Electronic Communications of the EASST
Volume 31 (2010)

Proceedings of the
Second International Workshop on
Visual Formalisms for Patterns
(VFfP 2010)

Combination of Different Layout Approaches

Sonja Maier and Mark Minas

12 pages

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

Combination of Different Layout Approaches

Sonja Maier and Mark Minas

Universität der Bundeswehr München

Abstract: In an interactive environment such as a visual language editor, it is not sufficient to apply the same layout algorithm in every situation. Instead, the user often wants to select the layout behavior at runtime. With the approach presented, the user can control the layout behavior by choosing different layout patterns for different parts of a diagram, e.g., a graph drawing algorithm may be applied to some selected components while others are aligned vertically or horizontally.

To enable the specification of layout behavior, we introduced the concept of layout patterns in previous work. Each layout pattern encapsulates certain layout behavior, and hence enables modularization and reuse. To specify user-controlled layout behavior, a flexible combination of arbitrary layout patterns needs to be enabled. Therefore, we introduce an approach that is capable of combining diverse layout approaches, such as standard graph drawing algorithms, constraint-based algorithms, or rule-based layout algorithms. More specifically, an algorithm is presented that automatically computes the complete layout in a straightforward way.

Keywords: Layout Pattern, Pattern Combination, User-Controlled Layout

1 Introduction

To enable user-controlled layout behavior, the modularization of layout behavior is required. Our approach is based on the concept of layout patterns (LPs), which were introduced in [MM09a]: Each pattern encapsulates certain layout behavior; that way the behavior can easily be reused. This is a beneficial feature as different visual languages show similar layout characteristics, e.g., class diagrams and statechart diagrams both have a graph-like structure and enable the nesting of nodes. With this concept, different layout algorithms may be combined, e.g., standard graph drawing algorithms and constraint-based algorithms. In addition, rule-based layout algorithms [MM09b] may be used that are specifically tailored to the interactive nature of diagram editors.

To support the user in an interactive environment, it is not sufficient to apply the same layout algorithm in every situation. Instead, the user wants to alter the layout behavior at runtime. With the approach presented, the user can control layout behavior: He may choose parts of the diagram and the layout pattern to be used for these parts. Layout patterns may either be applied once (*single application*), or until the user explicitly removes the application request (*permanent application*). For instance, the editor user may permanently align diagram components vertically.

Up to now, patterns were combined manually, meaning that a control program had to be provided for every visual language editor. In this paper, we focus on this aspect and propose a generic propagation algorithm that enables the combination of these diverse layout approaches without being forced to manually write a control algorithm. With this propagation algorithm, the layout is computed in a straightforward way most of the time.

The rest of the paper is structured as follows: Related work is discussed in Section 2, and Section 3 introduces the graph diagram editor, the running example used in this paper. Layout algorithms in general as well as the layout of the graph diagram editor are presented in Section 4. The concept of layout patterns as well as the user-controlled instantiation of them is described in Section 5. Section 6 gives details of the combination of layout patterns. Section 7 discusses the results, and Section 8 concludes the paper.

2 Related Work

The most prominent, but also quite old tool, that deals with dynamic diagram drawing is Sketchpad [Sut63]. As many "modern" tools, Sketchpad is based on constraints. Interestingly, besides solving the constraints via a relaxation method, it also provides an additional mechanism called "one pass method" to layout the diagram in only one run. Most of the current tools use one-way constraints or multi-way constraints [SMFB93]. In many approaches, local propagation algorithms, such as the DeltaBlue algorithm, are used for multi-way constraints. The algorithm presented in this paper is a local propagation algorithm that enables the combination of constraint-based algorithms and other layout approaches, such as graph drawing algorithms.

Many tools, such as GLIDE [RMS97] or Dunnart [DMW09], deal with interactive graph drawing. Both tools are based on declarative constraints and provide some user-controlled layout behavior. For instance, they provide a mechanism for aligning nodes. Internally, a constraint solver needs to be invoked, and hence performance is an issue. In our approach, for most visual languages, no "classic" constraint solver is needed.

Rules are the underlying concept of our rule-based layout algorithm, as also done in [BGL06]. There, interaction dynamics are defined via rules, and the definition of interaction is based on one language-independent meta model for all diagram languages. Instead, we introduce several language-independent meta models, one for each layout pattern, to enable reuse. In [BG04], the same authors propose "a suite of metamodels as a basis for the classification of visual languages".

The term *pattern*, in the context of layout, has been coined by Schmidt [SK03]. He uses tree grammars for specifying visual languages and layout while we propose meta models instead.

3 Graph Diagram Editor

In the following, we briefly introduce our running example, the graph diagram editor. The simplified version considered in this paper consists of *rectangles* and *arrows*. Figure 1 shows a graph diagram editor, which was created with the editor generation framework *DiaMeta* [Min06]. *DiaMeta* allows for generating visual language editors from a specification. The core of the specification consists of two meta models: The abstract syntax meta model (ASMM), representing the language's abstract syntax, and the concrete syntax meta model (CSMM), representing the language's concrete syntax. The CSMM resembles all visual components and their spatial relationships, whereas the ASMM is an instance of the diagram language's meta model that defines the language's abstract syntax. In the CSMM, each component type is represented by a class connected with its abstract syntax counterpart. Each meaningful relationship between component types is represented by an association.

For graphs, the ASMM contains the components `Node` and `Edge` and the relationships `from` and `to`, the CSMM the components `Rectangle` and `Arrow` and the relationship `attach`. Both meta models, CSMM and ASMM, form a combined meta model, called language-specific meta model (LMM) in the following. When a diagram is drawn, the editor instantiates the LMM, obtaining the language-specific model (LM). The CSM is an instance of the concrete syntax meta model (CSMM) and shows the diagram components together with their spatial relationships. The ASM is an instance of the abstract syntax meta model (ASMM).

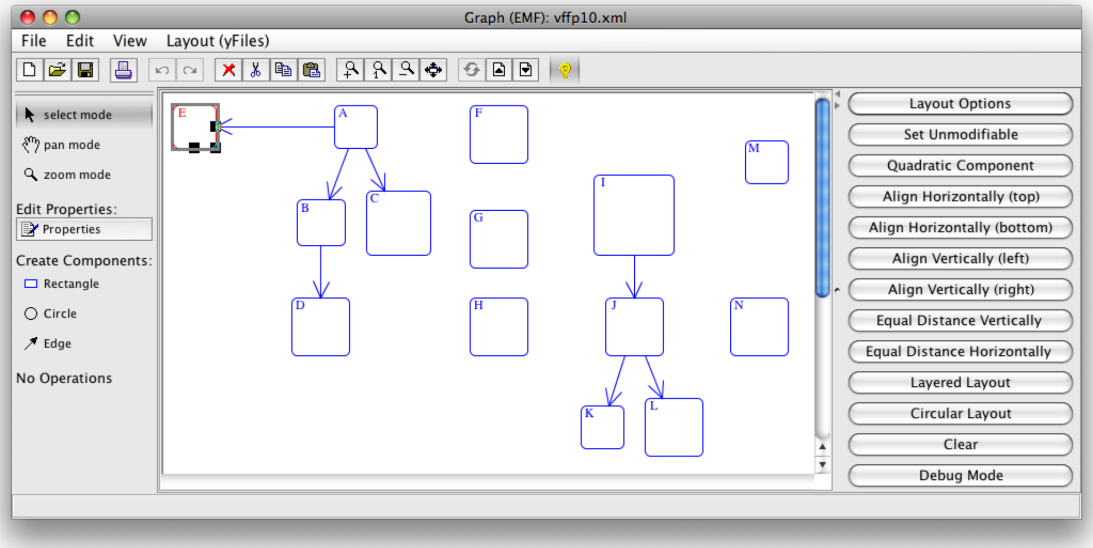


Figure 1: Graph Diagram Editor

4 Layout Algorithms

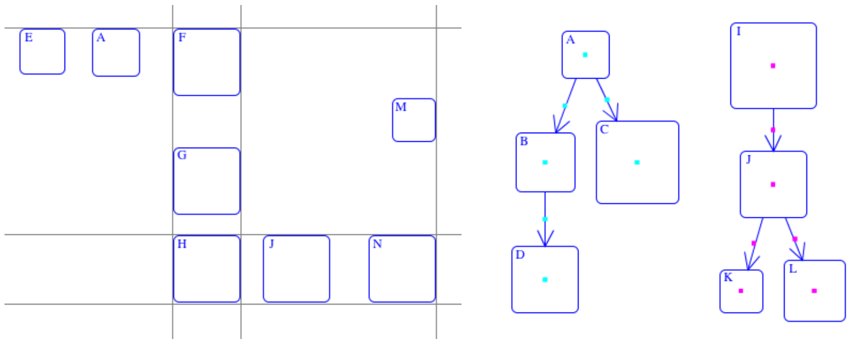
In the context of diagram editors, besides writing the layout algorithm by hand, usually one of the following layout concepts is chosen: Standard graph drawing algorithms, constraint-based algorithms or rule-based layout algorithms. These three types of layout algorithms may be combined with the approach for specifying layout behavior presented in Section 5. Additionally, the editor user has the possibility to select and influence these layout algorithms at runtime.

Graph drawing algorithms tend to be quite complex. Hence, it is reasonable to implement them, not to define them on an abstract level. Therefore, we offer the possibility to reuse existing implementations, i.e., graph drawing libraries. Examples for standard graph drawing algorithms are force-directed layout or layered layout. A constraint-based algorithm is defined by providing a set of declarative constraints. In our approach, the constraints are attached to a meta model. A standard constraint solver then computes a solution to this constraint satisfaction problem. Rule-based layout algorithms are a variation of constraint-based algorithms. They were successfully applied in our context, as described in [MM09b].

Layout of the Running Example For the graph diagram editor, several layout algorithms have been combined. The associated layout engine uses rule-based layout algorithms and graph drawing algorithms. No constraint-based algorithms are needed.

- *Size*: This rule-based layout algorithm makes sure that circles and rectangles never get smaller than their specified minimal *size*.
- *Align*: This rule-based layout algorithm *aligns* selected components vertically or horizontally. They may be aligned at each side of the component: top, bottom, left and right.
- *Edge follower*: This graph drawing algorithm implements an *edge follower* that makes sure that arrows stay attached to components. This feature is needed, as DiaMeta supports freehand editing.
- *Layered layout*: This graph drawing algorithm creates a *layered layout* for selected arrows and rectangles.

The algorithms *size* and *edge follower* are automatically applied, whereas the algorithms *align* and *layered layout* are controlled by the user. An example application of these two algorithms is shown in Figure 2. The difference will be discussed in more detail in Section 5.1.



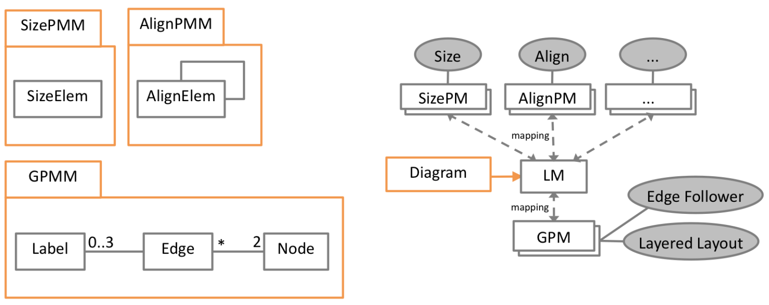
(a) Horizontal & Vertical Alignment (b) Layered Layout

Figure 2: User-Controlled Layout

5 Layout Patterns

This section revisits layout patterns as a means to encapsulate certain layout behavior. Our pattern based approach has been described in [MM09a]. In our approach, each layout pattern is based on a language-independent, but *pattern-specific meta model* (PMM). This way, reuse of layout behavior is enabled, and the complexity of the layout specification is usually decreased, due to the customized meta model. Graph drawing algorithms, constraint-based algorithms, and rule-based layout algorithms are defined on top of these pattern-specific meta models. In order to apply a layout pattern to a certain visual language, i.e., in order to instantiate the pattern, a correspondence between the PMM and the LMM needs to be defined. This correspondence between the LMM and the PMM specifies the *mapping* between the instances LM and PM.

In our example, the graph drawing algorithms *edge follower* and *layered layout* both operate on the same PMM, called graph pattern meta model (GPMM). The specification of rule-based layout algorithms is based on the corresponding PMM, e.g., SizePM or AlignPM. All PMMs are visualized in Figure 3(a). As can be seen in Figure 3(b), in our example, three mappings are defined. E.g., in case of the pattern *layered layout*, a correspondence between the class `Rectangle` and the class `Node` has to be established. Furthermore, a correspondence between the class `Arrow` and the class `Edge` must be defined.



(a) Pattern Meta Models (PMMs) (b) Correlation: Diagram, LM, PMs and Patterns

Figure 3: Internal Diagram Representation

5.1 Automatic & User-Controlled Application of Layout Patterns

The instantiation of layout patterns is either performed automatically or it is influenced by the user. These two mechanisms of instantiation may be combined. In case of automatic instantiation, the user may only alter the behavior by completely turning a pattern on or off. In case of user-controlled instantiation, by selecting one or more components, the user may define a part of a diagram to which a certain layout pattern is applied. In addition, the user may choose whether the layout pattern is applied once (*single application*) or permanently until he removes the application request (*permanent application*) explicitly. Instantiation is done as follows: Internally, the model is represented by a graph. For each pattern, the required instances are determined via graph matching. In case of automatic instantiation, graph matching is done on the complete graph, whereas in case of user-controlled instantiation, graph matching is performed on an excerpt of the graph, which corresponds to the user-selected components.

In the following, one pattern is presented that is automatically instantiated. Here, the rule-based layout algorithm *edge follower* is chosen. Afterwards, two patterns are described that are instantiated by the user. Here, the graph drawing algorithm *layered layout* and the rule-based layout algorithm *align* are depicted.

Edge Follower for Arrows An example is the pattern *edge follower*. For the example shown in Figure 2, the following instantiation is performed: Sixteen `Node` instances (one for each rectangle) and six `Edge` instances (one for each arrow) are created.

Layered Layout of Rectangles The instantiation of the layered layout is controlled by the user. To do so, the user selects a set of components, and afterwards chooses the layout he wants to apply to this part of the diagram. From this point on, the layout pattern is applied to this part of the diagram only once (single instantiation) or each time the layout engine is called (permanent instantiation) until the user removes this instantiation request again. In the example shown in Figure 2(b), the user added two instantiation requests. For the first instantiation, four `Node` instances `A`, `B`, `C` and `D`, and three `Edge` instances are created.

Horizontal and Vertical Align(ment) of Rectangles The instantiation of the *align* pattern is also performed manually by the user. Again, the user selects a set of components, and applies the layout wanted. As a consequence, the layout pattern is applied to this part of the diagram. In the example shown in Figure 2(a), the user added four instantiation requests. Two of them are the following: Three `AlignElem` instances `E`, `A` and `F` and three instances `F`, `G` and `H`. The first instantiation holds the option 'horizontal', whereas the second the option 'vertical'.

Debug Mode As can be seen in Figure 2, there is a *debug mode* available, which allows for the recognition of applied layout patterns. In case of the *layered layout* pattern, the existence of a user-controlled instantiation request is visualized by a dot in the middle of each component involved. This visualization is automatically generated. Besides, the visualization can also be specified. E.g., in case of the *align* pattern, the existence of a user-controlled instantiation request is visualized by a vertical or horizontal gray line.

Syntax Preservation While the user interacts with the diagram in *layout mode*, the syntax is preserved at any time. This functionality is achieved by an examination of the diagram, or more specifically, by an examination of the internal graph representation of the diagram. In case the syntax is changed by the layout algorithm, the user changes that led to this violation are undone.

6 Pattern Combination

As we saw, while computing the layout, a certain set of patterns comes into play. Some of these patterns are automatically added, while others are explicitly added by the user. Each component as well as each pattern has one or more dedicated constraints. All constraints present in a diagram form a constraint network. The term "constraint" used here should not be mistaken for the term "constraint" used in the context of constraint-based layout. Here, a constraint consists of a predicate and a set of rules. The layout of a diagram is "valid", if all constraints are satisfied. The purpose of the algorithm introduced in this section is to find a variable assignment for which all constraints are satisfied. There exist cases where the algorithm is unable to find a solution. If the algorithm fails to find a solution, an undo of the user changes is performed.

Constraints We distinguish three different types of constraints:

- *Layout Constraints*: These constraints "belong" to certain patterns and operate on pattern-specific variables. These variables are part of the pattern-specific meta models. They assure the functionality of the pattern. We distinguish three types of layout constraints, which encapsulate rule-based layout algorithms, constraint-based layout algorithms and graph drawing algorithms.
  - In case a pattern encapsulates a rule-based layout algorithm, predicates as well as rules are defined.
  - If a pattern encapsulates a constraint-based layout algorithm, the predicate is defined via a set of constraints, and the constraint solver is one associated rule.
  - In case a pattern encapsulates a graph drawing algorithm, it is not necessary to define a predicate. Instead, it is "checked" if executing the algorithm changes one or more variable values. If this is the case, the algorithm needs to be applied.
- *Component Constraints*: These constraints "belong" to certain components. They keep attributes consistent inside a single component and operate on language-specific variables. These variables belong to the langugage-specific meta model.
- *Mapping Constraints*: These constraints relate pattern-specific and language-specific variables. Hence, they complete the relationship between the language-specific meta model and the pattern-specific meta models.

Definition 1 A *constraint net* consists of a set V of variables and a set C of constraints. An *assignment* $\alpha : V \to \mathbb{R}$ assigns a value $\alpha(v)$ to each variable $v \in V$. A *status* function $\sigma : V_c \to \{u, f, c\}$ assigns a status $\sigma(v)$ to each variable $v \in V$. A constraint $c \in C$ consists of a constraint predicate P_c and a set $R_c = \{r_1, ..., r_j\}$ of (repair) rules. An assignment α *satisfies* a constraint $c \in C$ iff α assigns values to the variables such that P_c is satisfied, denoted by $P_c(\alpha)$. Each rule $r \in R_c$ comes with an applicability predicate A_r, and r is applicable iff $A_r(\alpha, \sigma)$ holds. If applicable, r computes a new assignment α' and a new status function σ' such that $P_c(\alpha')$ holds.

If a constraint c is not satisfied, a rule may be applied, which changes the value and the status of some variables, and this way "repairs" the constraint. Therefore, a rule operates on the variables V_c, reads $\alpha(v_1), ..., \alpha(v_i)$ and replaces α by α'. Besides, it replaces σ by σ'. A variable $v \in V_c$ may either be unchanged and unfixed ($\sigma(v) = u$), changed ($\sigma(v) = c$) or fixed ($\sigma(v) = f$). A repair rule can either change the value of a variable or it may fix the value of a variable. A rule $r \in R_c$ is applicable, iff $\forall v \in V_c : \alpha(v) \neq \alpha'(v) \Rightarrow \sigma(v) = u \wedge \sigma'(v) = c$ and $\sigma(v) \neq \sigma'(v) \Rightarrow \sigma(v) = u$. This means that every variable may be changed or fixed only once.

6.1 Algorithm

Based on these definitions, the algorithm that computes a variable assignment for which all constraints are satisfied, is defined as follows:

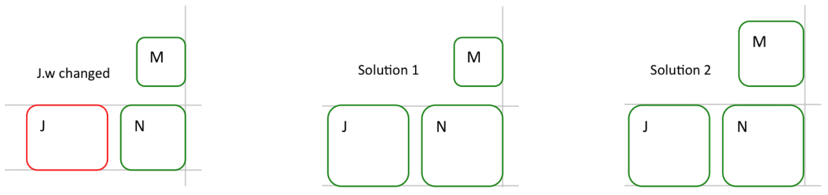
```
function search $(\alpha, \sigma)$ begin
    if $P_c(\alpha)$ holds for each constraint $c \in C$ then
        the solution is $\alpha$;
        return success;
    fi;
    compute ordered list $L_1$ of constraints $c \in C$ such that $\neg P_c(\alpha)$ for each $c \in L_1$;  (*)
    if each $c \in L_1$ is repairable then
      for each $c \in L_1$ do
          compute ordered list $L_2$ of rules in $R_c$ such that $A_r(\alpha, \sigma)$ for each $r \in L_2$;  (**)
          for each $r \in L_2$ do
              apply $r$, obtaining $\alpha', \sigma'$;
              if search$(\alpha', \sigma')$ is successful then
                  return success;
              fi;
          od;
      od;
    fi;
    return failure;
end
```

The idea is that changes made by the user are propagated in the diagram. Starting with the attribute(s) changed, all constraints are checked that involve these attributes. For each violated constraint, one of the corresponding rules is applied, which changes one or more variable(s). Again, all constraints are checked that involve these variables. This procedure is continued until all constraints that need to be checked are satisfied. The first result found is chosen and the algorithm stops afterwards. In one step, it might be the case that more than one constraint needs to be repaired and hence more than one rule needs to be exectued. In line (*), the algorithm creates an ordered list L_1 of all constraints that are not satisfied. The order is determined by a certain heuristics. Furthermore, if a constraint is violated, there might be several rules available to "repair" this violation. Line (**) determines an ordered list of repair rules based on some prioritization scheme. The rules are tried out by backtracking until the algorithm terminates either by finding a new assignment satisfying the predicates of all constraints, or by signaling a failure.

6.2 Example

We discuss the approach presented in Section 6 via the example shown in Figure 4(a). The user has resized component J, and now the layout engine has to update the diagram accordingly.

Variables The layout algorithm is based on a certain set of variables: The shape of each component A is defined via its language-specific variables A.x, A.y, A.w and A.h, which denote its top-left corner (x-position, y-position), width and height. The pattern *align* refers to the pattern-specific variables A.t, A.b, A.l and A.r, which stand for the top, the bottom, the left, and the right border of the component A.



(a) User Modification (b) J and N are resized, M is moved to the right (c) J, N and M are resized

Figure 4: Running Example: Two Layout Alternatives

Constraints Figure 5 visualizes the constraints and the variables involved that are present in our example. It shows variables as ovals, constraints as rectangles, and dependencies as lines. The following constraints are present in the example: Three *layout constraints* were added by the user: "Align components J and N at the top" (layout constraint JN:align_t), "align components J and N at the bottom" (layout constraint JN:align_b) and "align components M and N at the right" (layout constraint MN:align_r). Four *mapping constraints* for each component A were automatically added to relate pattern-specific and language-specific variables: (mapping constraint A:map_align_t), (mapping constraint A:map_align_b), (mapping constraint A:map_align_l) and (mapping constraint map_align_r). One *component constraint* was added automatically: "Nodes are always quadratic" (component constraint A:comp).

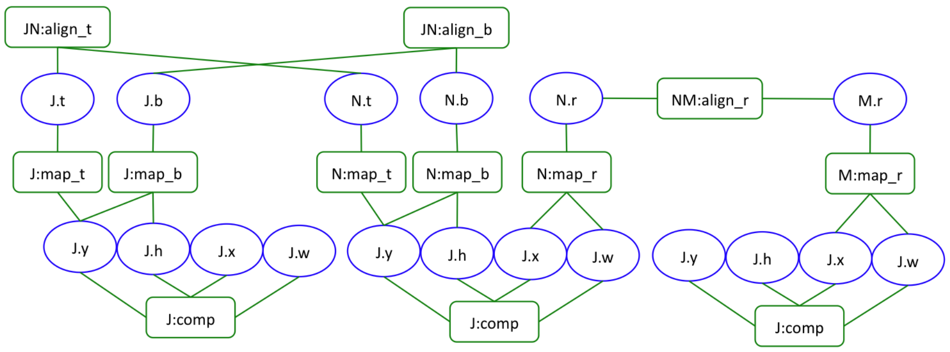


Figure 5: Constraint Network of the Example

Rules In our example, each layout constraint as well as each component constraint only has one rule assigned. Each mapping constraint has two rules assigned. This is a typical structure, although the system does not require this restriction. In case only one rule is assigned, a constraint may be "repaired" by applying this rule. E.g., if A.w is changed, then the rule that corresponds to A:comp updates A.h. In case more than one rule is available, a choice can be made. To describe this, we depict the constraint A:map_align_b. If A.y or A.h was changed previously, and A:map_align_b is violated, A.b is updated. If A.b was changed previously, and A:map_align_b is violated, we have two choices. Rule 1 moves component A (attribute A.y is changed), whereas rule 2 resizes component A (attribute A.h is changed) such that A.b is correct afterwards.

Propagation Algorithm In order to compute a "valid" layout, a set of rules is applied. During execution of the backtracking algorithm, different combinations are tried, until a valid layout is found. For the example, the algorithm may determine two valid solutions: The first valid solution is that J and N are resized, and M is moved to the right, as can be seen in Figure 4(b). The second valid layout is that J, N and M are resized, as can be seen in Figure 4(c). The example run of the algorithm is shown in Figure 6. Each arrow denotes a call of the recursive function *search*. Only the two branches are visualized that lead to the two valid solutions. In the end, the algorithm chooses the first solution as the result.

Starting point is the variable A.w, which was changed by the user. As a consequence, the constraints J:map_r and J:comp are violated and the corresponding rules are applied. The rule associated with J:map_r updates the variable J.r, the rule associated with J:comp changes the variable J.h. Afterwards, the constraint J:map_b is violated, and the corresponding rule is applied. This proceeding is continued until the first variable assignment is found, which satisfies all constraints. In our case, this is "Solution 1", as can be seen in Figure 6.

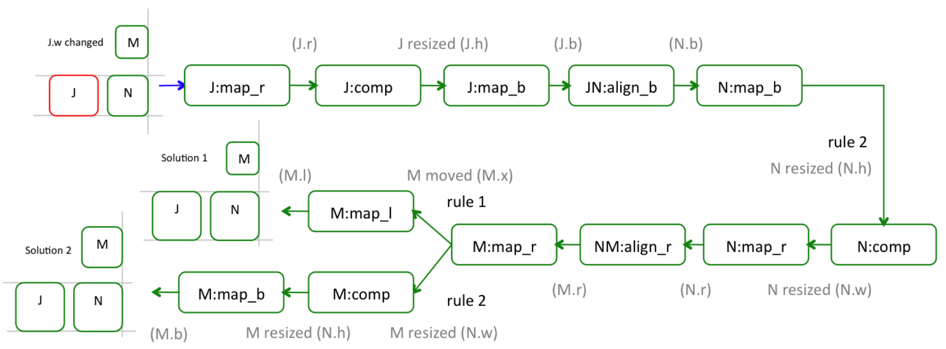
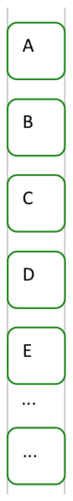


Figure 6: Example Run of the Algorithm

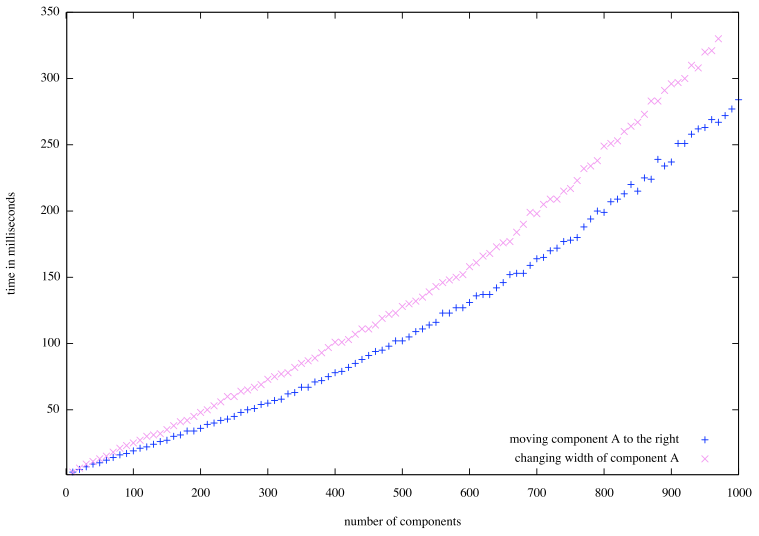
6.3 Evaluation

Figure 7(a) shows an example diagram, where a certain number of components are aligned vertically. Besides, each component is quadratic and its minimal size is preserved. The user may interact as follows: He may either move an arbitrary component or resize it. As a consequence, all other components need to be updated accordingly. Performance was measured on a machine equipped with an Intel 2 Core Duo 3.06 GHz processor, 4 GB RAM, running Mac OS X Version 10.5.8 and Java JDK 1.6.0_20. The chart visualized in Figure 7(b) shows the time it takes to update the diagram after the user changes a component. As can be seen, updating 500 components takes about 0.1 seconds and updating 1000 components still takes less than 0.4 seconds.

One could guess that the creation of all the artifacts, such as the meta model instances, could lead to a bad performance. This is not the case because most of the computations are very cheap and do not influence the overall performance. The only “expensive” part is the algorithm for pattern combination. In some scenarios, the use of backtracking can lead to an explosion of cases and would result in a bad performance. The number of cases increases if many constraints involve the same variables. This is usually not the case in “real world scenarios”. Instead, in most cases, a result is found without doing backtracking at all. Hence, our expectation is that performance is satisfactory on average. For instance, the layout computation for the diagram shown in Figure 1 takes less than 12 milliseconds for every user input imaginable.



(a) Example



(b) Runtime

Figure 7: Performance of the Algorithm

7 Discussion

In case a user changes more than one attribute, e.g., if he moves several nodes at once, or if he adds a new pattern request, there exist several solutions. There also exist several solutions if an initial layout is computed. Right now, in all these cases, an arbitrary solution is chosen. Here, also all solutions can be investigated and then the ”best” solution can be determined.

There exists a huge variety of patterns that can be defined, e.g., a pattern ensuring the same height or width of a set of components, or a pattern ensuring an equal distance between a set of components. With the approach presented, it can be checked if a certain pattern is applicable in a certain situation. This information can be provided to the user in a certain way and can guide the user in the process of beautifying the diagram. The system can even make layout suggestions based on applicability and a certain metric. Right now, the editor simply forbids user manipulations, if one or more patterns are not applicable.

8 Conclusions

In this paper, we have outlined a general approach for the specification of layout behavior for diagram editors. It is possible to define a great variety of functionalities that support the editor user. The functionality is defined via layout patterns that encapsulate certain layout behavior and make it easier to reuse the behavior. Instantiation of these patterns is either performed automatically or user-controlled. Different layout algorithms may be combined with our approach: standard graph drawing algorithms, constraint-based algorithms and rule-based layout algorithms. Therefore, we presented a propagation algorithm which computes the layout in a straightforward way.

Bibliography

[BG04] P. Bottoni, A. Grau. A Suite of Metamodels as a Basis for a Classification of Visual Languages. In *Proc. of the Symposium on Visual Languages - Human Centric Computing (VLHCC '04)*. IEEE, 2004.

[BGL06] P. Bottoni, E. Guerra, J. de Lara. Metamodel-based definition of interaction with visual environments. In *Proc. of the 2nd Intl. Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI '06)*. CEUR Workshop Proc. 214. 2006.

[DMW09] T. Dwyer, K. Marriott, M. Wybrow. Dunnart: A Constraint-Based Network Diagram Authoring Tool. In *Graph Drawing: 16th Intl. Symposium (GD 2008)*. LNCS 5417, pp. 420–431. Springer-Verlag, 2009.

[Min06] M. Minas. Generating Meta-Model-Based Freehand Editors. In *Proc. of the 3rd Intl. Workshop on Graph Based Tools (GraBaTs'06)*. ECEASST 1. 2006.

[MM09a] S. Maier, M. Minas. Pattern-Based Layout Specifications for Visual Language Editors. *Proc. of the Workshop on Visual Languages and Computing (VLC 2009)*, 2009.

[MM09b] S. Maier, M. Minas. Rule-based Diagram Layout using Meta Models. In *Proc. of the 1st Intl. Workshop on Visual Formalisms for Patterns (VFfP'09)*. ECEASST 25. 2009.

[RMS97] K. Ryall, J. Marks, S. Shieber. An interactive constraint-based system for drawing graphs. In *Proc. of the 10th ACM Symposium on User Interface Software and Technology (UIST '97)*. Pp. 97–104. ACM, 1997.

[SK03] C. Schmidt, U. Kastens. Implementation of Visual Languages using Pattern-based Specifications. *Software: Practice and Experience* 33(15):1471–1505, 2003.

[SMFB93] M. Sannella, J. Maloney, B. Freeman-Benson, A. Borning. Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm. *Software – Practice and Experience* 23:529–566, 1993.

[Sut63] I. E. Sutherland. Sketchpad: A Man-machine Graphical Communication System. 1963.