# Two Opportunities and Challenges of Automatic Layout in Visual Languages

Christoph Daniel Schulze
Kiel University
cds@informatik.uni-kiel.de

## Abstract

Diagram-based visual languages are an established means for describing and developing software systems, but have an important drawback: users need to place diagram elements properly for a diagram to be properly readable, which can take a significant amount of time (Klauske and Dziobek 2010). Automatic layout algorithms aim to automate that task to improve user productivity. These algorithms enable new ways for working with visual languages, but present new challenges as well.

In this paper, I suggest to think about automatic layout algorithms in terms of *challenges* and *opportunities*. Challenges are problems that need to be solved for layout algorithms to yield good results. Opportunities are problems inherent in visual languages or new user interaction scenarios that can be tackled with automatic layout. To illustrate these terms, I summarize previous research on one example of each.

***Categories and Subject Descriptors*** D.1.7 [*Programming Techniques*]: Visual Programming

***Keywords*** Automatic layout, comment attachment, label management

## 1. Introduction

Visual languages are popular among developers, both for describing existing software systems (UML being an example) and to develop new ones (languages such as LabVIEW,[1] a system design and development platform). Visual languages have a reputation for being easy to understand and use, a natural way to communicate. After all, we quickly start scribbling diagrams on whiteboards or pieces of paper when trying to understand things or to explain them to others. If this serves us well, why then should computer-based visual languages be any different? For the purposes of this paper, we shall look at two approaches to answer that question.

First, there is the sheer size of diagrams. In contrast to diagrams scribbled on a sheet of paper, diagrams used to develop software systems can quickly grow so large that they cannot be displayed on a single screen without losing legibility. This forces developers to only look at small excerpts of the diagram at a time, which causes the mental overview to suffer and thereby decreases the diagram's usability. Often enough, the problem is not just the number of diagram elements. Most visual languages contain text labels to identify elements and to specify behavior. Depending on the language, these labels can grow very large and force the diagram to grow very large as well. To make matters worse, long text labels do not simply enlarge a diagram; they do so by making it wider, causing the aspect ratio to grow less and less favorable for displaying the diagram on today's common computer screens.

Second, a diagram is not easy to understand just because it is visual in nature (Petre 1995). In fact, a diagram whose elements are all stacked atop each other at the same position is visual in nature indeed, but not readable at all. Of course, this is an artificial example, but it does serve to highlight the fact that element placement hugely influences how quickly one can grasp what the diagram should express. Getting the placement right includes ensuring that a visual language's formatting guidelines are met. In class diagrams, for instance, people expect superclasses to be placed somewhere above their subclasses, while data flow languages require the majority of edges to point in the same direction to visually highlight the flow of data. Good placement also includes taking care that the diagram does not exhibit visual clues that contradict its content. For example, placing two unrelated elements neatly aligned in close proximity makes them look related, whether or not this is in fact the case (Wertheimer 1923). This is called *secondary notation*: information that through placement transcend what is contained in the diagram's pure structure. Getting the placement right is a lot of work. Based on observing developers in the automotive in-

[1] http://www.ni.com/labview

dustry, Klauske et al. (Klauske and Dziobek 2010) estimate that placement accounts for up to 30% of a developer's time.

Reducing that hit on developer productivity is what automatic layout algorithms aim to do. Instead of having to place all diagram elements manually, that task is delegated to the computer. While this solves the problem of too much time spent moving diagram elements around, it gives rise to new problems when it comes to secondary notation. Consider a diagram with two closely spaced elements. A human viewer will immediately assume the two to be related in some way and, when asked to modify the diagram, will probably try to keep this visual clue intact. What a given automatic layout algorithm will do in this situation depends strongly on the particular algorithm. Misue et al. (Misue et al. 1995) distinguish *layout adjustment* and *layout creation* algorithms. The former will try to keep the general topology of a drawing intact, and may in fact succeed in not destroying the implicit relation between the two elements. A layout creation algorithm, on the other hand, computes a new layout from scratch. Since it does not know about the relationship between the two elements, this can lead to them being placed far apart in the final layout, thereby destroying a potentially important visual clue.

Through the fast computation of layouts, automatic layout algorithms can serve as enablers for ways to improve the usability of visual languages. Let us come back to the problem of large text labels. Depending on what the user is trying to do at a given moment, the full content of one label may be important while not all the details of another label are of interest. Yet another label may not need to be shown at all, allowing the diagram to become smaller. However, what is and what is not interesting to a user changes dynamically as they concentrate on different parts of a diagram. This dynamic makes it impossible to exploit size changes with manual placement, but automatic layout is fast enough to quickly rearrange diagram elements to fit more of them on the screen.

### 1.1 Contributions

In this paper, we take a step back from the details of how automatic layout algorithms work and think about them instead in terms of what keeps them from being used and what new usage scenarios they enable. In this spirit, the main contribution is the motivation and introduction of the notion of *challenges and opportunities* of automatic layout algorithms. This helps us bring together two formerly separate lines of research we reported on previously: *comment attachment* (Schulze et al. 2016b) and *label management* (Schulze et al. 2016a). In doing so, we summarize results obtained so far in each of these areas.

### 1.2 Related Work

The ultimate goal of our research is to improve the usability of visual languages. It is thus worth recognizing that other researchers have also found that there is, in fact, room for improvement. This mainly affects two areas: the tooling, and the visual languages themselves. While editing environments for textual languages have matured to provide very advanced and effective editing features, editors for visual languages still leave something to be desired (Pietriga 2005; Klauske and Dziobek 2010). The languages themselves require a certain expertise when it comes to using secondary notation well (Petre 1995), which is a skill that must be learned.

It has been recognized that comments can relate to diagram elements differently (Eichelberger 2005). However, to the best of our knowledge we are the first to tackle the problem of inferring those relations to support automatic layout algorithms (Schulze and von Hanxleden 2014; Schulze et al. 2016b). The results of this line of research may be combined with research done on comments in textual languages. For example, checking the consistency between comments and the code they relate to (Tan et al. 2012) requires knowledge about that relation. Comment attachment may ultimately make such techniques applicable to visual languages.

The term *label management* and basic label shortening techniques were introduced by Fuhrmann as part of his dissertation (Fuhrmann 2011). We have since implemented and added to the techniques, have started to evaluate them, and have developed more advanced concepts as to how they can be integrated with automatic layout.

Label management aims to do something about the problem of too little screen space and can be used to reduce the amount of unnecessary information visible at a given time. There are other techniques that focus on similar problems. Fisheye views (Leung and Apperley 1994; Sarkar and Brown 1992) emphasize a region of focus over its surroundings by scaling diagram elements up and down, but suffer from legibility problems either due to optical distortion or due to very small font sizes. Other approaches show or hide elements depending on the zoom level (Been et al. 2006), but cannot reduce the amount of information shown by an element. Osman et al. (Osman et al. 2014) take the approach of ranking elements in class diagrams by importance and allow the user to hide less important elements. In contrast to our methods, they do not take spatial considerations into account and can only show and hide elements, not modify them. Similarly, Musial and Jacobs progressively hide details of classes in UML class diagrams as the classes are further away from what the user is currently focusing on (Musial and Jacobs 2003). They rely purely on the concept of *focus and context* (Card et al. 1999), which our methods are compatible with, but not limited to.

### 1.3 Outline

We start by properly introducing what we mean by opportunities and challenges. This is followed by a discussion of two examples of opportunities and challenges, along with a summary of results obtained so far in these areas. We conclude with a discussion of the presented material.

## 2. Challenges and Opportunities

The problem of placing diagram elements and routing edges is a very complex one. Consider the three main phases of the layered layout approach (Sugiyama et al. 1981): layer assignment, crossing minimization, and horizontal coordinate assignment. Depending on the optimization criteria, the first two must solve NP-complete problems (Di Battista et al. 1999): finding an order of nodes that minimizes the number of edge crossings, for example. This leads us to the following definition:

> A *challenge* is a problem inherent in a layout approach that needs to be solved in order for the approach to work or to improve the layout results it produces.

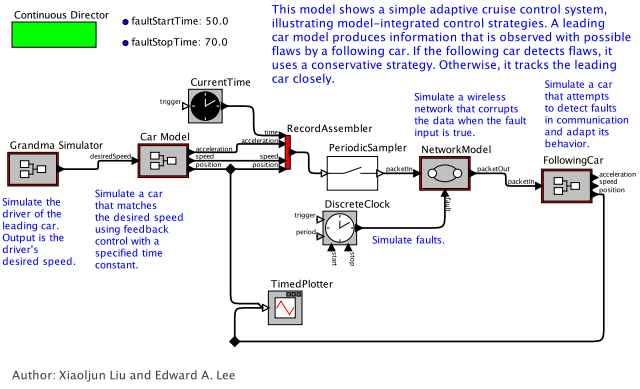
Research on layout algorithms usually focuses on efficient methods to tackle the complex and hard algorithmic problems, which of course are big challenges for algorithm developers. In the process, seemingly smaller and perhaps less theoretically interesting problems often seem to be overlooked and regarded as trivial implementation details. It is the experience at our research group, however, that if left unsolved these problems can be the reason why users fail to use automatic layout more than once. The experience that the layout algorithm has produced an unfortunate and obvious placement problem which is easily spotted and which "I could have easily done better" can lead users to never hit a layout button again. What's more, many small and simple problems turn out to be not as small and simple once properly investigated. If layout algorithm developers want their algorithms to be widely adopted, both the obviously hard and the seemingly smaller challenges have to be addressed.

Problems are not to be found only in layout algorithms, but also in the design of visual languages. One example are long labels: if a visual language tends to make developers use long labels, the diagrams they produce will often grow too wide to be properly displayed on standard computer screens. This leads us to the following definition:

> An *opportunity* is a usability problem in the design of a visual language which can be mitigated through the use of automatic layout, or a new user interaction scenario enabled by automatic layout.

In the case of long labels, automatic layout algorithms can be used to shorten them and then modify the layout to take advantage of the saved space. Of course, taking advantage of such opportunities does pose additional problems to be solved on the layout algorithm side. In contrast to challenges, however, these problems are not inherent to the layout algorithms, but arise from problems with the languages that are to be laid out.

Starting with the next section, we will look at a challenge and an opportunity and summarize results obtained so far.




Author: Xiaojun Liu and Edward A. Lee

Figure 1: A small node-link diagram as laid out manually using the Ptolemy II block diagram language. The developer in this case took care to make the relation between comments and diagram elements clear through placement. This particular diagram is part of the example models that ship with Ptolemy II.

## 3. Comment Attachment

Similar to textual languages, comments in visual languages help explain what is expressed through a diagram (see Figure 1). In textual languages it is usually very apparent what a comment refers to. Languages such as Java even make rules part of the language definition. There are usually no such rules in visual languages. If the comment is not explicitly connected to the diagram element it refers to (which not all languages support, and not all developers use), their relation is defined through more implicit and ill-defined means, such as how the comment is placed relative to other diagram elements, or the comment's text.

If an automatic layout algorithm encounters a comment with no explicit connection to the diagram element it relates to, it may end up placing the two far apart. After all, it has no idea that they should be placed in close proximity. We thus need to make these relations available to layout algorithms. This requires developing methods that recognize implicit clues in order to turn them into *explicit attachments* between comments and diagram elements that layout algorithms can process. Computing these attachments is called the *comment attachment problem* (Schulze and von Hanxleden 2014).

Note that we would classify this as a challenge. Relating comments and diagram elements through placement in manual layouts works just fine, so this is not a problem of the visual language itself. The problem much rather lies in the layout algorithms that do not recognize these relations and may consequently fail to preserve them.

It is obvious that all comment attachment methods are limited to be heuristics: there simply are no universal rules people follow when writing and placing comments. Still, we wanted to find out whether a number of obvious and easy-to-implement heuristics might be enough to infer the relations between comments and diagram elements. In the past,

we have evaluated the performance of several such heuristics (Schulze et al. 2016b) with a set of example diagrams taken from Ptolemy II (Ptolemaeus 2014), a set of visual languages developed at UC Berkeley focusing on the design and interplay of heterogeneous systems.

**Font Size** Text documents usually start with a heading set in a font size larger than the rest of the text. Similarly, if a diagram contains a comment whose font size is larger than that of all other comments, we might hypothesize that it contains the diagram's title. Such a comment of course does not relate to any specific diagram element and should thus be left alone. This heuristic works well in that if it flags a comment as containing the diagram's title, this is correct in 98% of all cases.

**Text Prefix** We found that comments that describe the diagram as a whole as opposed to a single element often start with a similar prefix, such as "This model…", or "Authors: …". Such comments do not relate to any single diagram element. How successful this heuristic is depends on how developers use a given visual language. It works well for the surveyed example diagrams in that if the heuristics flags a comment it is always correct, but it may fail for other languages.

**Text References** It seems obvious that if a comment actually mentions a diagram element by name, the two will probably be related. In practice, though, the comment may mention other elements as well, or the named element may just serve to distinguish it from the element the comment actually relates to.

**Distance** The distance between a comment and other diagram elements is probably the most obvious heuristic. However, the distance between a comment and the element it relates to is rather fuzzy in practice. Worse, often that element is not even the one closest to the comment. There is also the question of how far a comment and a diagram element may be apart for them to still be considered to be related.

**Alignment** Graphic designers use alignment as one of the main tools to establish relations between graphical elements. To our surprise, we found that developers working with visual languages care a lot less about alignment, thus rendering this heuristic almost useless.

In our evaluations, the best combination of heuristics was able to make correct decisions in about 90% of all cases. The heuristics that fared best were those based on established conventions for the given visual language, such as the font size and text prefix heuristic. Heuristics that rely more on developers having a grasp on graphic design are more problematic or even useless.

These results have yet to be verified with other visual languages. Still, they already allow for two observations. First, a good attachment result largely depends on how the heuristics are configured. The distance heuristic, for example, is very sensitive to the maximum distance that comments and diagram elements may be apart to still be considered related. Finding good settings can be challenging, especially since every team of developers can have different coding conventions. The second observation directly follows from this difficulty: inferring relations between comments and other diagram elements will always be prone to some margin of error. It therefore seems sensible for language designers to make explicit relations part of the language's design, and to make it as easy as possible to define them through the tooling.

## 4. Label Management

Few visual languages can get away without incorporating at least a bit of text. Languages based on state machines usually use text labels to specify the conditions upon which a transition can be taken as well as the actions executed when it is. SCCharts (von Hanxleden et al. 2014) is such a language. Even for small diagrams, transition labels can grow rather long, which in turn forces the diagrams into awkward aspect ratios (see Figure 2. Such diagrams cannot be shown completely without sacrificing readability, even if they could be drawn perfectly well if the labels were shorter.

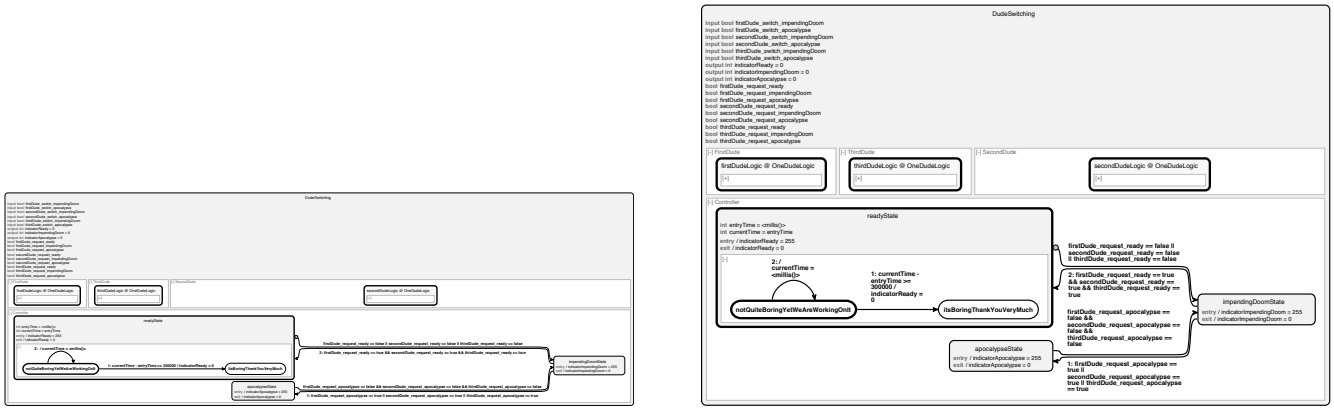
Label management aims to improve this situation by changing the content of labels and computing a new layout that takes advantage of the label size changes. The actual motivation to do so can vary:

- Increase the zoom level. If the diagram should fit on the screen, a smaller diagram results in a larger zoom level, thereby increasing legibility. If the zoom level is fixed, smaller labels allow for more diagram elements to fit on the screen, thereby increasing the user's overview.
- Reduce visual clutter. Whether the information a label contributes is of interest to the user or not depends on the task they are working on and what they are currently focusing on. Reducing the detail of or even removing unnecessary labels reduces visual clutter and allows more interesting elements to fit on the screen,

As mentioned in Section 2 this is an example of an opportunity. The problem to be solved—labels quickly becoming too large and increasing the size of diagrams—is inherent in the visual language. Automatic layout is used as an enabler for label management. After all, simply changing the size of labels does not accomplish much if we do not change the diagram's size as well.

When reducing the size of labels two questions need to be answered: how do we shorten labels, and when do we do it?

Let us begin with the *how*. Fuhrmann (Fuhrmann 2011) proposed a number of basic shortening strategies shown in Table 1. Syntactical abbreviation simply cuts off a label's text once a given target length is reached. Semantical abbreviation tries to cut out unnecessary details, only leav-



(a) Without label management.

(b) With label management.

Figure 2: The same diagram (a) with all labels displayed without modification, and (b) with label size managed by wrapping the label's text appropriately. Assuming that the space available to draw both diagrams has a typical aspect ratio (such as 4:3), or alternatively assuming a fixed width (as in this figure), the diagram in (b) can be drawn with a larger zoom factor.

| Type | | Example |
|---|---|---|
| Abbreviate | syntactical | `(not SignalA) xor (...` |
| | semantical | `SignalA, SignalB / SignalC` |
| Wrap | syntactical (hard) | `(not SignalA) xor (not SignalB) / SignalC(cou nter)` |
| | syntactical (soft) | `(not SignalA) xor (not SignalB) / SignalC( counter)` |
| | semantical | `(not SignalA) xor (not SignalB) / SignalC(counter)` |

Table 1: Basic label shortening strategies (Fuhrmann 2011).

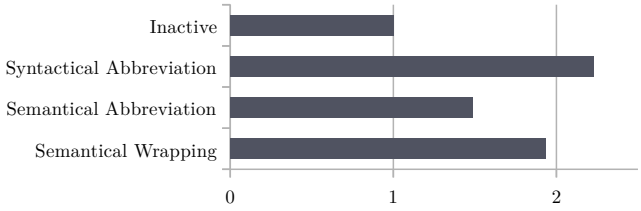



Figure 3: Increase of the zoom factor we achieved for our set of evaluation models with different label management strategies as compared to without label management.

ing in what is required to get an idea of the original text. The three wrapping strategies are based in the observation that the width of labels is often more problematic than their height. Syntactical hard and soft wrapping work just like they do in text editors. Semantical wrapping makes sense in cases where labels follow a strict syntax: it restricts the points where line breaks can be inserted in an attempt to visually reflect the text structure.

We applied some of these heuristics to a set of 93 SCCharts containing a total of 6230 transition labels and measured the zoom level required to fit a whole diagram on the screen, with and without shortened labels. The results, shown in Figure 3, indicate that significant zoom level increases are possible. It does not come as a surprise that syntactical abbreviation is most effective in that regard: it always achieves the target length and does not increase a label's height. Other strategies may be better, though, in terms of usability.

Let us now turn to the *when* of shortening labels, which will also answer the question of how to determine the target length. The most obvious way to decide which labels to shorten is to use a *preprocessing approach* before the automatic layout algorithm is triggered. The preprocessing can make decisions based on information such as what parts of a diagram are active while a simulation or debugging session is running, or which diagram elements are selected. It is here that the preprocessing approach can easily be integrated with focus and context concepts (Card et al. 1999).

The preprocessing approach has the potential drawback that labels are shortened regardless of whether this actually has any impact on the diagram's size. This is because the necessary information are not available until automatic layout is triggered. This is why we developed the *feedback loop approach*. Here, a preprocessing still configures how each label *can* be shortened, but the decision of whether or not it *will* be shortened is then left to the layout algorithm.

The two approaches also differ in how the target length for labels is determined. The preprocessing approach needs to set a target length more or less arbitrarily, perhaps based on user input. The feedback loop approach can allow labels to be as long as possible before having too much of an impact on their diagram's size.

## 5. Discussion

We have introduced the notions of opportunities and challenges in the context of automatic layout algorithms. To provide an example for each, we have summarized our findings regarding two lines of research: comment attachment (a challenge) and label management (an opportunity). The former served to illustrate how automatic layout algorithms are still posing challenges for algorithm designers when it comes to secondary notation. The latter served to illustrate, however, that automatic layout algorithms can also open up opportunities for new ways to help users accomplish their tasks, for example by auto-generating synthesized views that dynamically adapt to changing requirements.

We believe that algorithm designers often neglect the (seemingly) small challenges. In practice, however, we found that it is these details that keep users from embracing automatic layout. We also believe that we are just starting to realize the opportunities enabled by automatic layout algorithms.

There is much potential for future work. Regarding comment attachment, the methods need to be tested with more visual languages. Also, more complex heuristics could improve the quality of computed attachments. Regarding label management, a study will be required to show how label management impacts usability.

## References


K. Been, E. Daiches, and C. Yap. Dynamic map labeling. *IEEE Transactions on Visualization and Computer Graphics*, 12(5): 773–780, Sept 2006. doi: 10.1109/TVCG.2006.136.

S. K. Card, J. Mackinlay, and B. Shneiderman. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, Jan. 1999. ISBN 1558605339.

G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999. ISBN 0-13-301615-3.

H. Eichelberger. *Aesthetics and Automatic Layout of UML Class Diagrams*. PhD thesis, Bayerische Julius-Maximilians-Universität Würzburg, 2005.

H. Fuhrmann. *On the Pragmatics of Graphical Modeling*. Dissertation, Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, Kiel, 2011.

L. K. Klauske and C. Dziobek. Improving modeling usability: Automated layout generation for Simulink. In *Proceedings of the MathWorks Automotive Conference (MAC'10)*, 2010.

Y. K. Leung and M. D. Apperley. A review and taxonomy of distortion-oriented presentation techniques. *ACM Transactions on Computer-Human Interaction*, 1(2):126–160, June 1994.

K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages & Computing*, 6(2):183–210, June 1995.

B. Musial and T. Jacobs. Application of focus + context to UML. In *APVis '03: Proceedings of the Asia-Pacific symposium on Information visualisation*, pages 75–80, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc. ISBN 1-920682-03-1.

M. H. Osman, M. R. V. Chaudron, and P. van der Putten. Interactive scalable abstraction of reverse engineered UML class diagrams. In *21st Asia-Pacific Software Engineering Conference, APSEC 2014, Jeju, South Korea, December 1-4, 2014. Volume 1: Research Papers*, pages 159–166, 2014. doi: 10.1109/APSEC.2014.34. URL `http://dx.doi.org/10.1109/APSEC.2014.34`.

M. Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6): 33–44, June 1995.

E. Pietriga. A toolkit for addressing HCI issues in visual language environments. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 145–152, Sept 2005. doi: 10.1109/VLHCC.2005.11.

C. Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. URL `http://ptolemy.org/books/Systems`.

M. Sarkar and M. H. Brown. Graphical fisheye views of graphs. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 83–91. ACM, 1992. ISBN 0-89791-513-5. doi: http://doi.acm.org/10.1145/142750.142763.

C. D. Schulze and R. von Hanxleden. Automatic layout in the face of unattached comments. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'14)*, Melbourne, Australia, July 2014.

C. D. Schulze, Y. Lasch, and R. von Hanxleden. Label management: Keeping complex diagrams usable. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'16)*, 2016a.

C. D. Schulze, C. Plöger, and R. von Hanxleden. On comments in visual languages. In *Proceedings of the 9th International Conference on the Theory and Application of Diagrams (DIAGRAMS'16)*, 2016b.

K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109–125, Feb. 1981.

S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. @tComment: Testing Javadoc comments to detect comment-code inconsistencies. In *Fifth IEEE International Conference on Software Testing, Verification and Validation*, pages 260–269, April 2012. doi: 10.1109/ICST.2012.106.

R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. O'Brien. SCCharts: Sequentially Constructive Statecharts for safety-critical applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, Edinburgh, UK, June 2014. ACM.

M. Wertheimer. Untersuchungen zur Lehre von der Gestalt. II. *Psychologische Forschung*, 4(1):301–350, 1923.