Electronic Communications of the EASST
Volume 13 (2008)

Proceedings of the Second International Workshop on Layout of (Software) Engineering Diagrams (LED 2008)

Exploiting the Layout Engine to Assess Diagram Completions

Steffen Mazanek, Sonja Maier, Mark Minas

14 pages

Guest Editors: Andrew Fish, Harald Störrle
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Exploiting the Layout Engine to Assess Diagram Completions

Steffen Mazanek[1], **Sonja Maier**[2], **Mark Minas**[3]

[1] steffen.mazanek@unibw.de
[2] sonja.maier@unibw.de
[3] mark.minas@unibw.de
Institut für Softwaretechnologie
Universität der Bundeswehr München, Germany

Abstract: A practicable approach to diagram completion is to first compute model completions on the abstract syntax level. These can be translated to corresponding diagram changes by the layout engine afterwards. Normally, several different model completions are possible though. One way to deal with this issue is to let the user choose among them explicitly, which is already helpful. However, such a choice step is a quite time-consuming interruption of the editing process. We argue that users often are mainly interested in completions that preserve their original diagram as far as possible. This criterion cannot be checked on the abstract syntax level though. In fact, minimal model changes might still result in enormous changes of the original diagram. Therefore, we suggest to use the layout engine in advance for assessing all possible model completions with respect to the diagram changes they eventually cause.

Keywords: layout, diagram completion, assessment

1 Introduction

With the advent of more and more visual domain-specific modeling languages, user assistance for diagram editors becomes increasingly important. Therefore, we have presented a novel approach to *diagram completion* recently [MMM08b], which can be applied to diagram languages specified by means of graph grammars. The possible completions are gathered while parsing.

Given a (possibly incorrect) diagram, a diagram completion basically is a set of modifications that transform the input diagram into a correct target diagram. Consider the Nassi-Shneiderman diagram (NSD) given at the left-hand side of Fig. 1. It consists of two simple statements that do not form a correct NSD, because they do not touch each other vertically as required. Some

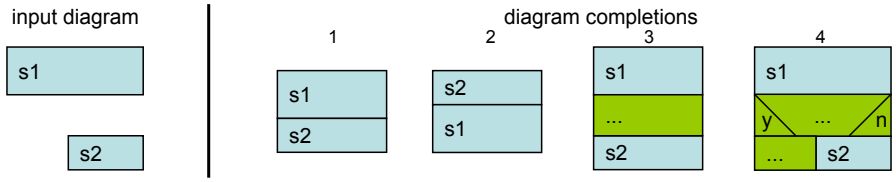

Figure 1: Diagram completion

possible completed diagrams are shown at the right-hand side of the figure. The given statements, e.g., can be moved and resized such that they touch each other properly. Their order in the resulting diagram thereby can be freely choosen (cf. diagrams 1 and 2). Alternatively, the two isolated statements can be connected indirectly by inserting one or more new components right between them (cf. diagrams 3 and 4). In general, a diagram completion either inserts additional diagram components or spatially rearranges existing ones (or both). Applying a diagram completion, thus, heavily relies on a powerful layout engine.

This simple example already shows that in general several completions are possible for a given diagram. In fact, we could attach an arbitrary number of additional statements to an arbitrary NSD and still get a correct resulting NSD (provided at the end there are no isolated statements left). So it is surely necessary to restrict the number of new diagram components to be introduced. Even so, there might be way too many different completions to be still helpful for the user.

Fig. 2 shows a (textual) choice dialog with all completions of size one (only one new component) for our example diagram. This actually is a screenshot of a corresponding DIAGEN-editor [Min02] with completion support [MMM08b]. The phrase "with gluing" means that new spatial relations are introduced between existing components. For NSDs this means that touch-relations are established between previously apart corners of existing components. These new relations are highlighted in the example preview diagrams at the right-hand side of the figure. In our implementation such a preview indeed can be triggered by clicking on a particular item in the list.

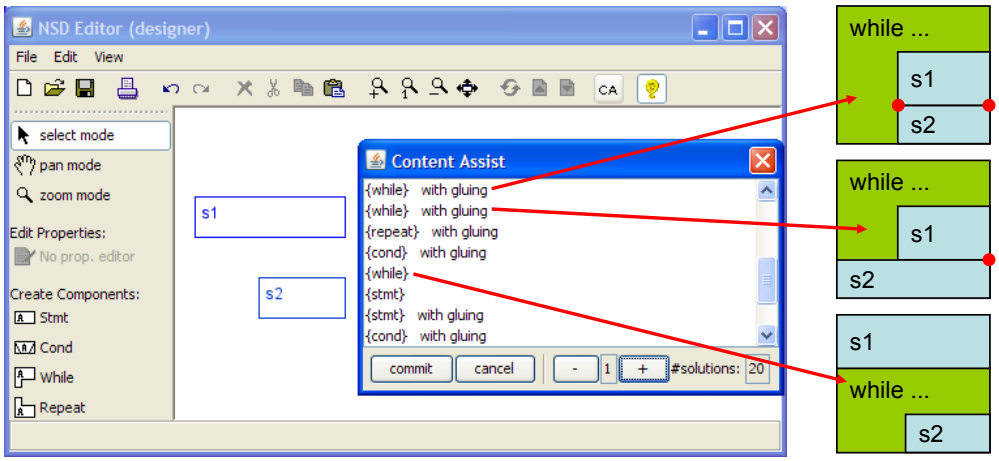

Figure 2: Choosing completions as suggested in [MMM08b]

Although such choice has some merits already, the cognitive overload caused by this multitude of completions[1] is problematic for a user (even if solutions directly would be presented in a more graphical manner). To overcome this problem, in this paper we propose a generic approach for ranking sets of completions. We basically use information from the concrete syntax level to rank abstract completions. That way, the benefit gained from the diagram completions can

[1] For the example in Fig. 2 there are 20 possible completions of size one, alone six where a new while-component is inserted (for each of the given previews the two existing statements can also be interchanged).

be maximized, because we can significantly reduce the cognitive overload. For instance, we can only provide the best completions as a choice to the user. We even can provide a shortcut for the automatic application of the best completion, which we show to be useful in certain situations. All in all, our approach appears to be very useful for both learning visual languages and improving editing productivity.

We proceed as follows: First, we discuss the scope of our approach and state the problem more clearly (Sect. 2). Thereby, we justify the claim that our approach is quite widely applicable (even for model-based diagram editors). In Sect. 3 we introduce our approach in its generality. Thereafter, we discuss its integration in the DIAGEN system (Sect. 4). Finally, we summarize the benefits of our approach from the usability's perspective (Sect. 5) and conclude (Sect. 6).

2 Scope and Problem Description

Although this paper continues our previous work on the DIAGEN system, the described problem has a much wider scope. Diagram editors based on the following principles are affected:

- distinction between concrete and abstract syntax, i.e. diagram and model,
- temporarily incorrect diagrams are allowed (basic requirement for free-hand editing),
- syntax analysis is performed on the abstract syntax level, and additionally
- completions can be computed on the abstract syntax level.

The distinction between abstract and concrete syntax is widely accepted. For instance, Rekers and Schürr have proposed a graph based framework for visual environments [RS96] where a so-called *spatial relations graph* (SRG) is derived from the physical layout by a graphical scanner; the SRG in turn is a representation of the corresponding *abstract syntax graph* (ASG). Their general approach, however, is so powerful that parsers unfortunately run in exponential time.

A more practicable approach has been presented by the third author [Min02]: In his DIAGEN system editors use hypergraphs as a model for diagrams [Min00]. Diagram components thereby are represented by hyperedges whose incident nodes represent the component's attachment points/areas. Similar to the approach of Rekers and Schürr, an SRG is created by a graphical scanner, which adds spatial relations edges where appropriate. The corresponding ASG is created from the SRG by the application of special graph transformation rules.[2] This ASG then is syntactically analyzed according to a hyperedge replacement grammar (HRG). HRGs [DHK97] are a context-free graph grammar formalism well-suited for the definition of visual languages. In particular, the parsing complexity is polynomial for practically relevant languages. In [MMM08a] we have shown that hypergraph completions with respect to HRGs also can be computed efficiently. Such a completion can perform an arbitrary number of the following two kinds of modifications: "glue distinct nodes" and "insert a new hyperedge". The resulting hypergraph is guaranteed to be a member of the grammar's language.

Nowadays, however, model-based approaches for defining visual languages are also very popular. Here, the abstract syntax of a visual language is defined via a metamodel. In this domain,

[2] In DIAGEN the SRG and the ASG actually are hypergraphs rather than graphs.

syntax analysis means that a corresponding object structure for the given diagram has to be created. Thereby, conformance to the metamodel has to be ensured. For this purpose constraint logic programming is frequently used. For instance, in DIAMETA, the metamodel-based sibling of DIAGEN, the syntactical analysis of freely drawn diagrams is performed by solving a constraint satisfaction problem [Min06]. In addition, Sen et al. recently have presented an approach to model completion with respect to a metamodel [SBV07], which has been incorporated into AToM^3 [LVA04]. Their completions are also derived using constraint logic programming. A similar approach is used in the Generic Eclipse Modeling System GEMS [WSNW08] and the SmartEMF [HCW07] inference engine for the Eclipse Modeling Framework. We refer to these approaches, because the method presented in this paper might also be applicable there.

To make use of an abstract model/graph completion, it has to be translated into corresponding changes of the diagram again. In [MMM08b] we have described how we actually have extended the DIAGEN editing process in order to translate hypergraph completions into diagram completions. As expected, the layout engine has played an important part in this context. However, this kind of assistance is only useful, if the choice among the different completions is easy. Just sorting them by their size is not sufficient though. There still might be several solutions which take to much time to inspect (cf. Fig. 2). Even worse, the user's train of thought [BKC00] is interrupted in a way counter to the idea of free-hand editing. So an additional, complementary instrument is sought.

Another challenge is the fact, that the changes the layout engine will actually perform in order to apply a completion are sometimes hard to predict and may badly surprise the user. For instance, the orthogonal ordering of the existing diagram components might be completely destroyed such that existing inter-component relations like below, above, left from or right from do not hold anymore. A small change of the model, thus, might result in enormous changes in the user's original diagram. Larger changes of the model, in contrast, might sometimes cause only little visual changes of existing components (cf. Fig. 1).

Generally, we can say that little changes to existing components better preserve the mental map of the user [ELMS91, Bra01, vP07], which we already know to be a highly desirable objective. A completion rearranging the whole diagram probably does not meet the user's intention. Even if we can animate the changes (a common technique in dynamic graph drawing for the preservation of the mental map [Bra01]), the user might still lose track if the diagram is of practical size.

In Fig. 3 this problem is clarified using the example diagram from Fig. 1 again. We also show its corresponding model, a hypergraph as used in the DIAGEN system.[3] In the middle row of the figure we provide some of the hypergraph completions as computed by the parser discussed in [MMM08a]. These completions are sorted by the number of edges they add. At the bottom the corresponding diagrams are shown. In ASGs of NSDs a common node visited by component edges represents touching corners of the corresponding diagram components. The problem is obvious here. Even if a completion is minimal on the abstract syntax level (diagrams a) and b) in Fig. 3), it might still destroy the orthogonal ordering of the diagram components and, thus, the mental map of the user. For instance, in diagram b) the vertical order of the two given statements is turned upside down. This annoying effect is caused by the layout constraint that asks the height

[3] A hyperedge is represented by a box with the particular edge label inside. A node is represented by a black dot. The lines between edges and nodes indicate that an edge visits a node.

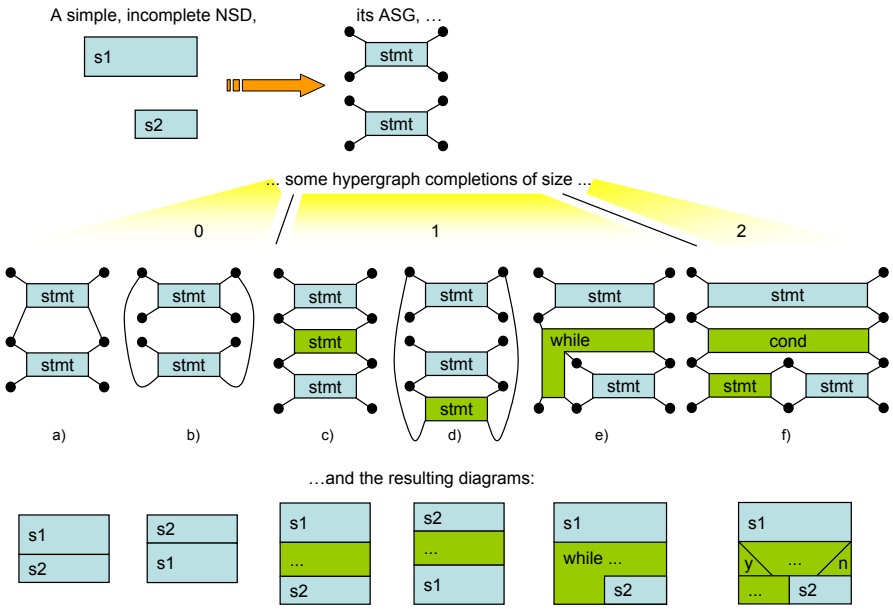

Figure 3: Clarification of the problem by example

of statements to be non-negative and greater than a given minimal height.

This example demonstrates, that choosing explicitly among these model completions can be a quite frustrating task. In the following we propose an approach to avoid this problem. Whereas completions can be computed on the abstract syntax level most conveniently, the concrete syntax by all means has to be considered in order to get a helpful ranking.

3 Our Approach in general

In this section, we suggest a general approach that makes this try then inspect cycle for all possible completions much more convenient. Even better, we show that the distracting choice step often can be completely avoided.

We assume that users want their original diagram to be preserved as far as possible. The idea now is to sort the completions with respect to the diagram changes they eventually cause (on concrete syntax). That way, we can present those completions first that best preserve the original diagram. This is very helpful for beginners, since they precisely see and understand which adaptations are necessary in order to correct their diagram. We even can provide a shortcut for the automatic application of the best completion. This is meaningful for expert users who might already have in mind this distinguished completion. It might be quite tedious though to actually perform the necessary diagram changes by hand. So they probably appreciate assistance for this step.

This kind of assistance is motivated in Fig. 4. In the situation shown at the left-hand side of the figure, three complex mouse actions (click+drag) are necessary in order to correct the given diagram in the obvious way (the situation would be even worse if the statements had different

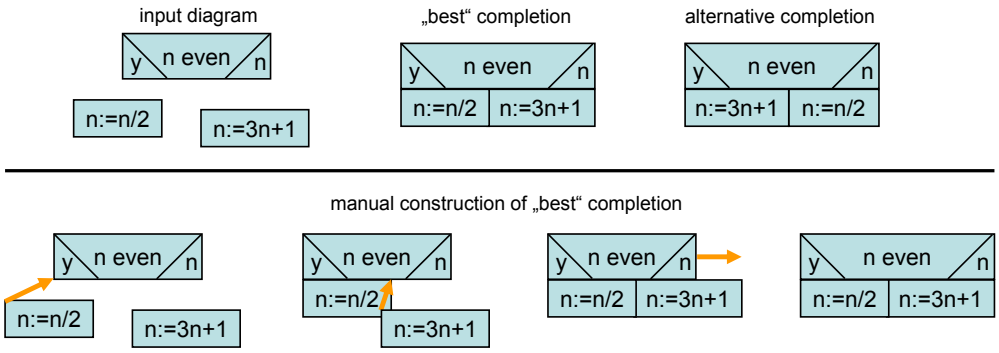

Figure 4: Correcting diagrams by hand is tedious

heights). These manual editing actions can be completely avoided by using the shortcut for the automatic application of the best completion (also shown in the figure). In contrast, a shortcut for applying just an *arbitrary* completion would feel like rolling the die. For instance, the given input diagram could also be put together in a different way, where the branches below the condition are swapped. However, this is not the solution we suppose to be intended by the user.

In Fig. 5 the diagram completions shown in Fig. 3 are sorted. The first and the second diagram are best in the sense that existing diagram components do not need to be changed at all. The first one, however, should be preferred, because fewer edges are added to the ASG (so only using concrete syntax of existing components does not seem to be enough either). The third diagram is also quite good. Here, the two statements are connected by introducing an additional statement. Therefore, the sizes of the existing statements have to be adapted. Minimal changes on the ASG are caused by solutions four and five. Only nodes are glued and no edge is inserted at all. However, on the diagram level larger changes are caused, because the two statements have to touch each other vertically and thus have to be moved. Diagrams five and six are really undesirable solutions since they destroy the orthogonal ordering of the existing components.

An overview of our approach is given in Fig. 6. First, the given diagram is translated to its model, e.g., by using graph or model transformation. This model in turn is syntactically analyzed

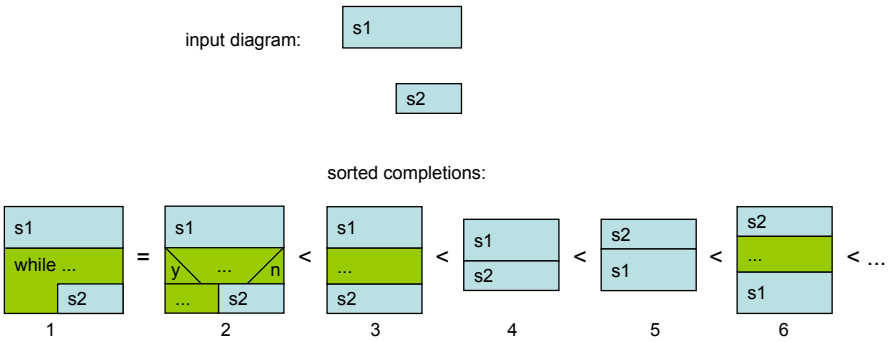

Figure 5: Sorting possible completions

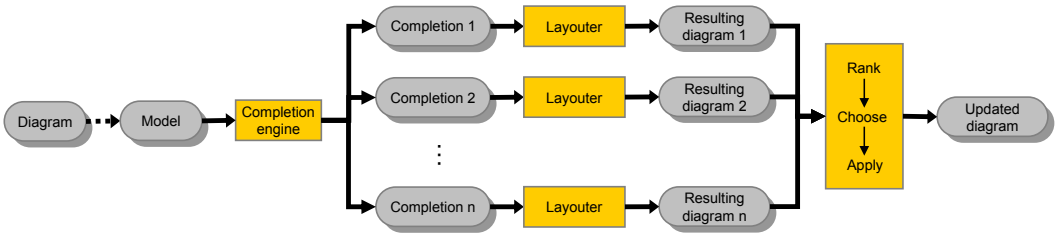

Figure 6: Ranking completions, survey of our approach

and completions are computed if required. The resulting diagrams for all possible completions are precomputed by the layouter and compared with respect to a metric. If the user wants to choose explicitly, he can get a sorted list of all completions where the best ones are presented first. If the described shortcut has been invoked, the best completion can be directly applied to the user's original diagram.

Note, that this approach heavily relies on layout with minimal changes as, e.g., discussed in [MV93]. This means, that the layouter must not apply a static layout algorithm to the whole diagram. Rather it should preserve existing layout parameters like positions and sizes as far as possible. Otherwise, the assessments for comparing the resulting diagrams are meaningless.

Several possibilities, some simple, some more complicated, are available for a metric. We do not go into detail here, since this question already has been discussed in "mental map"-papers like [ELMS91] or [BT98]. However, the most prominent candidates probably are "orthogonal ordering", "Euclidean distance", and "Manhattan distance". We have not evaluated yet which one is most appropriate.[4] One additional factor might be incorporated though: the information from the abstract syntax level, e.g., how many new edges have been embedded, how many nodes are glued and so on. This might play a role.

4 Realization in DIAGEN

In this section we describe our editing process with completion support as realized in DIAGEN. Complementary to [MMM08b], we focus on layout here.

As suggested in [MV93], we can use *constraint hypergraph grammars* (CHG) to define the syntax of diagrams and specify layout constraints at the same time. CHGs are an extension of HRGs, where edges and nodes can receive attributes, the values of which represent layout-related parameters like size or position on the screen. Relationships between these attributes can be added to the productions (similar to attributed grammars as known from the string setting). Fig. 7 shows the CHG for NSDs. Note, that it is also possible to provide an all hand-written layouter, but CHGs have appeared to be very well-suited for our purpose (as we further discuss later).

Each production consists on its left-hand side of a hypergraph with a single nonterminal edge and the nodes visited. The right-hand side of every production is an arbitrary hypergraph of terminal and nonterminal hyperedges. Application of a production to a hypergraph is similar to

[4] For our application, however, we suppose that their results are quite similar in most cases.

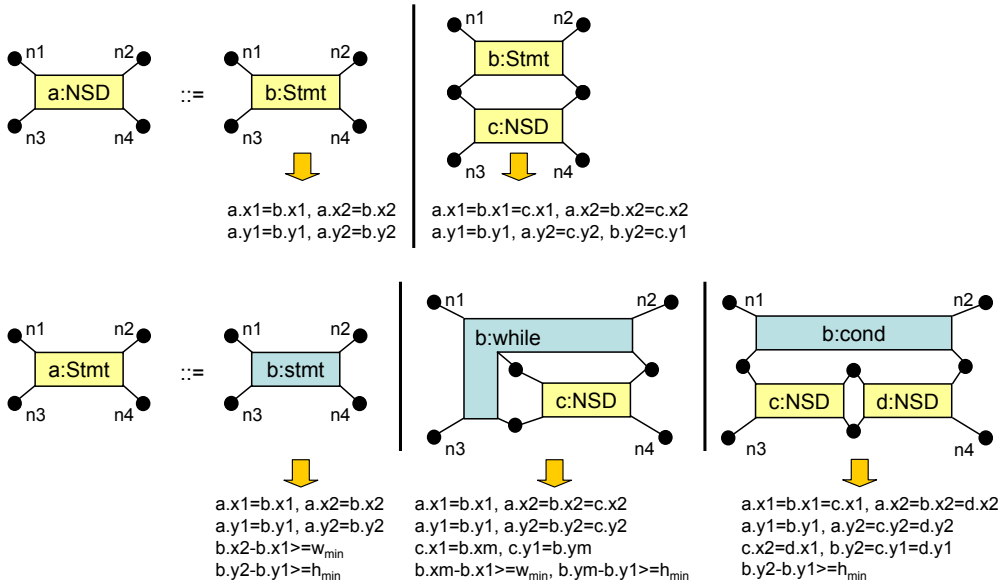

Figure 7: Constraint hypergraph grammar of NSDs

string grammars: if the left-hand side is a subgraph of the hypergraph, this subgraph is removed and replaced by the corresponding right-hand side. The resulting hypergraph is said to be derived from the first hypergraph. In order to specify which node from the right-hand side replaces which node from the left-hand side, corresponding nodes are labelled with the same names. The language of a hypergraph grammar is the set of hypergraphs that consist of terminal hyperedges only and that are derivable from the starting graph (in case of NSDs this is a single nonterminal NSD edge with incident nodes).

The language of NSDs is recursively defined. An NSD is basically a chain of statements (nonterminal Stmt), where a statement in turn is either a primitive statement (terminal stmt), a while-loop whose body is an NSD again, or a condition followed by a yes and a no branch (NSDs again). The constraints, for instance, ensure that a condition component is left aligned with its yes branch and right aligned with its no branch. Furthermore, the right-hand side of the left branch has to touch the left-hand side of the right branch, and so on.

A bird's eye view on our editing process is shown in Fig. 8. The user of a DIAGEN editor can freely edit diagrams using the drawing tool. After each editing operation a chain of processing steps is performed. First, the scanner derives the SRG from the arrangement of diagram components. Thereby, spatial relations edges are introduced where components are connected in a way relevant for the particular visual language. An SRG of an example NSD is part of Fig. 9.

Thereafter, a reduction step is performed to reduce complexity, very similar to lexical analysis in string parsing. This allows for more efficient parsing and readable grammars. In the case of NSDs, nodes connected via spatial relationship edges are unified. This results in an ASG like the one also shown in Fig. 9.

Next, the parser performs the syntactical analysis of the ASG. Thereby, a derivation structure is constructed (if any). In the following, this derivation structure can be used by the layouter

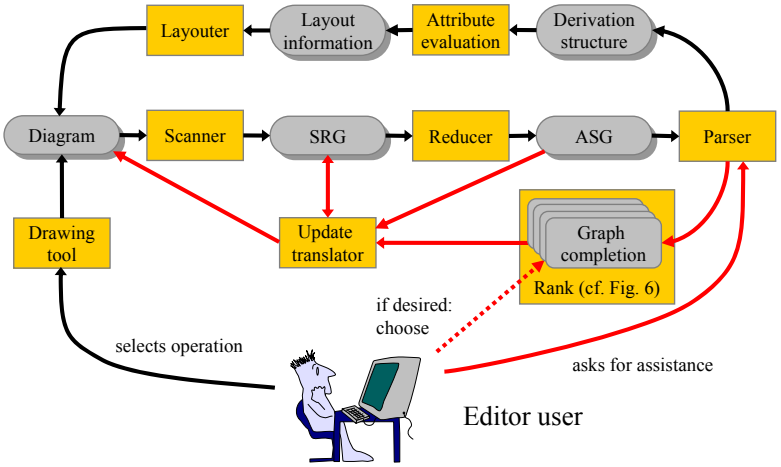

Figure 8: Editing process with completion support

and for highlighting correctly recognized parts of the diagram. In case of a CHG, this derivation directly establishes a constraint satisfaction problem whose solution is a layout for the diagram [MV93].

The feature diagram completion now is incorporated as follows into this standard DIAGEN editing process: If the user explicitly asks for assistance, the parser is triggered again with the desired size of completions as a parameter. It computes all possible graph completions of this size (cf. [MMM08a]). Those are ranked as described in the last section (we have used Manhattan distance [Kra87] as a metric). The user now can choose among them (if desired). Since the best completions are presented first, this choice can be made very fast. In addition, we provide a shortcut for the automatic application of the best completion. Therefore, the editor user can specify the maximal size of desired completions in the editor properties. For our NSD editor, two is a practicable default value for this parameter.

To actually apply a particular completion, it first has to be embedded into the SRG using the update translator. For the example given in Fig. 9, spatial relationship edges “at” are inserted between the corresponding corners of components that visit the same node in the ASG. Thereafter, the reducer and the parser are invoked again, so that finally the layouter can arrange the new components within the actual diagram.

Discussion of constraint-based layout for diagram completion

Constraints generally are known to be well-suited for the description of layout on a rather high level of abstraction. However, in our domain, an additional benefit becomes noticeable: The constraints normally used for diagram beautification can be directly used to embed the completions, i.e., no extra specification effort is necessary for this purpose. While parsing, layout constraints for the completions are raised automatically. Thus, a solution of these constraints directly yields a proper embedding of the new diagram components.

The constraint solver QOCA [MC02] appeared particularly suited for our setting. It is based

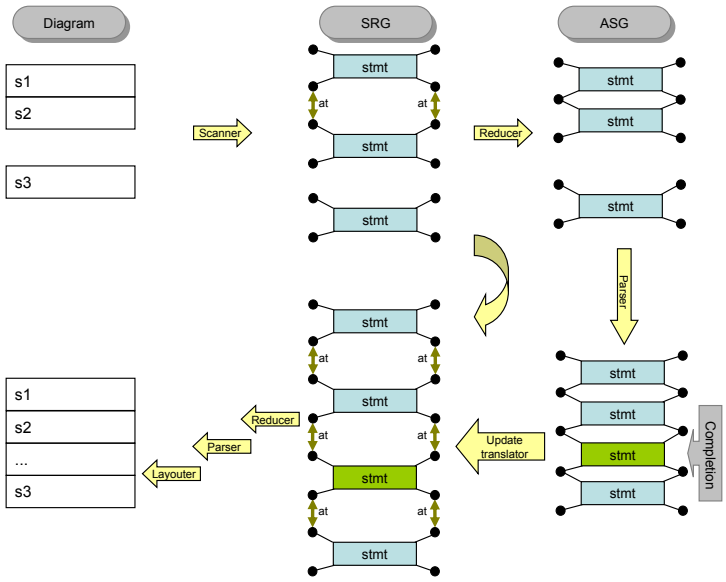

Figure 9: Example process for the embedding of a completion

on the metaphor of the *metric space model*, i.e., it computes incremental solutions with minimal changes according to a given metric. This is very convenient for interactive applications and an important requirement for our approach. Furthermore, a so-called stayweight can be assigned to variables that indicates the importance of leaving the particular variable unchanged. This is especially useful in our domain, since we do not need to guess initial values for the variables related to completions. Rather it is sufficient to just set their stayweight to zero. After embedding the completion we have to increase these stayweights again, of course (otherwise, the affected components will show an unpredictable behavior later).

But there is also one serious issue with constraint-based layout that needs further consideration. In order to find useful places for the new components, there need to be sufficiently many constraints, i.e., not too many degrees of freedom. This, in a sense, restricts the freedom users of a free-hand editor are so fond of. In the future, we want to investigate other mechanisms for a sensible embedding of completions in order to avoid this problem.

5 Benefits

In this section we summarize the key benefits of our approach with respect to the usability of diagram editors.

In [Nie94] Nielsen discusses the most important usability factors. One of these factors is "Flexibility and efficiency of use". It subsumes, among other things, the following important usability heuristics, which we have tackled with our approach:

- Accelerators should be provided,
- Shortcuts: Accelerators to speed up dialogue,

- System should be efficient to use, and
- Keyboard core functions should be supported

Furthermore, we consider the factor "Consistency and standards". Indeed our new editor functionality does always work the same way for all kinds of editors generated with DIAGEN. Thus, editors show a consistent behavior. According to Nielsen, another important usability factor is the support of recognition, understanding and processing of errors; in particular the automatic construction of solutions (as we do) is widely agreed to improve the usability of tools.

We also know, that users want to learn (e.g. visual languages) exploratively. Carroll and Rosson have called this the "paradox of the active user" [CR87]: Users do not read manuals even if they could considerably improve their productivity. Rather they learn tools by playing around with them following the trial and error principle. Language exploration is greatly supported by our approach, because we can use our completion engine for language generation, i.e., the user can generate and inspect all possible (correct) diagrams up to a particular size. This feature (together with the correction of the user's incorrect diagrams) significantly reduces the time needed for learning a particular visual language.

Next, we demonstrate the usability improvement with a concrete example. Consider the shortcut for the application of the best completion again, which has been motivated in Fig. 4. The increase in editing productivity possible by introducing such a shortcut is stated more precisely in Fig. 10. There, we compare both approaches to correct the diagram of Fig. 4 by using the GOMS method [JK96]. In the first column of the figure the primitive operations are listed that are necessary when performing the changes manually. The second column shows the operations needed when using a shortcut that automatically applies the best completion. Following [JK96] we use the following times to compute the overall task performance:

mental act of routine thinking (M)	1.2s
point with mouse to a target (P)	1.1s
press or release mouse button (B)	0.1s
keystroke (K)	0.28s
home hands to keyboard or mouse (H)	0.4s
waiting for the system to respond (W(t))	t

Therewith, we get an overall execution time of 8.4s for the first way. This is even quite optimistic. We have assumed that the two statements have the same height. Furthermore, we have not considered editing errors, which are likely to occur when performing such sensitive mouse actions. For the second way – the shortcut – we get a task performance of 4.06s: more than twice as fast. Here, we have estimated that it takes 1.5s to compute and rank the completions.[5]

When discussing usability, last but not least, the factor "joy of use" should not be underestimated. We are convinced that the feature diagram completion is fun to use and, thus, hopefully increases the usage and acceptance of our tool.

[5] This is a realistic assumption for small and medium-sized diagrams. In [MMM08a] we have provided some performance data for our parser with completion support. For instance, completions of size two for an NSD of size 20 can still be computed in less than a second. Our parser has not even been optimized with respect to performance yet. Prelayouting the resulting completions (provided their number is not too big) can also be done quite efficiently. Furthermore, this is a task that can be parallelized easily.

dragging components explicitly:

- think (M)
- point to move handle of statement 1 (P)
- press and hold mouse button (B)
- drag component such that its left upper corner touches left lower corner of condition component (P)
- release mouse button (B)
- point to move handle of statement 2 (P)
- press and hold mouse button (B)
- drag component such that its left upper corner touches right upper corner of statement 1 (P)
- release mouse button (B)
- point to resize handle of condition (P)
- press and hold mouse button (B)
- drag handle such that the right lower corner of condition touches the right upper corner of statement 2 (P)
- release mouse button (B)

shortcut for correction:

- think (M)
- move hand to keyboard (H)
- hit command key (K+K)
- wait for result to be computed (W(1.5s))
- move hand back to mouse (H)

Figure 10: Comparison of explicit dragging and shortcut

6 Concluding Remarks

In this paper we have described an approach on how to assess and sort diagram completions. The problem we have addressed is practically relevant, in particular when compared to the incredible success of content assist in textual environments, see e.g. [GS04, NJ07, BW06]. Even for simple diagrams the number of completions increases rapidly with their maximal possible size. However, choosing from a high number of completions while editing causes too much cognitive overload for users. Such time-consuming interruptions generally should be avoided. We can help the user in this respect by providing a shortcut to a distinguished completion: the one causing minimal changes to his original diagram.

We have proposed to exploit the layout engine for this purpose. In fact, we precompute all possible target diagrams and the corresponding changes to the existing components. Different metrics can be used to actually compare the resulting diagram completions. Thanks to the layouter we get nice-looking results that meet the user's expectation very well. The resulting editors effectively combine the advantages of free-hand and structured editing: The user can draw his diagrams with maximal freedom, but he also can ask for assistance at any time.

In the future we have to conduct an extensive user study. As already argued in this paper, we strongly expect further productivity gains for experts. But we also hope to find more evidence on how learning of visual languages actually is improved. We plan to extend our approach to languages that are not context-free (DIAGEN's *embedding productions*). Finally, we will use our approach to compute situation-dependent, correctness-preserving structured editing operations.

Bibliography

[BKC00] B. Bailey, J. Konstan, J. Carlis. Measuring the effects of interruptions on task performance in the user interface. *Systems, Man, and Cybernetics, 2000 IEEE International Conference on* 2:757–762 vol.2, 2000.

[Bra01] J. Branke. Dynamic graph drawing. In *Drawing graphs: methods and models*. Pp. 228–246. Springer-Verlag, London, UK, 2001.

[BT98] S. S. Bridgeman, R. Tamassia. Difference Metrics for Interactive Orthogonal Graph Drawing Algorithms. In *GD '98: Proceedings of the 6th International Symposium on Graph Drawing*. Pp. 57–71. Springer-Verlag, London, UK, 1998.

[BW06] H. Bast, I. Weber. Type less, find more: fast autocompletion search with a succinct index. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*. Pp. 364–371. ACM, New York, NY, USA, 2006.

[CR87] J. M. Carroll, M. B. Rosson. Paradox of the active user. In *Interfacing thought: cognitive aspects of human-computer interaction*. Pp. 80–111. MIT Press, Cambridge, MA, USA, 1987.

[DHK97] F. Drewes, A. Habel, H.-J. Kreowski. Hyperedge Replacement Graph Grammars. In Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations*. Chapter 2, pp. 95–162. World Scientific, 1997.

[ELMS91] P. Eades, W. Lai, K. Misue, K. Sugiyama. Preserving the Mental Map of a Diagram. Technical report, FUJITSU LABORATORIES, 1991.

[GS04] K. Grabski, T. Scheffer. Sentence completion. In *SIGIR '04: Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*. Pp. 433–439. ACM, New York, NY, USA, 2004.

[HCW07] A. Hessellund, K. Czarnecki, A. Wasowski. Guided Development with Multiple Domain-Specific Languages. In Engels et al. (eds.), *MoDELS*. Lecture Notes in Computer Science 4735, pp. 46–60. Springer, 2007.

[JK96] B. E. John, D. E. Kieras. The GOMS family of user interface analysis techniques: comparison and contrast. *ACM Trans. Comput.-Hum. Interact.* 3(4):320–351, 1996.

[Kra87] E. Krause. *Taxicab Geometry: An Adventure in Non-Euclidean Geometry*. Dover, 1987.

[LVA04] J. de Lara, H. Vangheluwe, M. Alfonseca. Meta-Modelling and Graph Grammars for Multi-Paradigm Modelling in AToM^3. *Journal on Software and Systems Modelling*, pp. 193–209, 2004.

[MC02] K. Marriott, S. S. Chok. QOCA: A Constraint Solving Toolkit for Interactive Graphical Applications. *Constraints* 7(3-4):229–254, 2002.

[Min00] M. Minas. Hypergraphs as a Uniform Diagram Representation Model. In *TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*. Pp. 281–295. Springer-Verlag, London, UK, 2000.

[Min02] M. Minas. Concepts and Realization of a Diagram Editor Generator Based on Hypergraph Transformation. *Science of Computer Programming* 44(2):157–180, 2002.

[Min06] M. Minas. Syntax analysis for diagram editors: a constraint satisfaction problem. In *AVI '06: Proceedings of the working conference on Advanced visual interfaces*. Pp. 167–170. ACM, New York, NY, USA, 2006.

[MMM08a] S. Mazanek, S. Maier, M. Minas. An Algorithm for Hypergraph Completion according to Hyperedge Replacement Grammars. In *Proc. of the 4th Intl. Conference on Graph Transformation*. LNCS. Springer, 2008.

[MMM08b] S. Mazanek, S. Maier, M. Minas. Auto-completion for Diagram Editors based on Graph Grammars. In *Proc. of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE Computer Society Press, 2008.

[MV93] M. Minas, G. Viehstaedt. Specification of diagram editors providing layout adjustment with minimal change. *Visual Languages, 1993., Proceedings 1993 IEEE Symposium on*, pp. 324–329, Aug 1993.

[Nie94] J. Nielsen. Enhancing the explanatory power of usability heuristics. In *CHI '94: Proceedings of the SIGCHI conference on Human factors in computing systems*. Pp. 152–158. ACM, New York, NY, USA, 1994.

[NJ07] A. Nandi, H. V. Jagadish. Effective phrase prediction. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*. Pp. 219–230. VLDB Endowment, 2007.

[vP07] J. von Pilgrim. Mental Map and Model Driven Development. In *Proc. of the Workshop on the Layout of (Software) Engineering Diagrams (LED 2007)*. 2007.

[RS96] J. Rekers, A. Schurr. A graph based framework for the implementation of visual environments. *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pp. 148–155, Sep 1996.

[SBV07] S. Sen, B. Baudry, H. Vangheluwe. Domain-specific model editors with model completion. In *Multi-paradigm Modelling Workshop at MoDELS 2007*. 2007.

[WSNW08] J. White, D. C. Schmidt, A. Nechypurenko, E. Wuchner. Model Intelligence: an Approach to Modeling Guidance. *UPGRADE* 9(2):22–28, 2008.