size and shape simultaneously, we obtain a visually pleasing result.

In contrast to most applications of force-directed layout algorithms, we are not dealing with the *creation* of a complete layout from an abstract graph representation only, but instead, we are merely using them to *adapt* an existing layout to relatively minor changes. As Misue et al. [17] have already stated, this situation is ideally suited to heuristic optimization techniques like those employed by force-directed layout: We do not have to look for a globally optimal layout from a randomly chosen initial setup, but instead only need to do a few iterations to find the nearest local optimum from a relatively good start position. Complex optimizations like the minimization of link crossings can be left to the user; automatic corrections of this sort would appear to be confusing.

[5] based on the Qoca constraint solver [13] from Monash University, `http://www.csse.monash.edu.au/projects/qoca/`

7 Discussion

When we compare our approach of treating abstraction as a special case of diagram transformation with other focus and context techniques, we find that it provides some important advantages:

- The proposed technique can be applied across a wide range of diagram types, but can always be customized to the requirements of the specific diagram language. The resulting displays conform to the language rules and are easy to understand.
- Abstraction operations are well integrated with the normal editing behavior, which means that, instead of adding an extra layer of visualization control, the display can be manually adjusted and fine-tuned using standard editing functions.
- Features of "normal" diagram transformations directly extend to abstraction operations. In particular, transformations can be undone and redone to an arbitrary depth and the transitions can be animated smoothly; both capabilities improve the usability of the editor (cf. [23]).

Of course, there are also some drawbacks:

- Abstraction operations must be implemented separately for every diagram language. Fortunately, we found that they usually employ a simple common scheme (cf. Section 6), thus their specification is usually short and can be partly automated for general cases.
- Layout support for the changing size of diagram parts is expensive to program. We hope to remedy this by providing a library of customizable layout algorithms for common diagram classes (e. g. graph-like diagrams).
- Abstraction operations modify the internal representation of a diagram instead of merely providing a different view on them. This implies that it is up to the programmer to ensure that the "meaning" of the diagram remains consistent; it also does not allow to present simultaneous views on the same diagram with different abstraction levels in separate windows.

While the proposed technique is certainly not a panacea for dealing with complex VL descriptions, we think that it may serve as a valuable tool for easily creating VL editors that can be conveniently used for large, "real-world" problems. Some preliminary experiments have been conducted which support this hypothesis: Some subsystems of the DiaGen editor framework have been re-engineered to UML class diagrams. Students have been asked to compare a commercial UML tool with the UML class diagram editor with abstraction support. They confirmed that those abstractions offered valuable help for faster software understanding.

8 Related Work

There are several other approaches for generating visual language editors from a formal specification. However, we know about no approach whose editors support

abstraction with varying level of detail as it has been described in Section 5. All of the existing approaches fall into one of two categories: the ones *with* a graph-like model and the ones *without* such a model. DiaGen is a member of the first category. Other approaches and tools which are based on a graph-like model are GenGEd [1], Kogge [8], VisPro [24], and the approach by Rekers and Schürr [20]. With the exception of DiaGen and Reker's and Schürr's approach, all of them support syntax-directed editing only. Editing operations primarily modify the graph-like model, and the visual representation of the model is adapted accordingly. Direct manipulation – as described in Section 2 – which requires graph parsing is only supported by DiaGen and Reker's and Schürr's approach.[6] Unfortunately, their approach has never been implemented whereas DiaGen has already been applied to many different diagram languages.

Vlcc [7] and Penguins [5] are examples of the approaches of the approaches which do not depend on an internal graph-like model, but parse the diagram directly. Whereas Vlcc makes use of *Positional Grammars*, Penguins uses *Constraint Multiset Grammars* for specifying diagram syntax. These approaches, therefore, also support direct manipulation editing, but – unlike DiaGen and Reker's and Schürr's approach – they do not support syntax-directed editing which requires the use of an abstract internal model.

Our approach to diagram abstraction builds on a lot of work about information visualization. It fills a gap between optical fisheye techniques as described, for example in [10, 23], and solutions for specific diagram types, like hierarchical graphs [21, 17]. The latter do not provide a general basis for other diagram types, while the former can deal with arbitrary information but cannot be customized to follow the rules of a specific visual language and result in unnaturally distorted views. In [22] an empirical study is presented with the result that focus and context techniques indeed lead to improved usability compared simple zooming. "Semantic zooming" toolkits [2, 3, 19] provide a general base for applications using abstraction views, but they require that the shape and size of objects remain constant at different detail levels and they cannot support selective abstraction to create focus and context views.

The layout techniques that we use for the UML editor are very similar to those described by Misue et al. [17], who especially advocate the use of force-directed layout algorithms for layout adjustment as opposed to layout creation. Brandenburg et al. [4] give a broad overview over different variants of force-directed graph layout algorithms. A complete discussion of the background and implementation of our abstraction technique, the UML example and the layout algorithms used for it can be found in [11].

9 Conclusion

We have shown that the combination of direct manipulation and parsing techniques can be used to create editors for complex diagram languages and may

[6] VisPro does not support direct manipulation editing although it depends on graph parsing and could, therefore, support direct manipulation.

provide more intuitive manipulation capabilities than structure-based editing alone. The use of graphs as a common framework for specifying abstract diagram representations makes it possible to augment these editors with powerful syntax-oriented editing operations.

Such editing operations can also be used as a means to support language-specific abstraction of diagram parts. They can be easily defined based on information about the hierarchical structure of a diagram that is inherently contained the grammar describing the VL and the parsing structure for an actual diagram. By combining individual abstraction operations, it is possible to generate focus and context views of complex diagrams. These present the user's area of interest in sufficient detail and still show how it fits into the whole diagram, and thus make working with large VL constructs lot easier. To create such focus and context views, the diagram layout correction needs to provide special support for the changing sizes of diagram parts.

The principle of direct manipulation can still be taken a step further by supporting pen-based editing (e.g., [5]): When creating a diagram, editors force the user to select among several diagram component types and thus force the user into a editor-specific way of editing. Pen-based editing, i. e. sketching a diagram with a pen and diagram recognition by the editor, would simplify the editing process. For instance, such an editor could be used as an easy means for sketching ideas in group discussions, e.g., when analyzing and designing software with UML diagrams. Future work will examine how the fault-tolerant parser of DiaGen can be used for recognizing diagrams which have been sketched with a pen. Pen-based editing will profit from current DiaGen features, in particular from abstract views: Instead of sketching diagram components with full detail level (e.g., left side of Figure 3), the user may sketch a simpler abstract diagram (e.g., right side of Figure 3) first and add more details later.

References

[1] R. Bardohl. GenGEd: A generic graphical editor for visual languages based on algebraic graph grammars. In *Proc. 1998 Symp. on Visual Languages (VL'98)*, pages 48–55, 1998.

[2] B. Bederson and J. Hollan. Pad++: a zooming graphical interface for exploring alternate interface physics. In *Proc. Symposium on User Interface Software and Technology 1994 (UIST'94)*, pages 17–26, 1994.

[3] B. Bederson and B. McAlister. Jazz: an extensible 2d+zooming graphics toolkit in Java. HICL Technical Report 99–07, University of Maryland, 1999.

[4] F. Brandenburg, M. Himsolt, and C. Rohrer. An experimental comparison of force-directed and randomized graph drawing algorithms. In *Proc. Graph Drawing 1995 (GD'95)*, LNCS 1027, pages 76–87, 1995.

[5] S. S. Chok and K. Marriott. Automatic construction of user interfaces pen-based computers. In *Proc. 3rd Int. Workshop on Advanced Visual Interfaces, Gubbio, Italy*, 1996.

[6] J. Clark and S. DeRose. XML path language (XPath). W3C recommendation 16 November 1999, W3C, 1999. http://www.w3.org/TR/xpath.

[7] G. Costagliola, A. De Lucia, S. Orefice, and G. Tortora. A parsing methodology for the implementation of visual systems. *IEEE Transactions on Software Engineering*, 23(12):777–799, Dec. 1997.

[8] J. Ebert, R. Süttenbach, and I. Uhe. Meta-CASE in practice: a case for KOGGE. In *Advanced Information Systems Engineering, Proc. 9th Int. Conf. (CAiSE'97)*, LNCS 1250, pages 203–216. Springer, 1997.

[9] G. W. Furnas. The fisheye view: a new look at structured files. Technical Memorandum #81-11221-9, Bell Laboratories, 1981.

[10] T. A. Keahey and E. L. Robertson. Techniques for non-linear magnification transformations. In *Proc. IEEE Symp. on Information Visualization, IEEE Visualization*, pages 38–45, 1996.

[11] O. Köth. Semantisches Zoomen in Diagrammeditoren am Beispiel von UML. Master thesis, University of Erlangen-Nuremberg, 2001. (in German).

[12] O. Köth and M. Minas. Generating diagram editors providing free-hand editing as well as syntax-directed editing. In *Proc. International Workshop on Graph Transformation (*GraTra *2000), Berlin*, March 2000.

[13] K. Marriott, S. S. Chok, and A. Finlay. A tableau based constraint solving toolkit for interactive graphical application. In *4th International Conference on Principles and Practice of Constraint Programming (CP'98), Pisa, Italy*, pages 340–354, Oct. 1998.

[14] M. Minas. Creating semantic representations of diagrams. In M. Nagl and A. Schürr, editors, *Int. Workshop on Applications of Graph Transformations with Industrial Relevance (*Agtive*'99), Selected Papers*, LNCS 1779, pages 209–224. Springer, Mar. 2000.

[15] M. Minas. *Specifying and Generating Graphical Diagram Editors [in German: Spezifikation und Generierung graphischer Diagrammeditoren].* Shaker-Verlag, Aachen, 2001.

[16] M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. Appears in *Journal of Science of Computer Programming (SCP)*, 2002.

[17] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages and Computing*, 6:183–210, 1995.

[18] Object Management Group. *Unified Modelling Langugage Specification.* http://www.omg.org/uml/.

[19] S. Pook, E. Lecolinet, G. Vaysseix, and E. Barillot. Context and interaction in zoomable user interfaces. In *Proc. Advanced Visual Interfaces 2000 (AVI'2000)*, pages 227–231, 2000.

[20] J. Rekers and A. Schürr. A graph based framework for the implementation of visual environments. In *Proc. 1996 Symp. on Visual Languages (VL'96)*, pages 148–155, 1996.

[21] M. Sarkar and M. H. Brown. Graphical fisheye views of graphs. In *Proc. Conference on Human Factors in Computing Systems 1992 (CHI'92)*, pages 83–91, 1992.

[22] D. Schaffer, Z. Zuo, S. Greenberg, L. Bartram, J. Dill, S. Dubs, and M. Roseman. Navigating hierarchically clustered networks through fisheye and full-zoom methods. *ACM Transactions on CHI*, 3(2):162–188, 1996.

[23] M. Sheelagh, T. Carpendale, D. Coperthwaite, and F. Fracchia. Making distortions comprehensible. In *Proc. 1997 Symp. on Visual Languages (VL'97)*, pages 36–45, 1997.

[24] D.-Q. Zhang and K. Zhang. VisPro: A visual language generation toolset. In *Proc. 1998 Symp. on Visual Languages (VL'98)*, pages 195–201, 1998.