Electronic Communications of the EASST
Volume 32 (2010)

Proceedings of the
Fourth International Workshop on
Graph-Based Tools
(GraBaTs 2010)

Sketch-based Diagram Editors with User Assistance based on Graph Transformation and Graph Drawing Techniques

Steffen Mazanek, Christian Rutetzki, and Mark Minas

14 pages

Guest Editors: Juan de Lara, Daniel Varro
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Sketch-based Diagram Editors with User Assistance based on Graph Transformation and Graph Drawing Techniques

Steffen Mazanek, Christian Rutetzki, and Mark Minas

(Steffen.Mazanek, Christian.Rutetzki, Mark.Minas)@unibw.de
Universität der Bundeswehr München, Germany

Abstract: In the last years, tools have emerged that recognize *sketched diagrams* of a particular visual language. That way, the user can draw diagrams with a pen in a natural way and still has available most processing capabilities. But also in the domain of conventional diagram editors, considerable improvements have been achieved. Among other features, powerful *user assistance* like auto-completion has been developed, which guides the user in the construction of correct diagrams. The combination of these two developments, sketching and guidance, is the main contribution of this paper. It not only shows feasibility and usefulness of the integration of user assistance into sketching editors, but also that novel user strategies for identifying and dealing with recognition errors are made possible that way. The proposed approach heavily exploits graph transformation and drawing techniques. It was integrated into a meta-tool, which has been used to generate an editor for business process models that comprises the features described in this paper.

Keywords: sketching, meta-tools, user assistance, graph transformation, graph drawing, process models

1 Introduction

An important benefit of sketch-based diagram editors is that diagrams can be drawn with maximal freedom in a very natural way. With the appearance of powerful and permissive approaches to their subsequent recognition – among others [HD05, CMP05, CDR05] – many advantages of traditional WIMP interfaces (Window, Icon, Menu, Pointer) can be carried over. Most importantly, diagrams, once recognized, can be further processed. However, one feature of state-of-the-art conventional diagram editors, namely user assistance, has not yet been integrated into sketch tools. The user assistance we aim at guides the user in the construction of correct diagrams. Indeed, the only existing attempt in this direction we are aware of is the work on *symbol completion* by Costagliola et al. [CDR07]. This approach helps the user in completing individual symbols (lexical level), but the overall diagram structure (syntactical level) is not at all considered – not even to mention language semantics or pragmatics. Moreover, this approach has not been integrated into a visual environment yet. In this paper we fill this gap by integrating a user assistance component into a sketching meta-tool, i.e., a framework for generating sketch editors from a language specification. We report on the challenges that had to be addressed and how graph transformation and graph drawing techniques have been used for solving them.

Fig. 1 shows a bird's eye view of the proposed approach, i.e., the overall architecture of sketching editors with assistance. The *user*, who is represented by the stickman in the middle, draws *strokes*, which are the basic input of most sketch recognition tools. The *recognizer* transforms these strokes either on-line or on user's request into a *set of diagram components*. Next, a language-specific *analysis* of the diagram is performed, e.g., a syntax check. For the diagram given in Fig. 1, it might be checked that there are no arrows without proper source and target components or that processing components (rectangles) are connected to data structures (ellipses) only. The result of this analysis step is passed back to the user as visual *feedback*.

The novel aspects of this work are surrounded by a dashed line. For the proposed approach, the analysis additionally has to return a *set of suggestions* for the user, e.g., how the diagram can be completed. The user can choose among these suggestions, e.g., by using a preview of the corresponding diagram changes. The selected suggestion then is integrated into the sketch. Therefore, a *translator* component generates the set of corresponding strokes and adds them to the user's sketch. That way, the next analysis cycle will directly consider the applied suggestion.

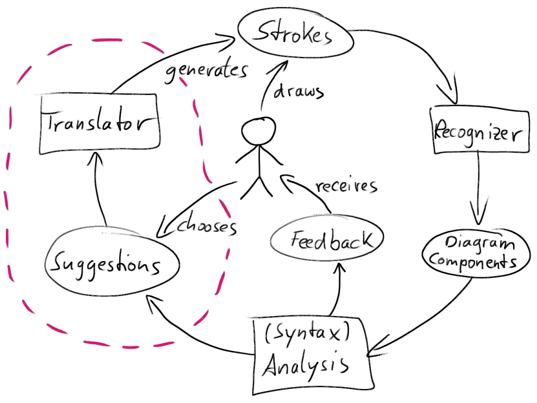

Figure 1: Proposed editor architecture

This paper covers the following assistance features, which are all based on *syntax*:

- *auto-completion*: the computation of missing diagram components that transform the incomplete diagram into a proper member of the underlying visual language,
- *auto-link*: the derivation of missing edges in graph-like languages according to node arrangement and other kinds of editing accelerators,
- *example generation*: the generation of correct example diagrams that can be explored by the user for the sake of language learning.

Suggestions that remove parts of a sketch are not considered.

Sketch tools with powerful recognizers [HD05, CMP05, CDR05] as well as tools for the computation or specification of suggestions [AHHG09, MMM08b, SBV08] already exist. Therefore, this paper focuses on the following three aspects:

- *User interaction*: how can the user invoke and control assistance,
- *Stroke generation*: how and where should the translator generate strokes from the suggestion (this actually is a graph drawing problem),
- *Dealing with recognition errors*: indeed, syntactical assistance not only provides clues for syntactical problems, but also simplifies the identification of recognition errors.

This paper is structured as follows: Sect. 2 introduces the running example language, namely business process models (BPMs). Our implementation relies on existing frameworks for sketch recognition and user assistance; Sect. 3 recapitulates their concepts. How these two approaches actually have been combined and integrated is described in Sect. 4. A discussion is provided in Sect. 5. Finally, related work is reviewed and the paper is concluded (Sect. 6 and 7).

2 Business Process Models

BPMs are used to represent the workflows within an enterprise and, thus, are a highly relevant diagrammatic language today. In recent years a standardized visual notation, the Business Process Modeling Notation BPMN [Obj09], has been developed. Since BPMs are frequently developed in creative team meetings, this language ideally should be supported by sketch editors. Fig. 2 shows a small sales process, which has been drawn and recognized by the sketching editor described in this paper. The magnified (assistance) toolbar will be described later.

BPMs basically are graphs, where the connecting arrows represent sequence flow. The example process starts with the receipt of an order, which is expressed by a start event (circle). Thereafter, the sequence flow is split by an exclusive gateway (diamond shape). If the ordered product is available, it is prepared and shipped, which is expressed by activities (rectangles). Otherwise, a notification is sent to the customer. Thereafter, the sequence flow is joined again by another gateway, and the process terminates as indicated by the end event (circle).

In the following, only well-structured BPMs are treated, i.e., we require splits and joins to be properly nested. This restriction improves the understandability of process models in the same way as structured programming improves the understandability of program code [MRA09]. For well-structured BPMs, moreover, powerful syntactical user assistance is available [MM09].

3 The Frameworks *PerSUADE* and *DSketch*

The general approach proposed in this paper (Fig. 1) is generic as it is not restricted to a particular visual language. Hence, an implementation requires frameworks for sketch recognition as well as syntax analysis and user assistance that are generic as well, i.e., they must be adaptable to different visual languages. Concretely, we have chosen the *DiaGen* approach [Min02] as a base, where hypergraphs are used as a model for diagrams and hypergraph grammars as a means for syntax definition. Accordingly, this formalism is introduced at first. Thereafter, the *PerSUADE* approach, an extension of *DiaGen* by syntax-based user assistance, is introduced. Finally, the sketching approach *DSketch*, which is also based on *DiaGen*, is recapitulated.

3.1 Hypergraphs and Hypergraph Grammars

Hypergraphs are generalized graphs whose edges can connect an arbitrary number of nodes. This notion of graphs allows a uniform representation of all kinds of diagrams. The key idea is that diagram components are represented by hyperedges and their attachment areas by nodes. Fig. 2 also shows the chosen hypergraph representation of the example BPM. The hyperedges are drawn as rectangular boxes and the nodes as black dots. If a hyperedge and a node are incident, they are connected by a line called tentacle. Activities and events have two attachment areas, i.e., incident nodes: one for incoming and one for outgoing sequence flow. Gateways have four attachment areas (namely their corners). Note that sequence arrows do not explicitly occur in this hypergraph representation. They are rather represented implicitly by the fact that connected components visit the same node: the source component via its outgoing tentacle, the target component via its incoming tentacle, respectively. The hypergraph shown in Fig. 2 actually is the result of a lexical analysis step, which performs such simplifications.

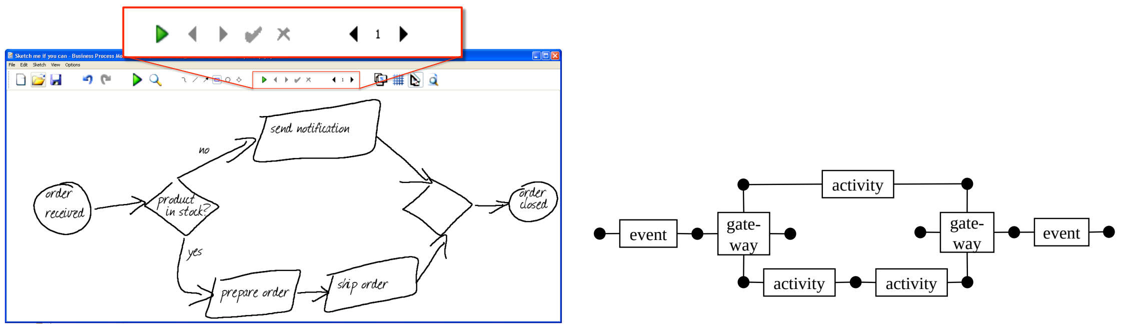

Figure 2: A sketched BPM fragment and its hypergraph representation

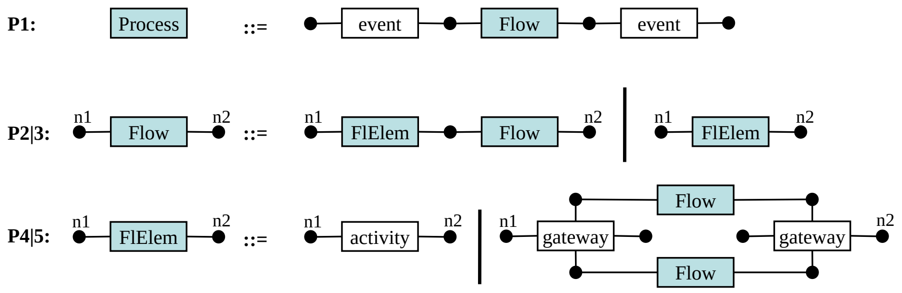

Figure 3: Hypergraph grammar for BPMs

In *DiaGen*, hypergraph grammars are used for language definition. For this paper, only context-free ones are considered [DHK97]. Such hypergraph grammars consist of two finite sets of terminal and nonterminal hyperedge labels and a starting hypergraph that contains only a single nonterminal hyperedge. Syntax is described by a set of productions. The hypergraph language generated by a grammar is defined by the set of terminally labeled hypergraphs that can be derived from the starting hypergraph.

Fig. 3 shows the productions of a hypergraph grammar G_{BPM} for very simple process models. A more comprehensive version that includes pools (process containers), different kinds of intermediate events, and embedded messages has been shown in [MM09]. The types *event*, *activity*, and *gateway* are the terminal hyperedge labels. The set of nonterminal labels consists of *Process*, *Flow*, and *FlElem*. The starting hypergraph consists of just a single *Process* edge. The application of a context-free production removes an occurrence e of the hyperedge on the left-hand side of the production from the host graph and replaces it by the hypergraph H_r on the right-hand side. Matching node labels of both sides of a production determine how H_r has to fit in after removing e. Fig. 4 shows an example derivation.

3.2 User Assistance with *PerSUADE*

For visual languages defined by hypergraph grammars, *hypergraph patches* have been proposed as a means for the realization of Syntax-based User Assistance in Diagram Editors (*PerSUADE*) [MMM08b]. A patch basically describes a modification of a given hypergraph H that transforms

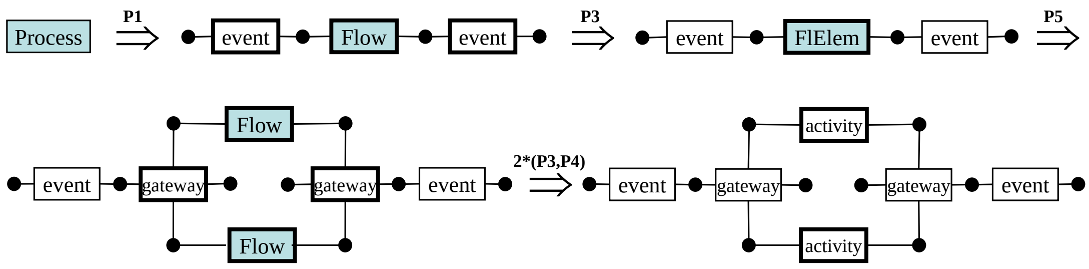

Figure 4: Example derivation starting from *Process*

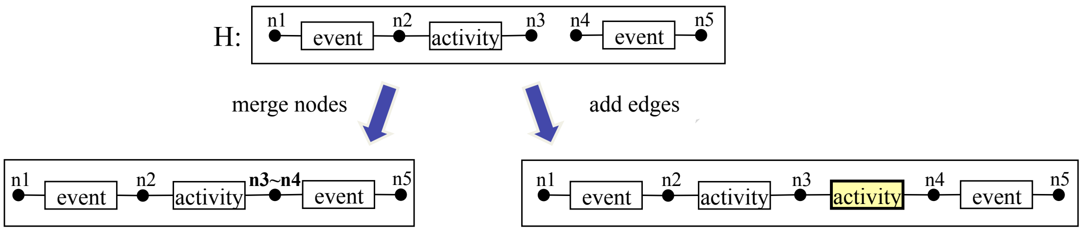

Figure 5: Hypergraph patches by example

H into a valid member of the language defined by a given grammar G. Two different kinds of atomic modifications are considered: merging nodes and adding edges. The application of a patch for a hypergraph H then corresponds to the construction of a so-called quotient hypergraph $H/\sim$ whose nodes are equivalence classes of the original nodes of H. Correcting patches indeed can be computed while parsing hypergraphs [MMM08a]. Consider the hypergraph H given in Fig. 5 as an example. Hypergraph H does not belong to the language of G_{BPM}, but it can be corrected by merging the nodes n3 and n4. It can also be corrected by inserting an *activity* hyperedge at the proper position. Note that there usually is an infinite number of correcting patches. Actually, according to G_{BPM}, an arbitrary number of activities could be inserted between the *activity* and the *event* hyperedge at the right. So, the size of desired patches, i.e., the number of additional hyperedges, must be restricted by the user. A special case of patches is the empty input hypergraph. Its patches can be used for exhaustive example generation.

Assistance based on hypergraph patches has been integrated into *DiaGen* editors as follows: The editor automatically maintains the hypergraph representation of the diagram. On user's request, the patch-computing parser [MMM08a] is applied to this hypergraph representation with the desired size of patches as a parameter. It computes all possible correcting hypergraph patches of this size. From those, the user has to choose via a preview functionality. The selected patch is translated to diagram modifications by a language-specific *update translator*. Finally, the diagram is beautified by a *layout* component. That way, powerful syntax-based user assistance for BPMs has been realized already [MM09] – however, only in the context of a conventional WIMP editor. A screencast is available at www.unibw.de/inf2/DiaGen/assistance/bpm.

For the computation of patches, *PerSUADE* can only consider the context-free part of a hypergraph [MM09]. This limitation naturally also applies to the sketch editors with guidance to be discussed in Section 4.

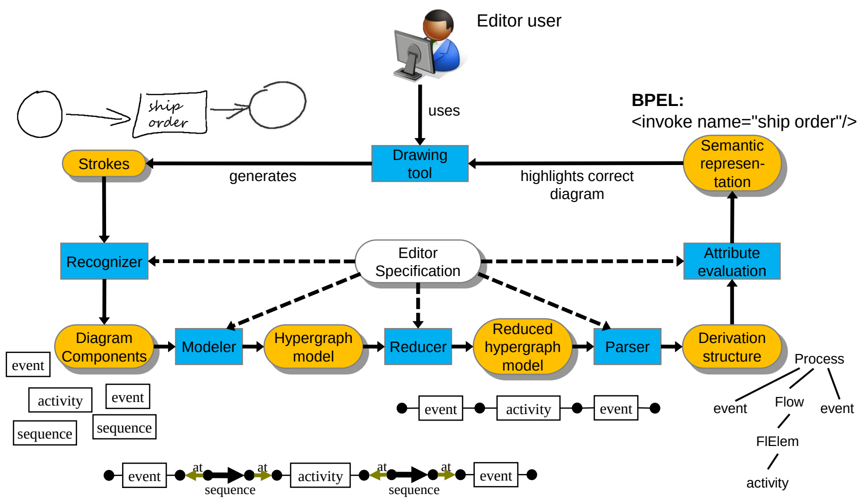

Figure 6: Architecture of *DSketch* (figure based on [BM08c])

3.3 Diagram Recognition à la *DSketch*

DSketch is an extension of *DiaGen* that complements the conventional WIMP-based GUI of diagram editors by a drawing canvas, which readily accepts all kinds of user strokes freely entered with a stylus. The integrated recognizer allows diagrams to be analyzed and further processed [BM08c]. The main characteristics of this approach are: (i) Little restrictions to drawing components, e.g., a rectangle can be drawn clockwise, counterclockwise, or even interleaved with other components. (ii) Syntactic and semantic information is used to resolve ambiguities that occur in the recognition process. For instance, if a sloppily drawn BPM component could be both an activity or an event, the actual decision is postponed to the analysis stage where the interpretation of the respective strokes might get clear from the context. And finally (iii), the approach is generic, i.e., editors for a wide range of languages can be specified.

Fig. 6 shows the overall architecture of this sketching approach. The first processing step is the *recognizer*, which analyzes the sketch's strokes and creates a corresponding set of diagram components. Actually, several primitive recognizers (called transformers in [BM08b]) for lines, arcs, circles, etc. search for corresponding primitives in the sketch. The main recognizer queries these primitive recognizers and – directed by the language specification – assembles the diagram components from those primitives. Generally, the recognition is very tolerant to avoid false negatives. The inevitably resulting false positives are resolved not until parsing. The actual analysis of the diagram now works in several steps similar to the analysis in conventional *DiaGen* editors: First, a hypergraph model is created from all components. Then the reducer is applied (lexical analysis) and yields the reduced hypergraph model as shown in Fig. 2. The parser syntactically analyzes this hypergraph and builds up a derivation structure that is similar to a derivation tree, but that also reflects non-context-free aspects of the diagram. The parser ensures that no two possible interpretations of the same stroke are integrated into the same derivation structure. That way, ambiguities are effectively resolved. Each derivation structure then represents a correct diagram and is rated according to its quality. Finally, a semantic representation of the best-rated

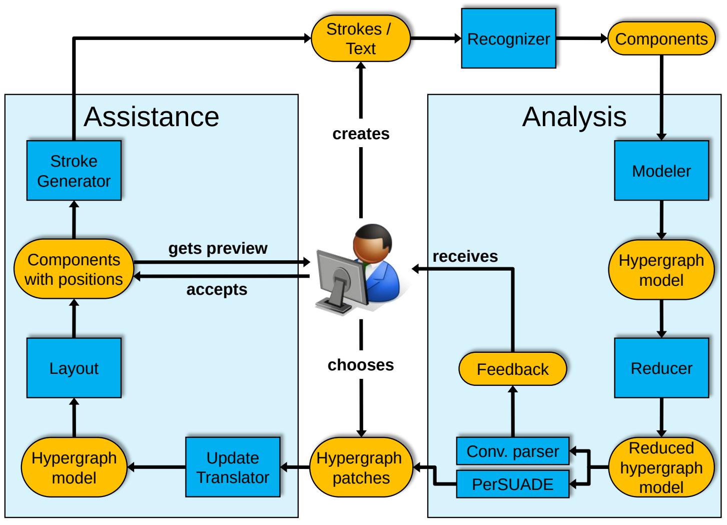

Figure 7: Novel architecture of sketch editors

derivation is computed via attribute evaluation. If this is not possible, the next best derivation is tried and so on. Details about this process can be found in [BM08a].

The *DSketch* approach is efficient and fully functional, but it cannot recognize dashed lines nor distinguish different line widths. BPM messages, which are usually drawn as dashed lines, and BPM end events, which are drawn as bold circles, thus, must be represented with another notation. Moreover, text recognition is not integrated into *DSketch*. Textual labels, hence, must be entered via keyboard or an extra text recognizer.

4 Integration of User Assistance into *DSketch*

In this section we describe how the assistance provided by *PerSUADE* has been integrated into *DSketch*. The overall architecture of the editors generated by the realized system is shown in Fig. 7, which basically refines Fig. 1. The right-hand side of Fig. 7 comprises the analysis steps of *DSketch*. The analysis performed by the *PerSUADE* parser belongs to this side, too. The left-hand side comprises the novel part of the system where the results of *PerSUADE* are further processed for the sake of assistance.

The processing steps up to the parser remain almost unchanged. Just the recognizer needed to be slightly adapted. Recall that in *DSketch* the recognizer is very error-tolerant. So, often the same stroke is accepted by several different primitive recognizers. This results in double findings that are resolved in *DSketch* during syntax analysis. The *PerSUADE* framework cannot deal with such ambiguities yet. Therefore, we have enforced the recognizer to make the decision if one of the assistance functions is invoked. Basically, the recognizer now selects the interpretation with the highest rating from the double findings. This rating depends on how precisely a primitive is drawn, how close the connections at the junctions are, and how well the corresponding constraints are met.

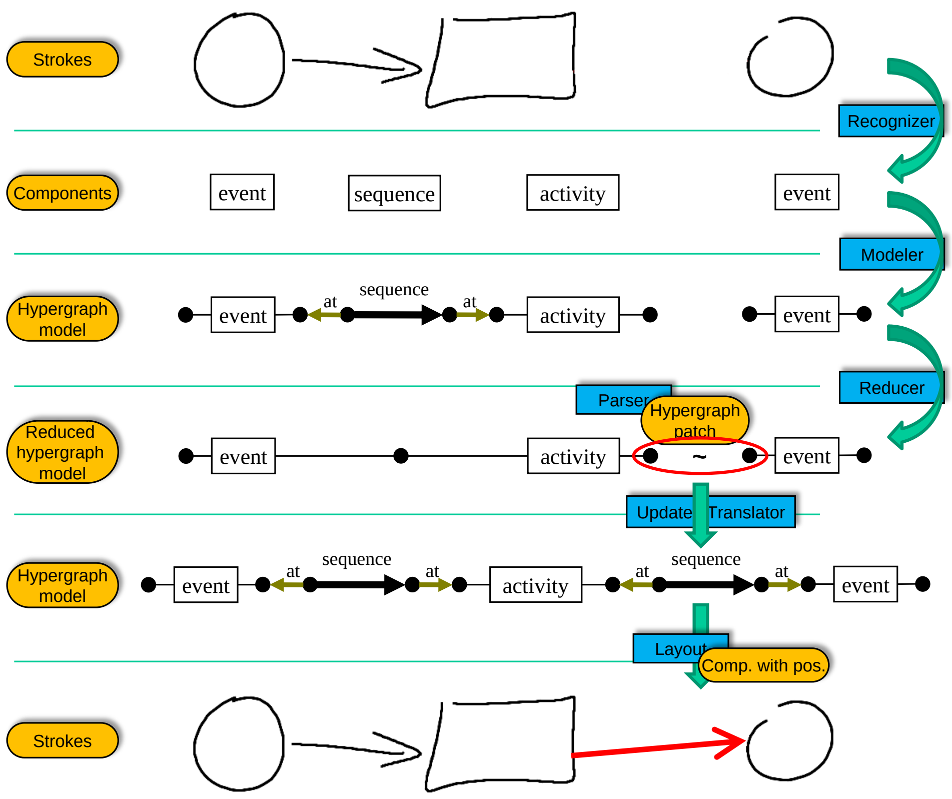

Figure 8: Processing steps by example

The next adapted component is the parser, i.e., the process of syntactically analyzing the diagram. Actually, the *DSketch* parser has remained unchanged, but an additional parser component from the *PerSUADE* framework now complements it. All kinds of assistance are supported by this parser instead of the normal *DSketch* parser. On user's request, this parser computes hypergraph patches for (the recognized parts of) the diagram's reduced hypergraph model. The user can explore these patches and choose one of them using a preview functionality.

Let us assume that the user has selected one of the patches. Consider the example traced in Fig. 8. There, the smallest existing patch just merges the outgoing node of the activity and the incoming node of the right-most event. The update translator translates this patch into changes of the hypergraph model. For our example, an arrow needs to be introduced that is attached to the activity and the event (indicated by the spatial relationship edges "at"). Thereafter, it is up to the layouter to find an appropriate position for the newly introduced components. For the example of Fig. 8, this is an easy task because the source and target components of the sequence arrow already exist. The more complex completion examples given in Fig. 9 and the generated example diagrams given in Fig. 10, however, show that this step is not always that simple. The used layout approach and the actual user interface are discussed in the following subsections. The last processing step, i.e., the stroke generator, is rather simple. It just draws perfect components with the optimal sample rate, thus maximizing the recognition rate.

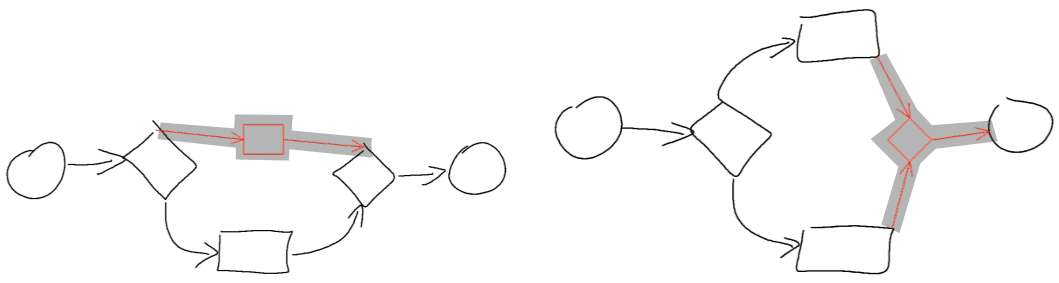

Figure 9: Auto-completion examples. Suggested completions are drawn in red.

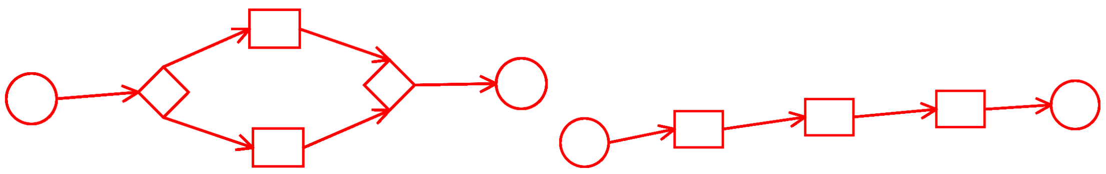

Figure 10: Example generation

4.1 Placement of New Components by the Use of Graph Drawing Techniques

A basic assumption of our implementation is that user strokes remain unchanged (in contrast to conventional *PerSUADE* editors, where the existing components can be adapted during assistance). That way, surprises are prevented and the special flavor of sketching is preserved. So, we need a flexible layout engine for graph-like languages (other languages would require some adaptations) that only integrates the new components and leaves the remaining diagram unchanged. These requirements can be satisfied by layout algorithms based on physical analogies [Bra01]. Concretely, we have adapted a spring embedder, which interprets edges as springs with their particular attraction forces. Furthermore, special repulsive forces take effect between the node components. During layout, the node components move in increments according to the respective sum of forces until an equilibrium state has been reached. However, in our context not all nodes can be moved around freely, but only the new ones introduced by the update translator. The existing nodes, in contrast, are locked into their positions. An important benefit of spring embedders besides their simplicity is that they can also be used for static layout in a straightforward way. Static layout is required here, e.g., for example generation (cf. Fig. 10).

There are two problems with spring embedders in our context: The top diagram of Fig. 9 would look much better if the new activity were positioned further to the top. However, spring forces pull the new activity to the presented position, i.e., springs prevent bent arcs that way. This behavior can only be avoided by introducing invisible components in a context-sensitive way. Another problem is that new components, if positioned randomly at the beginning, cannot "pass" existing components due to the repulsive forces. This may result in strange layouts. We have prevented this problem by introducing an additional processing step that guesses more appropriate initial coordinates for new node components to be refined afterwards.

Other layout algorithms might yield more common looking process models; actually, most professional business modeling tools apply Sugiyama-style layout algorithms. However, such algorithms are less suited in the context of this paper where the (usually user-chosen) position of existing nodes must be preserved when new nodes or edges have been introduced.

4.2 User Interface

The actual user interface is quite simple and easy to use — the complete editor window is shown in Fig. 2. There is a button for starting the computation of patches (see the magnified part of Fig. 2). After pressing this button, the first solution is shown immediately. Arrow buttons can be used for browsing through the other solutions. In particular the generation of examples usually results in a large number of solutions. A check button has to be pressed in order to accept the currently previewed solution. Strokes are then generated from the previewed diagram components. The resulting diagram looks like the preview, but the new components are not highlighted anymore, but drawn as normal, although perfect, strokes. Note that the user does not need to accept the previewed solution, but can use it as a kind of template for drawing the suggested components with his own strokes. That way, he will get a diagram that looks more homogeneous than the one with the generated perfect strokes. To continue with the UI, the preview also can be canceled, of course. Finally, the patch size can be set via the plus and minus buttons. This parameter basically indicates how many new diagram components (or more precisely, terminal hyperedges) are to be introduced. In the figure this parameter is set to its default value 1.

5 Discussion

Although an elaborate user study still remains to be done, the results so far are promising. As before, the user can freely sketch diagrams. He is not restricted in any way in the creative process of sketching. This actually is the reason why we have not realized a more pervasive assistance, e.g., on-line after every single stroke. In many conventional sketch editors, the user is in trouble if his sketch cannot be recognized. With the developed editor, he can ask for syntactical guidance instead. But this is not the only help he can get as we describe next.

5.1 Location of Recognition Errors

An important benefit of our approach is that it helps identifying *recognition errors* (besides syntax errors). Consider Fig. 11 as an example. A human can easily see that the sketched diagram is a structured business process. Still the diagram is not correctly recognized by *DSketch*. Normally, the user would have no clue what is wrong here. Has the start event mistakenly been recognized as an activity? Or has the end event been drawn too sloppily? Invoking assistance yields the answer. The red arrow between the activity and the end event clearly points out the problem: either the existing arrow has not been correctly recognized, or the gap between its head and the end event is too large. In either case, the user now can correct this problem without the need to redraw the whole diagram. It would even be possible to automatically mask or remove those strokes that do not contribute to the solution.

This problem, however, mainly arises for quite restricted visual languages where either the whole diagram is correct or nothing. Indeed, a single misrecognized arrow affects the correctness

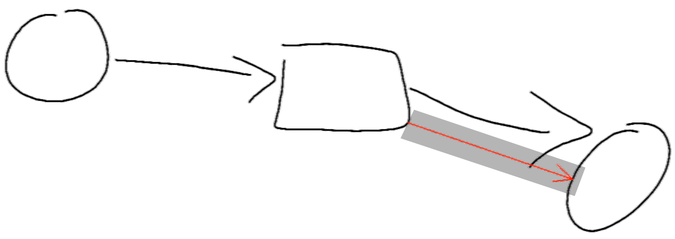

Figure 11: Identification of recognition errors

of the whole BPM. It simply is not well-structured anymore as we have required by the grammar. With a more relaxed syntax definition at least sub-diagrams would be recognized correctly so that the visual feedback given by the *DiaGen* parser might indicate what is wrong. Actually, languages, where either the whole diagram or nothing is recognized as correct, have been very critical for sketching systems so far, because the recognition rate exponentially drops down with the size of the diagram. This problem is solved with our approach (although it would be even better to re-feed the analysis result in the recognizer, so that it can try harder at the weak points).

5.2 Stroke Interference

A problem of integrating *PerSUADE* into a sketching system is that it may happen that a sketch is not recognized as correct after a suggested patch has been accepted by the user. When using *PerSUADE* in conventional WIMP editors, this cannot happen: diagrams resulting from the application of assistance are always correct. In the context of sketching, newly generated strokes may interfere with existing strokes that, e.g., had been ignored by stroke recognition before.

6 Related Work

Of course, there are also other sketch editors for BPMs such as [MS09]. Moreover, due to the practical relevance of this language, various kinds of guidance have been developed for conventional WIMP-based BPM environments (an example is [BBM$^+$09]). However, to our best knowledge such guidance has not been integrated into sketch editors yet.

As already noted in the introduction, the most closely related work is [CDR07] by Costagliola et al. Here, an LR parser as known from textual languages is used for syntax analysis with respect to a so-called sketch grammar. Thereby, syntactic information is exploited to resolve ambiguities similar to the *DSketch* approach [CDR08]. The symbol table of the parser then can be exploited to realize symbol completion in sketch editors (and so-called symbol prompting in conventional diagram editors). The strong points of this approach are that it is generic, that direct feedback is provided (the approach is actually incremental), that the user's own drawing style is used for completion (a stroke repository is filled by the different symbol recognizers to this end), and that the recognition of complex symbols can generally be improved that way. But, like with our approach, explicit user interaction is still required. In contrast to [CDR07], our approach does not stop at the lexical level, but also considers the overall diagram structure. Even other kinds of assistance not necessarily based on syntax could be integrated.

Another meta-tool where it should be possible to combine assistance with sketching is the Marama toolkit. For Marama, both a critic authoring tool [AHHG09] for the specification of user feedback and a sketching framework [GH07] are available. Here, however, critics would have to be specified manually whereas we gain the feedback automatically from the parser. The strong points of [GH07] are that only very little extra specification effort is needed for complementing a normal diagram editor with a sketching editor and that the user can easily overrule the recognizer when it makes a mistake.

7 Conclusion

In this paper we have shown that user assistance functionality can be provided by sketch editors and that this actually is useful. The presented approach allows to generate sketching editors with user assistance from a language specification based on the existing sketching editor generator *DSketch* and the user assistance library *PerSUADE*. As a representative example, we have created a sketch editor for business process models with assistance features such as auto-completion or example generation.

But we have noticed yet another benefit of this approach besides helping the user with the language. The very same assistance features actually can be put to a good use in locating recognition errors. Those often directly result in syntax errors, whose potential corrections then point the user precisely to the recognition error. If a new component is suggested as a correction where already a component exists, the user can conclude that the existing component had been drawn too sloppily and needs to be redrawn.

The developed sketch editor for business process models is demonstrated in several screencasts and can be downloaded from www.unibw.de/inf2/DiaGen/assistance/sketching.

Future Work

In the future we want to experiment with relaxations of the assumption that the existing user strokes must not be changed. It is certainly imaginable that sketched components are moved around or even resized similar to the assistance in conventional *DiaGen* editors [MM09]. In this context it should also be possible to integrate existing component fragments into the newly introduced components in order to reuse as many strokes of the user as possible.

It would be also important to integrate the suggestions into the diagram closely following the user's drawing style. Perfect components mixed with sloppily drawn components make the diagram look inhomogeneous. Costagliola et al. have proposed a stroke repository to this end, which is used already for their symbol completion [CDR07]. Alternatively, the user strokes could be beautified to close this gap.

Finally, *DSketch* and *PerSUADE* need to be more deeply intertwined. While *DSketch* originally postpones final decision of stroke recognition until syntax analysis in order to improve the recognition rate and to make ambiguity resolution possible, we had to enforce early recognition decisions in order to integrate *PerSUADE* into *DSketch*.

Bibliography

[AHHG09] N. M. Ali, J. Hosking, J. Huh, J. Grundy. Critic Authoring Templates for Specifying Domain-Specific Visual Language Tool Critics. In *Proc. 2009 Australian Software Engineering Conf.* Pp. 81–90. IEEE, 2009.

[BBM$^+$09] M. Born, C. Brelage, I. Markovic, D. Pfeiffer, I. Weber. Auto-completion for Executable Business Process Models. In *Business Process Management Workshops*. LNBIP 17, pp. 510–515. Springer, 2009.

[BM08a] F. Brieler, M. Minas. Ambiguity Resolution for Sketched Diagrams by Syntax Analysis Based on Graph Grammars. In *Proc. Seventh Int. Workshop on Graph Transformation and Visual Modeling Techniques*. Electronic Communications of the EASST 10. EASST, 2008.

[BM08b] F. Brieler, M. Minas. A Model-Based Recognition Engine for Sketched Diagrams. In *Proc. VL/HCC Workshop on Sketch Tools for Diagramming*. Pp. 19–28. 2008.

[BM08c] F. Brieler, M. Minas. Recognition and processing of hand-drawn diagrams using syntactic and semantic analysis. In *Proc. Working Conf. on Advanced Visual Interfaces*. Pp. 181–188. ACM, 2008.

[Bra01] U. Brandes. Drawing on Physical Analogies. In *Drawing Graphs: Methods and Models*. LNCS 2025, pp. 71–86. Springer, 2001.

[CDR05] G. Costagliola, V. Deufemia, M. Risi. Sketch Grammars: A Formalism for Describing and Recognizing Diagrammatic Sketch Languages. In *Proc. Eighth Int. Conf. on Document Analysis and Recognition*. Pp. 1226–1231. IEEE, 2005.

[CDR07] G. Costagliola, V. Deufemia, M. Risi. Using Grammar-Based Recognizers for Symbol Completion in Diagrammatic Sketches. In *Proc. Ninth Int. Conf. on Document Analysis and Recognition*. Pp. 1078–1082. IEEE, 2007.

[CDR08] G. Costagliola, V. Deufemia, M. Risi. Using Error Recovery Techniques to Improve Sketch Recognition Accuracy. In *Proc. 7th Int. Workshop on Graphics Recognition*. LNCS 5046, pp. 157–168. Springer, 2008.

[CMP05] R. Chung, P. Mirica, B. Plimmer. InkKit: A generic design tool for the tablet PC. In *Proc. 6th ACM SIGCHI NZ chapter's Int. Conf. on Comp.-Human Interaction*. Pp. 29–30. ACM, 2005.

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

[GH07] J. Grundy, J. Hosking. Supporting Generic Sketching-Based Input of Diagrams in a Domain-Specific Visual Language Meta-Tool. In *Proc. 29th Int. Conf. on Software Engineering*. Pp. 282–291. IEEE, 2007.

[HD05] T. Hammond, R. Davis. LADDER, a sketching language for user interface developers. *Computers & Graphics* 29(4):518–532, 2005.

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

[MM09] S. Mazanek, M. Minas. Business Process Models as a Showcase for Syntax-based Assistance in Diagram Editors. In *Proc. 12th Int. Conf. on Model Driven Eng. Lang. and Sys.* LNCS 5795, pp. 322–336. Springer, 2009.

[MMM08a] S. Mazanek, S. Maier, M. Minas. An Algorithm for Hypergraph Completion According to Hyperedge Replacement Grammars. In *Proc. 4th Int. Conf. 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 *2008 IEEE Symposium on Visual Languages and Human-Centric Comp.* Pp. 242–245. IEEE, 2008.

[MRA09] J. Mendling, H. Reijers, W. van der Aalst. Seven Process Modeling Guidelines (7PMG). *Information and Software Technology*, 2009.

[MS09] N. Mangano, N. Sukaviriya. Liberating Expression: A Freehand Approach to Business Process Modeling. In *Proc. 12th IFIP TC 13 Int. Conf. on Human-Comp. Interaction*. LNCS 5727, pp. 834–835. Springer, 2009.

[Obj09] Object Management Group. Business Process Modeling Notation (BPMN). 2009. http://www.omg.org/docs/formal/09-01-03.pdf.

[SBV08] S. Sen, B. Baudry, H. Vangheluwe. Domain-Specific Model Editors with Model Completion. In *Models in SE*. LNCS 5002, pp. 259–270. Springer, 2008.