Electronic Communications of the EASST
Volume 18 (2009)

Proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009)

Generating Correctness-Preserving Editing Operations for Diagram Editors

Steffen Mazanek, Mark Minas

12 pages

Guest Editors: Artur Boronat, Reiko Heckel
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Generating Correctness-Preserving Editing Operations for Diagram Editors

Steffen Mazanek[1], **Mark Minas**[2]

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

Abstract: In previous work it has already been shown that syntax-directed and free-hand editing can be gainfully integrated into a single diagram editor. That way, the user can arrange diagram components on the screen without any restrictions in free-hand editing mode, whereas syntax-directed editing operations provide powerful assistance. So far, editing operations had to be specified or programmed by the editor developer. In contrast, this paper proposes an approach where diagram-specific editing operations are generated on the fly during the editing process and without any additional specification effort. These operations provably preserve the correctness of the diagram. The proposed approach requires a specification of the visual language by a hypergraph grammar.

Keywords: syntax-directed editing operations, diagram editors, correctness preservation, hypergraph grammars

1 Introduction

Generally, two kinds of diagram editors are distinguished: A structure editor offers the user operations that transform correct diagrams into (other) correct diagrams. Users like this kind of guidance, because that makes editing much easier. But they also like to draw their diagrams freely just following the flow of their uninhibited associations. A free-hand editor allows them to arrange diagram components on the screen without any restrictions. Using syntax analysis, the free-hand editor decides whether the drawing conforms to the visual language and what structure the user intended to define.

A method for the combination of free-hand and syntax-directed editing has already been proposed and realized previously [Min02]. Thereby, free-hand editing is supported by a hypergraph parser [Min97], whereas syntax-directed editing is realized using hypergraph transformation [KM00]. That way, a wide range of operations can be implemented. For instance, operations for diagram execution (like firing of a transition in a petri net or processing the input of a finite state machine) can be defined. But one can also define more local operations, i.e. syntax-directed editing operations in the narrower sense, which require the user to select certain diagram components in advance (like adding a token to the selected place or connecting two selected states with an arrow). These are the operations considered in the following. Concerning such operations, the approach of [Min02] still has some weak points:

- Specifying syntax-directed editing operations is a tedious task: For each operation an additional hypergraph transformation rule has to be defined.
- Specifying *correct* syntax-directed editing operations is difficult: The editor developer has to ensure that his operations do not destroy correct diagrams of the user, i.e. all operations have to comply with the grammar.
- Specifying all possible syntax-directed editing operations is infeasible: In fact, there might even be infinitely many possible editing operations – at least if the number of diagram components that can be inserted at one go is not restricted. The editor developer might not know which of these operations the users actually need.

The approach presented in this paper offers solutions for these three weak points. Users still can draw their diagrams with maximal freedom. At the same time, they have access to powerful syntactical assistance whenever required. The provided assistance provably cannot do harm, i.e. the correctness of the user's diagram is preserved. These benefits come for free, i.e., the editor developer does not need to define these syntax-directed editing operations anymore.

Regarding interaction, the editor user has to select the diagram components in whose context additional components are to be inserted. On request, meaningful editing operations are computed as follows: First, the selection-induced part of the diagram's hypergraph representation is separated. The resulting hypergraph is analyzed by an error-correcting parser, which tries to complete it again by adding hyperedges and gluing nodes. Only those completions are presented to the user as editing operations that meet some language-independent relevance criteria.

Outline: The following sections 2 and 3 briefly introduce the running example and previous work. Sect. 4 then presents the main result of this paper, i.e., how syntax-directed editing operations can be generated on demand during the editing process. The proposed solution is discussed in Sect. 5. Related work is reviewed in Sect. 6, and Sect. 7 concludes the paper.

2 Running Example

Throughout this paper, the simple visual language of Nassi-Shneiderman Diagrams (NSDs) is used as a running example. However, the approach also has been applied successfully to several other visual languages. Its overall applicability is discussed in Sect. 5.

Fig. 1 shows an example NSD and a corresponding Abstract Syntax (Hyper-)Graph (ASG). Hyperedges are represented as boxes with the particular label inside. Nodes are represented as black dots. Lines indicate that a hyperedge visits a node. The correspondence between the diagram and the hypergraph is obvious: Components are mapped to hyperedges. Each corner of an NSD component is represented by an attachment node of the corresponding hyperedge. Hyperedges visit the same node if the respective corners of their components touch each other.

The hypergraph language of NSDs can be recursively defined using a hyperedge replacement grammar (HRG) [DHK97] as shown in Fig. 2. A BNF-like notation is used here. Nonterminal symbols are highlighted. The first two rules (i.e., the upper row) basically state that an NSD is a non-empty chain of successive statements. A statement in turn either is a primitive statement, a condition, a while or an until loop. The body of a loop and the branches after a condition have to be NSDs again.

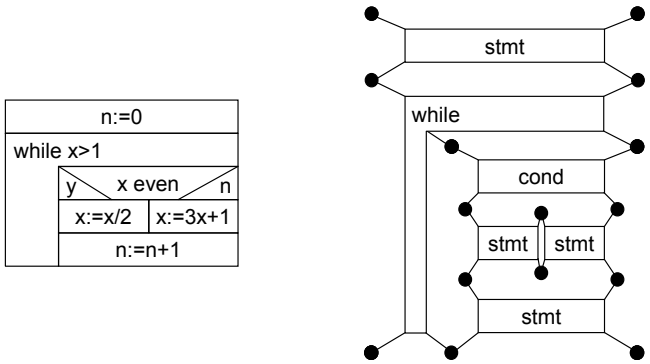

Figure 1: Example NSD and corresponding abstract syntax hypergraph

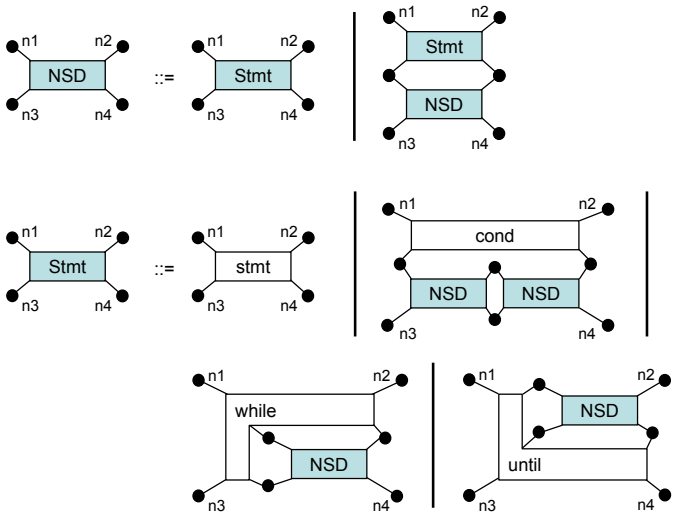

Figure 2: Hyperedge replacement grammar of NSDs

Syntax-directed editing operations are very handy in order to manipulate NSDs. For instance, the user might want to insert a statement `write x` right before the statement `n:=n+1` in the NSD shown in Fig. 1. This task cannot be performed conveniently in free-hand mode, because a lot of editing is required to make room for the new statement. In this situation, a syntax-directed editing operation would be preferable. That way, the user could just select the statement `n:=n+1` and call the operation "insert statement before".

3 Basic Formalism and Previous Work

Formally, a hypergraph $H = (V_H, E_H, att_H, lab_H)$ over a set C of labels consists of a set V_H of nodes, a set E_H of hyperedges, a mapping $att_H : E_H \to V_H^*$ that assigns a sequence of attachment nodes to the hyperedges of H, and a labeling function $lab_H : E_H \to C$ for the hyperedges of H. The HRG formalism as used in the following is extensively described in [DHK97].

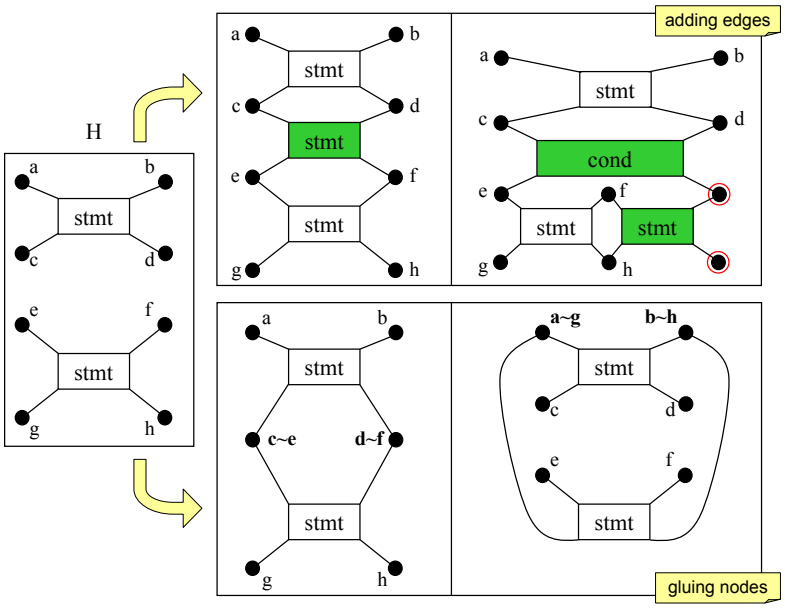

Figure 3: Application of hypergraph patches

Furthermore, this work relies on [MMM08a], where an algorithm for *hypergraph completion* with respect to HRGs has been proposed. Given a hypergraph H and an HRG G, this algorithm embeds additional hyperedges into H in a way, such that the resulting hypergraph H' is a member of the language defined by G. Besides hyperedges it might also introduce some fresh nodes (incident to these hyperedges). This is shown in Fig. 3 (top). The fresh nodes are highlighted by an extra circle. Since there might be an infinite number of possible completions, their size (i.e., the number of additional hyperedges) has to be restricted.

This algorithm has been extended recently, so that it (optionally) can also glue existing nodes where required. This is also shown in Fig. 3 (bottom). As expectable, the two isolated statements of the given hypergraph H can be combined in two different ways (orders). The extended parsing algorithm returns so-called *hypergraph patches* as a result. Formally, a patch $P_H = (\sim, V, E, att, lab)$ for a hypergraph H consists of an equivalence relation $\sim \subseteq V_H \times V_H$ on the nodes of H, a set V of additional nodes, and a set E of additional hyperedges with corresponding attachment and labeling functions. Applying a patch then basically means to embed the additional hyperedges (yields H') and to construct the quotient hypergraph $H'/\sim$, i.e. a hypergraph whose nodes actually are equivalence classes of the nodes of H'. Note that all patches computed by the parser can be used to transform the given hypergraph into a correct one.

Since hypergraphs have appeared to be well-suited as a model for diagrams [Min02], hypergraph patches can be naturally used in diagram editors. In this manner the DIAGEN toolkit has been extended to support diagram correction and completion [MMM08b].

The conventional DIAGEN editing process (as marked in Fig. 4) consists of several steps [Min02]: The modeler first creates a so-called Spatial Relationship (Hyper-)Graph (SRG) corresponding to the diagram. Thereafter, the reducer simplifies the SRG (similar to lexical analysis

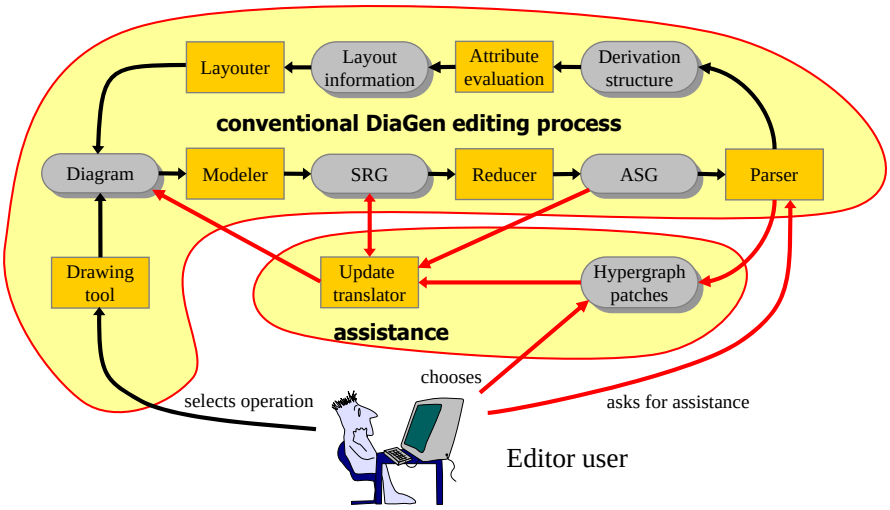

Figure 4: Extended DIAGEN editing process

in the string setting). This results in an abstract representation of the diagram, the ASG. The parser analyzes the ASG and constructs the derivation structure (if any). Finally, the layouter computes a layout for the diagram (using derivation information if required).

In [MMM08b] this process has been extended as follows (see also Fig. 4): If a user explicitly asks for assistance, the parser is triggered with the desired size of completions as a parameter. It computes all possible hypergraph patches up to this size [MMM08a]. From those, the user has to choose. Next, the selected patch is applied and embedded into the SRG using a language-specific update translator component. The editor then calls the reducer and parser again, so that the layouter can arrange the new components within the actual diagram and adapt existing components if necessary.

4 Generating Syntax-Directed Editing Operations

Since an editing operation might be applicable to several parts of a diagram, the user normally has to select the context in which additional components are to be inserted. For instance, the already mentioned operation “insert statement before” requires the selection of the statement where a new statement should be inserted before. If operations are predefined by the editor developer it is easy to specify as a precondition what has to be selected by the user. This approach cannot be used if operations are to be generated at runtime. However, for a generic approach the user’s selection can be interpreted as follows: A selection should induce editing operations that separate the user-selected diagram part, add to it new diagram components, and finally paste the extended diagram part back into the remaining diagram such that it is correct again.

Fig. 5 shows some example operations following this idea. On the left-hand side four example NSDs are given. The components selected by the user are surrounded by a thick border. To the right of the arrows, the figure shows extended diagrams resulting from the application of such editing operations to the input diagram. All new diagram components are highlighted. The

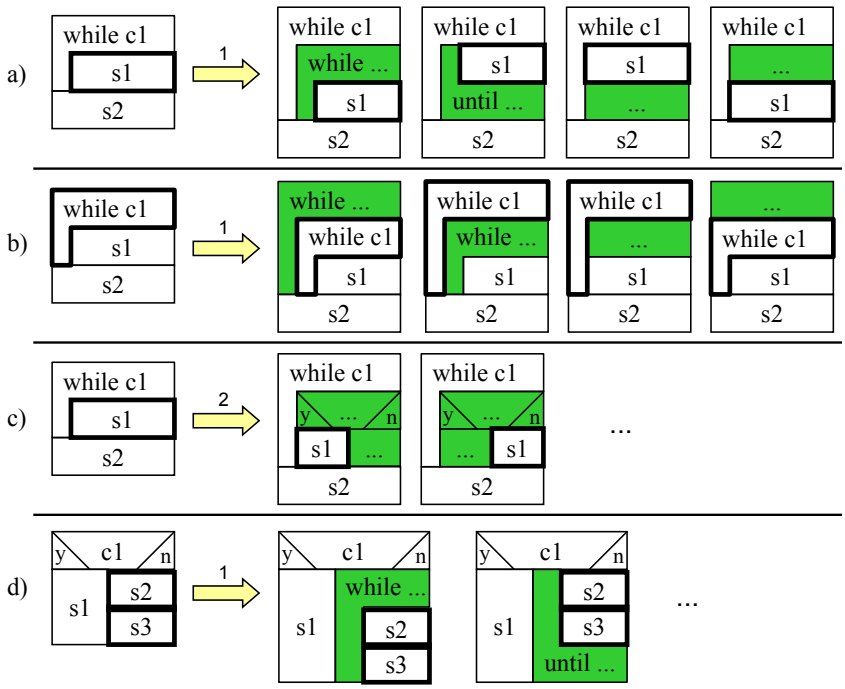

Figure 5: Example editing operations

numbers above the arrows indicate the size of the operations, i.e. the number of introduced components. Note that all shown operations preserve the correctness of the respective input NSD. All of them can be generated following the approach presented next.

In Fig. 5a the statement s1 is selected and one new component should be inserted. In this case four different operations of size 1 are evident: s1 could either be enclosed by a *while* or *until* loop or, alternatively, a primitive statement could be inserted below or above. In Fig. 5b, a *while* component is selected. Four operations of size 1 are evident: Another *while* component could be inserted outside or within the selected one. Alternatively a primitive statement could be inserted within or above the selected *while*.

The selection in Fig. 5c is equal to Fig. 5a. However, this time operations of size 2 are requested. Two useful operations are shown. Actually, both cannot be simulated by just repetitive application of operations of size 1, because intermediate results are required to be correct. This means, each operation has to yield a correct diagram. Note that many more reasonable operations of size 2 exist. However, in contrast to the insertion of a *cond*, these could also be constructed successively. The last row, Fig. 5d, demonstrates that sometimes it is even necessary to allow the selection of several diagram components at once. Otherwise, it would be quite unintuitive to insert a *while* or an *until* component around a correct sub-NSD (here just two successive statements). Again there are some more solutions, but those can already be realized by selecting just one of the existing components and, hence, have been omitted.

In order to generate such syntax-directed editing operations the visual language's hypergraph grammar can be exploited. The basic idea is to reuse the patch-computing parsing algorithm

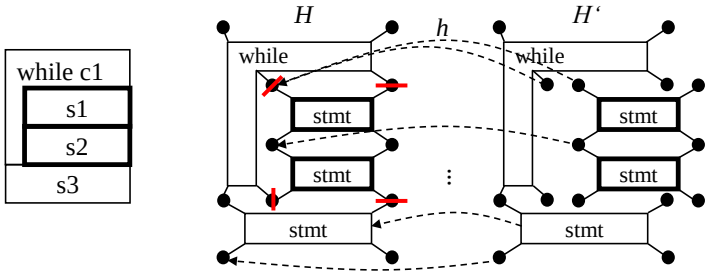

Figure 6: Separation of selected components

described in Sect. 3. The intuition of a user selection on the diagram level has already been described. On the level of a diagram's ASG, the separation of the selection means breaking up the ASG into two disjoint hypergraphs H_1 and H_2 where H_2 corresponds to the user-selected diagram part. Breaking up the ASG generally means splitting up some of its nodes (cf. Fig. 6). Adding new diagram components and re-merging the diagram parts just means to find and apply a hypergraph patch using the disjoint union of H_1 and H_2 as input. However, not every hypergraph patch constitutes a meaningful editing operation. Language-independent *relevance criteria* can be used to discard hypergraph patches that are inappropriate as editing operations.

Next, the generation of editing operations is described more formally and the relevance criteria are defined. Let H be the ASG of the diagram and $E_s \subseteq E_H$ the set of "selected" hyperedges of H. Let H_s and $H_{\bar{s}}$ be the sub-hypergraphs of H induced by the sets E_s resp. $E_H \setminus E_s$ of hyperedges. Finally, let H' be the disjoint union of H_s and $H_{\bar{s}}$. H' differs from H, because nodes in H being visited by selected as well as non-selected hyperedges are "split" in H'. The epimorphism h maps H' to H as shown in Fig. 6. Let V_{split} be the split nodes of H', i.e., those nodes that are merged by h (in Fig. 6 there are 8 split nodes).

H' is incorrect in general, so that the application of the patch-computing parser normally yields a wide range of solutions. Not all of them form meaningful editing operations though. This issue is illustrated in Fig. 7 using the diagram of Fig. 5a as an example. The trivial patch of size 0, which just glues the separated statement back to its original position, is omitted in Fig. 7. Rather this figure shows all patches of size 1 and their resulting hypergraphs. However, only 4 of these 10 patches constitute meaningful editing operations; the other 6 are crossed out. They are not meaningful since either the selected statement has been glued back to its original position and a new component has been added at a remote position, or the selected statement has been moved to a remote position and its original position has been "filled" by a new component. Such a behavior is not meaningful for NSDs or any other diagram language. This observation motivates the definition of *relevant* hypergraph patches that describe meaningful editing operations independent of the specific diagram language:

A patch $P_{H'} = (\sim, V_P, E_P, att_P, lab_P)$ is *relevant* if and only if

1. $\forall e \in E_P : sequenceToSet(att_P(e)) \subseteq V_{split} \cup V_P$, i.e., additional hyperedges do not visit any nodes that are not related to the selection, and

2. $\forall n_1, n_2 \in V_{H'} : n_1 \sim n_2 \Rightarrow h(n_1) = h(n_2)$, i.e., only the split nodes can be glued, but only to the respective nodes they have been separated from.

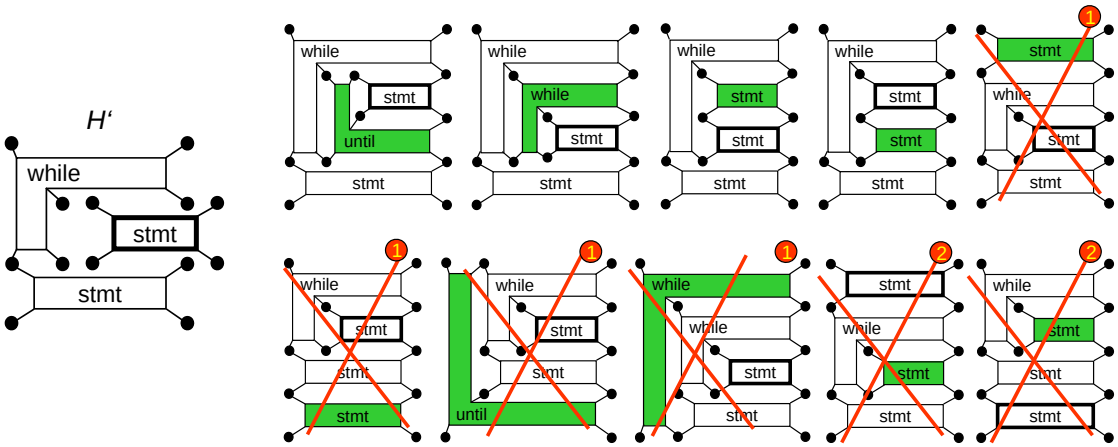

Figure 7: Relevance criteria, input diagram Fig. 5a

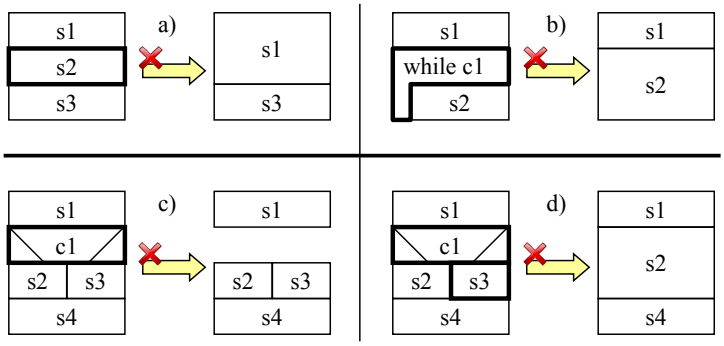

Figure 8: Intelligent component removal

Note that Fig. 7 also shows which criterion excludes a particular patch. This is indicated by numbers in the upper right corners of the resulting hypergraphs.

Intelligent Remove and Replace

Syntax-directed editing operations generated by the previous process do not delete any diagram components. However, generating *intelligent remove* operations (and replacement similarly) is also straightforward. Conventional removal of one or more components does not modify the remaining diagram, which may become invalid after the removal. In contrast, *intelligent remove* adjusts the remaining diagram so that it becomes valid again. It is performed in two steps: First, a conventional remove of the selected components is performed. Subsequently, corrections are computed. Again, not each correction of the remaining diagram is meaningful. A relevant hypergraph patch here is neither allowed to add new hyperedges, nor is it allowed to glue remote nodes, i.e., nodes that have not been visited by the hyperedges deleted previously.

Fig. 8 shows four examples of intelligent removal. In Fig. 8a the middle *stmt* in a chain of three successive *stmt*s is selected. Intelligent removal deletes this *stmt* and glues the other *stmt*s

together preserving their order. In Fig. 8b, a *stmt* s1 is followed by a *while* component that encloses another *stmt*, s2. Intelligently removing the *while* component glues the *stmt*s s1 and s2 together (again preserving their order). Fig. 8c shows an example where intelligent removal cannot help. In this situation, intelligent remove corresponds to conventional remove. Indeed, an NSD cannot be kept correct if just a *cond* component is to be removed. However, if one of the branches is selected, too, intelligent removal works as desirable, cf. Fig. 8d.

All in all, intelligent removal is a useful function for such visual languages where the removal of components is likely to yield incorrect diagrams. Otherwise, it just converges to the conventional remove function, i.e. it simply removes the selected components.

5 Discussion

Following the presented approach, the user can quickly access local editing operations. In most cases, selecting just a single component is sufficient already. In fact, the selection of several components usually makes sense only if they "share nodes". This behavior most likely conforms to the user's intuition. Nevertheless, there are still some challenges that are addressed next.

Performance: It is important to stress that the presented approach does not generate generic operations at "compile time". The generated operations are rather specific to the current diagram and completely generated on the fly. As a consequence, each time the user asks for operations the whole diagram needs to be analyzed again. This additional effort is not necessary if the operations are predefined. However, a basic requirement for free-hand editors is an efficient parser, since the diagram has to be reanalyzed after every single modification. The current implementation of the patch-generating parser is a prototype with reasonable speed. For instance, the computation of operations of size 1 for an NSD of size 20 takes less than a second on standard hardware. Further performance improvements are subject of current work.

Information Overload: When increasing the possible size of operations their number might explode. The problem is that the user can hardly distinguish between the really new solutions and the solutions that he could also get by successively applying smaller operations. It might be useful to apply a special filter to avoid this issue. But this has not been realized yet.

Understandability: In certain (rare) situations knowledge of the abstract syntax seems to be necessary to understand why a particular operation currently is not possible. For instance, if a *cond* is selected, a *stmt* can only be inserted above, but not in the branches below. However, we have not found a meaningful operation yet that cannot be generated at all. The user just needs to select the "right" components (in the example, the first statement in the respective branch).

Applicability: Since the patch-computing hypergraph parser relies on context-free HRGs, the applicability of this approach naturally is restricted. However, the approach can also be applied to hypergraph grammars that contain so-called *embedding rules* [Min02]. Indeed, all practically relevant visual languages can be described that way. Operations then are only computed for the context-free part of the particular diagram. So, if a language exhibits a significant context-free core, the presented approach can still be used. For instance, it has been applied to sequence diagrams where only the messages need to be embedded. The generated syntax-directed editing operations have appeared to be helpful even for this non-context-free diagram language. In addition, the approach has been applied to other examples like flowcharts, logic gates, and trees.

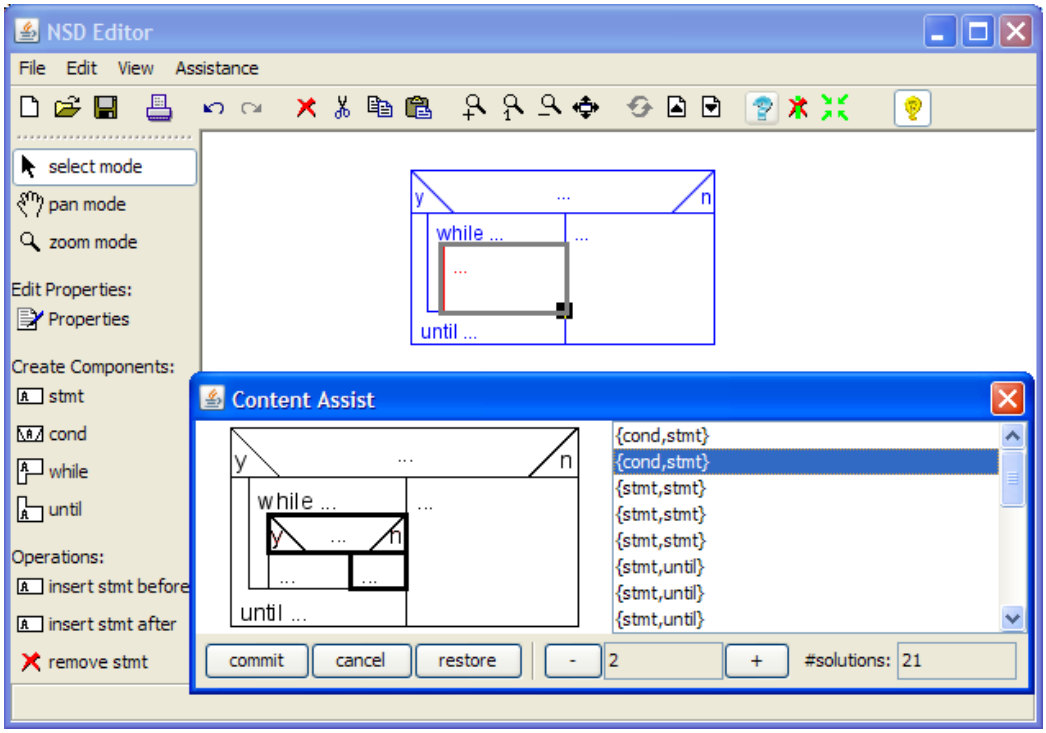

Figure 9: Screenshot of the user interface

The proposed approach has been integrated into the DIAGEN toolkit. Fig. 9 shows a screenshot of a DIAGEN editor for NSDs. The user has selected a primitive statement and opened the assistance dialog. Here, he can set up the size of operations he is interested in using the buttons at the bottom. There are 21 operations of size 2 already as can be seen in the lower right corner. Selecting one of them updates the preview pane on the left-hand side. Both the application of the operation and refocusing of the diagram are animated, so that the user can easily see what is going on. Committing finally applies the selected operation to the actual diagram. For comparison, the shown editor also provides some operations that have been explicitly specified by transformation rules, among others the example operation “insert stmt before”.

6 Related Work

In TIGER [EEHT05] editing operations are specified by means of graph transformation rules. These operations directly define the language, so that the editor developer does not have to ensure that they comply with a grammar. Unfortunately, TIGER does not support free-hand editing. The predecessor of TIGER, GENGED [BST01], had supported some kind of free-hand editing. It even had generated some initial editing operations from the type graph at compile time. Those, however, only allowed the insertion of nodes and edges in graph-like languages. Recently, the TIGER developers have extended the popular Eclipse GMF framework with support for complex editing commands [TCSE08] – at the price of additional specification effort though.

CIDER [JMM04] supports both free-hand and syntax-directed editing. Its transformation mechanism is fully integrated with an incremental parser. Thus, transformations can be defined in terms of high-level diagram components. Here, the basic idea behind a transformation is to change the parse forest of a diagram from one valid state to a different valid state. This is similar to the conventional DIAGEN approach to syntax-directed editing, where information from the derivation can also be accessed. In CIDER all transformations have to be specified manually.

The grammar-based system VLDESK [CDPR05] provides support for so-called symbol prompting. Here, the parsing table is exploited to extract information about possible contexts of a particular symbol. That way, local suggestions can be computed without additional specification effort (similar to our approach). This kind of assistance is efficient and permissive, but does not ensure the correctness of the resulting diagram from an overall perspective.

For widely used and highly relevant languages specific tool support is still implemented by hand. For instance, Gschwind et al. have proposed a set of powerful operations on business process models [GKW08]. Their approach has been realized as a plugin for the well-known WebSphere Business Modeler. As long as generic assistance mechanisms are not powerful enough this is a reasonable approach to improve the usability of modeling tools. Our approach, where applicable, can help to reduce the burden of implementing language-specific syntactical assistance. Business process models are context-free to a large extent, so that our approach is applicable.

7 Conclusion

The approach proposed in this paper can be used to generate powerful, correctness-preserving editing operations for free-hand editors and, at the same time, is easy to apply and understand by users: They just have to select one (or more) components and ask for assistance.

The editor user can trust the generated operations, because they provably cannot do harm. He can use an animated preview to inspect all possible operations of a particular size. Thereby, he is likely to get new insights into the visual language at hand. Although the improvement of user support has been our primary goal, the burden for the editor developer is also significantly reduced. He does not need to specify certain editing operations in compliance with the grammar anymore. So, he can focus on special-purpose operations and diagram execution. Since our approach is just complementary, the previous flexibility of DIAGEN operations is fully preserved.

In the future we will try to relax the precondition “correctness of the input diagram” that has been assumed throughout this paper. Indeed operations would be also very useful for correct sub-diagrams. Furthermore, we want to improve the support for the non-context-free parts of languages.

Screencasts of the NSD example editor and further examples can be found at the website http://www.unibw.de/inf2/DiaGen/assistance/. The editor can be downloaded from there, too.

Bibliography

[BST01] R. Bardohl, T. Schultzke, G. Taentzer. Visual language parsing in GenGEd. *Electronic Notes in Theoretical Computer Science* 50(3):289 – 294, 2001.

[CDPR05] G. Costagliola, V. Deufemia, G. Polese, M. Risi. Building syntax-aware editors for visual languages. *Journal of Visual Languages and Computing* 16(6):508–540, 2005.

[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.

[EEHT05] K. Ehrig, C. Ermel, S. Hänsgen, G. Taentzer. Generation of visual editors as Eclipse plug-ins. In *ASE '05: Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering*. Pp. 134–143. ACM, New York, NY, USA, 2005.

[GKW08] T. Gschwind, J. Koehler, J. Wong. Applying patterns during business process modeling. In *BPM '08: Proceedings of the 6th International Conference on Business Process Management*. LNCS 5240, pp. 4–19. Springer, 2008.

[JMM04] A. R. Jansen, K. Marriott, B. Meyer. Cider: A component-based toolkit for creating smart diagram environments. In *Diagrams*. LNCS 2980, pp. 415–419. Springer, 2004.

[KM00] O. Köth, M. Minas. Generating diagram editors providing free-hand editing as well as syntax-directed editing. In Ehrig and Taentzer (eds.), *Proceedings of the Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems*. Technical Report 2000-2, pp. 32–39. Technical University, Berlin, 2000.

[Min97] M. Minas. Diagram editing with hypergraph parser support. In *Proceedings of the 1997 IEEE Symposium on Visual Languages (VL '97)*. Pp. 230–237. IEEE Computer Society, Washington, DC, USA, 1997.

[Min02] M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming* 44(2):157–180, 2002.

[MMM08a] S. Mazanek, S. Maier, M. Minas. An algorithm for hypergraph completion according to hyperedge replacement grammars. In Ehrig et al. (eds.), *Proceedings of the 4th International Conference on Graph Transformations*. LNCS 5214, pp. 39–53. Springer, 2008.

[MMM08b] S. Mazanek, S. Maier, M. Minas. Auto-completion for diagram editors based on graph grammars. In Bottoni et al. (eds.), *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. Pp. 242–245. IEEE, 2008.

[TCSE08] G. Taentzer, A. Crema, R. Schmutzler, C. Ermel. Generating domain-specific model editors with complex editing commands. In Schürr et al. (eds.), *Proc. Third International Symposium on Applications of Graph Transformation with Industrial Relevance (AGTIVE 2007)*. LNCS 5088, pp. 98–103. Springer, 2008.