

Business Process Models as a Showcase for Syntax-based Assistance in Diagram Editors

Steffen Mazanek and Mark Minas

Universität der Bundeswehr München, Germany,
{`steffen.mazanek|mark.minas`}@unibw.de

Abstract. Recently, a generic approach for syntax-based user assistance in diagram editors has been proposed that requires the syntax of the visual language to be defined by a graph grammar. The present paper describes how this approach can be applied to the language of business process models (BPMs), which is widely used nowadays. The resulting BPM editor provides the following assistance features: combination or completion of BPM fragments, generation of BPM examples, an extensive set of correctness-preserving editing operations for BPMs, and auto-link, i.e., the automatic connection of activities by sequence flow. Furthermore, this paper contains a discussion of the scalability and scope of the used approach. This also comprises a characterization of the languages where it can be put to a good use.

1 Introduction

These days, meta-tools are widely used for the development of diagram editors. That way, an editor can be developed with virtually no programming effort. Just an abstract language specification, e.g., based on a metamodel or a kind of grammar, has to be provided from which the complete editor is generated. Well-known examples of meta-tools are MetaEdit+ [1] and GMF [2]. Beyond these there are several research tools like AToM³ [3], Pounamu [4], or DIAGEN [5].

State-of-the-art meta-tools provide a lot of assistance for the *editor developer*. For instance, the appearance of diagram components and the syntax of the language mostly can be specified in a visual way. In contrast, the generated editors often do not provide a lot of assistance for their *actual users* (such as help with incorrect diagrams). This observation motivates the development of generic approaches to user assistance in diagram editors. An important requirement for the adoption of such an approach surely is that minimal additional programming or specification effort should be imposed on the editor developer. Rather the already existing specification should be pushed to its limit.

Recently, such an approach has been proposed and integrated into DIAGEN [6, 7]. Furthermore, the different assistance features enabled by this approach have been described, i.e., auto-completion (deduce missing diagram components), diagram correction (combine diagram fragments) and example generation [7], correctness-preserving editing operations [8], and diagram contraction/auto-link [9] — Sect. 3 provides concrete examples of all those. However,

up to now this approach only has been applied to toy examples such as the archaic Nassi-Shneiderman diagrams or the equally simple Flowcharts. Although the results have been promising, a real world example is required to foster further adoption of the approach. Therefore, this paper discusses in detail how the approach can be applied to business process models (BPMs), which are certainly a highly relevant language today, and what has been achieved in doing so.

The paper is structured as follows: First, the language BPM is briefly introduced (Sect. 2). For the sake of motivation, this paper continues with the presentation of the actual outcome, i.e., the assistance features of the generated BPM editor (Sect. 3). Only then it is explained how the language had to be modeled in order to apply the approach (Sect. 4). Along the way the approach itself is recapitulated to make this paper self-contained (Sect. 5). Moreover, a basic understanding of the approach is also required in order to understand its scope, which is discussed in Sect. 6. This section contains some performance data as well. Finally, related work is reviewed (Sect. 7) and the paper is concluded.

2 Business Process Models

BPMs are mostly used to represent the processes (i.e., workflows) within an enterprise. In recent years a standardized visual notation, the Business Process Modeling Notation BPMN [10], has been developed, which is readily understandable by different kinds of business users with different levels of expertise. In this paper a subset of BPMN is considered that is outlined by example next.

Fig. 1 shows a typical sales process: A customer orders a product from a company, which ships it if available. The diagram consists of two pools (big rectangles), “Customer” and “Sales Department”, which act as containers for the actual processes. The upper pool contains a start event (simple circle), an activity (rounded rectangle), an intermediate receive event (two nested circles), and an end event (thick circle), which are connected by arrows representing the sequence flow. The other pool contains, among others, two gateways (diamonds), which are used to split and join the sequence flow. Finally, messages (dashed arrows) can be sent in the course of an activity.

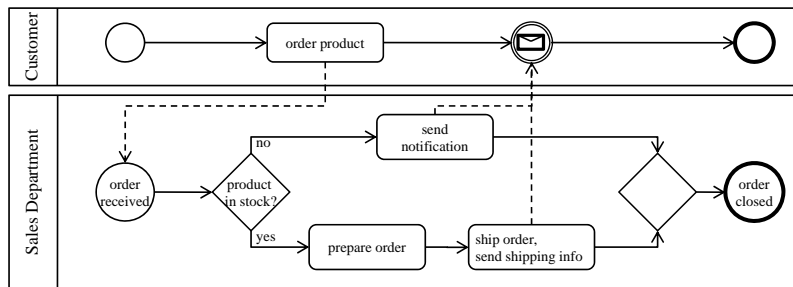


Fig. 1. Example BPM

In the following only well-structured BPMs are treated, i.e., it is required that splits and joins are properly nested such that each split has a corresponding join. This restriction is considered to improve the quality of process models [11] similar to structured programming, which improves the quality of program code.

3 Resulting Assistance

Next, the assistance features are described that are provided by the generated BPM editor. It has to be stressed again that the realization of all these features has required virtually no extra programming effort.

Auto-completion: Incomplete BPMs usually occur as intermediate diagrams during modeling. But they might also result from a lack of knowledge of a beginner user. The developed editor can generate suggestions on how to complete such diagrams [7]. Actually, it computes all possible completions up to a user-defined size. Fig. 2 shows three examples how given incomplete BPM diagrams can be completed. For the first one, the smallest possible completion consists of a gateway and three arrows. The second one can be completed by adding a fresh activity with a default text and linking it properly. Finally, the two BPM fragments given at the right-hand side can be combined into a well-structured process just by introducing two arrows. Regarding the user interface, auto-completion is supported by a dialog that allows the user to browse and preview all possible completions for his diagram.

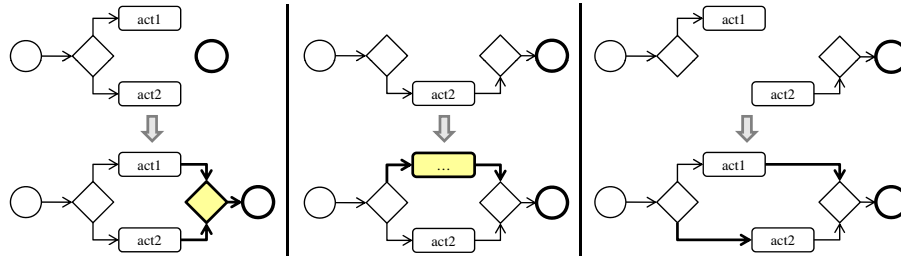


Fig. 2. Auto-completion of BPMs

Example generation: An important special case of completion is the empty diagram, completions of which can be used to enumerate the language. Given the number of diagram components (arrows are not counted), all possible BPMs of this size without messages (cf. Sect. 6) can be generated. The user can browse this set to get valuable insight into the language. This is shown in Fig. 3. There are no BPMs of size less than three (intermediate events and pools are not considered in the figure for the sake of conciseness). Since all structurally different examples are generated, their number grows exponentially with the size parameter. Still it is useful to have a look at some of them, in particular because an example can be selected to be the starting point of further editing.

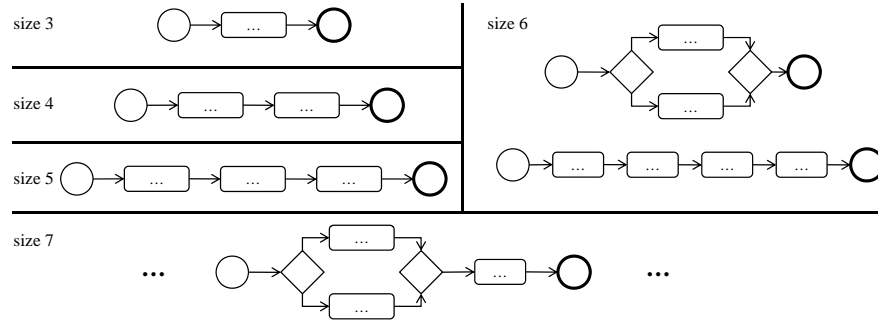


Fig. 3. Generation of BPM examples

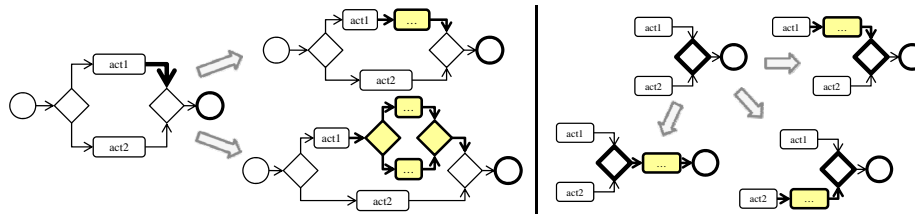


Fig. 4. Generation of correctness-preserving editing operations

Editing operations: In addition, powerful correctness-preserving editing operations can be generated [8]. Some example operations are shown in Fig. 4. In the diagram at the left-hand side, the thick arrow has been selected by the user to determine the context for the operations he is interested in. The highlighted components are introduced by the particular operation. The figure also shows that it is sometimes meaningful to allow operations to introduce more than one new component. Otherwise, a gateway could not be inserted into a correct BPM. On the right-hand side the operations provided in the context of a gateway are shown (only a fragment of a correct BPM is shown though). Besides operations that add components, there also is an intelligent remove operation. Therewith, one or several selected components can be removed and the remaining diagram is reconnected automatically (if possible), cf. Fig. 5.



Fig. 5. Intelligent remove

Auto-link: Finally, inspired by [12], the editor provides auto-link to further improve the user's editing performance. This is shown by example in Fig. 6. The

missing arrows are derived from the spatial arrangement of the activities, which is mostly preserved. The realization of auto-link is explained in detail in [9].



Fig. 6. Auto-link

4 Modeling BPMs with a Graph Grammar

The editor, whose features have been discussed in the previous section, has been realized using the DIAGEN framework [5]. It has to be stressed, though, that the assistance part is provided as a separate library [6] independent of DIAGEN.

The main advantage of DIAGEN editors is that they seamlessly integrate syntax-directed and free-hand editing. In free-hand mode, diagrams can be drawn without any restrictions in the manner of a drawing tool. Thereby, feedback about the syntactical correctness of the diagram is consistently provided by an analysis module. On the other hand, syntax-directed editing operations can be defined by the editor developer to simplify frequent editing tasks of the users. With the newly developed user assistance described in the previous section further guidance is provided. So, for the sake of editing freedom the editor still does not prevent the user from drawing an incorrect diagram. But on request it offers powerful syntactical assistance helping him to end with a correct one.

DIAGEN editors use hypergraphs as a diagram model and hypergraph grammars to define the syntax of the language. Furthermore, all editors generated with DIAGEN follow the same general editing process. This process consists of several steps as shown in Fig. 7. Those are informally described next.

With the *drawing tool*, the editor user can create, delete, arrange and modify the diagram components as defined in the editor specification. Components usually have some layout attributes, e.g., the position and size of an activity. Additionally, a set of properties like the label of an activity or the type of an intermediate event can be defined, which can be manipulated via a special dialog.

The first processing step, the *modeler*, creates the Spatial Relationship Graph (SRG) from these components. Therefore, it first creates component hyperedges for each diagram component and nodes for each of their attachment areas. An attachment area is a determined part of a diagram component that can interact with other diagram components. Afterwards, the modeler checks for each pair of attachment areas whether they are related as defined in the specification. For instance, an arrow end and an event are *at*-related if both attachment areas overlap. As another example, a pool p_1 is *below* another pool p_2 if $p_1.y > p_2.y + p_2.h$. For each relationship detected, the modeler adds a corresponding

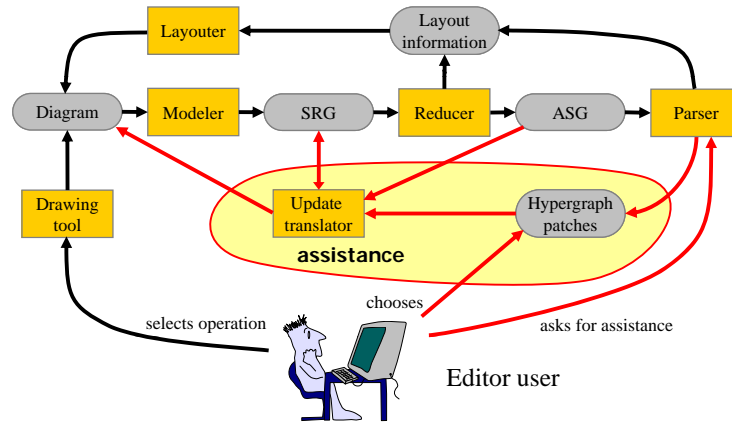


Fig. 7. Editing process of DIAGEN editors

relationship edge between the two attachment nodes involved. This graphical scanning step is crucial for free-hand editing.

In Fig. 8, a BPM fragment and its SRG are shown. The hyperedges are represented as rectangular boxes and the nodes as black dots. If a hyperedge and a node are incident, they are connected by a line.¹ Binary hyperedges (such as all relation edges, the sequence arrows, and the messages) are simply represented as arrows. Activities and intermediate events have three attachment areas: two for incoming resp. outgoing sequence flow, one for messages. Gateways have four attachment areas (namely their corners), and start events have three (one for the corresponding pool, one for messages, and one for outgoing sequence flow).

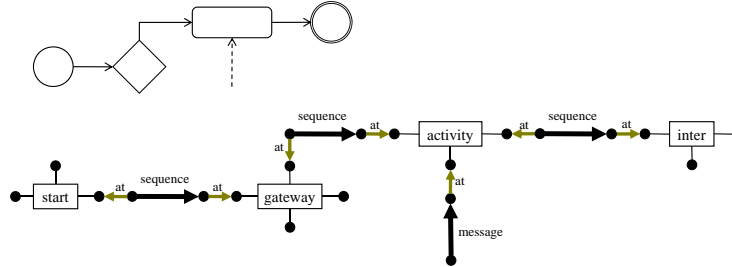


Fig. 8. BPM fragment and its SRG

As one can already see, the SRG becomes quite large. DIAGEN editors therefore do not analyze the SRG directly, but simplify it first according to the spec-

¹ Each line represents a particular role (like “incoming sequence flow”). However, instead of using labels the graphical arrangement implicitly determines the roles.

ification (similar to lexical analysis in compilers). This step is performed by the *reducer*, which creates the so-called Abstract Syntax Graph (ASG). In case of BPM there is a close correspondence between SRG and ASG. In the ASG, nodes of the SRG that are connected by a relationship edge are merged. Furthermore, the arrows for sequence flow do not occur in the ASG anymore. Rather the nodes connected by such an arrow are also merged. Fig. 9 shows the complete ASG of the example sales process. This ASG now directly represents the structure of the diagram and, thus, can be syntactically analyzed by the *parser*.

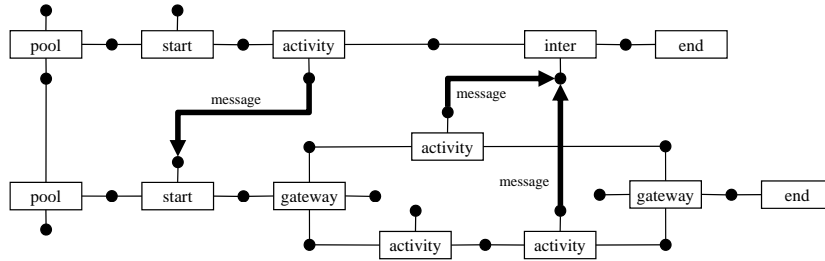


Fig. 9. ASG of the example sales process shown in Fig. 1

In DIAGEN, hypergraph grammars are used for language definition. They generalize the idea of Chomsky grammars for strings as used by standard compiler generators. Each hypergraph grammar consists 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 the grammar is defined by the set of terminally labeled hypergraphs that can be derived from the starting hypergraph.

Fig. 10 shows the productions of the hypergraph grammar G_{BPM} whose language is just the set of all ASGs of structured BPMs. For conciseness, productions $L ::= R_1, L ::= R_2, \dots$ with the same left-hand side are drawn as $L ::= R_1|R_2|\dots$. The types *pool*, *start*, *end*, *inter*, *activity*, *gateway* and *message* are terminal hyperedge labels being used in ASGs. The set of nonterminal labels consists of *BPM*, *Pool*, *Process*, *Flow* and *FlElem*. The starting hypergraph consists of just a single *BPM* edge with an incident node. Most of the required productions are context-free, i.e., their left-hand side consists of just a single nonterminally labeled hyperedge together with the appropriate number of nodes. There are only three non-context-free productions, P10-P12, that embed a *message* between two activities or an activity and a start resp. intermediate event.

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 . Context-free hypergraph grammars are described in detail in [13]. Fig. 11 shows an example derivation starting from *Process*. Only for the last step, the

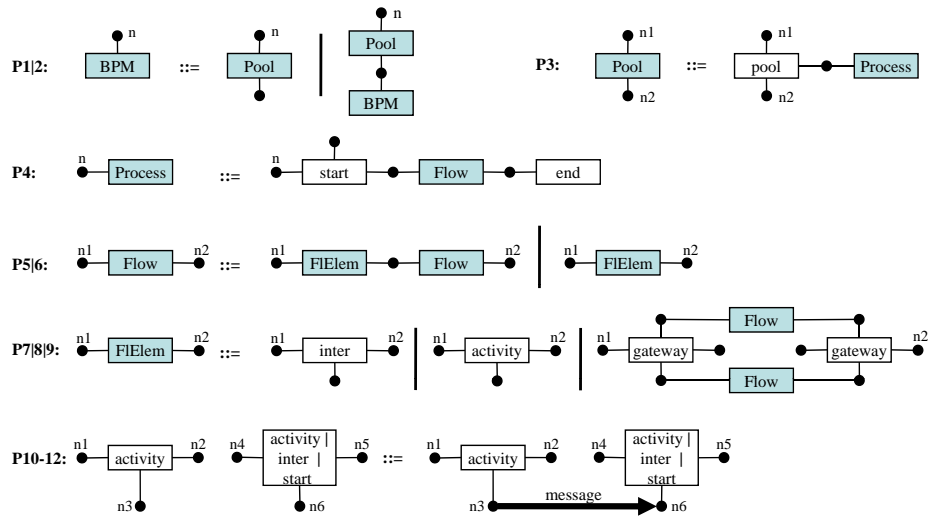


Fig. 10. Hypergraph grammar for BPMs

introduction of a *message*, an embedding production has to be applied. The grammar G_{BPM} is *unambiguous*, so that there is a unique derivation tree for the context-free part of every hypergraph of the language, which can be constructed by the parser.

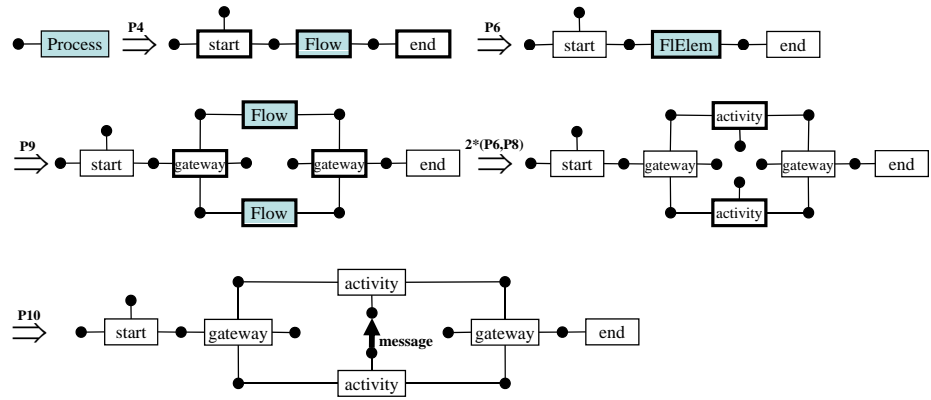


Fig. 11. Example derivation

The final processing step of a DIAGEN editor is the *layout*, which computes a layout for the diagram. The developed BPM editor relies on a constraint-based layout, where the constraints are gathered, among others, from the derivation information resulting from the parser.

5 Hypergraph Patches for User Assistance

In previous work [7], *hypergraph patches* have been proposed as a means for the realization of user assistance in diagram editors. A patch basically describes a modification of a given hypergraph H . Two different kinds of atomic modifications are considered: merging nodes and adding edges. Of course, arbitrary modifications of a hypergraph are not very helpful. Rather those modifications are required that transform H into a valid member of the language defined by a given grammar G . Such patches indeed can be computed while parsing [6].

Consider the hypergraph H given in Fig. 12 as an example. For simplicity, assume that *Process* is the start symbol of G_{BPM} , i.e., disregard pools for a moment. H then can be corrected by merging the nodes $n5$ and $n6$. However, it can also be corrected by inserting an *activity* hyperedge at the proper position. Note that there might be an infinite number of correcting patches. Actually, an arbitrary number of activities or intermediate events could be inserted between the *activity* and the *end* hyperedge in H . So the size of desired patches (i.e., the number of additional hyperedges) has to be restricted.²

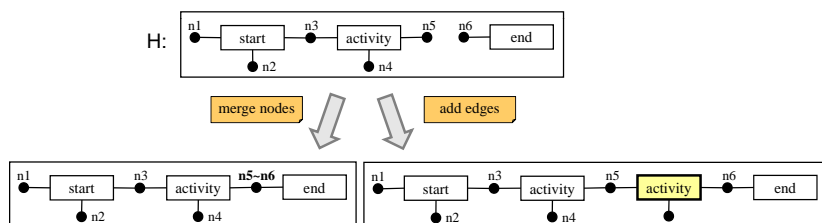


Fig. 12. Hypergraph patches in the context of BPM

Fig. 7 also shows how patches can be integrated into the DIAGEN editing process: On user’s request the parser is triggered 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. Next, the selected patch is applied and embedded into the SRG using a language-specific *update translator*. The editor then calls the reducer and parser again. Finally, the layouter arranges the new components within the diagram and adapts existing components if necessary.

The update translator for BPMs can be implemented quite straightforwardly. For instance, if $n5$ and $n6$ in Fig. 12 are to be merged, this is translated to embedding a sequence arrow (and some relations) between the *activity* and the *end* event. As another example, imagine $n1$ had to be merged with an attachment node of a particular *pool* edge. Then a spatial relationship edge *inside* has to

² The deletion of edges and the splitting of nodes has not been considered, because existing diagram components and “relevant” spatial relations of the user’s diagram should be respected (except for intelligent remove). Also, patches are not allowed to introduce nonterminal edges, because those do not have a visual representation.

be inserted between the corresponding nodes in the SRG. After reduction and parsing, the layouter moves the corresponding sub-process inside of the pool.

Patches are also useful if the given diagram is correct already. For instance, the operations introduced in [8] even require the input diagram to be correct. Additionally, the user has to select a context in order to access possible operations. The key idea has been to separate those edges on the ASG level that correspond to the user's selection from the non-selected part. Roughly speaking, the ASG is artificially broken into pieces and repaired again with certain, relevant patches that constitute meaningful operations.

6 Discussion

It is important to stress that the described patches are solely computed from the grammar. Semantics is not at all considered. Therefore, an activity (or any other diagram component) inserted as assistance only carries a default label.

Generally, the possible assistance is heavily affected by the way a language is modeled. For instance, with the grammar shown in Fig. 10 only the start event is connected to the containing pool on the abstract syntax level. Therefore, an unconnected activity could even be moved to another pool for the sake of correction. One could have also modeled the language in a way such that each activity is connected to the pool it is surrounded by. This would prevent the movement of activities to other pools.

Another important issue is how to deal with the different gateway types that are possible in BPMN. At the moment, the type (parallel, exclusive, etc.) can be adjusted in a property dialog, but it only affects the visual appearance of the gateway component. However, a branching can easily be prevented from being joined by a different type of gateway: It is sufficient to add another terminal symbol *par_gateway* and another production equal to P9, but with *par_gateway* edges instead of *gateway*. The grammar could also be extended to support more than two parallel branches or other kinds of structured concepts such as loops. However, with loops it has to be accepted that mostly there will not be a unique solution for auto-link anymore.

Note that based on G_{BPM} example generation yields a very large number of results (at least if used straight away). The problem is that activities and intermediate events can be used interchangeably according to the syntax of the language. So, in order to get an example with n activities, a lot of (i.e., $2^n - 1$) further examples will be generated with intermediate events instead of certain activities. Therefore, the editor developer can list such (somehow redundant) edge types so that the parser does not create those edges for the sake of completion. Of course, intermediate events drawn by the user are still perfectly accepted.

6.1 Who benefits?

An empirical user study has not been conducted yet, but is being planned. However, some benefits of the provided assistance can already be discussed in an abstract way. Remarkably, both the users and the developer benefit.

Beginner users can quickly explore and learn the visual language at hand by looking at example diagrams. That way, they do not need to know anything about grammars, which makes the editor more accessible. After having gained some insight in the particular language, they can start drawing diagrams. In case of modeling errors they can rely on the provided assistance.

Advanced users benefit from auto-link, which avoids the tedious work of drawing connections, and the generated editing operations, which provide a lot of flexibility and speed up editing.

The *editor developer* does not need to specify (certain) editing operations by hand anymore, which is a tedious and error-prone task. Rather diagram-specific editing operations are automatically computed from the grammar at runtime. Moreover, prototyping and testing of editors is simplified, because the generated examples can be used to quickly validate the specification.

6.2 Scope of the Approach

There is a severe general restriction of the approach: Only languages that are mostly context-free can benefit from its application. So, for the computation of patches embedding productions are not considered. In case of BPMs this means that messages, which can be embedded between arbitrary activities, are not part of the computed patches. Existing messages drawn by the user are tolerated by the parser, of course. If only well-structured BPMs have to be covered, all other productions are context-free (cf. Fig. 10), so that the approach is still applicable. Arbitrary sequence flow, however, cannot be supported.

But why not consider embedding productions for the sake of completion? There is a simple intuition behind this restriction: The more restricted a language is, the more powerful the possible assistance can be. If everything is allowed, no corrections are required at all. Since messages can be inserted between arbitrary activities, just too many solutions would exist (n^2 for n activities). Those cannot be browsed easily anymore. A completely different user interface would be required to make effective use of this information, e.g., connector handles. Therefore, the parser does not create messages (and other kinds of embedded edges) as part of patches.

Another reason why languages such as class diagrams can hardly be supported is that graphs of this language usually consist of a lot of different connected components. However, heavily disconnected graphs can be derived from a grammar in a lot of different ways, i.e., the language is inherently ambiguous. Therefore, one and the same completion would be returned as a result very often. Even worse, for disconnected graphs parsing is known to be inefficient. To overcome this issue, DIAGEN provides so-called set-productions that can be used if the order in which components are derived does not matter. But those are not supported (yet) for completion.

However, several practical languages already can be modeled in a way that avoids these issues. For instance, an editor for Message Sequence Charts (MSCs) with user assistance has also been specified already. Lifelines thereby have to be

arranged in the ASG in a particular order, i.e., from left to right.³ Messages, again, are just embedded. Fig. 13 shows an MSC and its ASG to clarify this issue. Modeling MSCs that way allows for plenty of user assistance: Examples (without messages) can be generated, isolated actions can be moved onto lifelines, editing operations can be used to insert an action between existing actions and so on.

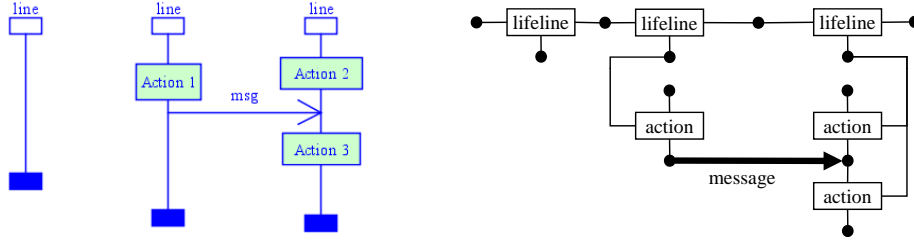


Fig. 13. Modeling MSCs: example diagram and corresponding ASG

6.3 Performance

Performance has always been a problem of the approach (or, more precisely, its realization). However, recently a substantial improvement factor has been achieved with respect to performance, so that the patch-computing parser is now ready for practice. On the one hand, the algorithm itself has been improved by using more intelligent data structures. On the other hand, support for multi-threading has been added. Indeed, the filling of layers by the parser as described in [6] can be parallelized in a straightforward way (one thread per production).

Fig. 14 provides some performance data gathered on a standard Notebook (Intel Core2 Duo with 2GHz each, 2GB RAM). As input BPMs with several pools have been used that contain a process at a time consisting of a start event, an activity, and an end event. Every input graph contains exactly one error that can be repaired by merging two nodes. The x-axis determines the size of the input graph, i.e., the total number of edges. Data for different patch sizes (number of additional edges to be added by the parser) has been collected. With zero edges (add 0) just the two separated nodes are merged. If several edges are to be introduced (add 1,2,3) the only result is the introduction of this number of activities between the separated nodes. Performance naturally drops if too many edges are to be introduced. However, it can be seen that for most small and medium-sized diagrams assistance can be computed in less than a second.

Future performance improvements are possible by introducing better support for modularization. For instance, in BPM better performance could be achieved, if the pools with their respective contents would be treated independently.

³ The same approach has also been used in BPM for modeling the pools.

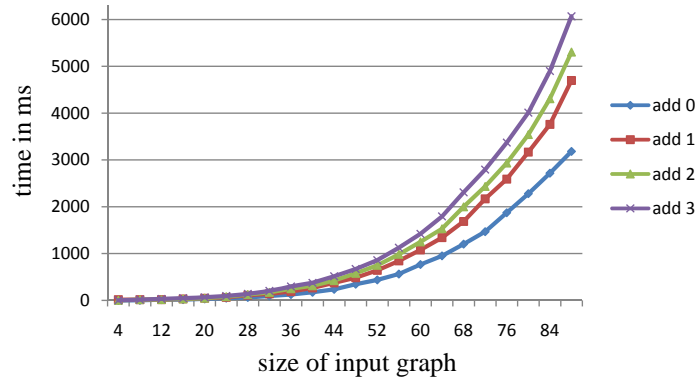


Fig. 14. Some performance data

7 Related Work

Text editors: In the context of text editors, approaches for the combination of different editing modes have a long tradition. A widely known system is the Synthesizer Generator [14], which allows the generation of editors that support both structure and text editing. However, following this approach the editor developer has to decide which syntactic construct is to be edited in which mode. The Syned system [15] overcomes this limitation by seamlessly integrating both editing modes, so that the users can edit as they like and not as the developer thought they would like. However, this approach cannot directly be transferred to diagram editors, because in diagrams there is no obvious concrete representation of nonterminal symbols in general. Moreover, for a novice user it is probably easier to only deal with fully expanded diagrams (and still have syntax-directed operations at hand). Regarding error recovery, modern textual IDEs, be they generated or hand-crafted, certainly provide sophisticated user support [16].

Meta-tools: GMF [2] editors provide connector handles (drag connections out of a node) and action toolbars (fill compartments of a node, e.g., adding an attribute to a class). MetaEdit+ [1] provides different kinds of static assistance [17]. So, it automatically creates a language help based on the data entered by the language developer. It also supports the creation of so-called tutorial projects. The tool AToM³ has been extended to support model completion [18]. Constraint logic programming is exploited to this end (as in SmartEMF [19], GEMS [20], or the check engine of [21]). Ehrig et al. suggest the generation of instance models of a particular metamodel by means of a graph grammar [22].

BPM tools: Koschmider et al. have proposed ideas for user assistance in BPM tools. In a recent approach [23] relevant process fragments are recommended based on a repository of semantically annotated processes. Such repository-based approaches have the advantage that best practices in modeling can be promoted. Indeed, a syntactically correct process resulting from our assistance might still violate particular modeling guidelines or comprise semantical problems such as

deadlocks. In [12] a pattern-based approach for BPM editing has been proposed and implemented for the IBM WebSphere Business Modeler. Also the auto-link feature has been described already in [12]. Finally, it has to be admitted that special-purpose approaches usually are more efficient. Language-specific optimizations often can be incorporated into the parser. For instance, in [24] an efficient parser for workflow graphs has been proposed that runs in linear time.

8 Conclusion

In this paper a BPM editor with user assistance has been presented as a showcase for the previously introduced *generic* approach to syntax-based user assistance in diagram editors. With virtually no programming effort powerful assistance features have been realized most special-purpose modeling tools not even provide. Those features help both in learning and dealing with BPMs. Moreover, thanks to the underlying formal approach (hypergraph grammars and hypergraph transformation) the provided assistance satisfies several desirable properties, e.g., the preservation of correctness. Since the editor is generated from an abstract specification it can be extended or adapted easily. A screencast of this editor in action is provided at <http://www.unibw.de/inf2/DiaGen/assistance/bpm>. There, the editor is also available for download as an executable Java archive.

In future, an incremental version of the used parser has to be developed. This would have two benefits: First, the performance can be further improved. And second, it would be possible to provide assistance in a more pervasive way, i.e., suggestions could be given while editing and not only on user's request. Finally, a comparative user study has to be performed.

Acknowledgment We thank Jana Koehler for fruitful discussions and the anonymous referees for their insightful remarks.

References

1. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley-IEEE Computer Society (2008)
2. Gronback, R.C.: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley Longman, Amsterdam (2009)
3. de Lara, J., Vangheluwe, H., Alfonseca, M.: Meta-modelling and graph grammars for multi-paradigm modelling in AToM³. *Softw. and Syst. Modeling* (2004) 193–209
4. Zhu, N., Grundy, J., Hosking, J., Liu, N., Cao, S., Mehra, A.: Pounamu: A meta-tool for exploratory domain-specific visual language tool development. *Systems and Software* **80**(8) (2007) 1390–1407
5. Minas, M.: Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming* **44**(2) (2002) 157–180
6. Mazanek, S., Maier, S., Minas, M.: An algorithm for hypergraph completion according to hyperedge replacement grammars. In *Proc. of the 4th Int. Conf. on Graph Transformations*. Volume 5214 of LNCS. Springer (2008) 39–53

7. Mazanek, S., Maier, S., Minas, M.: Auto-completion for diagram editors based on graph grammars. In Proc. of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing, IEEE (2008) 242–245
8. Mazanek, S., Minas, M.: Generating correctness-preserving editing operations for diagram editors. In Proc. of the 8th Int. Workshop on Graph Transformation and Visual Modeling Techniques. Volume 18 of Electronic Communications of the EASST. European Association of Software Science and Technology (2009)
9. Mazanek, S., Minas, M.: Contraction of unconnected diagrams using least cost parsing. In Proc. of the 8th Int. Workshop on Graph Transformation and Visual Modeling Techniques. Volume 18 of Electronic Communications of the EASST. European Association of Software Science and Technology (2009)
10. Object Management Group: Business Process Modeling Notation (BPMN) (2009) <http://www.omg.org/docs/formal/09-01-03.pdf>.
11. Gruhn, V., Laue, R.: What business process modelers can learn from programmers. *Science of Computer Programming* **65**(1) (2007) 4–13
12. Gschwind, T., Koehler, J., Wong, J.: Applying patterns during business process modeling. In Proc. of the 6th Int. Conf. on Business Process Management. Volume 5240 of LNCS. Springer (2008) 4–19
13. Drewes, F., Habel, A., Kreowski, H.J.: Hyperedge replacement graph grammars. In Rozenberg, G., ed.: *Handbook of Graph Grammars and Computing by Graph Transformation*. Vol. I: Foundations. World Scientific (1997) 95–162
14. Reps, T.W., Teitelbaum, T.: *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer (1989)
15. Horgan, J.R., Moore, D.J.: Techniques for improving language-based editors. *SIGSOFT Softw. Eng. Notes* **9**(3) (1984) 7–14
16. Nilsson-Nyman, E., Ekman, T., Hedin, G.: Practical scope recovery using bridge parsing. In Proc. of the First Int. Conf. on Software Language Engineering. Volume 5452 of LNCS. Springer (2009) 95–113
17. Tolvanen, J.P.: How to support language users? (2008) <http://www.metacase.com/blogs/jpt/blogView?entry=3405240161> [accessed 09-July-2009].
18. Sen, S., Baudry, B., Vangheluwe, H.: Domain-specific model editors with model completion. In: *Models in Software Engineering*. Volume 5002 of LNCS. Springer (2008) 259–270
19. Hesselund, A., Czarnecki, K., Wasowski, A.: Guided development with multiple domain-specific languages. In Engels, G., Opdyke, B., Schmidt, D.C., Weil, F., eds.: *MoDELS*. Volume 4735 of LNCS. Springer (2007) 46–60
20. White, J., Schmidt, D.C., Nechypurenko, A., Wuchner, E.: Model intelligence: an approach to modeling guidance. *UPGRADE* **9**(2) (2008) 22–28
21. Blanc, X., Mounier, I., Mougnot, A., Mens, T.: Detecting model inconsistency through operation-based model construction. In Proc. of the 30th Int. Conference on Software Engineering, New York, NY, USA, ACM (2008) 511–520
22. Ehrig, K., Küster, J.M., Taentzer, G.: Generating instance models from meta models. *Software and Systems Modeling* (2008)
23. Hornung, T., Koschmider, A., Lausen, G.: Recommendation based process modeling support: Method and user experience. In Proc. of the 27th Int. Conf. on Conceptual Modeling. Volume 5231 of LNCS. Springer (2008) 265–278
24. Vanhatalo, J., Völzer, H., Koehler, J.: The refined process structure tree. In Proc. of the 6th Int. Conf. on Business Process Management. Volume 5240 of LNCS. Springer (2008) 100–115